Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №5

Технології розроблення програмного забезпечення

«*Патерни проектування. Паттерн «Adapter»*»

«10. VCS all-in-one»

Виконав
студент групи ІА–32:
Варивода К. С.

Перевірив:

Мягкий Михайло Юрійович

Київ 2025

Зміст

Варіант

10. VCS all-in-one (iterator, adapter, factory method, facade, visitor, p2p)

Клієнт для всіх систем контролю версій повинен підтримувати основні команди і дії (commit, update, push, pull, fetch, list, log, patch, branch, merge, tag) для 3-х основних систем управління версіями (svn, git, mercurial), а також мати можливість вести реєстр репозиторіїв (і їх типів) і відображати дерева фіксації графічно.

Теоретичні відомості

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях. Крім того, патерн проєктування обов’язково має загальновживане найменування. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову.

Шаблон "Adapter" використовується для адаптації інтерфейсу одного об'єкту до іншого. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

Проблема: Ви реалізовуєте аудіо-плеєр, який може програвати аудіо різних форматів. Ви почали аналізувати і бачите що для програвання аудіо із різних форматів краще використовувати різні компоненти.
Таким чином ви реалізуєте складну логіку роботи з різними компонентами. У вас у коді з'являється багато логіки з перевіркою, якщо формат один, то викликаємо такий метод у такого компонента, якщо інший, то викликаємо декілька методів у іншого компонента, і т.д. Логіка роботи з компонентами стає досить складною та заплутаною.

Коли потрібно добавити підтримку нового аудіо-формату, то потрібно добавити роботу з новим компонентом і при цьому внести зміну в уже існуючу логіку, що може привести до помилок там де все коректно працювало.

Рішення: Для вирішення цих проблем можна використати патерн Адаптер. Визначаємо загальний інтерфейс IPlayer для програвання музики через компоненти. Далі для кожного компонента робимо свій адаптер. Адаптер має посилання на об'єкт компонента та реалізує інтерфейс викликаючи методи з компонента, який він адаптує. Таким чином, адаптер ніби обгортає специфічний компонент і далі весь клієнтський код буде працювати з адаптерами через інтерфейс IPlayer. Класи, які будуть працювати з адаптером через цей інтерфейс не будуть знати інтерфейси кожного специфічного компонента.

При такій реалізації, якщо буде потрібно добавити підтримку нового формату, то просто створюється новий адаптер, який обгортає новий компонент, але робота з цим адаптером буде виконуватися через існуючий інтерфейс. При цьому існуючий код не буде зазнавати змін, а значить і не "поламаємо" те що вже працює.

Переваги та недоліки:

+ Відокремлює інтерфейс або код перетворення даних від основної бізнес-логіки.

+ Можна добавляти нові адаптери не змінюючи код у класі Client.

- Умовним недоліком можна назвати збільшення кількості класів, але за рахунок використання патерна Адаптер програмний код, як привило, стає легше читати.

Релевантність до проекту:

Так як розробляється система, яка має уніфікувати 3 системи контролю версій це є чи не основним патерном для рішення цієї проблеми, так як я розробляю єдиний інтерфейс, який виконує список дій, які потім «під капотом» виконуються для кожної системи своєю логікою.

Хід роботи

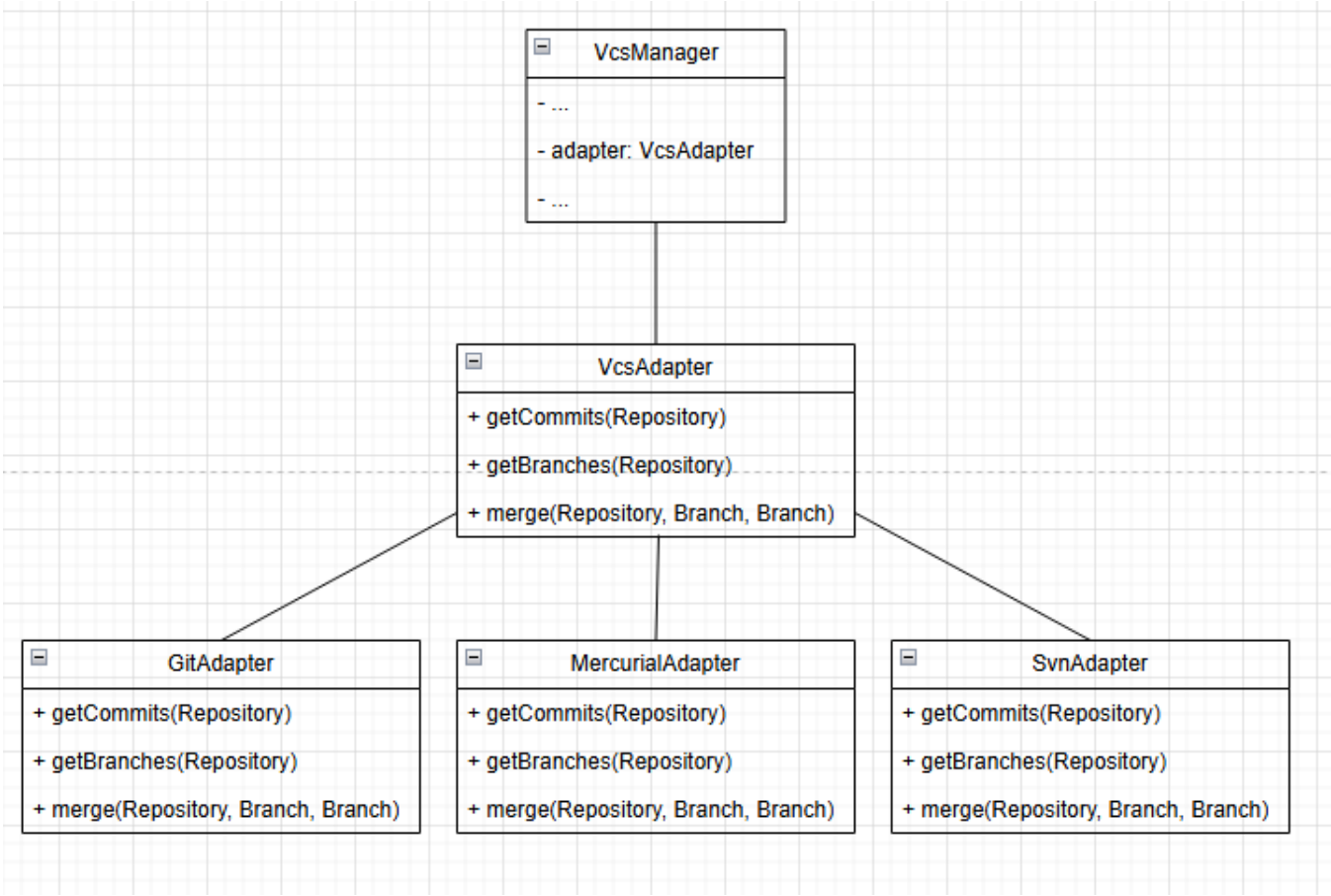


Рис 1 – Діаграма класів модулю з використанням паттерну «Адаптер»

```
1   package vcs.repository.adapters;
2
3   import vcs.repository.classes.Repository;
4   import vcs.repository.classes.Branch;
5   import vcs.repository.classes.Commit;
6   import vcs.repository.classes.MergeResult;
7
8   import java.util.List;
9
10  public interface VcsAdapter {  8 usages  3 implementations  mavr1iq
11      List<Commit> getCommits(Repository repo);  1 usage  3 implementations  mavr1iq
12      List<Branch> getBranches(Repository repo);  no usages  3 implementations  mavr1iq
13      MergeResult merge(Repository repo, Branch source, Branch target);  1 usage  3 implementations  mavr1iq
14  }
15
```

Рис 2 – інтерфейс VcsAdapter

```java
package vcs.repository.adapters;

import vcs.repository.classes.Repository;
import vcs.repository.classes.Branch;
import vcs.repository.classes.Commit;
import vcs.repository.classes.MergeResult;

import java.util.List;

public class SvnAdapter implements VcsAdapter {
    @Override
    public List<Commit> getCommits(Repository repo) {

        return List.of();
    }

    @Override
    public List<Branch> getBranches(Repository repo) {

        return List.of();
    }

    @Override
    public MergeResult merge(Repository repo, Branch source, Branch target) {

        return new MergeResult( success: true, List.of());
    }
}

```

Рис 3 – імплементація SvnAdapter

```java
package vcs.repository.adapters;

import vcs.repository.classes.Repository;
import vcs.repository.classes.Branch;
import vcs.repository.classes.Commit;
import vcs.repository.classes.MergeResult;

import java.util.List;

public class MercurialAdapter implements VcsAdapter {
    @Override
    public List<Commit> getCommits(Repository repo) {

        return List.of();
    }

    @Override
    public List<Branch> getBranches(Repository repo) {

        return List.of();
    }

    @Override
    public MergeResult merge(Repository repo, Branch source, Branch target) {

        return new MergeResult( success: true, List.of());
    }
}

```

Рис 4 - імплементація MercurialAdapter

```java
package vcs.repository.adapters;

import vcs.repository.classes.Repository;
import vcs.repository.classes.Branch;
import vcs.repository.classes.Commit;
import vcs.repository.classes.MergeResult;

import java.util.List;

public class GitAdapter implements VcsAdapter {  2 usages  mavr1iq
    @Override  1 usage  mavr1iq
    public List<Commit> getCommits(Repository repo) {

        return List.of();
    }

    @Override  no usages  mavr1iq
    public List<Branch> getBranches(Repository repo) {

        return List.of();
    }

    @Override  1 usage  mavr1iq
    public MergeResult merge(Repository repo, Branch source, Branch target) {

        return new MergeResult( success: true, List.of());
    }
}

```

Рис 5 - імплементація GitAdapter

Висновок

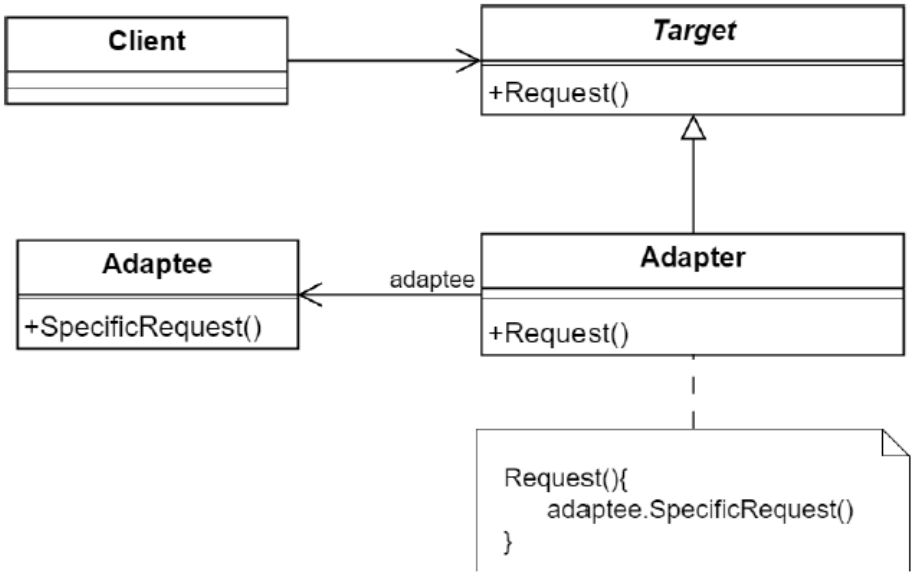
Отже, після виконання даної лабораторної роботи було вивчено паттерн програмування «Адаптер», його сильні та слабкі сторони, тонкощі імплементації та use кейси. Для закріплення отриманих знань було розроблено паттерн у існуючому проекті. Було проведено аналіз потреб проекту та визначено потребу в імплементації вказаного патерну.

Відповіді на питання

1. Яке призначення шаблону «Адаптер»?

Призначення шаблону **«Адаптер» (Adapter)** — перетворити інтерфейс одного класу на інший, очікуваний клієнтом. Він діє як "перекладач" або "перехідник" (наприклад, як адаптер живлення, що дозволяє підключити американську вилку до європейської розетки), дозволяючи об'єктам з **несумісними інтерфейсами** працювати разом.

2. Нарисуйте структуру шаблону «Адаптер».



3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

- **Client (Клієнт):** Клас, який потребує певного інтерфейсу (Target), щоб виконати свою роботу.
- **Target (Цільовий інтерфейс):** Інтерфейс, який очікує та використовує Client.
- **Adaptee (Той, що адаптується):** Існуючий клас, який має несумісний інтерфейс. Це клас, який *вже є* і який ми не можемо (або не хочемо) змінювати.
- **Adapter (Адаптер):** Клас, який реалізує Target. Він "загортає" (містить посилання на) об'єкт Adaptee.

Взаємодія: Client викликає метод у Adapter (думаючи, що це Target). Adapter перехоплює цей виклик і **делегує** його (перенаправляє) у потрібному форматі об'єкту Adaptee.

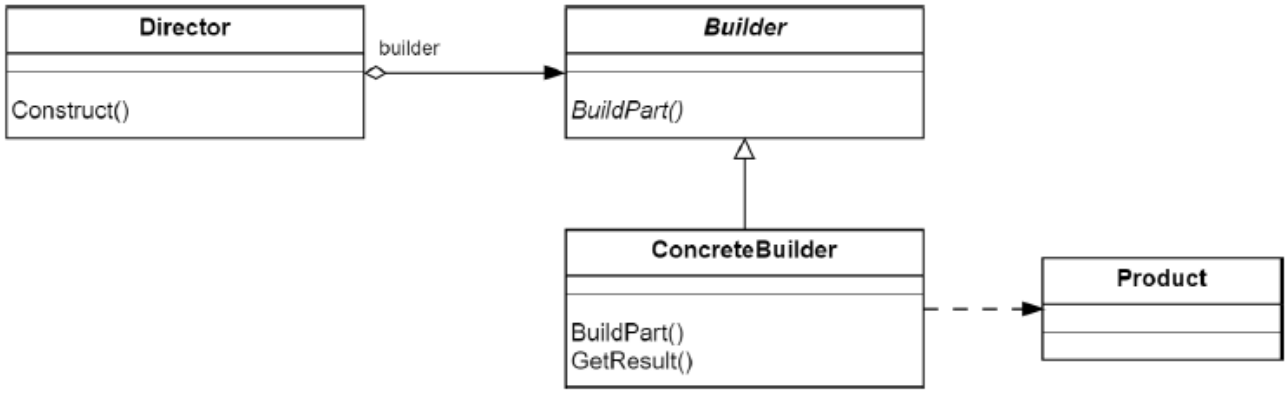
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

- **Адаптер об'єктів (Використовує композицію):** Адаптер реалізує цільовий інтерфейс (Target) і *містить екземпляр* (Adaptee). Це найпоширеніший підхід в Java, оскільки він гнучкий і дозволяє адаптувати навіть фінальні класи.
- **Адаптер класів (Використовує спадкування):** Адаптер одночасно успадковує Adaptee (клас) та реалізує Target (інтерфейс). Цей підхід менш гнучкий (оскільки Java не підтримує множинне спадкування класів) і "жорстко" прив'язує Адаптер до конкретного Adaptee.

5. Яке призначення шаблону «Будівельник»?

Призначення шаблону **«Будівельник» (Builder)** — відокремити процес **покрокового створення** складного об'єкта від його **кінцевого представлення**. Це дозволяє використовувати один і той же процес конструювання для створення різних версій об'єкта.

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

- **Product (Продукт):** Складний об'єкт, який потрібно створити.
- **Builder (Будівельник):** Абстрактний інтерфейс, що визначає кроки для створення частин Product (наприклад, buildPartA(), buildPartB()).
- **ConcreteBuilder (Конкретний Будівельник):** Клас, що реалізує Builder. Він знає, *як* створювати та збирати частини, і містить метод для отримання готового Product.
- **Director (Директор):** Клас, який *керує* процесом конструювання. Він викликає кроки (методи) у Builder у правильній послідовності.

Взаємодія: Клієнт створює Director та ConcreteBuilder. Клієнт передає Builder у Director. Director викликає методи на Builder (напр., buildPartA()), не знаючи, яку конкретну версію він створює. Коли все готово, Клієнт забирає Product у Builder.

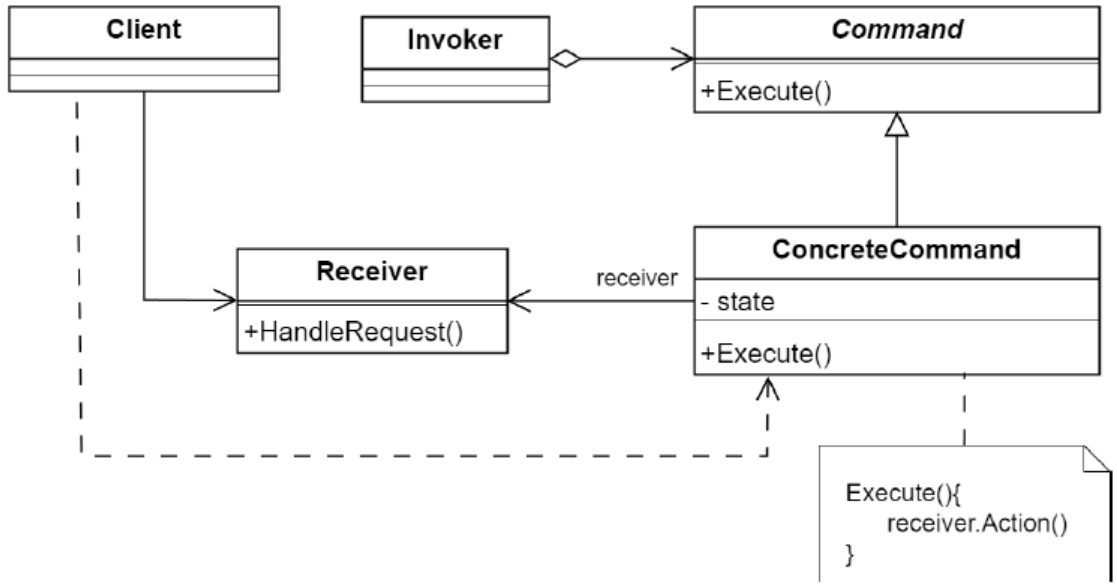
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

1. **"Телескопічний конструктор":** Коли клас має конструктор з величезною кількістю параметрів, багато з яких є необов'язковими.
2. **Складне створення:** Коли процес створення об'єкта складається з багатьох кроків або вимагає певної послідовності дій.
3. **Різні представлення:** Коли потрібно створити різні версії одного об'єкта (наприклад, "автомобіль з двигуном V8" та "автомобіль з електродвигуном"), використовуючи той самий процес побудови.

9. Яке призначення шаблону «Команда»?

Призначення шаблону **«Команда» (Command) — інкапсулювати запит (дію) як об'єкт**. Це дозволяє передавати дії як параметри, ставити їх у чергу, логувати, а також реалізовувати операції "скасувати" (Undo) та "повторити" (Redo).

10. Нарисуйте структуру шаблону «Команда».



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

- **Command (Команда):** Інтерфейс, що зазвичай оголошує один метод (напр., execute()) і, можливо, undo().
- **ConcreteCommand (Конкретна Команда):** Клас, що реалізує Command. Він зберігає посилання на Receiver і знає, який його метод потрібно викликати.
- **Receiver (Отримувач):** "Справжній" об'єкт, який виконує бізнес-логіку (наприклад, TextEditor, Light).

- **Invoker (Той, що викликає):** Клас, який *ініціює* виконання команди (наприклад, Button, MenuItem). Він не знає нічого про Receiver, він просто викликає command.execute().

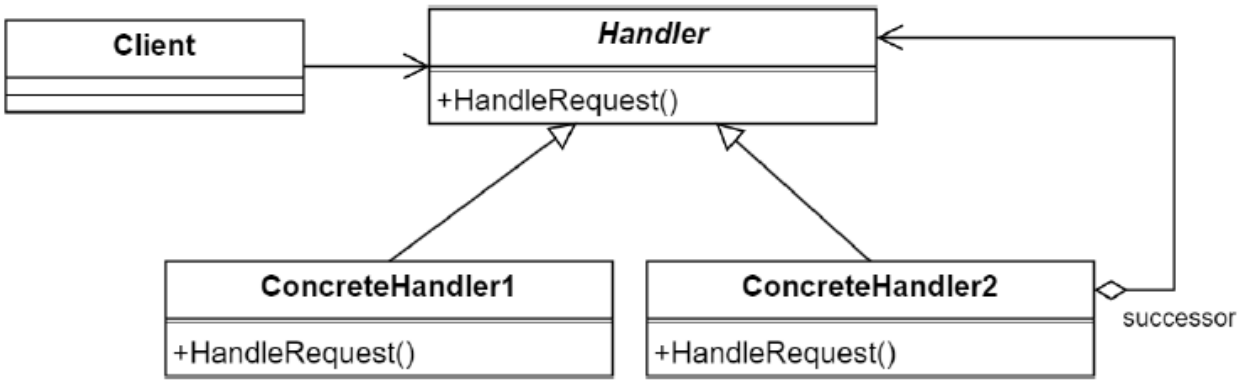
12. Розкажіть як працює шаблон «Команда».

1. **Зв'язування:** Клієнт створює об'єкт ConcreteCommand і передає йому об'єкт Receiver (напр., SaveCommand отримує TextEditor).
2. **Налаштування:** Клієнт передає цей ConcreteCommand об'єкту Invoker (напр., кнопці SaveButton).
3. **Виклик:** Коли відбувається подія (натискання кнопки), Invoker (SaveButton) не викликає TextEditor напряму. Він просто викликає метод execute() у об'єкта Command, який у нього є.
4. **Делегація:** ConcreteCommand (SaveCommand) отримує цей виклик і, у свою чергу, викликає потрібний метод у свого Receiver (напр., textEditor.save()).

13. Яке призначення шаблону «Прототип»?

Призначення шаблону **«Прототип» (Prototype)** — дозволити створювати нові об'єкти шляхом **клонування** (копіювання) існуючого об'єкта ("прототипу"). Це дозволяє уникнути витрат на створення об'єкта "з нуля" через конструктор, особливо якщо ініціалізація є складною.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

- **Prototype (Прототип):** Інтерфейс, що оголошує метод clone() (або аналогічний).
- **ConcretePrototype (Конкретний Прототип):** Клас, що реалізує інтерфейс Prototype. Він реалізує логіку копіювання самого себе.

Взаємодія: Клієнт має екземпляр ConcretePrototype. Коли клієнту потрібен *новий* об'єкт, який є точною копією першого (або дещо зміненою копією), він викликає newObject = prototype.clone().

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

- **Обробка HTTP-запитів (Middleware):** Найпоширеніший приклад. HTTP-запит послідовно проходить через ланцюжок обробників: 1. Перевірка автентифікації, 2. Перевірка прав доступу (авторизація), 3. Кешування, 4. Основний контролер, що виконує логіку.
- **Системи логування:** Повідомлення (log message) передається ланцюжком. Перший обробник (рівень DEBUG) може його пропустити, другий (INFO) теж, а третій (ERROR) обробляє і записує.
- **Системи техпідтримки:** Запит клієнта (тікет) спочатку потрапляє до оператора 1-го рівня. Якщо він не може вирішити проблему, він передає запит "далі" по ланцюжку — оператору 2-го рівня, потім інженеру тощо.