Министерство Образования, Культуры и Исследований Республики Молдова
Технический Университет Молдовы
Факультет Вычислительной Техники, Информатики и Микроэлектроники
Департамент ISA

Курсовая работа

По предмету

«Техника и Методы Проектирования Систем»

Тема

«Использование шаблонов проектирования в реализации приложения для тренировки устного счёта»

Выполнил: ст. гр. TI-196 Вака Вадим

Проверили: унив. лектор Поштару А

Кишинёв, 2022

Содержание

Введение

Опыт позволяет человеку развиваться и расти над собой на протяжении всей жизни. Перенос накопленного опыта в источники, из которых его могут подчерпнуть другие, одно из важнейших условий развития, как в области умственного, так и в области физического труда. В любой сфере существуют наилучшие или попросту общепринятые решения, отточенные годами, если не десятилетиями практики.

Программирование - область не способная насчитать в своей истории много веков, однако все когда-либо принятые в данной области решения являются задокументированными изначально, а потому неудивительно, что существуют подобные приёмы, как шаблоны проектирования.

К сожалению, или к счастью, шаблоны проектирования не являются универсальными, а скорее наоборот достаточно специфичны и для их верного использования необходимо полностью понимать их суть и принцип действия.

Данная работа, написанная на основе приложения для тренировки навыков устного счёта, в первую очередь ставит передо мной задачу понимания необходимости и эффективности использования шаблонов проектирования в уместных для них фрагментах кода. Изначально данное приложение было выполнено мной без использования шаблонов, чтобы после, видя всю картину приложения, внедрить подходящие шаблоны там, где они действительно способны улучшить читаемость кода и/или его эффективность, что и было подтверждено в процессе выполнения работы.

1. Паттерны проектирования

При создании программных систем перед разработчиками часто встает проблема выбора тех или иных проектных решений. В этих случаях на помощь приходят паттерны (шаблоны) проектирования. Дело в том, что почти наверняка подобные задачи уже решались ранее и уже существуют хорошо продуманные элегантные решения, составленные экспертами. Если эти решения описать и систематизировать в каталоги, то они станут доступными менее опытным разработчикам, которые после изучения смогут использовать их как шаблоны или образцы для решения задач подобного класса. Паттерны как раз описывают решения таких повторяющихся задач.

Иными словами, можно вывести более строгое определение, а шаблоны проектирования приложений — это многократно применяемое решение регулярно возникающей проблемы в рамках определённого контекста архитектуры приложения. Шаблон — это не законченное архитектурное решение, которое можно напрямую преобразовать в исходный или машинный код, а описание подхода к решению проблемы, который можно применять в разных ситуациях.

1.1 История создания шаблонов проектирования

В 1970-е годы архитектор Кристофер Александр составил набор шаблонов проектирования. В области архитектуры эта идея не получила такого развития, как позже в области программной разработки.

В 1987 году Кент Бэк (Kent Beck) и Вард Каннингем (Ward Cunningham) взяли идеи Александра и разработали шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке Smalltalk.

В 1988 году Эрих Гамма (Erich Gamma) начал писать докторскую диссертацию при цюрихском университете об общей переносимости этой методики на разработку программ.

В 1989—1991 годах Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на C++ и опубликовал в 1991 году книгу Advanced C++ Idioms.

В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides) публикует книгу Design Patterns — Elements of Reusable Object-Oriented Software. В этой книге описаны 23 шаблона проектирования. Также команда авторов этой книги известна общественности под названием «Банда четырёх» (англ. Gang of Four, часто сокращается до GoF). Именно эта книга стала причиной роста популярности шаблонов проектирования.

1.2 Типы шаблонов

В настоящее время наиболее популярными паттернами являются паттерны проектирования. Одной из распространенных классификаций таких паттернов является

классификация по степени детализации и уровню абстракции рассматриваемых систем. Паттерны проектирования программных систем разделяются на т\три основные категории, как:

1. Порождающие паттерны (Creational). Отвечают за описание создание одного или группы объектов.
2. Структурные паттерны (Structural). Отвечают за способы построения связей между различными объектами.
3. Поведенческие паттерны (Behavioral). Отвечают за способы эффективного взаимодействия между объектами.

1.3 Порождающие паттерны проектирования

Как правило, знакомство с паттернами проектирования всегда начинают именно с этой категории. Данная группа паттернов отвечает за оптимальное создание объектов или семейств объектов. Создающие паттерны проектирования абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту. В группу Создающих шаблонов проектирования входят следующие паттерны проектирования:

1. Одиночка (Singleton)
2. Фабричный метод (Factory Method)
3. Абстрактная фабрика (Abstract Factory)
4. Прототип (Prototype)
5. Строитель (Builder)

1.4 Структурные паттерны проектирования

Структурные шаблоны в основном связаны с композицией объектов, другими словами, с тем, как сущности могут использовать друг друга. В данную категорию шаблонов проектирования, отвечающих за структуру связей между объектами, принято отнести следующий список шаблонов проектирования:

1. Адаптер (Adapter)
2. Декоратор (Decorator)
3. Заместитель (Proxy)
4. Компоновщик (Composite)
5. Легковес (Flyweight)
6. Мост (Bridge)
7. Фасад (Facade)

1.5 Поведенческие паттерны проектирования

Данная группа шаблонов проектирования определяет алгоритмы и способы реализации взаимодействия различных объектов и классов. Это последняя категория паттернов проектирования, и отличается от первых двух многочисленностью паттернов, входящих в данную категорию. Шаблоны, входящие в категорию паттернов поведения, отвечают за взаимодействие объектов друг с другом. К ним относятся:

1. Стратегия (Strategy)
2. Команда (Command)
3. Наблюдатель (Observer)
4. Посетитель (Visitor)
5. Хранитель (Memento)
6. Посредник (Mediator)
7. Интерпретатор (Interpreter)
8. Итератор (Iterator)
9. Состояние (State)
10. Шаблонный метод (Template Method)
11. Цепочка обязанностей (Chain of Responsibility)

2. Мотивация к возданию приложения

Вычислительная техника достаточно прочно и глубоко вошла в нашу жизнь, и можно смело утверждать, что уже никогда не покинет её. Известно, что с развитием различных устройств, они перенимают часть функций, которые раньше человек выполнял самостоятельно. Самый простой пример - номера телефонов, до появления встроенного в телефон списка контактов, люди запоминал до 40 и более номеров, не прикладывая особых усилий, сегодня же, навряд ли многие перечислят и пять номеров.

Аналогичное можно сказать и про навыки устного счёта. Однако, данный навык носит больше прикладной характер и достаточно полезен как в работе, так и в повседневной жизни.

Я решил выбрать подобную тему по нескольким причинам. В первую очередь личным, по собственному опыту и наблюдениям могу заявить, что навыки устного счёта студентов понижаются во время обучения, так как в них нет необходимости. Любой студент моей группы согласится с тем, что его личные навыки в данной области ухудшились. Второй причиной стал мой потенциальный выбор для дипломной работы - обучающее приложения. Для создания оного в будущем я решил сейчас создать нечто упрощённое, чем и явился данный проект, где нет обучающих элементов, а есть только закрепление материала. Таким образом, с учётом полученного опыта, я буду лучше представлять себе будущую структуру дипломного проекта.

3. Проектирование и разработка

К составлению самой задачи я подходил с учётом имеющихся у меня навыков и средств для минимизации неожиданностей в поведении программы и устранения простоев из-за неопределённости в дальнейших действиях.

Для выполнения поставленной задачи подходит большинство средств доступных студенту третьего курса ТУМ, но предпочтительней всего выглядит компилируемый язык, поэтому я выбрал язык C++ как наиболее известный мне и обладающий всеми необходимыми функциями.

В качестве среды разработки выступила IDE Visual Studio 2019, являющаяся наилучшим выбором для разработки приложений различного уровня сложности на языке C++ и технологию WIN32 API соответственно.

3.1 Структура приложения

Для более упрощённого использования шаблонов и с целью обеспечения стабильности работы приложения, я решил разделить весь проект на несколько решений. В результате приложение выглядит следующим образом: есть несколько исполнительных файлов, в ручную запускается главный файл, он представляет из себя меню выбора дальнейших действий.

Пользователю предоставляется на выбор три уровня. В первом упор делается на сложение и вычитание, во втором на умножение и деление, в третьем же появляются степени. Нажатие левой клавишей мыши по одному из уровней приводит к запуску соответствующего приложения и закрытию любых других открытых ранее уровней.

Для реализации данных возможностей я использовал шаблон Посредник(Mediator).

3.1.1 Шаблон проектирования Посредник

Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда используется паттерн Посредник?

- Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
- Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

Участники

- Mediator: представляет интерфейс для взаимодействия с объектами Colleague
- Colleague: представляет интерфейс для взаимодействия с объектом Mediator
- ConcreteColleague1 и ConcreteColleague2: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
- ConcreteMediator: конкретный посредник, реализующий интерфейс типа Mediator

Схематично с помощью UML паттерн можно описать следующим образом:

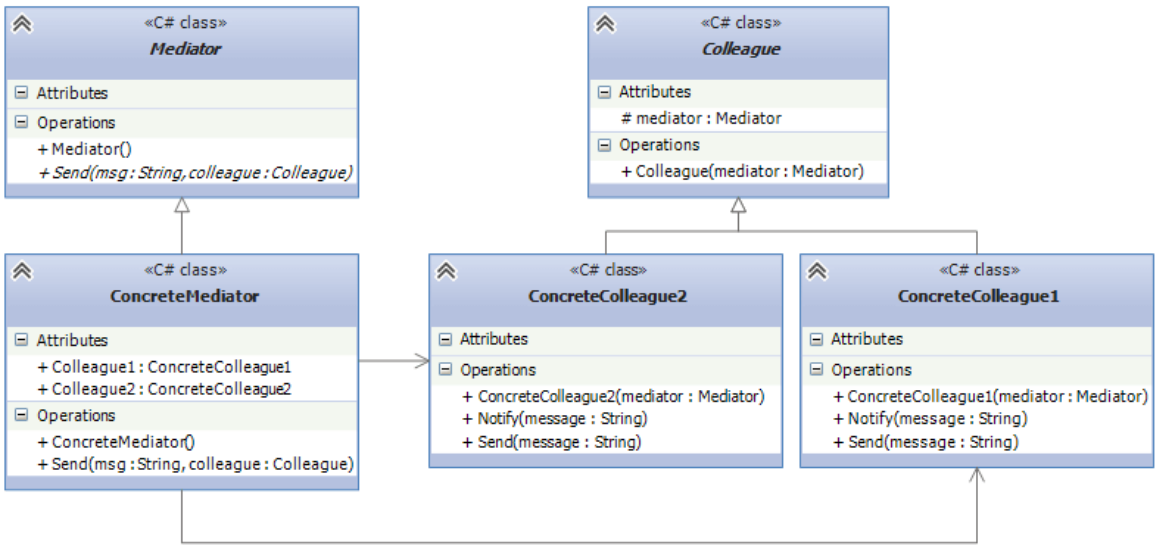


Рисунок 1. Общая диаграмма uml для посредника.

В рамках приложения реализация выглядит так:

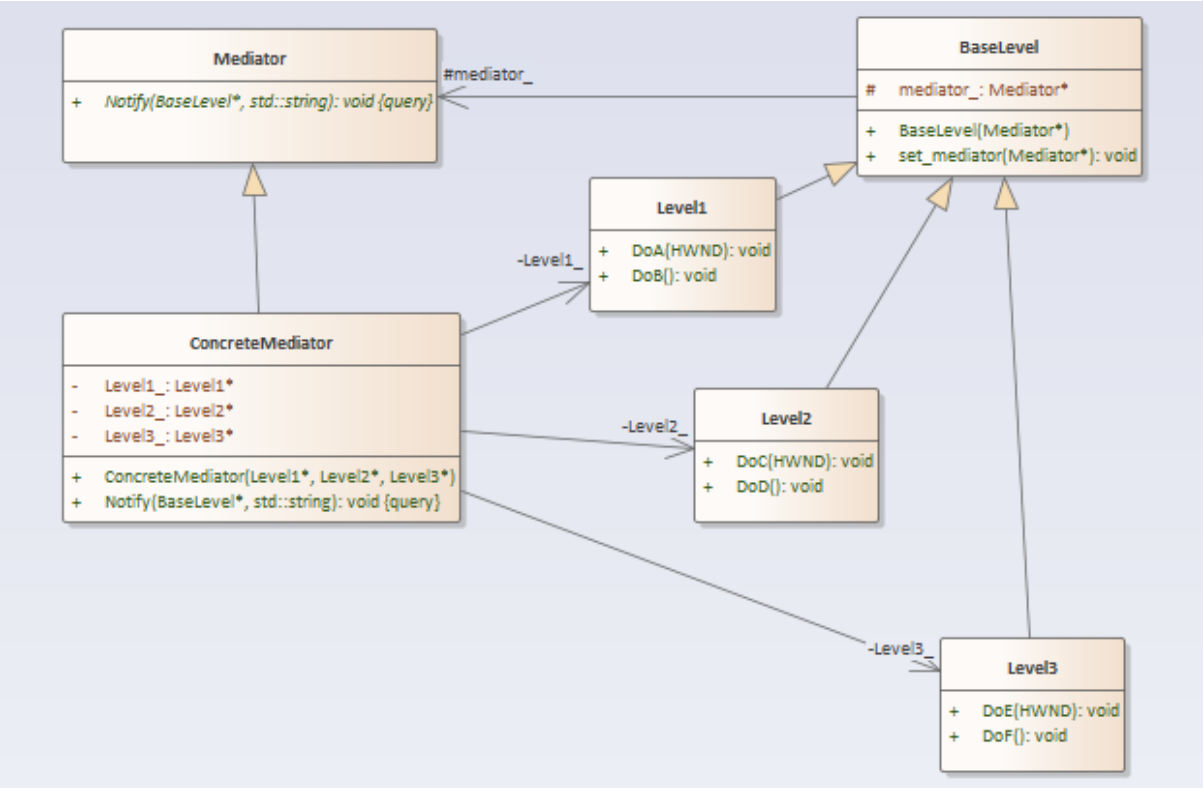


Рисунок 2. Диаграмма классов посредника.

На диаграмме, представленной на Рисунке 2, имеются следующие классы-участники:

- Mediator: представляет интерфейс для взаимодействия с объектами BaseLevel
- BaseLevel: представляет интерфейс для взаимодействия с объектом Mediator
- Level1, Level2 и Level3: конкретные классы уровней, которые обмениваются друг с другом через объект Mediator
- ConcreteMediator: конкретный посредник, реализующий интерфейс типа Mediator

3.2 Взаимодействие с интерфейсом

После открытия главного окна, необходимо выбрать уровень, это осуществляется кликом мыши по соответствующей строке текста. Изначально, для этого использовался обработчик команд мыши и проверка координат курсора с помощью оператора if с 4-мя аргументами, что естественно было не удобно и грозило ещё большим неудобством при дальнейшем росте программы. Так появилась необходимость в переводе координат курсора в номер строки текста, для чего идеальным и лаконичным решением выглядел адаптер.

3.2.1 Адаптер

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

Когда надо использовать Адаптер?

Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям

Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

Участники

Target: представляет объекты, которые используются клиентом

Client: использует объекты Target для реализации своих задач

Adaptee: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target

Adapter: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target

Формальное определение паттерна на UML выглядит следующим образом:

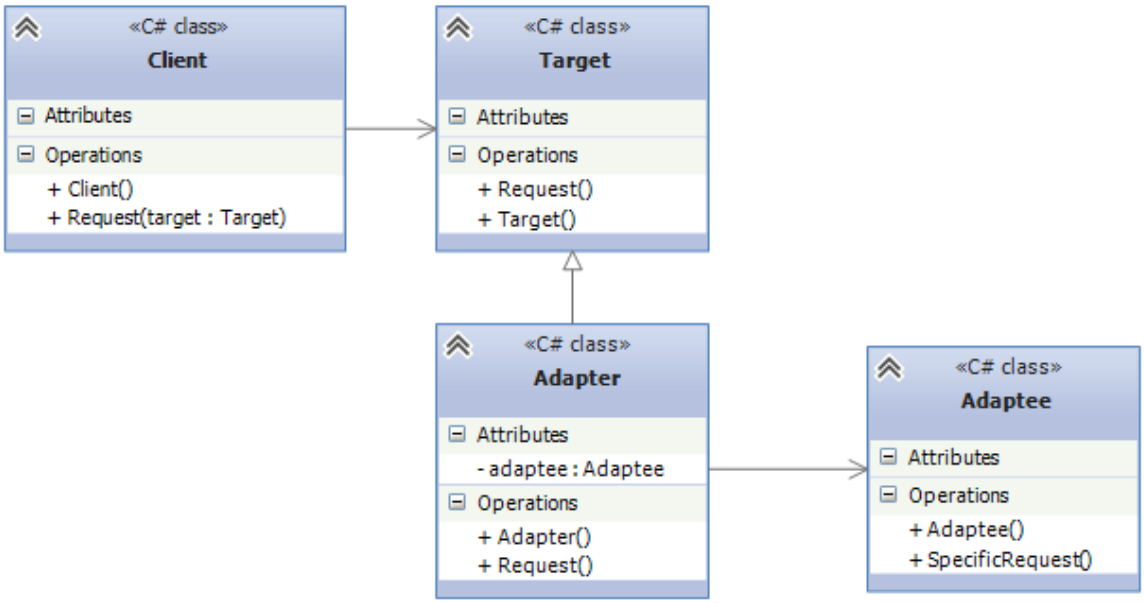


Рисунок 3. Диаграмма классов адаптера.

Однако есть альтернативный случай применения адаптера, когда речь идёт о преобразовании типов данных, что и происходит в моём случае. Так от POINT, в котором хранятся координаты, я перехожу к int, причём именно к нумерации, а не просто значению.

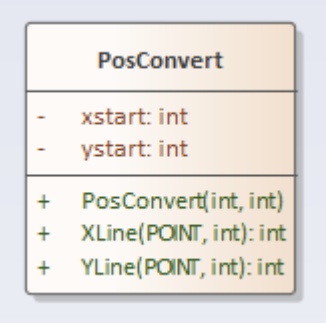


Рисунок 4. Диаграмма адаптера.

Надо понимать, что число участников де факто больше чем показано на диаграмме, только вот указывать базовые типы данных как участников выглядит немного странно, также как и указывать обработчик команд окна windows, который и выступает клиентом.

3.3 Вывод интерфейса

Для вывода интерфейса в окне программы используется ощутимое количество различных графических и около графических объектов. При относительном небольшом интерфейсе, всё это начинает занимать достаточное пространство, что по началу нисколько не мешает, однако при дальнейшем развитии программы в интерфейсе будут появляться различные блоки и уже сейчас пора задумываться о применении подходящего в данной ситуации шаблона - Фасада.

3.3.1 Фасад

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

Когда использовать фасад?

Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.

Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

В UML общую схему фасада можно представить следующим образом:

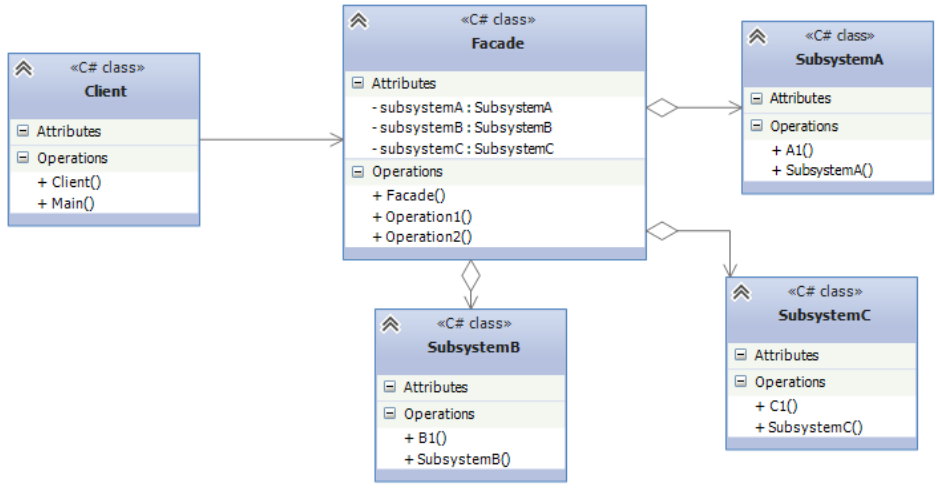


Рисунок 5. Диаграмма Фасада.

Участники

Классы SubsystemA, SubsystemB, SubsystemC и т.д. являются компонентами сложной подсистемы, с которыми должен взаимодействовать клиент

Client взаимодействует с компонентами подсистемы

Facade - непосредственно фасад, который предоставляет интерфейс клиенту для работы с компонентами

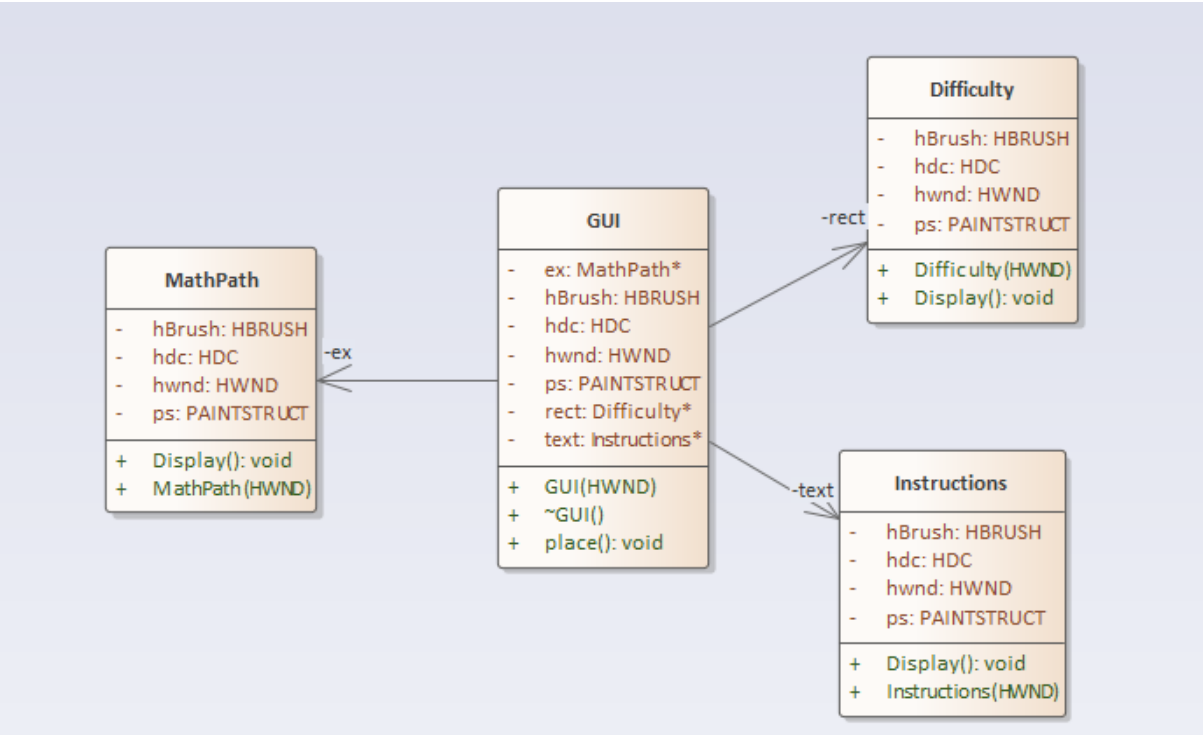


Рисунок 6. Диаграмма Фасада.

Участники

Классы Difficulty, MathPath, Instructions и т.д. являются компонентами сложной подсистемы, с которыми должен взаимодействовать клиент - обработчик команд windows.

GUI - непосредственно фасад, который предоставляет интерфейс клиенту для работы с компонентами.

3.4 Согласование работы уровней

В приложении большая часть действий производится по команде пользователя, в том числе и генерация нового примера, что может вызвать непонимание со стороны пользователя. В таком случае часть действий можно перенаправить на различные изменения состояния которые может улавливать Наблюдатель.

3.4.1 Наблюдатель

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

Когда использовать паттерн Наблюдатель?

Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях

Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.

Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

Участники

IObservable: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)

ConcreteObservable: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.

IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.

ConcreteObserver: конкретная реализация интерфейса IObserver.

При этом наблюдаемому объекту не надо ничего знать о наблюдателе кроме того, что тот реализует метод Update(). С помощью отношения агрегации реализуется слабосвязанность обоих компонентов. Изменения в наблюдаемом объекте не виляют на наблюдателя и наоборот.

В определенный момент наблюдатель может прекратить наблюдение. И после этого оба объекта - наблюдатель и наблюдаемый могут продолжать существовать в системе независимо друг от друга.

С помощью диаграмм UML данный шаблон можно выразить следующим образом:

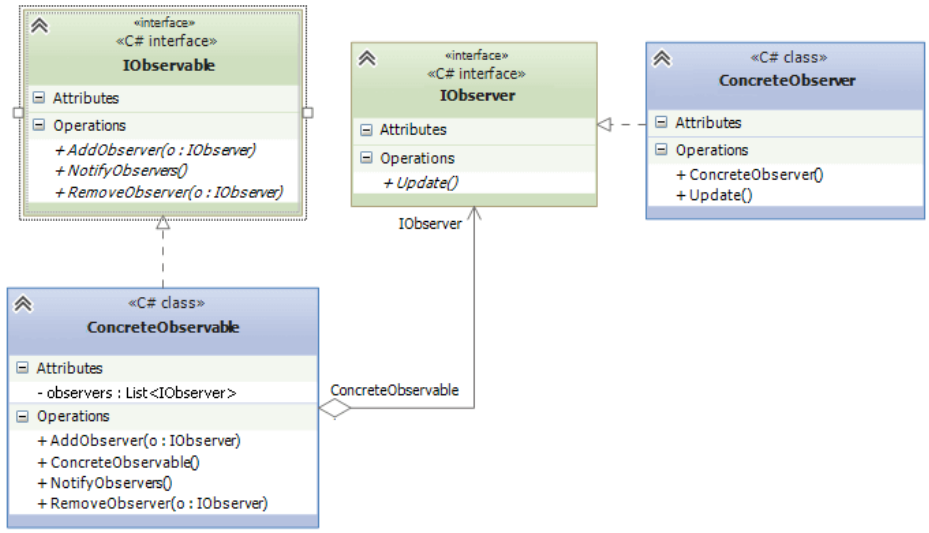


Рисунок 7. Диаграмма Наблюдателя.

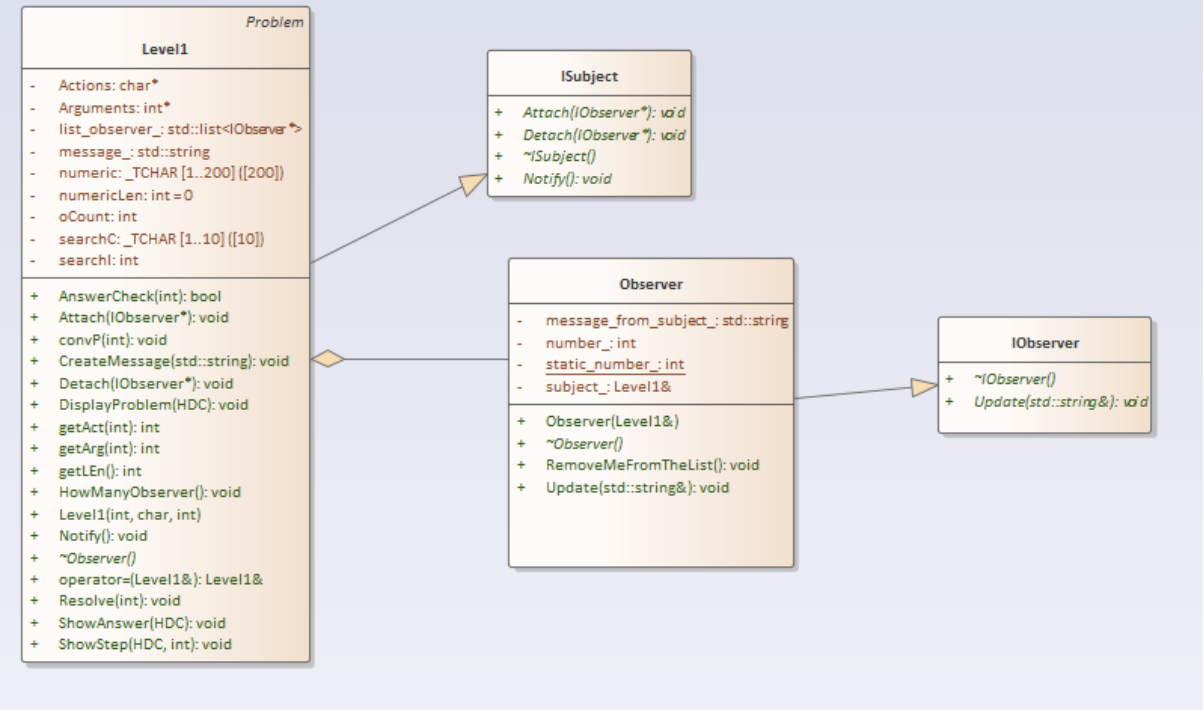


Рисунок 8. Диаграмма наблюдателя.

Участники

ISubject: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)

Level1: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.

IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.

Observer: конкретная реализация интерфейса IObserver.

3.4. Создание самого примера

Инициализация самого примера требует изменения значительного объёма данных. То есть можно сказать, что необходим сложный конструктор, на что естественно напрашивается Строитель. Ну и чтобы он не терялся, давайте сделаем одиночкой.

3.4.1 Строитель

Строитель (Builder) - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Когда использовать паттерн Строитель?

Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой

Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания

Формально в UML паттерн мог бы выглядеть следующим образом:

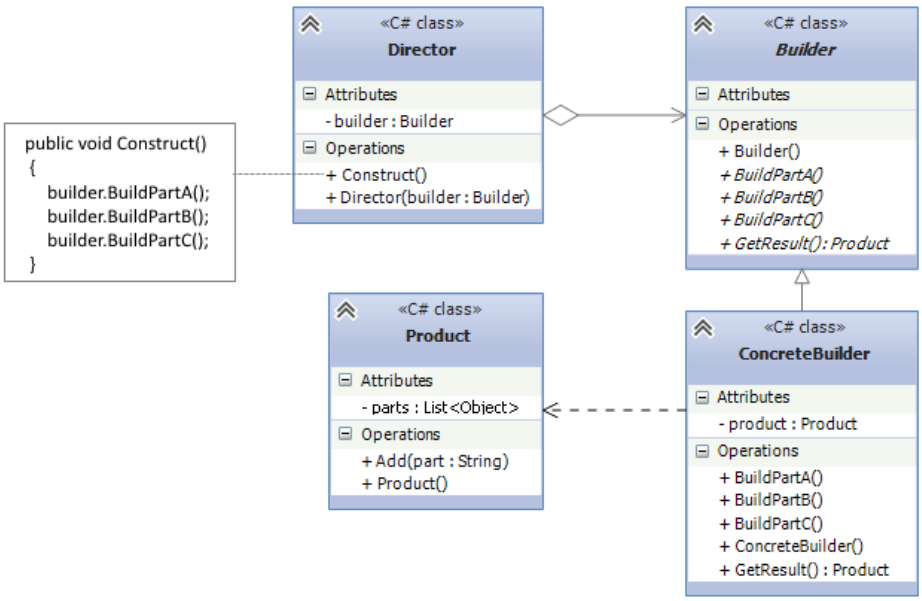


Рисунок 9. Диаграмма строителя.

Участники

Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.

Builder: определяет интерфейс для создания различных частей объекта Product

ConcreteBuilder: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему

Director: распорядитель - создает объект, используя объекты Builder

В моей интерпретации Строитель немного отличается, по частично уже озвученным ранее причинам.

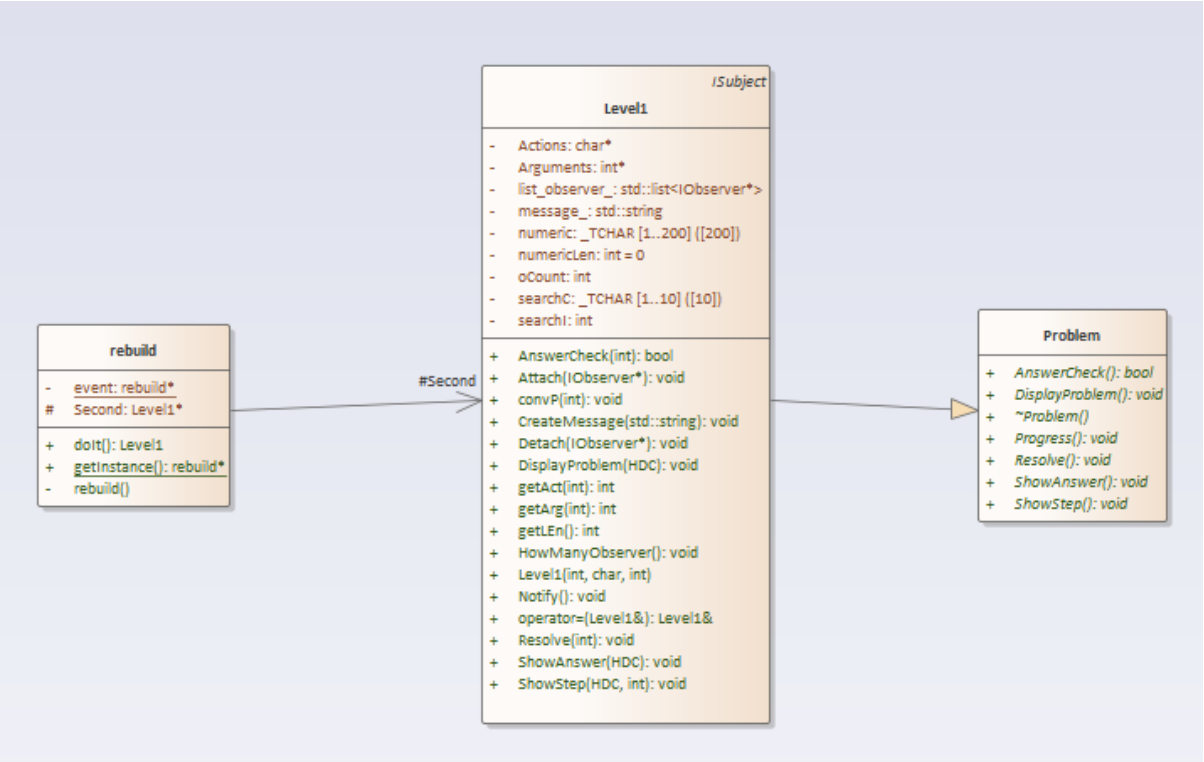


Рисунок 10. Диаграмма Строителя-одиночки.

Здесь нет клиента и интерфейса строителя, по причине общих соображений, а комментировать особо и нечего.

Также если обратить внимание на самого строителя, то можно заметить реализацию в нём одиночки, основная причина для этого то, что строитель вызывается в нескольких местах и при этом затрагивает глобальные переменные, а в будущем ещё и поток данных.

3.4.1 Одиночка

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр.

Участниками данного шаблона обычно является сам одиночка и ещё возможен клиент, вызывающий его.

4. Архитектура приложения в виде диаграмм uml

Диаграмма – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей). Диаграммы рисуются для визуализации системы с различных точек зрения, поэтому отдельная диаграмма – это проекция системы. Для всех систем, кроме самых тривиальных, диаграмма представляет собой ограниченный взгляд на элементы, составляющие систему. Один и тот же элемент может появляться либо во всех диаграммах, либо в некоторых (наиболее частый случай), либо вообще ни в одной (очень редкий случай). Теоретически диаграмма может включать в себя любую комбинацию сущностей и связей. На практике, однако, используется лишь небольшое число общих комбинаций, состоящих из пяти наиболее часто применяемых представлений архитектуры программных систем. UML включает в себя следующие типы диаграмм:

1.Диаграмма прецедентов.

2.Диаграмма взаимодействия.

2.1 Диаграмма последовательности.

2.2 Диаграмма кооперaций.

3. Диаграмма классов.

4. Диаграмма поведения.

4.1 Диаграмма состояний.

4.2 Диаграмма деятельности.

5. Диаграмма компонентов.

6. Диаграмма развёртывания.

4.1 Диаграмма вариантов использования

Визуальное моделирование в UML можно представить как некоторый процесс поуровневого спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует цели:

Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.

Сформулировать общие требования к функциональному поведению проектируемой системы.

Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.

Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

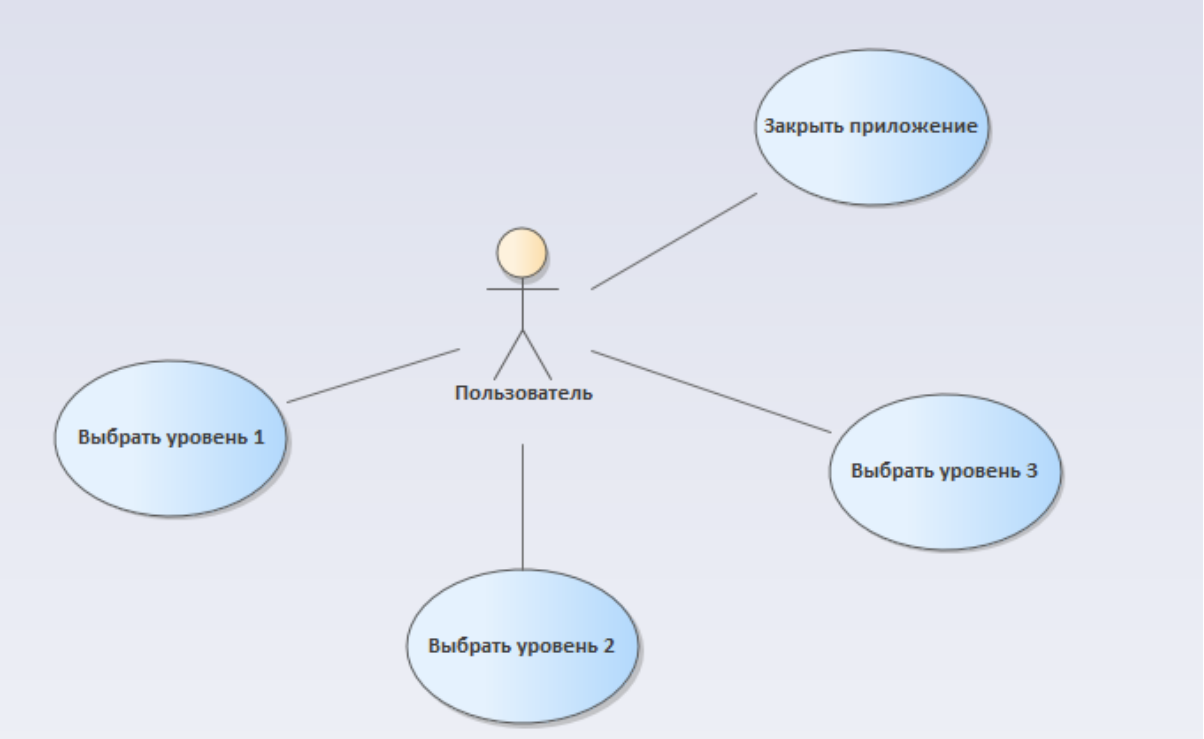


Рисунок 11. Диаграмма вариантов использования для приложения.

На данной диаграмме отображены возможные действия пользователя после входа в приложение, в рамках самого приложения у пользователя больше действий нет, но они есть в рамках уровня.

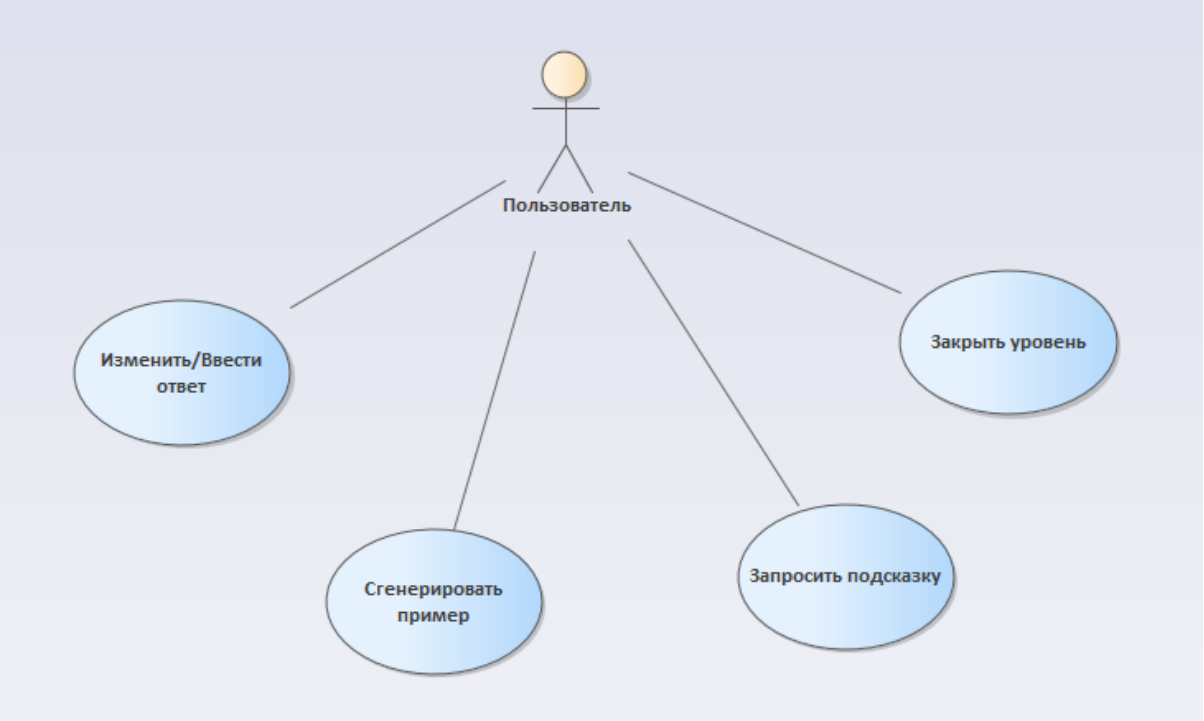


Рисунок 12. Диаграмма вариантов использования для уровня.

4.2 Диаграмма последовательности

Одной из характерных особенностей систем различной природы и назначения является взаимодействие между собой отдельных элементов, из которых образованы эти системы. Речь идет о том, что различные составные элементы систем не существуют изолированно, а оказывают 'определенное влияние друг на друга, что и отличает систему как целостное образование от простой совокупности элементов.

В языке UML взаимодействие элементов рассматривается в информационном аспекте их коммуникации, т. е. взаимодействующие объекты обмениваются между собой некоторой информацией. При этом информация принимает форму законченных сообщений. Другими словами, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать направленное влияние на своего получателя. Это полностью согласуется с принципами ООАП, когда любые виды информационного взаимодействия между элементами системы должны быть сведены к отправке и приему сообщений между ними.

Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы взаимодействия. Говоря об этих диаграммах, имеют в виду два аспекта взаимодействия. Во-первых, взаимодействия объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма

последовательности. Этот вид канонических диаграмм является предметом изучения настоящей главы.

Ранее, при изучении диаграмм состояния и деятельности, было отмечено одно немаловажное обстоятельство. Хотя рассмотренные диаграммы и используются для спецификации динамики поведения систем, время в явном виде в них не присутствует. Однако временной аспект поведения может иметь существенное значение при моделировании синхронных процессов, описывающих взаимодействия объектов. Именно для этой цели в языке UML используются диаграммы последовательности.

Во-вторых, можно рассматривать структурные особенности взаимодействия объектов. Для представления структурных особенностей передачи и приема сообщений между объектами используется диаграмма кооперации.

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения. Одно - слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии.

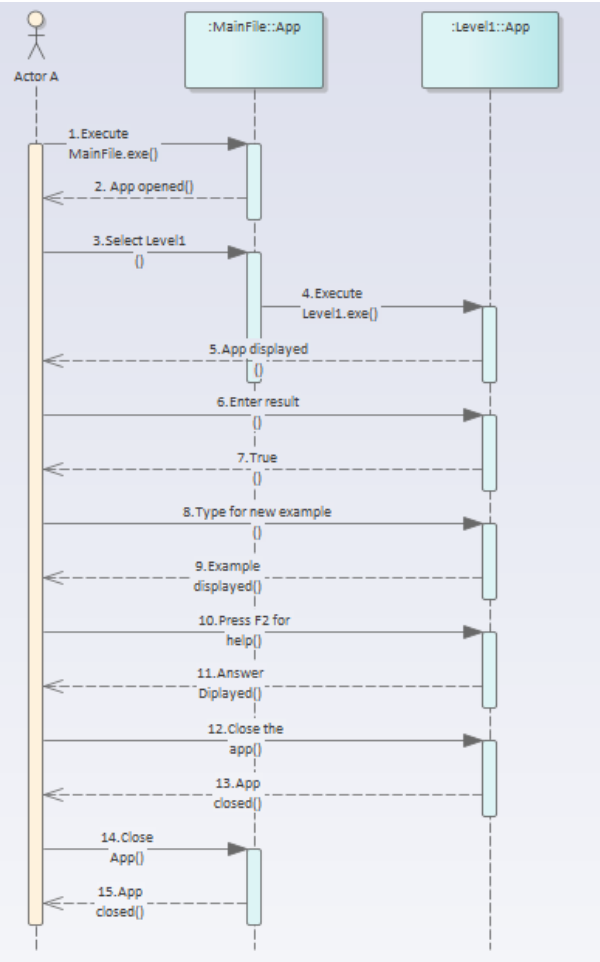


Рисунок 13. Диаграмма последовательности.

4.3 Диаграмма компонентов

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. .Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Диаграмма компонентов разрабатывается для следующих целей:

Визуализации общей структуры исходного кода программной системы.

Спецификации исполнимого варианта программной системы.

Обеспечения многократного использования отдельных фрагментов программного кода.

Представления концептуальной и физической схем баз данных.

Для представления физических сущностей в языке UML применяется специальный термин - компонент (component). Компонент реализует некоторый набор интерфейсов и служит для общего обозначения элементов физического представления модели. Для графического представления компонента может использоваться специальный символ - прямоугольник со вставленными слева двумя более мелкими прямоугольниками.

Компоненты могут иметь следующие стандартные стереотипы:

1) «file» – любой файл, кроме таблицы:
2) «executable» – программа (исполняемый файл);
3) «library» – статическая или динамическая библиотека;
4) «source» – файл с исходным текстом программы;
5) «document» – остальные файлы (например, файл справки);
6) «table» – таблица базы данных.

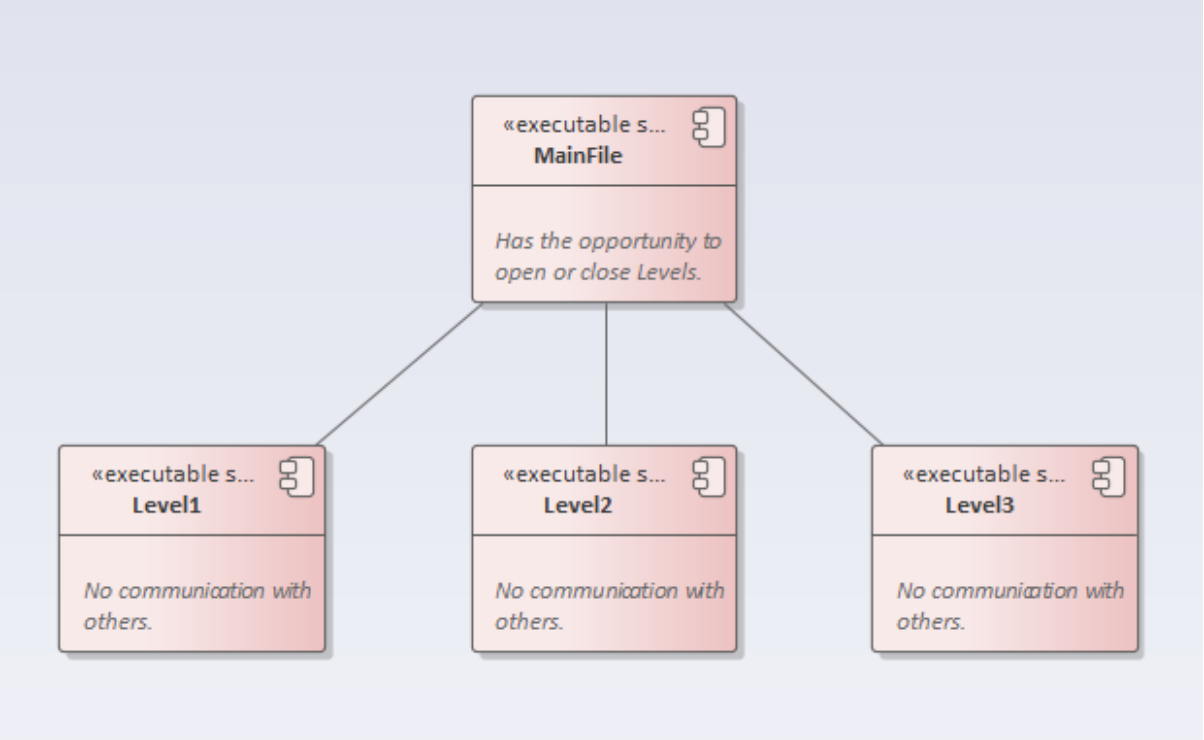


Рисунок 14. Диаграмма компонентов.

На данный момент все объекты - это четыре исполнительных файла, в будущем планируются ещё текстовые файлы.

4.4 Диаграмма развёртывания

Физическое представление программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Конечно, если разрабатывается простая программа, которая может выполняться локально на компьютере пользователя, не задействуя никаких периферийных устройств и ресурсов, то в этом случае нет необходимости в разработке дополнительных диаграмм. Однако при разработке корпоративных приложений ситуация представляется совсем по-другому.

Во-первых, сложные программные системы могут реализовываться в сетевом варианте на различных вычислительных платформах и технологиях доступа к распределенным базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач по рациональному размещению компонентов по узлам этой сети, что определяет общую производительность программной системы.

Во-вторых, интеграция программной системы с Интернетом определяет необходимость решения дополнительных вопросов при проектировании системы, таких как обеспечение безопасности, криптозащищенности и устойчивости доступа к информации для корпоративных клиентов. Эти аспекты в немалой степени зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных.

Наконец, технологии доступа и манипулирования данными в рамках общей схемы "клиент-сервер" также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом. Эти аспекты также требуют визуального представления с целью спецификации программных и технологических особенностей реализации распределенных архитектур.

Как было отмечено в главе 10, первой из диаграмм физического представления является диаграмма компонентов. Второй формой физического представления программной системы является диаграмма развертывания (синоним - диаграмма размещения). Она применяется для представления общей конфигурации и топологии распределенной программной системы и содержит распределение компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений - маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе ее исполнения (runtime). При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единой для системы в целом, поскольку должна всецело отражать особенности ее реализации. Эта диаграмма, по сути, завершает процесс ООАП для конкретной программной системы и ее разработка, как правило, является последним этапом спецификации модели.

Итак, перечислим цели, преследуемые при разработке диаграммы развертывания:

Определить распределение компонентов системы по ее физическим узлам.

Показать физические связи между всеми узлами реализации системы на этапе ее исполнения.

Выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Узел (node) представляет собой некоторый физически существующий элемент системы, обладающий некоторым вычислительным ресурсом. В качестве вычислительного ресурса узла может рассматриваться наличие по меньшей мере некоторого объема электронной или магнитооптической памяти и/или процессора. В последней версии языка UML понятие узла расширено и может включать в себя не только вычислительные устройства (процессоры), но и другие механические или электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы.

Возможность включения людей (персонала) в понятие узла позволяет создавать средствами языка UML модели самых различных систем, включая бизнес-процессы и технические комплексы. Действительно, реализация бизнес-логики предприятия требует рассматривать в качестве узлов системы организационные подразделения, состоящие из персонала. Автоматизация управления техническими комплексами также требует рассмотрения в качестве самостоятельного элемента человека-оператора, способного принимать решения в нештатных ситуациях и нести ответственность за возможные последствия этих решений.

Графически на диаграмме развертывания узел изображается в форме трехмерного куба (строго говоря, псевдотрехмерного прямоугольного параллелепипеда). Узел имеет собственное имя, которое указывается внутри этого графического символа.

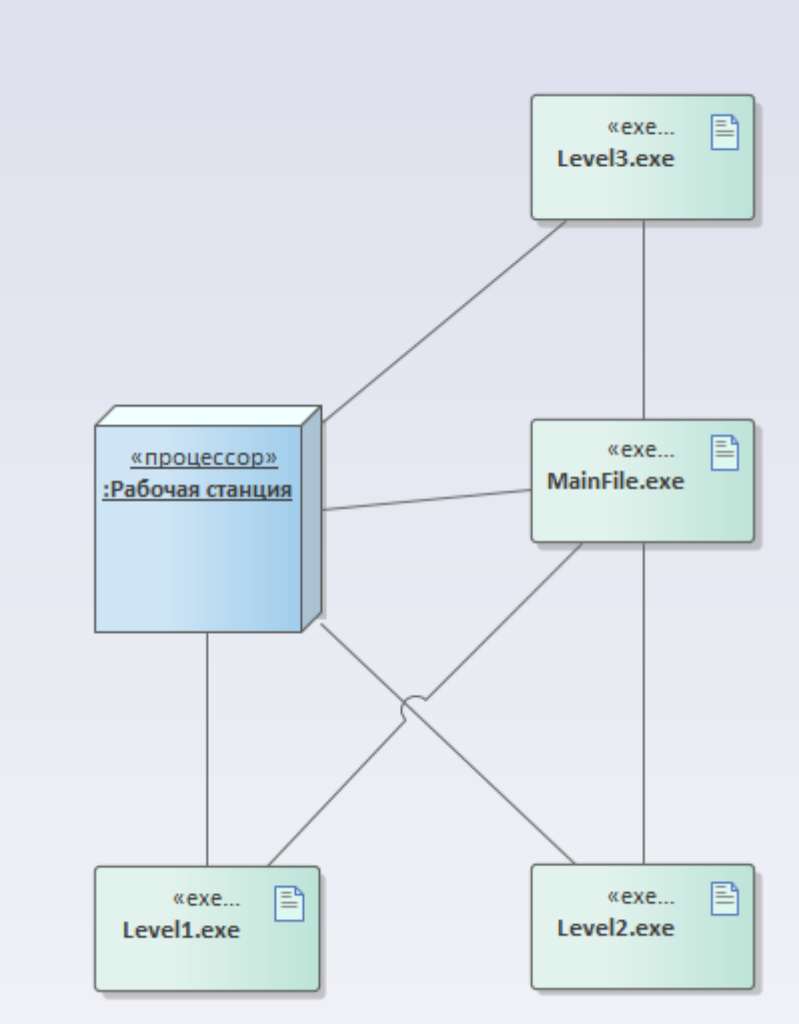


Рисунок 15. Диаграмма развёртывания.

Заключение

Создания приложения - это всегда увлекательный и непредсказуемый процесс, если вы не занимаетесь штамповкой. Не важно какая была изначально цель, в пути её достижения очень многое может измениться, так порой некоторые успешные проекты в индустрии фактически не имеют ни чего общего с тем, что планировалось изначально. Этот аспект придаёт всей профессии долю романтики и духа приключений, но работа есть работа, и здесь всё же должна присутствовать стабильность. И раз уж душа не лежит клепать однотипные приложения, то необходимо хоть как-то внести стабильность в процесс разработки. Вот здесь и появляются все возможные шаблоны и принципы программирования типа S.O.L.I.D.

Они представляют собой общепринятые нормы и правила, зная которые можно попытаться понять и даже модернизировать чужой поток сознания воплотившийся в программном коде.

Изучение и обязательное применение шаблонов учит студентов некоей дисциплине в рамках написания кода. Так даже самые ярые противники структуризации и любители беспорядочного хаоса вынуждены воплощать в своём творении общепринятые нормы и стандарты, а после видя и, сто более важно, осознавая их удобство, начнут использовать правила хорошего тона программирования и при решении обычных, порою самых простых задач.

Для меня лично данный курсовой проект оказался в некотором роде испытанием, хотя сам по себе он достаточном лёгок, да и если подсчитать затраченное на него время, то едва ли выйдет больше пары суток, но как говорится обстоятельства сильнее нас. Данная работа требует хотя бы минимальных усилий с точки зрения абстрактного мышления и фантазии, и если у человека возникают трудности с одним из этих пунктов, то он вынужден в некоем роде превозмогать и выполнять действия прикладывая чрезмерные усилия, что в прочем не поможет достичь хорошего результата. В принципе именно так я и оцениваю свою работу, она сделана, но с точки зрения шаблонов проектирования и факта курсовой работы в целом, сделана так себе, я бы сказал слабенько, за то работает, разве что.

В ходе выполнения данной работы я повысил свои навыки в рамках ООП и смог осознать фронт работ предстоящий мне на дипломную работу. В целом считаю данное задания важным к выполнению студентами, поскольку оно заставляет внедрять чужие пусть и не наработки, но мысли в собственное приложение, заставляет стыковать всё это вместе, приводить к единому формату. Наиболее точно, мне кажется, это можно сравнить с работой в команде, где передо мной как разработчиком стоит цель - использование ряда инструментов и методик, и каким бы не было решение к которому я приду, оно должно содержать эти пункты.

Библиография

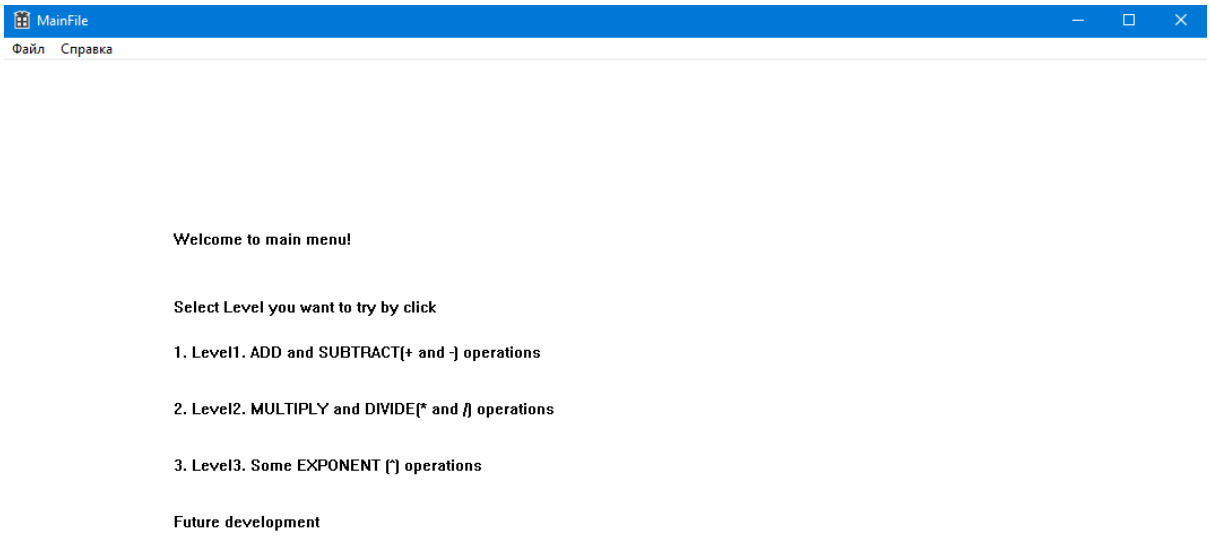
1. Леоненков А.В., Самоучитель UML. – 2-е изд.. – СПб.:БХВ–Петербург, 2004. – 432 с.: ил
2. Руководство по языкам программирования [Электронный ресурс] – Режим доступа: https://metanit.com/

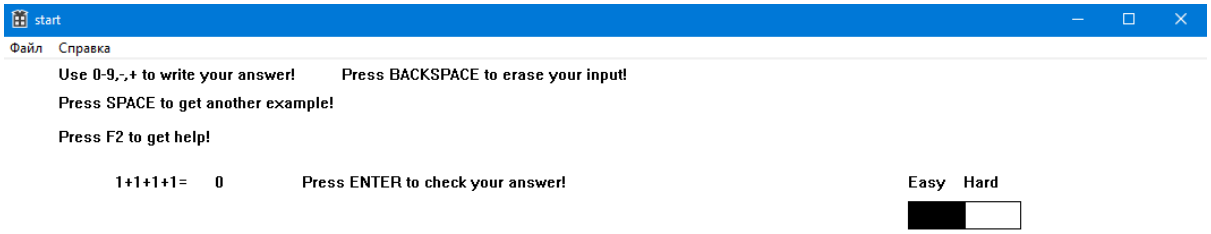
Приложение А

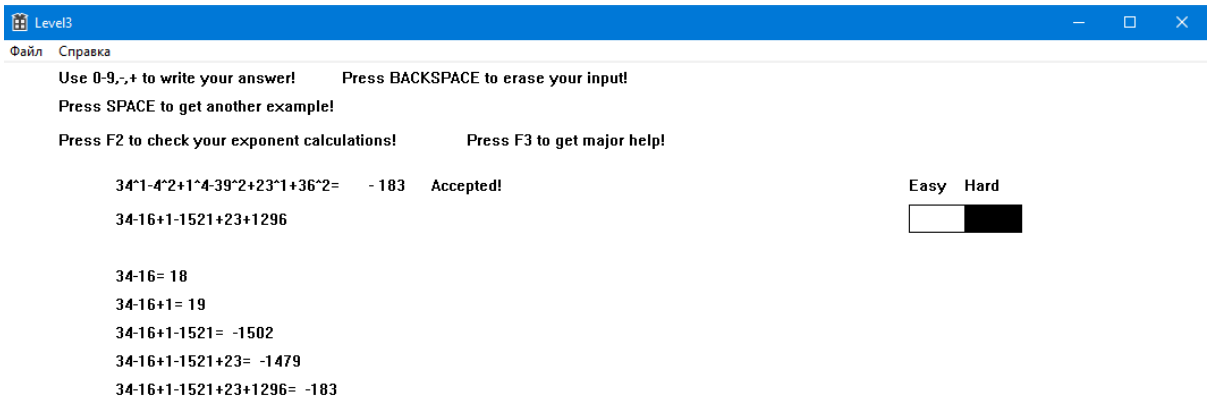
Ссылка на репозиторий проекта: https://github.com/VadimVaca/TMPS

Приложение В

Несколько скриншотов работающего приложения




start
Файл Справка
Use 0-9,-,+ to write your answer! Press BACKSPACE to erase your input!
Press SPACE to get another example!
Press F2 to get help!
1+1+1+1= 0 Press ENTER to check your answer!
Easy Hard


Level3
Файл Справка
Use 0-9,-,+ to write your answer! Press BACKSPACE to erase your input!
Press SPACE to get another example!
Press F2 to check your exponent calculations! Press F3 to get major help!
34^1-4^2+1^4-39^2+23^1+36^2= -183 Accepted!
Easy Hard
34-16+1-1521+23+1296
34-16= 18
34-16+1= 19
34-16+1-1521= -1502
34-16+1-1521+23= -1479
34-16+1-1521+23+1296= -183