Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Мультиметоди

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
Доцент
Бублик В. В.

_____

(Підпис)

"___" _____ 2021 року

Виконав студент КН-4
Кочмар В.В

"___" _____ 2021 року

Київ 2021

ЗМІСТ

Тема: Мультиметоди

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	Листопад 2020 р.	
2.	Огляд літератури та джерел за темою роботи.	Листопад-грудень 2020 р.	
3.	Ознайомлення з мультиметодами в C++	Грудень-січень 2020 р.	
3.	Ознайомлення з мультиметодами в Clojure	Січень-лютий 2021 р.	
4.	Написання роботи	Лютий-квітень 2021 р.	
5.	Створення слайдів для доповіді.	Квітень 2021 р.	
6.	Надання роботи керівнику на перевірку.	Квітень 2021 р.	

Студент _____
Керівник _____
“_____” _____

Анотація

У даній курсовій роботі розглядається та аналізуються механізм в мовах програмування, який називається мультиметоди – кратна диспетчеризація. Досліджується сучасний стан справ в диспетчеризації. Виконується порівняльний аналіз мультиметодів в різними мовами програмування – C++ та Clojure.

ВСТУП

Мультиметоди – механізм в мовах програмування, який дає можливість вибирати одну або декілька функцій потрібних до виклику в залежності від динамічних типів або декількох значень аргументів функції(в деяких мовах програмуваннях носить назву – перегрузка). Мультиметоди є шляхом до вирішення багатьох класичних проблем об'єктно-орієнтованого програмування. Існують інші назви-синоніми мультиметодів, такі як множинна диспетчеризація, множинне перемикання по типу, перевантаження функцій під час виконання.

Мультиметод або кратна диспетчеризація є варіацією концепціїї для вибору методу в ООП, що викликається під час виконання програми, а не під час компіляції. Іноді треба називати дві або більше функцій одним і тим самим іменем зазвичай через те, що вони виконують дуже схожі задачі, но працюють з різними типами параметрів або з різною кількістю параметрів. Саме в таких випадках недостатньо тільки назви функції в місці, де вона викликається для визначення викликаємого коду. В даному випадку до назви функції для вибора потрібної реалізації також використовують кількість аргументів та тип аргументів функції, яку викликають.

Мультиметоди змушують писати код, там де часто можна скористатись даними. Один з прикладів – сучасний графічний процесор уміє виконувати тільки одну операцію - ``намалювати трикутник``. У відеокарти немає функції ``намалювати собаку``, ``намалювати літак`` і т.д. Вона має тільки одну функції - ``малювати трикутник``, а хто це буде визначається тільки вхідними даними.

Не всі мови програмування підтримують мультиметоди. Наприклад C#, Common List підтримують мультиметоди. Існують мови, такі як C++, Java, які не підтримують динамічну диспетчеризацію, то в них вона має бути

реалізована вручну. Одними з можливих варіантів реалізації динамічної диспетчеризації, наприклад в C++, є використання шаблону проектування Відвідувач або динамічного приведення типу.

Метою даної курсової роботи є дослідження мультиметодів, розгляд проблем, які вирішуються за допомогою мудьтиметодів. Також в курсовій роботі реалізуються кратна диспетчеризація в мовах, які не підтримують її нативно.

Перший розділ курсової присвячено мультиметодам в C++. Йде визначення мультиметодів, розглядаються ситуації коли треба використовувати поліморфізм мультиоб’єктів. Описується за допомогою яких механізмів можно виконувати диспетчеризації виклику функції.

Другий розділ курсової присвячено мультиметодам в Clojure. Виконується порівняльний аналіз з C++.

Третій розділ - заключення й висновки, які були отримані внаслідок дослідження мультиметодів.

Розділ 1. Мультиметоди та C++

1.1 Загальна інформація

Кратна диспетчеризація(мультиметоди) – advanced техніка абстрагування, яка доступна в деяких мовах та реалізується програмістами в інших мовах.

В даному розділі мова йде про C++, отже розглядаються мультиметоди в C++. C++ не підримує мультиметодів напряму, але можуть бути реалізовані декількома способами.

Мультиметоди треба для диспетчеризації виклику функції в залежності від типів декількох об’єктів.

1.2 Поліморфізм, кратна диспетчеризация

У програмування існує багато видів поліморфізму. В мові програмування C++ поліморфізм означає, що виклик якоїсь даної функції пов’язаний з різними реалізаціями и залежить від типу контексту – статичного чи динамічного.

В мові C++ реалізовано два види поліморфізму:

- Compile-time polymorphism(статичний полізморфізм) – який виконується за допомогою перевантаження та шаблонних функції
- Run-time polymorphism(динамічний поліморфізм) – який реалізується за допомогою віртуальних функцій.

Тип поліморфізму про який зараз буде йтися – динамічний поліморфізм, в якому поведінка вибирається динамічно, в залежності від типів об’єктів під

час виконання програми. Саме кратна диспетчеризація - це і є про динамічні типи кількох об’єктів виконання.

Для поступового розуміння кратної диспетчеризації, спочатку мова піде про одноразову диспетчеризацію(single dispatch). Одноразова диспетчеризація полягає в тому, що ми маємо об’єкт та функцію, яку викликаємо. Функція, яка буде викликана під час виконання, залежить від типу середовища виконання об’єкту. В C++ для цього є віртуальні функції. Назва віртуальної функції зв’язується з конкретною реалізацією під час виконання програми, в залежності від динамічного типу об’єкту, до якого вона виконується.

Приклад одноразової диспетчеризації:

```
class Shape {
public:
  virtual void Compute() const = 0;
};

Class Rect: public Shape {
public:
  virtual void Compute () const {
    cout << "Rect ";
  }
};

class Ellip : public Shape {
public:
  virtual void Compute () const {
    cout << "Ellip";
  }
};

int main(int argc, const char** argv) {
  unique_ptr<Shape> pr(new Rect);
  unique_ptr<Shape> pe(new Ellip);
```

```
  pr->Compute();
  pe->Compute ();

  return 0;
}
```

pr та pe – є указниками на об’єкт Shape, два різних виклики Compute () диспетчеризуються до різних функцій під час виконання програми. Це все виконується за допомогою динамічного поліморфізму, реалізованого за допомогою віртуальних функцій.

Є очевидним те, що об’єкт який ми відправляємо є указником на Shape. В даному прикладі є указник pr і викликається метод Compute. Компілятор C++ видає код для цього виклику таким чином, що під час виконання програми викликається потрібна функція. Яку функцію потрібно викликати вирішується на дослідження об’єкту на який указує наш указник pr. Звідки і випливає одноразова диспетчиризація(single dispatch).

Доповненням цієї ідеї є – кратна диспетчеризація(multiple dispatch). При кратній диспетчеризації рішення яку функцію викликати основується на типах багатьох об’єктів. Показовим знаком того, що кратна диспетчеризація може мати місце на існування є те, що наприклад у нас операція, яка включає більше одного класу і не існує жодного класу, куди ця операція належить. Яскравим прикладом можу бути обчислення перетину фігур, яке може мати місце в комп’ютерній графіці. Обчислення перетину фігур різних форм може бути складним для реалізації. Наприклад обчислення перетинів прямокутників з прямокутниками тривіальним, але якщо маємо справу з колами та еліпсами, прямокутниками та трикутниками справи можуть бути

дещо складнішими. Для реалізації цього нам потрібна функція перетину - Overlap приймає дві форми та обчислює їх перетин.

Така функція має багато різних випадків для різних фігур, які можна легко розрахувати перш ніж вдасться до складнішого випадку – який матиме загальний підхід перетину багатокутників. Такий код функції перетину був би дуже громоздним та складно підтримуваним.

Було б краще як би у нас було:

```
void Overlap(const Rect* r, const Ellip* e) {}

void Overlap(const Rect* r1, const Rect* r2) {}

void Overlap(const Shape* s1, const Shape* s2) {}
```

І тоді виклик функції перетину – Overlap диспетчеризується до потрібної функції. Ця можливість і називається кратної диспетчеризацією(мультиметоди).

1.3 Кратна диспетчеризація в C++ з використанням шаблону Відвідувач

Даний підхід називають шаблоном відвідувач, хоча мабуть ближче підходить назва обернутого шаблону відвідувача.

В минулому розділі йшлось, що віртуальна диспетчеризація в C++ запускається тоді і лише тоді, коли віртуальний метод викликається вказівником на батьківський обєкт. Ідея підходу полягає в тому, щоб фукнція перетину OVerlap переходила через віртуальні відправлення своїх аргументів, щоб дійти до правильної функції для типів, які викликаються в функції.

Визначимо класс Shape:

```
class Shape {
```

```
public:
  virtual std::string name() const {
    return typeid(*this).name();
  }
  virtual void Overlap(const Shape*) const = 0;
  virtual void OverlapWith(const Shape*) const {}
  virtual void OverlapWith(const Rect*) const {}
  virtual void OverlapWith(const Ellip*) const {}
};
```

Метод Overlap - це метод, який користуються корустувачі. Щоб мати можливість використовувати віртуальну диспетчеризації, нам потрібне перетворити виклик функції з двума параметрами Overlap(A*, B*) в виклик функції виду – A -> Overlap(B).

Методи OVerlapWith – це методи конкретної реалізації перетину, на які буде диспетчеризуватись код і які обов’язково мають бути реалізовані для кожного підкласа для кожного можливого випадку перетину.

```
class Rect : public Shape {
public:
  virtual void Overlap(const Shape* s) const {
    s->OverlapWith(this);
  }

  virtual void OverlapWith(const Shape* s) const {
    cout << "Rect With Shape" << endl;
  }

  virtual void OverlapWith(const Rect* r) const {
    cout << "Rect With Rect
<< endl;
  }
};

class Ellip : public Shape {
public:
```

```
  virtual void Overlap(const Shape* s) const {
    s->OverlapWith(this);
  }

  virtual void OVerlapWith(const Rect* r) const {
    cout << "Ellip with Rect
<< endl;
  }
};
unique_ptr<Shape> pr1(new Rect);
unique_ptr<Shape> pr2(new Rect);
unique_ptr<Shape> pe(new Ellip);

pr1->Overlap(pe.get());
pr1->Overlap(pr2.get());
```

В виводі програми ми отримаємо:

```
Ellip With Rect
Rect With Rect
```

Хоть ми маємо справу лише з указниками на клас Shape, шуканий перетин обраховується вірно.

Pr1->Overlap(pe.get());

Pr1 – указник, що вказує на Shape, а метод Overlap- віртуальний метод.

Відповідно тип середовища виконання Overlap викликається тут, що є Rect::overlap. Параметр, який передається в метод – є іншим указником на Shape, який указує на Ellip(pe). Rect::overlap викликає s->OverlapWith(this). Компілятор C++ бачить, що s є указником на Shape те, що OverlapWith віртуальний метод так маєто ще одну віртуальну диспетчеризації у виконанні – викликається Ellip::OverlapWith.

Постає питання, яка з перегрузок методу викликається?

Знову повертаємось до Rect::Overlap.

```
virtual void Overlap(const Shape* s) const {
  s->OverlapWith(this);
}
```

s->OverlapWith викликається з this, про який компілятор C++ знає, що це є указник на Rect. В результаті виклик pr1->Overlap(pe.get()) перенаправляється на Ellip::OverlapWith(const Rect*), завдяки двом віртуальним диспетчеризаціям та перенавантаженню методів – результатом є подвійна диспетчеризація.

1.4 Симетрія і значення по замовченню для базового класу

Коли ми маємо справу з кратною диспетчеризацією в будь якій мові програмування, є дві важливі речі про які не потрібно забувати:

1. Чи має право симетрія? Іншими словами перефразувавши отримуємо це – чи існнує значення для порядку диспетчеризації об’єктів? Якщо ні, то скільки потрібно дописати додаткового коду для вираження цього?
2. Чи працює диспетчеризація базового класа за замовчення так як потрібно? Наприклад, ми створимо новий підклас Прямокутника, який називається Square і ми не будемо створювати метод для перетину OverlapWith для Еліпсу та квадрату. Коли ми попрогсимо перетин для Еліпсу та Квадрату викличається метод для перетину Квадрату та Еліпсу. Результат виконання є вірним, тому що це очікувано – з ієрархії класі в об’єктно-орієнтованому програмуванні

В рішенні вище, два даних аспекти є, хоча якщо дивитись вже так то для симетрії потрібно трішки додаткового коду. Для повноти нам треба реалізувати симетрію між перетинами прямокутника та еліпса.

Зробимо це наступним чином:

```
void SymmetricOverlapRectXEllip (const Rect* r, const Ellip* e) {
  std::cout << "OverlapRectWithEllip”
<< endl;
}
```

Даний код гарантує, що rect->Overlap(ellip) та ellip->Overlap(rect) опиняться в одній функції. Якщо симетрія є бажана то в даному підході необхідно трохи додаткового коду.

1.5 Проблеми в підході з шаблоном Відвідувач

Даний підхід забезпечує чистий клієнтський код і є досить ефективним, але є одна значуща проблема, якщо проглянути код – він є складнопідтримуваним і складно додавати новий функціонал. Нехай уявимо, що треба додати новий тип фігури. Також уявимо, що існує ефективний алгоритм перетину нової фігури наприклад з еліпсом. Було б добре, якби нам треба було дописати код лише для нової функціональності:

1. Створити новий клас, що є похідним від Shape;
2. Написати загальний алгоритм перетину нової фігури з Shape;
3. Реалізувати конкретний алгоритм перетину нової фігури з Ellip;

Но в реальності це не так – ми маємо змінити визначення базового класу Shape, щоб додати перевантаження OverlapWith для нової фігури. Якщо ми хочемо добитись симетричності меж перетином нової фігури та еліпсом, нам також доведеться змінити еліпс, щоб додати таке ж перенавантаження.

У нас з’являються проблеми, якщо ми не контролюємо базовий клас Shape. Дана проблема носить назву ‘’expression problem’’. Це проблема, яка дає можливість розробнику додавати в програму нові сутності та операції над ними без зміни реалізації яку вже маємо. Можу відзначити, що ця проблема не є легко вирішуємою в C++.

1.6 Кратна диспетчеризація в C++ прямим підходом за допомогою dynamic_cast

Підхід реалізований за допомогою паттерну Відвідувач – є досить складним – використовуються віртуальні функції для кратної диспетчеризації. Якщо повернутись на початок, існує очевидніше рішення пробеми, яка є набагато простіша за підхід на базі Відвідувача – метод грубою сили за допомогою умовних перевірок if-else.

```
class Shape {
public:
  virtual std::string name() const {
    return typeid(*this).name();
  }
};

class Rect : public Shape {};

class Ellip : public Shape {};

class Triangle : public Shape {};

void Overlap(const Shape* s1, const Shape* s2) {
  if (const Rect* r1 = dynamic_cast<const Rect*>(s1)) {
    if (const Rect* r2 = dynamic_cast<const Rect*>(s2)) {
      cout << "Rect With Rect
<< endl;
    } else if (const Ellip* e2 = dynamic_cast<const Ellip*>(s2)) {
      cout << "Rect with Ellip
<< endl;
```

```
    } else {
      cout << "Rect With Shape
<< endl;
    }
  } else if (const Ellip* e1 = dynamic_cast<const Ellipse*>(s1)) {
    if (const Ellip* e2 = dynamic_cast<const Ellipse*>(s2)) {
      cout << "Ellip with Ellip
<< endl;
    } else {
      // other with ellip
    }
  } else {
    // Triangle
  }
}
```

Відразу можна побачити, що вирішується проблема ‘’expression problem’’. Зараз у нас Overlap є окремою функцією, яка інкапсулює диспетчеризацію. Коли ми захочемо в майбутньому додати ще фігури, нам залишиться лише модифікувати функцію Overlap.

Помітним є те, що розмір коду збільшується дуже швидко від того, що з’являються нові умови if-else. Якщо у нас 30-40 фігур, то який великий розмір коду буде. Але це ще не все, функція перетину – це один алгоритм, а якщо треба ще якісь алгоритми операцій то розмір коду буде об’єм коду буде дуже великим.

Інша проблема, яка з’явилась – порядок умовних операторів if, щоб батьківский клас не закрив всі свої підкласи.

Насправді, писати весь цей код розробникам було б не дуже охоче, тому придумали різні автоматизатори для написання if-else ланцюжків.

Значимий знавець C++ Андрій Александреску присвятив цілий розділ своєї книги Сучасне проектування C++ цій проблемі. В розділі автор реалізовував різні види автоматичних рішень за допомогою шаблонів метапрограмування. Якщо проглянути бібліотеку шаблонів C++ Loki, розроблену Александреску і продивиться Multimethods.h то можна побачити його зі списками типів, шаблонами шаблонів. Loki варта уваги, якщо треба використовувати кратну диспетчеризацію.

1.7 Спроба стандартизації

Б. Страуструп спробував вирішити цю проблему. Ідея, яку використовував Страуструп було дозволити параметрам функції бути віртуальними, таким чином вони виконують динамічну диспетчеризацію, а не просто звичайне статичне перевантаження.

Можна було б тоді реалізувати перетину так(код не є реальним синтаксисмо c++):

```
void Overlap(virtual const Shape*, virtual const Shape*);

void Overlap(virtual const Rect*, virtual const Ellip*);
```

Все що ми додали – це virtual ключове слово для аргументів і диспетчеризація перетворюється зі статичної у віртуальну. Але ця пропозиція ніколи не потрапила у стандарт.

1.8 Висновок

В даному розділі розглянулась проблема кратної диспетчеризації. Описується для чого потрібна диспетчеризація, які види її є. Було розглянуто два варіанти рішення за допомогою мови програмування C++. Було проаналізовано проблеми, які виникають при кожному з них та можливі варіанти їх рішення. Кожне рішення має свої переваги та недоліки і вибір залежить тільки від потреб які треба в програмі. Перший підхід це на основі

шаблону Відвідувач, який несе за собою проблему ‘’expression problem’’, інший підхід – прямий підхід грубої сили який несе проблему громіздкого коду через if-else. Завершення розділу є опис спроби стандартизації Страустропом за допомогою віртуальних параметрів. Можна зробити висновок, що диспетчеризація дає змогу зробити вибір яку функцію потрібно викликати під час виконання програми, іноді даний інструмент може бути досить корисним, як наприклад в прикладі коду вище.

Розділ 2. Мультиметоди та Clojure

Clojure – відносно нова мова програмування, яка була створена в 2007 році Р.Хікі. Останнім часом Clojure набрала популярності як сучасна мова програмування що працює на Java Virtual Machine(JVM).

В даному розділі розглядається кратна диспетчеризація в Clojure.

2.1 Одноразова диспетчеризація на основі класу Clojure

Диспетчеризація в Clojure оголошується за допомогою ключового слова defmulti.

Defmulti оголошує мультимед і встановлює функцію диспетчеризації, яка є довільною в Clojure. Під час виконання, коли мультиметод викликається дана функція диспетчеризації викликається спочатку на аргументах, а значення яке повертає використовується, щоб обрати метод який потрібно викликати.

Приклад цього:

```
(defmulti d-thing class)

(defmethod d-thing java.lang.Int
  [th] (println "int" (str thing)))

(defmethod d-thing java.lang.String
  [th] (println "str" (str thing)))
```

В прикладі вище, оголосили мультиметод d-thing і дві реалзіації – одна для вбудованого типу Int інша для вбудованого типу String. Дана функція буде визначати який тип передано в функцію:

Виклик: Multi.core=> (d-thing 100500)

Результат: Int 100500

Виклик: Multi.core=> (d-thing “clojure”)

Результат: str clojure

Також в Clojure ми можемо визначати мультиметоди для кастомних типів:

```
(defrecord Worker [name phone])

(defmethod d-thing Worker
  [thing] (println(:name thing)))
```

В Clojure існує багато різних способів оголошення нових типів, в прикладі вище використовуємо defrecord, який чимось нагадує C++ структури.

2.2 Одноразова диспетчеризація на основі значення Clojure

У прикладах вище ми використовували клас як функцію диспетчеризації. У Clojure функцією диспетчеризації може бути довільна функція. Отже ми могли б використати будь яку функцію, наприклад диспетчеризація на основі значення:

```
(defmulti p-due :position)

(defmethod p-due :engineer
  [emp] (> (:lines-of-code emp) 100000))
```

```
(defmethod p-due :manager
  [emp] (> (:num-reports emp) 10))


(defrecord Employee [name position num-reports lines-of-code])
```

У прикладі вище можна побачити як switch на основі позиції делегується самому Clojure.

Давайте прогляно на код вище і де ж тут функція диспетчеризації? Функція диспетчеризаця – це :position. У Clojure зарезервовані слова(ключові), що стоять на першому місці в формі приймаються як функції що отримують доступ до ключа приймаються як функції які отримують доступ до ключа названим ключовим слово із значень які підходять цьому ключовому слову. Зараз перевіримо чи зможе наш promotion-due працювати з мапом:

```
=> (def joe {:name "Jack", :position :manager, :num-reports 8})
=> (promotion-due Jack)
false
=> (def tim {:name "Bob", :position :engineer, :lines-of-code 132000})
=> (promotion-due Bob)
True
```

Clojure мультиметоди не переймаються який клас значень вони диспетчеризують. Їм абсолютно неважливо чи це взагалі нестандартний клас. До тих пір поки функція диспетчеризації може бути успішно викликана і повертає те що може бути диспетчеризовано мультиметоди працюють. Це дуже класна форма качиної типзації – техніка в програмування, яка стала популярною в мові програмування Python, також є доступна в C++.

Насправді техніка програмування Clojure полягає в виконанні якомога більше качиних типізацій на вбудованих типах. Якщо сутність можна представити у вигляді комбінацій з векторів та мепів, то це означає, що окрім

користуваць функцій, які пишуться для виконання дій з цими типами ми можемо користуватись всіма іншими функціями які працють на вбудованих типах. На думку творців Clojure ідея полягає, що замість багатьох класів з декількома методами використовується невелика кількість структурданих з великою кількістю оізних функцій, що працюють з цими структурами даних.

Записи Clojure мають подібні до мепів поведінку, яка полегшує змішувати методи , що працюють з ними з методами що працюють зі звичайними мепами.

Невеличкий приклад коду використовує :position як функцію диспетчеризації. Альтернативний метод для досягнення цього:

```
(defmulti p-due
  (fn [emp]
    (:position emp)))
```

Попередня форма є кращою, за рахунок того що вона є меньшою. Але цей варіант демонструє використання будь якої функції у вигляді диспетчера. Наприклад, він може повернути пари значень і потім ми напишемо метод для диспетчеризації на них.

2.3 Кратна диспетчеризація Clojure

Методи Clojure приймають декілька аргументів(так само як і звичайні функції). Всі аргументи передаються в функцію диспетчеризації, яка потім може повернути кортеж.

```
(deftype Shape [])
(deftype Rect [])
(deftype Ellip [])
(deftype Triangle [])

(defmulti overlap
  (fn [a b]
    [(class a) (class b)]))
```

```
(defmethod overlap [Rect Ellip]
  [r e] (printf "Rect With Ellip "
                (class r) (class e)))

(defmethod overlap [Rect Rect]
  [r1 r2] (printf "Rect With Rect "
                  (class r1) (class r2)))

(defmethod overlap [Rect Shape]
  [r s] (printf "Rect With Shape "
                (class r) (class s)))
```

Функція диспетчеризації бере обидва аргументи і повертає вектор їх класів. Потім ми можем диспетчеризувати цей вектор щоб обрати потрбіний метод для перетину фігур:

```
=> (overlap (Rect.) (Ellip.))
Rect With Ellip
nil
=> (overlap (Rect.) (Rect.))
Rect With Rect
nil
=> (overlap (Rect.) (Shape.))
Rect With Shape
```

А тепер спробуємо викликати базовий клас. У нас немає методу для перетину трикутника прямокутником, тому логічно, що тут у нас мав би бути варіант [Rectangle Shape]:

```
=> (Overlap (Rect.) (Triangle.))


IllegalArgumentException No method in multimethod 'overlap' for dispatch value: [...
]
```

Отримуємо помилку, тому що в нашому коді не описано, що трикутник унаслідується від Shape. Deftype в Clojure не дозволяє вказувати базові класи. Потрібно зрозуміти як працює наслідування в Clojure.

2.4 Наслідування в Clojure

Наслідування в Clojure є трошки специфічним. Його називають спеціаьним(ad-hoc inheritance). Загалом наслідування в мовах програмування визначається між класами. У Clojure наслідування відокремлене від конкретного механізму класу – так би мовити успадкування живе саме собою. У Clojure наслідування визначається між абстрактними тегами, які відповідають простору імен або є ключомими словами.

Приклад визначимо ієрархію тварин:

```
multi.core=> (derive ::dog ::mammal)
nil
multi.core=> (derive ::cat ::mammal)
nil
multi.core=> (derive ::husky ::dog)
nil
```

isa? Використовується щоб відповісти на зв’язок між об’єктами:

```
multi.core=> (isa? ::dog ::mammal)
true
multi.core=> (isa? ::husky ::dog)
true
multi.core=> (isa? ::husky ::cat)
false
```

Мультиметоди в Clojure використовують isa? Для диспетчеризації. Коли функція диспетчеризації викликається та повертає результат щоб знайти метод який слід викликати для очікуваного значення кожного методу.

Isa? A b завжди повертає істину якщо = a b тому співпадіння працюють як треба.

2.5 Кратна диспетчеризація в Clojure з батьківскими класами

Тепер ми можемо зробити перетин між прямокутником та трикутником:

```
(derive ::rect ::shape)
(derive ::ellip ::shape)
(derive ::triangle ::shape)
(defn create-shape [] {:kind ::shape})
(defn create-rect [] {:kind ::rect})
(defn create-ellip [] {:kind ::ellip})
(defn create-triangle [] {:kind ::triangle})
```

Форми представлені мепами. Ключове слово :kind відповідає за тип фігури.

Диспетчеризація перетину фігур трішки змінилась:

```
(defmulti overlap
  (fn [a b]
    [(:kind a) (:kind b)]))

(defmethod overlap [::rect ::ellip]
  [r e] (printf "Rect With Ellip"))

(defmethod overlap [::rect ::rect]
  [r1 r2] (printf "Rect With Rect"))

(defmethod overlap [::rect ::shape]
  [r s] (printf "Rect Witj Shape"))
```

Виконаємо перетини:

```
=> (overlap (create-rect) (create-ellip))
Rect With Ellip
=> (overlap (create-rect) (create-rect))
Rect With Rect
=> (overlap (create-rect) (create-shape))
```

```
Rect With Shape
=> (overlap (create-rect) (create-triangle))
Rect With Shape
```

Тепер перетин Прямокутника та трикутника працює як і треба, вибравши варіант rect :: shape, тому що трикутник походить від Shape. Коли викликається функція перетину трикутника та прямокутника повертається значення функції диспетчеризації ::rect ::triangle, який передає isa? На перевірку ::rect ::shape.

2.6 Гнучкість диспетчеризації в Clojure

Механізм диспетчеризації в Clojure є загальним, тому що функцією диспетчеризації може бути довільна функція. Функція диспетчеризації не обмежена вона може робити все що завгодно, що стосується Clojure.

Через таку гнучкість з’являються проблеми. Така гнучкість дає можливість писати запутаний та незрозумілий код. Частіше за все рекомендується використовувати диспетчеризацію на основі isa.

Друге – це продуктивність. Кожен виклик методу має пройти через механізм диспетчеризації спочатку, тому краще ці шляхи добре оптимізувати. Вікладання цього на час виконання ускладнює оптимізацію компілятором. Розробники Clojure мали це занепокоєння, коли представляли протоколи – такий собі Java-подібний віртуальний механізм диспетчеризації який є швидшим, але який несе в собі деякі обмеження в порівнянні зі стандартними мультиметодами.

2.7 Спеціальне(Ad-hoc) наслідування

Спосіб визначення ієрархій в Clojure є трошки дивним в порівнянні з більшостями мов програмування. Ми не можемо просто позначити базові класи у визначенні типів, як в інших мовах. Ми повинні надати надати набір оголошень паралельно з власне визначеними типами. Даний підхід може бути схильним до помилок, але за наявності тестування він не може привести до глибоких помилок.

Якщо думати то вибір все таки може впасти на Clojure. На відміну від інших мов програмування, дана мова не є зосередженою навколо об’єктів. Clojure віддає перевагу роботі з мепами та композиціями вбудованих типів даних, оскільки тоді всі можливості стандартної бібліотеки Clojure застосовуються до даних, тому нам прийдеться писати меньше коду. Отже відокремлені інструменти визначення ієрархій є цілком розумним підходом. У Clojure можемо визначати нові типи за допомогою мепів, deftype, defrecord.

Висновок такий, що у Clojure незвичний спосіб, але він узгоджується з загальною філософією мови і спрямовує думки розробників на вирішення задач Clojure ніж на пошук знайомих рішень з іншими об’єктно-орієнтованих мовами.

2.8 Висновок

У цьому розділі було розглянуто мультиметоди у такій мові програмування, як Clojure, які вбудовані в неї, на відміну від C++. Було розглянуто приклад одноразової диспетчеризації. Наступним була кратна диспетчеризація і зіштовхування з наслідуванням в Clojure та виявлення проблеми пов’язаної з deftype. Окрему увагу приділено гнучкості мультиметодам в Clojure та основним проблемам, що ця гнучкість несе за собою.

Розділ 3. Порівняння вбудованих мультиметодів та емульованих

В першому розділі розглядалась емуляція мультиметодів в C++ двома методами – за допомогою шаблону Відвідувач та методом грубої сили – за допомогою оператор dynamic_cast. В другому розділі розглядалась мова програмування Clojure, яка має вбудовані мультиметоди. В обох випадках код мав одну і ту ж саму ідею – перетин геометричних фігур.

Порівнюючи два варіанти мульметодів я дійшов такого висновку, що використання вбудованих мультиметодів є більш прозорішим та зрозумілішим під час написання коду. Код виходить меншим, та не таким нагромодженим. Також не треба вирішувати який з можливих методі емуляції мультиметодів краще використовувать.

На прикладі C++, побачили що створення абстракцій високого рівня в C++ це ще те завдання, оскільки ця мова є статично типізованою. Абстракції також мають бути дешевшими з точки зору продуктивності і використання пам’яті, що також ускладнює задачу. А от на прикладі мови програмування Clojure – яка є більш динамічною і структурно-гнучкою ця проблема вирішується дешевше та продуктивніше. Але для C++ можна скористатись бібліотеками для мултиметодів, одною з якою є Loki розробником якої є А.Александреску.

Виходячи з цих характеристик, я дійшов висновку, що все ж таки вбудовані мультиметоди є продуктивніші, економніші та більш зрозумілі для розробника під час написання коду та додавання нового функціоналу.

Висновки

Отже можна дійти таких висновків. Під час курсової роботи було розглянуто такий механізм в програмуванні як мультиметоди(кратна диспетчеризація). Мультиметоди – це вибір функції, яку треба викликати базуючись на динамічному типу двох або більше параметрів. Мультиметоди корисні, коли в програмі є операція яка маніпулює декількома поліморфними об’єктами за допомогою указників чи відсилок на їх батьківський клас та необхідно модифікувати цю операцію відповідно з динамічними типами цих об'єктів. Для даної роботи я обрав дві мови для дослідження мультиметодів – Clojure з вбудованими мультиметодами та C++ без вбудованих мультиметодів, але з можливістю емуляції мультиметодів. В якості прикладу було обрано приклад, яким займався Б.Страуструп – перетин геометричних фігур.

Під час реалізації мультиметодів на C++ я використав два можливі варіанти реалізації – перший за допомогою шаблону Відвідувач, а другий – методом грубої сили. Оба варіанти мають свої недоліки про які було згадано в розділі, але порівнюючи їх - варіант на основі шаблону відвідувач більш прозоріший та зрозуміліший при великому розмірі коду, хоча й мають бути зміни в декількох місцях при додаванні нових сутностей. Було описано спробу, яка так і не стала реальністю, стандартизації Страуструпом, за допомогою додавання ключового слова virtual до параметрів функції.

В другому розділі, я розглянув мультиметоди, які є вбудованими в мові програмування Clojure. Було розглянуто теж приклад перетину геометричних фігур. Під час написання коду, зіштовнувся з проблемою, яка була пов’язана з наслідування в Clojure, тому було звернуто особливу увагу наслідуванню в Clojure. Після розбору наслідування було реалізовано кратну диспетчеризацію з батьківским класом. На даному прикладі, я дійшов думки, що мультиметоди в Clojure є дуже гнучкими, за рахунок того, що функцією диспетчеризації може бути довільна й зрозумів, що така гнучкість теж

накладує деякі проблеми у вигляді запутаного коду, тому не варто використовувати диспетчеризацію на основі значення, а краще за все, якщо є можливість використовувати диспетчеризацію на основі isa?.

В третьому розділі курсової роботи піводиться аналітичний підсумок порівняння мультиметодів в C++ та Clojure. В якому дійшов висновку, що все таки вбудовані мультиметоди зручніше для розробника та є більш продуктивнішими по часу виконанню та використаної пам’яті.

Список джерел

1. Сучасне проектування C++ - A. Александреску
2. Effective Modern C++ - S. Meyers
3. Clojure documentation [Електронний ресурс]: https://clojure.org/reference/multimethods
4. Open-std [Електронний ресурс]: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html
5. Open Multi-Methods for C++ - B. Stroustroup [Електронний ресурс]: https://www.stroustrup.com/multimethods.pdf