МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра інформаційних систем
(повна назва кафедри)

ДИПЛОМНА РОБОТА

МАРШРУТИЗАЦІЯ КОРИСТУВАЦЬКИХ ЗАПИТІВ. ВИКОРИСТАННЯ FLUENTAPI

Виконав: студент групи ПМІ-42
спеціальності 122 – комп'ютерні науки
(шифр і назва спеціальності)

Мручко І.В.
(підпис) (прізвище та ініціали)

Керівник Горлач В.М
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

2023

ЗМІСТ

ВСТУП

На основі попередніх досліджень питання маршрутизації користувацьких запитів[1] виникали питання формування системи, яка б дозволила розробнику інтуїтивно створити необхідні обробники маршрутизації та зв’язати свої методи з рядком-запитом. Вирішенням такого питання став підхід FluentApi[2] – спосіб створення програмного інтерфейсу, особливістю якого є створення ланцюжків викликів, який сприймається на рівні людської мови[3].

При вирішення проблеми маршрутизації за допомогою атрибутів постає питання адаптації цієї системи на інші об’єктно орієнтовні мови програмування такі як C++ чи python. У мові програмування Java доступна підтримка такого типу як Attribute, проте для усіх мов програмування є підхід побудови бібліотеки на основі FluentApi[4].

У мові програмування C# цей підхід використовується досить часто: вбудована мова запитів LINQ (Language Integrated Querry)[5] чи найпопулярніша бібліотека для доступу до бази даних – EntityFramework[6].

Враховуючи широку застосовність підходу FluentApi, попередні дослідження маршрутизації потрібно інтерпретувати згідно згаданої методології.

Проблема створення бібліотеки, яку використовуватимуть інші розробники, полягає у написанні правильної документації, опису реалізованої функціональності та пояснення алгоритму використання розроблених класів. Формування такої документації може витрачати більше часу, аніж сам процес програмування. Більше того, при зміні системи потрібно буде вносити зміни у описі цього API. Під час вирішення питання маршрутизації, поставали проблеми економного розширення обробників запиту користувача, проте такий спосіб не дозволяв контрольовано використовувати атрибути, зобов’язуючи користувача бібліотеки додавати зв’язані об’єкти типу Attribute між собою, що виключало можливості інформування розробника про те, що використання наданої функціональності є неправильним.

Таким чином, розробник при налаштуванні маршрутизаторів, що містить помилки, не міг відслідкувати причину, через яку його методи не були додані до

реєстру команд. Це спричиняє опрацювання документації що супроводжується витрачанням часу який можна витратити на реалізацію власної функціональності.

Повідомлення про помилку, яка виникає є ключовим під час розробки будь-якого програмного забезпечення. Розробник повинен розуміти, чому бібліотека, яку він використовує не працює так, як він очікує, а користувач цієї системи маршрутизації повинен розуміти, чому саме він не може отримати відповіді на свій запит.

Через адаптивність підходу FluentApi та можливість вдосконалення цієї архітектури побудови бібліотек задля контролю дій розробника систему маршрутизації варто інтерпретувати.

ПОСТАНОВКА ЗАДАЧІ

На основі попередніх розробок зроблено висновок, що така система маршрутизації не є досить зручною саме для користувача бібліотеки. Основними завданнями, які вирішуються цим дослідженням є:

а) Інтерпретування досліджуваної системи маршрутизації за допомогою підходу FluentApi:

1) Створення єдиної точки початку додавання маршрутизаторів;

2) Контроль за додаванням потрібних значень до об’єктів маршрутизації, правильності використання;

3) Створення підказок, які використовуватимуться у середовищі розробки програмного забезпечення для швидкого пристосування до функціоналу;

б) Реалізація системи перевірок на основі FluentAPI для доступного інформування користувача та розробника;

в) Реалізація розширення функціоналу за допомогою концепції метапрограмування:

1) Створення атрибутів для ініціалізації параметрів маршрутизації;

2) Створення тестів на основі правильних та не правильних значень до розробленої функціональності;

г) Створення правильного способу отримання інформації з цілей атрибутів на основі підходу FluentApi.

Головною ідеєю є розширення підходу написання архітектури бібліотеки на основі наслідування. Це дозволить розробнику крок за кроком отримувати інформацію про розроблену функціональність та правильно вказувати необхідні дані для створення маршрутизаторів.

РОЗДІЛ 1. РЕАЛІЗАЦІЯ FLUENTAPI

Використання бібліотечної функціональності повинно бути контрольоване, тому варто зменшити залежність від мови програмування, яка містить атрибути. Один із основних патернів – шаблонів програмування – , які можна використати для створення нових об’єктів є паттерн Builder[8], який дозволяє будувати структури визначеним чином. Для реалізації маршрутизації користувацьких запитів додано об’єкти Command, Overload, RequiredArgument, OptionalArgument, які можна правильно та послідовно створити. Варто зауважити, що для спрощення написання коду усі методи для створення об’єктів було названо однаково, щоб розробнику було легше освоїти бібліотеку, оскільки створено єдиний початок побудови.

1.1 ArgumentBuilder

Ієрархія об’єктів RequiredArgument та OptionalARgument наведена на рисунку 1.1.

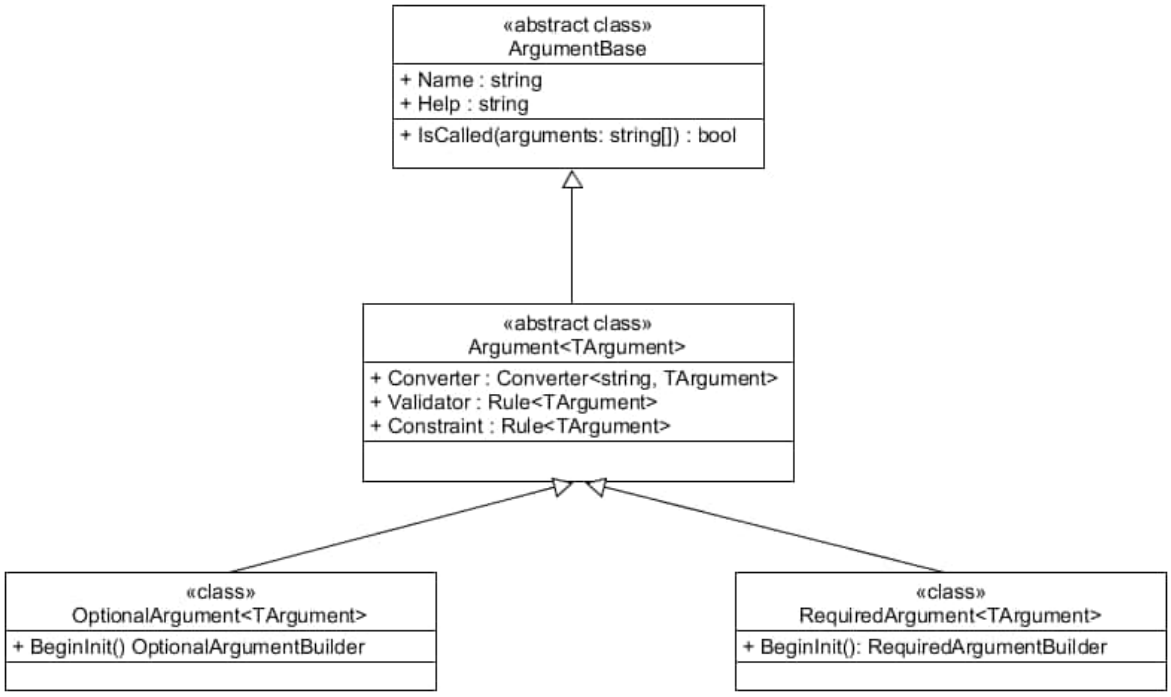

Рис. 1.1 – Ієрархія класів Arguments.

Із рисунка 1.1 випливає, що спільні параметри аргументів є лише на рівні абстракції. Тому можна об’єднати налаштування спільних властивостей у один ArgumentBuilder, проте галуження в залежності від обраного типу реалізовувати на кроці завершення ініціалізації.

Розглянемо набір інтерфейсів (проте для мов програмування python, C++ потрібно використовувати абстрактні класи), які реалізує об’єкт ArgumentBuilder.

```
public abstract class ArgumentBuilder<TArgumentObject, TArgument>
    : IArgumentNameSetter<TArgumentObject, TArgument>,
      IHelpSetter<IConverterSetter<TArgumentObject, TArgument>>,
      IConstraintSetter<TArgumentObject, TArgument>,
      IConverterSetter<TArgumentObject, TArgument>,
      IArgumentFinalizer<TArgumentObject, TArgument>,
      IRuleBaseSetter<TArgumentObject, TArgument>,
      IValidationOnlySetter<TArgumentObject, TArgument>,
      IConstraintOnlySetter<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
```

Рис. 1.2 – Реалізація інтерфейсів класом ArgumentBuilder.

Для того, щоб контролювати дії розробника, усю функціональність ArgumentBuilder було розбито на етапи, які за допомогою коду визначаються як інтерфейси чи абстрактні класи (іншими об’єктно-орієнтованими мовами). Також з рисунка 1.2 бачимо, що назначається обмеження на тип TArgumentObject. Це дозволяє зробити галуження в залежності від об’єкту, який необхідно створити (RequiredArgument, OptionalArgument) та дозволить створювати інші аргументи, якщо система буде розширюватись. До прикладу, у мові програмування C# є параметри, які є типом reference, що дозволяє модифікувати їх значення після того, як вони потрапили в метод. Отже, наслідуючи клас Argument<TArgument>, ми можемо розширити ієрархію ArgumentBuilder.

Також потрібно взяти до уваги, що така паралельна ієрархія може бути ознакою важкої розширюваності коду[9], але, враховуючи зв’язок між ієрархіями у вигляді методів початку процесу налаштування (рисунок 1.3) та невелику кількість можливих розширень цих ієрархій – кількість параметрів,

що підтримуються у мовах програмування є обмежена – можна вважати, що таке дублювання ієрархій є змістовним[9].

```
public static IArgumentNameSetter<OptionalArgument<TArgument>, TArgument> BeginInit()
    => new OptionalArgumentBuilder<TArgument>();

public static IArgumentNameSetter<RequiredArgument<TArgument>, TArgument> BeginInit()
    => new RequiredArgumentBuilder<TArgument>();

public static ICommandNameSetter BeginInit()
    => new CommandBuilder();

public static IHelpSetter<IParametersSetter> BeginInit()
    => new OverloadBuilder();

public static IRuleBodySetter<TValue> BeginInit()
    => new RuleBuilder<TValue>();
```

Рис. 1.3 – Приклади однотипного іменування.

На рисунку 1.4 наводиться приклад побудови аргрументу та розкриваються особливості викликів методів для налаштування аргументу.

```
RequiredArgument<int>.BeginInit()
                     .WithName("number1")
                     .WithHelp("First number of the sequence.")
                     .UseDefaultConverter()
                     .EndInit();
```

```
RequiredArgument<int>.BeginInit()
                     .WithName("number2")
                     .WithHelp("Second number of the sequence.")
                     .WithConverter(v => int.Parse(v))
                     .UseDefaultConverter()
                     .EndInit();
```

Рис. 1.4 – Обмеження встановлення конвертерів.

На рисунку 1.4 зображено помилка компіляції під час написання коду. Створюється обов’язковий аргумент типу int з назвою та описом та встановленням конвертера. Можна використати конвертер за замовчуванням або встановити власний конвертер, який перетворить string до цілочислового значення. Основною ознакою контролю за дією розробника – помилка компіляції, а точніше, попереднього перегляду коду середовищем розробки. Не усі середовища підтримують таку функцію, тому під час перетворення написаного коду у виконувану програму виникне помилка, яка буде вказувати, що на поточному етапі не можна використати цей метод.

Цей приклад демонструє використання «виключне або» за допомогою інтерфейсів і реалізовується за допомогою етапу, визначеним інтерфейсом IConverterSetter.

```
public interface IConverterSetter<TArgumentObject, TArgument> : IAfterHelpSetter
    where TArgumentObject : Argument<TArgument>
{

    IRuleBaseSetter<TArgumentObject, TArgument> WithConverter(Converter<string, TArgument> converter);


    IRuleBaseSetter<TArgumentObject, TArgument> UseDefaultConverter();
}
```

Рис. 1.5 – Інтерфейс IConverterSetter.

Інтерфейс, який відповідає за етап налаштування конвертеру агрументу, містить два методи WithConverter та UseDefaultConverter, які реалізує ArgumentBuilder. Кожен із цих методів переходить на наступний етап налаштування аргументу, тому одночасно встановити конвертер за замовчуванням та власний алгоритм перетворення рядка на число не можливо.

Також IConverterSetter містить реалізацію інтерфейсу IAfterHelpSetter, що дозволяє реалізовувати інтерфейс IHelpSetter в залежності від потрібного наступного етапу після встановлення інформації про допомогу.

```
public interface IRuleBaseSetter<TArgumentObject, TArgument> :
    IConstraintSetter<TArgumentObject, TArgument>,
    IValidationSetter<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{
}
```

Рис. 1.6 – Інтерфейс IRuleBaseSetter.

На рисунку 1.6 зображено інтерфейс, який є наступним етапом після встановлення конвертера. Він реалізує інтерфейси IConstraintSetter та IValidationSetter, які у свою чергу реалізують інтерфейс IArgumentFinalizer, щоб можна було завершити налаштування не встановивши обмеження та валідацію – перевірка правильності значення – на параметр.

```
public interface IConstraintSetter<TArgumentObject, TArgument> :
    IArgumentFinalizer<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{

    IValidationOnlySetter<TArgumentObject, TArgument> WithConstraint(Rule<TArgument> constraint);
}
public interface IValidationSetter<TArgumentObject, TArgument> :
    IArgumentFinalizer<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{

    IConstraintOnlySetter<TArgumentObject, TArgument> WithValidator(Rule<TArgument> validation);
}
public interface IArgumentFinalizer<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{

    TArgumentObject EndInit();
}
```

Рис. 1.7 – Інтерфейси IConstraintSetter, IValidationSetter, IArgumentFinalizer.

Наведені на рисунку інтерфейси повертають етапи, у яких уже не можна встановити обмеження чи валідацію відповідно, таким чином було реалізовано логічне «або», оскільки при налаштуванні параметру можна встановити або лише обмеження, або лише валідацію, або і то, і то. За рахунок того, що встановлення обмеження не є обов'язковим, цей етап можна пропустити, завершивши ініціалізацію за допомогою методу EndInit().

```
public interface IValidationOnlySetter<TArgumentObject, TArgument> :
    IArgumentFinalizer<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{

    IArgumentFinalizer<TArgumentObject, TArgument> WithValidator(Rule<TArgument> validation);
}
public interface IConstraintOnlySetter<TArgumentObject, TArgument> :
    IArgumentFinalizer<TArgumentObject, TArgument>
    where TArgumentObject : Argument<TArgument>
{

    IArgumentFinalizer<TArgumentObject, TArgument> WithConstraint(Rule<TArgument> validation);
}
```

Рис. 1.8 – Інтерфейси IConstraintOnly, IValidationOnly.

Для того, щоб було зрозуміліше, як інтерфейси один з одним взаємодіють, наведено діаграму класів[10] та зв'язків між ними а також схему етапів налаштування параметрів згідно цих інтерфейсів [11]. Важливо зауважити, що при додаванні нових етапів до налаштування, потрібно редагувати окіл деякого етапу окремо – якщо додається новий інтерфейс, то

метод повинен повертати тип наступного стану налаштування, у якому потрібно буде лише змінити сигнатуру методу за результатом.

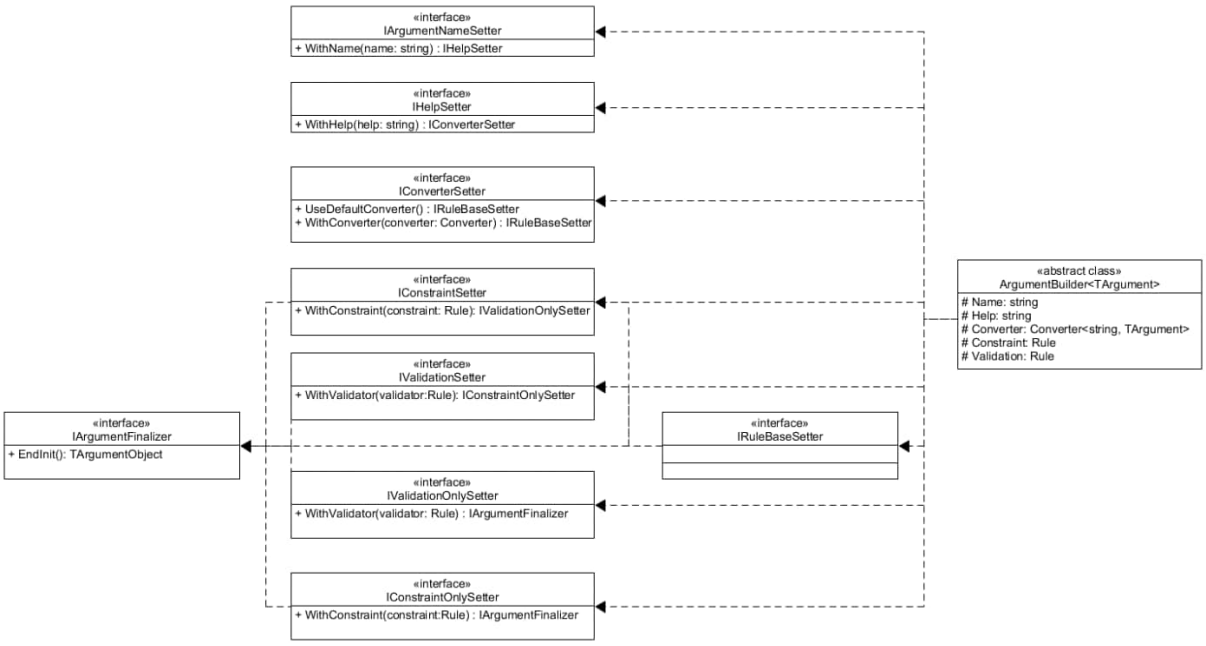

Рис. 1.9 Діаграма класів ArgumentBuilder

На рисунку 1.9 зображена взаємодія інтерфейсів та класу ArgumentBuilder.

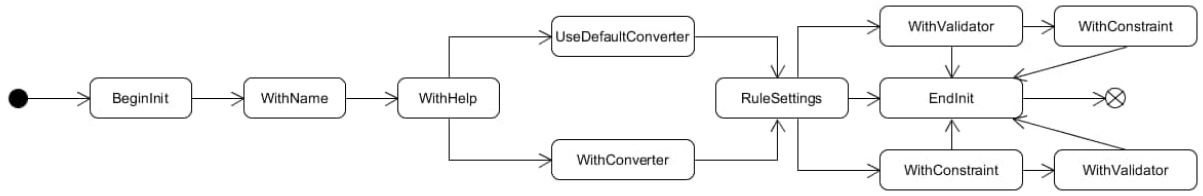

Рис. 1.10 Можлива послідовність викликів методів ArgumentBuilder.

На рисунку 1.10 зображено діаграму станів, де кожен стан – певний етап налаштування параметру (аргрументу). На зображені видно послідовність, які методи можуть бути використані на кожному етапі.

Розглянемо кожен етап детальніше:

а) На першому етапі можна встановити лише назву параметру;

б) На другому етапі можна встановити лише інформацію про параметр;

в) На третьому етапі можна або встановити конвертер за замовчуванням, або встановити потрібний конфертер самостійно;

г) На четвертому етапі можна встановити валідатор, завершити побудову аргументу або встановити обмеження;

д) На п’ятому етапі (встановлено обмеження чи валідацію) можна встановити лише валідацію чи обмеження, в залежності від того, що було встановлено на етапі 4, а також завершити побудову параметру.

е) На шостому етапі можна лише завершити побудову параметрів.

1.2 OverloadBuilder

Перевантаження (Overload) – набір параметрів різної послідовності та типів – містить метод, який приймає на вхід обов’язкові (RequiredArgument) та не обов’язкові (OptionalArgument) параметри.

Об’єкт перевантаження містить як синхрону реалізацію (яку можна застосовувати у консольних додатках) так і асинхрону версію методу (для застосування системи маршрутизації у чат-ботах).

Для побудови такого об’єкту використано принцип логічного «або» для встановлення потрібних параметрів: можна встановити Required чи Optional Argument у довільній послідовності, проте, після встановлення тіла перевантаження додавання аргументів стає неможливим.

Розглянемо етапи, які містить OverloadBuilder. Кожен етап виражений певним інтерфейсом, який цей клас реалізує.

```
public class OverloadBuilder : IHelpSetter<IParametersSetter>,
                               IParametersSetter,
                               IBodySetter,
                               IOverloadFinalizer
```

Рис. 1.11 – Інтерфейси (етапи) налаштування перевантаження.

Спочатку потрібно встановити допоміжну інформацію для перевантаження, після чого можна встановлювати довільну кількість параметрів. Коли аргументи, які були потрібні, встановлено – встановлюється виконуване тіло перевантаження, після чого можна завершити ініціалізацію.

Інтерфейс IHelpSetter уже описувався у розділі 1.1, проте він повертав інтерфейс іншого етапу налаштування. Після встановлення допомоги встановлюється стан налаштування параметрів.

Інтерфейс IParameterSetter надає 4 методи, які можна використовувати під час налаштування параметрів.

```
public interface IParametersSetter : IAfterHelpSetter, IBodySetter
{
    IParametersSetter WithRequired<TArgument>(RequiredArgument<TArgument> argument);

    IParametersSetter WithRequired<TArgument>(Func<IArgumentNameSetter<RequiredArgument<TArgument>, TArgument>,
        RequiredArgument<TArgument>> argument);

    IParametersSetter WithOptional<TArgument>(OptionalArgument<TArgument> argument);

    IParametersSetter WithOptional<TArgument>(Func<IArgumentNameSetter<OptionalArgument<TArgument>, TArgument>,
        RequiredArgument<TArgument>> argument);
}
```

Рис. 1.12 – Інтерфейс IParameterSetter.

Параметри можна не встановлювати взагалі, тому що цей інтерфейс реалізує IBodySetter, який встановлює тіло – виконуваний код – перевантаження. Таким чином можна створити об’єкт, який не буде містити параметрів при маршрутизації, тобто команда може містити перевантаження, яке не приймає аргрументів.

Ще слід зауважити, що окрім можливості додати уже створений параметр завчасно, можна створити його у параметрі методів WithOptional чи WithRequired, який містить делегат Func[12], що приймає параметром початковий стан ArgumentBuilder та повертає уже сконструйований аргрумент. Оскільки система станів налаштування параметру є завершеною,

то при створені RequiredArgument чи OptionalArgument через параметр методу гарантовано буде отримано необхідний об’єкт.

Інтерфейс IBodySetter встановлює тіло перевантаження, після чого можна завершити створення.

```
public interface IBodySetter : IAfterHelpSetter
{
    IOverloadFinalizer WithBody(Func<List<ArgumentBase>, List<ArgumentBase>, string> body);
}
```

Рис. 1.13 – Інтерфейс IBodySetter.

На цьому етапі визначено лише один метод, після чого налаштування переходить до свого завершення за допомогою методу EndInit().

```
public interface IOverloadFinalizer
{
    Overload EndInit();
}
```

Рис. 1.14 – Інтерфейс IOverloadFinalizer.

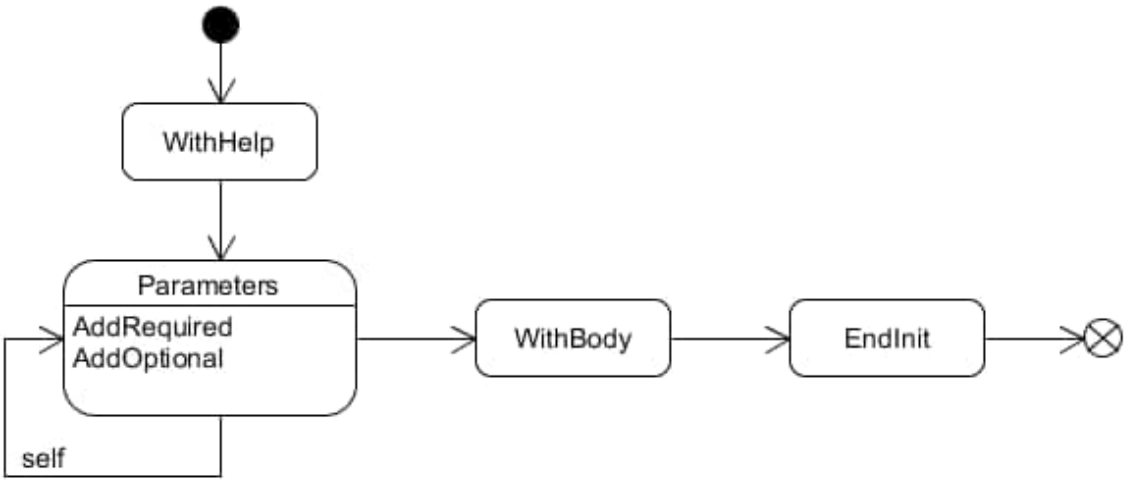

Рис. 1.15 – Етапи налаштування перевантаження.

1.3 CommandBuilder

```
public class CommandBuilder : ICommandNameSetter,
                              IHelpSetter<ICallableSetter>,
                              ICallableSetter,
                              ICommandFinalizer,
                              ICommandInitializer
```

Об’єкт типу Command є ключовим об’єктом при налаштуванні маршрутизації, оскільки він містить інформацію про внутрішні команди та виконуваний код (перевантаження).

Рис. 1.16 – Етапи налаштування команди у вигляді інтерфейсів.

Для налаштування команди створено наступні етапи:

а) Перший етап – встановлення назви команди;

б) Другий етап – встановлення довідкової інформації;

в) Третій етап – встановлення внутрішньої команди чи перевантаження. Також на цьому етапі не можливо створити об’єкт команди, оскільки необхідно, щоб команда містила Overload чи іншу команду, яка її містить;

г) Четвертий етап – встановлення команди чи перевантаження з можливістю завершити налаштування.

Для встановлення вимоги, що команда повинна містити хоча б одне перевантаження інтерфейс ICallableSetter не реалізує ICommandInitializer, проте повертає своїми методами ICommandFinalizer – який має можливість завершити налаштування.

```
public interface ICallableSetter : IAfterHelpSetter
{
    ICommandFinalizer AddInner(Command innerCommand);

    ICommandFinalizer AddInner(Func<ICommandNameSetter, Command> innerCommandInstruction);

    ICommandFinalizer AddOverload(Overload overload);

    ICommandFinalizer AddOverload(Func<IHelpSetter<IParametersSetter>, Overload> overloadInstruction);
}
public interface ICommandFinalizer : ICallableSetter, ICommandInitializer
{
}
```

Рис. 1.17 – інтерфейси ICallableSetter, ICommandFinalizer.

З рисунку 1.17 випливає можливість додавати перевантаження чи внутрішні команди аналогічно, як можна додавати параметри до об’єктів Overload – за допомогою інструкцій, які містять параметром початковий стан відповідного Builder`a.

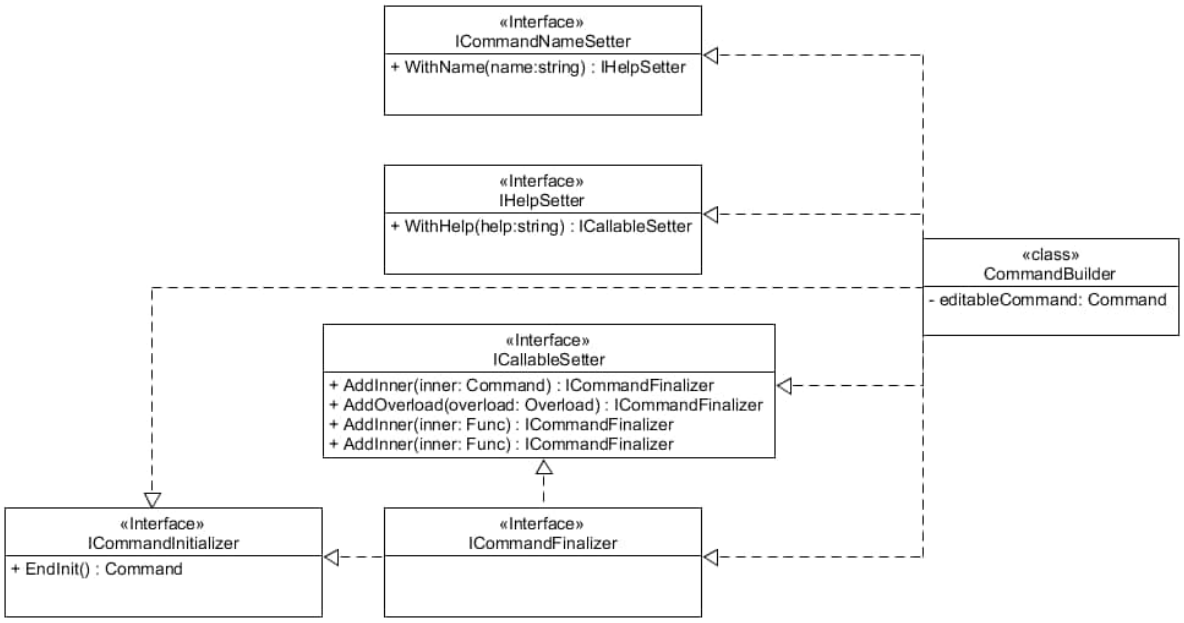

Рис. 1.18 – Ієрархія інтерфейсів CommandBuilder.

Також зображено діаграму станів, яка виражає етапи налаштування команди.

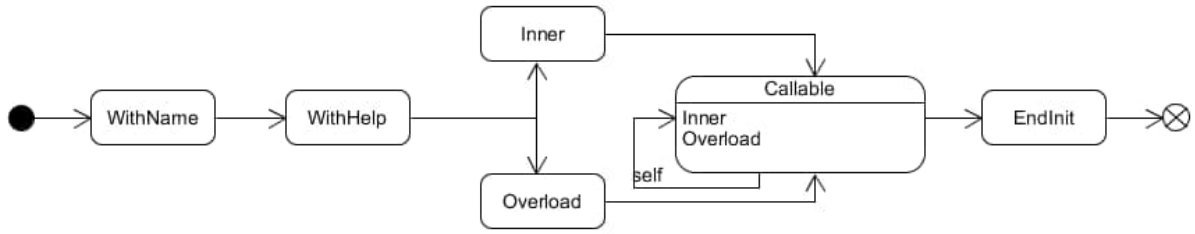

Рис. 1.19 – Множина станів налаштування команди.

Таким чином, використовуючи статичний метод Command.BeginInit() можна створити команду з перевантаженнями та внутрішніми командами, не використовуючи зовнішніх довідок.

1.4 RuleBuilder

Для того, щоб створити обмеження чи правила валідації[13], реалізовано систему відповідну систему (ця система описана у розділі 2). Проте для створення самого обмеження (правила) реалізовано алгоритм побудови, який дозволяє розробнику контрольовано використовувати функціональність валідаційного процесу.

Початок побудови базується на такому ж принципі, що й інші алгоритми побудови: точка входження – це статичний метод BeginInit() класу Rule<>. Особливістю цієї системи є те, що можна повернутись на передостанній етап та створити логічний вираз.

Клас, який відповідає за правильне конструювання об’єкту Rule є RuleBuilder.

```
public class RuleBuilder<TValue> : IRuleBodySetter<TValue>,
                                   IRuleInformationLevelSetter<TValue>,
                                   IRuleConstraintSetter<TValue>,
                                   IPropertySelectorSetter<TValue>,
                                   IOnlyConstraintSetter<TValue>,
                                   IOnlyPropertySelectorSetter<TValue>,
                                   IRuleOperationSetter<TValue>,
                                   IRuleFinalizer<TValue>,
                                   IRuleConstraintBuilder<TValue>
```

Рис. 1.20 – Список інтерфейсів, що реалізує RuleBuilder.

Після виклику методу BeginInit() використовується метод FromMetod(), який задає основне правило і знаходиться у інтерфейсі IRuleBodySetter.

```
public interface IRuleBodySetter<TArgument>
{
    IRuleInformationLevelSetter<TArgument> FromMethod(Func<TArgument, bool> method);
}
```

Рис. 1.21 – Інтерфейс IRuleBodySetter.

Цей інтерфейс дозволяє встановити метод, який відповідає делегату типу Func, де вхідний аргумент залежить від типу, на яке націлено правило, а результатом якого є булева змінна. Після цього, налаштування об’єкту Rule переходить на етап е рівня відображення інформації.

Рівень відображення інформації – це клас, який опрацьовує повідомлення у разі невідповідності значення об’єкту до обмеження, вказаного у правилі. Детальніше про рівні опрацювання помилки можна прочитати у розділі 2.1.

Другим етапом створення правила є встановлення рівня інформування користувача. Розробник на цьому етапі може встановити рівень за замовчуванням (метод OnDefaultLevel), встановити як клас, який наслідує абстрактний клас Inform або з наявних рівнів інформування, що винесені в перелік InformingLevel.

```
public interface IRuleInformationLevelSetter<TValue>
{
    IRuleOperationSetter<TValue> OnDefaultLevel();

    IRuleConstraintBuilder<TValue> OnLevel(Informing level);

    IRuleConstraintBuilder<TValue> OnLevel(InformingLevel level);
}
```

Рис. 1.22 – Інтерфейс IRuleInformationLevelSetter.

Коли розробник визначить рівень інформування правила, налаштування об’єкту Rule перейде на наступний етап, який встановлює інформацію про обмеження. У випадку, коли встановлюється рівень за замовчуванням, налаштування повідомлення пропускається, оскільки така інформація ігнорується. Такий підхід потрібен у випадках, коли правило використовується для перевірок у операторі галуження if. Проте, коли інформація про помилку повинна відобразитись розробнику, то потрібно встановити обмеження та делегат Func для отримання повної довідки про невідповідність даних.

Інтерфейс IRuleConstraintBuilder реалізує IRuleConstraintSetter та IPropertySelectorSetter, що дозволяє на цьому етапі встановити або повідомлення про суть обмеження, або метод, який буде отримувати інформацію про об’єкт.

```
public interface IRuleConstraintBuilder<TValue> :
    IRuleConstraintSetter<TValue>,
    IPropertySelectorSetter<TValue>
```

Рис. 1.23 – Інтерфейс етапу побудови інформації про обмеження.

Кожен з цих інтерфейсів змінює етап так, щоб встановити другу частину повідомлення про обмеження. Тобто при встановлені опису обмеження наступним етапом можна встановити лише метод отримання інформації з об’єкту і навпаки: при встановлені методу, наступний етап – встановлення опису.

```
public interface IRuleConstraintSetter<TValue>
{
    IOnlyPropertySelectorSetter<TValue> WithConstraint(string message);
}
public interface IPropertySelectorSetter<TValue>
{
    IOnlyConstraintSetter<TValue> WithPropertySelector(Func<TValue, string> selector);
}
```

Рис. 1.24 – Інтерфейси встановлення інформації про обмеження.

Інтерфейси IOnlyConstraintSetter та IOnlyPropertySelectorSetter переходять на етап, при якому можна налаштувати операції між правилами. До таких операцій відносять And, Or, Not – основні операції алгебри логіки.

```
public interface IRuleOperationSetter<TValue> : IRuleFinalizer<TValue>
{
    IRuleOperationSetter<TValue> And(Rule<TValue> another);

    IRuleOperationSetter<TValue> And(Func<IRuleBodySetter<TValue>, Rule<TValue>> anotherBuilding);

    IRuleOperationSetter<TValue> Or(Rule<TValue> another);

    IRuleOperationSetter<TValue> Or(Func<IRuleBodySetter<TValue>, Rule<TValue>> anotherBuilding);

    IRuleOperationSetter<TValue> Not();
}
```

Рис. 1.25 – Методи інтерфейсу IRuleOperationSetter.

Під час налаштування логічної операції можна створити складнішу формулу, яка міститиме операції заперечення, логічного «або» чи логічного «і». Крім того, до цього етапу налаштування можна повернутись, уже створивши об’єкт Rule для того, щоб описати формулу алгебри логіки за допомогою FluentApi.

Для об’єднання двох правил у бінарні зв’язки використовуються методи Or, And які можуть параметром приймати як уже налаштоване правило, так і делегат Func[12], що створює можливість передати параметром інструкцію налаштування правила.

У випадку, коли не потрібно створювати зв’язків між іншими правилами, можна завершити ініціалізацію і отримати об’єкт Rule, який можна використовувати для валідації.

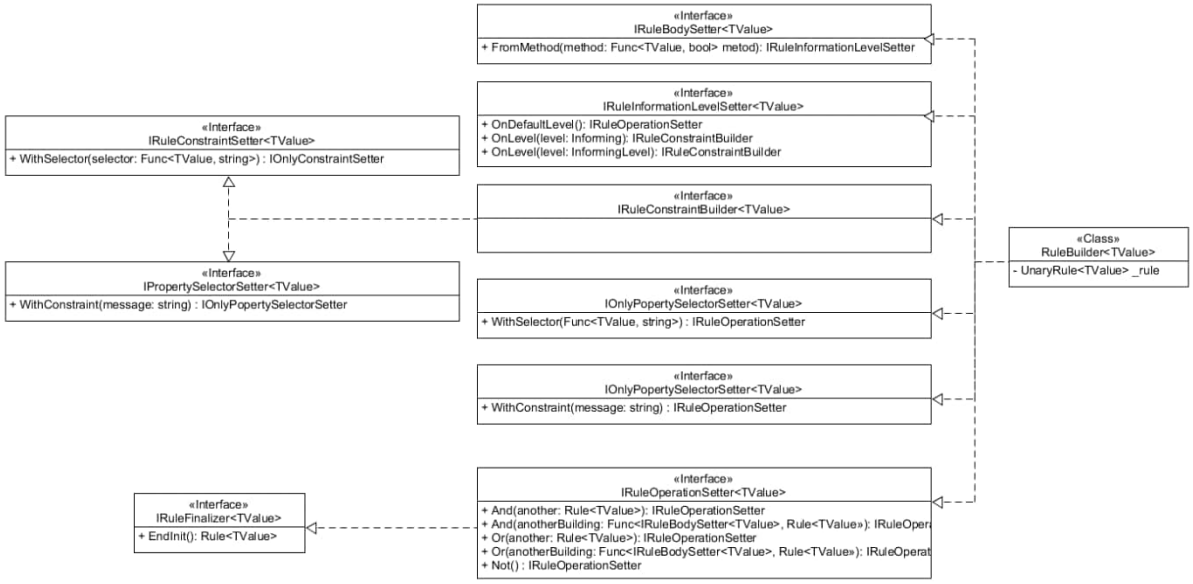

Рис. 1.26 – Ієрархія інтерфейсів RuleBuilder.

На рисунку 1.26 продемонстровано зв’язки між інтерфейсами, описаних у цьому розділі. Кожен інтерфейс відповідає за певний етап налаштування об’єкту Rule. Для розуміння, яким чином можна викликати методи під час створення правила, наведемо діаграму станів на рисунку 1.26.

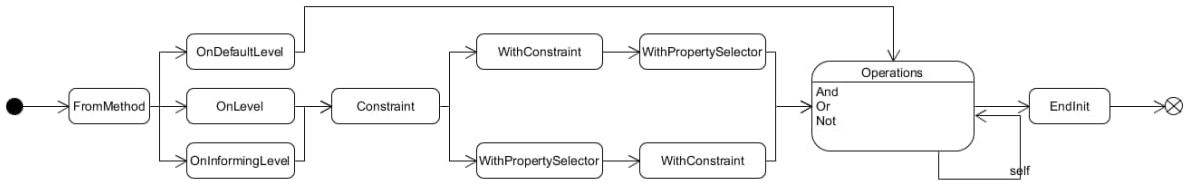

Рис. 1.27 – Діаграма станів для етапів створення об’єкту Rule.

На основі методу валідації числа, яке перевіряє, чи цілочислове значення є більшим, за деяке число, можна перевірити правилу послідовність етапів налаштування правила.

Рис. 1.28 – Підказки під час налаштування правила

```
public static Rule<int> GraterThan(int value, InformingLevel level = InformingLevel.Ignore)
    => Rule<int>.BeginInit()
                .FromMethod(v => v > value)
                .OnLevel(level)
                .WithConstraint($"Value must be grater than {value}")
                .WithPropertySelector(v => v.ToString())
                .EndInit();
```

Рис. 1.29 – Налаштоване правило для перевірки для типу int.

```
public static Rule<int> GraterThan(int value, InformingLevel level = InformingLevel.Ignore)
    => Rule<int>.BeginInit()
                .FromMethod(v => v > value)
                .OnLevel(level)
                .WithPropertySelector(v => v.ToString())
                .WithConstraint($"Value must be grater than {value}")
                .EndInit();
```

Рис. 1.30 – Змінений порядок викликів налаштування опису обмеження.

Використовуючи підхід FluentApi, створено систему для побудови різних об’єктів, які мають єдиний початок побудови (метод BeginInit). Цей підхід реалізації бібліотеки дозволяє контролювати дії розробника та підказувати йому, які методи потрібно використовувати під час налаштування маршрутизаці.

РОЗДІЛ 2. СИСТЕМА ОБМЕЖЕНЬ

Під час маршрутизації користувацьких запитів потрібно перевірити вхідні данні у методи так, щоб при виклику цього методу не виникла помилка. До прикладу, якщо метод містить параметр, який є знаменником числа, на нього потрібно поставити обмеження, щоб число, яке отримується з командного рядка чи рядка-повідомлення мессенджера не було рівне нулю і користувач про це знав.

Проте деякі параметри можуть потребувати обмежень, які складаються із логічних операцій. Прикладом може слугувати метод перевірки чи число знаходиться у певних межах. Таке може використовуватись, якщо потрібно вказати години, хвилини чи секунди. Якщо параметром є значення годин, то це число повинно бути у межах між 0 та 24. У випадку, коли параметр відображає значення хвилин або секунд, то це значення є правильним на проміжку від 0 до 60. Таким чином, правило, яке відповідає за перевірку годин складається з логічного «і»: число більше нуля *і* менше 24.

2.1 Створення логічних формул

У розділі 1.4 розглядався етап налаштування логічних формул, за який відповідав інтерфейс IRuleOperationSetter. Проте, якщо правило уже створене та на основі нього потрібно створити правило, що містить операцію «і», «або» чи заперечення варто використати метод InitOperations() об’єкту Rule.

```
public IRuleOperationSetter<TValue> InitOperations()
    => new RuleBuilder<TValue>(this);
```

Рис. 2.1 – Метод InitOperations.

З рисунку зрозуміло, що цей метод повертає об’єкт типу RuleBuilder на основі самого правила, проте повертається інтерфейс IRuleOperationSetter, що дозволить розробнику додати операції Or, And чи Not.

```
public static Rule<int> IsValidHours(InformingLevel level = InformingLevel.Ignore)
    => EqualBetween(0, 24, level);

public static Rule<int> EqualBetween(int lower, int grater, InformingLevel level = InformingLevel.Ignore)
    => GraterOrEqual(Math.Min(lower, grater), level).InitOperations()
                                                    .And(LessOrEqual(Math.Max(lower, grater), level))
                                                    .EndInit();
public static Rule<int> GraterOrEqual(int value, InformingLevel level = InformingLevel.Ignore)
    => GraterThan(value, level).InitOperations()
                               .Or(EqualTo(value, level))
                               .EndInit();
public static Rule<int> LessThan(int value, InformingLevel level = InformingLevel.Ignore)
    => Rule<int>.BeginInit()
                .FromMethod(v => v < value)
                .OnLevel(level)
                .WithConstraint($"Value must be less than {value}")
                .WithPropertySelector(v => v.ToString())
                .EndInit();
```

Рис. 2.2 – Реалізація перевірки чи параметр є значенням години.

До прикладу правило, яке перевірятиме параметр значення години базується на правилах, що перевіряють, чи число є меншим або більшим за вказане. Після чого, потрібно об’єднати їх з правилами, що вказують, що число є рівним до даного. Це реалізовується за допомогою логічного «або». Після чого ці два об’єкти типу Rule знову пов’язуються за допомогою логічного «і».

На рисунку 2.2 продемонстроване створення об’єкту Rule для цілочислового типу, яке перевіряє, чи параметр є в межах від нуля до двадцяти чотирьох включно.

```
var rule = IntRules.IsValidHours(InformingLevel.Error);

for (var i = -1; i <= 25; ++i)
{
    Console.WriteLine($"{i} - {rule.Validate(i)}");
    rule.DisplayMessage(i);
}
```

Рис. 2.3 – Реалізація перевірки чи параметр є значенням години.

На рисунку 2.3 зображена перевірка правильності описаного правила за допомогою підходу FluentAPI на проміжку від -1 до 25 включно.

```
-1 - False
-1 is not fit constraint `Value must be grater than 0`.
0 - True
1 - True
2 - True
3 - True
4 - True
5 - True
6 - True
7 - True
8 - True
9 - True
10 - True
11 - True
12 - True
13 - True
14 - True
15 - True
16 - True
17 - True
18 - True
19 - True
20 - True
21 - True
22 - True
23 - True
24 - True
25 - False
25 is not fit constraint `Value must be less than 24`.
```

Рис. 2.4 – Результат перевірки, виведений у консолі.

На вказаному проміжку усі значення, які є правильним значенням години після виконання методу Validate() повертають булеве значення True, однак -1 та 25 не пройшли перевірку та повідомлення причини вказано червоним кольором.

Значення -1 не є правильним, оскільки не відповідає обмеженню «Число повинне бути більшим за 0».

Число 25 не пройшло перевірку за причиною «Значенням повинно бути меншим за 24».

2.2 Види правил

Щоб правильно створити логічні операції, кожна операція є наслідником абстрактного класу Rule<>, якщо операція бінарна, то ця операція наслідує абстрактний клас BinaryRule<>, який у свою чергу є дочірнім класом Rule.

Для розуміння ієрархії об’єктів правил створено діаграму класів, що зображена на рисунку 2.5.

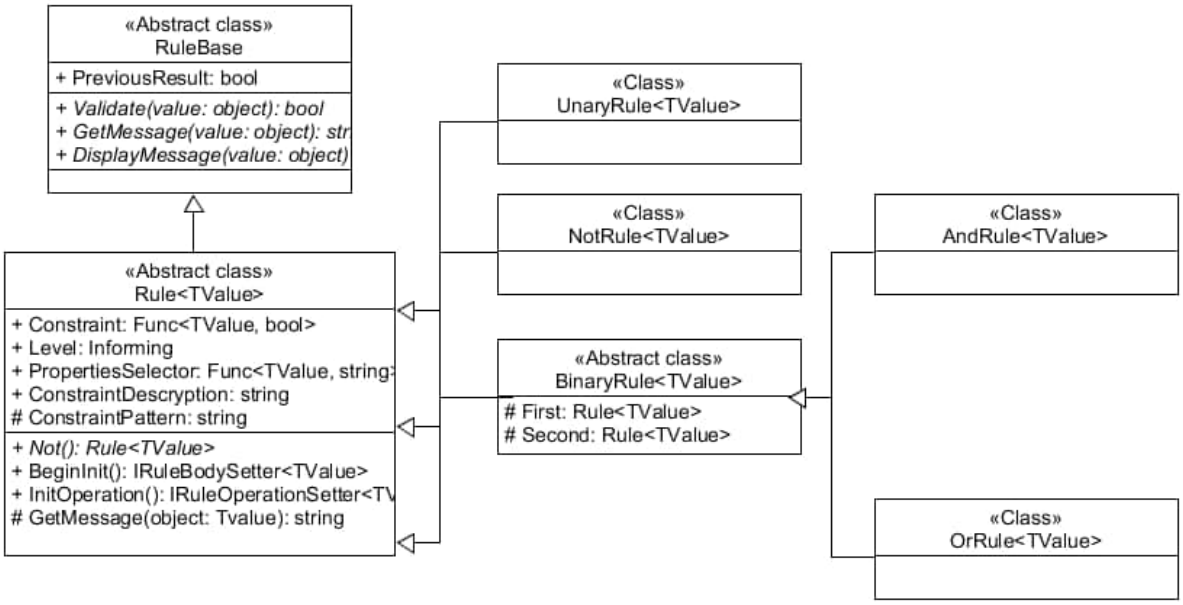

Рис. 2.5 – Ієрархія об’єктів для валідації.

Така ієрархія дозволить додавати нові типи зв’язку між операндами (UnaryRule, NotRule) такими як еквіваленція[14], виключне «або» чи імплікація[15].

2.3 Рівні інформування

Одним із основних завдань перевірки є вказання причини, чому значення, яке знаходиться у запиті, не може бути використане. Щоб розробник міг встановити власний підхід до повідомлення було реалізовано систему інформування.

В основі цієї системи лежить абстрактний клас Informing, який містить єдиний абстрактний метод OnError, що приймає повідомлення-причину, чому значення є неправильним.

У таблиці 2.1 описано рівні, які розроблено за замовчуванням.

Таблиця 2.1. – Рівні інформування за замовчуванням.

Назва класу	Опис реалізації	Зображення в переліку	Відповідний метод
Ignore	Нічого не виконує.	Ignore	InformingLevels.Ignore
Information	Виводить на консоль повідомлення.	Information	InformingLevels.Info
Warning	Виводить на консоль повідомлення, змінюючи колір тексту на жовтий.	Warning	InformingLevels.Warn
Error	Виводить на консоль повідомлення, змінюючи колір тексту на червоний.	Error	InformingLevels.Error
Critical	Створює виняткову ситуацію та генерує об’єкт Exception з повідомленням, отриманим від правила.	Critical	InformingLevels.Critical

Розробник може реалізувати клас, який наслідує клас Informing та перевизначити абстрактний метод OnError.

РОЗДІЛ 3. МЕТАПРОГРАМУВАННЯ

Одна із проблем розширення програми – створення однотипних класів, які згідно архітектури відповідають за вирішення різних завдань, проте в їх основі знаходиться єдиний програмний код. До прикладу створення атрибутів обмеження та валідації на основі певних правил або тестування наявних методів для певності, що функціональність реалізована правильно.

Вирішенням цієї проблеми є концепція метапрограмування – це підхід, коли під час виконання коду генерується інший програмний код[16]. Щоб не виконувати монотонну роботу створення подібних за структурою типів або тестування написаних правил, можна використати атрибути Convert та Test.

3.1 Генерація коду

Генерування програмного коду є застосовним до статичного, публічного, не абстрактного методу, який повертає об’єкт правила. Якщо атрибут додано до методу, який задовольняє ці умови, то відповідні файли вдасться згенерувати. У проєкті, який має доступ до типів, які видимі лише у певній збірці (модифікатор доступу internal [17]), можна команду MetaprogrammingManager.Generate().

Цей статичний метод дозволить опрацювати усі методи, які містять атрибути Convert та Test, перевірити на виконання умов та згенерувати потрібний код.

Розробник, який користуватиметься бібліотекою не матиме можливості використати цю команду, проте при оновлені бібліотеки та додаванні нових обмежень цей метод доступний.

Щоб під час доповнення бібліотеки отримати доступ до методу Generate класу MetaprogrammingManager потрібно додати атрибут на об’єкт Assembly[18].

```
[assembly: InternalsVisibleTo("FluentRequest.Console")]

MetaprogrammingManager.Generate();
```

Рис. 3.1 – Надання доступу до об’єктів, які видимі лише у поточній збірці.

У методі Generate серед усіх збірок знаходяться методи, які містять атрибути метапрограмування: Convert, Test, після чого кожен метод на основі наявних атрибутів може згенерувати файл, що містить потрібний код (визначення атрибуту валідації чи обмеження або тестування).

Перевірка, чи метод може згенерувати код, здійснюється на основі FluentApi валідації. Це дозволить відслідковувати, який метод не відповідає певній умові.

```
private static readonly Rule<MethodInfo> _methodRules =
   MethodInfoRules.ContainsAttribute<ConvertAttribute>()
                  .InitOperations()
                  .Or(MethodInfoRules.ContainsAttribute<TestAttribute>())
    .EndInit()
    .InitOperations()
                 .And(MethodInfoRules.IsPublic(InformingLevel.Warning).InitOperations()
                 .And(MethodInfoRules.IsStatic(InformingLevel.Warning))
                 .And(MethodInfoRules.IsAbstract(InformingLevel.Warning).Not())
                 .And(MethodInfoRules.HasReturningType(typeof(RuleBase), InformingLevel.Warning)).EndInit())
    .EndInit();
```

Рис. 3.2 – Правило перевірки чи метод може згенерувати код.

3.2 Конвертування

Атрибут Convert можна додати лише до методу за рахунок вказання AttributeUsage.Method[19]. Цей атрибут дозволяє створити файли, які можна накладати на параметри методів, що використовуватимуться у маршрутизації.

На основі прикладу із розділу 2 розглядається генерація атрибутів IsHourConstraintAttribute чи IsHourValidationAttribute.

```
[Convert]
public static Rule<int> IsHours(InformingLevel level = InformingLevel.Ignore)
    => EqualBetween(0, 24, level);
```

Рис. 3.3 – Додавання атрибуту для генерації файлів.

На основі інформації про метод отримуються потрібні дані, що дозволить правильно згенерувати файл.

Таблиця 3.1 – Інформація про метод та її використання атрибутом Convert.

Назва	Спосіб отримання	Використання при генерації
Назва методу	methodInfo.Name	Виклик методу для ініціалізації обмеження; Найменування файлу;
Набір параметрів	methodInfo.GetParameters()	Ініціалізація конструктора атрибуту; Додавання потрібних просторів імен; Виклик методу з параметрами.
Назва класу	methodInfo.DeclaringType.Name	Виклик статичного методу; Найменування простору імен; Найменування папки;

Для отримання інформації з методу використовується клас Decomposer. Цей клас дозволяє використовувати делегат Func, який приймає MethodInfo та задає алгоритм отримання потрібної інформації.

Перетворення об’єкту з якого потрібно отримати потрібну інформацію за допомогою декомпозиції починається з використання методу розширення Decompose.

```
public class Decomposer<TTarget>
{
    private readonly TTarget _target;


    public Decomposer(TTarget value)
    {
        _target = value;
    }


    public Decomposer<TTarget> Get<TResult>(Func<TTarget, TResult> projection, out TResult result)
    {
        result = projection(_target);
        return this;
    }
}
public static Decomposer<TTarget> Decompose<TTarget>(this TTarget target)
    => new Decomposer<TTarget>(target);

method.Decompose()
      .Get(m => m.Name, out var methodName)
      .Get(m => m.GetParameters().ToArray(), out var parameters)
      .Get(m => m.DeclaringType.Name, out var className)
      .Get(m => $"{m.DeclaringType.Name}.{m.Name}", out var calling);
```

Рис. 3.4 – Реалізація та приклад використання Decomposer.

Після запуску команди MetaprogrammingManaget.Generate() у папці IntRulesAttributes/ConstraintAttributes було створено файл з назвою, яка базується на назві методу IsHoursConstraintAttribute з розширенням cs.

У папці ValidationAttributes також створився файл, який містить інформацію про правило для перевірки та обмеження параметру методу.

```
using FluentRequests.Lib.Validation.Error;
using FluentRequests.Lib.Validation.Rules;

namespace FluentRequests.Lib.Attributes.ParameterAttributes.IntRulesAttributes.ConstraintAttributes
{

    public class IsHoursConstraintAttribute: ConstraintAttribute
    {

        public IsHoursConstraintAttribute(InformingLevel level = InformingLevel.Ignore)
             => Rule = IntRules.IsHours(level);
    }
}
using FluentRequests.Lib.Validation.Error;
using FluentRequests.Lib.Validation.Rules;

namespace FluentRequests.Lib.Attributes.ParameterAttributes.IntRulesAttributes.ValidationAttributes
{

    public class IsHoursValidateAttribute: ValidateAttribute
    {

        public IsHoursValidateAttribute(InformingLevel level = InformingLevel.Ignore)
             => Rule = IntRules.IsHours(level);
    }
}
```

Рис. 3.5 – вміст згенерованих файлів.

3.3 Тестування

Створені правила варто протестувати, чи вони правильно описані та на очікуваних значеннях об’єкту видають очікуваний результат. Однак створення нових класів для тестування, реалізація методів перевірки, додавання тестових випадків є однотипним. Для спрощення цього процесу є атрибут Test, який дозволить згенерувати файл у проєкті який базується на бібліотеці NUnit[20].

На основі методу, написаного у розділі 2 продемонстровано генерацію файлів для тестування об’єктів типу Rule.

До методу IsHours додано атрибут Test з вказанням правильних та неправильних тестових випадків.

```
[Test(validTestCases:new string[] {"0", "2", "7", "13", "24"},
      invalidTestCases:new string[] { "-1", "25", "104", "-404", "505"})]
public static Rule<int> IsHours(InformingLevel level = InformingLevel.Ignore)
    => EqualBetween(0, 24, level);
```

Рис. 3.6 – Додавання атрибуту Test до методу IsValidHours.

Метод, до якого додається атрибут Test повинен відповідати правилам, описаних у розділі 3.2.

Атрибут Test використовує інформацію про метод і також використовує клас Decomposer для отримання потрібних значень.

Таблиця 3.2 – Інформація про метод та її використання атрибутом Test.

Назва	Спосіб отримання	Використання при генерації
Назва методу	methodInfo.Name	Найменування файлу; Виклик для тестування; Найменування методу тестування.
Параметри методу	methodInfo.GetParameters()	Виклик методу для створення правила з параметрами; Додавання потрібних просторів імен;
Назва класу	methodInfo.DeclaringType.Name	Додавання директорії тестування; Ініціалізація простору імен тестування;

Параметри атрибуту Test відповідають за створення TestCase атрибутів до методу тестування. Це дозволяє для різних випадків протестувати правило (об’єкт Rule).

Після запуску команди MetaprogrammingManager.Generate() створюється файл, який тестує функціональність.

```
public partial class IntRulesTest
{

    [Test]
    [TestCase(0)]
    [TestCase(2)]
    [TestCase(7)]
    [TestCase(13)]
    [TestCase(24)]
    public void IntRules_IsHours_ValidValue(object value)
        => Assert.That(IntRules.IsHours().Validate(value), Is.True);


    [Test]
    [TestCase(-1)]
    [TestCase(25)]
    [TestCase(104)]
    [TestCase(-404)]
    [TestCase(505)]
    public void IntRules_IsHours_InvalidValue(object value)
        => Assert.That(IntRules.IsHours().Validate(value), Is.False);
}
```

Рис. 3.7 – Згенерований клас тестування.

Згенеровані класи містять модифікатор partial – модифікатор, що дозволяє розміщувати один клас у різних файлах [21], оскільки різні методи тестування знаходитимуться в одному і тому самому класі у якому методи визначені з додаванням приписки “Test”.

Рис. 3.8 – результат тестування на основі згенерованих файлів.

Файли для тестування, що були створені у наслідок використання атрибутів метапрограмування можна виконати. На рисунку 3.8 зображено результат виконання згенерованих тестів для методу IsHours.

Можна вказати номер тестування та змінити параметр ініціалізації, що дозволить протестувати метод для різних значень конструювання правил Rule.

```
[Test(validTestCases: new string[] { "1", "2", "3", "4", "5" },
      invalidTestCases: new string[] { "-1", "-2", "-3", "-4", "-5" },
      initParameters: new string[] { "0" })]

public partial class IntRulesTest
{
    [Test]
    [TestCase(3)]
    [TestCase(4)]
    [TestCase(5)]

    public void IntRules_GraterThan_ValidValue_2(object value)
        => Assert.That(IntRules.GraterThan(2).Validate(value), Is.True);

    [Test]
    [TestCase(1)]
    [TestCase(2)]
    [TestCase(-1)]
    [TestCase(-2)]
    [TestCase(-3)]
    [TestCase(-4)]
    [TestCase(-5)]

    public void IntRules_GraterThan_InvalidValue_2(object value)
        => Assert.That(IntRules.GraterThan(2).Validate(value), Is.False);
}
```

```
[Test(testNumber: 1,
      validTestCases: new string[] { "3", "4", "5" },
      invalidTestCases: new string[] { "1", "2", "-1", "-2", "-3", "-4", "-5" },
      initParameters: new string[] { "2" })]
public partial class IntRulesTest
{
    [Test]
    [TestCase(1)]
    [TestCase(2)]
    [TestCase(3)]
    [TestCase(4)]
    [TestCase(5)]

    public void IntRules_GraterThan_ValidValue(object value)
        => Assert.That(IntRules.GraterThan(0).Validate(value), Is.True);

    [Test]
    [TestCase(-1)]
    [TestCase(-2)]
    [TestCase(-3)]
    [TestCase(-4)]
    [TestCase(-5)]

    public void IntRules_GraterThan_InvalidValue(object value)
        => Assert.That(IntRules.GraterThan(0).Validate(value), Is.False);
}
```

Рис. 3.9– Згенерований тест для методу, що містить параметр.

РОЗДІЛ 4. АТРИБУТИ МАРШРУТИЗАЦІЇ

Окрім ввідного способу створення об’єктів, що відповідатимуть за маршрутизацію користувацького запиту, реалізовано підхід, у якому за допомогою атрибутів[22] класи та методи додаються до реєстру. Це реалізовано з метою отримання альтернативного підходу до побудови системи опрацювання запитів користувача.

4.1 Атрибут команди

Використовуючи атрибут Command, можна додати клас у систему маршрутизації користувацьких запитів. Щоб правильно створити обробник користувацького запиту потрібно до основі публічного, не абстрактного класу додати відповідні метадані – назву команди та допомогу.

Обмеження на атрибут команди встановлюється на основі системи правил, що перевіряє властивості типу.

Таблиця 4.1 – Обмеження типу з атрибутом команди.

Перевірка	Рівень інформування	Опис помилки
Наявність атрибуту Command	Ignore	Відсутній.
Наявність атрибуту Help	Critical	Тип, що містить атрибут команди повинен містити атрибут допомоги.
Абстрактність типу	Critical	Тип-ціль не повинен бути абстрактним.
Шаблонність типу	Critical	Тип не є шаблонним.

Усі обмеження, окрім наявності атрибуту Command є критичним, тобто розробник під час використання маршрутизаторів, згенерованих на основі метаданих отримає сповіщення, що деякий клас не може використовуватись при опрацюванні запиту користувача через обмеження, які мають рівень інформування Critical.

Оскільки система маршрутизації є ієрархічною – команди можуть містити підкоманди, – то даний тип перевіряється на наявність внутрішніх класів, які також містять атрибут Command.

Обов’язкова умова, яка не входить в первинну перевірку, опрацьовується після отримання інформації про внутрішні типи чи методи, які можуть застосовуватись при маршрутизації. Якщо такий тип не містить перевантаження, тоді програма згенерує виняткову ситуацію, щоб розробник додав перевантаження.

```
[AttributeUsage(AttributeTargets.Class)]
public class CommandAttribute : Attribute
{

    internal string CommandName { get; set; }


    public CommandAttribute(string commandName)
    {
        CommandName = commandName;
    }
}

public Register AddFromAttributes()
    => RegistrationManager.GetRegisterFromAttribute(this);

public static Register GetRegisterFromAttribute(Register source = null)
{
    var commandTypes = AppDomain.CurrentDomain.GetAssemblies()
        .SelectMany(a => a.GetTypes())
        .Where(t => _isValidRegistrationType.ValidateWithError(t))
        .Select(t => GetCommandFromType(t))
        .ToArray();

    var register = source ?? new Register();

    foreach (var command in commandTypes)
        register.Add(command);

    return register;
}
```

Рис. 4.1 – Атрибут Command та його опрацювання.

4.2 Атрибут перевантаження

Додавання методу як обробника маршрутизації відбувається на основі атрибуту OverloadAttribute. Цей атрибут повинен міститись у методі, якщо клас, де цей метод визначений містить метадані про створення об’єкту Command. Додавання цього атрибуту також перевірятиметься за допомогою системи накладання правил.

Таблиця 4.2 – Правила для створення перевантаження.

Перевірка	Рівень інформування	Опис помилки
Наявність атрибуту Overload	Ignore	Відсутній.
Наявність атрибуту Help	Critical	Перевантаження повинно містити опис.
Публічність методу	Critical	Метод повинен бути з модифікатором public.
Статичність методу	Critical	Метод повинен бути статичним.
Наявність при усіх параметрах атрибуту Help	Critical	Усі параметри перевантаження повинні містити опис.

Для того, щоб викликати метод він повинен бути статичним. У іншому випадку, потрібно створити об’єкт, який містить цей метод, що додає обмеження типу – наявність конструктору за замовчуванням.

Також обмеження на параметри накладається згідно необхідності встановлення допомоги при створенні маршрутизаторів за допомогою FluentAPI.

Створення параметрів відбувається на основі інформації чи параметр є не обов'язковим. Складнощі з створенням об'єктів типу Argument полягає в шаблонності об'єкту. Динамічно утворити Required чи Optional аргумент потрібно на основі інформації про тип параметру. У випадку з опціональним аргументом потрібно ще використати значення властивості DefaultValue.

Після додавання параметрів додається і саме тіло перевантаження. Оскільки метод може містити довільну кількість параметрів різних типів, потрібно їх звести до методу, який приймає список об'єктів ArgumentBase. За допомогою методу розширення створюється делегат Func, який повертає результат у вигляді string.

```
private static readonly Rule<MethodInfo> _isValidRegistrationOverloadsRule
    = MethodInfoRules.ContainsAttribute<OverloadAttribute>()
    .InitOperations()
    .And(MethodInfoRules.ContainsAttribute<HelpAttribute>(InformingLevel.Critical))
    .And(MethodInfoRules.IsPublic(InformingLevel.Critical))
    .And(MethodInfoRules.IsStatic(InformingLevel.Critical))
    .And(rb => rb.FromMethod(m => m.GetParameters().All(p => p.GetCustomAttribute<HelpAttribute>() != null))
                 .OnLevel(InformingLevel.Critical)
                 .WithConstraint($"All parameters of the method must be marked with {nameof(HelpAttribute)}")
                 .WithPropertySelector(m => m.Name)
                 .EndInit())
    .EndInit();


[AttributeUsage(AttributeTargets.Method)]
internal class OverloadAttribute : Attribute
{
}


private static Overload GetOverloadFromMethod(MethodInfo methodInfo)
{
    var methodParameters = methodInfo.Decompose()
                                     .Get(m => m.GetCustomAttribute<HelpAttribute>(), out var helpAttribute)
                                     .Get(m => m.GetParameters());

    var overload = Overload.BeginInit()
        .WithHelp(helpAttribute.Help)
        .WithBody(methodInfo.ToRoutingMethod())
        .EndInit();

    foreach(var parameter in methodParameters)
        overload = parameter.ToRoutingParameter(overload);

    return overload;
}
```

Рис. 4.2 – Атрибут Overload та його використання.

4.3 Атрибут Help

Кожен елемент маршрутизації користувацьких запитів повинен містити інформацію та опис про те, для чого він створений. Цей атрибут є застосовним до класів, методів та параметрів – об’єктів маршрутизації, налаштованих за допомогою атрибутів.

Отримана інформацію встановлюється за допомогою відповідного методу конструювання, описаних у розділі 1 (1.1-1.3), після чого можна продовжити створення об’єктів маршрутизації.

На основі цього атрибуту реалізована команда надання допомоги. Детальніше використання вбудованих команд розглядається у розділі 5.

РОЗДІЛ 5. РЕЄСТРУВАННЯ

Для додавання команд реалізується на основі FluentApi та атрибутів. Для усієї програми наявний лише один реєстр, на основі якого будується вбудована система надання інформацію про команди, вихід з консольного додатку та налаштування консолі.

5.1 Додавання маршрутизаторів за допомогою атрибутів

Визначивши клас, який буде використовуватись для маршрутизації – додано необхідні атрибути, описані у розділі 4 – розробник має можливість додати команди з усієї збірки чи на основі одного окремого типу.

Для того, щоб додати такий обробник користувацького запиту, потрібно від реєстру викликати метод AddFromType, вказавши тип, який перетвориться на об’єкт команди.

```
RegistrationManager.RoutingRegister.AddFromType<ConsoleCommand>()
                                   .AddFromType<HelpCommand>();

[Command("console")]
[Help("Provides access to the console functions.")]
public class ConsoleCommand
{
    [Command("cls")]
    [Help("Cleans console.")]
    public class ClearConsole...

    [Command("color")]
    [Help("Handle color of the console")]
    public class ColorHandling...
}

[Command("help")]
[Help("Provides information for the all used commands.")]
public class HelpCommand
{
    [Overload]
    [Help("Gets help for the command with selected name")]
    public static string GetCommandHelp([Help("Name of the command.")] string commandName,
                                        [Help("Defines is all command must be displayer with their inner commands and overloads.")] bool all = false)
    {
        var command = RegistrationManager.RoutingRegister.Commands.FindCommandByName(commandName);

        return (all ? command?.ToInnerString() : command?.ToString()) ?? $"Cannot find command with name {commandName}";
    }

    [Overload]
    [Help("Gets help for the all roots commands or all commands.")]
    public static string GetCommandHelp([Help("Defines is all command must be displayer with their inner commands and overloads.")] bool all = false)
        => all
        ? $"All possible commands:\n\t{string.Join("\n\t", RegistrationManager.RoutingRegister.Commands.Select(c => c.ToInnerString()))}"
        : $"All root commands:\n\t{string.Join("\n\t", RegistrationManager.RoutingRegister.Commands.Select(c => $"{c.Name.ToUpper()} - {c.Help}"))}";
}
```

Рис. 5.1 – Додавання та приклад атрибутної маршрутизації.

Після додавання цих обробників користувач їх можна викликати.

На основі наявних команд формується інформація про допомогу до кожної команди. При написанні у командний рядок слова «help», користувач отримає список усіх корінних команд системи. Якщо вказати параметр -all рівним true, то користувач зможе отримати усю інформацію про параметри, перевантаження та внутрішні команди.

```
help
All root commands:
        CONSOLE - Provides access to the console functions.
        HELP - Provides information for the all used commands.
help -all=t
All possible commands:
        Information for command console:
        InnerCommands:
                Information for command cls:
                Overloads:
                        () - Cleans console.
                Information for command color:
                InnerCommands:
                        Information for command foreground:
                        Overloads:
                                (ConsoleColor consoleColor, bool cls = False) - Changes foreground color of the console.
                                (bool cls = False) - Resets foreground color of the console.
                        Information for command background:
                        Overloads:
                                (ConsoleColor background, bool cls = False) - Changes background color of the console.
                                (bool cls = False) - Resets background color for the console.
                        Information for command reset:
                        Overloads:
                                (bool cls = False) - Resets all color to default console values.
        Information for command help:
        Overloads:
                (string commandName, bool all = False) - Gets help for the command with selected name
                (bool all = False) - Gets help for the all roots commands or all commands.
```

Рис. 5.2 – Приклад виклику допомоги.

Також користувач матиме можливість додати усі обробники, які містяться у поточній збірці. За допомогою методу AddFromAssembly з вказанням типу, який розташований у цій збірці буде проведено повний аналіз типів, які відповідають правилам, описаними у розділі 4.

```
RegistrationManager.RoutingRegister.AddFromAssembly<ConsoleCommand>();
help
All root commands:
        CONSOLE - Provides access to the console functions.
        EXIT - Exit the application
        HELP - Provides information for the all used commands.
help help -all=t
Information for command help:
        Overloads:
                (string commandName, bool all = False) - Gets help for the command with selected name
                (bool all = False) - Gets help for the all roots commands or all commands.
```

Рис. 5.3 – Додавання усіх обробників, які містяться у збірці.

Якщо провести додавання усіх типів зі збірки, а потім – додати окремий тип, то виникне помилка, яка зазначить, що така команда уже була додана.

5.2 Додавання команд через FluentApi

Окрім визначених типів за допомогою атрибутів, зберігається можливість налаштувати маршрутизацію за допомогою розробленої системи, описаній у главі 1.

За допомогою методів Add, які можуть приймати як сам об’єкт команд так і об’єкт конструктора команди, можна додати обробники запиту користувача до загального реєстру.

На основі простих математичних команд, які виконують додавання та чи множення чисел, розглянуто приклад додавання обробників до системи.

```
RegistrationManager.RoutingRegister
    .Add(mullCommand)
    .Add(cb => cb.WithName("add")
                 .WithHelp("Adds float values.")
                 .AddOverload(addTwoOverload)
                 .EndInit())
    .AddHelpBuiltIn();
```

Рис. 5.4 – додавання маршрутизаторів через FluentAPI.

Ці команди автоматично можна відобразити за допомогою вбудованого обробника Help.

```
help
All root commands:
        MULL - Evaluates multiplication.
        ADD - Adds float values.
        HELP - Provides information for the all used commands.
        CONSOLE - Provides access to the console functions.
help mull -all=t
Information for command mull:
        Overloads:
                (double first, double second) - Mulls two parameters
help add -all=t
Information for command add:
        Overloads:
                (float first, float second) - Adds two parameters
```

Рис. 5.5 – відображення допомоги доданих маршрутизаторів.

РОЗДІЛ 6. МАРШРУТИЗАЦІЯ ЗАПИТУ

Для відображення відповіді користувачу була використано патерн стану[23], який кожну команду на можливість виклику. За рахунок застосування цього підходу кожна команда під час маршрутизації міститиме деякий коефіцієнт, який відповідатиме за найбільш ймовірне знаходження обробників. Таким чином, користувач отримуватиме інформацію, що саме він написав не так та як можна змінити запит, щоб отримати потрібну реалізацію.

6.1 Особливості ієрархії

Оскільки маршрутизація стосуватиметься декількох видів обробників, а саме Command та Overload, реалізовано дві гілки ієрархії: одні стани відповідають за опрацювання Command, Інші – Overload.

Окрім цього реалізовано гілку станів термінального (екстреного) завершення маршрутизації – тобто сам потрібний обробник не було знайдено, проте можливості продовжити його шукати уже нема. Такий підхід використовується для того, щоб в залежності від термінального стану користувач отримував потрібну інформацію про причину, чому саме не можна встановити відповідність між запитом та обробниками.

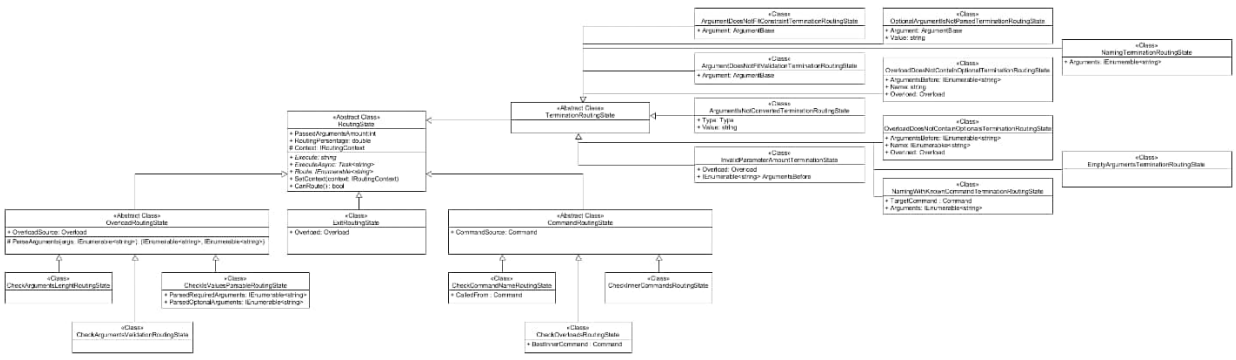

Рис. 6.1 – Ієрархія класів станів маршрутизації.

Чіткіше зображення наведено у додатку А.

6.2 Зв’язки між станами

6.2.1 Початковий стан Check Command Name

Початковим станом маршрутизації кожної команди є CheckCommandName, який перевіряє, чи введений параметр відповідатиме певній команді. Якщо параметрів узагалі нема, то виведеться помилка, що користувач не ввів команди і запропонується скористатись командою help для отримання довідки. Це є термінальний стан, який зупиняє маршрутизацію. Після чого наступних перевірок не відбувається.

```
There is no arguments. Use help to find out possible commands
```

Рис 6.2 – Спрацювання термінального стану EmptyArguments.

У випадку, коли ім’я не було опрацьовано, маршрутизація переходить в термінальний стан. В залежності, чи це внутрішня команда чи корінна – команда, яка не є внутрішньою командою іншої – встановлюється відповідний обробник.

Якщо команда є корінною, то маршрутизація переходить у термінальний стан NamingTerminationRoutingState, який відображатиме помилку, що такої команди загалом не знайдено.

```
command that not exist
Cannot route `command that not exist` request. Use help to find out existing command
```

Рис. 6.3 – Повідомлення про неправильну корінну команду.

У іншому випадку користувач може отримати довідку про команду.

```
console color
Cannot find proper routing for `console color`.
May be you mean this command:
Information for command color:
        InnerCommands:
                Information for command foreground:
                Overloads:
                        (ConsoleColor consoleColor, bool cls = False) - Changes foreground color of the console.
                        (bool cls = False) - Resets foreground color of the console.
                Information for command background:
                Overloads:
                        (ConsoleColor background, bool cls = False) - Changes background color of the console.
                        (bool cls = False) - Resets background color for the console.
                Information for command reset:
                Overloads:
                        (bool cls = False) - Resets all color to default console values.
```

Рис. 6.4 – Відображення можливої команди.

6.2.2 Стан перевірки внутрішніх команд

Після встановлення відповідності користувацького запиту та певної команди маршрутизація зі стану перевірки імені переходить у стан перевірки внутрішніх команд.

Коли команда не містить внутрішніх команд, маршрутизація переходить у стан перевірки наявних перевантажень.

У іншому випадку, кожній внутрішній команді задається початковий стан та вказується, яка команда ініціює маршрутизацію внутрішніх. Після чого внутрішні команди сортуються за коефіцієнтом, який визначає успішність маршрутизації. Чим вищий коефіцієнт, тим дальше маршрутизація змогла пройти для встановлення відповідності між запитом та обробниками.

Під час сортування на першому місці опиняється найкращий спосіб встановлення відповідності, проте, якщо він є термінальним, то потрібно перевірити перевантаження даної команди.

У випадку, коли найкращий стан маршрутизації є заключним – знайдено правильне перевантаження – то станом цієї команди стає стан найкращої внутрішньої команди.

Особливістю стану CheckInnerCommands є те, що він не виходить в термінальний стан самостійно, він рекурсивно передає інформацію про стани внутрішніх команд на рівень вище, оскільки сама структура маршрутизації є деревоподібною.

6.2.3 Стан перевірки перевантажень

На початку цього стану перевіряється, чи команда узагалі має перевантаження. Якщо таких не знаходить, то маршрутизація переходить у стан найкращої внутрішньої або корінної команди.

Якщо перевантаження містяться у команді, то кожному перевантаженню встановлюється стан перевірки кількості аргументів після чого кожне перевантаження проходить маршрутизацію починаючи зі стану CheckArgumentsLenght .

Коли усі перевантаження були поставлені у відповідність з користувацьким запитом, вони сортуються за коефіцієнтом маршрутизації. Обирається найкращий результат і встановлюється його стан, навіть якщо він був термінальним.

6.2.4 Стан перевірки кількості аргументів

На цьому етапі перевіряються аргументи за кількістю. Спочатку проводиться зчитування параметрів та їх поділ на опційні та необхідні.

Ситуація, коли кількість зчитаних аргументів не відповідає кількості аргрументів, які містить перевантаження, користувач отримає помилку з довідкою про поточне перевантаження.

```
console color foreground red 2
Overload does not fit arguments amount. See more about this overload of console color foreground:
(ConsoleColor consoleColor, bool cls = False) - Changes foreground color of the console.
```

Рис. 6.5 – Відповідь системи при неправильній кількості параметрів.

Наступним кроком перевіряється, чи кількість зчитаних аргументів рівна 0, система переходить на заключний стан та може виконати тіло перевантаження.

```
console color foreground
Console foreground color was reset.
```

Рис. 6.6 – Виконання перевантаження без параметрів.

Коли маршрутизація продовжила свою роботу та не вийшла в один із термінальних чи заключний стан, перевіряються необов’язкові параметри.

Якщо команда загалом не містить ніяких параметрів зі значенням за замовчуванням, то маршрутизація перейде до термінального стану OverloadDoesNotContainOptionals, який виведе повідомлення про те, що така команда не містить ніяких необов’язкових параметрів та запропонує ознайомитись з можливим визначенням перевантаження.

```
mull 2.6d 2.7d -all=t -power=t -persicion=4
Overload does not contains any optional arguments. But `all`,`power`,`persicion` was provided. Find out more about the mull overload:
 (double first, double second) - Mulls two parameters
```

Рис. 6.7 – Повідомлення про відсутність опційних параметрів.

Якщо команда все-таки містить набір параметрів з наданим значенням, то перевіряється, чи назва параметру співпадає у рядку-запиті. Якщо ні, то система перейде у термінальний стан OverloadDoesNotContainOptional, який виведе інформацію про перевантаження та зазначить, що такого опційного аргументу не існує у перевантажені.

```
console color foreground red -all=t
Overload does not contain all optional argument. Find out more about the console color foreground overload:
 (ConsoleColor consoleColor, bool cls = False) - Changes foreground color of the console.
```

Рис. 6.8 – Повідомлення про відсутність наданого опційного параметру.

Останнім станом, який переводить систему до наступного етапу перевірки запиту користувача – перевірка перетворення об’єкту string до типу, який потрібен перевантаженню. Цей стан міститиме значення зчитаних опційних та необхідних параметрів.

6.2.5 Стан перевірки зчитування аргументів

Кожен параметр намагається конвертувати значення, яке прийшло з рядка-запиту користувача. Якщо якийсь із аргументів не можна зчитати, система перейде у стан ArgumentIsNotConverted. У цьому випадку користувачу виведеться повідомлення, яке зазначить, що не можливо перевести аргумент до даного типу і буде виведено інформацію про поточне перевантаження для ознайомлення з типом.

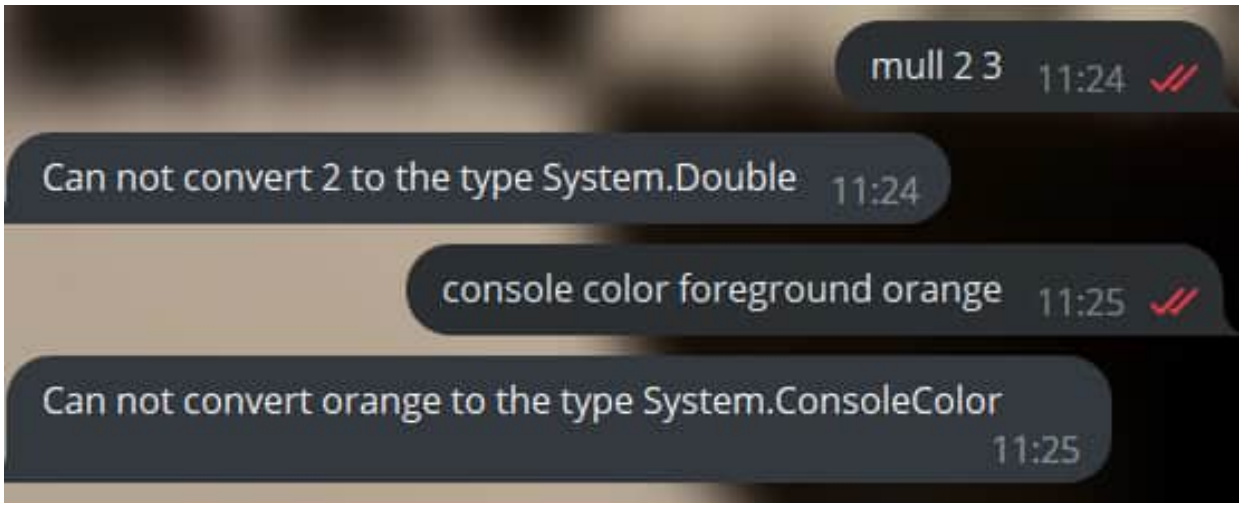

Рис. 6.9 – Повідомлення про помилку конвертації.

У випадку, коли користувач не передавав ніяких опційних аргументів система маршрутизації перейде на стан перевірки обмежень, які вказуються за допомогою методів FluentApi WithValidation та WithConstraint при побудові об’єкту Argument або за атрибутами параметрів Constraint чи Validation.

При ситуації з наявними аргументами зі значеннями за замовченням вони також перевірятимуться на можливість конвертування. У випадку, коли значення не вдасться конвертувати система перейде до термінального стану OptionalArgumentIsNotParsed, який дозволить користувачу побачити який саме опційний аргумент не було конвертовано.

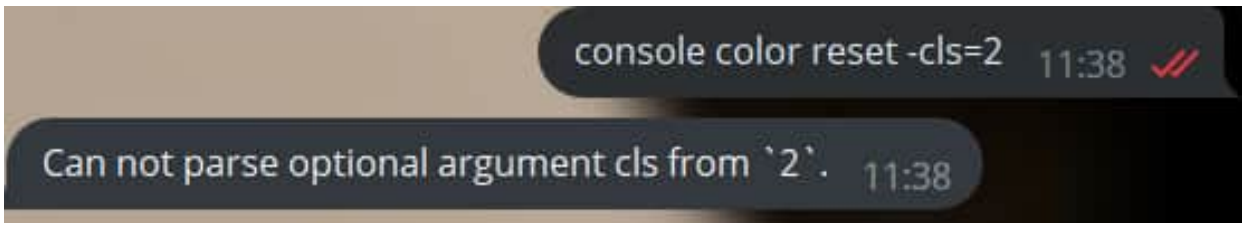

Рис 6.10 – Не можливе конвертування опційного параметру.

6.2.6 Стан перевірки правил

На цьому етапі маршрутизації кожен аргумент перевіряється на відповідність згідно обмеження (Constaint) та правила (Validation).

Коли якийсь аргумент не відповідатиме одному із правил, система перейде у термінальний стан, який відобразить, аргумент та причину, чому він не відповідає правилу.

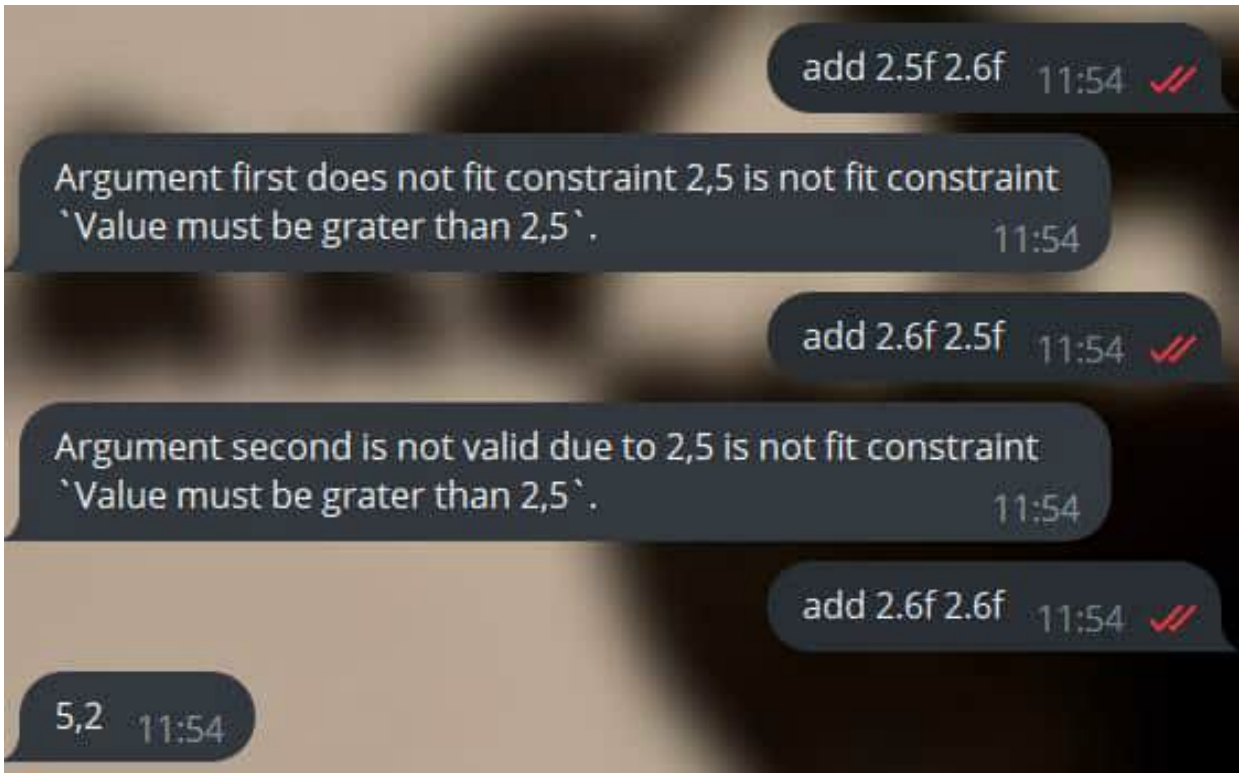

Рис. 6.11 – Помилки при перевірці аргументів.

Коли усі аргументи пройшли перевірку, система переходить у заключний стан, який міститиме перевантаження.

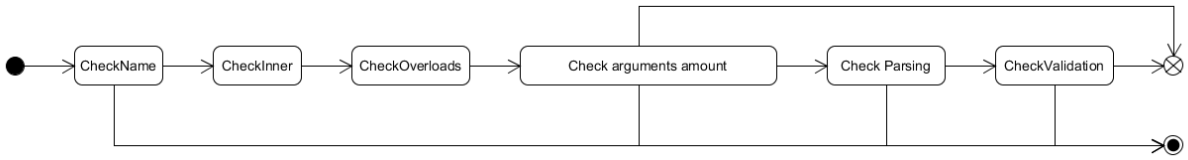

Рис. 6.12 – Діаграма станів маршутизації.

РОЗДІЛ 7. РОЗШИРЕННЯ ФУНКЦІОНАЛЬНОСТІ

У багатьох випадках розробник використовує свої власні типи як параметри або застосовує інший тип проєкту, крім консольного. У такому випадку потрібно надати можливість розширювати систему згідно потреб. Таким чином, система маршрутизації дозволяє користувачу по різному реагувати на підсистему правил чи перетворення параметрів з рядка запиту користувача. У цьому розділі розглядатимуться підсистеми, які можна розширити.

7.1 Система перетворення параметрів.

При застосуванні атрибутів для перетворення методів у перевантаження чи використання методу розробника UseDefaultConverter використовуються класи, які наслідують ArgumentConverterProvider.

При написанні бібліотеки було розроблено деякі конвертери за замовчуванням для наступних типів:

а) Bool - для перетворення рядка у значення true чи false;

б) Int – перетворення рядка у цілочислове значення;

в) Float – перетворення числа у форматі;

г) String – перетворення рядка у рядок;

д) ConsoleColor – перетворення рядка у значення кольору консолі – цей тип розроблений для команди console color.

На основі типу ConsoleColor розглянуто спосіб розширення.

```
internal class ConsoleColorArgumentConverterProvider : ArgumentConverterProvider
{

    public override object Converter
        => new Converter<string, ConsoleColor>(color => (ConsoleColor)Enum.Parse(TargetingType, color, true));


    public override Type TargetingType => typeof(ConsoleColor);
}
```

Рис. 7.1 – Приклад розширення конвертеру типу ConsoleColor.

7.2 Система інформування

Для розширення системи інформування розробник може наслідувати клас Informing та перевантажити метод OnError, який приймає повідомлення про помилку.

На першому кроці наслідується клас Informing. Визначається метод OnError та необхідні додаткові властивості (у даному випадку телеграм клієнт та повідомлення).

```
internal class TelegramInforming : Informing
{
    private readonly ITelegramBotClient _client;


    public Message? Message {get; set;}


    public TelegramInforming(ITelegramBotClient client)
    {
        _client = client;
    }


    public override async void OnError(string errorMessage)
    {
        if (Message is null)
        {
            return;
        }

        await _client.SendTextMessageAsync(Message.Chat.Id, errorMessage);
    }
}
```

Рис. 7.2 – реалізація абстрактного класу інформування.

Тепер при створені правила ми можемо застосувати цей клас для інформування.

```
var rule = Rule<int>.BeginInit()
                    .FromMethod(i => i.IsEven())
                    .OnLevel(new TelegramInforming(botClient) { Message = message })
                    .WithConstraint("Number must be even")
                    .WithPropertySelector(i => i.ToString())
                    .EndInit();

await Task.FromResult(rule.ValidateWithError(int.TryParse(message.Text, out var result) ? result : 0));
```

Рис. 7.3 – Приклад використання нового рівня інформування.

Для перевірки результату використано телеграм бот, який використовувався для перевірки маршрутизації. При отримані повідомлення, яке містить число парне ніякої відповіді не поступає, проте при введені непарного числа користувач отримує повідомлення про те, що введене число не є парним.

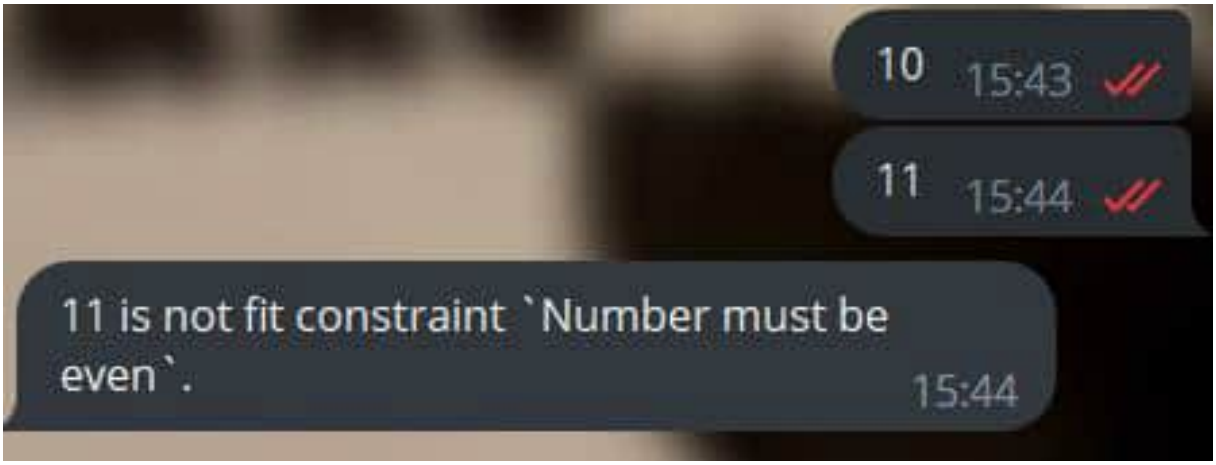

Рис. 7.4 – Відповідь телеграм бота на основі перевірки.

ВИСНОВКИ

Досліджене оновлення системи маршутизації користувацьких запитів, що базується на підході розширеного FluentApi під час використання виконує функції контролю та підказування щодо використання розробленої бібліотеки. У такий спосіб розробник зможе інтуїтивно використовувати функціонал та додавати його до своїх проєктів, які базуються на консольних програмах чи ботах у чат месенджерах.

За означенням архітектурний паттерн Fluent interface використовує правило найменування методів таким способом, щоб ланцюжок викликів можна було перетворити на речення, зрозуміле людині. Оскільки поточне дослідження базувалось на цьому принципі, налаштування маршрутизації є зрозумілим для розробників і при зміні різних параметрів системи, програмний код буде читабельним, контрольованим та матиме можливість інтуїтивного оновлення.

На основі розробленої системи метапрограмування розробнику доступний функціонал, який дозволить різні об’єкти перетворювати на джерело даних, що спрощує процес отримання необхідної інформації. Цей підхід також піддається принципам розробки FluentApi, тому його використання також є зрозумілим.

Також розробник зможе використовувати розроблену систему створення правил для перевірки параметрів, що приходять у методи. На базі реалізованої методики інформування при написанні програмного коду користувач бібліотеки зможе налагоджувати код та отримувати повідомлення про невідповідність значення за правилом.

Результатом роботи є опис підходу до створення бібліотеки, яка надає функціонал налаштування маршрутизації користувацьких запитів, декомпозиції об’єктів на потрібні властивості та систему перевірки параметрів.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Мручко І. Маршрутизація користувацьких запитів. 35с. Режим доступу: Курсова робота (Мручко Ігор ПМІ-32).pdf
2. Мручко. І. Застосування FluentAPI при створені бібліотек // Міжнародна студентська наукова конференція з питань прикладної математики та комп'ютерних наук (МСНКПМК-2023), 4-5 травня 2023р. – Львів:2023. – С. 55-58. Режим доступу: https://ami.lnu.edu.ua/wp-content/uploads/2023/05/ISSCAMCS-2023.pdf
3. Fluent Interface [Wikipedia]. – Режим доступу: https://uk.wikipedia.org/wiki/Fluent_interface
4. Fluent Interface [Java design patterns] – Режим доступу: https://java-designpatterns.com/patterns/fluentinterface/
5. LINQ in C# [Learn Microsoft] – Режим доступу: https://learn.microsoft.com/enus/dotnet/csharp/programming-guide/concepts/linq/features-that-supportlinq#extension-methods
6. Entity Framework – Fluent Api [EFTutorial] – Режим доступу: https://www.entityframeworktutorial.net/efcore/configure-one-to-one-relationshipusing-fluent-api-in-ef-core.aspx
7. C++ Builder pattern with Fluent Api [RipTutorial] – Режим доступу: https://riptutorial.com/cplusplus/example/30166/builder-pattern-with-fluent-api
8. Refactoring guru. Pattern Builder [Refactoring Guru] – режим доступу: https://refactoring.guru/uk/design-patterns/builder
9. Refactoring guru. Parallel Inheritance Hierarchies [Refactoring Guru] – режим доступу: https://refactoring.guru/uk/smells/parallel-inheritance-hierarchies
10. UML Class Diagram Tutorial [Visual Paradigm] – режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

11. All you need to know about state diagrams [Visual Paradigm] – режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/about-state-diagrams/
12. Делегат Func [Learn Microsoft] – режим доступу: https://learn.microsoft.com/uk-ua/dotnet/api/system.func-2?view=net-7.0
13. Validation [ComputerScience] – режим доступу: https://www.computerscience.gcse.guru/theory/validation
14. Еквіваленція [Wikipedia] – режим доступу: https://uk.wikipedia.org/wiki/%D0%95%D0%BA%D0%B2%D1%96%D0%B2%D0%B0%D0%BB%D0%B5%D0%BD%D1%86%D1%96%D1%8F
15. Логічна імплікація [Wikipedia] – режим доступу: https://uk.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D0%B0_%D1%96%D0%BC%D0%BF%D0%BB%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F
16. Metaprogramming [lmu] – режим доступу: https://cs.lmu.edu/~ray/notes/metaprogramming/
17. Видимість у збірці [Learn Microsoft] – режим доступу: https://learn.microsoft.com/uk-ua/dotnet/csharp/language-reference/keywords/internal
18. Internal visible to Attribute [Learn Microsoft] – режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.internalsvisibletoattribute?view=net-7.0
19. Define custom attribute [Lear Microsoft] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/attribute-tutorial
20. NUnit [NUnit] – режим доступу: https://nunit.org/
21. Часткові методи та класу [Learn Microsoft] – Режим доступу: https://learn.microsoft.com/uk-ua/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods

22. Define and read custom attributes [Learn Microsoft] – режим доступу: https://learn.microsoft.com/uk-ua/dotnet/csharp/advanced-topics/reflection-and-attributes/attribute-tutorial
23. Стан [Refactoring guru] – режим доступу: https://refactoring.guru/uk/design-patterns/state

ДОДАТОК А. ІЄРАРХІЯ КЛАСІВ СТАНІВ МАРШРУТИЗАЦІЇ.