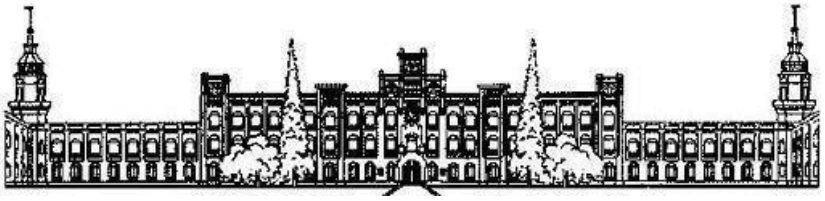


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №5

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування.»

Виконав:
студент групи ІА-32
Діденко Я.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.

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Теоретичні Відомості

Поняття та структура шаблонів «Adapter», «Builder», «Command», «Chain of Responsibility», «Prototype»

Шаблони проєктування (патерни) є типовими архітектурними рішеннями, що описують перевірені способи взаємодії об’єктів у системі. Вони дозволяють створювати гнучкі, розширювані й підтримувані програми, повторно використовуючи готові структури.

Adapter (Адаптер) - структурний шаблон, який забезпечує сумісність двох класів із різними інтерфейсами. Він «обгортає» один об’єкт іншим, щоб приховати невідповідність методів. Наприклад, адаптер може перетворювати формат даних або інтерфейс зовнішньої бібліотеки під потреби системи.

Builder (Будівельник) - породжувальний шаблон, що розділяє процес створення складного об’єкта на етапи, дозволяючи поетапно формувати різні представлення одного й того ж об’єкта. Це спрощує побудову об’єктів із багатьма параметрами.

Command (Команда) - поведінковий шаблон, який інкапсулює запит як окремий об’єкт. Це дозволяє зберігати, ставити в чергу або скасовувати операції. Команда має три основні ролі: відправник (ініціює дію), отримувач (виконує дію) і сама команда (описує запит).

Chain of Responsibility (Ланцюг відповідальностей) - поведінковий шаблон, що дозволяє передавати запит уздовж ланцюга обробників, поки один із них не зможе його обробити. Це зменшує зв’язність компонентів і спрощує додавання нових обробників без зміни існуючого коду.

Prototype (Прототип) - породжувальний шаблон, який створює нові об’єкти шляхом копіювання вже існуючих. Замість створення об’єкта через конструктор, система викликає метод `clone()`, що повертає копію вихідного екземпляра. Це корисно, коли створення об’єктів є дорогим або складним процесом.

Шаблон «Prototype»

Призначення: Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу.

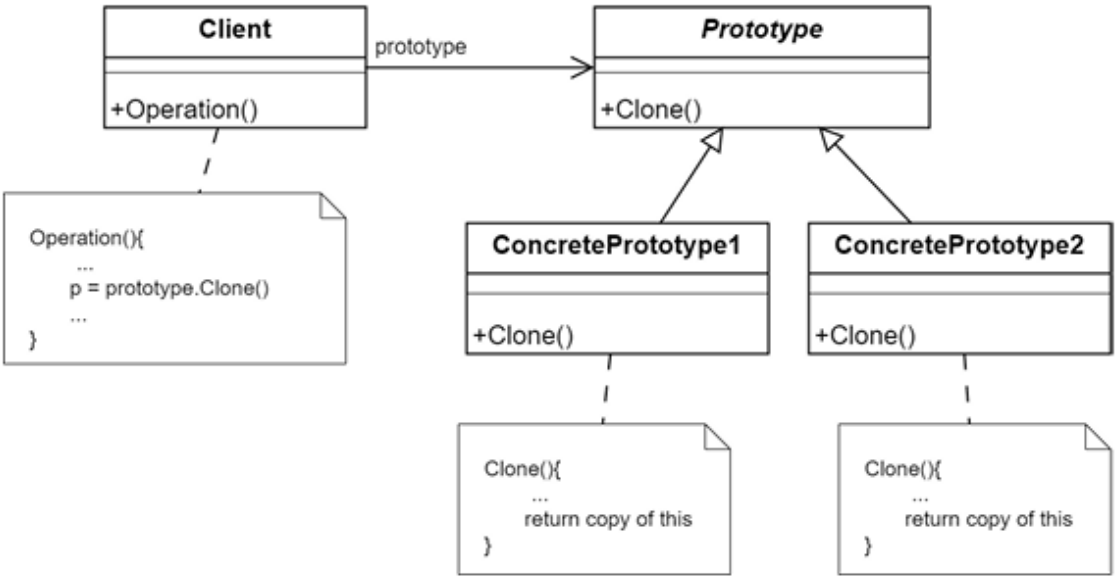


рис 1. Структура патерну Prototype

Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту - створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт. Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних прототипів; значно зменшується ієрархія наслідування (оскільки в іншому випадку це були б не прототипи, а вкладені класи, що наслідують).

Проблема: Ви розробляєте редактор рівнів для 2D гри на основі спрайтів. В панелі інструментів ви маєте багато кнопок для різних елементів, які можна розташовувати на екрані, такі як сходи, стіни, підлога, оздоблення та інші. Ці елементи у вас об'єднані в ієрархію з базовим класом GameObject. Під кожен елемент можна зробити свій тип кнопки, але тоді ми отримаємо паралельну ієрархію кнопок і при додаванні нового типу ігрового об’єкту потрібно буде додавати і новий тип кнопки.

Рішення: Використовуючи патерн прототип, додаємо до базового об'єкта GameObject метод Clone(), а кнопки будуть зберігати посилання на об'єкт базового типу GameObject. При натисканні на кнопку об'єкт який потрібно добавити на ігровому полі отримуємо не створенням нового, а клонуванням прототипу, який прив'язаний до кнопки. Таким чином, коли ми будемо додавати 66 нові типи ігрових об'єктів, то логіка роботи з кнопками не буде змінюватися, тому що не має прив'язки до конкретних типів.

Також слід відзначити, що при копіюванні об'єкту, навіть приватні поля будуть скопійовані, тому що реалізація методу копіювання знаходиться в цьому класі, що копіюється і таким чином є доступ для копіювання і до відкритих і до закритих полів.

Переваги:
+ За рахунок клонування складних об’єктів замість їх створення, підвищується продуктивність.
+ Різні варіації об’єктів можна отримувати за рахунок клонування, а не розширення ієрархії класів.
+ Вища гнучкість, тому що клоновані об’єкти можна модифікувати незалежно, не впливаючи на об’єкт з якого була зроблена копія.

Недоліки:
- Реалізація глибокого клонування досить проблематична, коли об’єкт, що клонується, містить складну внутрішню структуру та посилання на інші об’єкти.
- Надмірне використання патерну Прототип може привести до ускладнення коду та проблем із супроводом такого коду.

Хід роботи

1. Загальний опис виконаної роботи

В рамках даної лабораторної роботи було реалізовано шаблон проектування **«Прототип» (Prototype)**.

Основна мета полягала в тому, щоб створити гнучку систему створення об'єктів шляхом копіювання (клонування) вже існуючих екземплярів, які

називаються прототипами. Такий підхід дозволяє уникнути прямої залежності клієнтського коду від класів об'єктів, що створюються, та значно спрощує процес ініціалізації складних, багаторівневих структур.

Програмна система складається з наступних ключових компонентів:

- **Інтерфейс прототипу (FileSystemObject та Cloneable):** Абстрактний клас FileSystemObject, що реалізує стандартний інтерфейс Cloneable. Він оголошує метод clone(), який повинні реалізувати всі конкретні класи-прототипи, та визначає загальну поведінку для всіх об'єктів файлової системи.
- **Конкретні прототипи (File, Directory, TextReport):** Класи, що розширюють FileSystemObject та реалізують логіку клонування. Додавання нового класу TextReport демонструє гнучкість архітектури, яка дозволяє легко розширювати систему новими типами об'єктів.
    - File та TextReport реалізують **поверхневе копіювання**, оскільки є простими об'єктами без вкладених посилань на інші об'єкти.
    - Directory реалізує **глибоке копіювання**, рекурсивно клонуючи всі дочірні елементи. Це гарантує, що копія теки та вся її внутрішня структура є повністю незалежною від оригіналу.
- **Реєстр Прототипів (FileManager):** Клас, який відповідає за керування прототипами. Він зберігає набір шаблонів (прототипів) і на запит клієнта створює нові об'єкти, клонуючи відповідний шаблон.
- **Клієнт (Shell):** Кінцевий клієнт, який ініціює створення об'єктів. Він не створює об'єкти напряму, а делегує це завдання FileManager, демонструючи, як клієнтський код залишається незалежним від конкретних класів прототипів.

Процес створення нових об'єктів делегується самим об'єктам-прототипам. Замість того, щоб конструювати новий об'єкт з нуля, клієнт просить FileManager клонувати вже існуючий шаблон, що робить систему гнучкою та легко розширюваною.

2. Опис класів програмної системи.

Клас FileSystemObject.java

Цей абстрактний клас є **базовим прототипом** і служить спільною основою для всіх об'єктів у віртуальній файловій системі (File, Directory, TextReport). Він реалізує маркерний інтерфейс Cloneable та метод clone(), що є центральним елементом шаблону «Прототип».

Ключові обов'язки класу:

- **Інкапсуляція спільного стану:** Визначає private поля name та path, що є спільними для всіх об'єктів файлової системи. Доступ до них надається через публічні get та set методи, що відповідає принципам інкапсуляції.
- **Оголошення спільного інтерфейсу:** Визначає набір методів, які повинні мати всі нащадки. Метод display(int depth) є абстрактним, що змушує дочірні класи надавати власну реалізацію для візуалізації.
- **Надання базової функціональності клонування:** Його реалізація методу clone() викликає super.clone(), забезпечуючи механізм **поверхневого копіювання**, який є достатнім для простих нащадків і служить відправною точкою для реалізації глибокого копіювання у складних.

```java
public class Shell {

    public static void main(String[] args) {
        try {
            FileManager manager = new FileManager();

            TextReport reportTemplate = new TextReport(
                    name: "ReportTemplate.txt",
                    path: "/templates",
                    author: "Admin",
                    body: "Щоденний звіт про роботу системи..."
            );
            manager.registerTemplate( key: "textReport", reportTemplate);

            Directory projectTemplate = new Directory( name: "ProjectTemplate",  path: "/templates");

            Directory srcDir = new Directory( name: "src",  path: "/templates/ProjectTemplate/src");
            srcDir.add(new File( name: "Main.java",  path: "/templates/ProjectTemplate/src",  sizeKb: 120));
            srcDir.add(new File( name: "Utils.java",  path: "/templates/ProjectTemplate/src",  sizeKb: 80));

            Directory docsDir = new Directory( name: "docs",  path: "/templates/ProjectTemplate/docs");
            docsDir.add(new TextReport(
                    name: "ProjectReport.txt",
                    path: "/templates/ProjectTemplate/docs",
                    author: "TeamLead",
                    body: "Опис функціональності проєкту..."
            ));

            projectTemplate.add(srcDir);
            projectTemplate.add(docsDir);

            manager.registerTemplate( key: "projectDir", projectTemplate);
```

```java


            FileSystemObject dailyReport = manager.createFromTemplate(
                    key: "textReport",
                    newPath "/reports",
                    newName: "DailyReport_25_10.txt"
            );

            FileSystemObject myProject = manager.createFromTemplate(
                    key: "projectDir",
                    newPath "/projects",
                    newName: "MyAwesomeApp"
            );

            System.out.println("=== Демонстрація патерну Prototype ===\n");

            System.out.println("1. Оригінальний шаблон звіту:");
            reportTemplate.display( depth: 0);

            System.out.println("\n2. Новий звіт, створений шляхом клонування:");
            dailyReport.display( depth: 0);

            System.out.println("\n3. Новий проєкт, створений шляхом глибокого клонування:");
            myProject.display( depth: 0);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

рис 2.1 - код класу FileSystemObject.java

Клас File.java

Цей клас є **конкретним прототипом**, що представляє простий, атомарний об'єкт - файл. Він успадковує FileSystemObject і додає специфічну для файлу властивість sizeKb.

Ключові обов'язки класу:

- **Розширення базового стану:** Додає приватне поле sizeKb для зберігання розміру файлу в кілобайтах.
- **Конкретна реалізація поведінки:** Перевизначає метод display(int depth) для відображення інформації про файл, включаючи його ім'я та розмір.
- **Реалізація поверхневого клонування:** Клас успадковує реалізацію методу clone() від батьківського класу. Оскільки File не містить посилань на інші об'єкти, стандартного **поверхневого копіювання** цілком достатньо для створення повної та незалежної копії.

```
1   public class File extends FileSystemObject {  3 usages
2
3       private int sizeKb;  4 usages
4
5       public File(String name, String path, int sizeKb) {  3 usages
6           super(name, path);
7           this.sizeKb = sizeKb;
8       }
9
10      public int getSizeKb() {  no usages
11          return sizeKb;
12      }
13
14      public void setSizeKb(int sizeKb) {  no usages
15          this.sizeKb = sizeKb;
16      }
17
18      @Override  4 usages
19      public void display(int depth) {
20          String indent = " ".repeat(depth);
21          System.out.println(indent + getName() + " (" + sizeKb + " KB)");
22      }
23
24  }
```

рис 2.2 - код класу File.java

Клас Directory.java

Цей клас є **конкретним прототипом**, що представляє складний, композитний об'єкт - теку. Він реалізований за принципом **імутабельності (незмінності)**, що є ключовою архітектурною особливістю. Стан об'єкта Directory неможливо змінити після його створення, що гарантує безпеку та передбачуваність, особливо при роботі з прототипами.

Ключові обов'язки класу:

- **Імутабельне зберігання дочірніх елементів:** Має приватне final поле children для зберігання вкладених об'єктів. Будь-які операції "модифікації" не змінюють цей список, а створюють новий.
- **Створення нових станів:** Метод add() не змінює поточний об'єкт, а натомість створює і **повертає новий екземпляр Directory**, який містить всі попередні елементи плюс новий. Це повністю виключає ризик випадкової зміни зареєстрованого прототипу.
- **Рекурсивна візуалізація:** Перевизначає метод display(), який рекурсивно викликає себе для кожного дочірнього елемента, створюючи ієрархічний вивід.
- **Реалізація глибокого клонування:** Метод clone() виконує **глибоке копіювання**, створюючи повну та незалежну копію теки та всієї її внутрішньої структури. Клонований об'єкт також є імутабельним.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Directory extends FileSystemObject {  12 usages

    private final List<FileSystemObject> children;  6 usages

    public Directory(String name, String path) {  3 usages
        super(name, path);
        this.children = Collections.emptyList();
    }

    private Directory(String name, String path, List<FileSystemObject> children) {  2 usages
        super(name, path);
        this.children = Collections.unmodifiableList(children);
    }

    public Directory add(FileSystemObject child) {
        List<FileSystemObject> newChildren = new ArrayList<>(this.children);
        newChildren.add(child);
        return new Directory(this.getName(), this.getPath(), newChildren);
    }

    public List<FileSystemObject> getChildren() {  no usages
        return children;
    }

    @Override
    public Directory clone() throws CloneNotSupportedException {
        Directory cloned = (Directory) super.clone();


        List<FileSystemObject> clonedChildren = new ArrayList<>();
        for (FileSystemObject child : this.children) {
            clonedChildren.add(child.clone());
        }

        return new Directory(cloned.getName(), cloned.getPath(), clonedChildren);
    }

    @Override  4 usages
    public void display(int depth) {
        String indent = " ".repeat(depth);
        System.out.println(indent + "[DIR] " + getName());
        for (FileSystemObject child : children) {
            child.display( depth: depth + 2);
        }
    }
}
```

рис 2.3 - код інтерфейсу Directory.java

Клас FileManager.java

Цей клас реалізує патерн **Реєстр Прототипів (Prototype Registry)** і виступає в ролі централізованого менеджера шаблонів. Він приховує від клієнта (Shell) логіку вибору та клонування конкретних прототипів, надаючи простий та зручний інтерфейс для створення об'єктів за ключем.

Ключові обов'язки класу:

- **Зберігання прототипів:** Використовує Map для зберігання набору попередньо створених та налаштованих об'єктів-прототипів, асоціюючи кожен з них з унікальним рядковим ключем (наприклад, "textReport", "projectDir").
- **Реєстрація прототипів:** Надає публічний метод registerTemplate(), який дозволяє клієнту додавати нові шаблони до реєстру під час виконання програми.
- **Фабрика об'єктів:** Метод createFromTemplate() діє як фабричний метод. Він знаходить потрібний прототип за ключем, викликає його метод clone() для створення копії, а потім налаштовує властивості (name, path) нового об'єкта відповідно до переданих параметрів. Таким чином, вся складність процесу клонування інкапсульована всередині цього класу.

```java
import java.util.HashMap;
import java.util.Map;

public class FileManager {  2 usages

    private Map<String, FileSystemObject> templates = new HashMap<>();  2 usages

    public void registerTemplate(String key, FileSystemObject prototype) {  2 usages
        templates.put(key, prototype);
    }

    public FileSystemObject createFromTemplate(String key, String newPath, String newName)  2 usages
            throws CloneNotSupportedException {

        FileSystemObject prototype = templates.get(key);
        if (prototype == null) {
            throw new IllegalArgumentException("Template not found: " + key);
        }

        FileSystemObject clone = prototype.clone();
        clone.setName(newName);
        clone.setPath(newPath);
        return clone;
    }
}
```

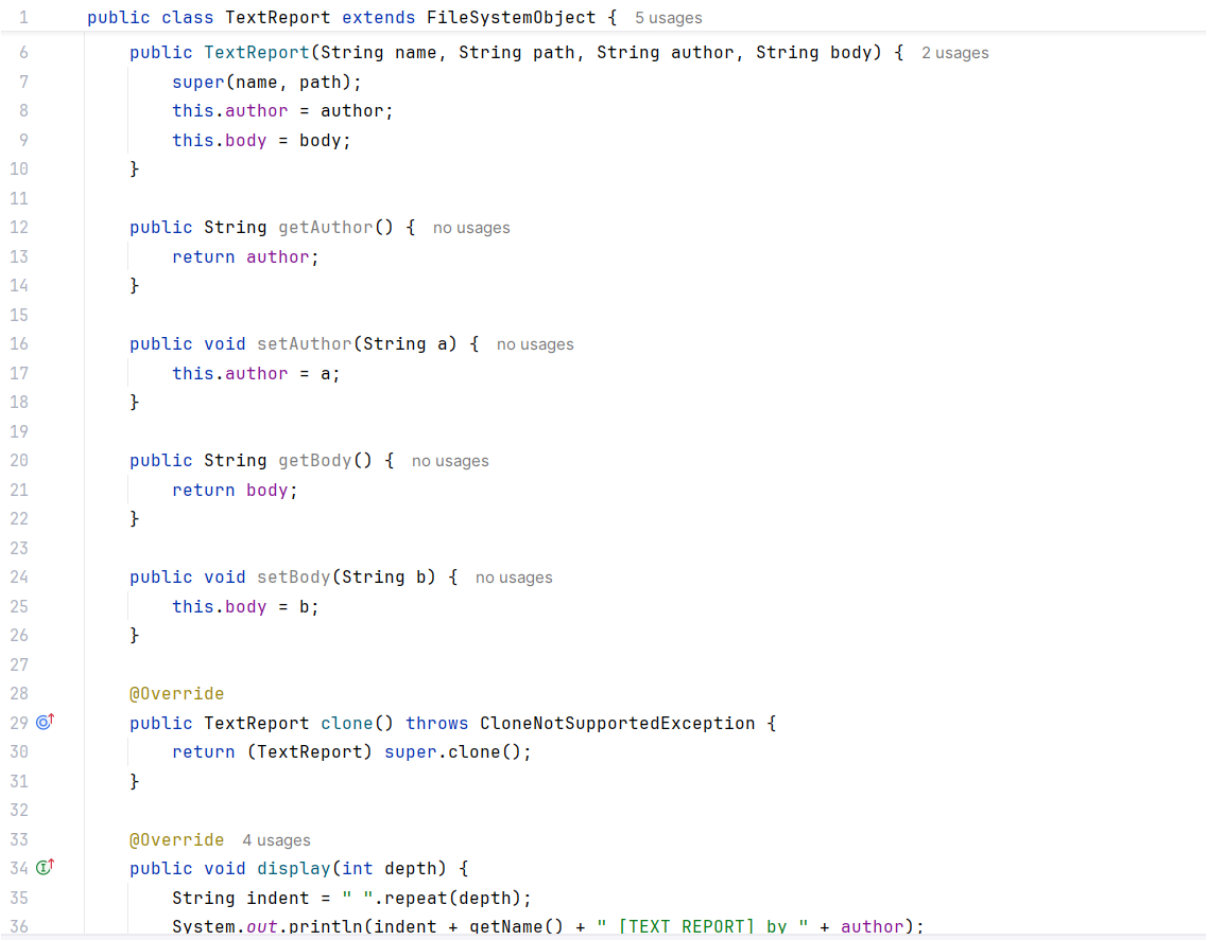
рис 2.4 - код інтерфейсу FileManager.java

Клас TextReport.java

Цей клас є новим **конкретним прототипом**, доданим для демонстрації розширюваності системи. Він представляє собою текстовий звітний файл, який, окрім стандартних властивостей (name, path), має додаткові атрибути, такі як автор (author) та зміст (body).

Ключові обов'язки класу:

- **Розширення функціональності:** Демонструє, як легко можна додати до системи новий тип об'єкта, успадкувавши FileSystemObject, без необхідності змінювати існуючу логіку FileManager чи інших компонентів.
- **Специфічний стан:** Інкапсулює унікальні для звіту поля author та body, що робить його спеціалізованим типом файлу.
- **Специфічна поведінка:** Надає власну реалізацію методу display(), яка відображає інформацію про звіт, включаючи ім'я автора.
- **Підтримка клонування:** Як і File, цей клас використовує успадкований механізм **поверхневого копіювання**, оскільки його поля (String) є незмінними (immutable), і для них цього достатньо, щоб створити функціонально незалежну копію.



```
1   public class TextReport extends FileSystemObject {  5 usages
6       public TextReport(String name, String path, String author, String body) {  2 usages
7           super(name, path);
8           this.author = author;
9           this.body = body;
10      }
11
12      public String getAuthor() {  no usages
13          return author;
14      }
15
16      public void setAuthor(String a) {  no usages
17          this.author = a;
18      }
19
20      public String getBody() {  no usages
21          return body;
22      }
23
24      public void setBody(String b) {  no usages
25          this.body = b;
26      }
27
28      @Override
29      public TextReport clone() throws CloneNotSupportedException {
30          return (TextReport) super.clone();
31      }
32
33      @Override  4 usages
34      public void display(int depth) {
35          String indent = " ".repeat(depth);
36          System.out.println(indent + getName() + " [TEXT REPORT] by " + author);
```

рис 2.5 - код класу TextReport.java

Клас Shell.java

Цей клас є **клієнтом** у контексті патерну «Прототип» і служить **точкою входу** в програму (main). Його основна мета - продемонструвати практичне застосування патерну та роботу FileManager. Він не створює об'єкти напряму, а повністю делегує це завдання менеджеру.

Ключові обов'язки класу:

- **Ініціалізація системи:** Створює екземпляр FileManager, який буде керувати всіма прототипами.
- **Створення та конфігурація прототипів:** Готує початкові об'єкти-шаблони — як простий (TextReport), так і складний (Directory). Важливо, що складний прототип projectTemplate будується послідовно, використовуючи імутабельний метод add(), що гарантує безпеку та передбачуваність стану прототипу.

- **Реєстрація прототипів:** Передає створені шаблони до FileManager за допомогою методу registerTemplate(), асоціюючи їх з логічними іменами ("textReport", "projectDir").
- **Демонстрація використання:** Викликає метод createFromTemplate(), делегуючи завдання створення об'єктів менеджеру і оперуючи лише логічними іменами, що демонструє повну незалежність клієнта від конкретних класів прототипів.
- **Візуалізація результатів:** Виводить створені об'єкти в консоль, що наочно показує, як складні структури були відтворені за одну операцію клонування, підтверджуючи коректну роботу глибокого копіювання.

```
public class Shell {

    public static void main(String[] args) {
        try {
            FileManager manager = new FileManager();

            TextReport reportTemplate = new TextReport(
                    name: "ReportTemplate.txt",
                    path: "/templates",
                    author: "Admin",
                    body: "Щоденний звіт про роботу системи..."
            );
            manager.registerTemplate( key: "textReport", reportTemplate);

            Directory projectTemplate = new Directory( name: "ProjectTemplate", path: "/templates");

            Directory srcDir = new Directory( name: "src", path: "/templates/ProjectTemplate/src");
            srcDir = srcDir.add(new File( name: "Main.java", path: "/templates/ProjectTemplate/src", sizeKb: 120));
            srcDir = srcDir.add(new File( name: "Utils.java", path: "/templates/ProjectTemplate/src", sizeKb: 80));

            Directory docsDir = new Directory( name: "docs", path: "/templates/ProjectTemplate/docs");
            docsDir = docsDir.add(new TextReport(
                    name: "ProjectReport.txt",
                    path: "/templates/ProjectTemplate/docs",
                    author: "TeamLead",
                    body: "Опис функціональності проєкту..."
            ));

            projectTemplate = projectTemplate.add(srcDir);
            projectTemplate = projectTemplate.add(docsDir);

            manager.registerTemplate( key: "projectDir", projectTemplate);
```

```
33
34              FileSystemObject dailyReport = manager.createFromTemplate(
35                      key: "textReport",
36                      newPath: "/reports",
37                      newName: "DailyReport_25_10.txt"
38              );
39
40              FileSystemObject myProject = manager.createFromTemplate(
41                      key: "projectDir",
42                      newPath: "/projects",
43                      newName: "MyAwesomeApp"
44              );
45
46              System.out.println("=== Демонстрація патерну Prototype ===\n");
47
48              System.out.println("1. Оригінальний шаблон звіту:");
49              reportTemplate.display( depth: 0);
50
51              System.out.println("\n2. Новий звіт, створений шляхом клонування:");
52              dailyReport.display( depth: 0);
53
54              System.out.println("\n3. Новий проєкт, створений шляхом глибокого клонування:");
55              myProject.display( depth: 0);
56
57          } catch (Exception e) {
58              e.printStackTrace();
59          }
60      }
61  }
```

рис 2.6 - код класу Shell.java

Опис результатів коду

```
=== Демонстрація патерну Prototype ===

1. Оригінальний шаблон звіту:
ReportTemplate.txt [TEXT REPORT] by Admin

2. Новий звіт, створений шляхом клонування:
DailyReport_25_10.txt [TEXT REPORT] by Admin

3. Новий проєкт, створений шляхом глибокого клонування:
[DIR] MyAwesomeApp
  [DIR] src
    Main.java (120 KB)
    Utils.java (80 KB)
  [DIR] docs
    ProjectReport.txt [TEXT REPORT] by TeamLead
```

рис 2.7 - результати виконання коду

Наданий скріншот демонструє консольний вивід, який є результатом роботи оновленої програми. Він наочно ілюструє всі ключові етапи та переваги використання патерну «Прототип», включаючи роботу з різними типами об'єктів та коректну реалізацію глибокого копіювання.

Аналіз виводу:

1. **Клонування простого (атомарного) прототипу:**
   Перші два блоки виводу порівнюють оригінальний шаблон ReportTemplate.txt з його клонованою копією DailyReport_25_10.txt. Ця частина демонструє:
   - **Збереження внутрішнього стану:** Новий звіт зберіг внутрішні властивості прототипу (автор "Admin").
   - **Можливість кастомізації:** Зовнішні властивості, такі як ім'я файлу, були успішно змінені після клонування, що підтверджує створення нового, незалежного екземпляра.

2. **Клонування складного (композитного) прототипу:**
   Третій блок є найбільш показовим, оскільки він демонструє створення нового проєкту MyAwesomeApp шляхом клонування складної, ієрархічної структури. Цей результат підтверджує:
   - **Коректність глибокого копіювання:** Була відтворена вся вкладена структура прототипу, включаючи підтеки src і docs, а також всі файли всередині них (Main.java, Utils.java).
   - **Робота з різними типами об'єктів:** Система успішно скопіювала об'єкти різних типів (Directory, File, TextReport) в рамках однієї операції, що підкреслює гнучкість архітектури.
   - **Повна незалежність копії:** Завдяки реалізації глибокого копіювання, новостворений проєкт MyAwesomeApp є повністю незалежною копією. Будь-які подальші зміни в ньому (наприклад, додавання нових файлів) не вплинуть на стан оригінального прототипу projectTemplate.

Можемо зробити висновок: вивід програми повністю підтверджує успішну реалізацію патерну «Прототип». Він демонструє, що система здатна ефективно створювати як прості, так і складні композитні об'єкти шляхом клонування, приховуючи від клієнта деталі цього процесу. Це підкреслює головні переваги патерну: зменшення складності коду, підвищення продуктивності при створенні "важких" об'єктів та гнучкість у конфігурації системи.

3. Діаграма класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами була побудована UML-діаграма класів. Вона наочно демонструє структуру, що відповідає шаблону проектування «Prototype».

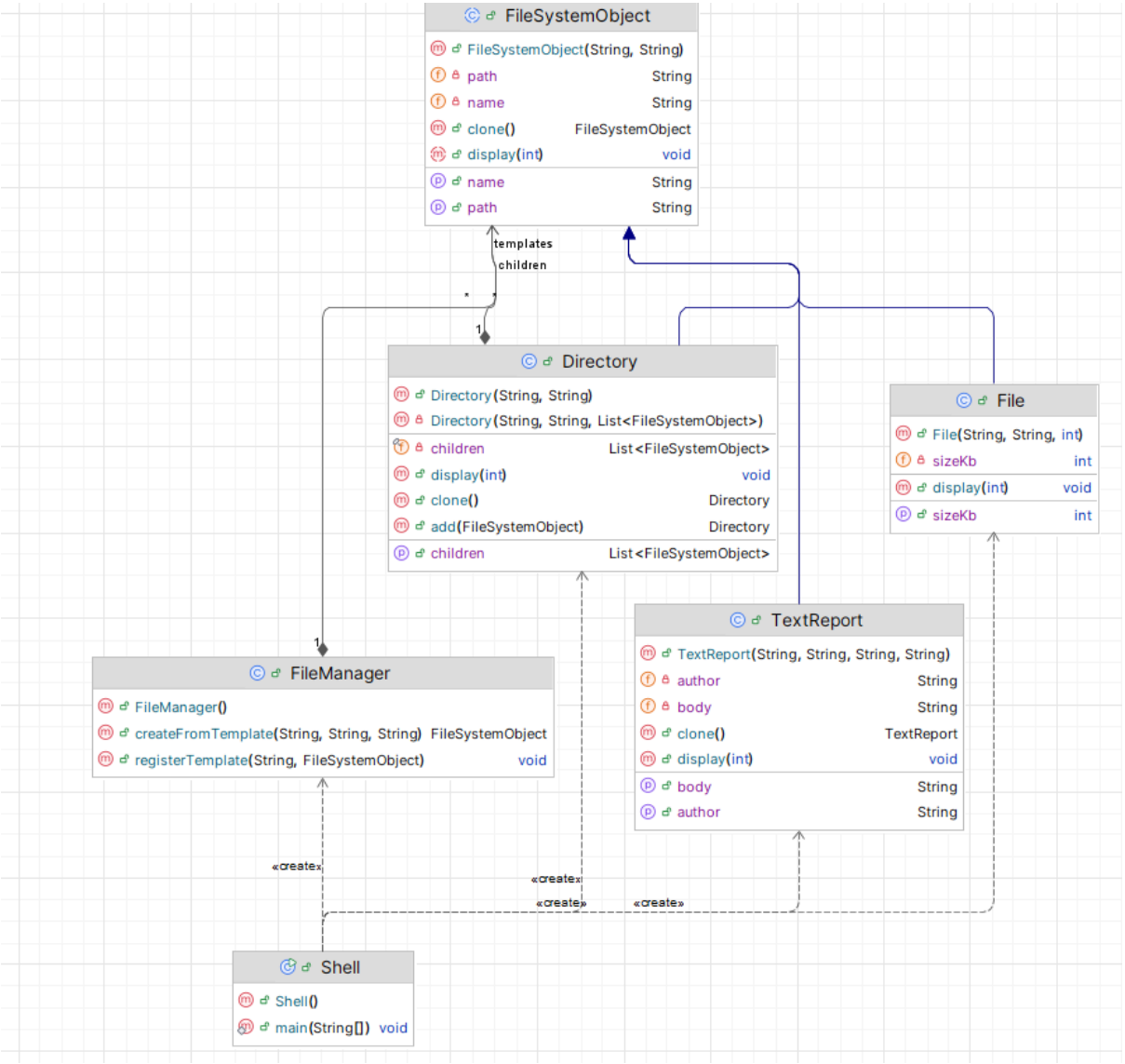


рис 3.1 - діаграма класів

На скріншоті представлена UML-діаграма, яка візуалізує статичну структуру розробленої програмної системи та взаємозв'язки між її ключовими компонентами. Діаграма точно відображає реалізацію патерну «Прототип» у поєднанні з Реєстром Прототипів, а також демонструє розширену ієрархію класів.

Основні елементи діаграми:

1. **Ієрархія прототипів:**
    - В основі архітектури лежить абстрактний клас FileSystemObject, який виступає в ролі **базового прототипу**.

Він визначає загальний інтерфейс (clone(), display()) та стан для всіх об'єктів.
    - Класи File, Directory та новий клас TextReport є **конкретними прототипами**. Вони успадковують FileSystemObject, що показано суцільною лінією з трикутною стрілкою (відношення наслідування). Додавання класу TextReport наочно демонструє гнучкість та розширюваність архітектури.
2. **Композиційні зв'язки (Агрегація):**
    - Зв'язок між Directory та FileSystemObject (позначений як children) показаний як агрегація (лінія з порожнім ромбом). Це означає, що екземпляр Directory містить у собі колекцію (*) інших об'єктів FileSystemObject, що дозволяє створювати ієрархічні деревоподібні структури.
    - Аналогічний зв'язок існує між FileManager та FileSystemObject (позначений як templates). FileManager агрегує колекцію прототипів, виконуючи роль їхнього сховища або **реєстру**.
3. **Клієнт та залежності:**
    - Клас Shell виступає в ролі **клієнта**. Його зв'язки з іншими класами показані пунктирними лініями (відношення залежності) з підписом «create». Це вказує на те, що Shell не зберігає постійних посилань на ці класи, а лише створює їхні екземпляри в методі main для ініціалізації системи та демонстрації її роботи.
4. **Відображення імутабельності:**
    - На діаграмі видно, що метод add() у класі Directory повертає тип Directory. Це візуально підтверджує реалізацію **імутабельного підходу**, де операція додавання не змінює поточний об'єкт, а створює і повертає новий екземпляр з оновленим станом.

Можемо зробити висновок, що: діаграма чітко показує розподіл ролей відповідно до патерну «Прототип», розширену ієрархію об'єктів, а також важливі архітектурні рішення, такі як використання імутабельності. Діаграма демонструє гнучку, безпечну та легко розширювану систему.

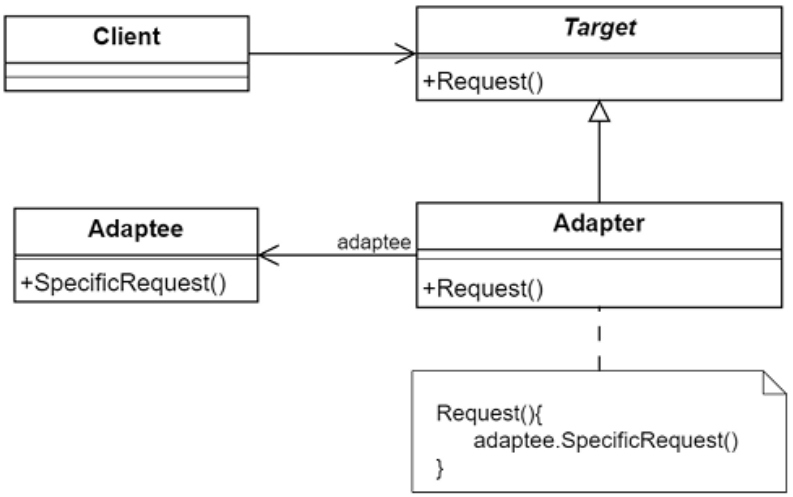
4. Висновки

Висновки: В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано патерн проектування «Прототип» для вирішення задачі гнучкого створення об'єктів у файловому менеджері. Була створена розширювана ієрархія класів (File, Directory, TextReport), здатних до клонування. Особливу увагу було приділено класу Directory, для якого було реалізовано механізм глибокого копіювання, що забезпечило повну незалежність копій, а також застосовано підхід імутабельності, що гарантує безпеку стану зареєстрованих прототипів та виключає їх випадкову зміну. Впровадження класу FileManager у ролі реєстру прототипів дозволило централізувати керування шаблонами та повністю абстрагувати клієнтський код від процесів ініціалізації конкретних класів. Практична реалізація продемонструвала ключові переваги патерну: значне спрощення процесу створення складних об'єктів, підвищення надійності коду завдяки імутабельності та гнучкість системи, що дозволяє динамічно додавати нові типи об'єктів без зміни існуючого коду. Таким чином, мета лабораторної роботи була повністю досягнута, а ефективність та архітектурні переваги патерну «Прототип» були підтверджені на практиці.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?
Шаблон «Адаптер» (Adapter) призначений для перетворення інтерфейсу одного класу в інтерфейс, очікуваний клієнтом. Він дозволяє класам працювати разом, навіть якщо їхні інтерфейси несумісні, виступаючи в ролі посередника між ними.

2. Нарисуйте структуру шаблону «Адаптер».



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

- **Target:** Інтерфейс, який очікує та використовує клієнт (Client).
- **Client:** Клієнтський клас, який взаємодіє з об'єктами через інтерфейс Target.
- **Adaptee:** Клас з несумісним інтерфейсом, який потрібно адаптувати.
- **Adapter:** Клас, що реалізує інтерфейс Target і містить посилання на об'єкт Adaptee. Коли клієнт викликає метод Adapter, той, у свою чергу, перетворює цей виклик у відповідний виклик методу об'єкта Adaptee.

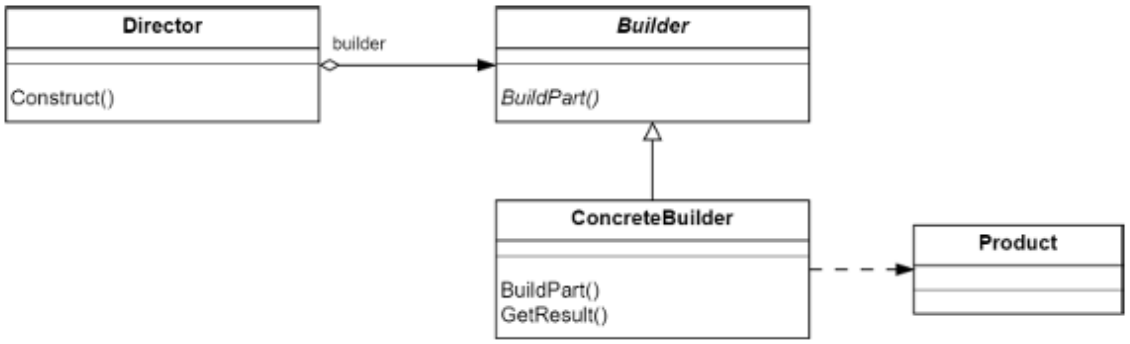
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

- **Адаптер класів** використовує **наслідування**. Клас Adapter успадковує одночасно і клас Adaptee, і інтерфейс Target. Цей підхід можливий у мовах, що підтримують множинне наслідування.
- **Адаптер об'єктів** використовує **композицію**. Клас Adapter реалізує інтерфейс Target і містить у собі екземпляр класу Adaptee. Цей підхід є більш гнучким, оскільки дозволяє адаптувати цілу ієрархію класів Adaptee.

5. Яке призначення шаблону «Будівельник»?
Шаблон «Будівельник» (Builder) призначений для того, щоб відокремити процес конструювання складного об'єкта від його представлення. Це

дозволяє використовувати один і той же процес конструювання для створення різних варіацій об'єкта.

6. Нарисуйте структуру шаблону «Будівельник».



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

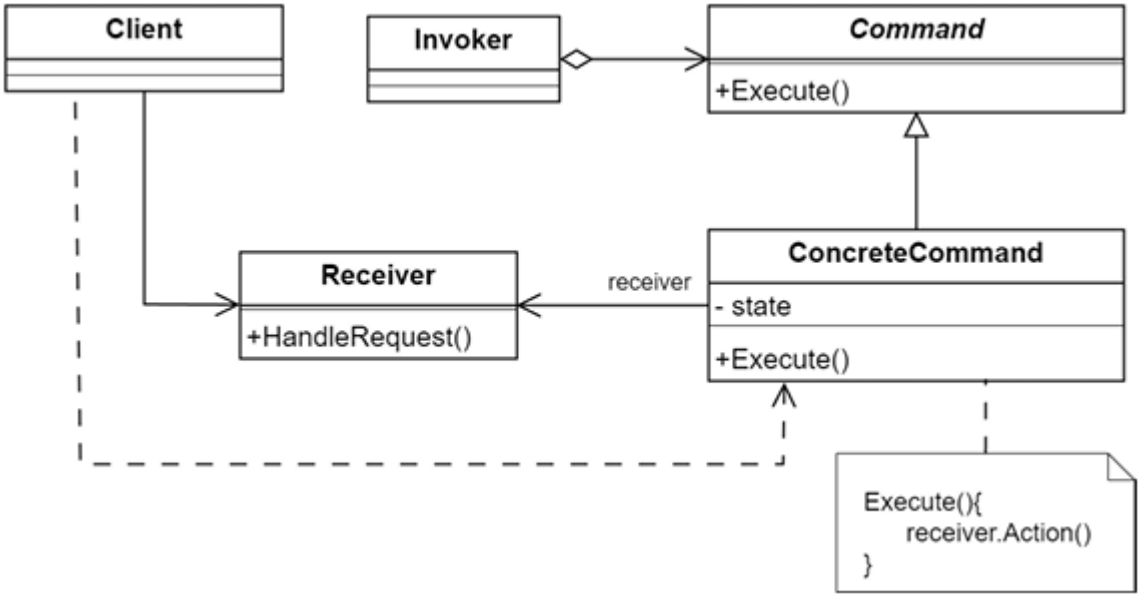
- **Product:** Складний об'єкт, який створюється.
- **Builder:** Абстрактний інтерфейс, що визначає кроки для створення частин об'єкта Product.
- **ConcreteBuilder:** Конкретна реалізація інтерфейсу Builder, яка конструює та збирає частини об'єкта.
- **Director:** Клас, який керує процесом конструювання, викликаючи методи Builder у правильній послідовності. Director не знає про конкретну реалізацію Builder чи Product.

8. У яких випадках варто застосовувати шаблон «Будівельник»"?
Шаблон варто застосовувати, коли:

- Процес створення об'єкта є складним, багатоетапним і не повинен бути прив'язаний до класів, що його складають.
- Необхідно створювати різні представлення одного й того ж об'єкта.
- Потрібно уникнути "телескопічного конструктора" (конструктора з великою кількістю необов'язкових параметрів).

9. Яке призначення шаблону «Команда»?
Шаблон «Команда» (Command) призначений для інкапсуляції запиту (виклику операції) у вигляді об'єкта. Це дозволяє параметризувати клієнтські об'єкти різними запитами, ставити запити в чергу, логувати їх, а також підтримувати операції скасування (undo).

10. Нарисуйте структуру шаблону «Команда».



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

- **Command:** Інтерфейс, що оголошує метод для виконання операції (зазвичай execute()).
- **ConcreteCommand:** Клас, що реалізує інтерфейс Command. Він містить посилання на об'єкт Receiver і викликає його методи для виконання запиту.
- **Receiver:** Об'єкт, який безпосередньо виконує операцію (містить бізнес-логіку).
- **Invoker:** Об'єкт, який ініціює виконання команди. Він не знає нічого про Receiver, а лише викликає метод execute() у об'єкта Command.
- **Client:** Створює об'єкт ConcreteCommand і встановлює для нього Receiver.

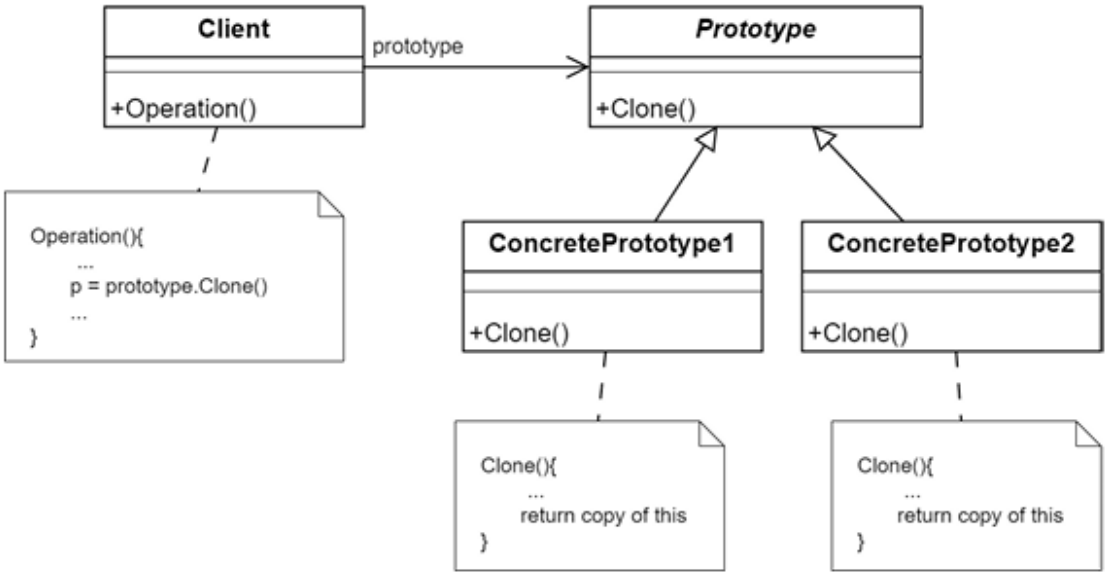
12. Розкажіть як працює шаблон «Команда».

Замість того, щоб Invoker безпосередньо викликав метод Receiver, клієнт створює об'єкт Command, який містить у собі посилання на Receiver та інформацію про те, яку дію потрібно виконати. Цей об'єкт Command передається Invoker. Коли настає час виконати дію, Invoker просто викликає єдиний метод execute() у команди. Команда, у свою чергу, делегує виклик відповідному методу Receiver.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) призначений для створення нових об'єктів шляхом копіювання (клонування) існуючого об'єкта, який називається прототипом. Це дозволяє уникнути прямої залежності від класів об'єктів, що створюються, і може бути ефективнішим за звичайне створення екземпляра.

14. Нарисуйте структуру шаблону «Прототип».



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

- **Prototype:** Абстрактний клас або інтерфейс, що оголошує метод клонування.
- **ConcretePrototype:** Конкретний клас, що реалізує метод клонування. Він відповідає за створення копії самого себе.
- **Client:** Клієнтський клас, який створює новий об'єкт, викликаючи метод клонування у існуючого об'єкта-прототипа.

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

- **Системи обробки GUI-подій:** Коли користувач клікає на кнопку, подія може бути оброблена спочатку самою кнопкою, потім її батьківською панеллю, потім вікном, і так далі по ієрархії, доки не знайдеться обробник.

- **Middleware у веб-фреймворках:** Вхідний HTTP-запит послідовно проходить через ланцюжок проміжних обробників (middleware) для аутентифікації, логування, кешування тощо.
- **Системи логування:** Повідомлення може передаватися по ланцюжку логерів (консольний, файловий, мережевий), кожен з яких вирішує, чи обробляти це повідомлення залежно від його рівня важливості.
- **Системи затвердження документів:** Заявка на відпустку або звіт про витрати послідовно передається на затвердження від одного менеджера до іншого по ланцюжку ієрархії.