«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)

Факультет И (шифр) Информационные и управляющие системы (наименование)

Кафедра И9 (шифр) Систем управления и компьютерных технологий (наименование)

Дисциплина Технологии программирования

КУРСОВАЯ РАБОТА
на тему

Разработка программно-технологического компонента системы с динамически-конфигурируемой функциональностью поддерживающей повторное использование существующих решений

Выполнил студент группы И9М33

Крылов К.А.
Фамилия И.О.

РУКОВОДИТЕЛЬ

Арсеньев Б.П. _____
Фамилия И.О. Подпись

Оценка _____

«_____» _____ 2018 г.

САНКТ-ПЕТЕРБУРГ
2018 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В индустрии ИТ можно выделить тенденцию, связанную с организацией процессов разработки программного обеспечения, которая направлена на повышение эффективности работы программистов. Это можно объяснить тем, что индустрия стремится успевать за возрастающим спросом на новые ИТ решения во всех сферах жизни.

В рамках данной работы будут рассмотрены две проблемы классического подхода к разработке программного продукта, а также предложен вариант программного продукта, который обеспечивает их решение.

Первую можно охарактеризовать, как необходимость каждый раз заново решать незначительно отличающиеся друг от друга задачи. Несмотря на то, что в некоторых случаях можно воспользоваться готовыми решениями, зачастую такой подход отвергается ввиду следующих причин:

- сторонние разработчики могут недостаточно сопровождать своё решение;
- интеграция готового решения с существующим может вызвать большие затруднения, чем написание его собственноручно;
- если существует большое множество подходящих решений, то выбор среди них наилучшего может быть весьма затруднительным [1].

Даже принимая во внимание то, что существующие средства разработки позволяют быстро разрабатывать программные продукты “с нуля”, тем не менее, они не предоставляют уже готовых компонентов, которые относятся к некоторой предметной области и выполняют реальные задачи. В большинстве случаев разработчики вынуждены использовать некоторые абстрактные компоненты, определять отношения между ними, описывать бизнес-логику целевой предметной области, разрабатывать интерфейс взаимодействия с пользователем и прочие ключевые задачи разработки программного продукта.

Второй существенной проблемой является неспособность программного продукта в полной мере удовлетворять потребности большого множества пользователей с различающимися потребностями. Поскольку конкретные цели пользователя в некоторый момент времени не известны заранее и определяются, для непроизводственных систем, в первую очередь личностью пользователя, то разработать программу, в которой были бы заранее учтены все возможные варианты её взаимодействия с пользователем, не представляется возможным [2].

Таким образом, решением озвученных проблем можно считать программный продукт, в котором пользователю будет предоставлена возможность самому “собрать” своё “личное” приложение, которое будет максимально полно отвечать его желаниям и потребностям. Процесс сборки такого приложения будет заключаться в отбирании желаемых функций приложения и аспектов пользовательского интерфейса, которые будут представлены соответствующими программными модулями. К тому же, если должным образом организовать процесс разработки подобного продукта, то общая трудоёмкость существенно сократится, так как реализованные функции не потеряют своего потенциала, а лишь будут заключены в удобные для использования другими разработчиками программные модули [3].

Целью курсовой работы является разработка программно-технологического компонента системы с возможностью динамической конфигурации функций, предоставляемы приложением, а также поддерживающей повторное использование программных решений посредством использования модульного подхода.

1 ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Для реализации программного продукта требуется составить техническое задание, которое наиболее полно раскрывает, какие функции должен выполнять программный продукт, описывает средства и методы реализации, а также устанавливает сроки выполнения работ.

Наименования проводимых работ и результат работ приведены в Таблице 1.

Таблица 1 – Проводимые по техническому заданию работы.

п/п	Название проводимых работ	Результат проводимых работ
1	Исследование предметной области, связанной с разработкой систем, обеспечивающих динамическое конфигурирование функциональности, а также определение возможных способов поддержки повторного использования существующих решений	Описание предметной области, достаточное для обоснования проектных решений из п/п 2
2	Разработка архитектуры программно-технологического компонента системы	Описание архитектуры программно-технологического компонента, достаточное для выполнения п/п 3, 4 и 5
3	Выбор и обоснование инструментов разработки	Приведение возможных вариантов выбора инструментов, сравнение их между собой, обоснование выбора

4	Разработка набора базовых компонентов и инструментов разработки, которые впоследствии будут служить для создания компонентов, которые в своей совокупности будут предоставлять динамически-конфигурируемую функциональность системы	Набор базовых компонентов и инструментов разработки для компонентов, которые в своей совокупности будут составлять динамически-конфигурируемую функциональность системы
5	Разработка компонентов, предоставляющих динамически-конфигурируемую функциональность системы, используя набор базовых компонентов и инструментов разработки, которые являются результатом выполнения п/п 4	Компоненты, соответствующие функциональными требованиям к пользовательским компонентам, разработанные с использованием набора базовых компонентов и инструментов разработки, которые являются результатом выполнения п/п 4

Требования к разрабатываемому программно-технологическому компоненту системы можно разделить на две категории:

1. Требования к компонентам, предоставляющих динамически-конфигурируемую функциональность системы, используемым конечным пользователем (пользовательским компонентам).

2. Требования к компонентам, используемым разработчиками пользовательских компонентов.

Функциональные требования к базовым компонентам и инструментам разработки, используемым разработчиками пользовательских компонентов:

1. Процесс разработки пользовательских компонентов с использованием базовых компонентов и инструментов разработки должен быть построен таким образом, чтобы поддерживать возможность повторного использования уже разработанных ранее пользовательских компонентов.

Нефункциональные требования к пользовательским компонентам:

1. Кроссплатформенность (Windows, MacOS, Linux).

Функциональные требования к пользовательским компонентам:

1. Планирование задач (добавление, редактирование, удаление и хранение данных, которые содержат описание личных задач пользователя).

2. Изменение состава атрибутов, описывающих личные задачи пользователя, в зависимости от текущего состава используемых пользовательских компонентов. Например, если в текущем составе содержатся компоненты, отвечающие за функции описания задачи, планирования времени выполнения и количества затраченного на её выполнение времени, то задача характеризуется атрибутами “Название”, “Время выполнения”, “Количество затраченного времени”. При изменении указанного состава пользовательских компонентов состав атрибутов должен также изменяться.

Нефункциональные требования к пользовательским компонентам:

1. Кроссплатформенность (Windows, macOS, Linux, Android, iOS).

2 ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

В ходе исследования предметной области необходимо определить способы обеспечения динамического конфигурирования функциональности, а также определить возможные способы поддержки повторного использования существующих решений.

Был определён набор связанных между собой областей, рассмотрение которых позволит наиболее полно раскрыть особенности решаемой задачи.

2.1 Повторное использование программного обеспечения

Для решения задач, возникающих в различных предметных областях, разработчики создают новое программное обеспечение (ПО), и если в дальнейшем кто-то столкнётся с подобной задачей, то он сможет воспользоваться уже существующим решением. Для описания такого процесса используется понятие повторного использования ПО.

Подходя к классификации существующих методов повторного использования программного обеспечения, следует отметить тот факт, что чёткого разделения между ними не существует – возможно использование их различных комбинаций в рамках одного проекта. Некоторые из них непосредственно предоставляют функции уже готовых систем, некоторые лишь позволяют заложить в текущую систему возможности повторного использования в дальнейшем, однако все они, так или иначе, обеспечивают возможности повторного использования программного обеспечения.

Можно разделить методы повторного использования в соответствии с этапами разработки, во время которых они могут привнести максимальный вклад.

1. Определение требований: анализ предметной области;
2. Проектирование: принципы проектирования, использование готовых архитектурных решений (паттернов проектирования);

3. Реализация: сниппеты, принципы разработки, генераторы исходного кода, использование стороннего кода на этапе разработки (статические библиотеки, фрагменты исходного кода);

4. Внедрение и поддержка: использование стороннего кода на этапе исполнения (динамические библиотеки, транслируемые языки), использование функций сторонних сервисов [1].

Применительно к задачам, поставленным в текущей работе, необходимо отдельно рассмотреть методы повторного использования, которые относятся к этапу внедрения и поддержки.

Использование стороннего кода на этапе исполнения, за счёт использования динамических библиотек или программного кода, написанного на транслируемом языке, делает возможным применение модульного подхода к организации структуры программы. При этом подходе программа разбивается на относительно независимые составные части – программные модули. При этом каждый модуль может разрабатываться, программироваться, транслироваться и тестироваться независимо от других [4].

Использование модульного подхода в текущем проекте позволит обеспечить динамическое конфигурирование функциональности, при условии предоставления возможности изменять состав подключенных модулей во время работы основной программы. Таким образом, определившись с одним из требований, следует рассмотреть возможность поддержки повторного использования существующих решений.

Повторное использование позволяет значительно сократить длительность разработки, более точно оценить сроки разработки, повысить качество конечного продукта за счёт использования уже отлаженных и проверенных компонентов. Однако у данного подхода существуют и недостатки.

1. Сопровождение – сторонние разработчики могут сопровождать ПО недостаточной документацией, а также не исправлять выявленные в процессе его работы ошибки;

2. Интеграция – внедрение выбранного ПО с другим решением, которое может также использоваться для решения поставленной задачи, может вызвать значительные затруднения;

3. Выбор – если существует большое множество доступных решений, которые подходят под условия поставленной задачи, то выбор среди них наилучшего может быть весьма затруднительным.

Один из наиболее эффективных подходов к повторному использованию базируется на понятии семейства программного обеспечения. Семейство программного обеспечения – это серия программных решений предназначенных для применения в некоторой предметной области, процесс разработки которых основан на использовании разделяемых базовых компонентов предопределённым способом. Вместе с тем все программные решения одной серии различны. Каждый раз при создании нового решения повторно используется общее множество разделяемых базовых компонентов, которые в совокупности формируют технологическую платформу программного обеспечения. Далее в процессе разработки создается несколько дополнительных компонентов, а некоторые компоненты адаптируются согласно новым требованиям [5, 6, 7].

В рамках решаемой задачи использование подхода семейства программного обеспечения полностью оправдывает себя, так как требование поддержки повторного использования существующих решений является его неотъемлемой частью. Определив возможные пути удовлетворения обоих требований, рассмотрим вопросы, связанные с организацией процесса разработки.

Обычно, разработка семейства программного обеспечения ведётся внутри компании, с привлечением только собственных сотрудников. Однако компания может сделать выбор в пользу раскрытия своей технологической платформы и предоставления открытого доступа к ней. Произойти это может от осознания факта, что для обеспечения того количества функциональности, которое необходимо предоставить пользователям чтобы удовлетворить их потребности, необходимо затратить неприемлемое количество ресурсов. Как только компания делает такой выбор, она совершает переход от семейства программного обеспечения к экосистеме программного обеспечения [7].

2.2 Экосистемы программного обеспечения

Экосистема программного обеспечения – такая организация процесса разработки программного обеспечения, при которой его независимые друг от друга участники, совместно используя какую-либо технологическую платформу, вступают в симбиотические отношения, характерные для биологических экосистем [8, 9].

Для биологических экосистем характерны следующие симбиотические отношения:

1. Мутуализм – два участника получают взаимную выгоду от взаимодействия друг с другом.
2. Комменсализм – один участник получает выгоду от взаимодействия, второй не получает ничего.
3. Антагонизм – два участника конкурируют за общие ресурсы и выгода одного, исключает выгоду другого.
4. Паразитизм – один участник получает выгоду от взаимодействия, второй получает ущерб.
5. Аменсализм – один участник получает ущерб от взаимодействия, второй не получает ничего.

6. Нейтрализм – два участника не получают ничего от взаимодействия.

В экосистемах ПО наиболее распространёнными отношениями являются мутуализм, комменсализм и антагонизм [9]. Выгода при взаимодействии может быть как коммерческой (продажи, отчисления), так и некоммерческой (известность, опыт, идеология).

Участники процесса могут занимать следующие роли [7, 10]:

1. Владелец платформы – организация, которая несёт ответственность за развитие экосистемы программного обеспечения, оценивает её состояние и принимает организационные решения.

2. Внутренний разработчик – коллектив является частью организации, которая предоставляет технологическую платформу. Имеет доступ к технологической платформе и занимается её непосредственным развитием.

3. Стратегический партнёр – сторонние организации, которые связаны долговременными отношениями с владельцем платформы. Разрабатывают дополнительные компоненты для своих нужд или нужд владельца платформы. Владелец платформы может предоставить возможность вносить изменения в технологическую платформу. Владелец платформы может напрямую повлиять на их деятельность и предсказать их поведение.

4. Сторонний разработчик – сторонние разработчики или организации, никак не связанные с владельцем платформы. Разрабатывают дополнительные компоненты для собственных нужд, с целью получения личной выгоды. Владелец платформы не имеет прямого влияния на них, а также не может достоверно предсказать их поведение. Способны обеспечить весьма существенный толчок в развитии экосистемы.

5. Пользователь – лицо или организация, пользующаяся программным обеспечением, которое было получено в результате взаимодействия прочих участников. Является участником, который, хоть и косвенно, но оказывает

самое существенное влияние на развитие экосистемы программного обеспечения.

Рассмотрение экосистем программного обеспечения можно разделить на три ключевых направления: программная инженерия, бизнес и менеджмент, взаимодействие участников. Так как в данной работе производится разработка программно-технологического компонента, то рассмотрение вопросов, касающихся программной инженерии является наиболее важным.

Далее приведены требования к организации процесса разработки в экосистеме программного обеспечения:

1. Использование интерфейсов позволяет сторонним разработчикам использовать возможности технологической платформы, не вдаваясь в особенности её реализации. Стабильность и простота интерфейсов платформы являются ключевыми факторами для дальнейшего развития экосистемы программного обеспечения.

2. Внесение изменений в существующий интерфейс какого-либо компонента может, в свою очередь, привести к несоответствию со всеми зависимыми от него компонентами.

3. Централизация управления не эффективна при широкомасштабной разработке, вместо этого следует использовать децентрализованный подход, в котором основным управляющим механизмом является архитектура экосистемы программного обеспечения.

4. Непрерывная разработка множеством участников требует внесения изменений в организацию процесса разработки: направленность на доступную и безопасную интеграцию компонентов, независимое внедрение новых версий компонентов, замедление темпа выпуска новых версий, чтобы сторонние разработчики успевали подстраиваться под вносимые изменения.

Приведённое в данном разделе описание аспектов предметной области достаточно для обоснования проектных решений в дальнейшем.

3 РАЗРАБОТКА АРХИТЕКТУРЫ

При разработке архитектуры следует учесть описанные в разделе исследования предметной области, требования к организации разработки экосистем программного обеспечения. На их основании можно определить ключевые требования, которые должны быть предъявлены к разрабатываемой архитектуре:

1. Децентрализованность принятия проектных решений.
2. Ограничение распределения ответственности за единичные проектные решения.
3. Обеспечение процесса непрерывной разработки компонентов требует предоставления его независимым друг от друга участникам доступа к компонентам, которые входят в состав экосистемы программного обеспечения. Это свойство особо важно для обеспечения интеграции между компонентами, разработка которых ведётся разными участниками.
4. Обеспечение повторного использования функций существующих решений посредством предоставления простых и доступных интерфейсов.
5. Наличие механизмов разрешения конфликтов совместимости между компонентами. Конфликт может возникнуть при изменении интерфейса такого компонента, функции которого уже используются зависящими от него компонентами.

Так как разработка архитектуры является проектным решением, то, в соответствии с указанными требованиями, ответственность за него не должна влиять на развитие всего проекта в целом. Следовательно, архитектура должна обладать значительной гибкостью.

В соответствии с таким требованием к организации процесса разработки, что стабильность и простота интерфейсов платформы являются ключевыми факторами для дальнейшего развития экосистемы программного обеспечения, следует также отметить, что архитектурные требования к компонентам не

должны значительно усложнять их реализацию и доступ к возможностям технологической платформы.

Было принято решение использовать при разработке приложения архитектуру подключаемых программных модулей, каждый из которых будет представлять компонент системы, отвечающий за набор своих функций, а задачи по организации их совместного взаимодействия возложить на ответственный за это компонент-агрегатор.

Для того чтобы обеспечить возможность расширения функций программы при включении в систему новых компонентов, необходимо определить некий набор интерфейсов, через которые между ними будет осуществляться взаимодействие. Однако если определить все эти интерфейсы не в подключаемом модуле, то использовать другой набор интерфейсов в ней будет невозможно – возникнет необходимость написать ещё одну программу. Поэтому было принято решение вынести компонент-агрегатор в отдельный модуль, а в основной программе оставить только один интерфейс, общий для всех модулей, реализующих функции компонента-агрегатора.

На основании рассуждений приведённых выше, было принято решение представить компонента-агрегатора интерфейсом RootModel (рисунок 1).

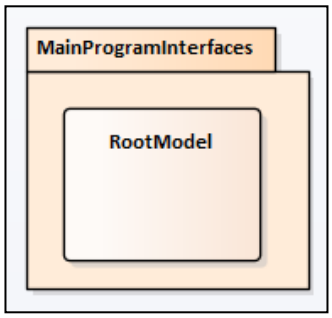

Рисунок 1 – Диаграмма классов.

Интерфейс, определённый в основной программе

Через него основное приложение передаёт ссылки на все загруженные модули в компонент-агрегатор, а он, в свою очередь, выбирает из них те, которые ей поддерживаются. Далее он производит их инициализацию и

связывание между собой. Процесс загрузки и инициализации модулей отражён на рисунке 2.

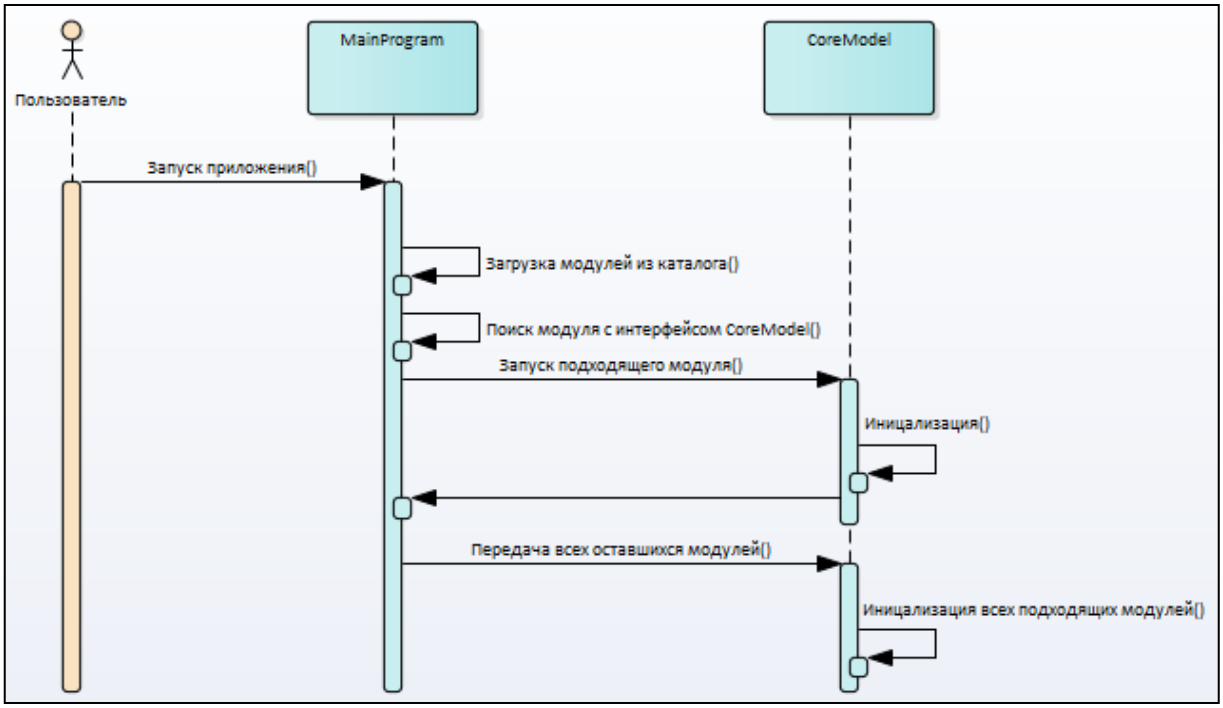

Рисунок 2 – Диаграмма последовательности.

Процесс подключения модулей

Определив отношение между основной программой и компонентом-агрегатором, можно приступить к архитектуре, которую компонент-агрегатор будет предоставлять и поддерживать. Следует начать с того, что в соответствии с требованием кроссплатформенности программы не должно существовать никаких ограничений в отношении:

– используемого источника данных;
– функций приложения;
– пользовательского интерфейса приложения.

Для решения этой задачи было принято решение разработать архитектуру, в соответствии с которой каждой из перечисленных областей будет соответствовать свой тип модуля, реализующий определённый интерфейс.

Таким образом, был разработан набор интерфейсов, которые будут реализованы в модулях. Разработанные интерфейсы представлены на рисунке 3.

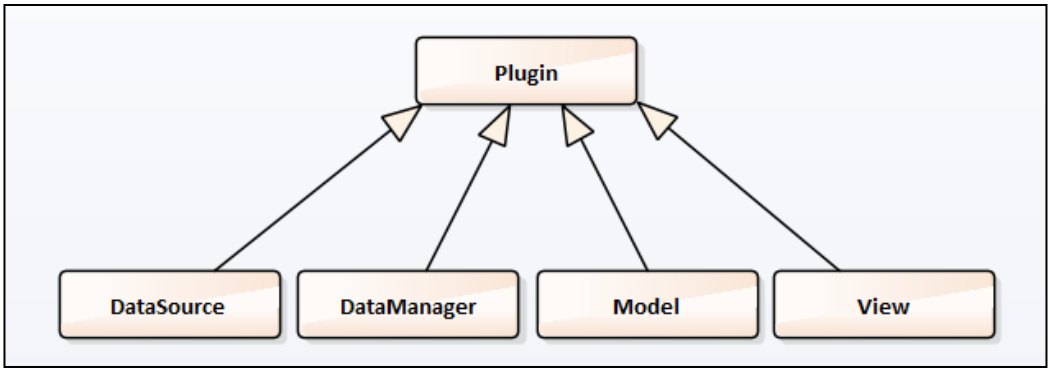

Рисунок 3 – Диаграмма классов.

Наследование интерфейсов

Далее идёт описание разработанных интерфейсов:

Plugin

– базовый интерфейс для интерфейсов DataSource, DataManager, Model, View;

– содержит набор методов, общих для всех модулей;

– является абстрактным и не представляет ни одного модуля в системе.

DataSource

– не связан с другими модулями;

– обеспечивает подключение к источнику данных (база данных, сеть и т.д.) и предоставляет набор базовых взаимодействий с ним.

DataManager

– должен быть связан с DataSource;

– использует методы DataSource зависящие от источника данных (SQL-запросы, интернет-запросы и т.д.);

– должен знать конкретный интерфейс источника данных, в данном случае это DataBaseSource;

– предоставляет внешний доступ к получению структур данных независимых от источника.

Model

– может быть связан с DataManager;

– может быть связан с Model;

– может быть связан с зависимыми от него View;

– предоставляет набор методов для взаимодействия со структурами данных, полученными от DataManager;

– организует работу зависимых от него View.

View

– должен зависеть от Model;

– реализует пользовательский интерфейс;

– предоставляет пользователю возможности для взаимодействия со структурами данных, посредством использования методов Model.

На рисунке 4 отображён пример возможной архитектуры приложения, реализованного на основании описанных принципов.

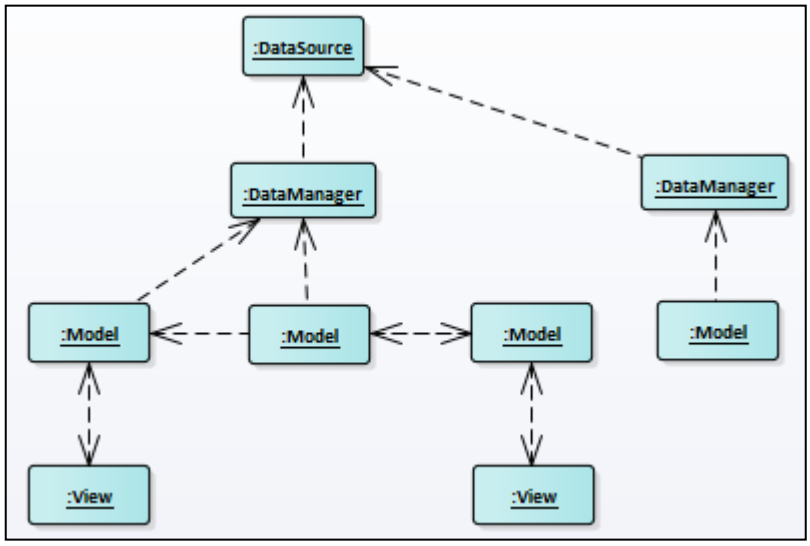

Рисунок 4 – Диаграмма классов.

Пример архитектуры приложения

Как можно заметить, что данный подход выглядит несколько сложнее, чем классический подход к проектированию программной архитектуры. Однако при использовании данного подхода возможно создание архитектуры, которую можно гибко подстроить под конкретные требования пользователя, что и является одной из главных задач стоящих в основании данной работы.

Таким образом, была разработана архитектура, которая может обеспечить взаимозаменяемость и добавление новых модулей, не нарушая при этом, работы всей системы. К тому же, учитывая тот факт, что компонент-агрегатор также содержится в отдельном модуле, то даже описанная архитектура может с лёгкостью быть заменена на другую, при использовании другого модуля компонента-агрегатора.

На рисунке 5 отражена диаграмма компонентов, содержащая все описанные на текущий момент компоненты архитектуры, а также отношения между ними.

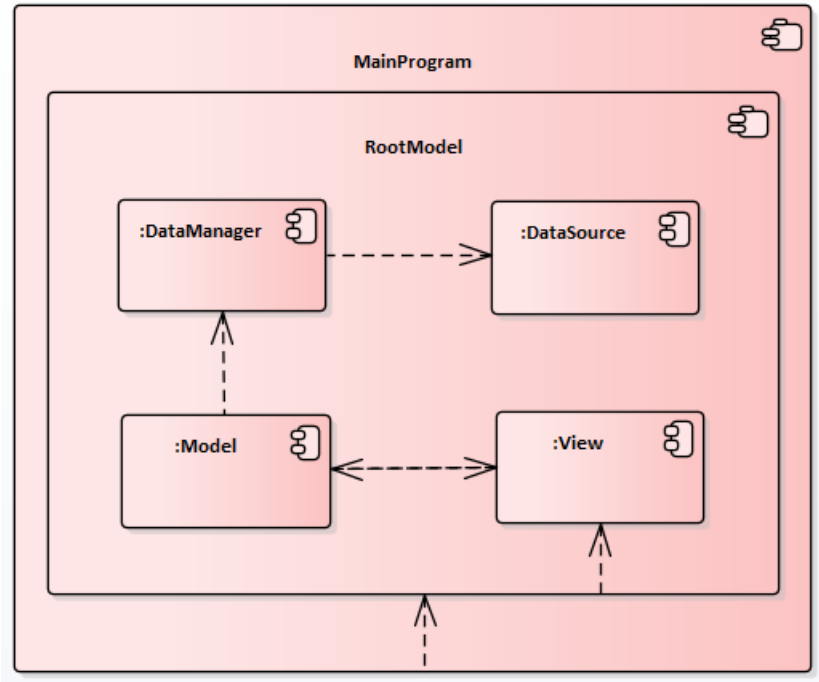

Рисунок 5 – Диаграмма компонентов.

Архитектура системы

4 ВЫБОР И ОБОСНОВАНИЕ ИНСТРУМЕНТОВ РАЗРАБОТКИ

Основными требованиями к инструментам разработки являются кроссплатформенность и возможность разработки программ с динамически подключаемыми модулями. Также важным аспектом инструмента разработки является доступность и простота использования.

В сравнении будут приведены языки программирования C#, Java и C++, так как эти языки на протяжении долгого времени интенсивно используются и развиваются, а также обладают крупными сообществами использующих их разработчиков.

Следует начать с того, что написание программного кода на языках C# и Java проще и быстрее, по сравнению с C++. Ключевым фактором в данном случае является наличие “сборщика мусора” – особого механизма платформы, который периодически освобождает память, удаляя объекты, которые уже не используются программой. Таким образом, программист не должен самостоятельно решать задачи выделения и освобождения памяти. Существуют предметные области, обычно связанные с интенсивными вычислениями или системами реального времени, в которых использование “сборщика мусора” недопустимо, но в основной массе проектов, связанных с бизнес-логикой и пользовательскими интерфейсами, этот фактор не влияет негативно на выполнение программы, а только упрощает её разработку и поддержку.

Все перечисленные языки обладают свойством кроссплатформенности исполнения, однако способы её обеспечения между ними несколько разнятся. В требованиях к инструментам разработки заявлена поддержка Windows, Mac OS и Linux, а в требованиях к пользовательским компонентам - Windows, Mac OS, Linux, Android, iOS. В таблице 2 представлены решения, используемые каждым из языков применительно к разработке программ для всех указанных платформ. Следует отметить, что решения должны обеспечивать не только

кроссплатформенность работы программного кода бизнес-логики, но и кода пользовательского интерфейса. В сравнении не указаны компиляторы, так как для каждой платформы их может быть несколько.

Таблица 2 - Решения, используемые для кроссплатформенной разработки

Платформа	Язык		
	C#	Java	C++
Windows	.NET + (Xamarin.Forms, WPF)	(JRE, AVIAN) + (Swing, SWT, Zetes)	(Qt, wxWidgets)
Mac OS	MONO + Xamarin.Forms	(JRE, AVIAN) + (Swing, SWT, Zetes)	(Qt, wxWidgets)
Linux	MONO + Gtk#	(JRE, AVIAN) + (Swing, SWT, Zetes)	(Qt, wxWidgets)
iOS	MONO + (Xamarin.Forms, Xamarin.iOS)	Codename One (ParparVM + Swing)	Qt
Android	MONO + (Xamarin.Forms, Xamarin.Android)	JRE + Swing	NDK + Qt

Как можно заметить, наборы решений для C# и Java под указанные платформы несколько отличаются. Используя решения, доступные для языка C#, одним набором нельзя охватить две группы, в одной из которых Windows, Mac OS, iOS и Android, а в другой Linux-подобные системы. При разработке коммерческого продукта, целью которого является охват как можно более большей аудитории, этот факт не является большим недостатком, так как аудитория пользователей Linux-подобных операционных систем существенно меньше, чем всех остальных. Для языка Java слабой стороной является поддержка iOS – так как для её поддержки существуют только специальные решения, предназначенные только для платформ мобильных устройств, не поддерживающих платформы настольных ПК. Таким образом, для разработки под все указанные в требованиях платформы, независимо от выбора C# или

Java, пришлось бы в ходе разработки поддерживать две версии с использованием нескольких решений, что существенно усложнило бы задачу.

Среди решений для C++ можно выделить одно решение, поддерживающее все указанные платформы – фреймворк Qt. Несмотря на то, что разработка на C++ медленнее и требует более высокой квалификации разработчика, такой выбор позволит в полной мере удовлетворить предъявляемые требования.

В качестве IDE для разработки наилучшим решением будет использование QtCreator, так как он тесно связан с фреймворком Qt: облегчает сборку приложений под различные платформы, обладает встроенной документацией Qt и доступен на Windows, Mac OS и Linux, в соответствии с предъявленными требованиями.

5 РАЗРАБОТКА НАБОРА БАЗОВЫХ КОМПОНЕНТОВ И ИНСТРУМЕНТОВ РАЗРАБОТКИ

Для внесения ясности, следует ещё раз отметить, что основной идеей разработки этого программно-технологического является построение архитектуры программы, используя модульный подход – вся функциональность программы содержится в различных наборах модулей, где каждый модуль должен нести ответственность лишь за выполнение одной задачи.

В данном разделе будет описана только функциональность, которая обеспечивает работу всей системы в целом, а именно метод запуска программы и работа компонента-агрегатора, связанная со связыванием модулей между собой.

На всех платформах программа представляет собой единственный исполняемый файл. В прямом доступе этого файла, в разделе файловой системы хранятся подключаемые модули. Модули представляют собой динамические библиотеки с расширениями *.dll для Windows, *.so для Linux, Mac OS, iOS и Android. После запуска, программа загружает все находящиеся в разделе библиотеки и ищет среди них модуль, который наследует интерфейс ICorePlugin. Если такой найден, то все прочие неразрешённые модули она направляет в метод AddPlugin() интерфейса ICorePlugin. После передачи всех модулей программа вызывает метод Run() интерфейса ICorePlugin.

Если модуль реализует интерфейс ICorePlugin, то он берёт на себя ответственность за настройку всех взаимодействий между модулями. Интерфейс выглядит следующим образом:

```
class ICorePlugin
{
public:
   virtual ~ICorePlugin() {}
   virtual void AddPlugin(QObject* instance, QJsonObject* meta) = 0;
   virtual void Run(QWidget* parentWidget) = 0;
};
```

На этом все функции программы, реализованные без использования модульного подхода, заканчиваются. Вся основная функциональность содержится в динамически загруженных модулях, модули загружаются с использованием возможностей класса QPluginLoader и должны содержать специфические атрибуты. Для упрощения процесса создания новых модулей был реализован генератор программного кода, который формирует весь массив кода, необходимый для корректной работы модуля в составе системы. Интерфейс генератора программного кода представлен на рисунке 5.

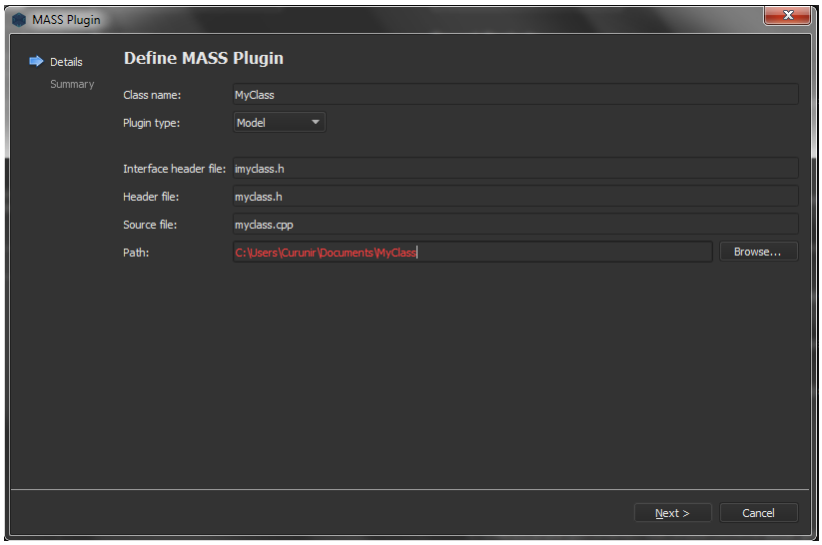

Рисунок 5 – Интерфейс генератора программного кода модуля

6 РАЗРАБОТКА КОМПОНЕНТОВ, ПРЕДОСТАВЛЯЮЩИХ ДИНАМИЧЕСКИ-КОНФИГУРИРУЕМУЮ ФУНКЦИОНАЛЬНОСТЬ СИСТЕМЫ

6.1 Основные сведения

Для решения поставленных задач был реализован только один модуль, реализующий интерфейс ICorePlugin – модуль MainMenuModel. Указанный модуль предоставляет набор интерфейсов, определяющих правила построения архитектуры и обеспечивает установление соединения между модулями.

Рассмотрим правила построения архитектуры, определяемые модулем MainMenuModel.

Как уже было упомянуто – каждый модуль должен нести ответственность лишь за выполнение одной задачи. В соответствии с этим были определены следующие типы модулей, определяющие их области ответственности:

1. Model – выполнение бизнес-логики и взаимодействие с другими модулями.
2. View – задачи связанные с пользовательским интерфейсом.
3. DataManager – предоставление структур данных, которые не зависят от источника, из которого они были получены (база данных, интернет, COM-порт и прочие).
4. DataSource – взаимодействие с источником данных.

Все перечисленные типы модулей охватывают большую часть возможных функций приложения. Вообще говоря, на данный момент не существует специального механизма, который бы контролировал, что делает тот или иной модуль с определённым типом – тип модуля лишь определяет модуль как поставщика какой-либо функции для всех прочих модулей.

6.2 Руководство по разработке и добавлению нового модуля в систему

6.2.1 Создание проекта

Создание проекта со всеми необходимыми файлами очень просто выполнить, используя генератор программного кода, описанный ранее. В нём необходимо выбрать название и тип модуля.

6.2.2 Взаимодействие с другими модулями

Для того чтобы взаимодействовать с другим модулем необходимо знать два его атрибута:

1. Его интерфейс.
2. Тип модуля.

Каждый модуль имеет свои мета-данные в формате JSON, этот файл хранится в разделе модуля. Мета-данные выглядят следующим образом:

```
{
  "Type": "PluginModel",
  "Interface": "IMyCustomModel",
  "Name": "MyCustomModel",
  "RelatedPluginInterfaces": [
  ]
}
```

Поля “Type”, “Interface” и “Name” заполняются данными, которые были указаны при создании модуля через генератор. Поле “RelatedPluginInterfaces” содержит ссылки на другие модули – если необходимо обеспечить взаимодействие с другими модулями, то здесь должны быть указаны соответствующие интерфейсы. Впоследствии указанные модули будут предоставлены модулем, который реализовал интерфейс ICorePlugin.

Для ясности следует на примере разобрать процесс настройки взаимодействия с модулем. Например, новый модуль должен

взаимодействовать с модулем типа DataManager, для получения каких-либо данных, и с модулем типа Model, для получения доступа к его бизнес-логике.

Сперва необходимо определить их типы и интерфейс – DATAMANAGERPLUGIN, IExtendableDataManager и COREPLUGIN, IMainMenuModel, соответственно.

Затем, необходимо разместить полученные атрибуты в мета-данных собственного модуля:

```
{
   "Type": "PluginModel",
   "Interface": "IMyCustomModel",
   "Name": "MyCustomModel",
   "RelatedPluginInterfaces": [
      "IExtendableDataManager",
      "IMainMenuModel"
   ]
}
```

После этого в сгенерированном файле “*.cpp” следует найти метод AddReferencePlugin(). В определении этого метода существует пример его использования:

```
void MyCustomModel::AddReferencePlugin(PluginInfo *pluginInfo)
{
   /* Select your reference plugin type case and get it. For example:
      case MODELPLUGIN:{
         myReferencedPlugin = qobject_cast<ISomePlugin*>(pluginInfo->Instance);
         if(!myReferencedPlugin)
            return;
         connect(this, SIGNAL(OnClose(PluginInfo*)), pluginInfo->Instance,
SLOT(ReferencePluginClosed(PluginInfo*)));
```

```
    } break;
*/
```

Запрошенные в мета-данных модули поступят в этот метод, и их нужно будет принять, определить и сохранить. Так как все модули наследуют класс QObject, для безопасного приведения типа необходимо использовать qobject_cast:

```
case COREPLUGIN:{
    mainMenuModel =
        qobject_cast<IMainMenuModel*>(pluginInfo->Instance);
    if(!mainMenuModel)
        return;
    connect(this, SIGNAL(OnClose(PluginInfo*)),
        pluginInfo->Instance, SLOT(ReferencePluginClosed(PluginInfo*)));
} break;

case DATAMANAGERPLUGIN:{
    extendableDataManager =
        qobject_cast<IExtendableDataManager*>(pluginInfo->Instance);
    if(!extendableDataManager)
        return;
    connect(this, SIGNAL(OnClose(PluginInfo*)), pluginInfo->Instance,
        SLOT(ReferencePluginClosed(PluginInfo*)));
}break;
```

6.2.3 Предоставление функций другим модулям

Как и все прочие модули, ваш модуль имеет интерфейс:

```
class IMyCustomModel :  public IModelPlugin
{
public:  // Write your interface methods here
};
```

Всё что нужно – добавить в него методы и реализовать их.

6.2.4 Разработанные модули

Далее будут рассмотрены спроектированные классы, реализующие интерфейсы, описанные в этом разделе, каждый рассмотренный класс является подгружаемым модулем.

В целях упрощения наглядного представления архитектуры приложения, её описание было разделено на две отдельных диаграммы классов – на рисунке 7 показаны отношения классов, реализующих интерфейс Model и связи между ними, а на рисунке 8 в отношении всех спроектированных классов, но описывая только связи между View-Model, Model-DataManager, DataManager-DataSource.

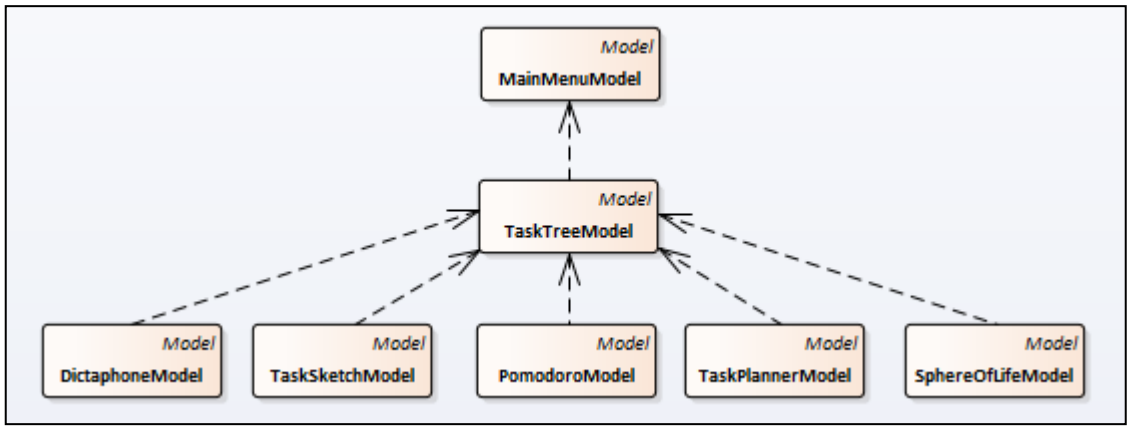

Рисунок 6 – Диаграмма классов.

Архитектура приложения – зависимости между Model-Model

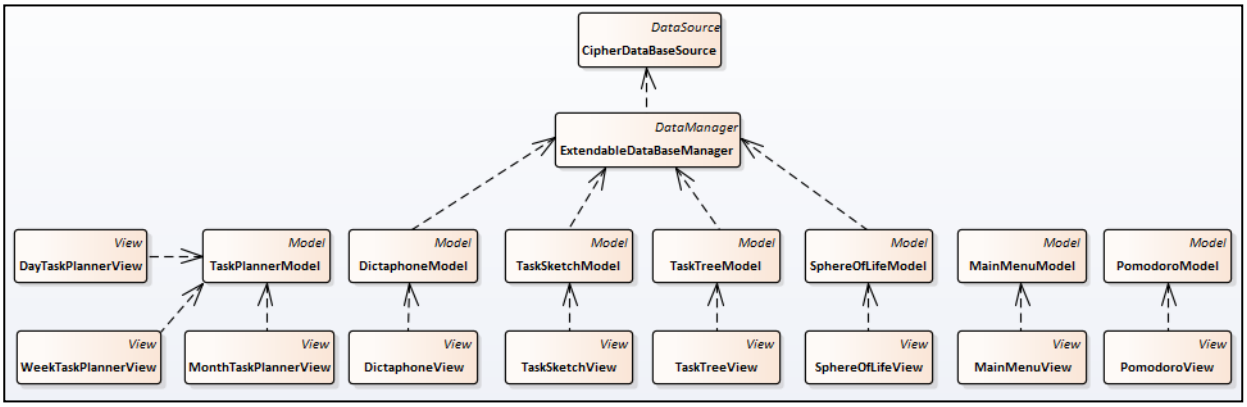

Рисунок 7 – Диаграмма классов.

Архитектура приложения – зависимости между View-Model, Model-DataManager, DataManager-DataSource

MainMenuModel

Является классом реализующим интерфейсы CoreModel и Model.

Реализуя интерфейс CoreModel, он становится системой-агрегатором, организующей всё взаимодействие между загружаемыми модулями, а также

становится “корневым” модулем – все прочие модули получают ссылки на него, как на родительский модуль.

Реализуя интерфейс Model, он предоставляет набор функций для запуска модулей, позволяя запускать и закрывать зависимые от него модули, в которых и реализованы все основные функции приложения.

DataBaseSource

Реализует интерфейс DataBaseSource – обеспечивает взаимодействие с базой данных.

ExtendableDataBaseManager

Реализует интерфейс DataManager. Зависит от DataBaseSource**.** Посредством набора методов, предоставляемых этим модулем, зависимые модули запрашивают необходимые им структуры данных, которые ExtendableDataBaseManager получает от DataBaseSource, посредством формирования SQL-запросов.

TaskTreeModel

Реализует интерфейс Model. Зависит от MainMenuModel и ExtendableDataBaseManager. Обеспечивает обработку структур данных, полученных от ExtendableDataBaseManager, представляя их в виде дерева. Расширяет атрибуты задачи, добавляя название задачи. Является центральным классом в контексте обработки пользовательских задач, так как именно в нём они все хранятся. Предоставляет набор методов в контексте пользовательских задач.

TaskTreeView

Реализует интерфейс View. Зависит от TaskTreeModel. Получает от TaskTreeModel структуру данных с пользовательскими задачами, которую отображает пользователю. Связывает пользовательский интерфейс с возможностями, предоставляемыми TaskTreeModel.

TaskSketchModel

Реализует интерфейс Model. Зависит от TaskTreeModel и ExtendableDataBaseManager. Расширяет атрибуты задачи, добавляя графическую информацию – зарисовку (простой рисунок пользователя). Хранит галерею с пользовательскими зарисовками. Предоставляет возможности создания новых задач на основании зарисовки, используя методы, предоставляемые TaskTreeModel.

TaskSketchView

Реализует интерфейс View. Зависит от TaskSketchModel. Предоставляет пользователю интерфейс, позволяющий просматривать галерею с зарисовками, а также возможность создания новых зарисовок. Используя методы TaskSketchModel возможно добавление в TaskTreeModel новых задач содержащих зарисовки из галереи.

DictaphoneModel

Реализует интерфейс Model. Зависит от TaskTreeModel и ExtendableDataBaseManager. Расширяет атрибуты задачи, добавляя аудиозапись. Хранит коллекцию пользовательских аудиозаписей. Предоставляет возможности создания новых задач на основании пользовательской аудиозаписи, используя методы, предоставляемые TaskTreeModel.

DictaphoneView

Реализует интерфейс View. Зависит от TaskTreeModel. Предоставляет пользователю интерфейс, с помощью которого можно создавать новые аудиозаписи, реализуя функции диктофона.

TaskPlannerModel

Реализует интерфейс Model. Зависит от TaskTreeModel. Расширяет атрибуты задачи, добавляя дату и время. Предоставляет возможности для предоставления задач за определённый период времени.

DayTaskPlannerView

Реализует интерфейс View. Зависит от TaskPlannerModel. Предоставляет пользователю интерфейс, позволяющий планировать день, присваивая задачам определённые временные промежутки.

WeekTaskPlannerView

Реализует интерфейс View. Зависит от TaskPlannerModel.Предоставляет пользователю интерфейс, позволяющий планировать неделю, присваивая задачам определённые дни недели.

MonthTaskPlannerView

Реализует интерфейс View. Зависит от TaskPlannerModel. Предоставляет пользователю интерфейс, позволяющий планировать месяц, присваивая задачам определённые дни недели.

PomodoroModel

Реализует интерфейс Model. Зависит от TaskTreeModel. Расширяет свойства задачи, добавляя количество затраченных на неё рабочих циклов Pomodoro.

PomodoroView

Реализует интерфейс View. Зависит от PomodoroModel. Предоставляет пользовательский интерфейс, посредством которого пользователь может использовать технику Pomodoro в отношении определённых задач.

SphereOfLifeModel

Реализует интерфейс Model. Зависит от TaskTreeModel и ExtendableDataBaseManager. Расширяет свойства задачи, добавляя атрибут, относящий эту задачу к определённой сфере жизни. Также хранит список, заданных пользователем сфер жизни.

SphereOfLifeView

Реализует интерфейс View. Зависит от SphereOfLifeModel. Предоставляет пользовательский интерфейс, позволяющий изменять текущий

список сфер жизни пользователя. Выводит статистику выполнения задач по отдельным сферам жизни.

В результате был разработан модуль, выполняющий функции компонента-агрегатора – класс MainMenuModel. Модуль получает от основной программы доступные модули, определяет их характеристики на основании мета-данных, а затем устанавливает необходимые зависимости между ними. После этого все модули получают сигнал о завершении этапа загрузки, и приложение начинает свою работу. Также модуль компонента-агрегатора реализует функции меню – предоставляет интерфейс, через который можно получить информацию о загруженных модулях и запустить один из них. Однако этот модуль не реализует пользовательский интерфейс – для этого он, на этапе загрузки, устанавливает связь с другим – MainMenuView, который как раз предназначен для этого. Результат работы модулей MainMenuModel и MainMenuView на платформе Windows представлен на рисунке 8, а на платформе Android на рисунке 9.

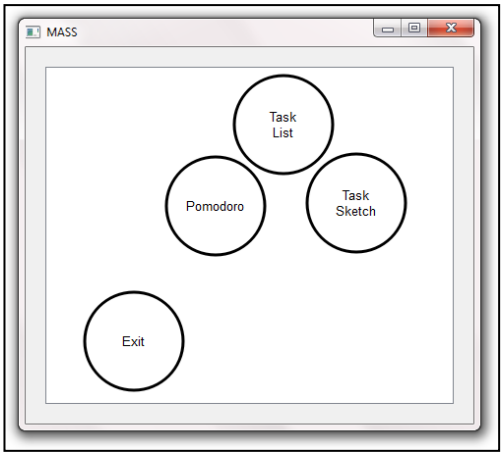

Рисунок 8 – Результат работы MainMenuModel и MainMenuView. Платформа Windows

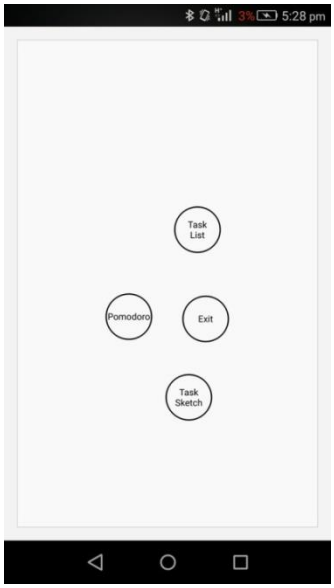

Рисунок 9 – Результат работы MainMenuModel и MainMenuView.

Платформа Android

Результат работы модулей TaskListModel и TaskListView на платформе Windows представлен на рисунке 10, а на платформе Android на рисунке 11. Из задача заключается в предоставлении пользователю списка задач, а также возможности добавления, редактирования и удаления задач в списке.

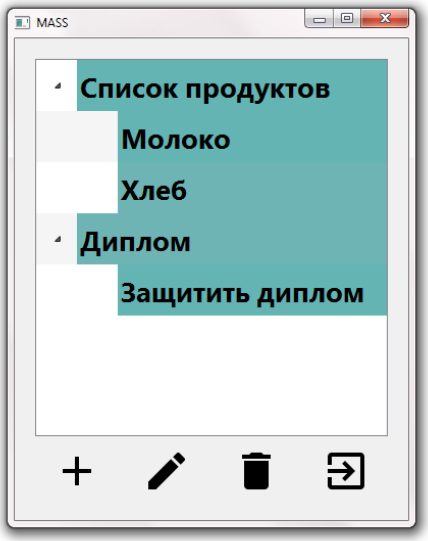

Рисунок 10 – Результат работы TaskListModel и TaskListView.

Платформа Windows

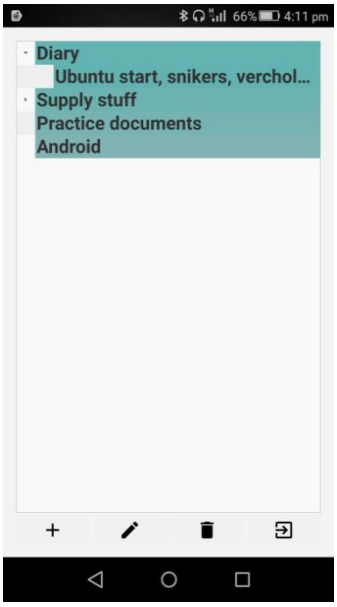

Рисунок 11 – Результат работы TaskListModel и TaskListView.

Платформа Android

Результат работы модулей TaskSketchModel и TaskSketchView на платформе Windows представлен на рисунке 12, а на платформе Android на рисунке 13. Их задача заключается в предоставлении пользователю возможности создавать простые зарисовки с использованием устройств ввода и сохранять их в галерее. В дальнейшем на основании зарисовки можно создать новую задачу, которую можно будет увидеть в списке задач, используя возможности модуля TaskListView.

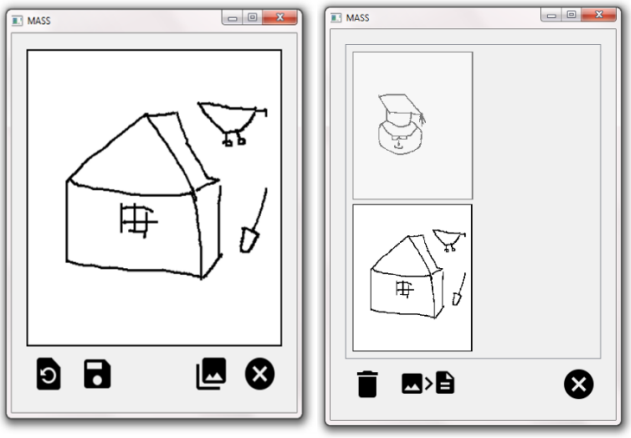

Рисунок 12 – Результат работы TaskSketchModel и TaskSketchView.

Платформа Windows

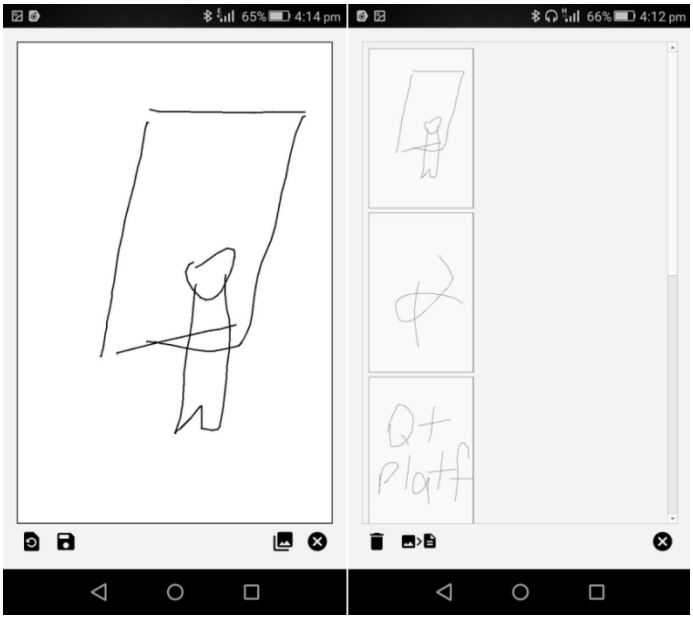

Рисунок 13 – Результат работы TaskSketchModel и TaskSketchView. Платформа Android

Результат работы модулей PomodoroModel и PomodoroView на платформе Windows представлен на рисунке 14, а на платформе Android на рисунке 15. Их задача заключается в предоставлении пользователю возможности использования техники Pomodoro. Через 25 минут, после запуска таймера будет подан звуковой сигнал, сигнализирующий о необходимости отдыха, а через 5 минут отдыха будет подан сигнал о возобновлении работы.

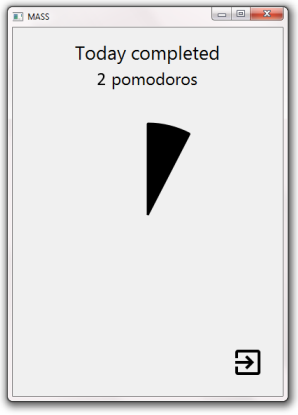

Рисунок 14 – Результат работы PomodoroModel и PomodoroView. Платформа Windows

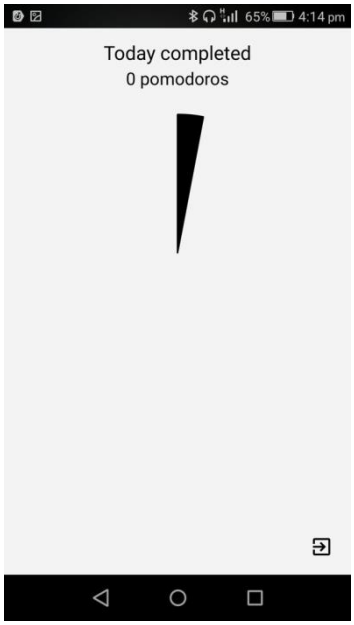

Рисунок 15 – Результат работы PomodoroModel и PomodoroView. Платформа Android

7 ВЫВОДЫ

Результаты проделанной работы продемонстрированы в таблице 3.

Таблица 3 – Результаты проведенных работ в соответствии с техническим заданием

№	Название проводимых работ	Ожидаемые результаты	Результаты работ	Соответствие техническому заданию
1	Исследование предметной области, связанной с разработкой систем, обеспечивающих динамическое конфигурирование функциональности, а также определение возможных способов поддержки повторного использования существующих решений	Описание предметной области, достаточное для обоснования проектных решений из п/п 2	Была описана предметная область, в объёме, достаточном для обоснования проектных решений из п/п 2	Соответствует
2	Разработка архитектуры	Описание архитектуры	Была описана архитектура	Соответствует

	программно-технологического компонента системы	программно-технологического компонента, достаточное для выполнения п/п 3, 4 и 5	программно-технологического компонента, достаточно для выполнения п/п 3, 4 и 5	
3	Выбор и обоснование инструментов разработки	Приведение возможных вариантов выбора инструментов, сравнение их между собой, обоснование выбора	Были приведены возможные варианты выбора инструментов, сравнение их между собой, обоснование выбора	Соответствует
4	Разработка набора базовых компонентов и инструментов разработки, которые впоследствии будут служить для создания компонентов, которые в своей совокупности будут предоставлять динамически-	Набор базовых компонентов и инструментов разработки для компонентов, которые в своей совокупности будут составлять динамически-конфигурируемую функциональность системы	Был разработан набор базовых компонентов и инструментов разработки для компонентов, которые в своей совокупности будут составлять динамически-конфигурируемую функциональность системы	Соответствует

	конфигурируемую функциональность системы			
5	Разработка компонентов, предоставляющих динамически-конфигурируемую функциональность системы, используя набор базовых компонентов и инструментов разработки, которые являются результатом выполнения п/п 4	Компоненты, соответствующие функциональными требованиям к пользовательским компонентам, разработанные с использованием набора базовых компонентов и инструментов разработки, которые являются результатом выполнения п/п 4	Были реализованы компоненты, соответствующие функциональными требованиям к пользовательским компонентам, разработанные с использованием набора базовых компонентов и инструментов разработки, которые являются результатом выполнения п/п 4	Соответствует

Вывод: в ходе проделанной работы был выполнен весь объем работ в соответствии с техническим заданием.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был разработан прототип программно-технологического компонента системы с динамически-конфигурируемой функциональностью и поддержкой повторного использования существующих решений.

В разделе исследования предметной области были определены возможные подходы к решению задач, связанных с разработкой программ обладающих динамически-конфигурируемой функциональностью и поддержкой повторного использования существующих решений.

В разделе разработки архитектуры были определены основные требования, которые позволили сформировать модель архитектуры программы, с учётом особенностей поставленной задачи.

В разделе выбора и обоснования инструментов разработки был произведён обзор существующих решений и приведено обоснование выбора одного из приведённых решений.

В разделе разработки набора базовых компонентов и инструментов разработки были описаны основные аспекты реализации программно-технологического компонента системы и необходимые для этого инструменты.

В разделе разработки компонентов, предоставляющих динамически-конфигурируемую функциональность системы, были приведены общие сведения по особенностям их реализации, руководство по разработке и добавлению новых компонентов в систему, а также описание разработанных компонентов, представляющих в своей совокупности прототип работающей системы.

Таким образом, все задачи, поставленные в рамках выполнения курсового проекта, были успешно решены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Крылов К.А., Повторное использование программного обеспечения в области разработки персональных информационных менеджеров – Спб.: БГТУ, III общероссийская молодежная научно-техническая конференция «Старт-2017», 2017.
2. Гущин А.Н., Личностно-ориентированные информационные системы – Спб.: БГТУ, 2012. – 122 с.
3. Существует ли идеальный планировщик личных задач? Разработка модульного планировщика. [Электронный ресурс]. URL: habrahabr.ru/post/340018/.com[iz-pesochnitsy]-suschestvuet-li-idealnyy-p
4. Дорот В. Л. Толковый словарь современной компьютерной лексики, 3 изд. – БХВ-Петербург, 2004.
5. Иан С. Инженерия программного обеспечения 6-е издание – М: Издательский дом «Вильямс. – 2002.
6. Clements P., Northrop L. Software product lines. – Addison-Wesley, 2002.
7. Bosch J. From software product lines to software ecosystems //Proceedings of the 13th international software product line conference. – Carnegie Mellon University, 2009. – С. 111-119.
8. Manikas K., Hansen K. M. Software ecosystems–A systematic literature review //Journal of Systems and Software. – 2013. – Т. 86. – №. 5. – С. 1294-1306.
9. Yu L., Ramaswamy S., Bush J. Symbiosis and software evolvability //IT Professional. – 2008. – Т. 10. – №. 4.
10. Manikas K., Hansen K. M. Software ecosystems–A systematic literature review //Journal of Systems and Software. – 2013. – Т. 86. – №. 5. – С. 1294-1306.