ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Д.09.03.04-ИИ.22.20/5384.КП

***Кафедра*** прикладной математики и

искусственного интеллекта

КУРСОВОЙ ПРОЕКТ

по дисциплине: «Архитектура и проектирование программного обеспечения»

Тема: «Шаблон проектирования Proxy для нотации BFT»

Руководитель:

\_\_\_\_\_\_\_\_\_\_\_\_\_ ст. преп. О.А. Гудаев

(дата, подпись)

Нормоконтроль:

\_\_\_\_\_\_\_\_\_\_\_\_\_ асс. Н.П. Пулинец

(дата, подпись)

Исполнитель:

\_\_\_\_\_\_\_\_\_\_\_\_ ст.гр. ПИ-20г Ващенко Д.И.

(дата, подпись)

Донецк – 2022

**ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**Факультет:** **Интеллектуальных систем и программирования**

**Направление: Программная инженерия**

**Профиль: Технология программного обеспечения интеллектуальных систем**

**Кафедра:** «**Прикладной математики и искусственного интеллекта»**

ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ

по дисциплине «Архитектура и проектирование программного обеспечения»

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студенту | Ващенко Данилу Игоревичу | | | | | | | группы | ПИ-20г |
|  | (фамилия, имя, отчество) | | | | | | |  |  |
|  |  | | | | | | |  |  |
| Тема проекта | Шаблон проектирования Proxy для нотации BFT | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
| Исходные данные к проекту | | | | Спецификация шаблонов проектирования, специфи- | | | | | |
| кация графического языка Unified Modeling Language, спецификация | | | | | | | | | |
| нотации Brain Fogic Tag | | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
| Перечень искомых результатов | | | | | Диаграмма классов, диаграмма последовательности, | | | | |
| диаграмма компонентов, диаграмма состояний, диаграмма вариантов  использования, листинг реализации шаблона | | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
|  | | | | | | | | | |
| Рекомендуемая литература | | | Крэг Ларман. Применение UML 2.0 и шаблонов проек- | | | | | | |
| тирования = Applying UML and Patterns : An Introduction to Object-Oriented Analysis and | | | | | | | | | |
| Design and Iterative Development. – 3-е изд. – М.: [Вильямс](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1" \o "Вильямс (издательство) (страница отсутствует)), 2006. – 736 с. | | | | | | | | | |
|  | | | | | | | | | |
| Дата выдачи задания | | 08.11.2022 | | | |  |  | | |
|  | |  | | | |  |  | | |
| Дата защиты проекта | | 29.12.2022 | | | |  |  | | |
|  | |  | | | |  |  | | |
| **Руководитель** | |  | | | |  | ст. преп. О.А. Гудаев | | |
|  | | (подпись) | | | |  | (должность, Ф.И.О.) | | |
|  | |  | | | |  | асс. Н.П. Пулинец | | |
|  | | (подпись) | | | |  | (должность, Ф.И.О.) | | |
| **Разработчик** | |  | | | |  | Д.И.Ващенко | | |
|  | | (подпись) | | | |  | (Ф.И.О) | | |

РЕФЕРАТ

Пояснительная записка: 53 с., 18 рис., 3 табл., 6 источников, 2 прил.

Курсовая работа рассматривает разработку программного обеспечения на основе паттернов объектно-ориентированного программирования (ООП). Архитектура разрабатываемой программы строится с использованием шаблона Proxy (заместитель).

**Паттерн "**Заместитель**" (**Proxy**) предоставляет из себя** структурный паттерн проектирования, перехватывающий вызов оригинального объекта, и выполняющие некоторые полезные действия **(одно из применений данного паттерна будет реализовано, для более наглядной его работы).**

**Нотация BFT (BrainFogicTag) – описание связанных данных, дополняющих контекст естественного текста.**

Для построения диаграмм используется язык диаграмм Unified Modelling Language (UML). Для реализации проекта используется язык программирования высокого уровня C#.

Целью работы является демонстрация паттерна на примере создания парсера нотации BFT.

В ходе проектирования получены следующие результаты: исходный код протоколов классов на языке программирования C# и UML-диаграмма классов.

Для тестирования классов на языке высокого уровня C# была использована среда разработки «Microsoft Visual Studio». Для создания диаграмм была использована веб-приложение «Draw.io».

ООП, АРХИТЕКТУРА, ПРОЕКТИРОВАНИЕ, C#, BFT, ПАТТЕРН, UML, БЛОК-СХЕМА, КЛАСС, НАСЛЕДОВАНИЕ, ИНТЕРФЕЙС, ШАБЛОН, PROXY, ИНКАПСУЛЯЦИЯ, ЗАМЕСТИТЕЛЬ

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 6](#_Toc6892)

[1 ОСНОВЫ ШАБЛОННОГО ПОДХОДА ПРОЕКТИРОВАНИЯ ПО 8](#_Toc3902)

[1.1 Методы проектирования и программирования 8](#_Toc2723)

[1.2 Паттерны проектирования 13](#_Toc5144)

[1.3 Нотация Brain Fogic Tag 17](#_Toc14013)

[2 ХАРАКТЕРИСТИКИ ШАБЛОНА ПРОЕКТИРОВАНИЯ Abstract Factory 20](#_Toc15345)

[2.1 Общие сведения 20](#_Toc9822)

[2.2 Структура шаблона Abstract Factory 20](#_Toc29862)

[3 РАЗРАБОТКА UML ДИАГРАММ ДЛЯ НОТАЦИИ BFT 22](#_Toc11794)

[3.1 Диаграмма состояний 23](#_Toc28210)

[3.2 Диаграмма последовательности 26](#_Toc7452)

[3.3 Диаграмма компонентов 28](#_Toc25906)

[3.4 Диаграмма вариантов использования 30](#_Toc26208)

[4 РАЗРАБОТКА ПРОГРАММЫ НА ОСНОВЕ ШАБЛОНА Abstract Factory 33](#_Toc3765)

[4.1 Диаграммы классов 33](#_Toc10631)

[4.1.1 Общие сведения 33](#_Toc5962)

[4.1.2 Диаграмма классов проектируемой системы 35](#_Toc26186)

[5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЧЕНИЯ 37](#_Toc30624)

[ЗАКЛЮЧЕНИЕ 42](#_Toc13063)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 43](#_Toc13131)

[Приложение А ТЕХНИЧЕСКОЕ ЗАДАНИЕ 44](#_Toc22795)

[А.1 Общие сведения 44](#_Toc31318)

[А.2 Назначение и цели создания проекта 45](#_Toc3136)

[А.3 Требования к КП 45](#_Toc11475)

[А.4 Требования к задачам КП 45](#_Toc24853)

[Приложение Б ЛИСТИНГ ПРОГРАММНЫХ МОДУЛЕЙ 47](#_Toc30605)

## ВВЕДЕНИЕ

Когда же при разработке нужны паттерны? Зачастую, на стадии создания программных систем у разработчиков возникает проблема выбора тех или иных проектных решений. В этих случаях как раз и нужны паттерны. Паттерн проектирования – это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

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

Для разработки и проектирования структуры создаваемого проекта используется графический язык UML. Язык UML предназначен для графического описания объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

Для создания шаблона был выбран язык высокого уровня C#.

Язык C# современный объектно-ориентированный и типобезопасный язык программирования. C# позволяет разработчикам создавать разные типы безопасных и надежных приложений, выполняющихся в .NET. Важная особенность таких компонентов – это модель программирования на основе свойств, методов и событий. Каждый компонент имеет атрибуты, предоставляющие декларативные сведения о компоненте, а также встроенные элементы документации. Язык программирования C# предоставляет языковые конструкции, непосредственно поддерживающие такую концепцию работы. Благодаря этому язык C# отлично подходит для создания и применения программных компонентов.

Программы на языке C# выполняются на платформе «.NET Framework», встроенном компоненте ОС Windows, которая включает виртуальную систему выполнения, называемую поддержкой общеязыковой среды выполнения (CLR), и унифицированный набор библиотек классов.

В данном курсовом проекте предполагается спроектировать программу для работы с нотацией BFT (Brain Fogic Tag) средствами графического языка UML и ряда поддерживаемых им диаграмм (диаграммы классов, компонентов, вариантов использования, состояний и последовательности), а также с применением шаблона проектирования Proxy.

В ходе реализации курсового проекта была рассмотрена нотация BFT (Brain Fogic Tag). Модель гармонична и проста: блоки тегов, на первый взгляд, произвольно разбросаны по тексту, образуют цепочки связанных данных и размечают близлежащие фразы текста. Осознавая, что чтение материалов затруднено, положительным и превалирующим моментом считается угождение клиповому мышлению. Клиповый формат знаний BFT детализирует текст семантическим контекстом, полезным при машинной обработке, но контекст не лишен смысла и при чтении.

UML **–** язык графического описания для объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов. Существует два основных типа диаграмм UML: структурные диаграммы и поведенческие диаграммы (а внутри этих категорий имеется много других). Эти варианты существуют для представления многочисленных типов сценариев и диаграмм, которые используют разные типы людей.

## 1 ОСНОВЫ ШАБЛОННОГО ПОДХОДА ПРОЕКТИРОВАНИЯ ПО

### Методы проектирования и программирования

В программировании имеется своя технология – это знания, правила, навыки и инструменты, позволяющие получать гарантированный качественный результат. Но само по себе соблюдение ряда правил не дает гарантию качества результата. Это объясняется спецификой программирования.

Технология программирования – это совокупность методов и средств для разработки программного обеспечения. В технологии должны быть определены последовательность выполнения операций, условия, при которых выполняется каждая операция, описание самих операций: исходные данные, нормативные документы, в том числе стандарты, критерии и методы оценки, результаты и др.

Наиболее известные из технологий:

* метод «северо-западного» угла (имеется в виду лист бумаги или экран дисплея), программа пишется сразу от начала до конца, без использования каких-либо общих принципов;
* технология структурного программирования, в ней предполагается придерживаться принципов модульности, нисходящего и пошагового проектирования программ, одновременного проектирования программ и структур данных;
* технология объектного программирования: связана с использованием при проектировании программы понятий объектов и их классов.

При изучении ООП наибольшей проблемой является использование новой терминологии и понимание нового подхода к решению старых задач – новой технологии программирования.

Простейший метод заключается в написании программы сразу от начала до конца, без использования каких-либо общих принципов.

#### 1.1.1 Парадигма программирования

Парадигма программирования – это совокупность принципов, методов и понятий, определяющих способ конструирования программ.

Парадигма задается использованием определенных сущностей, например:

* состояний программы и команд, изменяющих их (императивное программирование);
* математических функций без состояний (функциональное программирование);
* объектов и взаимодействий между ними (объектно-ориентированное программирование);
* алгоритмов и контейнеров, оперирующих с типами данных, переданными как параметр (обобщенное программирование);
* значений и операций, преобразующих значения (программирование на уровне значений), и т.д.

Следует отметить, что язык программирования не обязательно использует только одну парадигму. Языки, поддерживающие несколько парадигм, называются мультипарадигменными.

В технологии ООП взаимоотношения данных и алгоритма имеют более регулярный характер: во-первых, класс (базовое понятие этой технологии) объединяет в себе данные (структурированная переменная) и методы (функции). Во-вторых, схема взаимодействия функций и данных принципиально иная. Метод (функция), вызываемый для одного объекта, как правило, не вызывает другую функцию непосредственно. Для начала он должен иметь доступ к другому объекту (создать, получить указатель, использовать внутренний объект в текущем и т.д.), после чего он уже может вызвать для него один из известных методов. Таким образом, структура программы определяется взаимодействием объектов различных классов между собой. Как правило, имеет место иерархия классов, а технология ООП иначе может быть названа как программирование «от класса к классу».

Что первично: алгоритм (процедура, функция) или обрабатываемые им данные. В традиционной технологии программирования взаимоотношения процедуры – данные имеют более-менее свободный характер, причем процедуры (функции) являются ведущими в этой связке: как правило, функция вызывает функцию, передавая данные друг-другу по цепочке. Соответственно, технология структурного проектирования программ прежде всего уделяет внимание разработке алгоритма.

#### 1.1.2 Модульное программирование

Еще одной, но уже физической единицей программы является текстовый файл, содержащий некоторое количество функций и определений типов данных и переменных. Модульное программирование на уровне файлов – это возможность разделить полный текст программы на несколько файлов, транслировать их независимо друг от друга. Принцип модульности распространяется не только на программы, но и на данные: любой набор параметров, характеризующих логический или физический объект, должен быть представлен в программе в виде единой структуры данных (структурированной переменной). Олицетворением принципа модульности является библиотека стандартных функций. Она, как правило, обеспечивает полный набор параметризованных действий, используя общие структуры данных. Библиотеки представляют собой аналогичные Си-программы, независимо оттранслированные и помещенные в каталог библиотек.

#### 1.1.3 Нисходящее программирование

Способ разработки программ, при котором программирование ведется методом "сверху вниз", от общего к деталям. Алгоритм решения задачи разбивается на несколько более простых частей или подзадач. Их выделяют таким образом, чтобы,программирование подзадач было независимым. При этом составляют план решения всей задачи, пунктами которого и являются выделенные части. План записывают графически в виде блок-схемы, где определяют головную и подчиненные подзадачи и связи между ними, т. е. интерфейс. Здесь же устанавливают, какие начальные данные (или аргументы) получает каждая подзадача для правильного функционирования и какие результаты она выдает. По блок-схеме составляется программа, в которой содержатся вызовы подпрограмм (процедур или функций), соответствующих выделенным подзадачам.

Нисходящее проектирование программы заключается в том, что разработка идет от общей неформальной формулировки некоторого действия программы на естественном языке, «от общего к частному», к замене ее одной из трех формальных конструкций языка программирования:

* простой последовательности действий;
* конструкции выбора или оператора if;
* конструкции повторения или цикла.

В записи алгоритма это соответствует движению от внешней (объемлющей) конструкции к внутренней (вложенной). Эти конструкции также могут содержать в своих частях неформальное описание действий, то есть нисходящее проектирование по своей природе является пошаговым. Отметим основные свойства такого подхода:

* первоначально программа формулируется в виде некоторого неформального действия на естественном языке;
* первоначально определяются входные параметры и результат действия;
* очередной шаг детализации не меняет структуру программы, полученную на предыдущих шагах.

#### 1.1.4 Программирование при помощи ООП паттернов

Центральными в ООП являются понятия класса и объекта. Образно говоря, ООП заключается не столько в использовании классов и объектов в программе, сколько в замене принципа программирования «от функции к функции» принципом программирования «от класса к классу».

Технология ООП прежде всего накладывает ограничения на способы представления данных в программе. Любая программа отражает в них состояние физических предметов либо абстрактных понятий (назовем их объектами программирования), для работы с которыми она предназначена. В традиционной технологии варианты представления данных могут быть разными. В худшем случае программист может «равномерно размазать» данные о некотором объекте программирования по всей программе.

В противоположность этому все данные об объекте программирования и его связях с другими объектами можно объединить в одну структурированную переменную. В первом приближении ее можно назвать объектом. Кроме того, с объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования это функции, получающие в качестве обязательного параметра указатель на объект. Технология ООП запрещает работать с объектом иначе, чем через методы, то есть внутренняя структура объекта скрыта от внешнего пользователя. Описание множества однотипных объектов называется классом.

Объект – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс – это описание множества таких объектов и выполняемых над ними действий.

### 1.2 Паттерны проектирования

Шаблон проектирования или паттерн в [разработке программного обеспечения](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F" \o "Проектирование программного обеспечения)  – повторяемая [архитектурная конструкция](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F" \o "Архитектура программного обеспечения), представляющая собой решение проблемы [проектирования](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Проектирование) в рамках некоторого часто возникающего [контекста](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82" \o "Контекст).

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в [код](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0" \o "Компьютерная программа); это лишь пример решения задачи, который можно использовать в различных ситуациях. [Объектно-ориентированные](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Объектно-ориентированное программирование) шаблоны показывают отношения и [взаимодействия](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Взаимодействие) между [классами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Класс (программирование)) или [объектами](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Объект (программирование)), без определения того, какие конечные классы или объекты приложения будут использоваться.

Существуют несколько типов паттернов проектирования, каждый из которых предназначен для решения своего круга задач:

* порождающие паттерны, предназначенные для создания новых объектов в системе;
* структурные паттерны, решающие задачи компоновки системы на основе классов и объектов;
* паттерны поведения, предназначенные для распределения обязанностей между объектами в системе.

Что же дает нам применение паттернов. При написании программ мы можем формализовать проблему в виде классов и объектов, и связей между ними. И применить один из существующих паттернов для ее решения. В итоге нам не надо ничего придумывать. У нас уже есть готовый шаблон, и нам только надо его применить в конкретной программе.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Причем паттерны, как правило, не зависят от языка программирования. Их принципы применения будут аналогичны и в C#, и в Java, и в других языках.

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

* проблема, которую решает паттерн;
* мотивации к решению проблемы способом, который предлагает паттерн;
* структуры классов, составляющих решение;
* примера на одном из языков программирования;
* особенностей реализации в различных контекстах;
* связей с другими паттернами.

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

Также мышление паттернами упрощает групповую разработку программ. Зная применяемый паттерн проектирования и его основные принципы другому программисту будет проще понять его реализацию и использовать ее.

Вы можете вполне успешно работать, не зная ни одного паттерна. Более того, вы могли уже не раз реализовать какой-то из паттернов, даже не подозревая об этом.

В то же время не стоит применять паттерны ради самих паттернов. Хорошая программа предполагает использование паттернов. Однако не всегда паттерны упрощают и улучшают программу. Неоправданное их использование может привести к усложнению программного кода, уменьшению его качества. Паттерн должен быть оправданным и эффективным способом решения проблемы.

Существует множество различных паттернов, которые решают разные проблемы и выполняют различные задачи. Но по своему действию их можно объединить в ряд групп. В основу классификации основных паттернов положена цель или задачи, которые определенный паттерн выполняет.

Порождающие паттерны – это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов.

Другая группа паттернов, структурные паттерны, рассматривает, как классы и объекты образуют более крупные структуры – более сложные по характеру классы и объекты.

Третья группа паттернов называются поведенческими – они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение.

Таблица 1.1 – Основные группы паттернов

|  |  |  |
| --- | --- | --- |
| Порождающие | Структурные | Поведенческие |
| Абстрактная фабрика (Abstract Factory) | Приспособленец (Flyweight) | Цепочка обязанностей (Chain of responsibility) |
| Фабричный метод (Factory Method) | Компоновщик (Composite) | Шаблонный метод (Template method) |
| Строитель (Builder) | Мост (Bridge) | Посредник (Mediator) |
| Прототип (Prototype) | Декоратор (Decorator) | Интерпретатор (Interpreter) |
| Одиночка (Singleton) | Фасад (Facade) | Посредник (Mediator) |
|  | Заместитель (Proxy) | Хранитель (Memento) |
|  | Адаптер (Adapter) | Наблюдатель (Observer) |
|  |  | Состояние (State) |
|  |  | Стратегия (Strategy) |
|  |  | Команда (Command) |
|  |  | Итератор (Iterator) |

Существуют и остальныеклассификации паттернов в зависимости от того, относится паттерн к классам или объектам.

Паттерны классов описывают отношения между классами посредством наследования. Отношения между классами определяются на стадии компиляции. К таким паттернам относятся:

* фабричный метод (Factory Method);
* интерпретатор (Interpreter);
* шаблонный метод (Template Method);
* адаптер (Adapter).

Другая часть паттернов – паттерны объектов описывают отношения между объектами. Эти отношения возникают на этапе выполнения, поэтому обладают большей гибкостью. К паттернам объектов относят все шаблоны, перечисленные в таблице 1.1 за исключением тех паттернов, которые описывают отношения между классами посредством наследования.

Одни из них только начинают применяться, другие являются популярными на текущий момент, а некоторые уже менее распространены, чем раньше.

Прежде всего при решении какой-нибудь проблемы надо выделить все используемые сущности и связи между ними и абстрагировать их от конкретной ситуации. Затем надо посмотреть, вписывается ли абстрактная форма решения задачи в определенный паттерн. Например, суть решаемой задачи может состоять в создании новых объектов. В этом случае, возможно, стоит посмотреть на порождающие паттерны. Причем лучше не сразу взять какой-то определенный паттерн – первый, который показался нужным, а посмотреть на несколько родственных паттернов из одной группы, которые решают одну и ту же задачу.

При этом важно понимать смысл и назначение паттерна, явно представлять его абстрактную организацию и его возможные конкретные реализации. Один паттерн может иметь различные реализации. Не стоит использовать паттерн, если вы его не понимаете, даже если он на первый взгляд поможет в решении задачи.

И в конечном счете надо придерживаться принципа KISS (Keep It Simple, Stupid) – сохранять код программы по возможности простым и ясным. Ведь смысл паттернов не в усложнении кода программы, а наоборот в его упрощении.

Все паттерны проектирования удобно изображать при помощи UML диаграмм. Основные логические и физические отношения между ними определяются при помощи определенных графических элементов.

Отношения между классами:

Символhttps://habrastorage.org/getpro/habr/post_images/ca8/dca/2a5/ca8dca2a537a1ec8044e88984c3f8b02.png – агрегация (aggregation) – описывает связь «часть» – «целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».

Символ – композиция (composition) – подвид агрегации, в которой «части» не могут существовать отдельно от «целого».

Символ https://habrastorage.org/getpro/habr/post_images/43b/4e7/9ac/43b4e79ac9d10a4f8b57859b019c7c24.png – зависимость (dependency) – изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.

Символ https://habrastorage.org/getpro/habr/post_images/aa6/efe/a5f/aa6efea5f552569c7fa6ce4e5603e684.png – обобщение (generalization) – отношение наследования или реализации интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

### 1.3 Нотация Brain Fogic Tag

Нотация BFT представляет собой модель разметки данных в тексте на естественном языке. Нотация BFT реализуется вкраплениями в текст определенных тегов вроде «1[.]», «0[0]» которые образуют собой сетевую модель данных в отличие от языков разметок JSON и XML.

Нотация BFT, не нарушая читабельности естественного текста добавляет в него читаемые для человека и машины метаданные, которые в последствии возможно обработать. Список тегов нотации BFT приведен в таблице 1.2 с их аналогами представления данных в некоторых языках программирования.

Общей целью создания искусственной нотации BFT является улучшение коммуникации между носителями различных формальных систем.

Интегральные характеристики такого процесса коммуникации: неоднозначность понимания констант формальной системы; трудность перевода между искусственными языками, процессов вывода; сложность описания процесса конструирования формальных выражений. Все указанные проблемы порождают противоречия в интерпретации результатов, полученных автоматически. Естественный текст не содержит таких трудностей – он всегда неоднозначен для параллельных конструкций и имеет не точную, а достоверную интерпретацию.

Таблица 1.2 – Список тегов нотации BFT и их аналоги константных имен

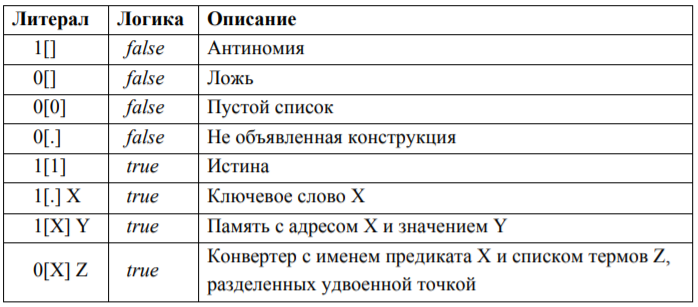
|  |  |
| --- | --- |
| Тег | Существующие аналоги, применяемые в компьютерной науке |
| 1[] | Абсурд, парадокс, не отношение, нет решения, ошибка (ложь) |
| 0[] | Ложь |
| 0[0] | Нуль, ноль, пусто, ничего, ни один, стоп (ложь) |
| 0[.] | Неопределенность, не присвоенный (ложь) |
| 1[1] | Истина |
| 1[.] value 0[0] | Тэг, символ, константное имя, объявить (истина) / закончить 0[0] |
| 1[label] value 0[0] | Ячейка памяти (истина) / конец 0[0] |
| 0[label] value 0[0] | Предикат, перевод значения в связи |

Основными применяемыми для разметки тегами являются:

* тег «1[.]value0[0]» обозначает какую-либо константную переменную;
* переменная «1[label]value0[0]» в виде «ключ: значение»;
* связь «0[label]value1..value2..valueN0[0]» семантика данного тега заключается в представлении связи (логической или смысловой) между какими либо использованными ранее переменными.

Также в таблице 1.3 приведен список базовых конструкций нотации BFT.

Таблица 1.3 – Список базовых конструкций метаязыка BFT



Перечисляя основные цели расширения текста искусственной нотацией: во-первых, занять нишу простого описания онтологий контента Интернета, когда, следуя хронологической траектории, развивается формат XML и JSON, переходя от формата JSON-LD к нотации BFT, во-вторых, предложить формат вкраплений в текст на естественном языке связанных онтологических данных, в-третьих, протоколировать автоматическое доказательство теорем выразительным описанием процесса принятия решений в интегральных системах, когда прозрачно комментируется конструирование компьютерных объектов по спецификации и в-четвёртых, не бороться с клиповым мышлением, а направить адекватное общение с робототехническими системами в интеллектуальное русло.

## 2 ХАРАКТЕРИСТИКИ ШАБЛОНА ПРОЕКТИРОВАНИЯ Abstract Factory

### 2.1 Общие сведения

Нужно определите интерфейс, который бы сделал заместитель и оригинальный объект взаимозаменяемыми. Затем создать класс заместителя, который должен содержать ссылку на объект. Также лучше использовать введение фабрики, которая решала бы, какой из объектов создавать — заместитель или реальный сервисный объект. На примере данной работы, было предложено использование абстрактной фабрики, согласно вышеописанным рекомендациям, в свою очередь было также принято решение о применение заместителя для реализация логирования (логирующий прокси), который отлично подходит в тематику данной работы, и хорошо показывает одну из основных концепцию паттерна.

### 2.2 Структура шаблона Abstract Factory

С помощью диаграмм UML классическое представление паттерна Proxy можно представить следующим образом (см. рис. 2.1).

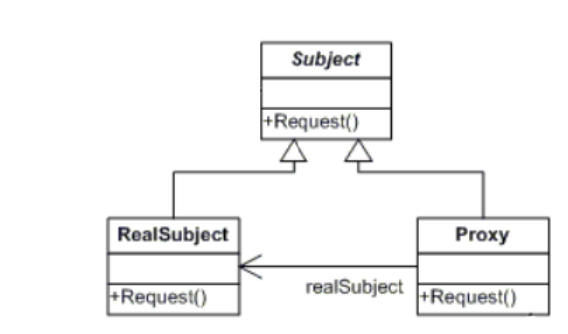


Рисунок 2.1 – Представление шаблона в UML

Участники:

* Абстрактные класс Subject определяют интерфейс для классов, объекты которых будут создаваться в программе.
* Конкретные класс RealSubject представляют конкретную реализацию абстрактных классов.
* Заместитель хранит ссылку на объект RealSubject.

В итоге применение паттерна Proxy имеет следующие преимущества:

* Позволяет контролировать сервисный объект незаметно для клиента;
* может контролировать жизненный цикл служебного объекта.

Имеет следующие минусы:

* Увеличивает время отклика от сервиса;
* усложняет код программы из-за введения дополнительных классов.

## 3 РАЗРАБОТКА UML ДИАГРАММ ДЛЯ НОТАЦИИ BFT

Язык UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов. Стандарт UML принятый OMG в 1997 г., предлагает следующий набор диаграмм для моделирования:

* диаграммы вариантов использования (use case diagrams) – для моделирования бизнес-процессов организации и требований к создаваемой системе;
* диаграммы классов (class diagrams) – для моделирования статической структуры классов системы и связей между ними;
* диаграммы поведения системы (behavior diagrams);
* диаграммы взаимодействия (interaction diagrams);
* диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams) – для моделирования процесса обмена сообщениями между объектами;
* диаграммы состояний (statechart diagrams) – для моделирования поведения объектов системы при переходе из одного состояния в другое;
* диаграммы деятельностей (activity diagrams) – для моделирования поведения системы в рамках различных вариантов использования, или моделирования деятельностей;
* диаграммы реализации (implementation diagrams);
* диаграммы компонентов (component diagrams) – для моделирования иерархии компонентов(подсистем) системы;
* диаграммы размещения (deployment diagrams) – для моделирования физической архитектуры системы.

Диаграмма в языке моделирования [UML](https://ru.wikipedia.org/wiki/UML" \o "UML) – наглядное представление некоей совокупности элементов модели системы в виде [графа](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Граф (математика)), на котором дуги (отношения) связывают вершины (сущности). В своём графическом виде различные виды диаграмм UML ([диаграммы классов](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2" \o "Диаграмма классов), [компонентов](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BE%D0%B2" \o "Диаграмма компонентов), [объектов](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BE%D0%B2" \o "Диаграмма объектов) и др.) применяются для визуализации разных аспектов устройства или поведения моделируемой системы.

### 3.1 Диаграмма состояний

#### 3.1.1 Общие сведения

Процесс начинается с начальной точки, затем следует самый первый переход в состояние «Проверка даты отчета». В поведении объекта в системе можно выделить действия, отображаемые переходами, и деятельности, отображаемые состояниями. Хотя и то, и другое – это процессы, реализуемые, как правило, некоторым методом класса «Отчет», они трактуются различным образом. Действия связаны с переходами и рассматриваются, как мгновенные и непрерываемые. Деятельности связаны с состояниями и могут длиться достаточно долго. Деятельность может быть прервана в результате наступления некоторого события.

Диаграмма состояний – это, по существу, диаграмма состояний из теории автоматов со стандартизированными условными обозначениями, которая может определять множество систем от компьютерных программ до бизнес-процессов.

Диаграммы состояний являются хорошо известным средством описания поведения систем. Они определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате влияния некоторых событий.

Переход может содержать метку. Синтаксически метка перехода состоит из трех частей, каждая из которых является необязательной: <Событие> [<Условие>]/<Действие>. Если метка перехода не содержит никакого события, это означает, что переход происходит, как только завершается какая-либо деятельность, связанная с данным состоянием. Из состояния «Проверка даты отчета» возможны два перехода. Метка одного из них включает условие. Условие – это логическое условие, которое может принимать два значения: «истина» или «ложь». Условный переход выполняется только в том случае, если условие принимает значение «истина», в противном случае выполняется переход, не помеченный условием.

Из конкретного состояния в данный момент времени может быть осуществлен только один переход; таким образом, условия являются взаимно исключающими для любого события. Существует два особых состояния: вход и выход. Любое действие, связанное с событием входа, выполняется, когда объект входит в данное состояние. Событие выхода выполняется в том случае, когда объект выходит из данного состояния. Диаграммы состояний хорошо использовать для описания поведения некоторого объекта в нескольких различных вариантах использования. Они не слишком пригодны для описания поведения ряда взаимодействующих объектов.

Рекомендуется строить диаграммы состояний только для тех классов, поведение которых влияет на общее поведение системы, например, для классов пользовательского интерфейса и управляющих объектов.

Если необходимо отразить уничтожение объекта используется узел завершения (terminate node), псевдосостояние, вход в который означает завершение выполнения поведения конечного автомата в контексте его объекта.

Узлы ветвления и объединения аналогичны узлам на диаграмме деятельности. Основная цель данных подсостояний показать параллельную работу подавтоматов. На диаграмме состояний обычно данные подсостяония используются распараллеливания переходов в композитных состояниях, о которых речь пойдет позже. После срабатывания перехода моделируемый объект одновременно будет находиться во всех целевых состояниях этого перехода.

В отличие от диаграммы деятельности, при отображении возможных вариантов перехода на диаграмме состояний узел выбора использовать не обязательно.

#### 3.1.2 Диаграмма состояний проектируемой системы

Если пользователь решает добавить BFT тег, то он выбирает ключ тега, вводит его значения и добавляет на форму, после чего снова переходит к вводу текста, если ввод которого окончен, он может произвести парсинг (анализ), а также посмотреть хронологию действий (логирование), затем пользователь выходит из программы.

Диаграмма состояний программы приведена на рисунке 3.1.

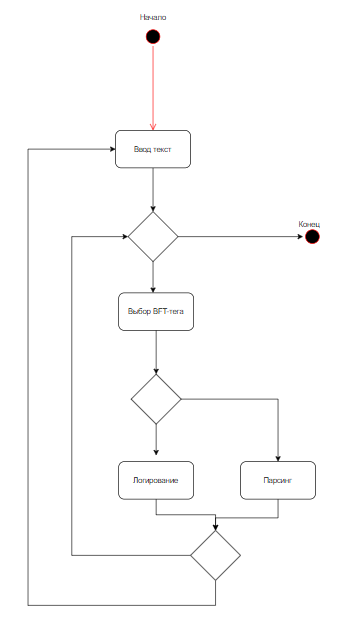


Рисунок 3.1 – Диаграмма состояний

### 3.2 Диаграмма последовательности

#### 3.2.1 Общие сведения

В контексте языка UML последовательность (sequence) представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме последовательности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Для моделирования процесса выполнения операций в языке UML используются так называемые диаграммы последовательности. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления не деятельностей, а действий, и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме последовательности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой, операции в предыдущем состоянии. Графически диаграмма последовательности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами – переходы от одного состояния действия к другому.

Таким образом, диаграммы последовательности можно считать частным случаем диаграмм состояний. Именно они позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Метамодель UML предоставляет для этого необходимые термины и семантику. Основным направлением использования диаграмм последовательности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы последовательности для описания реакций на внутренние события системы.

#### 3.2.2 Диаграмма последовательности проектируемой системы

В начале работы пользователь начинает ввод текста в окно программы. Добавляет BFT теги в окно программы, после чего они обрабатываются и в память добавляется новый BFT тег, который выводится в окно программы, пользователь записывает BFT файл, затем пользователь можен произвести анализ данных, так и посмотреть историю действий.

Диаграмма последовательности проектируемого приложения представлена на рисунке 3.2.

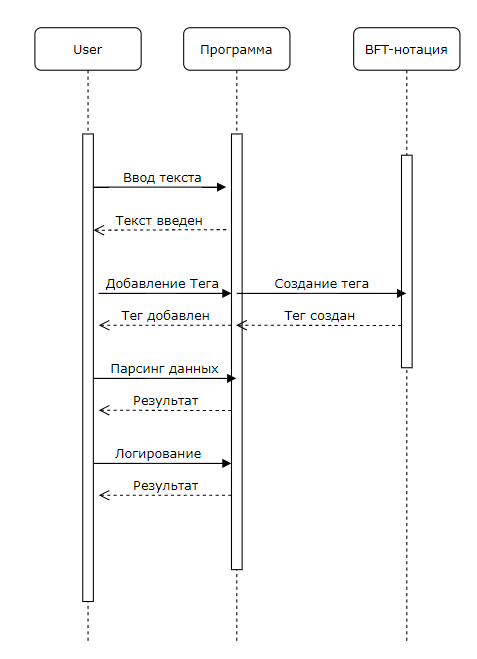


Рисунок 3.2 – Диаграмма последовательности

### 3.3 Диаграмма компонентов

#### 3.3.1 Общие сведения

Компонент (англ. component) – это физическая часть системы. Компоненты, представляющие собой файлы с исходным кодом классов, библиотеки, исполняемые модули и т.п., которые должны обладать согласованным набором интерфейсов. Для их графического представления используются следующие графические символы.

Интерфейс (англ. interface) – это внешне видимый, именованный набор операций, который класс, компонент или подсистема может предоставить другому классу, компоненту или подсистеме, для выполнения им своих функций. В некоторых языках программирования, в частности в C#, интерфейс представляет собой отдельный класс, включаемый и реализуемый (конкретизируемый) в части программного кода операций в составе других классов. На диаграмме компонентов интерфейс отображается так же, как и на диаграмме классов (слева от компонента необходимые для работы интерфейсы, справа – предоставляемые).

Отношение ассоциации отображается между компонентами и их интерфейсами. Отношение зависимости означает зависимость реализации одних компонентов от реализации других.

Диаграмма компонентов позволяет определить состав программных компонентов, в роли которых может выступать исходный, бинарный и исполняемый код, а также установить зависимости между ними. При разработке диаграмм компонентов преследуются цели:

* спецификация общей структуры исходного кода системы;
* спецификация исполнимого варианта системы;
* 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.

Данная диаграмма обеспечивает согласованный переход от логического к физическому представлению системы в виде программных компонентов. Одни компоненты могут существовать только на этапе компиляции программного кода, другие – на этапе его исполнения. Основными элементами диаграммы являются компоненты, интерфейсы и зависимости между ними. Кроме этого, на ней могут отображаться ключевые классы, входящие в компоненты.

#### 3.3.2 Диаграмма компонентов проектируемой системы

На диаграмме компонентов проектируемой программы было выделено несколько компонентов. Главный компонентом является окно программы mainWindow которое содержит три других компонента.

Компонент TextBox отвечает за хранение текста, вводимого пользователем.

Компоненты AddBtn отвечают за управление тегами и для этого вызывают компонент Abstract Factory.

С помощью диаграммы компонентов представляются инкапсулированные классы вместе с их интерфейсными оболочками, портами и внутренними структурами (которые тоже могут состоять из компонентов и коннекторов).

Когда диаграмма компонентов используется, чтобы показать внутреннюю структуру компонентов, предоставляемый и требуемый интерфейсы составного компонента, могут делегироваться в соответствующие интерфейсы внутренних компонентов.

Диаграмма компонентов представлена на рисунке 3.3.

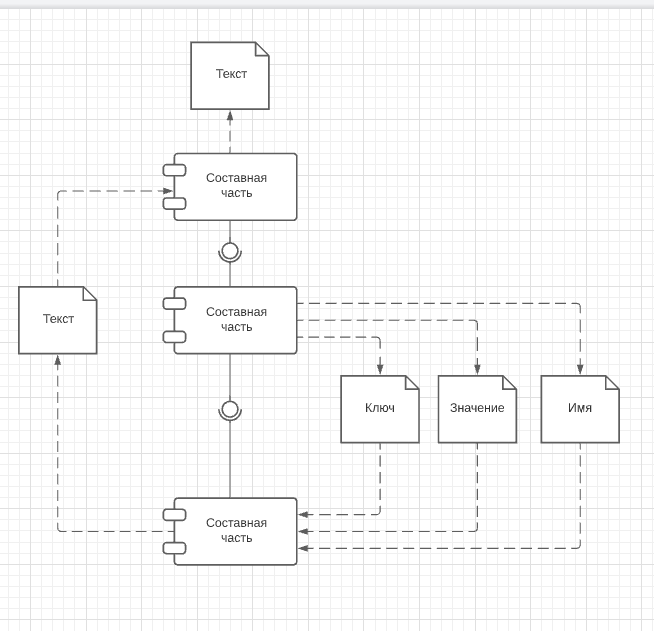


Рисунок 3.3 – Диаграмма компонентов

### 3.4 Диаграмма вариантов использования

#### 3.4.1 Общие сведения

Вариант использования описывает, с точки зрения действующего лица, группу действий в системе, которые приводят к конкретному результату.  
Варианты использования являются описаниями типичных взаимодействий между пользователями системы и самой системой. Они отображают внешний интерфейс системы и указывают форму того, что система должна сделать (именно что, а не как).

Одна из моделей формализации процесса постановки целей и задач проекта была предложена фирмой Rational и вошла в стандарт языка UML. Для этого применяются диаграммы вариантов использования (use-case), иногда называемые диаграммами прецедентов. Вариант использования представляет собой типичное взаимодействие пользователя и проектируемой системы. Варианты использования характеризуются рядом свойств:

* вариант использования охватывает некоторую очевидную для пользователей функцию;
* вариант использования, может быть, как небольшим, так и достаточно крупным;
* вариант использования решает некоторую дискретную задачу пользователя.

Диаграмма вариантов использования является самым общим представлением функциональных требований к системе. Основной поток событий описывает нормальный ход событий (при отсутствии ошибок). Альтернативные – описывают отклонения от нормального хода событий.

Достоинства модели вариантов использования:

* определяет пользователей и границы системы;
* определяет системный интерфейс;
* удобна для общения пользователей с разработчиками;
* используется для написания тестов;
* является основой для написания пользовательской документации;
* хорошо вписывается в любые методы проектирования (как объектно-ориентированные, так и структурные).

На диаграмме вариантов использования изображаются:

* актеры – группы лиц или систем, взаимодействующих с системой;
* варианты использования (прецеденты) – сервисы, которые система предоставляет актерам;
* комментарии;
* отношения между элементами диаграммы.

#### 3.4.2 Диаграмма вариантов использования проектируемой системы

Диаграмма вариантов использования приведена на рисунке 3.4.

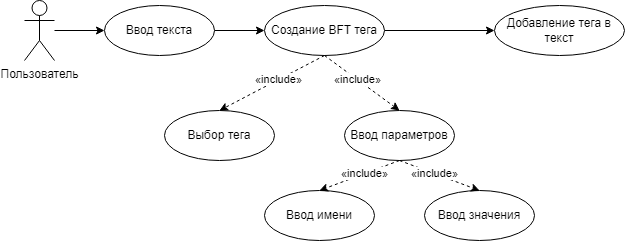


Рисунок 3.4 – Диаграмма вариантов использования (Use Case)

Начиная работу с программой, пользователь может загрузить нужный текст из файла, либо же ввести вручную.

Для добавления тегов нотации BFT пользователь формирует тег BFT, и вводит значения, которое включает в себя ввод ключа, ввод значения, выбор связи. После чего пользователь подтверждает свои действия.

## 4 РАЗРАБОТКА ПРОГРАММЫ НА ОСНОВЕ ШАБЛОНА PROXY

## 4.1 Диаграммы классов

## 4.1.1 Общие сведения

Диаграмма классов (Class diagram) – статическая структурная диаграмма, описывающая структуру системы, демонстрирующая классы системы, их атрибуты, методы и зависимости между классами.

Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

* концептуальная точка зрения – диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;
* точка зрения спецификации – диаграмма классов применяется при проектировании информационных систем;
* точка зрения реализации – диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

Диаграммы классов являются центральным звеном методологии объектно-ориентированных анализа и проектирования.

Атрибут (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым. Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Имя атрибута, как и имя класса, может представлять собой текст. На практике для именования атрибута используются одно или несколько коротких существительных, выражающих некое свойство класса, к которому относится атрибут. Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут представляет собой массив некоторых значений) и начальное значение по умолчанию.

Операция (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты. Графически операции представлены в нижнем блоке описания класса. Допускается указание только имен операций. Имя операции, как и имя класса, должно представлять собой текст. На практике для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например move (переместить) или isEmpty (проверка на пустоту). Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

Диаграмма классов показывает классы и их отношения, тем самым представляя логический аспект проекта. Отдельная диаграмма классов представляет определенный ракурс структуры классов. На стадии анализа диаграммы классов используются, чтобы выделить общие роли и обязанности сущностей, обеспечивающих требуемое поведение системы. На стадии проектирования диаграммы классов используются, чтобы передать структуру классов, формирующих архитектуру системы.

Каждый класс должен иметь имя; если имя слишком длинно, его можно сократить или увеличить сам значок на диаграмме. Имя каждого класса должно быть уникально в содержащем его проекте.

Диаграмма классов определяет типы объектов системы и различного рода статические связи, которые существуют между ними. Имеется два основных вида статических связей:

* 1. ассоциации (например, менеджер может вести несколько проектов);
  2. подтипы (работник является разновидностью личности).

На диаграммах классов изображаются также атрибуты классов, операции и ограничения, которые накладываются на связи между объектами.

## 4.1.2 Диаграмма классов проектируемой системы

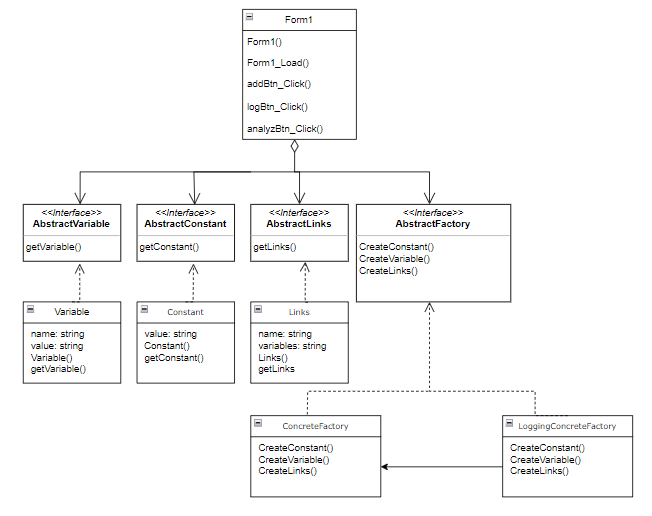


Рисунок 4.1 – Диаграмма классов

На рисунке 4.1 представлены следующие классы:

1. Form1: класс windows формы, который отвечает за ее настройки и поведение. Ключевая роль: «клиент».

Form1\_Load() - выступает в качестве окна при запуске

logBtn\_Click() - сценарий при нажатии на кнопку «Логирование»

addBtn\_Click() - сценарий при нажатии на кнопку «Добавить»

analyzeBtn\_Click() - сценарий при нажатии на кнопку «Произвести парсинг»

1. ConcreteFactory: реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать.
2. LoggingConcreteFactory: класс заместитель, предназначеный, для перехвата вызова основного объекта, и для улучшенной реализации, а также для выполнения одного из ключевых применений Proxy паттерна, такого как: Логирование - заместитель сохраняет историю действий (при нажатии на “Добавить”, можно считать, что происходит действие, хронологию которого можно будет посмотреть при помощи логирования)
3. AbstractFactory: класс производящий объекты других фабрик. Корректно в использовании, при реализации схожих объектов, с различной настройкой формирования этих самых объектов.
4. AbstractConstant/AbstractVariable/AbstractLinks: предназначены для интерфейсов классов.
5. Constant/Variable/Links: реализация абстрактных классов

## 5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЧЕНИЯ

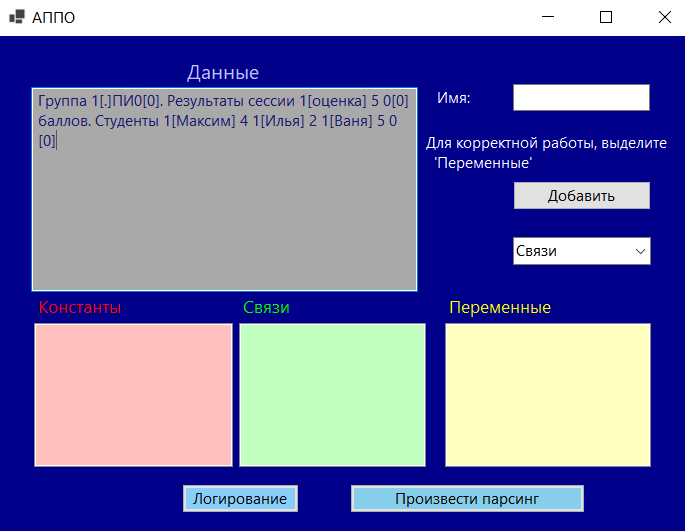


Рисунок 5.1 – Главная форма программы

Добавление константы BFT представлено на рисунке 5.2

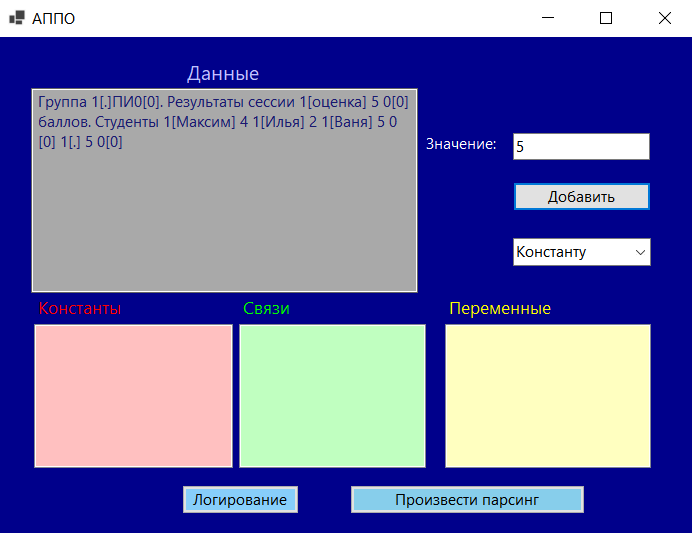


Рисунок 5.2 – Добавление константы

Добавление переменной представлено на рисунках 5.3-5.5

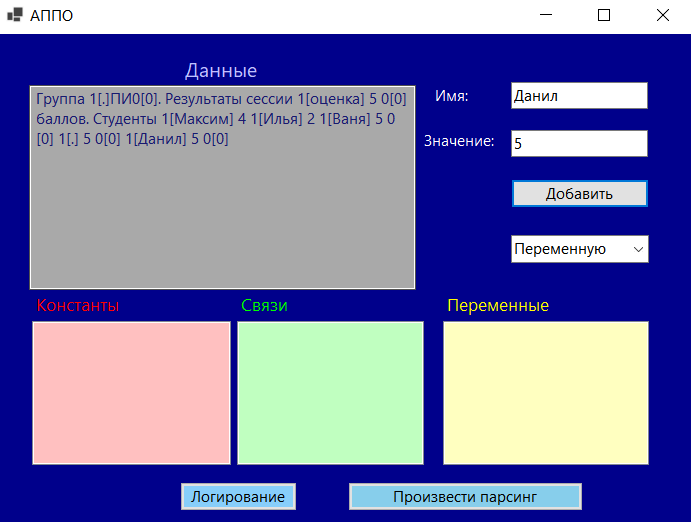


Рисунок 5.3 – Добавление переменной

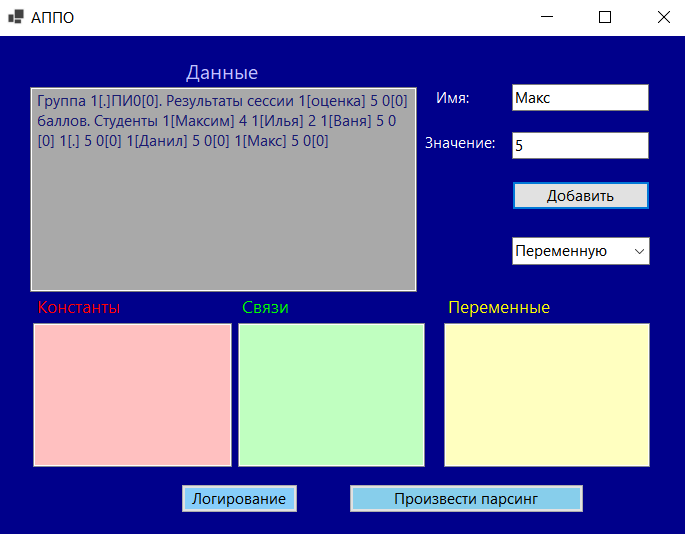


Рисунок 5.4 – Добавление переменной

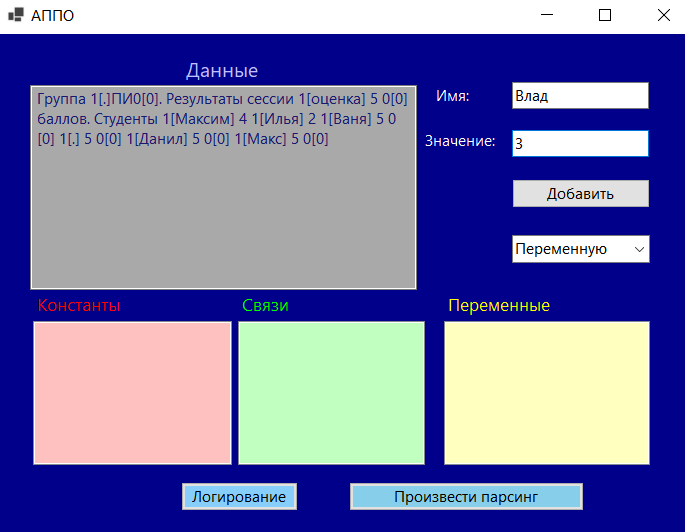


Рисунок 5.5 – Добавление переменной

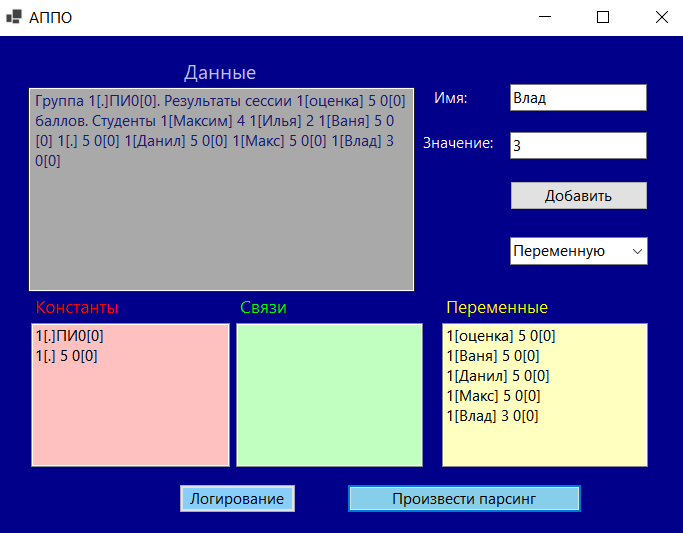


Рисунок 5.6 - Результат парсинга

Добавляем связи, результат на рисунке 5.4. Выделение нужных полей в столбце “Переменные”.

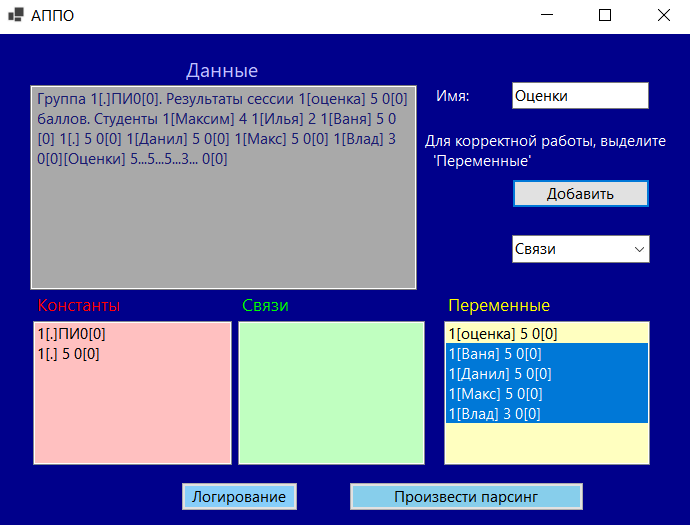


Рис. 5.7 - Добавление связи

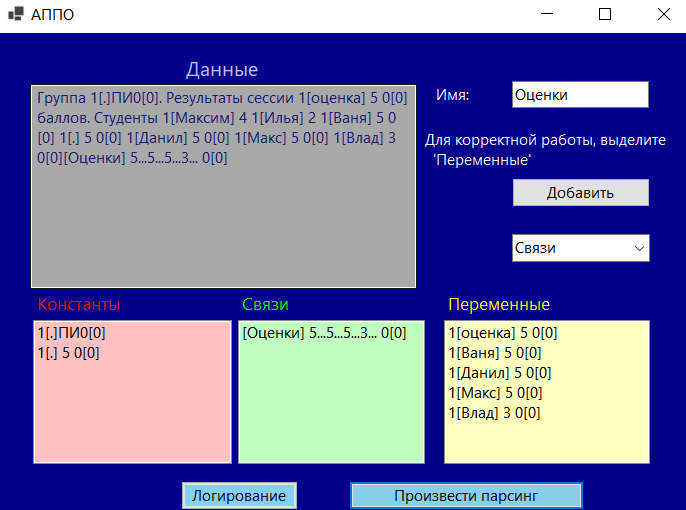


Рис. 5.7 - Парсинг текста BFT после добавления связи

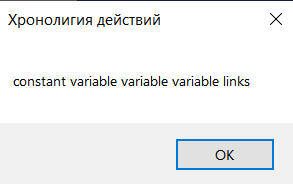


Рис. 5.8 - Логирование

На рисунке 5.9, показано, как будет происходить работа Proxy (логирования), если изменить фабрику “логирования” с заместителя, на абстрактную. Для этого изменим данные с помощью среды разработки Microsoft VS, перезапустим программу, произведем произвольное действие, и попробуем посмотреть историю действий.

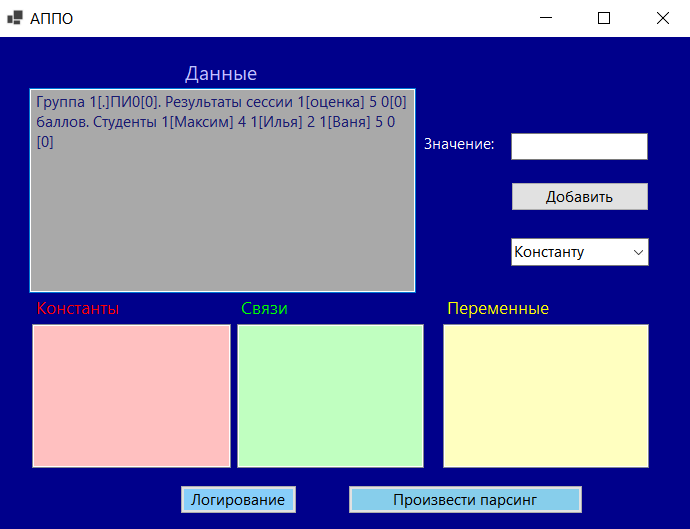


Рис. 5.9 - Запуск программы

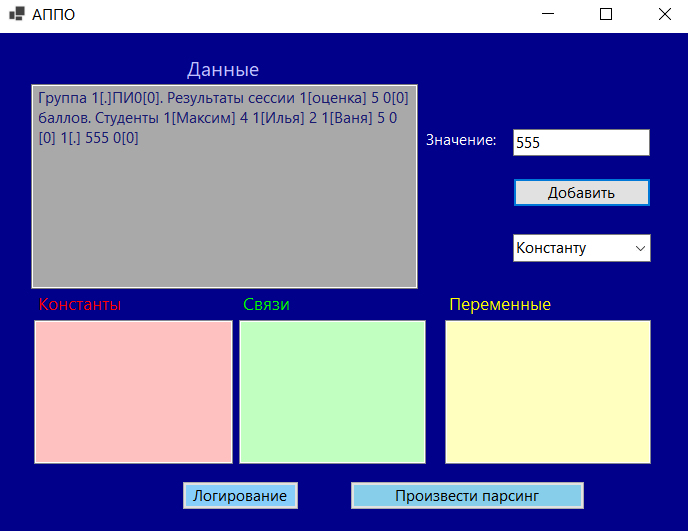


Рис. 5.10 - Делаем произвольное действие (в данном примере добавление костанты)

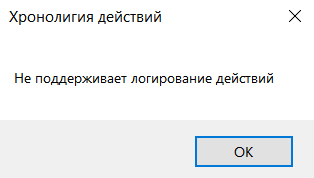


Рис. 5.11 - Попытка просмотра хронологии действий (Логирования)

Исходя из тестирования можно сделать вывод что программа со всеми ее функциями завершилась без ошибок и все необходимые методы реализованы в данной работе.

## ЗАКЛЮЧЕНИЕ

В ходе курсового проекта была разработана программа для работы с нотацией Brain Fogic Tag. Для ее проектирования был использован графический язык моделирования UML, который позволил описать и сформулировать четкую структуру создаваемого программного продукта.

В ходе разработки был применен шаблон проектирования Proxy, где реализовано одно из ключевых применений данного паттерна, а именно логирование, с данным действием вы регулярно сталкиваетесь, когда вам требуется хранить историю обращений к сервисному объекту, поэтому на примере данной программы, был реализован паттерн “Заместитель”, выполненный согласно рекамендациям его реализации, где он перехватывает работу введенной фабрики, и улучшает ее реализация, путем добавлением нового функционала.

Программа написана на языке высокого уровня C# в среде разработки «Microsoft Visual Studio», при помощи технологии WinForm (Windows Forms) и написана в ООП стиле. Для построения диаграмм было использовано веб-приложение «Draw.io».

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Макгрегор, Д. Тестирование объектно-ориентированного программного обеспечения: практическое пособие / Д. Макгрегор, Д. Сайкс; [пер. с англ. С.О. Шестакова]. – М.: ТИД ДС, 2012. – 432 с.
2. Буч, Г. Язык UML. Руководство пользователя. The Unified Modeling Language user guide / Г. Буч, Д. Рамбо, А. Якобсон; – 2-е изд. – М.: СПб.: [ДМК Пресс](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%9C%D0%9A_%D0%9F%D1%80%D0%B5%D1%81%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1" \o "ДМК Пресс (издательство) (страница отсутствует)), [Питер](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%82%D0%B5%D1%80_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)" \o "Питер (издательство)), 2004. – 432 с.
3. Джейсон, Мак-Колм Смит. [Элементарные шаблоны проектирования](http://www.williamspublishing.com/Books/978-5-8459-1818-5.html) Elemental Design Patterns / М. Джейсон. – М.: Вильямс, 2012. – 304 с.
4. Хелм, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Хелм, Р. Джонсон, Р. Влиссидес. – СПб.: [Питер](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%82%D0%B5%D1%80_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)" \o "Питер (издательство)), 2007. – 366 c.
5. Ларман, К. Применение UML 2.0 и шаблонов проектирования / К. Ларман. – М.: Вильямс, 2006. – 736 c.
6. Шмуллер, Д. Освой самостоятельно UML 2 за 24 часа. Практическое руководство. Sams Teach Yourself UML in 24 Hours, Complete Starter Kit / Д. Шмуллер. – М.: Вильямс, 2005. – 416 с.

## Приложение А ТЕХНИЧЕСКОЕ ЗАДАНИЕ

### А.1 Общие сведения

Тема курсового проекта: «Шаблон проектирования Proxy для нотации BFT».

Курсовой проект разрабатывается студентом 3-го курса Донецкого национального технического университета, факультета ИСП, группы ПИ-20г, Ващенко Д.И.

Основанием для разработки КП является задание, выданное кафедрой ПМИИ. Плановый срок начала работы по выполнению задания курсового проекта: 27.11.2021, срок окончания: 29.12.2021. Курсовой проект должен выполняться согласно графику, приведенному в таблице А.1.

Таблица А.1 – Этапы, результаты и сроки разработки программного продукта

|  |  |  |
| --- | --- | --- |
| № | Этап работы | Срок выполнения |
| 1 | Изучение паттерна проектирования | 1-4 недели |
| 2 | Разработка UML диаграмм для выбранного программного продукта | 5-8 недели |
| 3 | Разработка программного продукта с применением паттерна | 9-12 недели |
| 4 | Тестирование и отладка программного продукта | 13-14 недели |
| 5 | Оформление ПЗ | 15 неделя |
| 6 | Защита КП | 16 неделя |

### А.2 Назначение и цели создания проекта

Целью создания проекта является приобретение практических навыков в использовании паттерна проектирования Proxy на примере создания программного продукта на языке C#. Проект предназначен для демонстрации работы паттерна Proxy и тестового примера BFT.

Разработанный программный продукт может быть использован в учебных заведениях в качестве наглядного пособия при обучении принципам и правилам использования нотации Brain Fogic Tag.

### А.3 Требования к КП

Необходимо спроектировать программу при помощи графического языка UML, и разработать программный продукт для работы с нотацией Brain Fogic Tag, который будет использовать паттерн Proxy .

### А.4 Требования к задачам КП

#### А.4.1 Требования к системе в целом

В целом к системе предъявляются следующие требования:

1. корректная реализация используемого паттерна;
2. реализация функций ввода BFT тегов;
3. работа программы на различных данных для демонстрации работы паттерна.

#### А.4.2 Требования к задачам и функциям программного продукта

В процессе работы необходимо обеспечить выполнение следующих функций:

1. ввод параметров для моделирования;
2. вывод результатов на экран.

#### А.4.3 Требования к техническому обеспечению

К техническому обеспечению предъявляются следующие требования:

1. процессор – 32-битный x86-совместимый (уровня Pentium и выше);
2. объем оперативной памяти – не менее 32Мб;
3. свободное дисковое пространство – около 10 Мб. Не менее 5 Мб свободного дискового пространства для временных файлов;
4. графический адаптер – VGA-совместимый, HDMI-совместимый;
5. монитор – VGA-совместимый, HDMI-совместимый;
6. клавиатура, мышь.

#### А.4.4 Требования к программному обеспечению

Для стабильной работы к программному обеспечению предъявляется следующее требования:

1. обеспечить удобный и понятный интерфейс;
2. реализовать программу в виде отдельных классов.

Программным обеспечением для разработки программы является – «Visual Studio 2019». Для запуска программы необходимо наличие операционной системы Windows.

А.4.5 Требования к организационному обеспечению

В программную документацию должны входить:

1. пояснительная записка;
2. техническое задание
3. листинг программы.

## Приложение Б ЛИСТИНГ ПРОГРАММНЫХ МОДУЛЕЙ

**Form1.cs**

using System.Text.RegularExpressions;

namespace APPO

{

public partial class Form1 : Form

{

AbstractFactory factory;

public Form1()

{

InitializeComponent();

factory = new LoggingConcreteFactory();

}

private void Form1\_Load(object sender, EventArgs e)

{

// Стартовые значения -- нужны для наглядной демонстрации

comboBox1.SelectedIndex = 2;

textBox.Text = "Группа 1[.]ПИ0[0]. Результаты сессии 1[оценка] 5 0[0] баллов.\n" +

" Студенты 1[Максим] 4 1[Илья] 2 1[Ваня] 5 0[0]";

}

private void comboBox1\_SelectedIndexChanged(object sender, EventArgs e)

{

// Выпадающий список

if (comboBox1.SelectedIndex == 0)

{

nameLabel.Visible = false;

nameBox.Visible = false;

valueLabel.Visible = true;

valueLabel.Text = "Значение:";

valueBox.Visible = true;

}

else if (comboBox1.SelectedIndex == 1)

{

nameLabel.Visible = true;

nameBox.Visible = true;

valueBox.Visible = true;

valueLabel.Text = "Значение: ";

}

else if (comboBox1.SelectedIndex == 2)

{

nameLabel.Visible = true;

nameBox.Visible = true;

valueBox.Visible = false;

valueLabel.Text = "Для корректной работы, выделите \n 'Переменные'";

}

}

private void addBtn\_Click(object sender, EventArgs e)

{

if (comboBox1.SelectedIndex == 0)

{

Constant constant = factory.CreateConstant(valueBox.Text);

textBox.Text += " " + constant.getConstant();

}

else if (comboBox1.SelectedIndex == 1)

{

Variable variable = factory.CreateVariable(nameBox.Text, valueBox.Text);

textBox.Text += " " + variable.getVariable();

}

else if (comboBox1.SelectedIndex == 2)

{

string variables = "";

foreach (var item in variablesList.SelectedItems)

{

variables += " " + item.ToString();

}

Links links = factory.CreateLinks(nameBox.Text, variables);

textBox.Text += links.getLinks();

}

}

interface AbstractFactory

{

public Constant CreateConstant(string value);

public Variable CreateVariable(string name, string value);

public Links CreateLinks(string name, string variables);

}

class ConcreteFactory : AbstractFactory

{

public Constant CreateConstant(string value)

{

return new Constant(value);

}

public Variable CreateVariable(string name, string value)

{

return new Variable(name, value);

}

public Links CreateLinks(string name, string variables)

{

return new Links(name, variables);

}

}

// Proxy

class LoggingConcreteFactory : AbstractFactory

{

private string logs;

private AbstractFactory factory;

const string CONSTANT = "constant";

const string VARIABLE = "variable";

const string LINKS = "links";

public LoggingConcreteFactory()

{

factory = new ConcreteFactory();

}

public Constant CreateConstant(string value)

{

logs = logs + CONSTANT + " ";

return factory.CreateConstant(value);

}

public Variable CreateVariable(string name, string value)

{

logs = logs + VARIABLE + " ";

return factory.CreateVariable(name, value);

}

public Links CreateLinks(string name, string variables)

{

logs = logs + LINKS + " ";

return factory.CreateLinks(name, variables);

}

public string getLogs()

{

return logs;

}

}

public interface IMovable

{

void Move();

}

abstract class AbstractConstant

{ public abstract string getConstant(); }

abstract class AbstractVariable

{ public abstract string getVariable(); }

abstract class AbstractLinks

{ public abstract string getLinks(); }

class Constant : AbstractConstant

{

string value;

public Constant(string value)

{

this.value = value;

}

override public string getConstant()

{

return "1[.] " + this.value + " 0[0]";

}

}

class Variable : AbstractVariable

{

string name;

string value;

public Variable(string name, string value)

{

this.name = name;

this.value = value;

}

override public string getVariable()

{

return "1[" + this.name + "] " + this.value + " 0[0]";

}

}

class Links : AbstractLinks

{

string name;

string variables;

public Links(string name, string variables)

{

this.variables = variables;

this.name = name;

}

override public string getLinks()

{

string result = $"[{name}] ";

string[] vars = variables.Split(" ");

foreach (string var in vars)

{

if (var != "0[0]" && !var.Contains("1[") && var != "")

{

result += var + "...";

}

}

result += " 0[0]";

return result;

}

}

private void analyzeBtn\_Click(object sender, EventArgs e)

{

// Реализация парсинга

listBox1.Items.Clear();

variablesList.Items.Clear();

listBox3.Items.Clear();

string text = textBox.Text;

Regex regex = new Regex(@"1\[\.\](\s)\*(\w)\*(\s)\*0\[0\]");

MatchCollection match = regex.Matches(text);

foreach (Match m in match)

{

listBox1.Items.Add(m.Value);

}

regex = new Regex(@"1\[(\w\*)](\s)\*(\w)\*(\s)\*0\[0\]");

match = regex.Matches(text);

foreach (Match m in match)

{

variablesList.Items.Add(m.Value);

}

regex = new Regex(@"\[(\w\*)\](\s\*)((\w\*)\.\.\.)\*(\s\*)0\[0\]");

match = regex.Matches(text);

foreach (Match m in match)

{

listBox3.Items.Add(m.Value);

}

}

private void button1\_Click(object sender, EventArgs e)

{

if (!(factory is LoggingConcreteFactory))

{

MessageBox.Show("Не поддерживает логирование действий", "Хронолигия действий");

return;

}

LoggingConcreteFactory newFactory = (LoggingConcreteFactory)factory;

MessageBox.Show(newFactory.getLogs(), "Хронолигия действий");

}

}

}

**Form1.Designer.cs**

namespace APPO

{

partial class Form1

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

this.label1 = new System.Windows.Forms.Label();

this.textBox = new System.Windows.Forms.TextBox();

this.analyzBtn = new System.Windows.Forms.Button();

this.listBox1 = new System.Windows.Forms.ListBox();

this.label2 = new System.Windows.Forms.Label();

this.variablesList = new System.Windows.Forms.ListBox();

this.label3 = new System.Windows.Forms.Label();

this.listBox3 = new System.Windows.Forms.ListBox();

this.label4 = new System.Windows.Forms.Label();

this.addBtn = new System.Windows.Forms.Button();

this.comboBox1 = new System.Windows.Forms.ComboBox();

this.valueBox = new System.Windows.Forms.TextBox();

this.valueLabel = new System.Windows.Forms.Label();

this.nameLabel = new System.Windows.Forms.Label();

this.nameBox = new System.Windows.Forms.TextBox();

this.log = new System.Windows.Forms.Button();

this.SuspendLayout();

//

// label1

//

this.label1.AutoSize = true;

this.label1.Font = new System.Drawing.Font("Segoe UI", 12F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point);

this.label1.ForeColor = System.Drawing.Color.FromArgb(((int)(((byte)(192)))), ((int)(((byte)(192)))), ((int)(((byte)(255)))));

this.label1.Location = new System.Drawing.Point(184, 20);

this.label1.Name = "label1";

this.label1.Size = new System.Drawing.Size(84, 28);

this.label1.TabIndex = 0;

this.label1.Text = "Данные";

//

// textBox

//

this.textBox.BackColor = System.Drawing.Color.DarkGray;

this.textBox.ForeColor = System.Drawing.Color.MidnightBlue;

this.textBox.Location = new System.Drawing.Point(33, 51);

this.textBox.Multiline = true;

this.textBox.Name = "textBox";

this.textBox.Size = new System.Drawing.Size(387, 205);

this.textBox.TabIndex = 1;

//

// analyzBtn

//

this.analyzBtn.BackColor = System.Drawing.Color.SkyBlue;

this.analyzBtn.Location = new System.Drawing.Point(352, 448);

this.analyzBtn.Name = "analyzBtn";

this.analyzBtn.Size = new System.Drawing.Size(235, 29);

this.analyzBtn.TabIndex = 7;

this.analyzBtn.Text = "Произвести парсинг";

this.analyzBtn.UseVisualStyleBackColor = false;

this.analyzBtn.Click += new System.EventHandler(this.analyzeBtn\_Click);

//

// listBox1

//

this.listBox1.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)(255)))), ((int)(((byte)(192)))), ((int)(((byte)(192)))));

this.listBox1.FormattingEnabled = true;

this.listBox1.ItemHeight = 20;

this.listBox1.Location = new System.Drawing.Point(36, 287);

this.listBox1.Name = "listBox1";

this.listBox1.SelectionMode = System.Windows.Forms.SelectionMode.None;

this.listBox1.Size = new System.Drawing.Size(199, 144);

this.listBox1.TabIndex = 4;

//

// label2

//

this.label2.AutoSize = true;

this.label2.Font = new System.Drawing.Font("Segoe UI", 10F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point);

this.label2.ForeColor = System.Drawing.Color.Red;

this.label2.Location = new System.Drawing.Point(36, 259);

this.label2.Name = "label2";

this.label2.Size = new System.Drawing.Size(93, 23);

this.label2.TabIndex = 5;

this.label2.Text = "Константы";

//

// variablesList

//

this.variablesList.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)(255)))), ((int)(((byte)(255)))), ((int)(((byte)(192)))));

this.variablesList.FormattingEnabled = true;

this.variablesList.ItemHeight = 20;

this.variablesList.Location = new System.Drawing.Point(447, 287);

this.variablesList.Name = "variablesList";

this.variablesList.SelectionMode = System.Windows.Forms.SelectionMode.MultiExtended;

this.variablesList.Size = new System.Drawing.Size(206, 144);

this.variablesList.TabIndex = 6;

//

// label3

//

this.label3.AutoSize = true;

this.label3.Font = new System.Drawing.Font("Segoe UI", 10F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point);

this.label3.ForeColor = System.Drawing.Color.Yellow;

this.label3.Location = new System.Drawing.Point(447, 259);

this.label3.Name = "label3";

this.label3.Size = new System.Drawing.Size(112, 23);

this.label3.TabIndex = 7;

this.label3.Text = "Переменные";

//

// listBox3

//

this.listBox3.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)(192)))), ((int)(((byte)(255)))), ((int)(((byte)(192)))));

this.listBox3.FormattingEnabled = true;

this.listBox3.ItemHeight = 20;

this.listBox3.Location = new System.Drawing.Point(241, 287);

this.listBox3.Name = "listBox3";

this.listBox3.SelectionMode = System.Windows.Forms.SelectionMode.None;

this.listBox3.Size = new System.Drawing.Size(187, 144);

this.listBox3.TabIndex = 8;

//

// label4

//

this.label4.AutoSize = true;

this.label4.Font = new System.Drawing.Font("Segoe UI", 10F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point);

this.label4.ForeColor = System.Drawing.Color.Lime;

this.label4.Location = new System.Drawing.Point(241, 259);

this.label4.Name = "label4";

this.label4.Size = new System.Drawing.Size(57, 23);

this.label4.TabIndex = 9;

this.label4.Text = "Связи";

//

// addBtn

//

this.addBtn.Location = new System.Drawing.Point(515, 145);

this.addBtn.Name = "addBtn";

this.addBtn.Size = new System.Drawing.Size(138, 29);

this.addBtn.TabIndex = 4;

this.addBtn.Text = "Добавить";

this.addBtn.UseVisualStyleBackColor = true;

this.addBtn.Click += new System.EventHandler(this.addBtn\_Click);

//

// comboBox1

//

this.comboBox1.FormattingEnabled = true;

this.comboBox1.Items.AddRange(new object[] {

"Константу",

"Переменную",

"Связи"});

this.comboBox1.Location = new System.Drawing.Point(515, 201);

this.comboBox1.Name = "comboBox1";

this.comboBox1.Size = new System.Drawing.Size(138, 28);

this.comboBox1.TabIndex = 5;

this.comboBox1.SelectedIndexChanged += new System.EventHandler(this.comboBox1\_SelectedIndexChanged);

//

// valueBox

//

this.valueBox.Location = new System.Drawing.Point(515, 96);

this.valueBox.Name = "valueBox";

this.valueBox.Size = new System.Drawing.Size(137, 27);

this.valueBox.TabIndex = 3;

this.valueBox.Visible = false;

//

// valueLabel

//

this.valueLabel.AutoSize = true;

this.valueLabel.ForeColor = System.Drawing.SystemColors.ButtonHighlight;

this.valueLabel.Location = new System.Drawing.Point(424, 96);

this.valueLabel.Name = "valueLabel";

this.valueLabel.Size = new System.Drawing.Size(79, 20);

this.valueLabel.TabIndex = 13;

this.valueLabel.Text = "Значение:";

//

// nameLabel

//

this.nameLabel.AutoSize = true;

this.nameLabel.BackColor = System.Drawing.Color.Transparent;

this.nameLabel.ForeColor = System.Drawing.Color.Transparent;

this.nameLabel.Location = new System.Drawing.Point(435, 51);

this.nameLabel.Name = "nameLabel";

this.nameLabel.Size = new System.Drawing.Size(42, 20);

this.nameLabel.TabIndex = 14;

this.nameLabel.Text = "Имя:";

this.nameLabel.Visible = false;

//

// nameBox

//

this.nameBox.Location = new System.Drawing.Point(515, 48);

this.nameBox.Name = "nameBox";

this.nameBox.Size = new System.Drawing.Size(137, 27);

this.nameBox.TabIndex = 2;

this.nameBox.Visible = false;

//

// log

//

this.log.BackColor = System.Drawing.Color.LightSkyBlue;

this.log.Location = new System.Drawing.Point(184, 448);

this.log.Name = "log";

this.log.Size = new System.Drawing.Size(117, 29);

this.log.TabIndex = 15;

this.log.Text = "Логирование";

this.log.UseVisualStyleBackColor = false;

this.log.Click += new System.EventHandler(this.button1\_Click);

//

// Form1

//

this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 20F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.BackColor = System.Drawing.Color.DarkBlue;

this.ClientSize = new System.Drawing.Size(697, 509);

this.Controls.Add(this.log);

this.Controls.Add(this.nameBox);

this.Controls.Add(this.nameLabel);

this.Controls.Add(this.valueLabel);

this.Controls.Add(this.valueBox);

this.Controls.Add(this.comboBox1);

this.Controls.Add(this.addBtn);

this.Controls.Add(this.label4);

this.Controls.Add(this.listBox3);

this.Controls.Add(this.label3);

this.Controls.Add(this.variablesList);

this.Controls.Add(this.label2);

this.Controls.Add(this.listBox1);

this.Controls.Add(this.analyzBtn);

this.Controls.Add(this.textBox);

this.Controls.Add(this.label1);

this.Name = "Form1";

this.Text = "АППО";

this.Load += new System.EventHandler(this.Form1\_Load);

this.ResumeLayout(false);

this.PerformLayout();

}

#endregion

private Label label1;

private TextBox textBox;

private Button analyzBtn;

private ListBox listBox1;

private Label label2;

public ListBox variablesList;

private Label label3;

private ListBox listBox3;

private Label label4;

private Button addBtn;

private ComboBox comboBox1;

private TextBox valueBox;

private Label valueLabel;

private Label nameLabel;

private TextBox nameBox;

private Button log;

}

}