Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**ЗАСОБИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗА РАХУНОК ВПРОВАДЖЕННЯ ЗАЛЕЖНОСТЕЙ**»

Виконав: студент 4-го року навчання,

Освітньої програми «Інженерія програмного забезпечення» 121

Зимовець Руслан Олегович

Керівник Бублик В. В._____

доцент, кандидат фіз.-мат. наук

Рецензент _____

(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 2024 р.

Київ – 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультету інформатики

Кафедра мультимедійних систем

ЗАТВЕРДЖУЮ
зав. кафедри інформатики
доцент, к. ф.-м. н. С. С. Гороховський

_____ *(підпис)*

«____» _____ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту Зимовцю Руслану Олеговичу факультету інформатики 4-го курсу

Тема: Засоби підвищення ефективності програмного забезпечення за рахунок Впровадження Залежностей

Зміст ТЧ до кваліфікаційної роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Зміст
5. Вступ
6. Розділ 1: Засоби Впровадження Залежностей
7. Розділ 2: Організація Динамічного Контейнера Впровадження Залежностей
8. Розділ 3: Реалізація Динамічного Контейнера Впровадження Залежностей
9. Розділ 4: Створення застосунку Задачника (tmcli), базованого на Контейнері Впровадження Залежностей
10. Висновки
11. Список використаних джерел

Дата видачі 08.10.2023 р. Керівник Бублик В. В.

Завдання отримав Зимовець Руслан Олегович

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тема: Засоби підвищення ефективності програмного забезпечення за рахунок Впровадження Залежностей

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	08.10.2023	
2.	Дослідження теоретичної інформації з теми роботи	18.10.2023	
3.	Початок розроблення практичної частини роботи	20.12.2023	
5.	Аналіз та довершення практичної частини роботи	12.03.2024	
7.	Початок роботи над текстовою частиною роботи	26.03.2024	
9.	Аналіз та довершення текстової частини роботи	26.04.2024	
10.	Створення презентації	01.05.2024	
11.	Захист курсової роботи	27.05.2024	

Студент Зимовець Р. О.

Керівник Бублик В. В.

АНОТАЦІЯ

У роботі розглянуто поняття Впровадження Залежностей та засоби, якими його можна досягти. Досліджено Контейнери Впровадження Залежностей як засіб Впровадження Залежностей в програмних системах, що оперують з даними складної структури. Було розроблено бібліотеку мовою C++20, що містить Динамічний Контейнер Впровадження Залежностей та засоби для гнучкого конфігурування такого контейнера. Як приклад використання створеної бібліотеки був реалізований застосунок Задачник, який демонструє переваги та недоліки розглянутих контейнерів.

Ключові слова: впровадження залежностей, контейнер впровадження залежностей.

Зміст

ВСТУП

Упровадження Залежностей – доволі простий програмний взірець проектування, який, одначе, часто вимагає складних засобів його реалізації – Контейнерів Впровадження Залежностей. Такий контейнер зазвичай є громіздкою бібліотекою, яка значною мірою спрощує розроблення масштабних проєктів, надаючи розробникам гнучкі конфігураційні можливості та способи ефективного розподілу роботи між командою розробників.

Актуальність теми

З плином часу складність технологій без упину росте, що, як наслідок, значно збільшує розміри програмних проєктів, на яких вони базуються. Питання організації розроблення таких проєктів – завдання неабиякої складності, яким займаються не лише менеджери та технічні керівники, а й розробники, що відповідальні за проектування архітектури. У даній роботі досліджено та розроблено засіб, який такі проектувальники використовують для реалізації взірця Впровадження Залежностей, – Контейнер Впровадження Залежностей. На таких засобах базуються безліч сучасних проєктів, що, неодмінно, доводить їх актуальність для сьогодення.

Мета і завдання дослідження

Метою даної роботи було визначено реалізацію Контейнера Впровадження Залежностей з використанням стандарту C++20, створення гнучких способів його налаштування, а також засобів для ефективного розподілу роботи над проєктами, базованих на такому контейнері, між командою розробників.

Наукова новизна одержаних результатів

Результати даної роботи збагачують програмний світ мови C++ бібліотекою fast_di, що реалізує Контейнер Впровадження Залежностей, яка не має собі подібних серед інших бібліотек даної області. Розроблена бібліотека пропонує новий спосіб конфігурації Контейнера Впровадження Залежностей, який має переваги над іншими підходами до налаштування таких контейнерів.

Практичне значення одержаних результатів:

Створену в межах даної роботи бібліотеку можна використовувати для побудови та масштабування проєктів. Приклад такого застосування надано у вигляді застосунку Задачника, базованого на розробленому Контейнері Впровадження Залежностей.

РОЗДІЛ 1: Засоби Впровадження Залежностей

1.1. Сутність підтримуваності коду

Програмне забезпечення має різні вимоги залежно від його масштабності. Маленькі проєкти, які живуть протягом щонайбільше одного релізу, майже ніколи не потребують серйозного підходу до проектування задля спрощення їх подальшого розширення. Для них не виникає питання гнучкості архітектури з можливостями додавати новий функціонал та уникнення конфліктів змін, якщо над цим працюватимуть одночасно декілька розробників. Щоправда тестування таких проєктів може виявитись складним без належного підходу до нього.

Визначення: ***Паралельна розробка*** *– робота одночасно багатьох розробників над однією кодовою базою.*

Великі проєкти, з яких випливають сотні релізів, у свою чергу, вимагають особливої методики розробки через низку причин:

- Великі проєкти складно розширюються
- Паралельна розробка спричиняє конфлікти внесених до проєкту змін
- Юніт-тестування окремих модулів вищого рівня дуже ускладнюється через неможливість їх ізоляції від модулів нижчого рівня.

На цих трьох китах збудуймо одну важливу характеристику коду, яка й стане одним з головних об'єктів дослідження даної роботи.

Визначення: ***Підтримуваність коду*** *– міра ефективності розширення та зневадження кодової бази, що прямо пропорційна наступним мірам:*

- *Пристосованість коду до додавання нового функціоналу*
- *Пристосованість коду до паралельної розробки*
- *Тестоздатність коду.*

Тому, підтримуваність коду – те, чого дуже потребують масштабні проєкти: пристосованість до додавання нового функціоналу дозволить легко пройти через безліч релізів, паралельна розробка дає можливість ефективно задіяти усю інженерну команду, а тестоздатність дозволить забезпечити якість продукту під час розробки чи релізу.

Однак як досягти підтримуваності коду?

1.2. Визначення та мета Впровадження Залежностей

Відомо, що слабка зв'язність коду значною мірою сприяє його підтримуваності. Принцип «Program to an interface, not an implementation», сформульований «Бандою чотирьох» (Gang of Four) у книзі [1], по суті служить однією з форм слабкої зв'язності. Саме на цьому принципі побудовано Впровадження Залежностей.

Визначення з [2]: ***Упровадження Залежностей*** *– це множина методів та взірців дизайну програмного забезпечення, яка дозволяє розробляти слабко зв'язаний код за наступним принципом: інфраструктура проєкту має надавати класам екземпляри абстракцій, від яких вони залежать.*

Визначення: ***Пізнє Зв'язування*** *– здатність проєкту підмінювати одну реалізацію інтерфейсу впровадженої залежності на іншу без необхідності перекомпільовувати якусь частину наявного коду*

Згідно з [2], мета Упровадження Залежностей полягає у досягненні наступних властивостей коду:

- Розширюваність
- Можливість для паралельної розробки
- Тестоздатність
- Пізнє зв'язування.

Або, лаконічніше:

- Підтримуваність
- Пізнє зв'язування

1.3. Контейнери Впровадження Залежностей

Визначення з [2]: **Центр Композиції (Composition Root)** – централізація створення об'єктів усіх класів програми в одному місці (файлі) під час Впровадження Залежностей.

Загалом, існує два шляхи до досягнення Впровадження Залежностей – [2] Чисте Впровадження Залежностей та Контейнер Впровадження Залежностей. Під Чистим Впровадженням Залежностей зазвичай розуміють реалізацію Впровадження Залежностей «вручну», тобто шляхом явного створення об'єктів в коді та явної їх композиції, агрегації тощо. Хоча це позитивно впливає на підтримуваність коду загалом, проте, як результат, ми отримуємо слабке за

підтримуваністю місце – Центр Композиції програми. Аби додати якусь нову залежність, чи замінити вже наявну на якусь іншу, доводиться модифікувати Центр Композиції, що дуже ускладнює паралельну розробку. Модифікація одного файлу багатьма розробниками завжди веде до конфліктів злиття у системах контролю версій. Зокрема цю важливу проблему і вирішують деякі Контейнери Впровадження Залежностей. Хоча це не єдине їх застосування.

Визначення: **Впроваджена Залежність класу** – це тип об'єкту, який можна впровадити в екземпляр цього класу шляхом Впровадження залежностей.

Визначення: **Контейнер Впровадження Залежностей** – це програмна сутність, відповідальність якої полягає в автоматизації Впровадження Залежностей шляхом інкапсуляції коректного створення об'єктів, враховуючи Впроваджені Залежності їх класів, та надання гнучкого інтерфейсу для керування таким створенням або наданням доступу до створених об'єктів.

Визначення: **Динамічний Контейнер Впровадження Залежностей** – Контейнер Впровадження Залежностей, який здійснює створення об'єктів та надання доступу до них завжди на етапі виконання.

Найбільш важливим місцем будь-якого Контейнера Впровадження Залежностей є їх конфігураційні можливості. Щоб контейнер міг створити екземпляр якогось класу та надати до нього доступ, необхідно створити певні конфігурації, щоб він мав достатньо статичної інформації про тип цього екземпляру, впроваджені залежності його класу, про те, як довго протягом виконання програми має існувати цей екземпляр, тощо. Цілком сконфігурований контейнер має лише одну корисну функцію – надання доступу до створеного екземпляру.

Для прикладу, розглянемо наступне використання примітивного Контейнеру Впровадження Залежностей мовою C++.

```
 1 struct IAnimal {
 2     virtual ~IAnimal() = default;
 3     virtual void make_sound() const = 0;
 4 };
 5
 6 struct Voice {
 7     void say(const std::string_view to_say) const {
 8         std::cout << to_say << '\n';
 9     }
10 };
11
12 struct Dog : IAnimal {
13     explicit Dog(Voice& voice) : voice_ { voice }
14     {}
15
```

```
16      void make_sound() const override {
17          voice_.say("Woof!");
18      }
19
20  private:
21      Voice& voice_;
22  };
23
24  int main() {
25      DIContainer container{};
26      container.register_type<Dog, IAnimal>();
27      container.register_type<Voice>();
28
29      IAnimal& resolved_animal = container.resolve<IAnimal>();
30      resolved_animal.make_sound(); // Woof!
31  }
```

Лістинг 1: Використання Контейнера Впровадження Залежностей

Отже, спочатку визначаються класи IAnimal та Dog, які аніяк не залежать від Контейнера Впровадження Залежностей. Далі на рядку №25 створюється «пустий» екземпляр такого контейнера і на рядках №26-27 реєструється типи: Dog з прив'язкою до інтерфейсу IAnimal, Voice без прив'язки до інтерфейсу. На рядках №29-30 отримується доступ до створеного екземпляру класу Dog прив'язаного до інтерфейсу IAnimal, на якому потім викликається метод make_sound.

Важливо звернути увагу, що контейнер не викликає конструктор «за замовчуванням» і навіть не вимагає його існування, щоб створити екземпляр класу Dog. Він спочатку розглядає список впроваджених залежностей класу Dog – ним є список з одним класом Voice – створює екземпляри цих впроваджених залежностей та автоматично впроваджує їх в екземпляр класу Dog, який він створює шляхом виклику його конструктора. Зручно, чи не так?

На цьому етапі це може здаватися просто бібліотекою, яка просто створює об'єкти, автоматично впроваджуючи залежності. Проте, цим не обмежується можлива функціональність Контейнерів Впровадження Залежностей.

Розглянемо два підходи до конфігурування (налаштування) контейнерів.

Визначення: **Налаштування на основі конфігураційних файлів** – спосіб конфігурування, у якому вся інформація про налаштування Контейнера Впровадження Залежностей зберігається в одному або декількох файлів.

Визначення: **Конфігурація на основі коду** – спосіб конфігурування, у якому інформація про налаштування Контейнера Впровадження Залежностей цілком зберігається у вигляді коду програми, яка працює на основі цього контейнера.

Згідно з [2], перевагою конфігураційних файлів є можливість досягнути пізнього зв'язування, що неможливо конфігурацією на основі коду. У свою чергу, конфігурація на основі коду краща тим, що вона забезпечує більше перевірок коректності на етапі компіляції та зазвичай надає більше можливостей для контролю над конфігураціями. До недоліків конфігураційних файлів відносять відсутність вичерпних перевірок коректності на етапі компіляції. До недоліків конфігурації на основі коду відносять відсутність можливості досягти пізнього зв'язування, адже всі конфігурації статично вбудовано в сам код.

Важливим недоліком обох підходів є ускладнена паралельна розробка, подібно до недоліку Чистого Впровадження Залежностей. Зазвичай, Контейнери впровадження залежностей, за конфігурації цими підходами, не передбачають можливості зробити підтримуваним місце в коді, де будуть здійснюватись усі налаштування (аналогічне до Центру Композиції). Для конфігурації на основі коду часто доводиться розміщувати усі конфігурації в одному файлі, адже для поділу одного Контейнера Впровадження Залежностей між декількома файлами вимагає розробки додаткових, часто невиправдано складних, інструментів. У книзі [2] Стівен ван Деурсен також стверджує, що конфігураційні файли теж є зовсім нерозширюваними, спираючись на подібні міркування.

Особливу увагу в даній роботі звернуто на вирішення проблеми ускладненої паралельної розробки деяких реалізацій Контейнера Впровадження Залежностей.

1.4. Задача розробки Динамічного Контейнера Впровадження Залежностей

В межах результатів даної роботи лежить розробка Динамічного Контейнера Впровадження Залежностей мовою C++, який задовольняє наступні умови:

- Можливість автоматизовано реалізувати Впровадження Залежностей
- Можливість досягнення пізнього зв'язування не є обов'язковою
- Можливість здійснювати конфігурацію на основі коду
- Присутній набір інструментів для покращення паралельної розробку
- Можливість впровадження різних реалізацій одного інтерфейсу залежно від конфігурацій.

РОЗДІЛ 2: Організація Динамічного Контейнера Впровадження Залежностей

2.1. Типи життєвих циклів об'єктів, які надає розроблений контейнер

Автоматизація Впровадження Залежностей не означає лише автоматичне створення об'єктів та надання доступу до них; більшість Контейнерів Впровадження Залежностей, окрім цих операцій, здатні також здійснювати виявлення циклів залежностей, керування життєвим циклом об'єктів, перехоплення (декорування) тощо.

Розроблений Контейнер Впровадження Залежностей, зокрема, надає певні різновиди життєвих циклів об'єктів, якими можна оперувати. Життєві цикли визначають, як довго протягом виконання програми буде існувати об'єкт та коли його буде знищено. З використанням розробленого Динамічного Контейнера Впровадження Залежностей створеним об'єктам можна надавати Єдиносутній Життєвий Цикл або Тимчасовий Життєвий Цикл.

Визначення: **Єдиносутній Життєвий Цикл (Singleton Lifecycle)** – тип життєвого циклу об'єкта, який існує в єдиному екземплярі протягом всього виконання програми або починаючи з моменту, коли контейнер отримав запит на створення цього об'єкту, та до кінця виконання програми.

Визначення: **Тимчасовий Життєвий Цикл (Transient Lifecycle)** – тип життєвого циклу об'єкта, який наново створюється кожного разу, коли контейнер отримав запит на створення цього об'єкта, та існує доти, запитувач використовує його.

2.2. Архітектура розробленого контейнера

Розроблений Динамічний Контейнер Впровадження Залежностей є частиною розробленої бібліотеки з назвою fast_di та спроектовано згідно з взірцем проектування «Будівельник» (Builder, що описано в [1]). А саме, сформовано наступний набір типів:

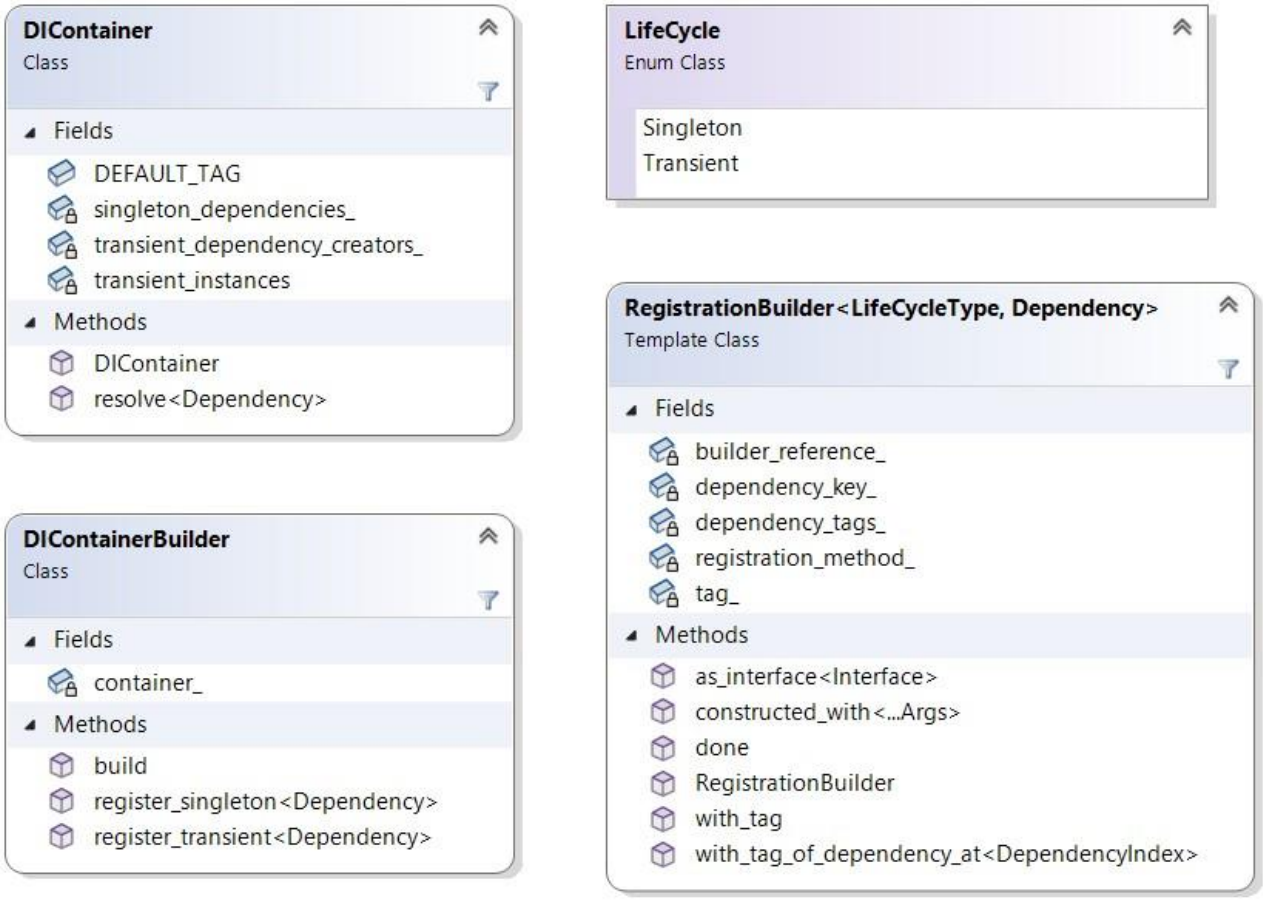

Рис. 1: Архітектура розробленого Динамічного Контейнера Впровадження Залежностей

Опис:

1) DIContainer – Динамічний Контейнер Впровадження Залежностей
   a) Поля:
      i) DEFAULT_TAG – статична константа, рядковий ідентифікатор типу за замовчуванням.
      ii) singleton_dependencies_ - контейнер з даними, необхідними для керування об'єктами з єдиносутнім життєвим циклом;
      iii) transient_dependency_creators_ - контейнер з даними, необхідними для створення об'єктів з тимчасовим життєвим циклом
      iv) transient_instances – контейнер, який містить у собі створені об'єкти з тимчасовим життєвим циклом
   b) Методи:
      i) DIContainer – конструктор контейнера .
      ii) resolve<Dependency> – метод, який надає доступ до об'єкту типу Dependency, створеного контейнером.

(1) Приймає: tag – рядковий ідентифікатор типу Dependency; Значення за замовчуванням: DIContainer::DEFAULT_TAG.
(2) Повертає: відсилка на об'єкт типу Dependency.
(3) Помилки: std::runtime_error – якщо тип Dependency не зареєстровано або не можна сконструювати.

2) DIContainerBuilder – «Будівельник», що створює об'єкти типу DIContainer.
   a) Поля:
      i) container_ – екземпляр типу DIContainer, який наповнюється даними про реєстрації типів, інтерфейсів, ідентифікаторів типів тощо.
   b) Методи:
      i) build – створює екземпляр DIContainer з поточним наповненням даними поля container_.
         (1) Повертає: об'єкт типу DIContainer.
      ii) register_singleton<Dependency> - реєструє тип Dependency та присвоює його об'єктам Єдиносутній Життєвий Цикл.
         (1) Повертає: об'єкт типу RegistrationBuilder<Singleton, Dependency>.
      iii) register_transient<Dependency> - реєструє тип Dependency та присвоює його об'єктам Тимчасовий Життєвий Цикл.
         (1) Повертає: об'єкт типу RegistrationBuilder<Transient, Dependency>.
3) LifeCycle – enum-тип, що слугує ідентифікатором для життєвого циклу типів, які реєструються в контейнері.
   a) Варіанти:
      i) Singleton – ідентифікатор єдиносутнього життєвого циклу.
      ii) Transient – ідентифікатор тимчасового життєвого циклу.
4) RegistrationBuilder<LifeCycleType, Dependency> - тип, що здійснює реєстрацію типу Dependency в контейнері з присвоєнням йому життєвого циклу типу LifeCycleType (Singleton/Transient) та інших конфігурацій.
   a) Поля:
      i) builder_reference_ - відсилка на об'єкт DIContainerBuilder.
      ii) dependency_key_ - поле, що містить std::type_info типу Dependency.
      iii) dependency_tags_ - список рядкових ідентифікаторів типів, які необхідно впровадити в Dependency.
      iv) registration_method_ - функтор, що здійснює реєстрацію типу та усіх його конфігурацій.
      v) tag_ - рядковий ідентифікатор типу Dependency.
   b) Методи:
      i) as_interface<Interface> – здійснює прив'язку типу Dependency до інтерфейсу Interface в межах поточної реєстрації.
         (1) Повертає: відсилку на об'єкт, до якого належить цей метод.
      ii) done – завершує поточну реєстрацію.

(1) Повертає: відсилку на об'єкт DIContainerBuilder.
(2) Помилки: std::runtime_error – якщо здійснено невалідну конфігурацію.

iii) RegistrationBuilder – конструктор типу RegistrationBuilder.
(1) Приймає: builder_reference – відсилка на об'єкт типу DIContainerBuilder.

iv) with_tag_of_dependency_at<Index> – вказує рядковий ідентифікатор для залежності Dependency, яка стоїть на індексі Index у списку параметрів функції-створювача (create) типу Dependency. DIContainer потім впровадить тип/реалізацію інтерфейсу Interface з необхідним рядковим ідентифікатором в об'єкт Dependency.
(1) Приймає: tag – рядковий ідентифікатор типу Interface
(2) Повертає: відсилку на об'єкт, до якого належить цей метод.

v) with_tag – вказує рядковий ідентифікатор типу Dependency.
(1) Приймає: tag – рядковий ідентифікатор типу Dependency.
(2) Повертає: відсилку на об'єкт, до якого належить цей метод.

2.3. Структури даних, що використовує розроблений контейнер

Незважаючи на широкий спектр можливостей мови C++, до стандарту C++20 вона все ще не отримала гарних інструментів рефлексії етапу виконання, таких як в мові C#, наприклад. Це зумовлено складністю архітектури мови. У C++ навіть ітерація по списку параметрів конструктора якогось класу є, взагалі кажучи, неможливою, тому розробка Контейнера Впровадження Залежностей авжеж ніяк не може бути тривіальною задачею.

Однак C++ містить вбудовану можливість отримати ідентифікатор довільного типу, який можна зберегти в пам'яті та користатися ним в межах виконання програми. Цього можна досягти, використовуючи тип std::type_info (який описано в [4]). Отримання ідентифікатора типу (екземпляру std::type_info) здійснюється за допомогою вбудованого оператора typeid. Наприклад, отримання ідентифікатора типу int:

```
const std::type_info& int_type_info = typeid(int);
```

Лістинг 2: Приклад отримання ідентифікатора типу int

Для реалізації Динамічного Контейнера Впровадження Залежностей тип std::type_info можна використати для отримання певного хешу типу (наприклад, SomeType), який буде зберігатись у контейнері, а, за виникнення запиту на отримання доступу до створеного екземпляру SomeType, будемо шукати

необхідні для цього дані в асоціативному контейнері за ключем typeid(SomeType).

Отже, Динамічний Контейнер Впровадження Залежностей міститиме наступні асоціативні контейнери:

```
using SingletonDependenciesMap =

    std::unordered_map<std::type_index, SomeData>;

using TransientDependenciesMap =

    std::unordered_map<std::type_index, SomeData>;
```

Лістинг 3: варіант асоціативних контейнерів для побудови Динмічного Контейнера Впровадження Залежностей

Згідно з [4], std::unordered_map обрано, бо це найефективніший з наявних у стандартній бібліотеці C++ асоціативних контейнерів для збереження великої кількості даних та для швидкого доступу до них за ключем.

Згідно з [4], std::type_index – клас обгортка над std::type_info, яку можна використовувати як тип ключа в асоціативних контейнерах STL.

SomeData – тимочасове позначення для типу, який міститиме в собі усі необхідні дані для створення об'єктів та надання до них доступу Контейнером Впровадження Залежностей.

Як реалізувати створення об'єктів та отримання доступу до них? Найбільш вдалим для Динамічного Контейнера Впровадження Залежностей стане рішення відкладеного (лінивого) створення об'єктів: кожен об'єкт буде створено лише за необхідності – запиту на доступ до нього. Для реалізації цього механізму можна використати функтори-створювачі, яких буде використано (викликано метод “operator()”), як тільки користувач контейнера здійснить запит на доступ до відповідного типу. Такі створювачі можна зберігати десь в межах SomeData.

Тип створювачів можна сформувати з допомогою STL:

```
using Creator = std::function<Object&()>;
```

Лістинг 4: Створювач об'єктів типу Object

Однак що має повертати створювач? Відсилку на об'єкт якого типу? В C++ не існує типу Object, від якого наслідуються всі класи.
Можливо, узагальнене програмування стане у нагоді?

```
template<typename Object>
```

```
using GenericCreator = std::function<Object&()>;
```

Лістинг 5: Узагальнений створювач об'єктів типу Object

На жаль, GenericCreator теж не допоможе. Ми не зможемо зберігати GenericCreator в межах типу-значення єдиного асоціативного контейнера, адже для різних типів Object1 та Object2 типи їх створювачів будуть різними: GenericCreator<Object1> та GenericCreator<Object2> - два типи, що не є залежними один від одного. Через це, нам довелося б мати по окремому асоціативному контейнеру для кожного окремого типу для збереження даних, що містять створювача об'єктів цього типу. Такий підхід, безсумнівно, є неефективним.

Гарне рішення надійшло з мови програмування C. Нехай створювачі повертатимуть тип void*:

```
using SingletonCreator = std::function<void*()>;
```

Лістинг 6: Створювач об'єктів довільного типу з єдиносутнім життєвим циклом

Справді, таким чином такий створювач об'єктів з єдиносутнім життєвим циклом буде повертати указник на створений всередині нього статичний об'єкт потрібного типу. До того ж, користувачу не доведеться працювати з сирим указником void*, оскільки контейнер перетворить void* на відсилку Dependency&, де Dependency – тип, на отримання доступу до якого користувач здійснив запит.

Чи спрацює такий підхід для створювачі об'єктів з Тимчасовим Життєвим Циклом? Не зовсім. Проблема у знищенні об'єктів з Тимчасовим Життєвим Циклом. void* не може повноцінно володіти об'єктом, на який посилається, адже він не буде викликати його деструктор та звільнювати пам'ять, виділену під нього, автоматично. void* здатен просто посилатись на об'єкт. Тому ми маємо або знищувати об'єкт вручну, або використовувати розумні указники з STL. Під час розробки Динамічного Контейнера Впровадження Залежностей у даній роботі було обрано підхід з розумними указниками, а саме з std::shared_ptr<void>. Такий указник може повноцінно володіти об'єктом, знищуючи його автоматично, коли об'єкт більше ніхто не використовує.

Тож, тип створювача об'єктів з Тимчасовим Життєвим Циклом буде неступним:

```
using TransientCreator =
  std::function<std::shared_ptr<void>()>;
```

Лістинг 7: Створювач об'єктів довільного типу з тимчасовим життєвим циклом

Проте і навіть цей тип не є остаточним. З ними теж є деяка проблема. Створювачі потребують відсилку на об'єкт типу DIContainer, до якого вони належать. Наприклад, уявимо, що ми ініціалізуємо якогось із створювачів з допомогою лямбда-виразу:

```
[this]() {
      auto constructor_args = this->resolve_constructor_arguments<Dependency>();
      static Dependency instance = construct_with(constructor_args);
      return &instance;
};
```

Лістинг 8: Невдалий варіант ініціалізатора створювачів об'єктів з єдиносутнім життєвим циклом

Проблема в тому, що, як тільки об'єкт типу DIContainer, на який вказує this, буде переміщено (Семантика Переміщень, що описано в [4]) або, ще гірше, знищено, то this буде вказувати на невалідний об'єкт або на пам'ять, до якої ми, імовірно, більше не маємо, що зазвичай приводить до помилки етапу виконання Segmentation Fault.

Найбільш вдало буде передавати відсилку на об'єкт типу DIContainer параметром створювача:

```
[](const DIContainer& self) {
      auto constructor_args = self.resolve_constructor_arguments<Dependency>();
      static Dependency instance = construct_with(constructor_args);
      return &instance;
};
```

Лістинг 9: Вдалий варіант ініціалізатора створювачів об'єктів з єдиносутнім життєвим циклом

Таким чином, відсилка self буде завжди вказувати на валідний об'єкт. Тож, визначимо остаточно типи створювачів:

```
using SingletonCreator =
      std::function<void*(const DIContainer&)>;
using TransientCreator =
      std::function<std::shared_ptr<void>(const DIContainer&)>;
```

Лістинг 10: Остаточні типи створювачів

Тепер запишемо типи асоціативних контейнерів, на яких можна збудувати цілком функціональний Динамічний Контейнер Впровадження Залежностей:

```
using SingletonDependenciesMap =
      std::unordered_map<std::type_index, SingletonCreator>;
using TransientDependenciesMap =
      std::unordered_map<std::type_index, TransientCreator>;
```

Лістинг 11: Варіант структур даних, на яких можна збудувати Динамічний Контейнер Впровадження Залежностей

Також, далі у даній роботі буде розглянуто алгоритм впровадження різних реалізацій інтерфейсу Контейнером Впровадження Залежностей. Остаточні типи асоціативних контейнерів є пристованими до цього алгоритму, тому насправді вони є дещо вдосконаленими. Це буде розглянуто далі в даній роботі.

РОЗДІЛ 3: Реалізація Динамічного Контейнера Впровадження Залежностей

3.1. Алгоритми створення об'єктів та надання доступу до них

У чому складність створення об'єктів в межах Контейнера Впровадження Залежностей? Перш за все, щоб створити об'єкт, треба спочатку створити об'єкти усіх типів, від яких залежить його клас. Для цього, також необхідно мати можливість отримувати типи залежностей класів.

Для досягнення зокрема цього функціоналу у даній роботі реалізовано бібліотеку di_utilities, яка є частиною бібліотеки fast_di. Вона вводить тип списку типів, узагальнені операції над ним, а також інструменти для статичного аналізу типів функцій.

Розглянемо тип списку типів з бібліотеки di_utilities:

```
template<typename...>
struct pack {};
```

Лістинг 12: Визначення списку типів як узагальненої структури

Тип pack – список типів, який займає лише 1 байт у пам'яті. Яке йому застосування, якщо в нього навіть немає жодних методів, полів тощо?

Головне призначення типу pack – можливість здійснити автоматичне виведення типів (template argument deduction, згідно з [4]), які входять в тип екземпляра узагальненої структури pack. Це можна робити шляхом передачі цього екземпляра параметром узагальненої функції:

```
template<typename... Types>
void do_type_deduction(pack<Types...>)
{
    // Тут маємо доступ до усіх типів Types
}

int main()
```

```
{
    pack<int, float, double> some_pack{};
    do_type_deduction(some_pack); // викликаємо функцію
}
```

Лістинг 13: Приклад автоматичного виведення типів з використанням списків типів (pack)

У даному прикладі ми створюємо екземпляр структури pack та передаємо його в узагальнену функцію do_type_deduction. Важливо помітити, що не вказано явно, з якими типами слід інстанціювати цю узагальнену функцію. Компілятор автоматично визначає, які типи входять в тип об'єкту some_pack, а саме: int, float та double.

Чим це може бути корисно? З допомогою таких списків типів можна здійснювати статичний аналіз типів функцій. А саме, pack будемо використовувати, для збереження списку типів параметрів функцій, які ми аналізуємо.

Бібліотека di_utilities реалізує наступну інструменти для статичного аналізу функцій:

```
template<typename T>
struct function_traits;

template<typename ReturnValue_, typename... Params_>
struct function_traits<ReturnValue_(Params_...)>
{
    using ReturnValue = ReturnValue_;
    using Params = pack<Params_...>;
};

template<typename Function>
using ReturnValueOf =
    typename function_traits<Function>::ReturnValue;

template<typename Function>
using ParamPackOf =

    typename function_traits<Function>::Params;
```

Лістинг 14: Засоби статичного аналіз функцій з бібліотеки di_utilities

Потужні можливості узагальненого програмування мови C++ дозволяють визначити типи параметрів функції на етапі компіляції. Для цього реалізовано структуру function_traits та її часткову спеціалізацію для типів, що підпадають

під шаблон ReturnValue_(Params_...), а саме для типів функцій, що повертають значення типу ReturnValue_ та приймають параметри типів Params_ (список типів). Варто зазначити, що без pack (чи інших альтернатив) ми б не змогли передати список типів Params_ в якусь іншу частину програми. Тобто наступний код не скомпілювався б:

```
template<typename T>
struct function_traits;

template<typename ReturnValue_, typename... Params_>
struct function_traits<ReturnValue_(Params_...)>
{
    using Params = Params_...;
};
```

Лістинг 15: Невдала спроба статичного аналізу функцій

Тож, тепер для довільної функції можливо отримати список типів параметрів, які вона приймає:

```
void foo(int, double&, float&&) { /* ... */ }
using ParamsOfFoo = ParamPackOf<decltype(foo)>;
static_assert(std::is_same_v<
    ParamsOfFoo,
    pack<int, double&, float&&>
>);
```

Лістинг 16: Спосіб отримання списку параметрів довільної функції foo

З допомогою статичного судження (static_assert declaration, згідно з [4]) перевіряємо правильність отриманого списку типів параметрів, а саме, ParamsOfFoo є рівним pack<int, double&, float&&>.
Однак, чи можемо ми отримати список параметрів конструктора довільного типу? На жаль ні. Засоби мови C++ не дозволяють отримувати тип конструкторів як функцій. Тому жодна спроба здійснити, на кшталт decltype(T::T), не спрацює. Для цього розроблений Динамічний Контейнер Впровадження Залежностей накладає вимоги на типи, які можна зареєструвати. Кожен з таких типів зобов'язаний мати статичну функцію create, яка буде приймати список залежностей типу, а повертати екземпляр цього типу. Тобто це альтернатива для конструктора, тип якої C++ дозволяє отримати з допомогою decltype.

Тож розроблений Динамічний Контейнер Впровадження Залежностей отримує список залежностей типу T наступним чином:

```
using T_Dependencies = ParamPackOf<decltype(T::create)>;
```

Лістинг 17: Отримання списку типів параметрів статичного методу create типу T

Вирішено одну зі складностей створення об'єктів Контейнером впровадження залежностей. Тепер перейдемо до самого алгоритму створення об'єктів контейнером.

Згідно з архітектурою розробленого Динамічного Контейнера Впровадження залежностей, існує дві сутності: DIContainer та DIContainerBuilder. Це реалізація взірця проектування «Будівельник» (Builder), описаного «Бандою Чотирьох» (Gang of Four) у [1].

Алгоритм створення екземпляру Динамічним Контейнером Впровадження Залежностей наступний:

1. Користувач створює екземпляр DIContainerBuilder;
2. Користувач викликає методи register_singleton та register_transient, вказуючи, які типи вони реєструють та конфігурації реєстрації;
3. Користувач викликає метод build на екземплярі DIContainerBuilder, який повертає створений екземпляр DIContainer.

Як працює DIContainerBuilder? DIContainer містить в собі поля, які мають типи розглянуті у розділі 2.3:

```
SingletonDependenciesMap singleton_dependencies_;
TransientDependenciesMap transient_dependency_creators_;
```

Лістинг 18: Поля Динамічного Контейнера Впровадження Залежностей бібліотеки fast_di

Методи register_singleton та register_transient класу DIContainerBuilder просто заповнюють ці асоціативні контейнери.

Виклик register_singleton<Dependency> додає до singleton_dependencies_ пару з ключем typeid(Dependency) та значенням у вигляді наступного лямбда-виразу, який виконує роль створювача об'єктів єдиносутнього життєвого циклу:

```
[](const DIContainer& self) -> Dependency& {
      auto creator_args_pack = ParamPackOf<decltype(Dependency::create)>{};
      auto creator_args = self.resolve_creator_args(creator_args_pack);
      static Dependency instance = std::apply(Dependency::create, creator_args);
      return instance;
};
```

Лістинг 19: Ініціалізатор створювача об'єктів з єдиносутнім життєвим циклом, використаний у бібліотеці fast_di

Такий створювач алокує об'єкт типу Dependency на статичній пам'яті, як тільки створювача було викликано (ліниво). Тип класу, який компілятор генерує для лямбда-виразу даного створювача, відповідає реалізації Синглтону Меєрса.

Виклик register_transient<Dependency> додає до transient_dependencies_ пару з ключем typeid(Dependency) та значенням у вигляді наступного лямбда-виразу, який виконує роль створювача об'єктів тимчасового життєвого циклу:

```
[](const DIContainer& self) {
      auto creator_args_pack = ParamPackOf<decltype(Dependency::create)>{};
      auto args = self.resolve_creator_args(creator_args_pack);
      return std::make_shared<Dependency>(std::apply(Dependency::create, args));
};
```

Лістинг 20: Ініціалізатор створювача об'єктів з тимчасовим життєвим циклом, використаний у бібліотеці fast_di

Такий створювач алокує об'єкт типу Dependency на купі, знову ж, ліниво.

На цьому базовий функціонал DIContainerBuilder закінчується. Коли використано DIContainerBuilder та створено екземпляр DIContainer (шляхом виклику метода build), користувач може отримувати доступ до створених об'єктів, наприклад типу SomeType, викликаючи метод resolve<SomeType>, що належить класу DIContainer. У свою чергу, resolve<SomeType> здійснює пошук за ключем typeid(SomeType) по наявним асоціативним контейнерам: спочатку по singleton_dependencies_, а потім по transient_dependencies_. Якщо пошук успішний, то в значенні, яке відповідає знаденому ключу, знаходиться створювач, який контейнер викликає і повертає відсилку на створений об'єкт користувачу. Якщо ключ не знайдено, контейнер сповіщає користувача про помилку (std::runtime_exception).

Лишилося лише розглянути, як працює метод resolve_creator_args, що належить класу DIContainer. Для його реалізації в межах бібліотеки di_utilities було розроблено функцію map_to_tuple:

```
template
<
    typename... TypeList, template<typename...> typename Pack,
    typename Mapper
>
constexpr auto map_to_tuple(Pack<TypeList...>&&, Mapper&& mapper);
```

Лістинг 21: Сигнатура функції map_to_tuple з бібліотеки di_utilities

Дана функція здійснює відображення списку типів pack<TypeList…> на тип std::tuple<…> шляхом викликання узагальненого функтора Mapper на кожному з типів, що належать до TypeList. Список типів всередині кортежу std::tuple<…> (що повертає функція) визначається на етапі компіляції, на основі типів, які повертає узагальнений функтор mapper, а саме:

Для індексу I з множини валідних індексів списку типів pack<TypeList…>, тип `decltype(std::get<I>(map_to_tuple_impl(pack<TypeList...>{}, mapper)))` у точності співпадає з типом `ReturnValueOf<decltype(mapper.template operator()<TypeAtI>)>`, де TypeAtI – тип на індексі I у списку типів pack<TypeList…>.
Таку функцію зручно використати в межах resolve_creator_args, аби здійснити прохід по всім залежностям типу, доступ до якого користувач запросив у контейнера, та створити усі ці залежності. Маємо наступну реалізацію метода resolve_creator_args:

```
template<typename... ArgTypes>
std::tuple<ArgTypes&...> resolve_creator_args(utilities::pack<ArgTypes...>) const
{
	return map_to_tuple(pack<ArgTypes...>{}, [this]<typename Arg>() -> Arg& {
		return resolve<std::remove_reference_t<Arg>>();
	});
}
```

Лістинг 22: Реалізація метода resolve_creator_args Динамічного Контейнера Впровадження Залежностей

Як бачимо, в даній реалізації до функції map_to_tuple передано список типів pack<ArgType…>, що є списком залежностей, а також узагальнений лямбда вираз (Lambda expressions with an explicit template parameter list, згідно з [4]), який для кожної залежності, що має тип Arg, викликає метод на DIContainer метод resolve<Arg>, отримуючи відсилку на створений об'єкт Arg. Таким чином для кожного типу, на доступ до якого користувач здійснив запит, контейнер рекурсивно створює усі його залежності, аби створити його об'єкт.

3.2. Алгоритм впровадження різних реалізацій інтерфейсу

Варто зазначити, що описаний до цього Динамічний Контейнер Впровадження Залежностей здатен працювати лише з однією реалізацією інтерфейсу. А саме, розгляньмо, як це відбувається. Нехай маємо наступну ієрархію класів:

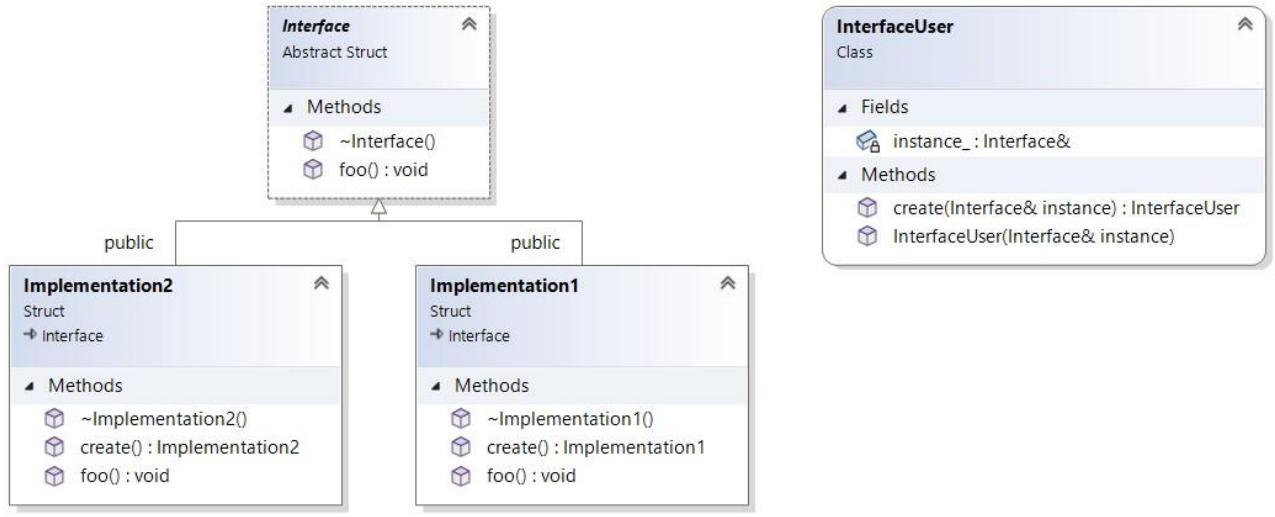

Рис. 2: Приклад ієрархії класів для реєстрації в Динамічному Контейнері Впровадження Залежностей

Interface – інтерфейс, що містить одну чисто віртуальну функцію foo; Implementation1, Implementation2 – реалізації інтерфейсу Interface, що визначають чисто віртуальну функцію foo, а також мають по статичній функції create, яку вимагає DIContainer.

InterfaceUser – клас, що використовує інтерфейс Interface шляхом агрегації відсилкою; він також має функцію create, необхідну для DIContainer.

Далі користувач даної діаграми та Динамічного Контейнера Впровадження залежностей реєструє описаний набір типів, перекладаючи відповідальність за створення їх об'єктів з себе на DIContainer:

```
int main()
{
    DIContainer container = DIContainerBuilder {}
        .register_singleton<InterfaceUser>()
            .done()
        .register_singleton<Implementation1>()
            .as_interface<Interface>
            .done()
        /*.register_singleton<Implementation2>()
            .as_interface<Interface>
            .done()*/
        .build();

    InterfaceUser& user = container.resolve<InterfaceUser>();
    // тут можна використовувати user як повноцінний об'єкт типу InterfaceUser
}
```

Лістинг 23: Реєстрація типів InterfaceUser та Implementation1 в Динамічному Контейнері Впровадження Залежностей

Таким чином користувач зареєстрував тип InterfaceUser, а також Implementation1 з прив'язкою до інтерфейсу Interface. Проте, як зазначалося раніше, такий Динамічний Контейнер Впровадження Залежностей здатен працювати лише з однією реалізацією інтерфейсу. Тобто, неможливо одночасно зареєструвати Implementation1 та Implementation2 з прив'язкою до одного й того ж самого інтерфейсу: так чи інакше контейнер обере якусь одну з реалізацій.

Для отримання можливості працювати одразу з декількома реалізаціями одного інтерфейсу необхідно розширити вже описаний Динамічний Контейнер Впровадження Залежностей.

У межах бібліотеки, розробленої в даній роботі введено систему рядкових ідентифікаторів, які використовуються для однозначного вибору необхідної реалізації інтерфейсу.

Для цього введемо тип рядкового ідентифікатора:

```
using Tag = std::string_view;
```

Лістинг 24: Тип рядкового ідентифікатора (Tag)

А також, модифікуємо структури даних Динамічного Контейнера Впровадження Залежностей, які було введено раніше:

```
using SingletonCreator = std::function<void*(const DIContainer&)>;
using TransientCreator = std::function<std::shared_ptr<void>(const DIContainer&)>;
using TagsMappingSingletonVec = std::vector<std::pair<Tag, SingletonCreator>>;
using TagsMappingTransientVec = std::vector<std::pair<Tag, TransientCreator>>;
using SingletonDependenciesMap =
      std::unordered_map<std::type_index, TagsMappingSingletonVec>;
using TransientDependenciesMap =

      std::unordered_map<std::type_index, TagsMappingTransientVec>;
```

Лістинг 25: Структури даних Контейнера Впровадження Залежностей, непристосованого до використання рядкових ідентифікаторів

Тепер замість створювачів (SingletonCreator та TransientCreator) типом значення асоціативних контейнерів є динамічний масив (std::vector, згідно з [4]), який зберігає в собі послідовність пар std::pair<Tag, Creator>, де Tag – тип рядкових ідентифікаторів усіх реалізацій інтерфейсу, ідентифікатор типу якого є ключем std::type_index, який відповідає даному значенню, а Creator – створювач конкретної реалізації (SingletonCreator або TransientCreator).

Тепер після реєстрації типу SomeType (методами register_singleton<SomeType> або register_transient<SomeType>, що належать DIContainerBuilder), контейнер зберігає усю необхідну інформацію, аби знайти реалізацію не тільки за її інтерфейсом, а й за рядковим ідентифікатором, який їй було присвоєно. А саме, тепер користувач контейнера має явно вказати рядковий ідентифікатор типу, який

реєструє. Інакше буде обрано рядковий ідентифікатор за замовчуванням – статична константа DIContainer::DEFAULT_TAG.

Наприклад, щоб зареєструвати тип Implementation1 як реалізацію інтерфейса Interface, необхідно написати наступний код:

```
DIContainer container = DIContainerBuilder{}
      // інші реєстрації
      .register_singleton<Implementation1>()
            .as_interface<Interface>
            .with_tag("implementation1-tag")
            .done()
      // інші реєстрації
      .build();
```

Лістинг 26: Реєстрація типу Implementation1 з прив'язкою до інтерфейса Interface та визначеним рядковим ідентифікатором

До цих змін також слід пристосувати метод resolve<SomeType>, що належить класу DIContainer. А саме, тепер resolve буде не шукати потрібного створювача, що відповідає ключу typeid(SomeType) асоціативних контейнерів, а шукакати потрібного створювача у послідовностях пар TagsMappingSingletonVec та TagsMappingTransientVec, що відповідають ключу typeid(SomeType) за рядковим ідентифікатором типу Tag. Потрібний користувачеві рядковий ідентифікатор буде передаватись параметром методу resolve:

```
template<typename Dependency>
Dependency& resolve(Tag tag = DEFAULT_TAG) const;
```

Лістинг 27: Сигнатура методу DIContainer::resolve , пристосованого до використання рядкових ідентифікаторів

Варто зазначити, що, за замовчуванням (виклик методу resolve без явного вказування потрібного рядкового ідентифікатора), контейнер буде шукати створювача, який відповідає рядковому ідентифікатору DEFAULT_TAG (з сигнатури методу resolve).

Повертаючись до Лістингу 26, тепер можна зареєструвати реалізації Implementation1 та Implementation2 інтерфейсу Interface, вказавши їх рядкові ідентифікатори:

```
int main()

{
    DIContainer container = DIContainerBuilder{}
        .register_singleton<InterfaceUser>()
            .done()
        .register_singleton<Implementation1>()
            .as_interface<Interface>
            .with_tag("implementation1-tag")
```

```
            .done()
        .register_singleton<Implementation2>()
            .as_interface<Interface>
            .with_tag("implementation2-tag")
            .done()
        .build();

    InterfaceUser& user = container.resolve<InterfaceUser>(); // Помилка!!!
}
```

Лістинг 28: Невдала спроба реєстрації типів InterfaceUser, Implementation1 та Implementation2 з використанням рядкових ідентифікаторів

Однак, виникне помилка зі сповіщенням, що реалізацію інтерфейсу Interface з рядковим ідентифікатором DEFAULT_TAG не знайдено. Чому так? Проблема у створювачі, що відповідає типу InterfaceUser. А саме, створювач типу SingletonCreator, що відповідає типу InterfaceUser, здійснює пошук залежності з типом Interface, вказуючи рядковий ідентифікатор за замовчуванням. Тобто, замість викликів resolve<Interface>(“implementation1-tag”) або resolve<Interface>(“implementation2-tag”) створювач робить resolve<Interface>(), що еквівалентно resolve<Interface>(DEFAULT_TAG). Справді, користувач не реєстрував жодну реалізацію, присвоївши їй рядковий ідентифікатор DEFAULT_TAG.

Для вирішення цього в межах розробленої бібліотеки було додано ще один тип конфігурацій – with_tag_of_dependency_at<Index>:

```
template<std::size_t DependencyIndex>
RegistrationBuilder& with_tag_of_dependency_at(Tag tag);
```

Лістинг 29: Сигнатура методу для вказування рядкових ідентифікаторів для впроваджених залежностей

Вона існує для того, щоб вказати, який саме рядковий ідентифікатор буде використано для пошуку однієї з залежностей типу, що реєструється. Цій залежностій відповідає індекс DependencyIndex зі списку залежностей типу, який реєструє користувач.

Тепер можна виправити помилку, що виникла в Лістингу 28:

```
int main()

{
    DIContainer container = DIContainerBuilder{}
    .register_singleton<InterfaceUser>()
       .with_tag_of_dependency_at<0>("implementation1-tag") // ідентифікатор залежності
       .done()
    .register_singleton<Implementation1>()
       .as_interface<Interface>
       .with_tag("implementation1-tag")
       .done()
    .register_singleton<Implementation2>()
```

```
        .as_interface<Interface>
        .with_tag("implementation2-tag")
        .done()
    .build();

    InterfaceUser& user = container.resolve<InterfaceUser>();
    // далі можна використовувати user як повноцінний об'єкт типу InterfaceUser
}
```

Лістинг 30: Вдала спроба реєстрації типів InterfaceUser, Implementation1 та Implementation2 з використанням рядкових ідентифікаторів

Тепер в об'єкт user впроваджено обрану реалізацію інтерфейсу Interface, а саме – Implementation1. Якщо необхідно впровадити Implementation2 замість Imlementation1, достатньо просто передати параметром методу with_tag_dependency_at рядковий ідентифікатор, що відповідає реалізації Implementation2 – «implementation2-tag».

3.3. Конфігураційні можливості розробленого контейнера

Розроблений Динамічний Контейнер Впровадження Залежностей підтримує наступні конфігураційні можливості реєстрацій типів:

1. Обрати тип життєвого циклу для об'єктів реєстрованого типу Type: єдиносутній чи тимчасовий. Методи, що реалізують це:
    a. DIContainerBuilder::register_singleton<Type>()
    b. DIContainerBuilder::register_transient<Type> ()
2. Присвоїти реєстрованому типу рядковий ідентифікатор (tag) для вибору необхідної реєстрації/реалізації інтерфейсу. Методи, що реалізують це:
    a. RegistrationBuilder::with_tag(tag)
3. Зафіксувати рядковий ідентифікатор залежності (tag), з яким її буде впроваджено в реєстрований об'єкт. Залежність обирається за індексом у списку залежностей реєстрованого типу. Методи, що реалізують це:
    a. RegistrationBuilder::with_tag_of_dependency_at<Index>(tag)
4. Зафіксувати інтерфейс Interface, яким надаватиметься доступ до створеного екземпляру типу Type. Методи, що реалізують це:
    a. RegistrationBuilder::as_interface<Interface>()
5. Обрати типи параметрів Args, з якими буде сконструйовано об'єкт. Методи, що реалізують це:
    a. RegistrationBuilder::constructed_with<Args…>()

Конфігураційне обмеження розробленого Динамічного Контейнера Впровадження Залежностей:

Аби тип Type можна було зареєструвати, необхідно щоб виконувалась хоча б одна з наступних умов:

1. Тип Type має містити єдиний статичний метод create з наступною сигнатурою:

```
static Type create(Args... args);
```

Лістинг 31: Сигнатура статичного метода Type::create

2. При реєстрації типу Type явно вказано список параметрів, з яким його треба сконструювати, шляхом виклику методу constructed_with, що належить класу RegistrationBuilder.

3.4. Глобальна конфігурація Динамічного Контейнера Впровадження Залежностей

Розроблений Динамічний Контейнер Впровадження Залежностей вирішує усі пункти «Задачі розробки Динамічного Контейнера Впровадження Залежностей», окрім одного: він все ще не пропонує засобів для покращення паралельної розробки.

Справді, цілком можливо реалізовувати взірець проектування «Центр Композиції» (Composition Root, згідно з [2]), проте це є вразливим місцем під час паралельної розробки. Контейнер підтримує конфігурацію на основі коду, і щоб налаштувати контейнер, треба створити один файл, в якому будуть конфігурації контейнера, – Центр Композиції. Однак, як тільки декілька розробників починають паралельно розширювати цей файл, додаючи нові конфігурації або модифікуючи старі, виникає безліч конфліктів злиття їх змін; до того ж, Центр Композиції швидко стає дуже складним для читання під час розширення проєкту.

Розроблена бібліотека fast_di пропонує набір засобів для покращення паралельної розробки, які забезпечують можливість розбивати Центр Композиції на декілька файлів, які можуть знаходитись в різних місцях проєкту, та які можуть незалежно один від одного створювати декілька розробників паралельно. Підмножина бібліотеки fast_di, що надає такі засоби, має назву global_di.

Визначення: **Глобальна Конфігурація Контейнера Впровадження Залежностей** – налаштування Контейнера Впровадження Залежностей, за якого конфігурації розміщуються в окремих файлах та стосуються єдиного глобального Контейнера Впровадження Залежностей.

Згідно з [2], налаштування Контейнера Впровадження Залежностей на основі конфігураційних файлів має недолік відносно налаштування на основі коду. А саме, конфігураційні файли не можуть забезпечити такого ж ефективного статичного аналізу та безпеки рівня типів, як конфігурація на основі коду. Тому, конфігураційні засоби бібліотеки global_di було розроблено таким чином, аби мовою конфігураційних файлів Глобальної Конфігурації Контейнера Впровадження Залежностей була нативна мова бібліотеки fast_di – C++. Такий підхід відокремлює код конфігурацій від коду бізнес-логіки користувацьких програм, забезпечує безпеку рівня типів та ефективний статичний аналіз конфігурацій, а також значно покращує паралельну розробку. Однак наразі розроблені засоби бібліотеки fast_di не підтримують пізнє зв'язування конфігурацій – основну перевагу налаштувань на основі конфігураційних файлів, згідно з [2].

Засоби Глобальної Конфігурації Динамічного Контейнера Впровадження Залежностей бібліотеки global_di починаються з сутності GlobalDI – глобальною обгорткою об'єкта типу DIContainerBuilder. GlobalDI реалізовано згідно із взірцем проектування «Одинак», що описано в [1].

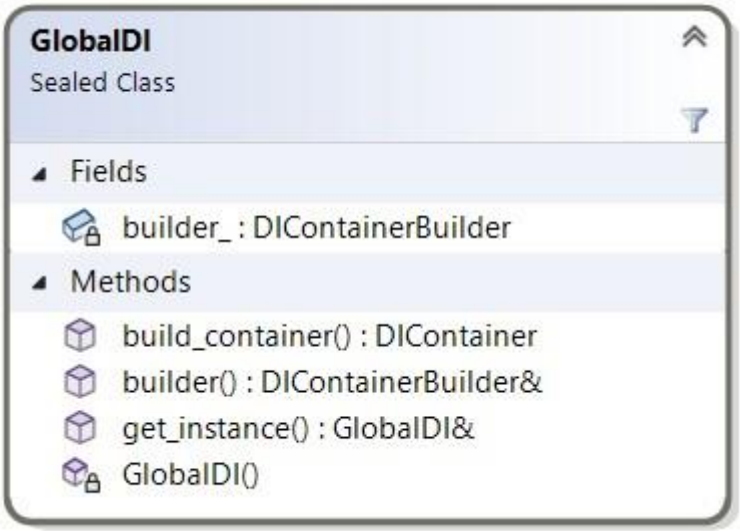

Рис. 3: Діаграма сутності GlobalDI з бібліотеки fast_di

До цього глобального «Будівельника» контейнерів можна звернутися з будь-якої точки проєкту. Першим рішенням було здійснювати налаштування контейнера, доступаючись до об'єкту GlobalDI зі звичайних файлів мови C++ з розширенням «cpp». Схоже рішення для реєстрації сутностей у глобальній фабриці було запропоновано Александреску в [3].

```
namespace {
    const bool entity_registered = []() {
```

```
        using fast_di::global_di::GlobalDI;
        using fast_di::dynamic_di::DIContainerBuilder;
        DIContainerBuilder& builder = GlobalDI::builder();
        builder.register_singleton<Entity>()
            .as_interface<IEntity>()
            .with_tag("entity-tag")
            .done();
        return true;
    } ();
}
```

Лістинг 32: Невдала спроба глобальної реєстрації типу Entity в Динамічному Контейнері Впровадження Залежностей

У даному прикладі реєстрація сутності Entity з єдиносутнім життєвим циклом в глобальному контексті файлу Entity.cpp.

У чому проблема такого підходу? На жаль, у випадку з реєстрацією сутностей мова C++ не дає жодної гарантії, що на всіх платформах змінну entity_registered буде ініціалізовано. Тобто, немає чіткого визначення, чи лямбда-вираз, що слугує ініціалізатором глобальної змінної entity_registered, взагалі буде викликано, а отже, сутність Entity може не бути зареєстрованою через такі оптимізації компілятора. Мова C++ має способи для обходу даної проблеми, проте рішення, яке б працювало на всіх платформах, в межах даної роботи знайдено не було.

Натомість було запропоновано ввести конфігураційні файли з розширенням «hppdi» та програму з назвою «Агрегатор Глобальних Конфігурацій», яка б генерувала необхідний код для агрегації усіх конфігурацій в одній функції, яку буде викликано з користувацької функції main, що буде гарантувати застосування усіх конфігурацій, які було поміщено в межах області проєкту, на якій оперує Агрегатор.

Розгляньмо приклад конфігураційного файлу Entity.hppdi:

```
#include "fast_di/global/GlobalDI.hpp"
#include "Entity.hpp"

namespace DI
{
    DI_DYNAMIC_SUBSCRIBER_FOR(Entity)
    {
        fast_di::global_di::GlobalDI::builder()
            .register_singleton<Entity>().done();
    }
}
```

Лістинг 33: Вміст конфігураційного файлу Entity.hppdi

Як бачимо, вміст Entity.hppdi цілком відповідає синтаксису мови C++. Насправді розширення «hppdi» необхідно виключно для Агрегатора Глобальних Конфігурацій, аби відрізняти конфігураційні файли від файлів з іншою користувацькою програмною логікою.

У файлі Entity.hppdi здійснюється глобальна реєстрація сутності Entity з єдиносутнім життєвим циклом.

DI_DYNAMIC_SUBSCRIBER_FOR(Entity) – макрос мови C++, який буде розгорнуто в заголовок функції subscribe_Entity() на етапі препроцесингу коду програми. Вміст файлу Entity.hppdi еквівалентний наступному:

```
#include "fast_di/global/GlobalDI.hpp"
#include "Entity.hpp"

namespace DI
{
    void subscribe_Entity()
    {
        fast_di::global_di::GlobalDI::builder()
            .register_singleton<Entity>().done();
    }
}
```

Лістинг 34: Вміст файлу Entity.hppdi після розкриття макросу DI_DYNAMIC_SUBSCRIBER_FOR

Тож, маємо чітко визначену функцію, яку необхідно просто викликати, аби гарантовано відбулась реєстрація сутності Entity. Однак хто викликатиме цю функцію? Користувач власноруч? – Авжеж ні, адже це дуже ускладнить паралельну розробку. Функцію subscribe_Entity буде викликано з коду, який згенерує Агрегатор Глобальних Конфігурацій.

Розгляньмо код, що згенерує Агрегатор Глобальних Конфігурацій, спираючись на наявну глобальну конфігурацію розміщену у файлі Entity.hppdi. Буде згенеровано два файли: DI.hpp та DI.cpp. Шляхи до цих файлів вказує користувач, передаючи їх параметрами.

Згенерований файл DI.hpp:

```
#ifndef SUBSCRIBE_ALL_GLOBAL_DI_HPP_
#define SUBSCRIBE_ALL_GLOBAL_DI_HPP_

void subscribe_all_dependencies();

#endif //SUBSCRIBE_ALL_GLOBAL_DI_HPP_
```

Лістинг 35: Вміст файлу DI.hpp, згенерованого Агрегатором Глобальних Конфігурацій

Згенерований файл DI.cpp:

```
#include "absolute/path/to/Entity.hppdi"

void subscribe_all_dependencies()
{
      DI::subscribe_Entity();
      // далі можуть іти інші глобальні конфігурацій для інших сутностей
```

```
}
```

Лістинг 36: Вміст файлу DI.cpp, згенерованого Агрегатором Глобальних Конфігурацій

Як бачимо, просто згенерував визначення для функції subscribe_all_dependencies, у якому й викликано функцію subscribe_Entity. Якби в межах області проєкту, на якій оперує «Агрегатор Глобальних Конфігурацій», було більше глобальних конфігурацій (для інших сутностей), то за викликом функції subscribe_Entity Агрегатор додав би виклики для всіх глобальних конфігурацій з області проєкту, на якій він оперує.

Хто ж викличе функцію subscribe_all_dependencies? Немає жодних проблем, через які користувачу було б шкідливо викликати функцію subscribe_all_dependencies вручну. Її буде викликано один раз на початку користувацької функції main, що не завдасть жодної шкоди паралельній розробці. До того ж, вміст користувацького файлу, що містить функцію main, тепер можна зробити сталим відносно будь-якого розширення проєкту. А саме, нехай таким файлом буде main.cpp, розглянемо його вміст:

```cpp
#include "fast_di/global/GlobalDI.hpp"
#include "DI.hpp"
#include "Entity.hpp"

int main()
{
    subscribe_all_dependencies();
    DIContainer container = fast_di::global_di::GlobalDI::build_container();
    Entity& root = container.resolve<Entity>();
    root.start_of_the_program();

    return EXIT_SUCCESS;
}
```

Лістинг 37: Приклад вмісту файлу main.cpp, пристосованого до паралельної розробки

Даний код спочатку викликає процедур subscribe_all_dependencies, чим застосовує усі глобальні конфігурації Контейнера Впровадження Залежностей, а далі отримує доступ до єдиносутнього об'єкту типу Entity з Динамічного Контейнера Впровадження Залежностей, який сконструйовано глобальною сутністю GlobalDI. Метод start_of_the_program розпочинає роботу програми за логікою сутності Entity.

Цей приклад гарно ілюструє потужність Контейнерів Впровадження Залежностей відносно ефективності розробки. Сутність Entity можна обрати таким чином, щоб вона була логічним початком усієї програми. З допомогою Впровадження Залежностей, в Entity будуть впроваджуватись інші типи,

множину яких можуть розширювати декілька програмістів паралельно без жодних конфліктів злиття змін.

РОЗДІЛ 4: Створення застосунку Задачника (tmcli), базованого на Контейнері Впровадження Залежностей

4.1. Специфікація проєкту tmcli

Застосунок tmcli – програма з інтерфейсом командного рядка, здатна виконувати базові операції із сутністю Task (задача), екземпляри якої зберігаються в базі даних PostgreSQL. Збірка проєкту базується на утиліті автоматизованої збірки CMake, яку описано в [5].

Функціонал застосунку простий та налічує наступний набір команд:

- **all** – вивести на екран усі екземпляри сутності Task з бази даних:
  - Без параметрів та помилок.
- **new** – створити новий екземпляр сутності Task та зберегти його в базі даних:
  - *Обов'язкові параметри:* name (ім'я задачі);
  - *Додаткові параметри:* description (опис задачі);
  - *Помилки:* наявний в базі даних екземпляр задачі вже має ім'я, рівне значенню введеного параметру name.
- **del** – видалити екземпляр сутності Task за його назвою з бази даних:
  - *Обов'язкові параметри:* name (ім'я задачі, яку буде видалено).
  - *Помилки:* серед наявних в базі даних екземплярів задач немає жодного, який би мав ім'я рівне значенню введеного параметру name.
- **chk** – помітити задачу як виконану та зберегти ці зміни в базі даних:
  - *Обов'язкові параметри:* name (ім'я задачі, яку необхідно помітити як виконану).
  - *Помилки:* серед наявних в базі даних екземплярів задач немає жодного, який би мав ім'я рівне значенню введеного параметру name.

Виконання команд можливе після запуску застосунку та відбувається в один з перелічених способів:

- Здійснюється введення рядка, який містить назву команди на початку, а також значення усіх параметрів після неї; кожному значенню, введеному у такий спосіб, передує назва відповідного параметру з префіксом «--»; список вказаних параметрів має включати в себе усю множину обов'язкових параметрів команди.
  - Приклади валідних вхідних даних для виклику команди **new**:
    - «new --name some_task» - створення задачі з ім'ям some_task;

        - «new --name some_task --description some_description» - створення задачі з ім'ям some_task та описом some_description.
- Здійснюється введення рядка, який містить назву команди на початку, а також значення довільних параметрів після неї; кожному значенню, введеному у такий спосіб, передує назва відповідного параметру з префіксом «--»; список вказаних параметрів може містити довільні параметри; на отримання усіх невказаних значень обов'язкових параметрів програма зробить запит одразу після введення рядка описаного рядка.
    - Приклади валідного створення задач у такий спосіб:
        - «new» - після такого введення програма зробить наступний запит на отримання значення обов'язкового параметру з назвою name: «Please input name:»; після чого вводимо бажане значення параметру name: «some_name»; після такої процедури буде створено задачу з ім'ям «some_name» та пустим описом.

4.2. Архітектура проєкту tmcli

Проєкт tmcli було спроектовано таким чином, щоб досягти якнайкращої паралельної розробки. Загалом, проєкт можна розбити на три модулі, впорядкованих від нижчого до вищого рівня:

- tmcli_database – модуль, що містить логіку взаємодії з базою даних, набір команд для роботи з базою даних, валідатори параметрів таких команд;
- tmcli_cli – модуль, що містить логіку введення та виведення даних, синтаксичні аналізатори вхідних даних, набір команд командного рядка, валідатори параметрів таких команд, та конфігурацію користувацького інтерфейсу;
- tmcli_root – вхідна точка проєкту, в якій відбувається застосування усіх конфігурацій та запуск застосунку.

Розгляньмо архітектури окремих модулів проєкту tmcli, починаючи з модуля найнижчого рівня.

Рис. 4: Діаграма архітектури модуля tmcli_database проєкта tmcli

Опис діаграми ірхітектури модулі tmcli_database:

- DbErrorType – тип-перелік, кожен варіант якого відповідає помилці, яка може виникнути під час взаємодії з базою даних:
  - Варіанти:
    - BAD_CONNECTION – помилка з'єднання з базою даних;
    - BAD_QUERY – некоректно сформований запит до бази даних;
    - INTERNAL_ERROR – порушення обмежень бази даних (наприклад, порушення обмеження зовнішнього ключа);
    - UNSUPPORTED_ARG – спроба передати команді бази даних непідтримуваний параметр;
    - REQUIRED_ARG_MISSING – спроба викликати команду бази даних, не вказавши значення для якогось з обов'язкових параметрів;
    - NOT_FOUND – взаємодія з екземпляром сутності Task, якого не існує в базі даних.
- DbError<Error> - клас-помилка, реалізація абстракції std::exception (згідно з [4]); Error – один з варіантів DbErrorType.
- QueryResult – структура даних для збереження множини рядків довільної сутності бази даних; використовується як тип відповідь бази даних на запит:
  - Поля:

        - affected_rows_ - кількість рядків бази даних, які було модифіковано внаслідок застосування запиту;
        - container_ - контейнер, у якому зберігаються значення, отримані з бази даних.
    - Методи:
        - add – метод додавання рядка до контейнера;
        - affected_rows – селектор для поля affected_rows_;
        - begin – фабричний метод для ітератора на початок контейнера;
        - end – фабричний метод для ітератора на кінець контейнера;
        - is_empty – перевірка на те, чи контейнер пустий;
        - QueryResult – конструктор
        - reserve – завчасне виділення достатньої кількості пам'яті для контейнера, аби знизити кількість перевиділень пам'яті;
        - size – селектор для розміру контейнера.
- ISqlConnection – інтерфейс з'єднання з базою даних:
    - Методи:
        - ~ISqlConnection – віртуальний деструктор;
        - query – метод для здійснення SQL-запиту, використовуючи встановлене з'єднання з базою даних.
    - Реалізації:
        - PqxxConnection – клас, що встановлює з'єднання з базою даних PostgreSQL та здатен здійснювати запити до неї.
- IDbCommand – інтерфейс команди бази даних; кожна з команд інкапсулює певну логіку взаємодії з базою даних (наприклад, здійснює SQL-запит):
    - Методи:
        - ~IDbCommand – віртуальний деструктор;
        - execute – метод виконання команди;
        - get_required_args – селектор для обов'язкових параметрів команди;
        - get_optional_args – селектор для вибіркових параметрів команди.
    - Реалізації:
        - UpdateTaskDbCommand – команда оновлення екземпляру задачі, що зберігається в базі даних;
        - CreateTaskDbCommand – команда створення екземпляру задачі в базі даних;
        - ReadAllTasksDbCommand – команда зчитування усіх наявних екземплярів задач з бази даних;
        - ReadTaskDbCommand – команда зчитування екземпляру задачі з бази даних за її ідентифікатором;
        - DeleteTaskDbCommand – команда видалення екземпляру задачі з бази даних за її ідентифікатором;

- DbCommandArgsValidator – реалізація взірця проектування «Декоратор» (згідно з [1]); здійснює валідацію параметрів команд бази даних.

Переваги та недоліки такої архітектури:

1. Як користувач, так і команди бази даних цілком ізольовані від типу бази даних, оскільки останній цілком інкапсулюється реалізацією інтерфейсу ISqlConnection. Усі сутності здійснюють з'єднання з базою даних саме через інтерфейс ISqlConnection, через який вони здатні здійснювати SQL-запити. Завдяки такому застосуванню Впровадження Залежностей можна розширбвати ієрархію, що відповідає інтерфейсу ISqlConnection; наприклад, створити нову реалізацію ISqlConnection, яка б могла з'єднуватись з базою даних SQLite. Проте у даного підходу є і недолік. Сутності, що залежать від інтерфейсу ISqlConnection не можуть використовувати діалекти мови SQL, специфічні для окремих баз даних. Це випливає з Принципу Підстановки Барбари Лісков, визначеного в [6], оскільки використання діалектів спричинило б недієздатність або некоректну роботу окремих реалізацій ISqlConnection, які не здатні сприймати ці діалекти.
2. Множину команд бази даних можуть розширювати скільки завгодно розробників одночасно, оскільки команди реалізують інтерфейс IDbCommand та впроваджуються як залежності у всі місця, де їх потребують. Набір їх параметрів не зафіксовано статично, оскільки метод IDbCommand::execute приймає динамічний контейнер з параметрами, який користувач може наповнити яким завгодно набором параметрів, залежно від того, яку команду він використовує.
3. Застосування взірця проектування «Декоратор» (Decorator, згідно з [1]) дозволяє додавати нові валідації або інші функціональні обгортки для команд баз даних непомітно для інших класів. Завдяки засобам Впровадження залежностей можна розширювати множину «Декораторів» без жодної необхідності змінювати та перекомпільовувати інші сутності, до того ж, паралельно – декількома розробниками одночасно.

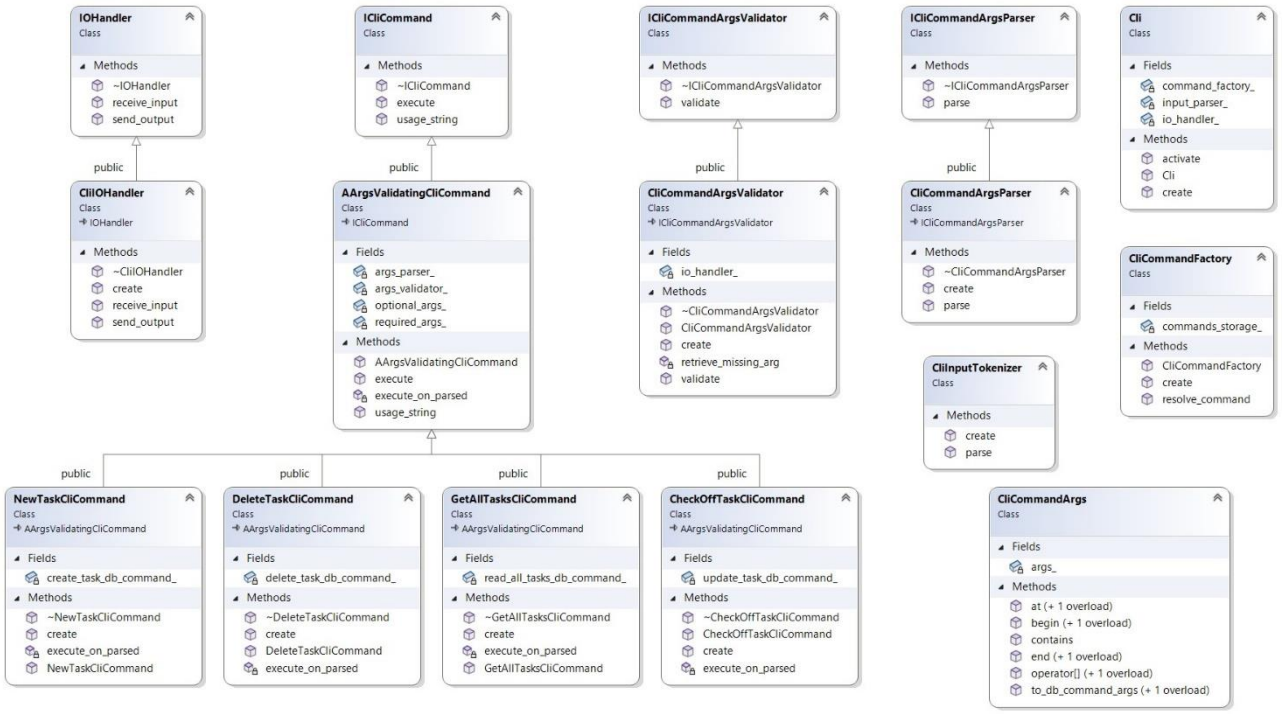

Рис. 5: Діаграма архітектури модуля tmcli_cli проєкта tmcli

Опис діаграми архітектури модуля tmcli_cli:

- IOHandler – інтерфейс для отримання вхідних даних, а також виведення вихідних даних застосунку:
  - Методи:
    - ~IOHandler – віртуальний деструктор;
    - receive_input – метод для отримання вхідних даних;
    - send_input – метод для виведення вихідних даних застосунку.
  - Реалізації:
    - CliIOHandler – клас для введення/виведення на основі інтерфейсу командного рядка.
- CliInputTokenizer – клас, що здійснює розбиття на лексеми вхідні дані застосунку:
  - Методи:
    - create – статична обгортка для конструктора, необхідна для Контейнера Впровадження Залежностей;
    - parse – метод, що здійснює розбиття вхідних даних на лексеми.
- ICliCommandArgsParser – інтерфейс для синтаксичного аналізу та розбору лексем, на які було розбито вхідні дані застосунку:
  - Методи:
    - ~ICliCommandArgsParser – віртуальний деструктор
    - parse – метод, що здійснює синтаксичний аналіз та розбір лексем.

    - Реалізації:
        - CliCommandArgsParser – клас, який здійснює синтаксичний аналіз та розбір лексем згідно з описаним функціоналом застосунку.
- ICliCommandArgsValidator – інтерфейс для валідації списку параметрів, які йдуть на вхід командам командного рядка:
    - Методи:
        - ~ICliCommandArgValidator – віртуальний деструктор;
        - validate – метод для здійснення валідації списку параметрів команд командного рядка.
    - Реалізації:
        - CliCommandArgsValidator – валідатор параметрів, що забезпечує наявність усіх обов'язкових параметрів та перевіряє, чи дані на вхід параметри підтримуються застосунком.
- ICliCommand – інтерфейс команди командного рядка:
    - Методи:
        - ~ICliCommand – віртуальний деструктор;
        - execute – метод виконання команди;
        - usage_string – селектор для інструкцій користування командою.
    - Реалізації:
        - AArgsValidatingCliCommand - «Шаблон» (реалізація взірця проектування Template Method згідно з [1]) для команд з валідацією їх параметрів; execute_on_parsed – «Шаблонний Метод»;
        - NewTaskCliCommand – команда створення нового екземпляра задачі;
        - DeleteTaskCommand – команда видалення екземпляра задачі;
        - GetAllTasksCliCommand – команда зчитування усіх наявних екземплярів задач;
        - CheckOffTaskCliCommand – команда помічення екземпляра задачі як виконаного.
- CliCommandFactory – параметризована «фабрика» команд командного рядка (Parametrized Factory Method, згідно з [1]):
    - Поля:
        - commands_storage_ - асоціативний контейнер для динамічної диспетчеризації отримання доступу до команд;
    - Методи:
        - CliCommandFactory – конструктор;
        - create – статична обгортка для конструктора, необхідна для Контейнера Впровадження Залежностей;

        - resolve_command – метод, що надає доступ до необхідної команди користувачам фабрики.
- CliCommandArgs – контейнер для збереження параметрів до команд командного рядка:
    - Поля:
        - args_ - внутрішній асоціативний контейнер
    - Методи:
        - at – отримання параметра за ключем; повертається помилка у разі, якщо вказаний ключ не було знайдено;
        - begin – фабричний метод для ітератора на початок контейнера;
        - contains – метод перевірки на існування ключа в контейнері;
        - end – фабричний метод для ітератора на кінець контейнера;
        - operator[] – довизначення оператора «квадратних дужок» для доступу до значень параметрів за ключем. Поведінка є невизначеною у разі, якщо ключ не існує.
        - to_db_command_args – перетворення вмісту контейнера на контейнер, який йде на вхід командам бази даних.
- Cli – коренева сутність для модуля tmcli_cli; запускає роботу застосунка та надає користувачу інтерфейс командного рядка:
    - Поля:
        - command_factory_ - фабрика команд командного рядка;
        - input_parser_ - екземпляр типу CliInputTokenizer;
        - io_handler_ - об'єкт, що відповідає на введення/виведення користувацького інтерфейсу застосунку.

Переваги такої архітектури:

1. Множину команд командного рядка можуть розширювати декілька розробників паралельно, згідно з поняттям паралельної розробки;
2. Реалізацію майже будь-якої сутності можна підмінити або задекорувати, завдяки Впровадженню Залежностей;
3. Абстракція введення/виведення ізолює застосунок того, як користувач буде з ним взаємодіяти: можливий перехід від інтерфейсу командного рядка до REST API, наприклад, без жодної модифікації коду, що вже існує.
4. Можливість чітко відокремлювати команди, що здійснюють валідацію згідно з AArgsValidatingCliCommand від інших команд (наприклад від тих, що не потребують валідації). Наслідування команди від «Шаблону» AArgsValidatingCliCommand автоматично надає функціонал валідації параметрів цій команді.

Недоліки такої архітектури:

1. Реалізація AArgsValidatingCliCommand згідно із взірцем проектування «Шаблонний Метод» унеможливлює перевикористання коду його наслідників для інших способів валідації параметрів або їх відсутності.

На прикладі проєкту tmcli можна порівняти взірці проектування «Декоратор» та «Шаблонний метод» (Decorator та Template Method, згідно з [1]). В більшості випадків «Декоратор» є більш вдалим застосуванням, у випадку, коли треба додати функціонал до якихось класів без їх модифікації. Це тому, що «Декоратор» уникає сильної зв'язності між декорованим класом та самим «Декоратором», що не можна сказати про «Шаблонний Метод», який реалізують шляхом наслідуванням, яке й спричиняє його сильне зв'язування з наслідниками. Кожного разу, коли клас наслідується від свого «Шаблону», його тепер ніколи не можна буде перевикористати без функціоналу, який додав «Шаблон». На противагу, «Декоратор» цілком дозволяє таке перевикористання.

Проте для Контейнерів Впровадження Залежностей, в яких не реалізовано можливість Перехоплення (Interception, згідно з [2]), використання «Декоратора» може дуже ускладнювати процедуру конфігурації; тому в деяких випадках «Шаблонний Метод» справді буде більш вдалим рішенням за «Декоратор». Це буде розглянуто далі у роботі.

Щодо модуля tmcli_root, то він не робить вкладу до архітектури застосунку, адже він містить лише загальне налаштування проєкту, зокрема конфігурацію Контейнера Впровадження Залежностей.

4.3. Застосування Динамічного Контейнера Впровадження Залежностей у розробці застосунку tmcli

По всьому проєкту tmcli розміщено файли з розширенням «hppdi», що містять глобальні конфігурації для глобального Динамічного Контейнера Впровадження Залежностей, доступ до якого надає бібліотека global_di. Такі глобальні конфігурації розбито по файлам таким чином, що кожному файлу з розширенням «hpp», у якому оголошено сутність, яку буде зареєстровано в Динамічному Контейнері Впровадження Залежностей, відповідає єдиний файл з розширенням «hppdi», у якому відбувається реєстрація конкретно цієї сутності.

Наприклад, реєстрація команди командного рядка NewTaskCliCommand відбувається наступним чином у файлі NewTaskCliCommand.hppdi:

```
namespace DI {

    DI_DYNAMIC_SUBSCRIBER_FOR(NewTaskCliCommand) {
```

```
            using namespace tmcli::cli::command;

            fast_di::global_di::GlobalDI::builder()
                  .register_singleton<NewTaskCliCommand>()
                        .as_interface<ICliCommand>()
                        .with_tag("new_task_cli_command")
                        .with_tag_of_dependency_at<2>("validated_db_create_task")
                        .done();
      }

}
```

Лістинг 38: Вміст файлу NewTaskCliCommand.hppdi

Реєстрація типу NewTaskCliCommand надає наступні конфігурації:

1. Об'єктам типу присвоєно єдиносутній життєвий цикл;
2. З об'єктами типу користувач взаємодіятиме через інтерфейс ICliCommand;
3. Типу присвоєно рядковий ідентифікатор «new_task_cli_command»;
4. Конструюючи об'єкт цього типу, Динамічний Контейнер Впровадження Залежностей впровадить в нього залежність (якій відповідає індекс 2 у списку залежностей типу NewTaskCliCommand) з рядковим ідентифікатором «validated_db_create_task».

Більш прості типи, які не вимагають конфігурацій з рядковими ідентифікаторами чи прив'язки до інтерфейсу, мають простіший процес реєстрації. Наприклад, реєстрація типу Cli відбувається в наступний спосіб у файлі Cli.hppdi:

```
namespace DI {
      DI_DYNAMIC_SUBSCRIBER_FOR(Cli) {
            using namespace tmcli::cli;
            fast_di::global_di::GlobalDI::builder()
                  .register_singleton<Cli>().done();
      }
}
```

Лістинг 39: Вміст файлу Cli.hppdi

До реєстрації типу Cli входить лише присвоєння його об'єктам єдиносутнього життєвого циклу.

Окрім конфігураційних файлів, проєкт tmcli містить певні скрипти – налаштовану систему збірки, що враховує особливість засобів Глобальної Конфігурації Контейнера Впровадження Залежностей бібліотеки global_di. Збірку проєкту tmcli можна поділити на наступні етапи:

1. Компіляція Агрегатора Глобальних Конфігурацій, якщо його ще не було скомпільовано;
2. Генерація файлів з агрегованими Глобальними Конфігураціями Агрегатором Глобальних Конфігурацій: DI.hpp, DI.cpp;
3. Подальша збірка проєкту засобами CMake, як звичайного CMake-проєкту.

В результаті такої збірки буде отримано виконуваний файл застосунку tmcli.

Файли, що генерує Агрегатор Глобальних Конфігурацій, – DI.hpp та DI.cpp – звичайні C++-файли, що містять функції для застосування Глобальних Конфігурацій Контейнера Впровадження Залежностей. Ці файли вважаються файлами збірки, тому не входять до вихідного коду tmcli. Файл DI.hpp містить єдине оголошення функції: void subscribe_all_dependencies(). Файл DI.cpp включає в себе приєднання усіх файлів з розширенням «hppdi» директивою препроцесора #include, а також реалізацію функції subscribe_all_dependencies, у якій викликаються функції Глобальної Конфігурації з кожного з включених конфігураційних файлів:

```
void subscribe_all_dependencies()
{
	DI::subscribe_Cli();
	DI::subscribe_CliCommandArgsParser();
	DI::subscribe_CliCommandArgsValidator();
	DI::subscribe_CliCommandFactory();
	DI::subscribe_CheckOffTaskCliCommand();
	DI::subscribe_DeleteTaskCliCommand();
	DI::subscribe_GetAllTasksCliCommand();
	DI::subscribe_NewTaskCliCommand();
	DI::subscribe_CliIOHandler();
	DI::subscribe_CliInputTokenizer();
	DI::subscribe_CreateTaskDbCommand();
	DI::subscribe_DeleteTaskDbCommand();
	DI::subscribe_ReadAllTasksDbCommand();
	DI::subscribe_ReadTaskDbCommand();
	DI::subscribe_UpdateTaskDbCommand();
	DI::subscribe_DBCommandArgsValidator();
	DI::subscribe_PqxxConnection();
}
```

Лістинг 40: Визначення функції subscribe_all_dependencies з файлу DI.cpp

Функцію subscribe_all_dependencies викликано у файлі main.cpp, який має наступний вміст:

```
#include "fast_di/global/GlobalDI.hpp"
#include "DI.hpp"
#include "tmcli_cli/Cli.hpp"

int main()
{
    subscribe_all_dependencies();
    auto container = fast_di::global_di::GlobalDI::build_container();
    tmcli::cli::Cli& cli = container.resolve<tmcli::cli::Cli>();
    cli.activate();

    return EXIT_SUCCESS;
}
```

Лістинг 41: Вміст файлу main.cpp проєкту tmcli

Після застосування усіх Глобальних Конфігурацій Контейнера Впровадження Залежностей у функції main отримується доступ до об'єкту типу Cli, створеного

Динамічним Контейнером Впровадження Залежностей, та викликаємо метод activate, після чого стосунок стає готовим до обробки вхідних даних користувача.

4.4. Здатності розширення множин команд tmcli, надбані за рахунок використаних засобів Впровадження Залежностей

До проєкту tmcli входять дві множини команд: команди командного рядка (реалізації інтерфейса ICliCommand) та команди бази даних (реалізації інтерфейса IDbCommand). Як зазначалося раніше у цій роботі, обидві множини можна доповнювати, не порушуючи паралельну розробку. Розгляньмо, яким чином досягається розширення цих множин майже без порушень паралельної розробки.

Візьмемо до уваги додавання нових команд як напрям розширення множин команд. Наявні множини по-різному пристосовані до такого розширення: кожна має свої переваги та недоліки відносно цього напряму.

Почнімо з множини команд командного рядка. Алгоритм додавання нової команди можна описати наступною послідовністю дій:

1. Створити клас, що містить логіку команди, яку необхіно додати;
2. Якщо необхідно додати функціонал стандартної валідації, наявний в класі AArgsValidatingCliCommand, то наслідуємо створений клас від AArgsValidatingCliCommand; інакше – реалізуємо створеним класом інтерфейс ICliCommand;
3. Додаємо конфігураційний файл з розширенням «hppdi» для здійснення реєстрації створеного класу в Контейнері Впровадження Залежностей;
4. Додаємо створену команду до динамічної фабрики CliCommandFactory;
5. Виконуємо збірку проєкту.

Для прикладу, наведемо код, який треба додати, аби розширити множину команд командою рядка командою HelpCliCommand:

1. Визначимо клас HelpCliCommand:

```
namespace tmcli::cli::command
{

class HelpCliCommand : public AArgsValidatingCliCommand
{
public:
	HelpCliCommand(
		args::ICliCommandArgsParser& args_parser,
		args::ICliCommandArgsValidator& args_validator
	);
```

```
        static HelpCliCommand create(
                args::ICliCommandArgsParser& args_parser,
                args::ICliCommandArgsValidator& args_validator
        );

        ~HelpCliCommand() override = default;

private:
        ResultType execute_on_parsed(args::CliCommandArgs&& args) override;
};

}
```

Лістинг 42: Вміст новоствореного файлу HelpCliCommand.hpp (директиви #include та захист від повторного включення пропущено)

```
namespace tmcli::cli::command
{

HelpCliCommand::HelpCliCommand(
        args::ICliCommandArgsParser& args_parser,
        args::ICliCommandArgsValidator& args_validator
)
        : AArgsValidatingCliCommand(
                args_parser,
                args_validator,
                {},                 // назви обов'язкових параметрів
                { "command_name" }  // назви додаткових параметрів
        )
{}

HelpCliCommand HelpCliCommand::create(
        args::ICliCommandArgsParser& args_parser,
        args::ICliCommandArgsValidator& args_validator
) {
        return { args_parser, args_validator };
}

auto HelpCliCommand::execute_on_parsed(args::CliCommandArgs&& args) -> ResultType {
        if (args.contains("command_name")) {
                const std::string& command_name = args["command_name"];
                return std::format("Help Message for {} command.", command_name);
        } else {
                return "General Help Message.";
        }
}

}
```

Лістинг 43: Вміст новоствореного файлу HelpCliCommand.cpp (директиви #include пропущено)

Як видно з визначення, клас HelpCliCommand є наслідником AArgsValidatingCliCommand, тож обрано команда буде мати стандартну валідацію параметрів;

2. Додаймо конфігураційний файл HelpCliCommand.hppdi:

```
namespace DI
{

DI_DYNAMIC_SUBSCRIBER_FOR(HelpCliCommand) {
	using namespace tmcli::cli::command;
	fast_di::global_di::GlobalDI::builder()
		.register_singleton<HelpCliCommand>()
			.as_interface<ICliCommand>()
			.with_tag("help_cli_command")
			.done();
}

}
```

Лістинг 44: Вміст новоствореного файлу HelpCliCommand.hppdi (директиви #include пропущені)

3. Додаємо рядки, виділені жовтим кольором, до файлів динамічної фабрики CliCommandFactory (CliCommandFactory.hpp, CliCommandFactory.cpp та CliCommandFactory.hppdi):

```
CliCommandFactory(
    ICliCommand& new_task_command,
    ICliCommand& get_all_tasks_command,
    ICliCommand& delete_task_command,
    ICliCommand& check_off_task_command,
    ICliCommand& help_command
)
    : commands_storage_({
        { "new", [&new_task_command]() -> ICliCommand& { return new_task_command; } },
        { "all", [&get_all_tasks_command]() -> ICliCommand& { return get_all_tasks_command; } },
        { "del", [&delete_task_command]() -> ICliCommand& { return delete_task_command; } },
        { "chk", [&check_off_task_command]() -> ICliCommand& { return check_off_task_command; } },
        { "help", [&help_command]() -> ICliCommand& { return help_command; }}
    })
{}

static CliCommandFactory create(
    ICliCommand& new_task_command,
    ICliCommand& get_all_tasks_command,
    ICliCommand& delete_task_command,
    ICliCommand& check_off_task_command,
    ICliCommand& help_command
)
{
    return {
        new_task_command,
        get_all_tasks_command,
        delete_task_command,
        check_off_task_command,
        help_command
    };
}

DI_DYNAMIC_SUBSCRIBER_FOR(CliCommandFactory)
{
	fast_di::global_di::GlobalDI::builder()
		.register_singleton<::tmcli::cli::command::CliCommandFactory>()
		.with_tag_of_dependency_at<0>("new_task_cli_command")
		.with_tag_of_dependency_at<1>("all_tasks_cli_command")
		.with_tag_of_dependency_at<2>("del_task_cli_command")
		.with_tag_of_dependency_at<3>("chk_task_cli_command")
		.with_tag_of_dependency_at<4>("help_cli_command")
		.done();
	std::cout << "Registered CliCommandFactory" << std::endl;
```

```
}
```

Лістинг 45: Впровадження новоствореної команди до CliCommandFactory

4. Виконуємо збірку проєкту.

Як бачимо, процедура додавання нової команди командного рядка є досить простою, однак існує декілька недоліків:

- Використання взірця проектування «Шаблонний Метод» спричиняє сильну зв'язність типів команд та типу AArgsValidatingCliCommand.
- Додавання нової команди командного рядка вимагає її впровадження як залежності до динамічної фабрики команд CliCommandFactory, що, згідно з прикладом додавання команди HelpCliCommand, порушує паралельну розробку. Якщо декілька розробників одночасно будуть додавати по новій команді, то зміни у файлах фабрики CliCommandFactory спричинять конфлікти злиття. Проте це не проблема архітектури проєкту; це проблема Контейнера Впровадження Залежностей. Уникнення порушення паралельної розробки в даному випадку було б можливе, якби використаний Динамічний Контейнер Впровадження Залежностей реалізовував би можливість створювати такі фабрики автоматично. Іншими словами, контейнеру бракує автоматизації взірців проектування «Шаблонний Метод» та «Абстрактна фабрика» (описаних у [1]). Без цього неможливо створити динамічну фабрику команд, стійку до паралельної розробки.

Перейдімо до множини команд бази даних, яка на відміну від попередньої множини не має жодного з описаних її недоліків. Алгоритм додавання нової команди можна описати наступною послідовністю дій:

1. Створити клас, що реалізує інтерфейс IDbCommand;
2. Додаємо конфігураційний файл з розширенням «hppdi» для здійснення реєстрації створеного класу в Контейнері Впровадження Залежностей;
3. Декоруємо об'єкти щойно зареєстрованого типу «Декоратором» DbCommandArgsValidator, додавши необхідну конфігурацію до створеного файлу з розширенням «hppdi»;
4. Виконуємо збірку проєкту.

Для прикладу, наведемо код, який треба додати, аби розширити множину команд бази даних командою DeleteAllTasksDbCommand:

1. Створюємо клас DeleteAllTasksDbCommand:

```
namespace tmcli::database::command
{
```

```
class DeleteAllTasksDbCommand : public IDbCommand
{
public:
	explicit DeleteAllTasksDbCommand(connection::ISqlConnection& connection);
	~DeleteAllTasksDbCommand() override = default;

	static DeleteAllTasksDbCommand create(connection::ISqlConnection& connection);

public:
	QueryResult execute(const ArgsContainer& args) override;
	const ArgsNamesIterable& get_required_args() const override;
	const ArgsNamesIterable& get_optional_args() const override;

private:
	connection::ISqlConnection& connection_;
};

}
```

Лістинг 46: Вміст новоствореного файлу DeleteAllTasksDbCommand.hpp (директиви #include та захист від повторного включення пропущено)

```
namespace tmcli::database::command
{

DeleteAllTasksDbCommand::
DeleteAllTasksDbCommand(connection::ISqlConnection& connection)
	: connection_{ connection }
{}

DeleteAllTasksDbCommand
DeleteAllTasksDbCommand::create(connection::ISqlConnection& connection)
{
	return DeleteAllTasksDbCommand{ connection };
}

auto DeleteAllTasksDbCommand::
execute(const ArgsContainer& args) -> QueryResult
{
	connection_.query("DELETE FROM tasks;");
}

auto DeleteAllTasksDbCommand::
get_required_args() const -> const ArgsNamesIterable&
{
	return {};
}

auto DeleteAllTasksDbCommand::
get_optional_args() const -> const ArgsNamesIterable&
{
	return {};
}

}
```

Лістинг 47: Вміст новоствореного файлу DeleteAllTasksDbCommand.cpp (директиви #include пропущено)

2. Додаємо файл DeleteAllTasksDbCommand.hppdi:

```
namespace DI
{

DI_DYNAMIC_SUBSCRIBER_FOR(DeleteAllTasksDbCommand)
{
	using namespace tmcli::database::command;

	fast_di::global_di::GlobalDI::builder()
		.register_singleton<DeleteAllTasksDbCommand>()
			.as_interface<IDbCommand>()
			.with_tag("db_delete_all_tasks")
			.done();
}

}
```

Лістинг 48: Вміст новоствореного файлу DeleteAllTasksDbCommand.hppdi (директиви #include пропущено)

3. Додаємо конфігурації для декоратора, виділені жовтим кольором,в до файлу DeleteAllTasksDbCommand.hppdi:

```
namespace DI
{

DI_DYNAMIC_SUBSCRIBER_FOR(DeleteAllTasksDbCommand)
{
	using namespace tmcli::database::command;

	fast_di::global_di::GlobalDI::builder()
		.register_singleton<DeleteAllTasksDbCommand>()
			.as_interface<IDbCommand>()
			.with_tag("db_delete_all_tasks")
			.done()
		.register_transient<DBCommandArgsValidator>()
			.as_interface<IDbCommand>()
			.with_tag("validated_db_delete_all_tasks")
			.with_tag_of_dependency_at<0>("db_delete_all_tasks")
			.done();
}

}
```

Лістинг 49: Оновлений вміст файлу DeleteAllTasksDbCommand.hppdi (директиви #include пропущено)

4. Виконуємо збірку проєкту.

З прикладу випливає, що процедура додавання нової команди бази даних є ще простішою за процедуру додавання команди командного рядка. Проте, складність останньої зумовлено необхідністю впроваджувати команди командного рядка до динамічної фабрики команд, чого немає у випадку з командами бази даних.

Як видно з прикладу, недоліком використання взірця проектування «Декоратор» для здійснення валідації параметрів команд баз даних є необхідність разом з

реєстрацією команди додавати чергову реєстрацію «Декоратора», який буде декорувати створену команду за рядковим ідентифікатором. У випадку з командами командного рядка, не було необхідності додатково реєструвати жодних валідаторів параметрів, оскільки валідація параметрів (клас AArgsValidatingCliCommand) тісно зв'язана з кожною командою командного рядка. Тобто, валідація параметрів реєструється одночасно з командою. Однак, складність реєстрації «Декораторів» це не є проблемою архітектури, а є проблемою Контейнера Впровадження Залежностей. Такої складності можна було б уникнути, якби Контейнер Впровадження Залежностей автоматизовував взірець проектування «Декоратор» або, іншими словами, - підтримував можливість Перехоплення (описано в [2]).

Як висновок, слід обирати взірці проектування, враховуючи функціональність Контейнера Впровадження Залежностей, що використовується. Якщо Контейнер Впровадження Залежностей реалізує можливість Перехоплення, слід надавати перевагу взірцю проектування «Декоратору» у випадках, де необхідно обгорнути додатковим функціоналом класи, що вже існують. Взірцю проектування «Шаблонному методу» слід надавати перевагу виключно у випадках, коли контейнер впровадження залежностей не підтримує можливості Перехоплення та сильне зв'язування «Шаблонів» та їх наслідників є припустимим для задачі, яку виконує програма. Окрім цього, відсутність автоматизації взірців проектування «Фабричний Метод» та «Абстрактна Фабрика» Контейнером Впровадження Залежностей може спричиняти порушення паралельної розробки у деяких випадках.

ВИСНОВКИ

У роботі було досягнуто наступних результатів:

1. Досліджено методи Впровадження Залежностей, їх критерії (підтримуваність коду та пізнє зв'язування), переваги та недоліки, а також Контейнер Впровадження Залежностей як засіб його реалізації.
2. Описано та порівняно способи конфігурування розглянутих контейнерів: на основі конфігураційних файлів та на основі коду.
3. Специфікована та реалізована задача розробки Динамічного Контейнера Впровадження Залежностей, яка полягала у розробленні бібліотеки fast_di, що містить Динамічний Контейнер Впровадження Залежностей та засоби для його Глобальних Конфігурацій.
4. Проектування вилилось у гнучку архітектуру бібліотеки fast_di, пристосовану до ефективного розширення функціоналу.
5. Запропоновано новий спосіб Глобальної Конфігурації Контейнера Впровадження залежностей, що дозволяє досягти якнайкращої паралельної розробки проєкту, ґрунтованого на бібліотеці fast_di.
6. Розроблено застосунок Задачник (tmcli), що демонструє приріст паралельної розробки завдяки використанню бібліотеки fast_di.
7. На прикладі архітектури застосунку tmcli здійснено порівняння та аналіз взірців проектування «Декоратор» (Decorator) та «Шаблонний Метод» (Template Method).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. Vlissides., 1994.

2. S. van Deursen Dependency Injection Principles, Practices and Patterns / S. van Deursen, M. Seemann.

3. Alexandescu A. Modern C++ Design: Generic Programming and Design Patterns Applied / Andrei Alexandescu.

4. C++ reference [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/cpp.

5. CMake Documentation and Community [Електронний ресурс] – Режим доступу до ресурсу: https://cmake.org/documentation/.

6. Martin R. C. Design Principles and Design Patterns / Robert Cecil Martin., 2000.