ВВЕДЕНИЕ

У многих есть любимые семейные рецепты — это может быть вариация классического блюда или особые семейные блюда, рецепт которых передается из поколения в поколение. Создание кулинарной книги, включающей в себя такие рецепты, — это веселый и творческий процесс, для реализации которого вам даже не придется выходить из дома.

Кулинарная, или поваренная, книга — справочное издание, содержащее в основном различные рецепты приготовления пищи и напитков.

В мире цифровых технологий стремятся всю информацию перевести в цифровой формат и соответственно отказываются от печатных книг. Определение, описанное выше, можно перефразировать к реалиям мира цифровых технологий:

Кулинарная, или поваренная, книга — приложение, позволяющее каким-либо образом манипулировать с сущностью “рецепт”, в минимальном функционале - просматривать информацию о различных рецептах приготовления пищи и напитков.

Большинство приложений “Книга Рецептов” созданы исключительно для android. Программы такого типа либо упрощают работу всем кто захотел приготовить что-либо путем сбора и хранением определенной однотипной информации; либо же предоставляет возможность записывать или добавлять свой рецепт для дальнейшей работы с ним.

Целью курсовой работы: Разработать программное обеспечение для ведения и учета доступных продуктовых ресурсов, а также обеспечить его интеграцию с базой данных кулинарных рецептов.

В соответствии с поставленной целью в работе определены следующие задачи:

* Разработать техническое задание;
* Изучить предметную область;
* Провести анализ возможностей разработки в секторе инструментальных прикладных приложений (в том числе анализ уже существующих разработок по теме);
* Описать основные этапы проектирования и разработки приложения;
* Учитывая полученные на этапе анализа результаты, разработать приложение для ПК в соответствии с техническим заданием.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Техническое задание

1.1.1 Введение

1.1.1.1 Наименование программного продукта или программы: «RecipeBook».

1.1.1.2 Краткая характеристика области применения: ведение и учет доступных продуктовых ресурсов.

1.1.2 Основания для разработки

1.1.2.1 Основания для проведения разработки: задание курсового проекта от преподавателя ГБПОУ НСО «Новосибирский химико-технологический колледж им. д. и. Менделеева».

1.1.2.1 Наименование и условное обозначение разработки: книга рецептов.

1.1.3 Назначение разработки

1.1.3.1 Функциональное назначение: ведение и учет доступных продуктовых ресурсов.

1.1.3.2 Эксплуатационное назначение: для использования приложения пользователями.

1.1.4 Требования к продукту

1.1.4.1 Программа должна обеспечивать возможность выполнения перечисленных ниже функций:

* Добавление/удаление ингредиентов.
* Добавление/редактирование/удаление продуктов.
* Добавление/редактирование/удаление рецептов.
* Сортировка рецептов по определенным параметрам.
* Фильтрация рецептов по определенным параметрам.
* Сохранение/загрузка состояния БД из файлов.
* Синхронизация данных на разных устройствах.
* Интеграция Telegram.

1.1.4.2 Организация входных данных: название приложения, иконки, картинки, формы QT, файл ресурсов QT, логотип, макет приложения, краткое описание функций приложения.

1.1.4.3 Организация выходных данных: исполняемый файл EXE инсталлятор, пакет файлов извлеченных из инсталлятора при установке.

1.1.4.4 Временные характеристики: время работы пользователя в программе.

1.1.4.5 Требования к функциональным характеристикам.

1.1.4.5.1 Поддерживаемая операционная система: Windows 10, некоторые дистрибутивы Linux.

1.1.4.5.2 Минимальные системные требования

* Процессор 2 ядра, с тактовой частотой: 1600 Mhz.
* Объем жёсткого диска: не менее 1 Gb свободной памяти.
* Видеоадаптер с объемом памяти: 256 Mb.
* Объём оперативной памяти: 2 Gb.

1.1.5 Требования к программной документации

1.1.5.1 Состав программной документации

* Инструкция по инсталляции.
* Руководство по эксплуатации пользователем программного продукта.
* Файл журнала.

1.1.6 Требования к методологии разработки

1.1.6.1 Методология разработки: Scrum

1.1.6.1.1 Требования к продолжительности спринтов: не более 21 дня.

1.1.7 Календарный план работ (сроки разработки, смотреть Приложение А. Таблица 1).

1.1.8 Приложения

* Инструкцию по инсталляции.
* Инструкция по эксплуатации.
* GitHub репозиторий с версионированием (релизами).
* Журнал логирования.

1.2 Описание выбранного инструментария

1.2.1 Выбор методологии разработки

Scrum — методика организации совместного рабочего процесса, в основе которой — поэтапная разработка и совершенствование продукта небольшой командой специалистов различного профиля.

1.2.2 Выбор языка программирования

Язык программирования Python 3 — это мощный инструмент для создания программ самого разнообразного назначения, гибок, полностью удовлетворяет требованием, не требует компиляции, кроссплатформенный.

1.2.3 Выбор среды разработки

PyCharm — это среда программирования для языка Python, или IDE от компании JetBrains. Средами называют программы, в которых можно писать, запускать и отлаживать код, устанавливать новые расширения и дополнительные модули. Это мощный многофункциональный инструмент для разработчиков.

1.2.4 Выбор фреймворков

Для выбора технологий разработки GUI необходимо учесть множество факторов.

Qt — фреймворк для разработки кроссплатформенного программного обеспечения на языке программирования C++. Для многих языков программирования существуют наборы библиотеки, позволяющие использовать преимущества Qt: Python — PyQt, PySide; Ruby — QtRuby; Java — Qt Jambi; PHP — PHP-Qt и другие.

Также будет использоваться прикладная программа QtDesigner для создания Qt форм.

unittest - фреймворк модульного тестирования.

Среда unittest модульного тестирования изначально была вдохновлена ​​JUnit и имеет тот же вкус, что и основные среды модульного тестирования на других языках. Он поддерживает автоматизацию тестирования, совместное использование кода установки и завершения тестов, объединение тестов в коллекции и независимость тестов от структуры отчетности.

1.2.5 Выбор библиотек

Библиотека — это набор готовых функций, классов и объектов для решения каких-то задач.

QT-PyQt-PySide-Custom-Widgets, удивительные пользовательские виджеты, созданные для настольных приложений QT. Упростите процесс разработки пользовательского интерфейса. Эти виджеты можно использовать в QT Designer, а затем импортировать в код PySide.

PySide2 — официальный модуль Python из проекта Qt for Python , предоставляющий доступ ко всему фреймворку Qt 5.12+.

1.3 Анализ существующих решений в предметной области

Paprika называет себя «менеджером рецептов», но самый мощный ингредиент - это открытие. Paprika использует встроенный веб-браузер, чтобы охватить любой рецепт с любого веб-сайта или блога.

При нажатии «Поделиться» на странице рецептов, открыв ее в Safari, и можно импортировать ее прямо в Paprika, что упрощает организацию.

Однако при обнаружении на основе браузера возникают проблемы с анализом рецептов блога, особенно если контент отформатирован нестандартным образом. Приложение стоит 4.99 доллара и доступно как для Android, так и для iOS (смотреть Рисунок 1).

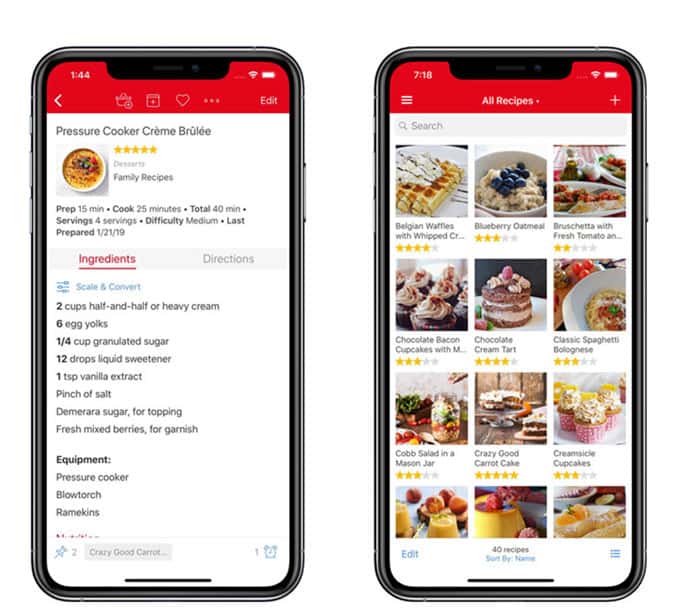


Рисунок «1 – Интерфейс Paprika»

Преимущества:

* Благодаря поиску очень удобно находить рецепты.
* Браузер в приложении может анализировать рецепты практически с любого веб-сайта.

Недостатки:

* Парсер веб-сайтов не идеален и может иметь проблемы с нестандартными форматами.
* Стоимость.

Recipe Box - создание рецептов с нуля, добавляя ингредиенты, требуемое время и инструкции. Можно добавить изображение блюда. Recipe Box может организовать рецепты по курсам (смотреть Рисунок 2).

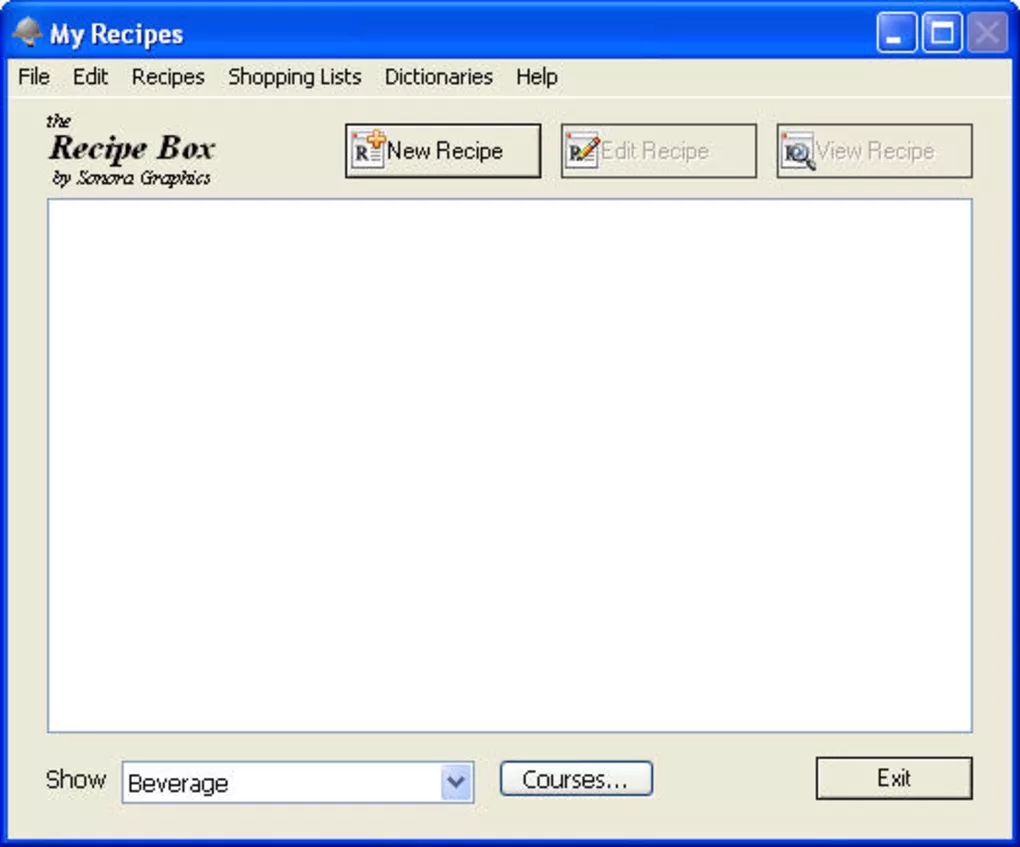


Рисунок 2 – «Интерфейс Recipe Box»

Преимущества:

* Ингредиенты, подготовка и словари измерения.
* Организовано по курсу.
* Подготовлено к печати.

Недостатки:

* Редактирование списков покупок.
* Ограничено форматом .rbook.

2 ПРОЕКТИРОВАНИЕ

Выполнение поставленных задач подразумевает разработку внутренней части приложения с предусмотрением интерфейсов для подключения графических окон управления разработанными программными модулями. Была разработана архитектура приложения и в дальнейшем это было программно реализовано.

2.1 Архитектура приложения

Архитектура программного обеспечения (англ. software architecture) — совокупность важнейших решений об организации программной системы. Архитектура включает:

* выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов.
* соединение выбранных элементов структуры и поведения во всё более крупные системы.
* архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

Документирование архитектуры программного обеспечения (ПО) упрощает процесс коммуникации между разработчиками, позволяет зафиксировать принятые проектные решения и предоставить информацию о них эксплуатационному персоналу системы, повторно использовать компоненты и шаблоны проекта в других.

2.1.1 Типы архитектур.

Многоуровневая архитектура - система делится на уровни, каждый из которых взаимодействует лишь с двумя соседними. Поэтому запросы к БД, которая обычно располагается в самом конце цепочки взаимодействия, проходят последовательно сквозь каждый «слой».

Архитектура не подразумевает какое-то обязательное количество уровней — их может быть три, четыре, пять и больше. Чаще всего используют трехзвенные системы: с уровнем представления (клиентом), уровнем логики и уровнем данных.

Преимущества:

* Каждый уровень этой архитектуры выполняет строго ограниченный набор функций (которые не повторяются от слоя к слою) и не знает о том, как устроены остальные уровни. Поэтому «содержимое» уровней можно изменять без риска глобальных конфликтов между слоями.

Недостатки:

* Низкая скорость работы. Очень много информации начинает бесполезно проходить от слоя к слою, не используя бизнес-логику. Иногда эту проблему называют sinkhole anti-pattern — шаблон проектирования, когда количество бесполезных операций начинает преобладать над полезными.
* Поиск багов в многоуровневых системах также может быть затруднен.

Хорошо подходит для:

* создания новых приложений, которые нужно развернуть по-быстрому. Это своеобразный «шаблон общего назначения».

2.1.1.2 Событийно-ориентированная архитектура

Разработчик прописывает для программы поведение (реакции) при возникновении каких-либо событий. Событием в системе считается существенное изменение её состояния.

Система, управляемая событиями, обычно содержит два компонента: источники событий (агенты) и их потребители (стоки). Типов событий обычно тоже два: инициализирующее событие и событие, на которое реагируют потребители.

Преимущества:

* Так как приложения состоят из большого количества асинхронных модулей (у которых нет информации о реализации друг друга), их легко масштабировать. Такие системы собираются как конструктор — прописывать зависимости не нужно, достаточно реализовать новый модуль.
* Дополнительно асинхронная модель позволяет добиться высокой производительности приложений.

Недостатки:

* Асинхронная натура таких приложений усложняет отладку. Одно событие может запускать сразу несколько цепочек действий. Если таких цепочек будет много, то понять, что именно вызвало сбой, может быть затруднительно. Для решения проблемы приходится прорабатывать сложные условия обработки ошибок.
* Проблема с журналированием — логи трудно структурировать.

Подходит для:

* Создания асинхронных систем. Это очевидно, поскольку сама архитектура состоит из большого количества асинхронных модулей.
* Можно применить для создания UI. Веб-страница выступает в роли контейнера, в котором каждый её компонент изолирован и реагирует на определённые действия пользователя.
* Для организации обмена сообщениями между различными информационными системами.

Микроядерная архитектура – этот тип архитектуры состоит из двух компонентов: ядра системы и плагинов. Плагины отвечают за бизнес-логику, а ядро руководит их загрузкой и выгрузкой.

Преимущества:

* Легко портировать приложение из одной среды в другую, поскольку модифицировать нужно только микроядро.
* Разделение высокоуровневых политик и низкоуровневых механизмов упрощает поддержку системы и обеспечивает её расширяемость.

Недостатки:

* Производительность приложения снижается, если подключать слишком много модулей. Однако бывает проблематично найти баланс между количеством плагинов и числом задач микроядра (обычно оно содержит лишь часто используемой код).
* Также сложно определить заранее (до начала разработки приложения) оптимальную степень дробления кода микроядра. А поменять подход позднее практически невозможно.

Хорошо подходит для:

* Создания расширяемых приложений, которыми пользуется большое количество людей. Например, ОС для iPhone имеет «микроядерные» корни — её разработчики черпали вдохновение в Mach (это один из самых первых примеров микроядра).
* Создания приложений с четким разделением базовых методов и высокоуровневых правил.
* Разработки систем с динамически меняющимся набором правил, которые приходится часто обновлять.

Похожи на архитектуру, управляемую событиями, и микроядро. Но используются тогда, когда отдельные задачи приложения можно легко разделить на небольшие функции — независимые сервисы. Эти сервисы могут быть написаны на разных языках программирования, поскольку общаются друг с другом при помощи REST API (например, с использованием JSON или Thrift).

В каких пропорциях делить код, решает разработчик, но Сэм Ньюмен (Sam Newman), автор книги «Создание микросервисов», рекомендует выделять на микросервис столько строк кода, сколько команда сможет воспроизвести за две недели. По его словам, это позволит избежать излишнего «раздувания» архитектуры.

Чаще всего микросервисы запускаются в так называемых контейнерах. Эти контейнеры доступны по сети другим микросервисами и приложениям, а управляет ими всеми система оркестровки: примерами могут быть Kubernetes, Docker Swarm и др.

Преимущества:

* Микросервисная архитектура упрощает масштабирование приложений. Чтобы внедрить новую функцию достаточно написать новый сервис. Если функция стала не нужна, микросервис можно отключить.
* Каждый микросервис — это отдельный проект, потому работу над ними легко распределить между командами разработчиков.

Недостатки:

* Сложно искать ошибки. В отличие от монолитных систем (когда все функции находятся в одном ядре), бывает сложно определить, почему «упал» запрос. За деталями приходится идти в логи «виновного» процесса (если их несколько, то проблема усугубляется).
* Появляются дополнительные накладные расходы на передачу сообщений между микросервисами. По нашим оценкам, рост сетевых издержек может достигать 25%.
* Необходимость мириться с концепцией eventual consistency (согласованность в конечном счёте). У микросервисов есть собственные хранилища данных, к которым обращаются другие микросервисы. Информация об изменении этих данных распространяется по системе не мгновенно. Потому возникают ситуации, когда у некоторых микросервисов (пусть и на крайне короткий промежуток времени) оказываются устаревшие данные.

Где использовать:

* В крупных проектах с высокой нагрузкой. Например, микросервисы используются стриминговыми платформами. Системы доставки контента и иные вспомогательные сервисы можно масштабировать независимо друг от друга, подстраиваясь под изменения нагрузки.
* В системах, использующих «разномастные» ресурсы. Если одной части приложения нужно больше процессорного времени, а второй — памяти, то имеет смысл разделить их на микросервисы. После чего их можно захостить на разных машинах — с мощным CPU или большим объемом памяти соответственно.
* Когда нужна безопасность. Так как микросервисы изолированы и общаются по API, можно гарантировать, что передаваться будет только та информация, которая нужна тому или иному сервису. Это важно при работе с паролями или данными платёжных карт.

2.1.2 Выбранная архитектура

Выбор пал на событийно-ориентированную архитектура с элементами многоуровневой архитектуры. Данный выбор поможет решить поставленные задачи:

* База данных ингредиентов:
* Возможность вносить и удалять ингредиенты.
* Обеспечить уникальность добавленного ингредиента.
* Возможность сохранять и загружать состояние БД из файлов.
* База данных продуктов в наличии:
* Возможность вносить и удалять продукты.
* Обеспечить уникальность добавленного продукта.
* Возможность сохранять и загружать состояние БД из файлов.
* База данных рецептов:
* Возможность вносить и удалять рецепты.
* Обеспечить уникальность добавленного рецепта.
* Возможность сохранять и загружать состояние БД из файлов.
* Возможность фильтрации рецептов по параметрам:
  + Избранное.
  + Наличие необходимых продуктов.
* Возможность сортировки рецептов по параметрам:
* Рейтинг.
* Наименование.
* Логирование процесса выполнения программы.
* Сущность или система для унифицированной, удобной работы с различными типами файлов.
* Предусмотреть интерфейсы для дальнейшего подключения графических окон управления представленными программными модулями.
* Предусмотреть интерфейсы для дальнейшего подключения интеграции FTP сервера для синхронизации данных на нескольких устройствах.
* Предусмотреть интерфейсы для дальнейшего подключения интеграции с Telegram.

Для ознакомления с построенной UML диаграммой классов смотреть Приложение Б. Рисунок 3. Для получения справки по UML диаграммам смотреть Приложение В.

Для ознакомления с блок-схемой аутентификации пользователя смотреть Приложение Г. Рисунок 9.

Для ознакомления с блок-схемой основного алгоритма подключенного к GUI смотреть Приложение Д. Рисунок 10.

2.1.3 Паттерны проектирования

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

* Проблема, которую решает паттерн.
* Мотивации к решению проблемы способом, который предлагает паттерн.
* Структуры классов, составляющих решение.
* Примера на одном из языков программирования.
* Особенностей реализации в различных контекстах.
* Связей с другими паттернами.

Такой формализм в описании позволил создать обширный каталог паттернов, проверив каждый из них на состоятельность.

Порождающие шаблоны — шаблоны проектирования, которые имеют дело с процессом создания объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

2.1.3.1 Singleton

Singleton (Одиночка) — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Одиночка решает сразу две проблемы, нарушая принцип единственной ответственности класс.

Гарантирует наличие единственного экземпляра класса. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.

Предоставляет глобальную точку доступа. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без ведома разработчика.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Интересно, что сейчас паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Применимость:

* Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).
* Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.
* Когда хочется иметь больше контроля над глобальными переменными.
* В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому можно всегда быть увереным в наличии лишь одного объекта-одиночки.
* Тем не менее, в любой момент можно расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод getInstance).

Шаги реализации:

* Добавление в класс приватного статического поля которое будет содержать одиночный объект.
* Объявление статического создающего метода, который будет использоваться для получения одиночки.
* Добавление «ленивой инициализации» (создание объекта при первом вызове метода) в создающий метод одиночки.
* Конструктор класса приватный.
* В клиентском коде заменить вызовы конструктора одиночка вызовами его создающего метода.

Преимущества:

* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.

Недостатки:

* Нарушает принцип единственной ответственности класса.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестировании.

Отношения с другими паттернами:

* Фасад можно сделать Одиночкой, так как обычно нужен только один объект-фасад.
* Паттерн Легковес может напоминать Одиночку, если для конкретной задачи получилось свести количество объектов к одному. Но, между паттернами есть два кардинальных отличия:
  + В отличие от Одиночки, можно иметь множество объектов-легковесов.
  + Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.
* Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

2.1.3.2 Observer

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Представьте, что имеется два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

Увидев, как складно всё работает, можно выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

Аналогия из жизни:

После того как оформили подписку на газету или журнал, больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо домой сразу после их выхода.

Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Можно в любой момент отказаться от подписки, и журнал перестанет приходить.

Структура:

* Издатель владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.
* Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в общем интерфейсе подписчиков.
* Подписчик определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.
* Конкретные подписчики выполняют что-то в ответ на оповещение, пришедшее от издателя. Эти классы должны следовать общему интерфейсу подписчиков, чтобы издатель не зависел от конкретных классов подписчиков.
* По приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант — передавать через параметры весь объект издателя, чтобы подписчик мог сам получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.
* Клиент создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

Применимость:

* Когда после изменения состояния одного объекта требуется что-то сделать в других, но не знаете наперёд, какие именно объекты должны отреагировать.

Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

* Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

Шаги реализации:

* Разбейте функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
* Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
* Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
* Нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.
* Но если интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае можно поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.
* Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
* Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с происшедшим событием.
* Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае будете вынуждены привязать класс подписчика к конкретному классу издателя.
* Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

Преимущества:

* Издатели не зависят от конкретных классов подписчиков и наоборот.
* Вы можете подписывать и отписывать получателей на лету.
* Реализует принцип открытости/закрытости.

Недостатки:

* Подписчики оповещаются в случайном порядке.

Отношения с другими паттернами:

* Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
* Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
* Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
* Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
* Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
* Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.

Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

2.2 Хранение данных.

Введем новые понятия:

* UUID (Universally unique identifier) - стандарт идентификации, используемый в создании программного обеспечения, стандартизированный Open Software Foundation как часть DCE — среды распределённых вычислений. Основное назначение UUID — это позволить распределённым системам уникально идентифицировать информацию без центра координации.
* JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript, но может использоваться в любом языке программирования. Легко читается человеком и машиной. Часто используется в REST API (чаще, чем XML из-за лучшей читабельности и меньшего веса).

Корректные значения JSON:

* JSON-объект — неупорядоченное множество пар «ключ:значение», заключённое в фигурные скобки { }.
* Массив — упорядоченный набор значений, разделённых запятыми. Находится внутри квадратных скобок [ ].
* Число (целое или вещественное).
* Литералы true (логическое значение «истина»), false (логическое значение «ложь») и null/None.
* Строка.

Все данные сущностей хранятся в JSON формате.

2.2.1 Структура хранения данных определенных сущностей

2.2.1.1 ingredientsDB

Так называемая база ингредиентов представляет собой JSON-объект (смотреть Рисунок 11), который содержит набор объектов, каждый из которых описывает измеряемый параметр или величину и содержит следующие свойства:

* "uuid": свойство, которое содержит уникальный идентификатор (uuid) объекта.
* "\_name": свойство, которое содержит имя ингредиента.
* "\_measure": свойство, которое содержит единицу измерения ингредиента. Например, "килограмм", "литр", "столовая ложка" и т.д.



Рисунок 11 – «ingredientsDB.json»

2.2.1.2 storage

Так называемая база продуктов представляет собой JSON-объект (смотреть Рисунок 12), состоящий из множества свойств:

* "uuid": уникальный идентификатор в формате UUID и связано со свойством "amount", которое содержит числовое значение.



Рисунок 12 – «storage.json»

2.2.1.3 recipe

Каждый рецепт представляет собой файл с uuid вместо названия с расширением json. Также рецепт представляет собой JSON-объект (смотреть рисунок 13), состоящий из следующего множества свойств:

* "\_name": свойство, которое содержит имя рецепта.
* "\_ingredients": свойство, которое содержит объект "ingredients", который содержит свойство "uuid" (идентификатор) и количество (amount) ингредиентов, которые используются в рецепте. Количество и формат этих свойств может различаться в зависимости от количества и разнообразия ингредиентов.
* "\_cookingSteps": свойство, которое содержит массив "cookingSteps" с последовательностью шагов, необходимых для приготовления блюда.
* "\_description": свойство, которое содержит описание рецепта (description).
* "\_isFavorite": свойство, которое содержит булевое значение, указывающее, является ли рецепт избранным (true) или нет (false).
* "\_rating": свойство, которое содержит дробное число, представляющее рейтинг рецепта.



Рисунок 13 – «recipe.json»

3 ПРАКТИЧЕСКАЯ ЧАСТЬ

Для начала следует ввести следующие понятия:

* Версионирование — разработка и управление несколькими выпусками продукта, которые имеют тот же общий функционал, но усовершенствованы, модернизированы либо индивидуализированы. Коротко, версия говорит об изменении продукта.
* Система контроля версий VCS (Version Control System) — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.

При разработке обязательно должна использоваться VCS. Выбран Git.

* Интерфейс командной строки CLI (Command-line interface) — способ взаимодействия между человеком и компьютером путём отправки компьютеру команд, представляющих собой последовательность символов.
* Графический интерфейс пользователя, графический пользовательский интерфейс GUI (Graphical User Interface) — система средств для взаимодействия пользователя с компьютером, основанная на представлении всех доступных пользователю системных объектов и функций в виде графических компонентов экрана.
* Прототипирование — это процесс в работе дизайнера, когда создается «иллюзию» рабочего интерфейса на базе нарисованных макетов или вайрфреймов в короткие временные сроки.

3.1 Версионирование

Все версии условно обозначаются: V0.0.0

Начальная версия: V0.0.0

Версии до V1.0.0 - относятся к CLI

Версии от V1.0.0 - относятся к GUI

3.2 Описание программных модулей

Пакет в Python – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл \_\_init\_\_.py. Пакеты используются для формирования пространства имен, что позволяет работать с модулями через указание уровня вложенности (через точку).

3.2.1 Пакет entities

Описывает сущности Ингредиента и Рецепта (смотреть рисунок 14).



Рисунок 14 – «Пакет entities»

3.2.1.1 ingredient.py

Модуль описывает класс "Ingredient" представляющий одноименную сущность Ингредиент (смотреть рисунок 15).



Рисунок 15 – «Класс Ingredient»

3.2.1.2 recipe.py

Модуль описывает класс "Recipe" представляющий одноименную сущность Рецепт (смотреть рисунок 16).

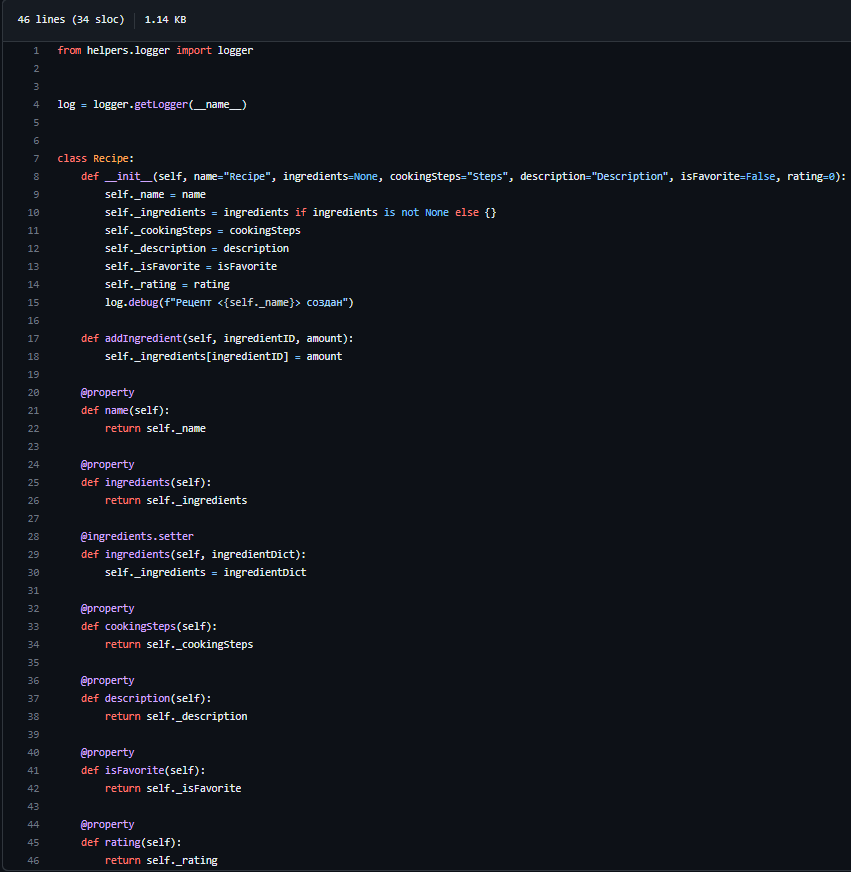


Рисунок 16 – «Класс Ingredient»

3.2.2 Пакет DataStructures

Описывает сущности Книга, Хранилище, База ингредиентов (смотреть Рисунок 17).

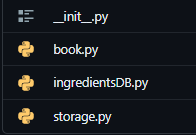


Рисунок 17 – «Пакет DataStructures»

3.2.2.1 book.py

Модуль определяет класс "RecipeBook" (смотреть Рисунки 18, 19, 20) описывающий одноименную сущность представляющую хранилище рецептов.

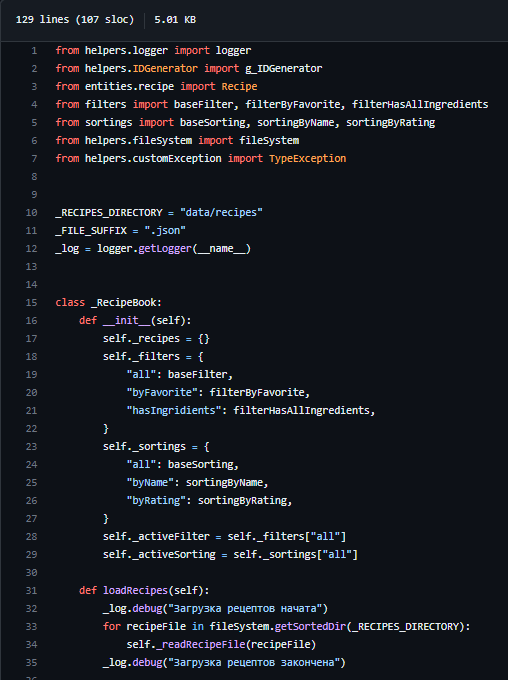


Рисунок 18 – «Класс Book (1)»

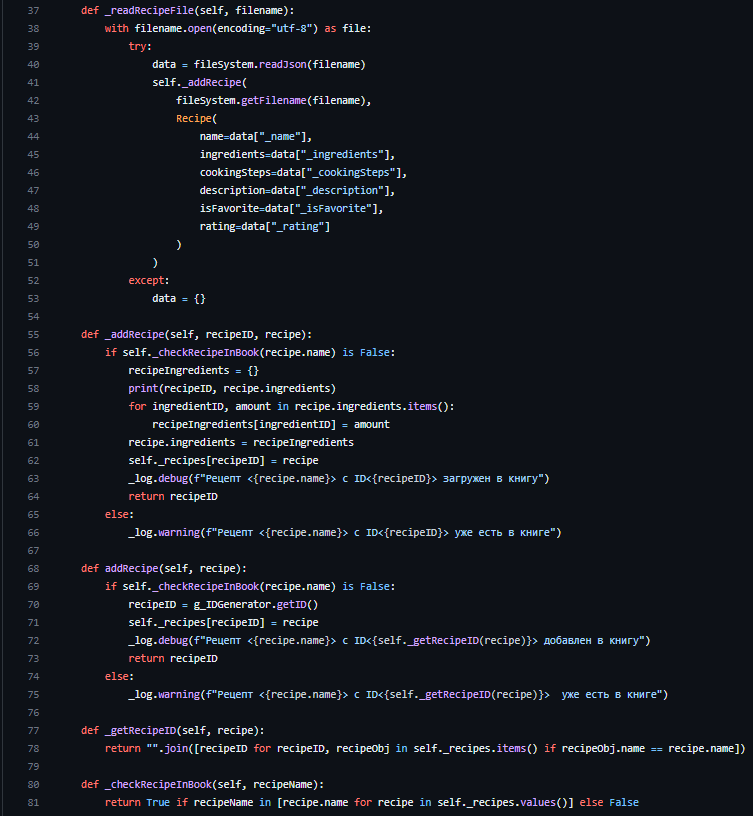


Рисунок 19 – «Класс Book (2)»

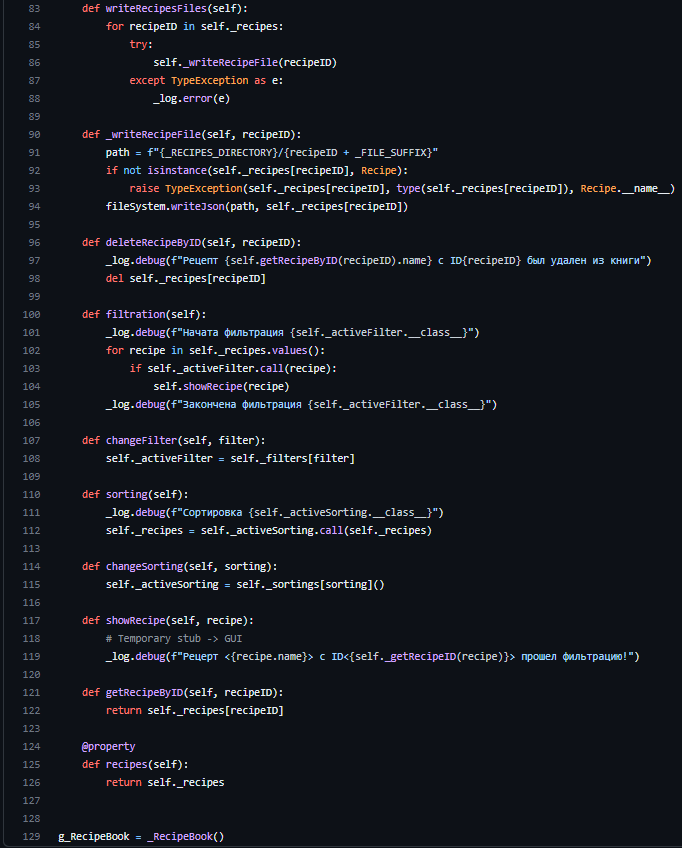


Рисунок 20 – «Класс Book (3)»

3.2.2.2 Модуль ingredientsDB.py

Модуль определяет класс "IngredientsDatabase" (смотреть Рисунки 21, 22) описывающий одноименную сущность представляющую базу ингредиентов.

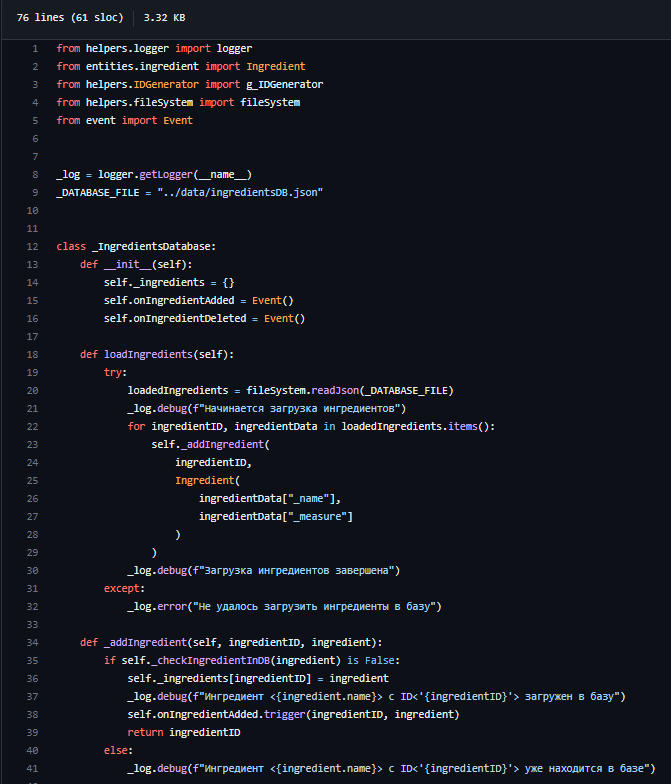


Рисунок 21 – «Класс IngredientsDatabase (1)»

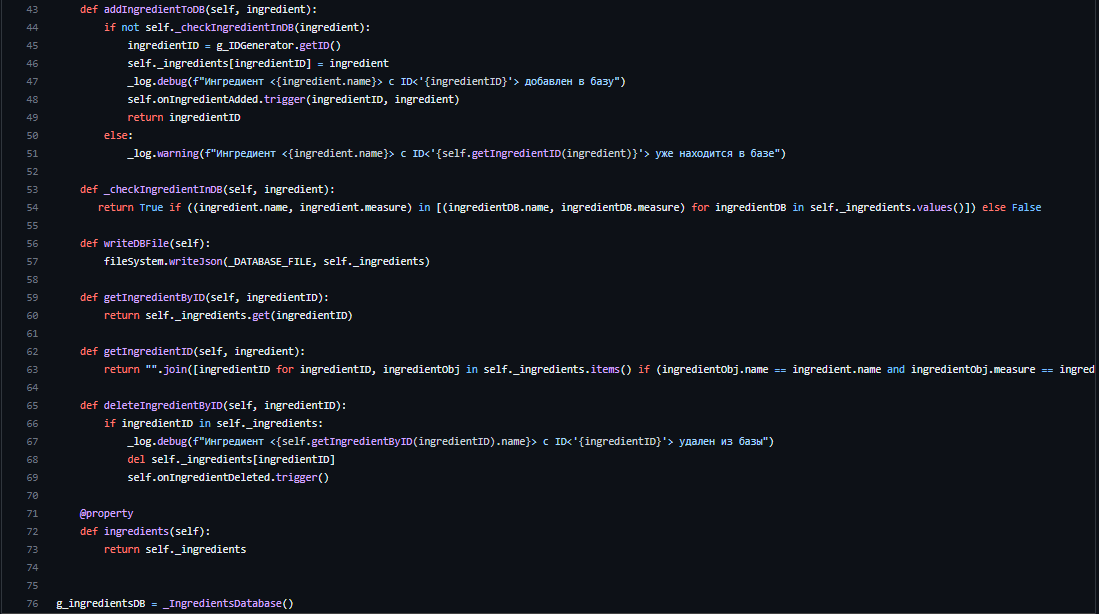


Рисунок 22 – «Класс IngredientsDatabase (2)»

3.2.2.3 Модуль storage.py

Модуль определяет класс "Storage" (смотреть Рисунки 23, 24) описывающий одноименную сущность представляющую базу продуктов.

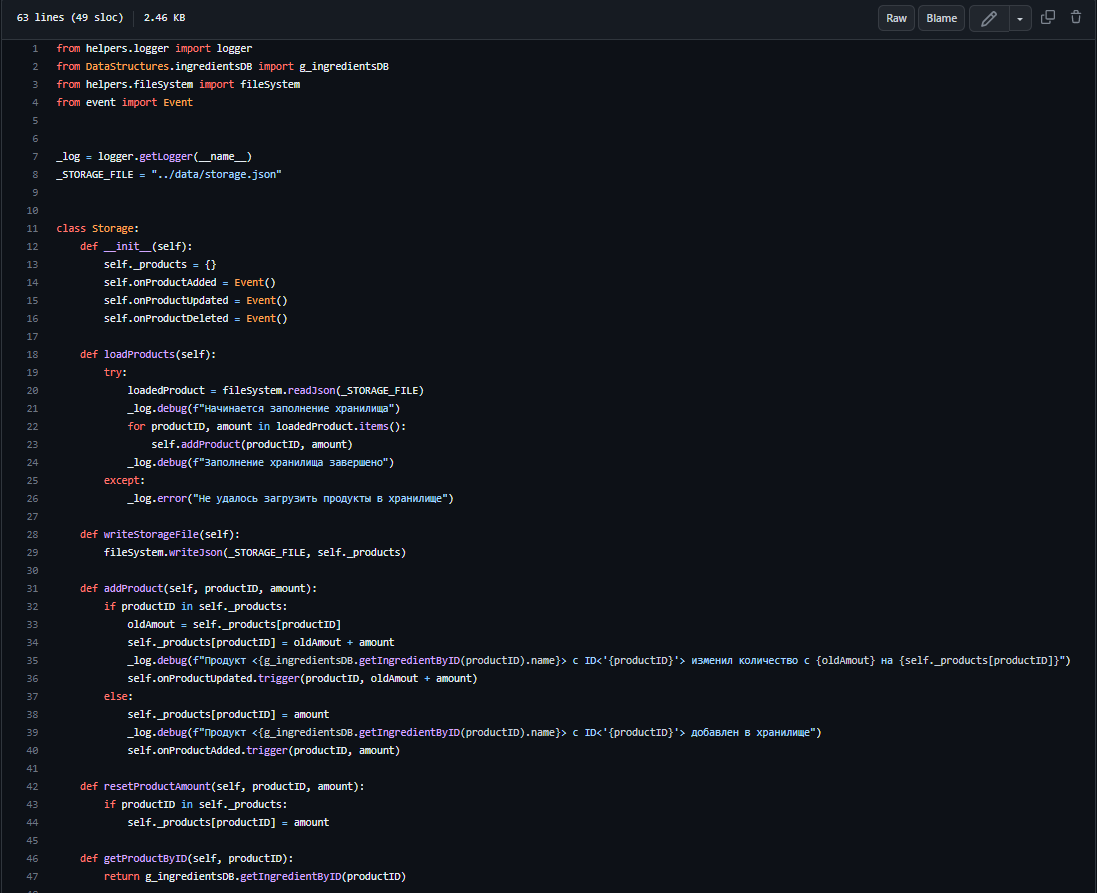


Рисунок 23 – «Класс Storage (1)»

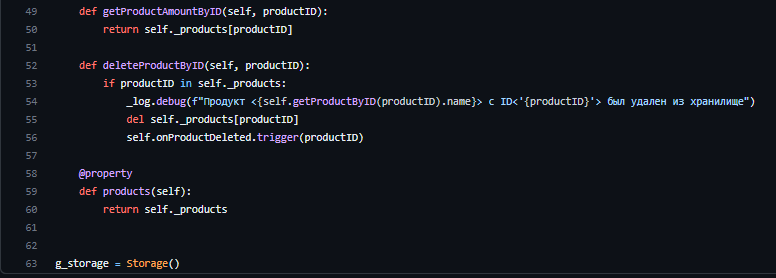


Рисунок 24 – «Класс Storage (2)»

3.2.3 Пакет helpers

Пакет предоставляет инструментарий различного рода (смотреть Рисунок 25).

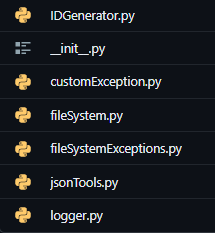


Рисунок 25 – «Пакет helpers»

3.2.3.1 Модуль IDGenerator.py

Модуль определяет класс IDGenerator реализующий генерацию ID по технологии UUID4 (смотреть Рисунок 26).

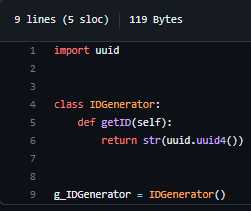


Рисунок 26 – «Класс IDGenerator»

3.2.3.2 Модуль jsonTools.py

Модуль описывает механизм сериализации JSON-объектов (смотреть Рисунок 27).

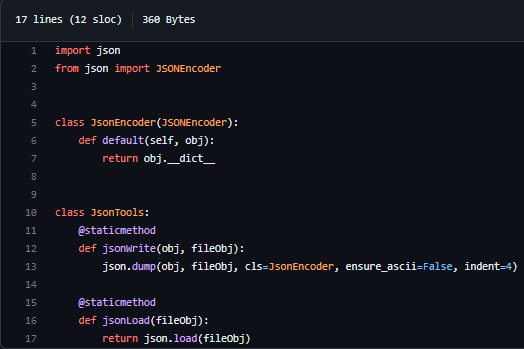


Рисунок 27 – «Модуль jsonTools.py»

3.2.3.3 Модуль logger.py

Модуль (смотреть Рисунок 28) инициализирует логер из подмодуля Logger <https://github.com/LunexCoding/Logger> .

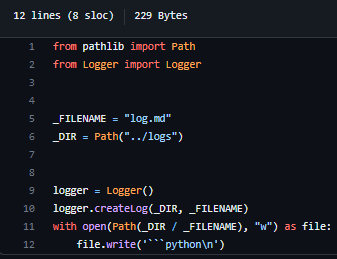


Рисунок 28 – «Модуль logger.py»

3.2.3.4 Модуль fileSystem.py

Модуль определяет класс FileSystem реализующий управление файловой системой (смотреть Рисунки 29, 30. 31, 32).



Рисунок 29 – «Класс FileSystem (1)»



Рисунок 30 – «Класс FileSystem (2)»



Рисунок 31 – «Класс FileSystem.py (3)»

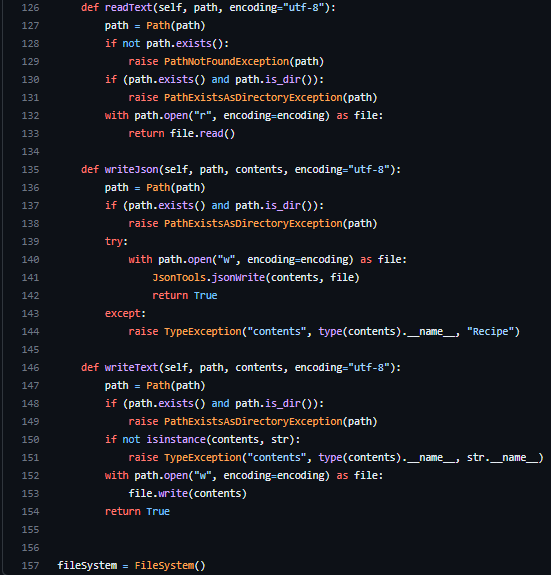


Рисунок 32 – «Класс FileSystem (4)»

3.2.3.5 Модуль fileSystemExceptions.py

Модуль описывает собственные исключения (смотреть Рисунок 33) для модуля файловой системы.



Рисунок 33 – «Модуль fileSystemExceptions.py»

3.2.3.6 Модуль customException.py

Модуль описывает собственные исключения (смотреть Рисунок 34), которые можно будет использовать в других модулях при необходимости.

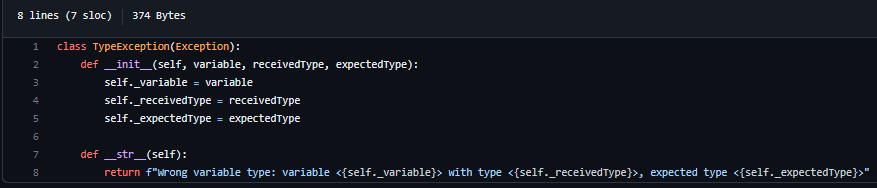


Рисунок 34 – «Модуль customException.py»

3.2.4 Модуль filters.py

Модуль определяет различные фильтры для рецептов (смотреть Рисунок 35).

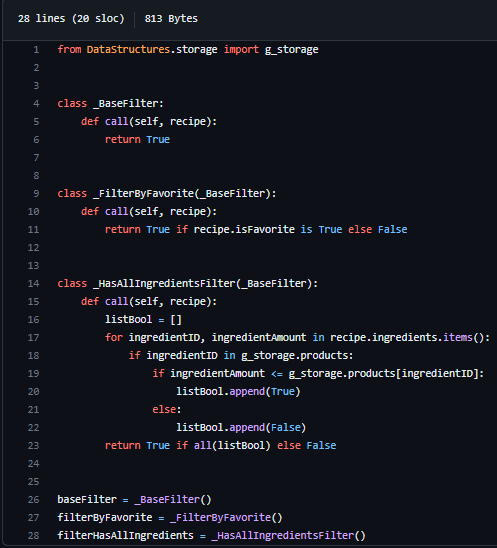


Рисунок 35 – «Модуль filters.py»

3.2.5 Модуль sortings.py

Модуль определяет различные сортировки для рецептов (смотреть Рисунок 36).

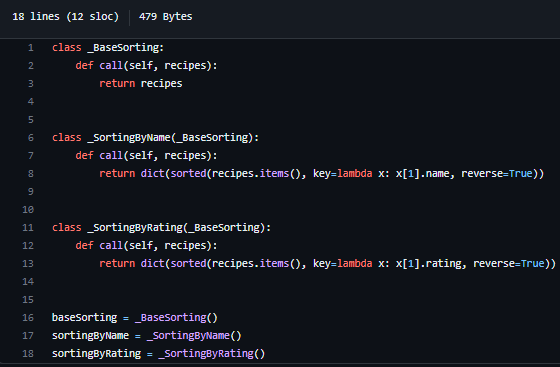


Рисунок 36 – «Модуль sortings.py»

3.4 Внедрение прототипа пользовательского интерфейса

3.4.1 Прототипы интерфейса

Были разработаны следующие прототипы интерфейса:

* Виджет ингредиента (смотреть приложение Е. Рисунок 37).
* Виджет продукта (смотреть приложение Е. Рисунок 38).
* Главная страница (смотреть Приложение Ё. Рисунок 39).
* Страница ингредиентов (смотреть Приложения Ж. Рисунок 40. Приложение З. Рисунок 41).
* Страница добавления ингредиента (смотреть Приложение И. Рисунок 42).
* Страница продуктов (смотреть Приложение Й. Рисунок 43. Приложение К. Рисунок 44).
* Страница рецептов (смотреть Приложение Л. Рисунок 45).
* Уведомление (смотреть Приложение М. Рисунок 46. Приложение Н. Рисунок 47).
* Раскрытое левое меню (смотреть Приложение О. Рисунок 48. Приложение П. Рисунок 49. Приложение Р. Рисунок 50).
* Раскрытое правое меню (смотреть Приложение С. Рисунок 51. Приложение Т. Рисунок 52).
* Раскрытая левая панель (смотреть Приложение У. Рисунок 53).

3.4.2 Описание программных модулей для GUI

3.4.2.1 Пакет widgets

Пакет описывает собственные разработанные Qt виджеты и формы (смотреть Рисунок 53).

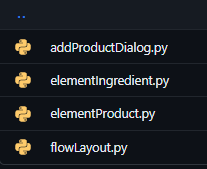


Рисунок 53 – «Пакет widgets»

3.4.2.1.1 Модуль addProductDialog.py

Модуль описывает класс AddProductDialog представляющий реализацию Qt виджета продукта (смотреть Рисунок 54).

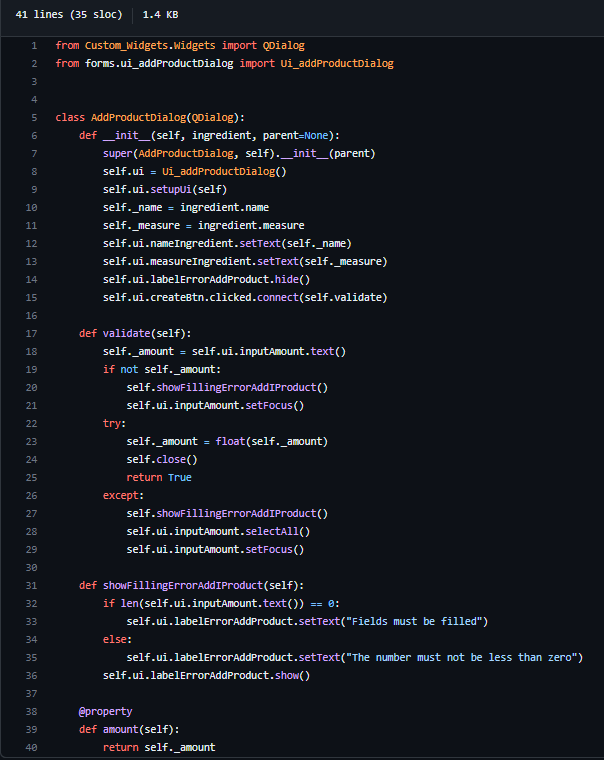


Рисунок 54 – «Класс AddProductDialog»

3.4.2.1.2 Модуль elementIngredient.py

Модуль определяет класс ElementIngredient представляющий реализацию диалогового окна Qt для создания продукта из ингредиента (смотреть Рисунок 55).

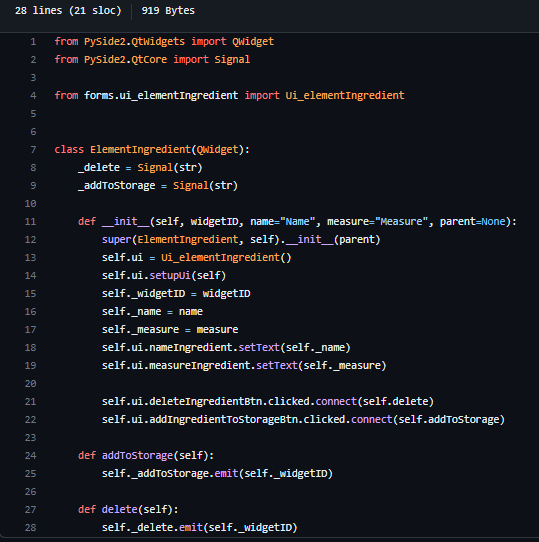


Рисунок 55 – «Класс ElementIngredient»

3.4.2.1.3 Модуль elementProduct.py

Модуль описывает класс ElementProduct прелставлящий реализацию Qt виджета продукта (смотреть Рисунки 56 и 57).

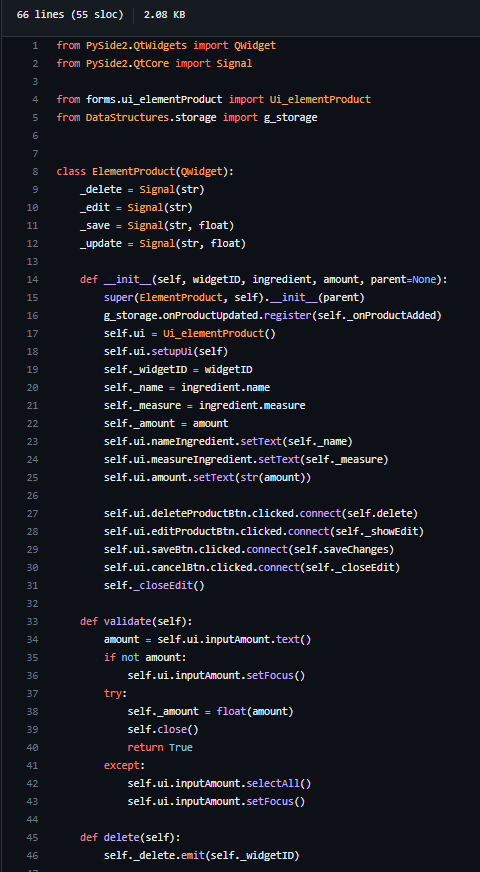


Рисунок 56 – «Класс ElementProduct (1)»

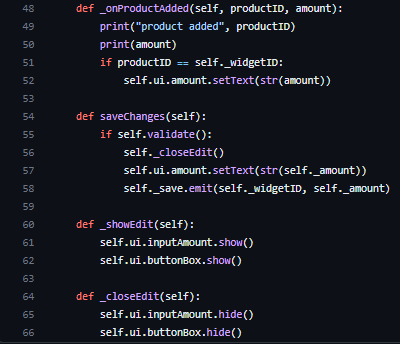


Рисунок 57 – «Класс ElementProduct (2)»

3.4.2.1.4 Модуль flowLayout.py

Модуль описывает класс FlowLayout представляющий пользовательский виджет, предоставляемый в библиотеке Qt для организации компоновки дочерних элементов в виде списка, где элементы располагаются построчно и при достижении конца строки переносятся на следующую строку. Это позволяет обеспечить гибкую и адаптивную компоновку элементов на форме пользовательского интерфейса в зависимости от размеров окна или других параметров (смотреть Рисунки 58 и 59).

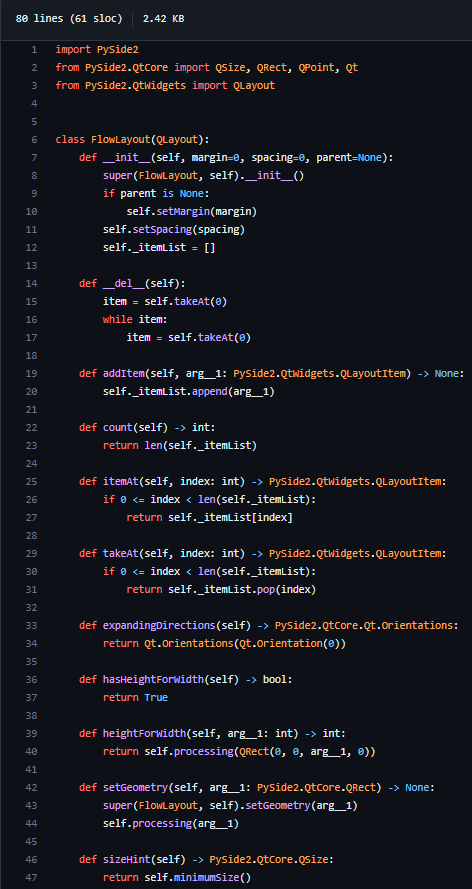


Рисунок 58 – «Класс FlowLayout (1)»



Рисунок 59 – «Класс FlowLayout (2)»

3.4.1.2 Модуль event.py

Модуль описывает класс Event представляющий простую реализацию шаблона проектирования "Наблюдатель" (смотреть рисунок 60). Класс позволяет зарегистрировать обратный вызов (callback) и затем вызывать этот обратный вызов при возникновении определенного события.

Конструктор класса инициализирует список \_\_callbacks, который будет использоваться для хранения всех зарегистрированных обратных вызовов.

Метод register(callback) добавляет переданный обратный вызов в список \_\_callbacks.

Метод remove(callback) удаляет переданный обратный вызов из списка \_\_callbacks.

Метод trigger(\*args, \*\*kwargs) вызывает все зарегистрированные обратные вызовы, передавая им все переданные аргументы.

Метод clear() очищает список \_\_callbacks.

Метод destroy() удаляет все зарегистрированные обратные вызовы, вызывая метод clear().

Перегруженный метод iadd(self, callback) позволяет использовать оператор "+=" для регистрации обратных вызовов, т.е. вместо вызова метода register(callback) можно написать event += callback.

Класс Event может быть использован для уведомления объектов-наблюдателей об изменении состояния других объектов.

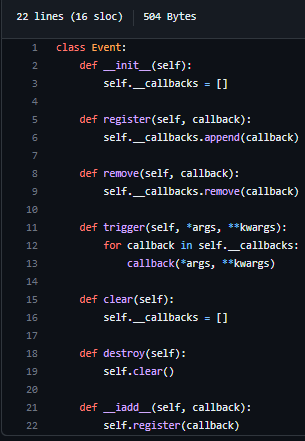


Рисунок 60 – «Класс Event»

В данном случае Наблюдателем будет являться класс реализующий графическое окно. Наблюдатель ожидает любое событие которое произойдет.

Итак, вот события:

* Создание ингредиента.
* Удаление Ингредиента.
* Создание продукта.
* Изменение продукта.
* Удаление продукта.

3.4.1.3 Модуль main.py

Модуль определяет класс MainWindow (смотреть Рисунки 61, 62, 63, 64), а также является точкой входа.

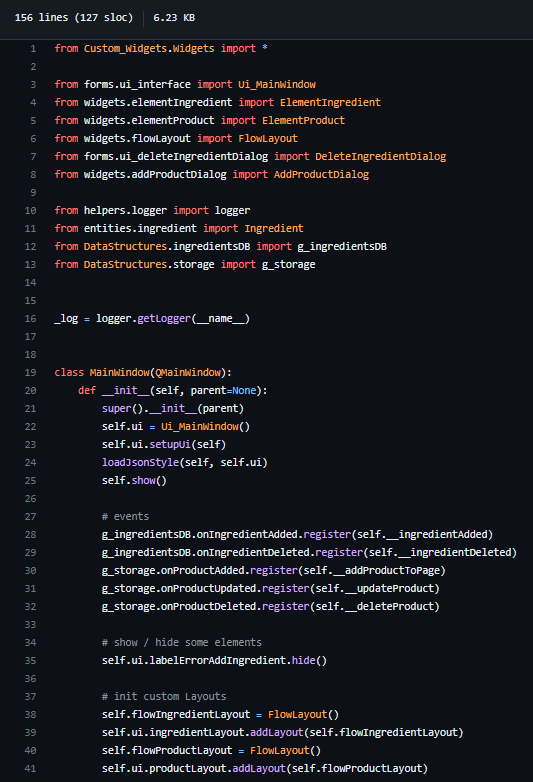


Рисунок 61 – «Класс MainWindow (1)»

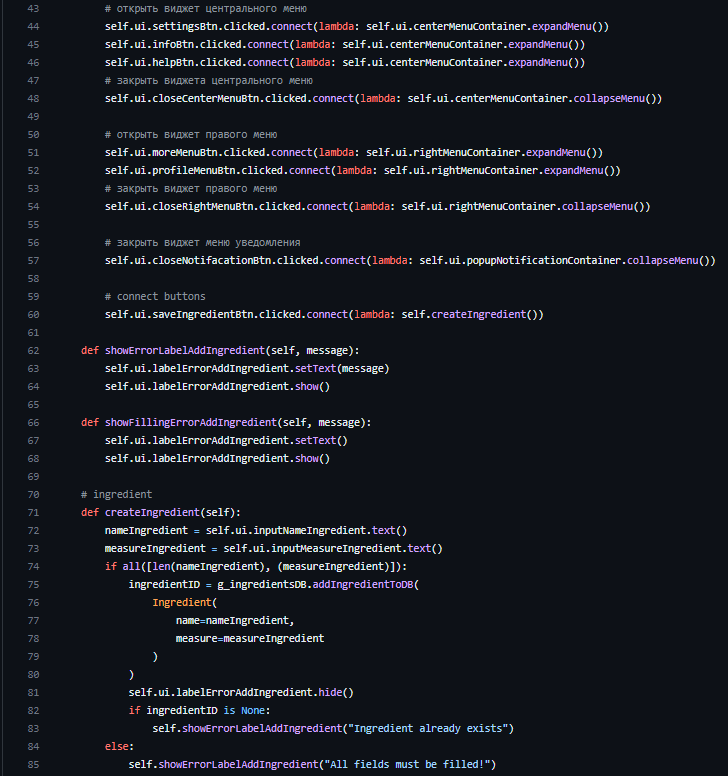


Рисунок 62 – «Класс MainWindow (2)»

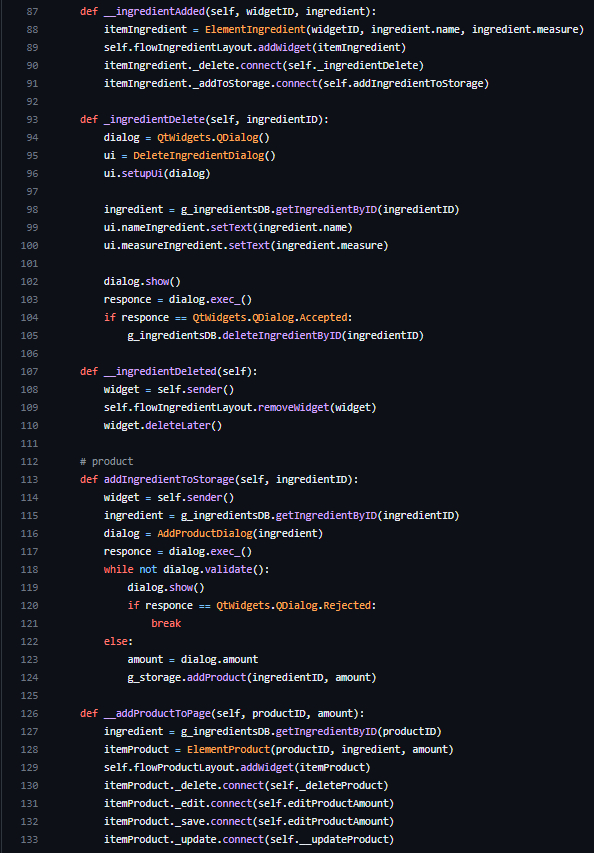


Рисунок 63 – «Класс MainWindow (3)»

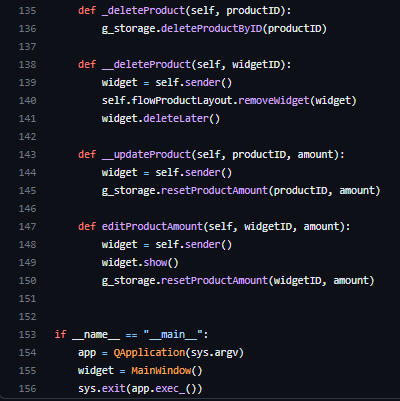


Рисунок 64 – «Класс MainWindow (4)»

3.5 Отладка

Тестирование — это процесс проверки программного обеспечения на соответствие требованиям и ожиданиям пользователей, а также на выявление ошибок, дефектов и недочетов.

Основная цель тестирования программного обеспечения - убедиться в том, что программа работает корректно и в соответствии с заданными требованиями. Это достигается путем создания тестовых сценариев и набора тестовых данных, которые позволяют проверять работу программы на различных этапах ее разработки и тестирования.

Процесс тестирования может быть проведен как вручную, так и с помощью автоматизированных инструментов, таких как фреймворки для автоматизации тестирования, которые позволяют создавать, запускать и анализировать результаты тестов автоматически.

Тестирование в программировании является важной частью процесса разработки программного обеспечения, так как позволяет убедиться в том, что программа работает корректно и соответствует требованиям, а также позволяет выявить ошибки и дефекты, которые могут быть исправлены до выпуска программного обеспечения в продакшн.

unittest - это стандартный фреймворк для модульного тестирования в Python. Он входит в стандартную библиотеку Python и предоставляет набор инструментов для создания тестовых сценариев и проверки результатов работы функций и классов.

unittest позволяет создавать наборы тестовых методов, которые могут быть запущены автоматически. Каждый тестовый метод обычно проверяет один аспект поведения функции или класса и возвращает результат в виде утверждений (assertions). Если все утверждения в тестовом методе выполняются успешно, то тест считается пройденным.

unittest предоставляет множество возможностей для настройки тестов, таких как настройка окружения, использование параметризованных тестов, управление выполнением тестов и многие другие.

3.5.1 Тестируемый функционал

Основной спектр кода, который нужно покрывать автотестами - базовый бекенд функционал.

Выделено 4 модуля тестерования:

* test\_book.py

Есть база данных рецептов.

В случае повреждения БД потенциальный риск не только неправильной работы программы но и потери данных пользователя без возможности восстановления.

Покрываем тестами создание, добавление, удаление, дублирование, сохранение и загрузка данных, смену фильтра.

Модуль представляет собой тесты для класса "\_RecipeBook", который представляет книгу рецептов.

Методы:

* "getRecipeIDByRecipe" - получает объект рецепта и возвращает его "ID".
* "test\_addRecipe" - проверяет добавление рецептов в книгу. Он вызывает метод addRecipe для "\_recipe1" дважды и для "\_recipe2" один раз, затем получает их "ID" с помощью "getRecipeIDByRecipe" и проверяет, что они присутствуют в книге и имеют правильные имена.
* "test\_changeFilter" - проверяет изменение фильтра для книги. Он вызывает метод "changeFilter" трижды с различными аргументами и проверяет, что типы объектов фильтров правильные.
* "test\_deleteRecipeByID" - проверяет удаление рецепта по его "ID". Он получает "ID" для "\_recipe1" с помощью "getRecipeIDByRecipe", затем удаляет его из книги и проверяет, что он больше не присутствует в книге.
* "test\_writeRecipesFilesAndLoadRecipes" - проверяет запись рецептов в файлы и загрузку их обратно в книгу. Он получает "ID" для "\_recipe2" с помощью "getRecipeIDByRecipe", затем записывает все рецепты из книги в файлы, используя метод "writeRecipesFiles". Затем он проверяет, что файл с рецептом "\_recipe2" существует, удаляет все рецепты из книги и загружает их обратно из файлов, используя метод "loadRecipes". Наконец, он проверяет, что количество рецептов больше нуля, что "\_recipe2" присутствует в книге и имеет правильное имя.

Переменная "\_FIXTURES\_DIRECTORY" определяет директорию с фикстурами, а "\_RECIPES\_DIRECTORY" - директорию для хранения рецептов. "\_FILE\_SUFFIX" - это суффикс для имен файлов с рецептами.

Метод "setUpClass" выполняется перед всеми тестами и инициализирует книгу рецептов "\_book" и объект "FileSystem". Кроме того, он создает два рецепта "\_recipe1" и "\_recipe2".

Метод "tearDownClass" выполняется после всех тестов и удаляет директорию с фикстурами и рецептами.

Функция "unittest.main()" запускает все тесты в классе "TestRecipeBook". Если все тесты успешно проходят, значит, функциональность класса "\_RecipeBook" работает правильно.

* test\_fileSystem.py

В программе используется кастомный модуль файловый системы, который был написан в связи с необходимостью логирования.

Хотя он является лишь враппером над общедоступной библиотекой pathlib, подобные модули являются базовыми для работоспособности программного обеспечение и хорошей практикой является покрытие их тестами.

Модуль представляет собой набор модульных тестов для класса "FileSystem", который реализует работу с файловой системой. В каждом методе определены тесты для проверки различных функций класса.

Перед началом каждого теста выполняется метод "setUpClass", который инициализирует экземпляр класса "FileSystem" и создает каталог fixtures, который будет использоваться в качестве тестового окружения. Метод "tearDownClass" удаляет этот каталог после выполнения всех тестов.

Методы:

* "test\_create" - проверяет, создается ли файл с заданным именем и содержанием, а также возникают ли исключения при попытке создать файл с неверным типом или существующим каталогом. Этот метод также проверяет, работает ли функция "wipe", которая удаляет файл перед созданием нового файла.
* "test\_getFilename" - проверяет, возвращает ли функция "getFilename" правильное имя файла из заданного пути. Этот метод также проверяет, работает ли функция с расширением имени файла.
* "test\_exists" - проверяет, существует ли заданный файл или каталог.
* "test\_isDir" - проверяет, является ли заданный путь каталогом.
* "test\_isEmpty" - проверяет, является ли заданный каталог пустым.
* Метод "test\_isFile" - проверяет, является ли заданный путь файлом.
* Метод "test\_makeDir" - проверяет, создается ли каталог с заданным именем, а также возникают ли исключения при попытке создать каталог с неверным типом или уже существующим файлом или каталогом. Этот метод также проверяет, работает ли функция "recreate", которая пересоздает каталог, если он уже существует.
* Метод "test\_makeDirs" - проверяет, создаются ли каталоги с заданным именем и путем, а также возникают ли исключения при попытке создать каталоги с неверным типом или уже существующим файлом или каталогом. Этот метод также проверяет, работает ли функция recreate, которая пересоздает каталог, если он уже существует.
* "test\_remove" - проверяет, удаляется ли заданный файл.
* "test\_removeDir" - проверяет, удаляется ли заданный каталог.
* "test\_removeTree" - проверяет, удаляется ли заданное дерево каталогов.
* "test\_readJson" - проверяет, может ли функция "readJson" прочитать файл JSON из заданного пути. Этот метод также проверяет, генерируется ли исключение, если файл не найден.
* test\_ingredientsDB.py

Есть база данных ингредиентов.

В случае повреждения БД потенциальный риск не только неправильной работы программы но и потери данных пользователя без возможности восстановления.

Покрываем тестами создание, добавление, удаление, дублирование, сохранение и загрузка данных.

Модуль содержит тестовый класс "TestIngredientsDB", который наследуется от класса "unittest.TestCase" и содержит три метода тестирования:

* "test\_addIngredientToDB" - тестирует метод "addIngredientToDB" класса "\_IngredientsDatabase", который добавляет ингредиенты в базу данных. В методе тестирования создаются несколько ингредиентов, они добавляются в базу данных, и затем проверяется, что ингредиенты были добавлены в базу данных, и что их свойства (название и единица измерения) соответствуют ожидаемым значениям.
* "test\_deleteIngredientByID" - тестирует метод "deleteIngredientByID" класса "\_IngredientsDatabase", который удаляет ингредиент по его идентификатору из базы данных. В методе тестирования выбирается один из ранее добавленных ингредиентов, он удаляется из базы данных, и затем проверяется, что ингредиент был удален.
* "test\_writeDBFileAndLoadIngredients" - тестирует методы "writeDBFile" и loadIngredients класса "\_IngredientsDatabase", которые записывают базу данных в файл и загружают ее из файла соответственно. В методе тестирования база данных записывается в файл, проверяется, что файл существует, затем база данных очищается и загружается из файла, и затем проверяется, что загруженные данные соответствуют ожидаемым значениям.

В начале класса "TestIngredientsDB" определены методы "setUpClass" и "tearDownClass", которые вызываются один раз в начале и в конце тестов соответственно. В методе "setUpClass" инициализируются необходимые для тестирования объекты и данные, а в методе "tearDownClass" освобождаются ресурсы, которые были выделены в методе "setUpClass".

В конце кода вызывается метод "unittest.main()", который запускает тесты в классе "TestIngredientsDB". Если все тесты проходят успешно, то выводится сообщение о том, что все тесты были пройдены. Если же какой-то из тестов не проходит, то выводится информация об ошибке, которую можно использовать для исправления проблемы.

* test\_storage.py

Есть база данных продуктов.

В случае повреждения БД потенциальный риск не только неправильной работы программы но и потери данных пользователя без возможности восстановления.

Покрываем тестами создание, добавление, удаление, дублирование, сохранение и загрузка данных.

Модуль содержит тестовый класс "TestIngredientsDB", который наследуется от класса "unittest.TestCase" и содержит три метода тестирования:

* "test\_addProduct" - проверяет, что метод "addProduct" добавляет ингредиенты в объект "Storage" и что метод "getProductAmountByID" возвращает правильное количество ингредиентов. Он также проверяет, что метод "addProduct" корректно обрабатывает дубликаты.
* "test\_deleteProductByID" - проверяет, что метод "deleteProductByID" удаляет ингредиенты из объекта "Storage".
* "test\_writeAndLoadStorageFile" - проверяет, что методы "writeStorageFile" и "loadProducts" корректно записывают и загружают ингредиенты в файл "storage.json".

Перед началом тестов класс "TestStorage" создает объекты "Ingredient", которые используются для тестирования. Он также создает объект "Storage" и объект "Mock" класса "IngredientsDatabase", который используется в качестве поддельной базы данных для ингредиентов.

Метод "setUpClass" вызывается перед запуском тестового класса и выполняет настройку тестовых данных, включая создание директории "fixtures" и объектов, которые будут использоваться для тестирования. Метод "tearDownClass" вызывается после завершения всех тестов и удаляет директорию "fixtures".

В конце файла вызывается функция "unittest.main()", которая запускает все тесты в классе "TestStorage".

3.5.2 Покрытие тестами

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Кратко о типах архитектур программного обеспечения, и какую из них выбрали мы для IaaS-провайдера. - [Электронный ресурс]: страница Habr.com. – Режим доступа: <https://habr.com/ru/company/1cloud/blog/424911/>

Сэм Ньюмен (Sam Newman), «Создание микросервисов» (Building Microservices), 2015. - [Книга]

Майкл Т. Фишер, Мартин Л. Абботт, «Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise», 2009. - [Книга]

Шаблон проектирования. - [Электронный ресурс]: страница Wikipedia. - Режим доступа: https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD\_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F

Паттерны проектирования. - [Электронный ресурс]: страница refactoring.guru. – Режим доступа: https://refactoring.guru/ru/design-patterns

Паттерн проектирования Singleton (Одиночка). - [Электронный ресурс]: страница refactoring.guru. – Режим доступа: https://refactoring.guru/ru/design-patterns/singleton

UML-диаграммы классов. - [Электронный ресурс]: страница prog-cpp.ru. – Режим доступа: https://prog-cpp.ru/uml-classes/

ПРИЛОЖЕНИЕ А

Таблица 1 – «Календарный план»

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № эпика | Задачи эпика | № спринта в эпике | № спринта | Дата начала спринта | Дата окончания спринта | Продолжительность спринта в днях | Задачи спринта | Результат спринта | Версии,  релизы | Отметка готовности, % |
| 1 | Реализация минимального функционала | 1 | 1 | 20.10.22 | 22.10.22 | 2 | Реализация минимального набора базовых модулей, реализация минимального функционала | ✓ | V0.0.1  V0.0.2 | 6% |
| 2 | 2 | 08.11.22 | 10.11.22 | 2 | Логирование, минимальная реализация модулей: ingredientsDB, storage | ✓ |

Продолжение таблицы 1.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | Проектирование.  Покрытие тестами. | 1 | 3 | 12.11.22 | 14.11.22 | 2 | Проектирование: реорганизация архитектуры, создание UML диаграммы. | ✓ | V0.1.0  V0.1.1 | 12% |
| 2 | 4 | 16.11.22 | 22.11.22 | 6 | Покрытие тестами. | ✓ |
| 3 | 5 | 18.11.22 | 28.11.22 | 10 | Добавлены фильтры рецептов: по имени, по рейтингу, по доступности, по избранному.  Исправлены некоторые ошибки. | ✓ |
| 3 | Избежание дубликатов в базах.  Переход на юниттесты.  Соблюдение стандартов PyFilesystem.  Перевод ID генератора на технологию UUID. | 1 | 6 | 07.02.23 | 17.02.23 | 10 | Изменена структура некоторых сущностей. Избежание дублирования объектов в базах.  Переработан модуль файловой системы. | ✓ | V0.1.2  V0.2.0  V0.2.1  V0.2.2 | 37% |
| 2 | 7 | 18.02.23 | 21.02.23 | 3 | Покрытие тестами базового функционала. Переход с интеграционных тестов на модульные. | ✓ |

Продолжение таблицы 1.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 3 | 8 | 21.02.23 | 26.02.23 | 5 | Модуль файловой системы переписан согласно [документации PyFilesystem](https://docs.pyfilesystem.org/en/latest/interface.html).  Перевод ID генератора на технологию UUID. | ✓ |  |  |
| 4 | Прототип GUI интерфейса.  Формы и кастомные Qt виджеты. | 1 | 9 | 28.02.23 | 03.03.23 | 4 | Базовый прототип GUI. | ✓ | V1.0.0  V1.0.1  V1.1.0 | 50% |
| 2 | 10 | 04.03.23 | 13.03.23 | 9 | Виджет ингредиента.  Виджет продукта.  Виджет рецепта.  Форма создания ингредиента.  Форма создания продукта.  Форма создания рецепта (❌). | ✓ ❌ |
| 5 | Базовой документация. | 1 | 11 | 15.03.23 | 18.04.23 | 35 | Написание базовой документации. | ✓ | - | 55% |

Обозначения результат спринта:

✓ – выполнен ❌ – провален ✓❌ – выполнен наполовину.

В задачах спринта пункт может быть отмечен (❌), это значит данный пункт не удалось реализовать по некоторых причинам.

ПРИЛОЖЕНИЕ Б

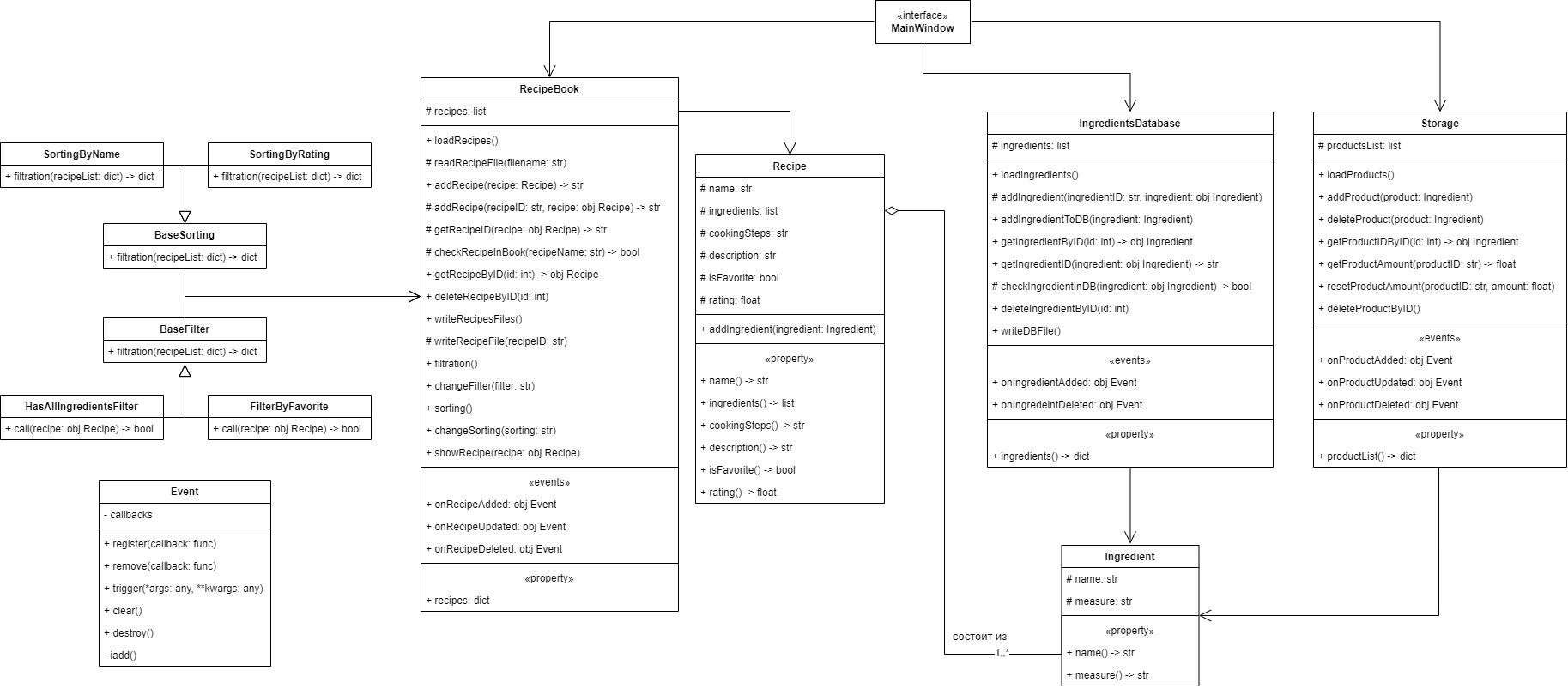


Рисунок 3 – «UML диаграмма архитектуры приложения»

ПРИЛОЖЕНИЕ В

UML (Unified Modeling Language) – унифицированный язык моделирования, система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.

Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем.

Словарь UML включает три вида строительных блоков:

* Диаграммы.
* Сущности.
* Связи.

Сущности – это абстракции, которые являются основными элементами модели, связи соединяют их между собой, а диаграммы группируют представляющие интерес наборы сущностей.

Диаграмма – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей).

Большинство элементов UML имеют уникальную и прямую графическую нотацию, которая дает визуальное представление наиболее важных аспектов элемента.Диаграммы классов оперируют тремя видами сущностей UML:

* Структурные.
* Поведенческие.
* Аннотирующие.

Структурные сущности – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является класс.

Поведенческие сущности – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение модели во времени и пространстве. Основной из них является взаимодействие – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели. Сообщение изображается в виде линии со стрелкой, почти всегда сопровождаемой именем операции.

Аннотирующие сущности – это поясняющие части UML-моделей, иными словами, комментарии, которые можно применить для описания, выделения и пояснения любого элемента модели. Главная из аннотирующих сущностей – примечание. Это символ, служащий для описания ограничений и комментариев, относящихся к элементу либо набору элементов. Графически представлен прямоугольником с загнутым углом; внутри помещается текстовый или графический комментарий.

Класс – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.

Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* Имя класса.
* Атрибуты (свойства) класса.
* Операции (методы) класса.

Для атрибутов и операций может быть указан один из трех типов видимости:

* — private (частный).
* # protected (защищенный).
* + public (общий).

Видимость для полей и методов указывается в виде левого символа в строке с именем соответствующего элемента.

Атрибут (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.

Операция (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты.

Графически операции представлены в нижнем блоке описания класса.

Отношения между классами:

* Зависимость.
* Ассоциация.
* Агрегация.
* Композиция.
* Обобщение.
* Реализация.

Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.

Зависимость – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой (смотреть Рисунок 4).



Рисунок 4 – «UML отношения. Зависимость.»

Зависимость – это связь использования, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют ее.

Ассоциация – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.

Двойные ассоциации представляются линией без стрелок на концах, соединяющей два классовых блока (смотреть Рисунок 5).

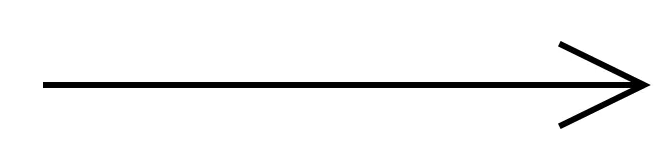


Рисунок 5 – «UML отношения. Ассоциация.»

Агрегация – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).

Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть» (смотреть Рисунок 6).



Рисунок 6 – «UML отношения. Агрегация.»

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком (смотреть Рисунок 7).



Рисунок 7 – «UML отношения. Композиция.»

Обобщение – выражает специализацию или наследование, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя.

Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя (смотреть Рисунок 8).



Рисунок 8 – «UML отношения. Обобщение.»

ПРИЛОЖЕНИЕ Г

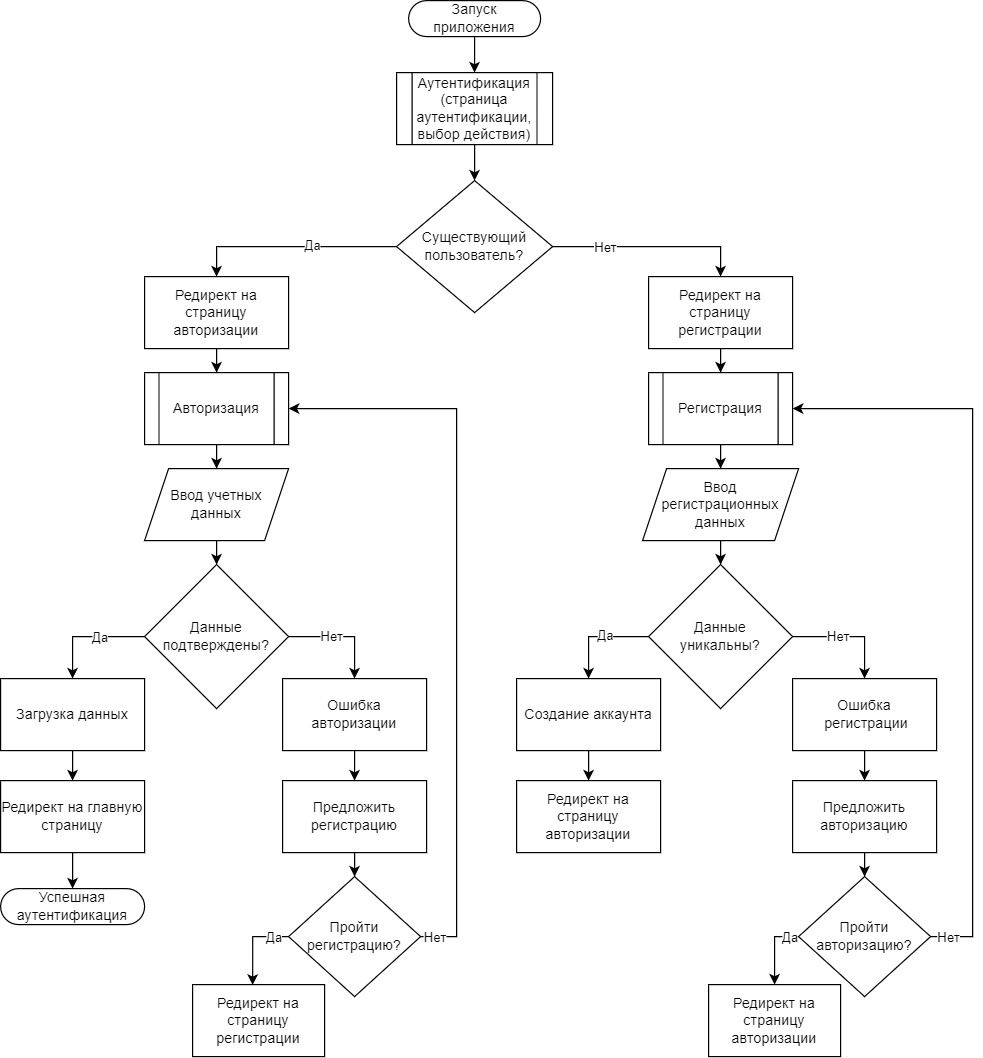


Рисунок 9 – «Блок-схема аутентификации»

ПРИЛОЖЕНИЕ Д



Рисунок 10 – «Блок-схема основного алгоритма с GUI»

ПРИЛОЖЕНИЕ Е



Рисунок 37 – «Виджет ингредиента»

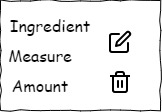


Рисунок 38 – «Виджет продукта»

ПРИЛОЖЕНИЕ Ё

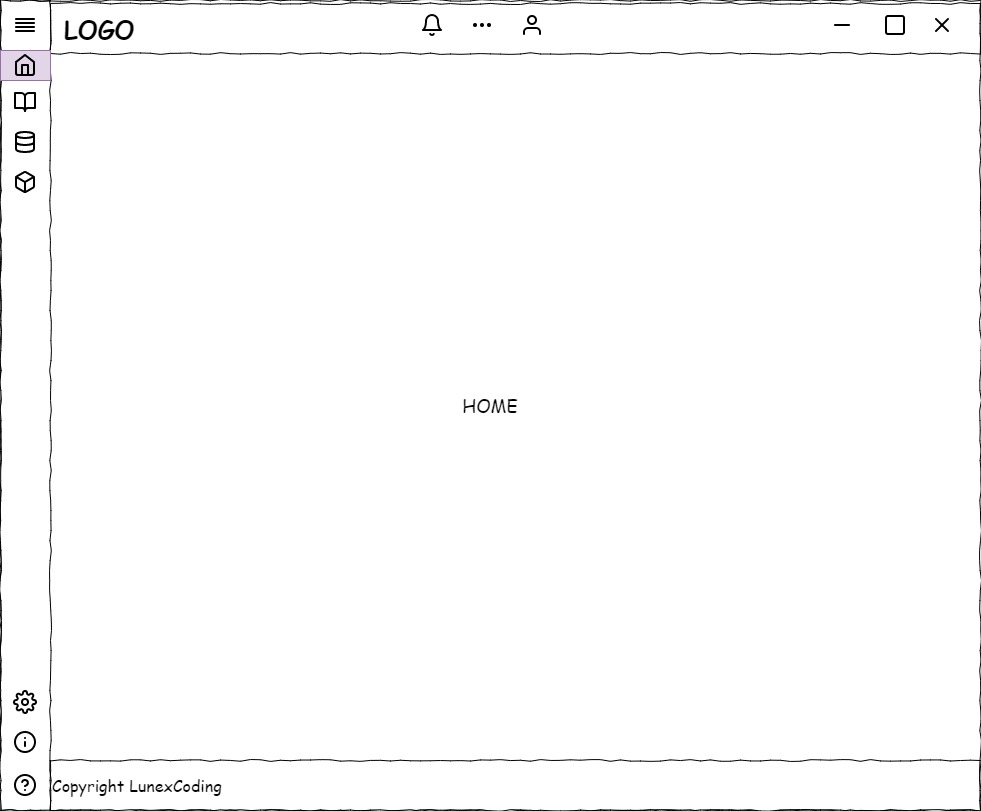


Рисунок 39 – «Главная страница»

ПРИЛОЖЕНИЕ Ж



Рисунок 40 – «Страница ингредиентов пустая»

ПРИЛОЖЕНИЕ З

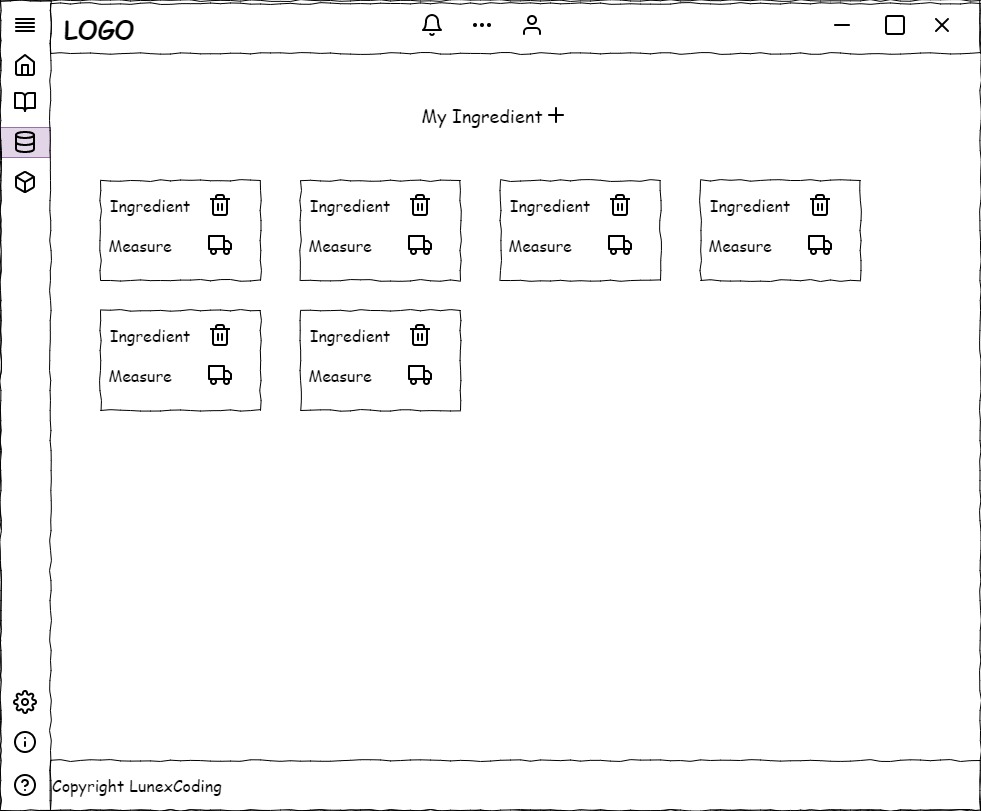


Рисунок 41 – «Страница ингредиентов наполненная»

ПРИЛОЖЕНИЕ И



Рисунок 42 – «Виджет продукта»

ПРИЛОЖЕНИЕ Й



Рисунок 43 – «Страница продуктов пустая»

ПРИЛОЖЕНИЕ К

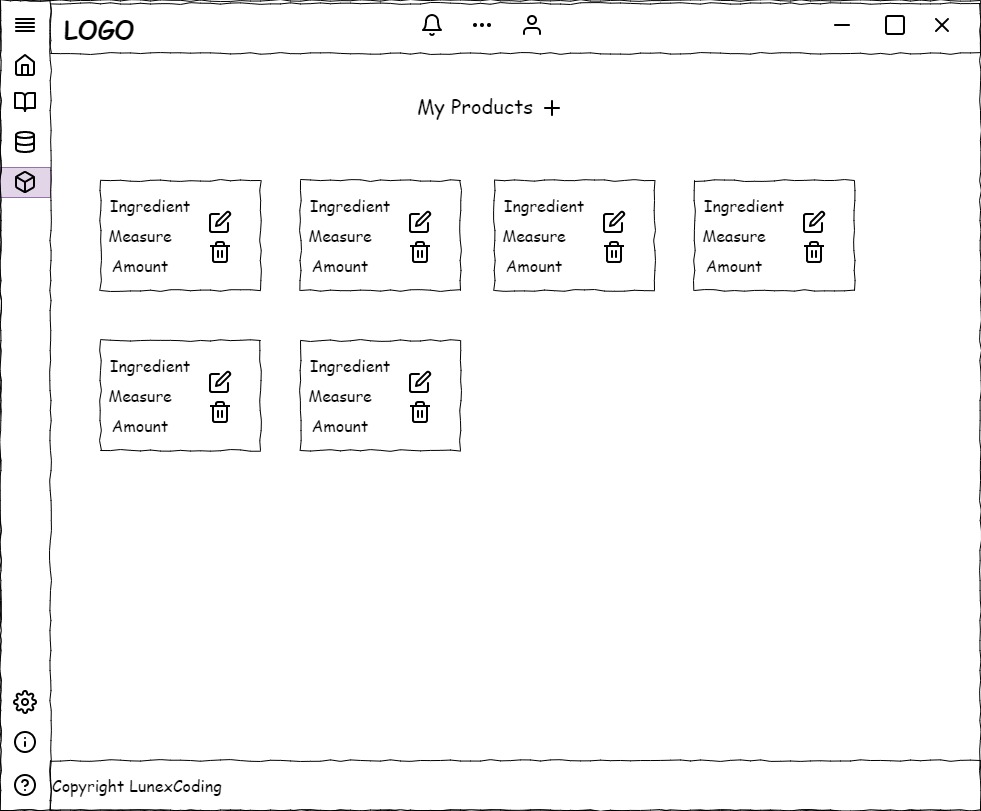


Рисунок 44 – «Страница продуктов наполненная»

ПРИЛОЖЕНИЕ Л

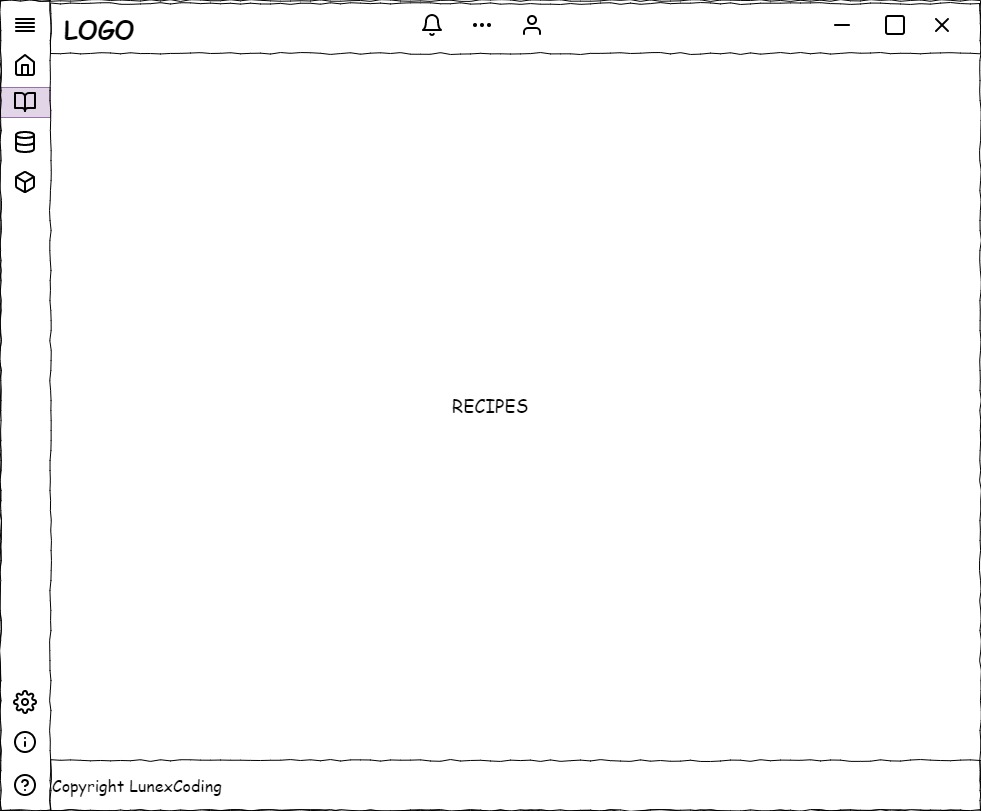


Рисунок 45 – «Страница рецептов»

ПРИЛОЖЕНИЕ М

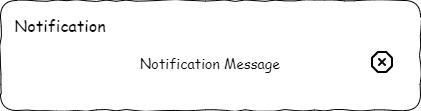


Рисунок 46 – «Уведомление»

ПРИЛОЖЕНИЕ Н

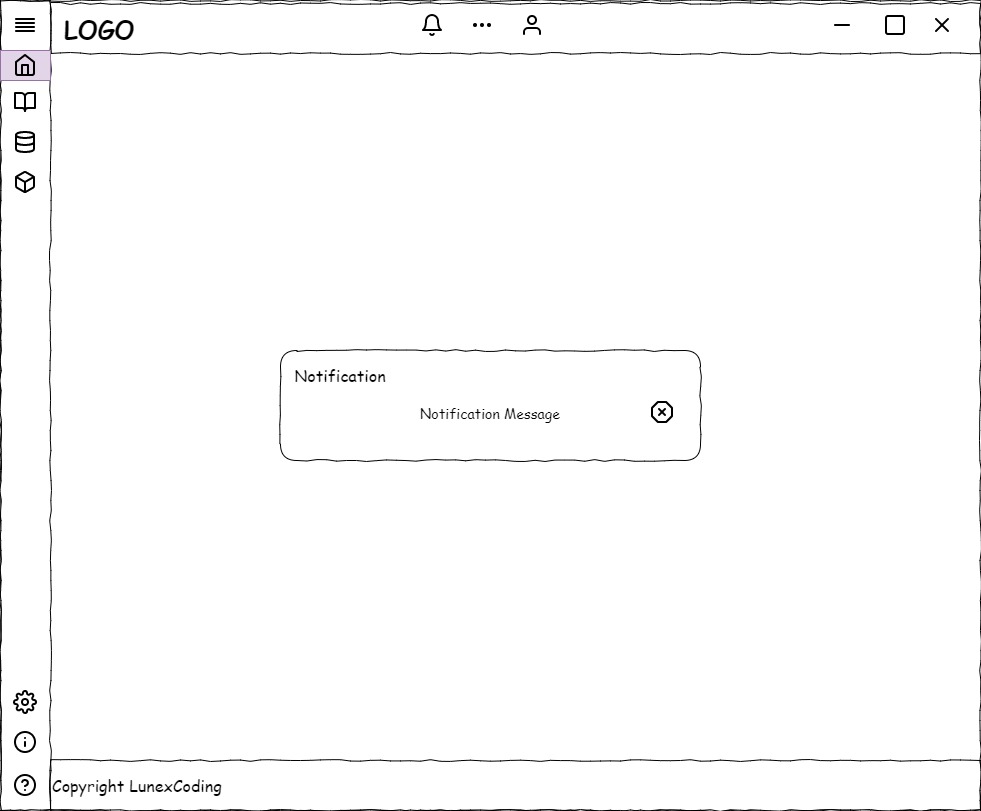


Рисунок 47 – «Уведомление на странице»

ПРИЛОЖЕНИЕ О

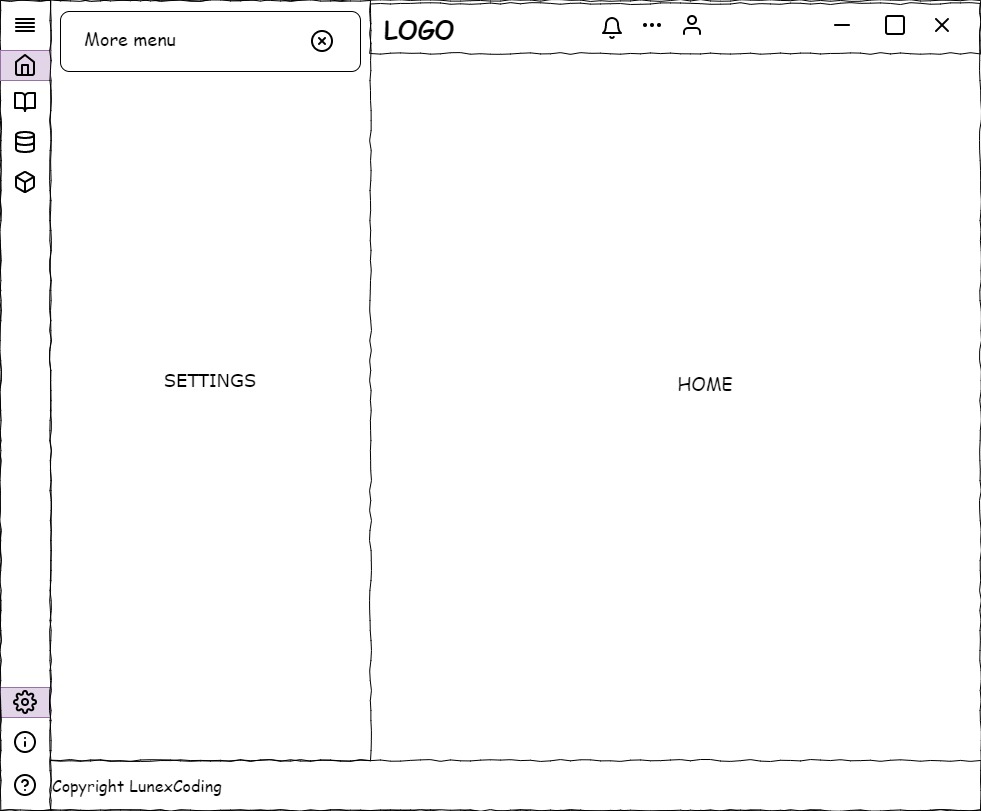


Рисунок 48 – «Раскрытое левое меню. Настройки»

ПРИЛОЖЕНИЕ П

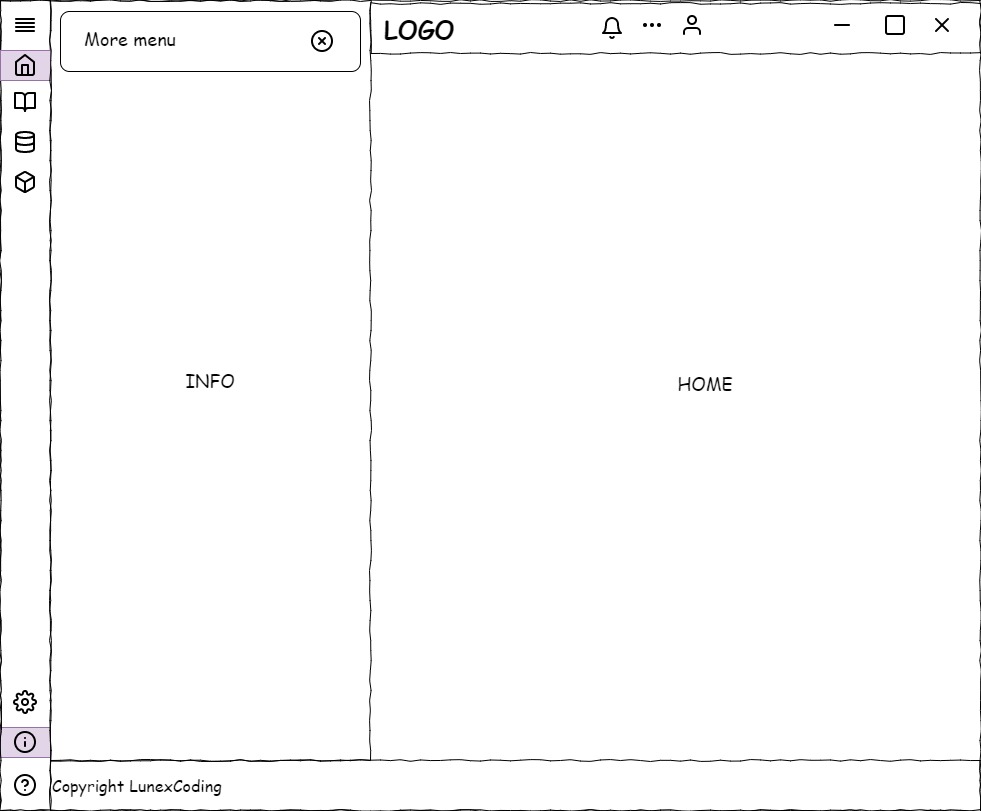


Рисунок 49 – «Раскрытое левое меню. Информация.»

ПРИЛОЖЕНИЕ Р

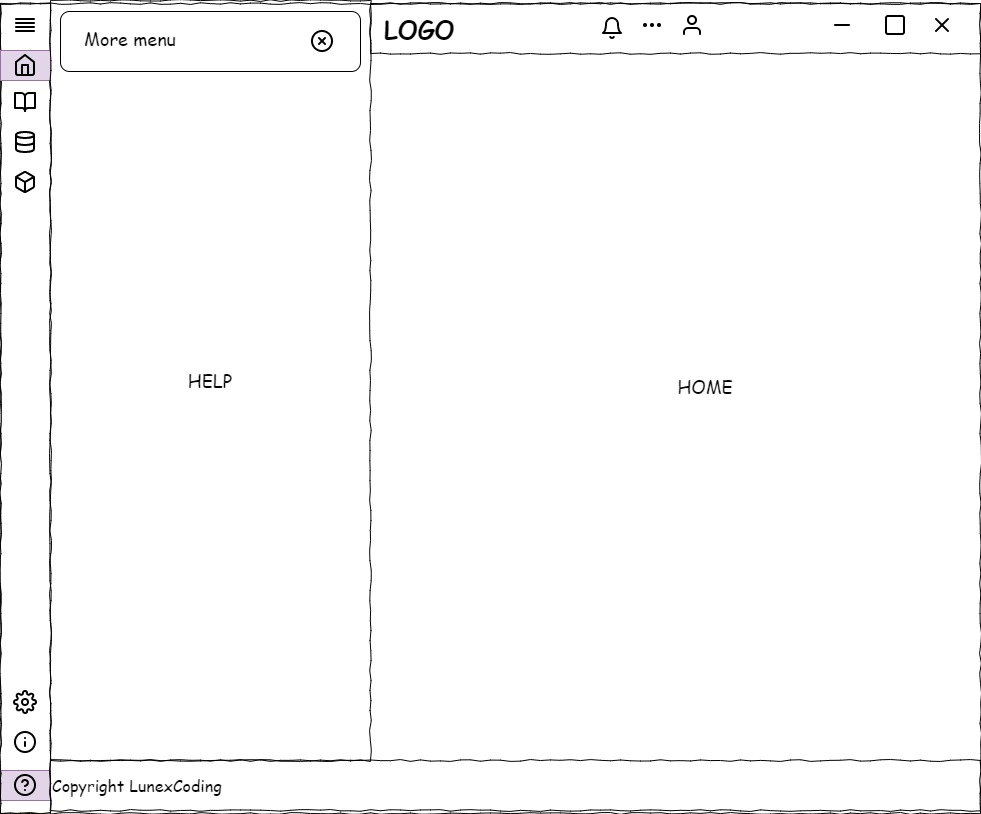


Рисунок 50 – «Раскрытое левое меню. Помощь.»

ПРИЛОЖЕНИЕ С

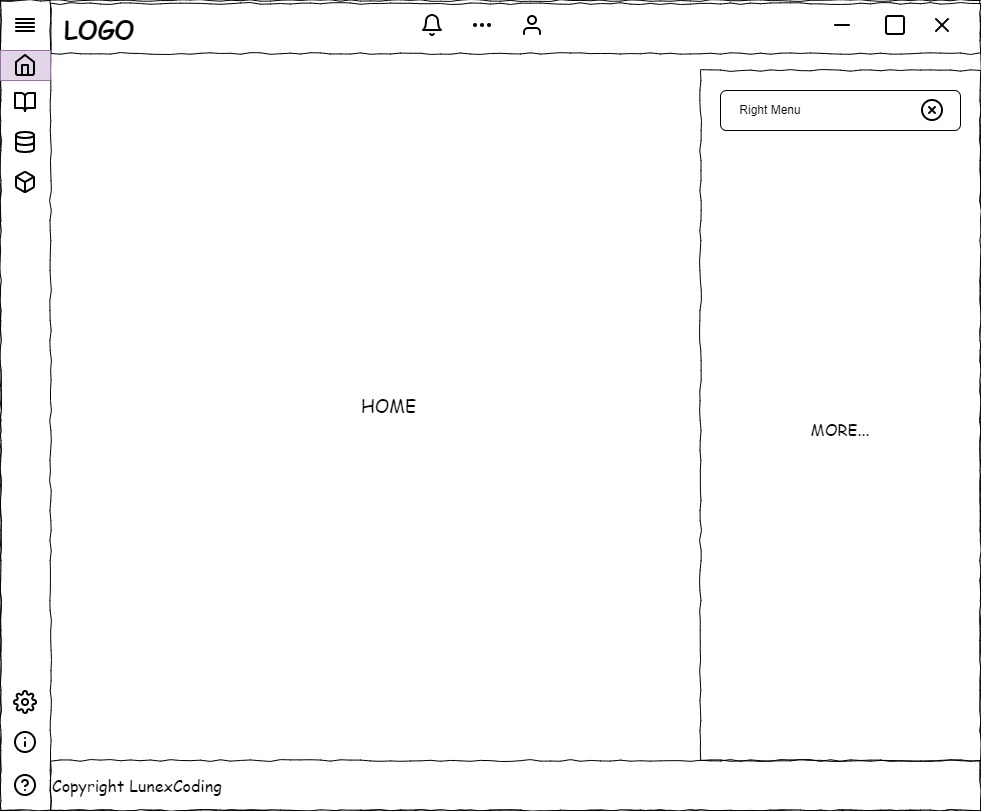


Рисунок 51 – «Раскрытое правое меню. Помощь.»

ПРИЛОЖЕНИЕ Т

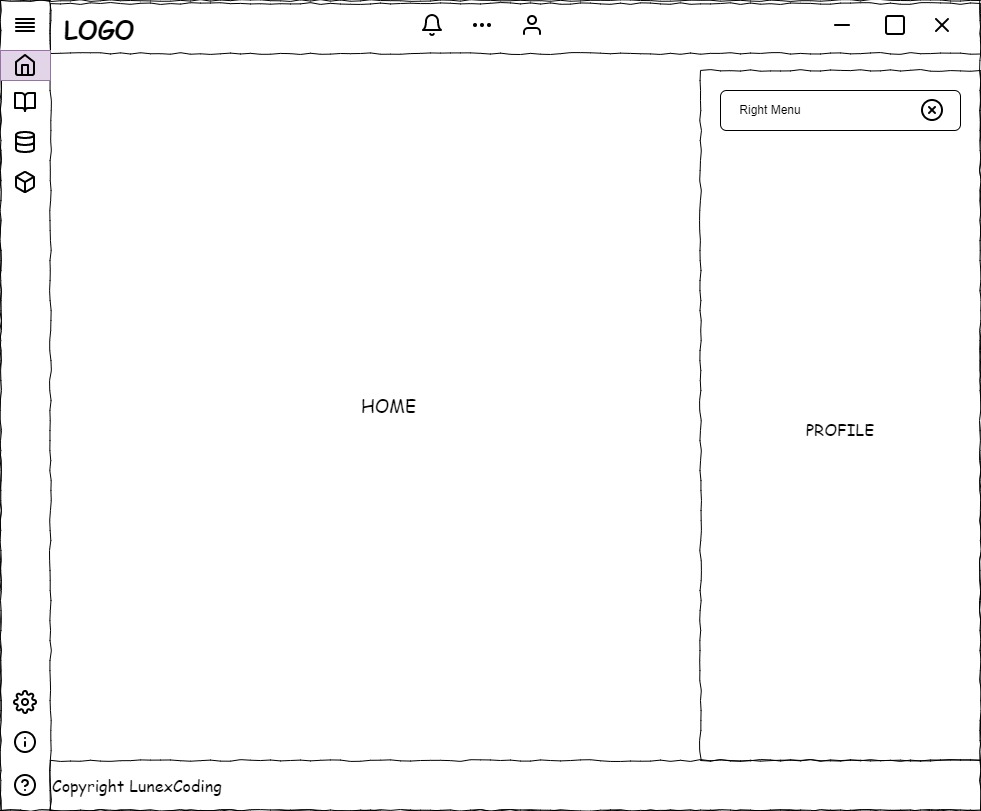


Рисунок 52 – «Раскрытое правое меню. Профиль.»

ПРИЛОЖЕНИЕ У

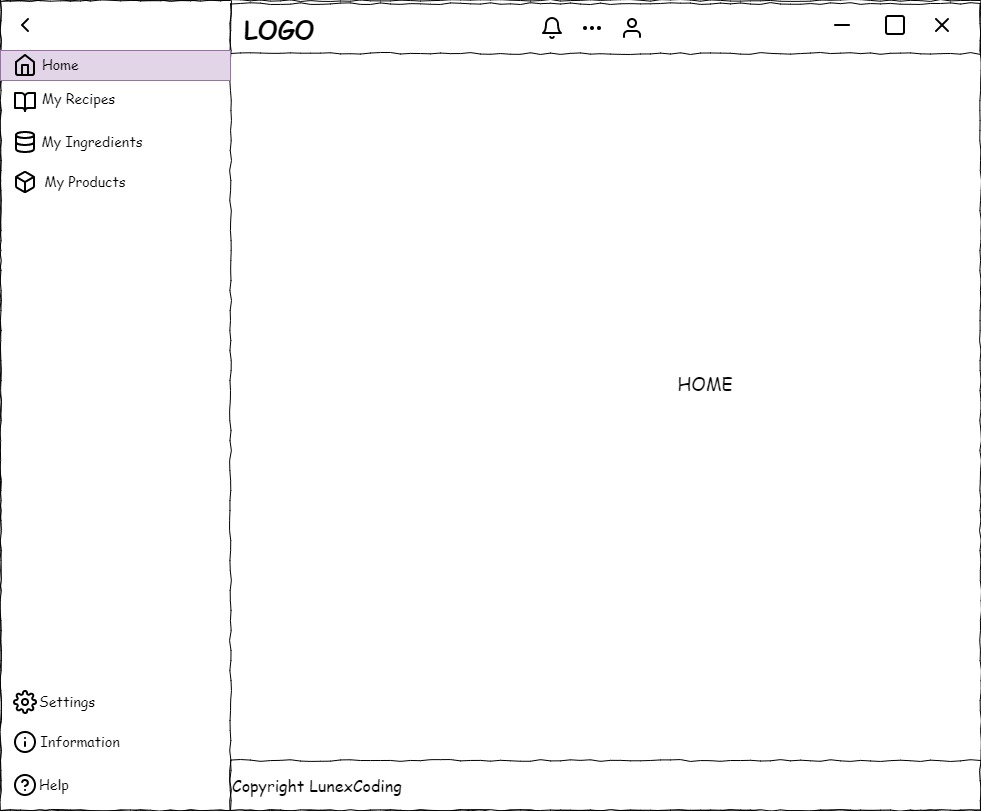


Рисунок 53 – «Раскрытая левая панель»

ПРИЛОЖЕНИЕ Ф

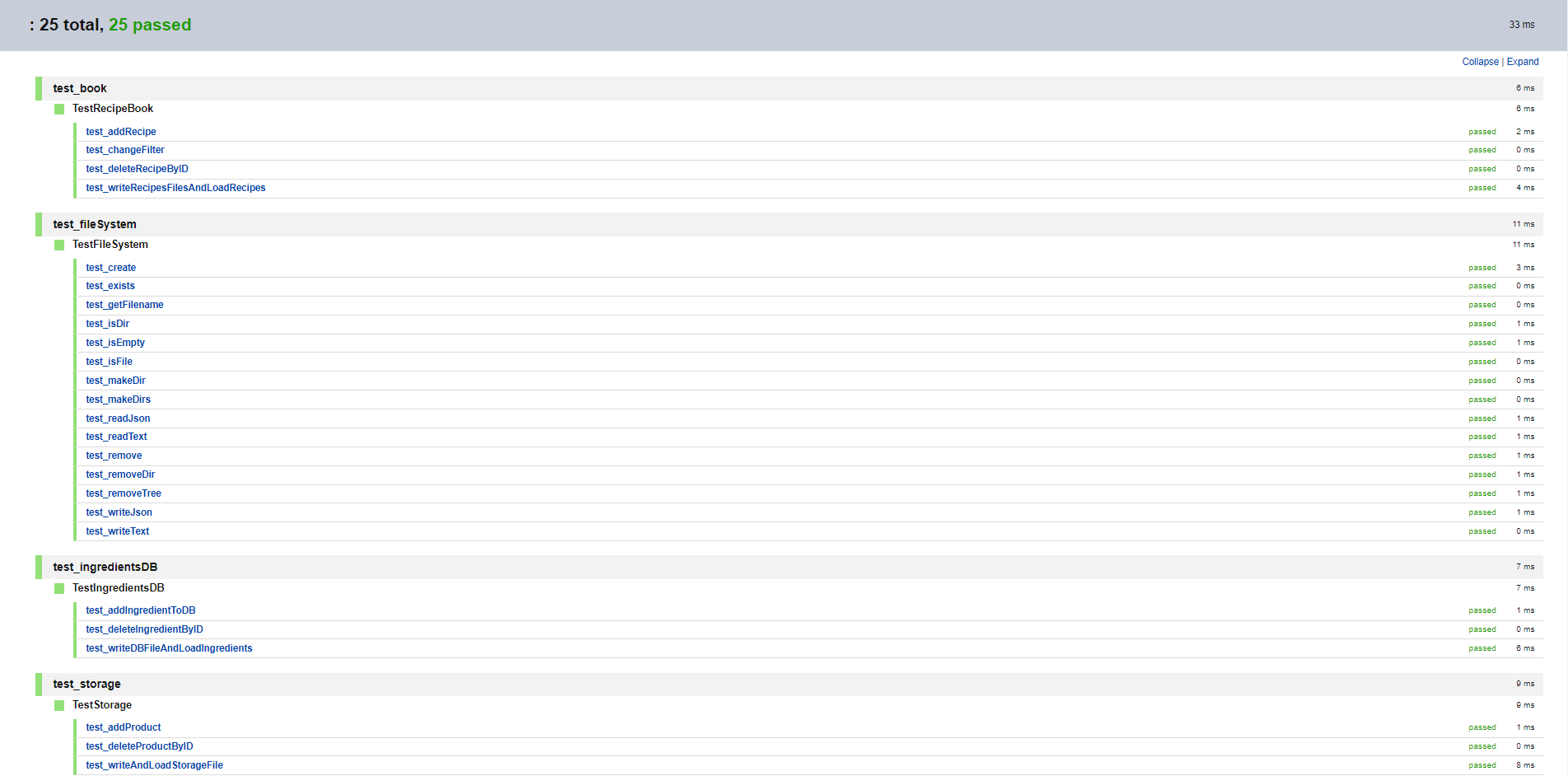


Рисунок 65 – «Отчет о тестировании»