



Міністерство освіти і науки України  
Національний технічний університет України  
“Київський політехнічний інститут імені Ігоря Сікорського”  
Факультет інформатики та обчислювальної техніки  
Кафедра інформаційних систем та технологій

Лабораторна робота № 5  
**Технології розроблення програмного забезпечення**  
«Патерни проектування.»  
**Тема: IRC-Клієнт**

Виконала:  
студентка групи ІА-32  
Красоха В.О.

Перевірив:  
Мягкий М.Ю.

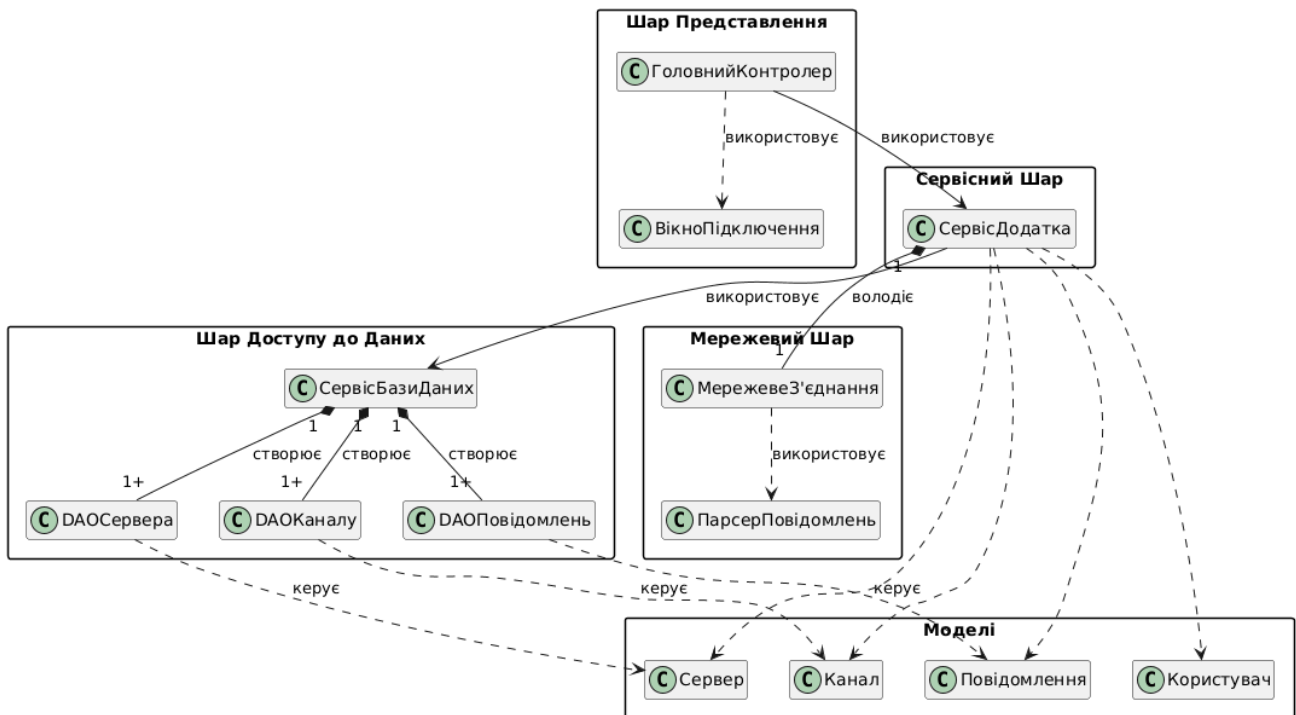
Київ 2025

**Тема:** Патерни проектування.

**Мета:** Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.

### Хід роботи

#### Діаграма класів:



#### Шаблон Command (Команда)

Це основний патерн, на якому базується взаємодія користувача з IRC-сервером. На діаграмі послідовностей та діаграмі прецедентів чітко видно блок «Надіслати команду». Коли користувач вводить у чат текст, що починається з / (наприклад, /join, /list, /part), MainController не виконує ці дії сам. Він передає «команду» до IRCService, який інкапсулює параметри запиту та відправляє їх на сервер. Це використовується для відокремлення відправника запиту (користувача) від виконавця (IRC-сервера).

#### Шаблон Prototype (Прототип)

Цей шаблон використовується у моделях даних проєкту. Цей шаблон знаходиться у структурі класів моделей, таких як Message або Server. Коли

отримується повідомлення з бази даних або мережі, часто потрібно створити копію об'єкта для відображення в різних вікнах чату або для пересилання. Замість створення нового об'єкта "з нуля", використовується метод `clone()` (інтерфейс `Cloneable` у Java), що дозволяє копіювати існуючий екземпляр класу `Message`. Це використовується для економії ресурсів при створенні складних об'єктів шляхом їх копіювання.

### Шаблон Chain of Responsibility (Ланцюжок відповідальності)

Він реалізований у логіці обробки (парсингу) вхідних рядків від сервера. Знаходиться у компоненті `MessageParser` (на діаграмі класів він позначений у мережевому шарі). Коли від сервера приходить рядок тексту, система пропускає його через ряд обробників (або умов `if-else`). Кожен обробник перевіряє: "Це приватне повідомлення? Це список каналів? Це помилка доступу?". Якщо обробник впізнає команду, він її обробляє, якщо ні — передає далі за ланцюжком. Це використовується для уникнення жорсткої прив'язки відправника запиту до його отримувача.

### Доданий код в проєкті:

#### **IRCCConnection.java:**

```
package org.ircclient.network;

import java.io.*;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.function.Consumer;

public class IRCCConnection {
    private Socket socket;
    private BufferedReader reader;
    private BufferedWriter writer;
    private ExecutorService executorService;
    private boolean connected = false;
    private Consumer<IRCMessage> messageHandler;

    public void connect(String host, int port) throws IOException {
        if (connected) {
            disconnect();
        }

        socket = new Socket(host, port);
        reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        connected = true;
    }
}
```

```
    executorService = Executors.newSingleThreadExecutor();
    executorService.submit(this::readLoop);
}

public void disconnect() {
    connected = false;
    try {
        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    if (executorService != null) {
        executorService.shutdown();
    }
}

public void sendCommand(String command) throws IOException {
    if (!connected || writer == null) {
        throw new IOException("Не підключено до сервера");
    }

    writer.write(command + "\r\n");
    writer.flush();
}

public void sendNick(String nickname) throws IOException {
    sendCommand("NICK " + nickname);
}

public void sendUser(String username, String realName) throws IOException {
    sendCommand("USER " + username + " 0 * :" + realName);
}

public void sendJoin(String channel) throws IOException {
    if (!channel.startsWith("#")) {
        channel = "#" + channel;
    }
    sendCommand("JOIN " + channel);
}

public void sendPart(String channel) throws IOException {
    sendCommand("PART " + channel);
}

public void sendPrivmsg(String target, String message) throws IOException {
    sendCommand("PRIVMSG " + target + " :" + message);
}

public void sendWhois(String nickname) throws IOException {
    sendCommand("WHOIS " + nickname);
}

public void sendList() throws IOException {
    sendCommand("LIST");
}

public void sendTopic(String channel, String topic) throws IOException {
    if (topic != null && !topic.isEmpty()) {
        sendCommand("TOPIC " + channel + " :" + topic);
    }
}
```

```

    } else {
        sendCommand("TOPIC " + channel);
    }
}

public void sendPong(String server) throws IOException {
    sendCommand("PONG :" + server);
}

private void readLoop() {
    try {
        String line;
        while (connected && (line = reader.readLine()) != null) {
            IRCMessage message = IRCMessageParser.parse(line);
            if (message != null && messageHandler != null) {
                messageHandler.accept(message);
            }
        }
    } catch (IOException e) {
        if (connected) {
            e.printStackTrace();
        }
    } finally {
        connected = false;
    }
}

public void setMessageHandler(Consumer<IRCMessage> handler) {
    this.messageHandler = handler;
}

public boolean isConnected() {
    return connected && socket != null && !socket.isClosed();
}
}

```

## IRCMessage.java:

```

package org.ircclient.network;

public class IRCMessage {
    private final String prefix;
    private final String command;
    private final String[] params;
    private final String rawLine;

    public IRCMessage(String prefix, String command, String[] params, String rawLine) {
        this.prefix = prefix;
        this.command = command;
        this.params = params != null ? params : new String[0];
        this.rawLine = rawLine;
    }

    public String getPrefix() {
        return prefix;
    }

    public String getCommand() {
        return command;
    }

    public String getParam(int index) {
        return index >= 0 && index < params.length ? params[index] : null;
    }
}

```

```

}
public String getTrailing() {
    return params.length > 0 ? params[params.length - 1] : null;
}

@Override
public String toString() {
    return rawLine;
}
}

```

## IRCMessageParser.java:

```

package org.ircclient.network;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class IRCMessageParser {
    private static final Pattern MESSAGE_PATTERN = Pattern.compile(
        "^(?::([^\ ]+ ))?([A-Z0-9]+)(?: (.*))?$"
    );

    public static IRCMessage parse(String line) {
        if (line == null || line.trim().isEmpty()) {
            return null;
        }

        Matcher matcher = MESSAGE_PATTERN.matcher(line.trim());
        if (!matcher.matches()) {
            return new IRCMessage(null, "UNKNOWN", new String[]{line}, line);
        }

        String prefix = matcher.group(1);
        String command = matcher.group(2);
        String params = matcher.group(3);

        String[] paramArray = parseParams(params);

        return new IRCMessage(prefix, command, paramArray, line);
    }

    private static String[] parseParams(String params) {
        if (params == null || params.isEmpty()) {
            return new String[0];
        }

        List<String> paramList = new ArrayList<>();
        int i = 0;
        StringBuilder current = new StringBuilder();

        while (i < params.length()) {
            char c = params.charAt(i);

            if (c == ' ' && current.length() > 0) {
                paramList.add(current.toString());
                current = new StringBuilder();
            } else if (c == ':' && i > 0 && params.charAt(i - 1) == ' ') {

```

```

        // Trailing параметр - все що залишилось
        current.append(params.substring(i + 1));
        break;
    } else if (c != ' ') {
        current.append(c);
    }

    i++;
}

if (current.length() > 0) {
    paramList.add(current.toString());
}

return paramList.toArray(new String[0]);
}

/**
 * Формує IRC команду для відправки
 */
public static String formatCommand(String command, String... params) {
    StringBuilder sb = new StringBuilder(command);

    if (params != null && params.length > 0) {
        for (int i = 0; i < params.length; i++) {
            sb.append(" ");
            if (i == params.length - 1 && (params[i].contains(" ") || params[i].startsWith(":"))) {
                sb.append(":").append(params[i]);
            } else {
                sb.append(params[i]);
            }
        }
    }

    return sb.toString();
}

/**
 * Витягує нікнейм з prefix (формат: nickname!username@hostname)
 */
public static String extractNickname(String prefix) {
    if (prefix == null) {
        return null;
    }
    int exclamationIndex = prefix.indexOf('!');
    return exclamationIndex > 0 ? prefix.substring(0, exclamationIndex) : prefix;
}
}

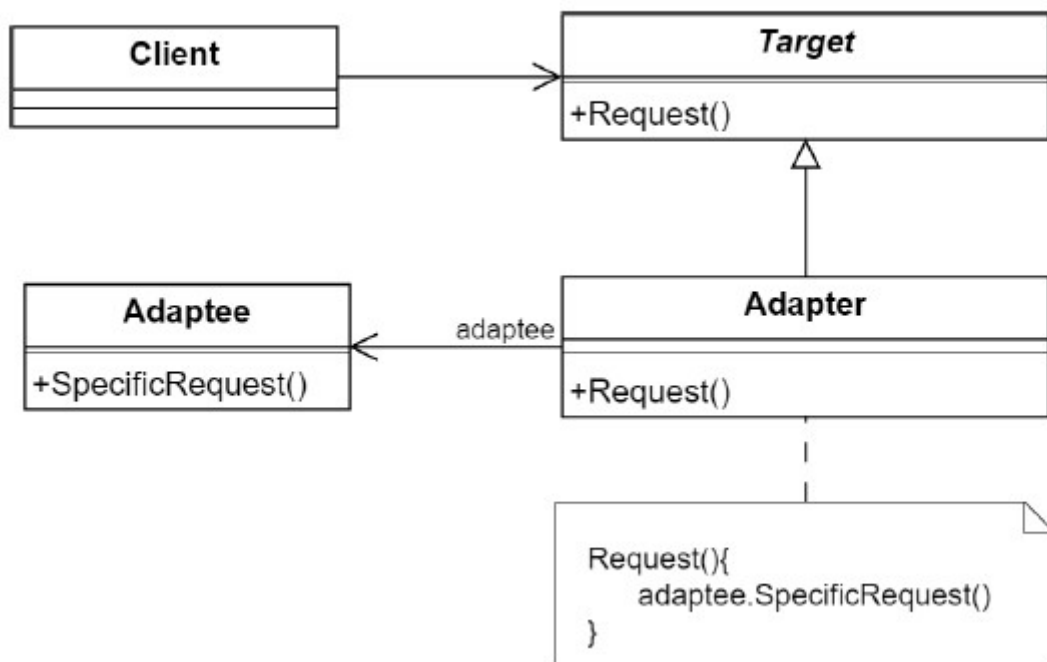
```

### Контрольні питання:

#### 1. Яке призначення шаблону «Адаптер»?

Шаблон Adapter (Адаптер) дозволяє підлаштувати інтерфейс одного класу під інтерфейс, який очікує клієнт, забезпечуючи сумісність несумісних класів.

#### 2. Нарисуйте структуру шаблону «Адаптер».



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

- **Target** — інтерфейс, очікуваний клієнтом
  - **Adapter** — клас, який реалізує **Target** і делегує виклики до **Adaptee**
  - **Adaptee** — клас з несумісним інтерфейсом, що потрібно адаптувати
- Клієнт звертається до **Adapter** через **Target**, а **Adapter** перетворює виклики під **Adaptee**.

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

- **Адаптер об'єктів:** **Adapter** містить посилання на **Adaptee** і делегує йому виклики (композиція).

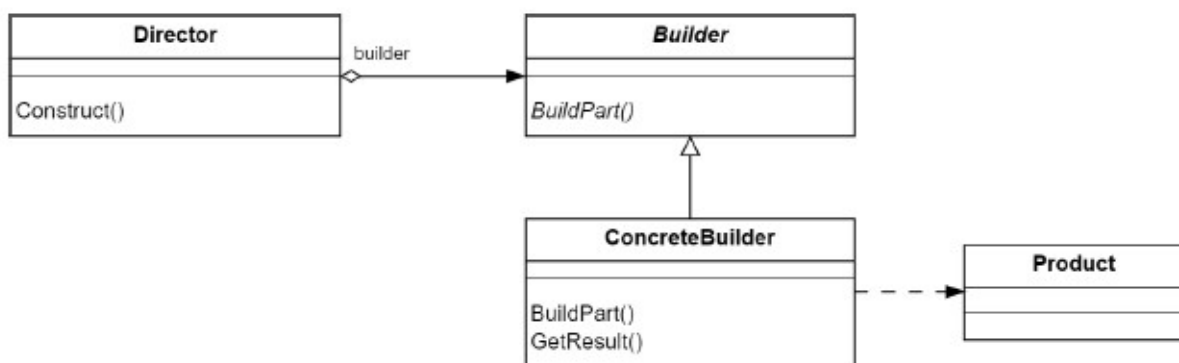
- Адаптер класів: Adapter успадковує Adaptee і реалізує Target (успадкування).

Об'єктний підхід більш гнучкий, а класовий — не дозволяє змінювати Adaptee без модифікації Adapter.

## 5. Яке призначення шаблону «Будівельник»?

Шаблон Builder (Будівельник) призначений для поступової побудови складного об'єкта, відокремлюючи процес конструювання від представлення.

## 6. Нарисуйте структуру шаблону «Будівельник».



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

- Builder — інтерфейс для побудови частин об'єкта
- ConcreteBuilder — конкретна реалізація Builder, яка створює частини об'єкта
- Director — керує побудовою об'єкта через Builder
- Product — складний об'єкт, що створюється  
Director викликає методи Builder для побудови частин, ConcreteBuilder збирає Product.

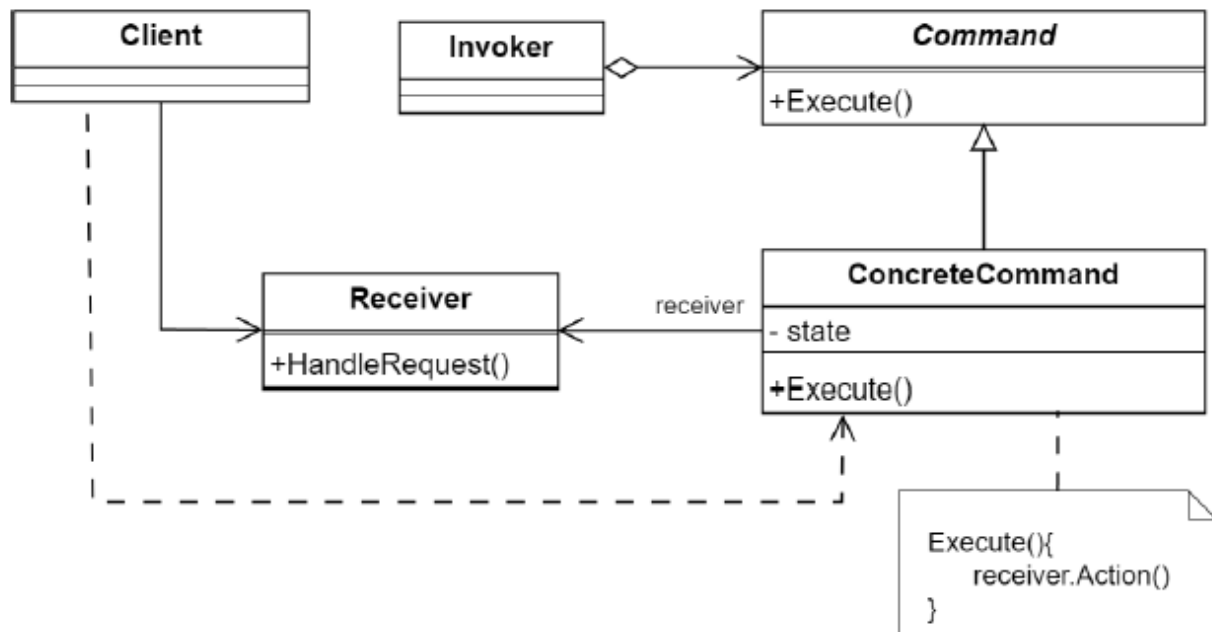
## 8. У яких випадках варто застосовувати шаблон «Будівельник»?

- Коли об'єкт має багато частин або конфігурацій
- Коли треба розділити процес створення і представлення об'єкта
- Для уникнення надмірних конструкторів із великою кількістю параметрів

## 9. Яке призначення шаблону «Команда»?

Шаблон Command (Команда) дозволяє інкапсулювати запит у вигляді об'єкта, що дозволяє відкладати виконання, зберігати історію або підтримувати відкат операцій.

## 10. Нарисуйте структуру шаблону «Команда».



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

- **Command** — інтерфейс з методом `execute()`
- **ConcreteCommand** — реалізація **Command**, делегує виконання **Receiver**
- **Receiver** — об'єкт, який виконує дію
- **Invoker** — викликає команду  
**Invoker** зберігає посилання на **Command** і виконує `execute()`, **ConcreteCommand** делегує виконання **Receiver**.

## 12. Розкажіть як працює шаблон «Команда».

1. Клієнт створює **ConcreteCommand**, прив'язуючи його до **Receiver**
2. **Invoker** отримує команду і викликає `execute()`

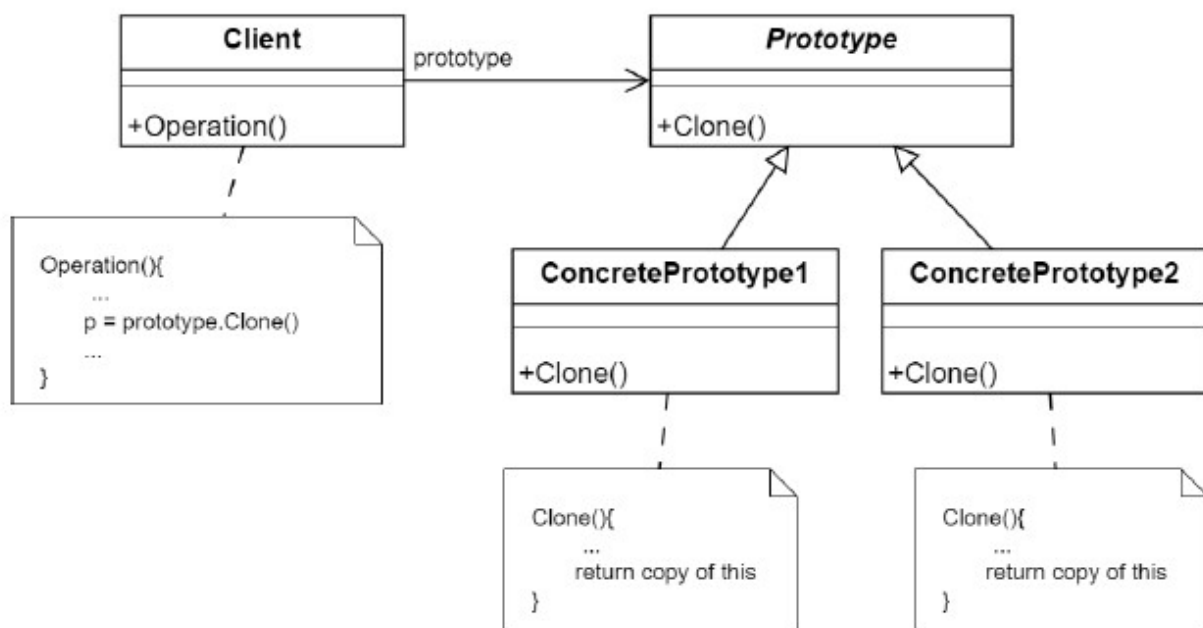
3. ConcreteCommand виконує методи Receiver

Це дозволяє відкладати виконання, чергувати команди, підтримувати Undo/Redo.

### 13. Яке призначення шаблону «Прототип»?

Шаблон Prototype (Прототип) дозволяє створювати нові об'єкти копіюванням існуючого об'єкта, замість створення з нуля, що спрощує створення складних об'єктів.

### 14. Нарисуйте структуру шаблону «Прототип».



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

- **Prototype** — інтерфейс з методом `clone()`
  - **ConcretePrototype** — конкретна реалізація, яка створює копію себе
- Клієнт викликає `clone()` на **ConcretePrototype** для отримання нового об'єкта.

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

- Обробка запитів у графічному інтерфейсі (клацання миші, події клавіатури)
- Логування та обробка повідомлень через кілька рівнів сервісів
- Перевірка доступу або валідація запитів у ланцюжку обробників
- Серія проміжних обчислень, де кожен об'єкт вирішує, обробляти запит чи передати далі

**Висновок:** Під час виконання лабораторної роботи я вивчила структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчилася застосовувати їх в реалізації програмної системи.