**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**:

1. Разработать техническое задание;

2. Изучить предметную область;

3. Провести анализ возможностей разработки в секторе инструментальных прикладных приложений (в том числе анализ уже существующих разработок по теме);

4. Описать основные этапы проектирования и разработки приложения;

5. 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;
* Протокол ошибок и тестирование;
* 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)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № эпика | № Спринта | Пр. в днях | Дата начала | Дата окончания | Задачи | Результат спринта | Версии, релизы | Отметка готовности в % |
| 1 | 1 | 2 | 20.10.22 | 22.10.22 | Реализация минимального набора базовых модулей, реализация минимального функционала | ✓ | V0.0.1  V0.0.2 | 6% |
| 2 | 2 | 08.10.22 | 10.11.22 | Логирование, минимальная реализация модулей:  ingredientsDB  storage | ✓ |
| 2 | 3 | 6 | 12.11.22 | 18.11.22 | Проектирование.  Сохранение данных в файлы.  Тестирование минимального базового функционала. | ✓ | V0.1.0  V0.1.1 | 12% |
| 4 | 9 | 29.01.23 | 06.02.23 | Добавлены фильтры рецептов:  По имени;  По рейтингу;  По доступности;  По избранному.  Исправлены некоторые ошибки. | ✓ |
| 3 | 5 | 10 | 07.02.23 | 17.02.23 | Изменена структура некоторых сущностей.  Избежание дублирования объектов в базах.  Переработан модуль файловой системы. | ✓ | V0.2.0  V0.3.0  V0.3.1  V0.4.0  V0.4.1 | 37% |
| 6 | 3 | 18.03.23 | 21.02.23 | Дописывание тестов базового функционала. Переход с интеграционных тестов на модульные. | ✓ |
| 7 | 4 | 21.02.23 | 23.02.23 | Модуль файловой системы переписан согласно [документации](https://docs.pyfilesystem.org/en/latest/interface.html). | ✓ |
| 8 | 2 | 23.02.23 | 25.02.23 | Перевод генератора ID на технологию UUID. | ✓ |
| 9 | 1 | 26.02.23 | 26.02.23 | Исправлены некоторые ошибки. | ✓ |
| 5 | 10 | 14 | 28.02.23 | 13.03.23 | Базовый прототип GUI интерфейса | ✓, ❌ | V1.0.0 | 50% |
| 6 | 11 | 10 | 15.03.23 | 25.03.23 | Написание базовой документации | ✓ | - | 55% |

*Таблица 1 – Календарный план.*

Обозначения результат спринта:

✓ – выполнен

❌ – провален

✓, ❌ – выполнен наполовину

**1.1.8. Приложения**

* Инструкцию по инсталляции;
* Инструкция по эксплуатации;
* GitHub репозиторий с версионированием (релизами)

**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.

**1.2.4.1. Qt**

**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.

**1.2.4.2. unittest**

Фреймворк модульного тестирования.

Среда unittestмодульного 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.

**1.2.5.1. 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.

**1.2.5.2. PySide2**

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

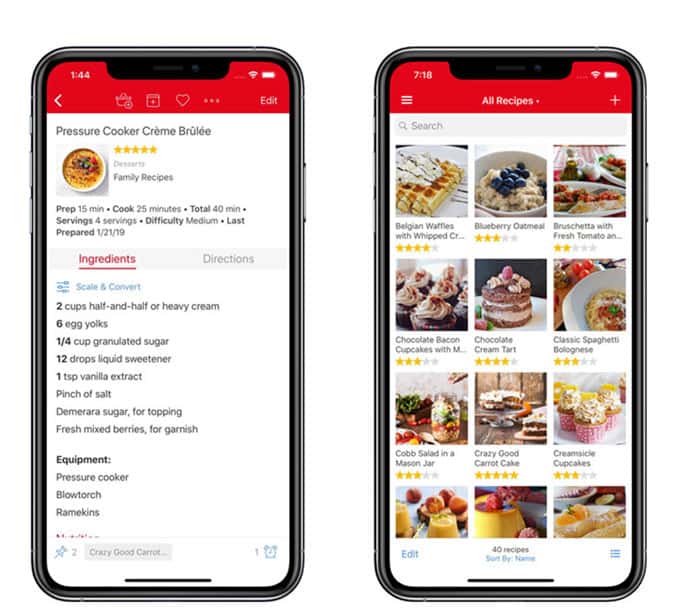
**1.3. Анализ существующих решений в предметной области**

**1.3.1. Paprika**

Паприка 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:

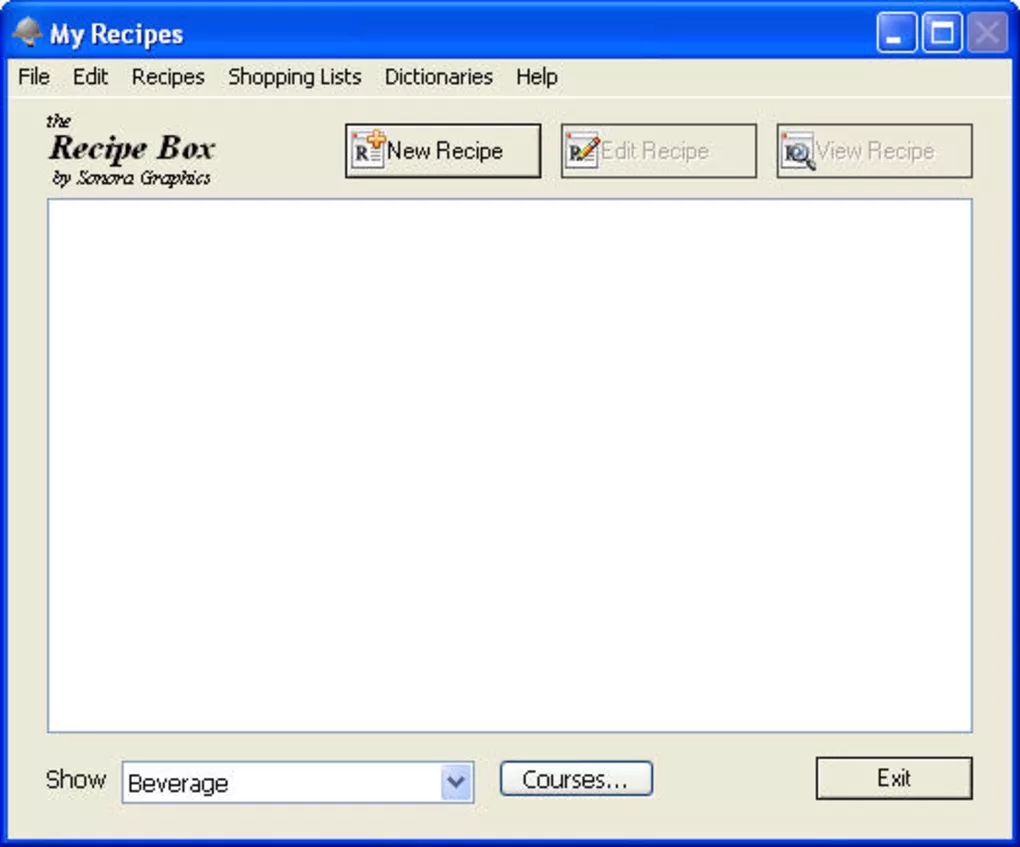
* Благодаря поиску очень удобно находить рецепты
* Браузер в приложении может анализировать рецепты практически с любого веб-сайта.

Недостатки:

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

**1.3.2. 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**

**2.1.1.1. 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.

**2.1.1.3. 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.

**2.1.1.4. Микросервисы**

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:

1. База данных ингредиентов:

1. Возможность вносить и удалять ингредиенты

2. Обеспечить уникальность добавленного ингредиента

3. Возможность сохранять и загружать состояние БД из файлов

2. База данных продуктов в наличии:

1. Возможность вносить и удалять продукты

2. Обеспечить уникальность добавленного продукта

3. Возможность сохранять и загружать состояние БД из файлов

3. База данных рецептов:

1. Возможность вносить и удалять рецепты

2. Обеспечить уникальность добавленного рецепта

3. Возможность сохранять и загружать состояние БД из файлов

4. Возможность фильтрации рецептов по параметрам:

1. Избранное

2. Наличие необходимых продуктов

5. Возможность сортировки рецептов по параметрам:

1. Рейтинг

2. Наименование

4. Логирование процесса выполнения программы.

5. Сущность или система для унифицированной, удобной работы с различными типами файлов.

6. Предусмотреть интерфейсы для дальнейшего подключения графических окон управления представленными программными модулями.

7. Предусмотреть интерфейсы для дальнейшего подключения интеграции FTP сервера для синхронизации данных на нескольких устройствах.

8. Предусмотреть интерфейсы для дальнейшего подключения интеграции с Telegram.

Смотреть. Приложение 1, Приложение 2.

**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.

**2.1.3.1. Порождающие паттерны проектирования**

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

**2.1.3.1.1. Singleton (Одиночка)**

**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.

**2.2. Хранение данных**

Все данные сущностей хранятся в файлах с расширением **json**.

**2.2.1. Структура хранения данных определенных сущностей**

**2.2.1.1. ingredientsDB**



*Рисунок 4 – ingredientsDB.json*

**2.2.1.2. storage**



*Рисунок 5 – storage.json*

**2.2.1.3. recipe**

Каждый рецепт представляет собой файл с **uuid** вместо названия с расширением json.

**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.



*Рисунок 6 – recipe.json*

**3. ПРАКТИЧЕСКАЯ ЧАСТЬ**

**Интерфейс командной строки CLI (Command-line interface)** — способ взаимодействия между человеком и компьютером путём отправки компьютеру команд, представляющих собой последовательность символов.

**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.

**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.

При разработке обязательно должна использоваться VCS. Выбор пал на **Git**.

При выбранной методологии **Scrum**, с максимальной продолжительностью спринта в **21 день**, следует четко определять задачи на спринт.

**3.1. Версионирование**

**Версионирование** — разработка и управление несколькими выпусками продукта, которые имеют тот же общий функционал, но усовершенствованы, модернизированы либо индивидуализированы. Коротко, версия говорит об изменении продукта.

**3.1.1. Требования к версионированию**

Все версии условно обозначаются: **V0.0.0**

Начальная версия: **V0.0.0**

Версии до V1.0.0 - относятся к CLI

Версии V1.0.0+ - относятся к GUI

**3.2. Написание кода**

**3.3. Пояснение кода**

**3.3.Тестирование**

**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]

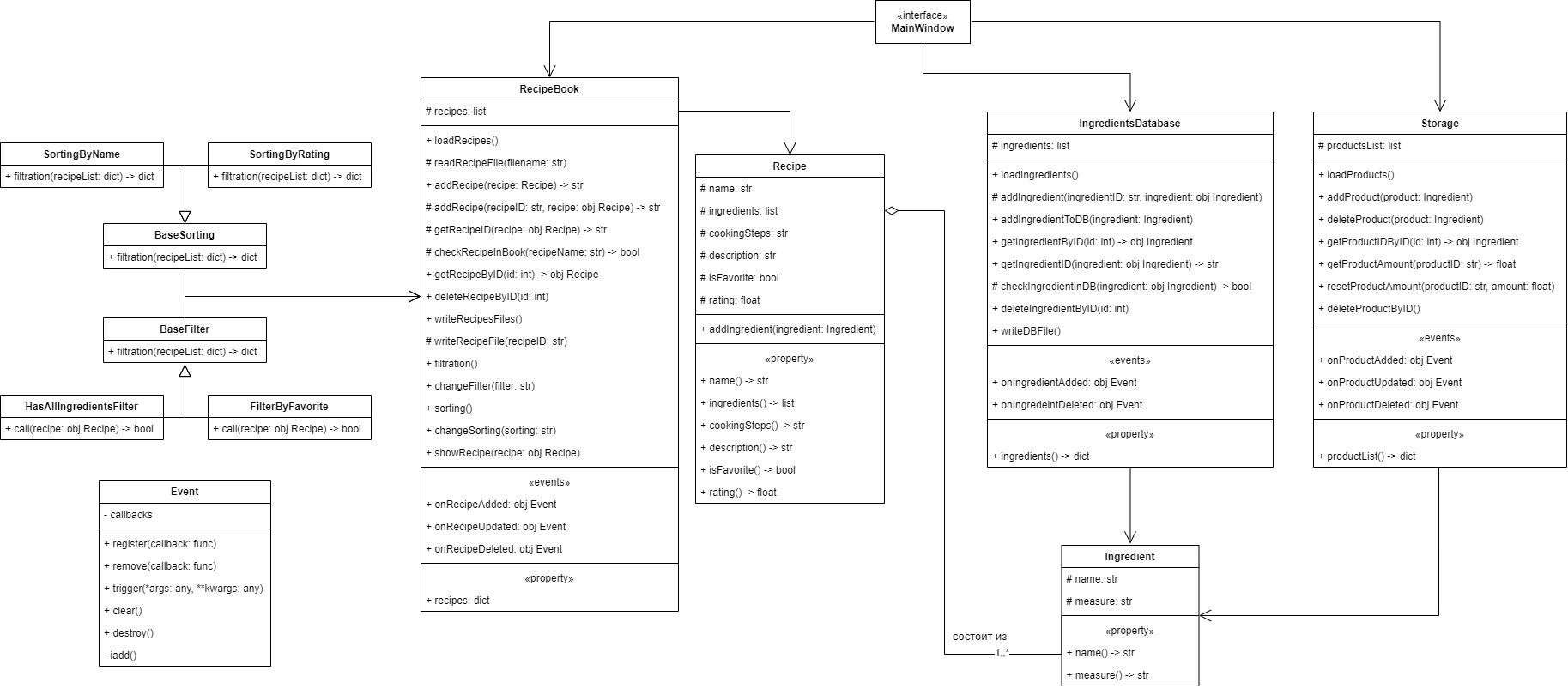
Шаблон проектирования. - [Электронный ресурс]: страница Wikipedia. - Режим доступа: https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD\_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F

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 1**



*Рисунок 7 – UML диаграмма архитектуры приложения*

**ПРИЛОЖЕНИЕ 2**

**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). Язык UML включает 13 видов диаграмм, среди которых на первом месте в списке — диаграмма классов.

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 8).



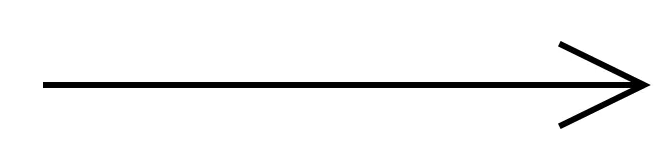
*Рисунок 8 – UML отношения. Зависимость.*

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 9).



*Рисунок 9 – 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 10).



*Рисунок 10 – 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 11).



*Рисунок 11 – 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 12).



*Рисунок 12 – UML отношения. Обобщение.*