Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

**Лабораторна робота №8**

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «ШАБЛОНИ «COMPOSITE»,

«FLYWEIGHT», «INTERPRETER»,

«VISITOR»»

Варіант №26

Виконав: Перевірив:

студент групи ІА-23 Мягкий М. Ю.

Мозоль В.О

Київ 2024

**Зміст**

[**Тема. 3**](#_Toc184660745)

[**Мета. 3**](#_Toc184660746)

[**Завдання. 3**](#_Toc184660747)

[**Обрана тема. 3**](#_Toc184660748)

[**Короткі теоретичні відомості. 4**](#_Toc184660749)

[**Хід роботи. 6**](#_Toc184660750)

[**Робота паттерну. 9**](#_Toc184660751)

[**Висновки. 10**](#_Toc184660752)

[**Додаток А. 11**](#_Toc184660753)

# **Тема.**

Шаблони «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»

# **Мета.**

Метою лабораторної роботи є вивчення та практичне застосування шаблонів проєктування: Composite, Flyweight, Interpreter та Visitor. Розробка функціоналу Download manager з використанням патерну Composite для організації структури завантажень. Реалізація механізму групування та відображення завантажень з різними статусами

# **Завдання.**

1 . Ознайомитися з короткими теоретичними відомостями.

2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

3. Застосування одного з розглянутих шаблонів при реалізації програми

# **Обрана тема.**

**26 Download manager (iterator, command, observer, template method,**

**composite, p2p)**

Інструмент для скачування файлів з інтернету по протоколах http або https з

можливістю продовження завантаження в зупиненому місці, розподілу

швидкостей активним завантаженням, ведення статистики завантажень,

інтеграції в основні браузери (firefox, opera, internet explorer, chrome).

# **Короткі теоретичні відомості.**

Патерн Active Record

Active Record — це підхід, у якому об'єкт управляє як даними, так і поведінкою. Цей патерн передбачає, що об’єкти сутностей, які відповідають рядкам таблиць бази даних, містять усю логіку для доступу до БД і маніпуляцій із даними. Такі об'єкти виступають «обгортками» для рядків з бази даних і включають методи для збереження, оновлення або видалення записів.

Цей підхід часто використовується завдяки простоті й зручності. У ньому кожна сутність прямо пов’язана зі своєю таблицею. Наприклад, ORM-фреймворки, такі як Active Record у Ruby on Rails, побудовані на цьому принципі. Однак, зі зростанням складності програми, логіка запитів може ставати занадто складною для підтримки в одному класі. У таких випадках її виносять до окремих об’єктів чи шарів, щоб поліпшити структуру коду.

Патерн Table Data Gateway

Table Data Gateway представляє підхід, у якому взаємодія з базою даних делегується окремому класу, що відповідає за певну таблицю. Цей клас містить методи для виконання CRUD-операцій (створення, читання, оновлення та видалення) і логіку формування SQL-запитів.

Такий підхід забезпечує більшу гнучкість і спрощує тестування, оскільки відокремлює дані від логіки взаємодії з базою даних. Щоб уникнути дублювання коду (наприклад, з'єднання з базою даних чи формування базових запитів), часто створюється базовий клас, який використовують усі шлюзи.

Наприклад, у системі з кількома сутностями можна створити окремі класи-шлюзи для кожної таблиці, що дозволяє централізувати SQL-запити й забезпечує можливість спрощеної заміни джерела даних, якщо це буде необхідно.

Патерн Data Mapping

Data Mapping вирішує проблему перетворення об'єктів даних в рядки реляційної бази даних або інші джерела. Маппери відповідають за трансформацію даних, коригуючи невідповідності між типами полів у базі даних і об’єктах.

Цей підхід створює окремі об’єкти (або методи в класі), які відповідають за перетворення. У типовій реалізації маппери забезпечують двостороннє перетворення між об’єктами даних і таблицями реляційної бази. Наприклад, ORM-фреймворки, як Hibernate, використовують цей підхід для автоматичного перетворення Java-об’єктів у SQL-запити.

Патерн Composite

Composite дозволяє створювати деревоподібну структуру об'єктів для представлення ієрархії типу «частина-ціле». Цей підхід дозволяє об'єднувати об'єкти в композити й працювати з ними так само, як і з окремими об'єктами.

Наприклад, у GUI-програмах форма може містити текстові поля, кнопки, зображення. При виконанні операції, як-от масштабування, всі елементи форми обробляються рекурсивно, незалежно від рівня їх вкладеності.

Composite широко використовується для роботи зі складними ієрархіями об'єктів, наприклад, у структурах військових підрозділів або для представлення складених замовлень у системах електронної комерції. Кожен компонент (лист або контейнер) реалізує однаковий інтерфейс, що спрощує виконання операцій над будь-яким елементом дерева.

Патерн Flyweight

Flyweight спрямований на оптимізацію використання пам'яті, коли програма працює з великою кількістю об'єктів, більшість із яких містять однакові дані. Основна ідея — розділення об’єктів на два стани: внутрішній (незмінний і спільний для багатьох об'єктів) і зовнішній (специфічний для кожного об'єкта).

Наприклад, у текстовому редакторі об'єкти для букв можуть ділити між собою однакові властивості (шрифт, розмір, колір), тоді як їхнє положення на сторінці визначається окремими параметрами.

Flyweight корисний у випадках, коли багато однакових об'єктів використовуються в різних контекстах, як у графічних системах для представлення частинок, що повторюються (сніжинки, кулі, зірки).

Патерн Interpreter

Interpreter застосовується для роботи з мовами програмування або розпізнавання шаблонів у текстах. Його суть у створенні об'єктів, які представляють правила граматики певної мови або шаблону.

Цей патерн дозволяє розділити процес інтерпретації на окремі класи, які відповідають за окремі правила або символи. Наприклад, у регулярних виразах інтерпретатор може представляти різні види символів (літери, цифри, спеціальні символи) і правила їхнього комбінування.

Патерн Visitor

Visitor дозволяє додавати нові операції до об'єктів без зміни їхніх класів. Цей підхід корисний, коли необхідно часто змінювати логіку обробки даних у структурі складних об'єктів.

Visitor забезпечує централізоване управління операціями, які застосовуються до різних типів об’єктів у складній структурі. Наприклад, якщо у вас є структура об’єктів, що представляють різні елементи документа (заголовки, параграфи, списки), Visitor дозволить реалізувати рендеринг, експорт або аналіз тексту без змін класів цих елементів.

# **Хід роботи.**

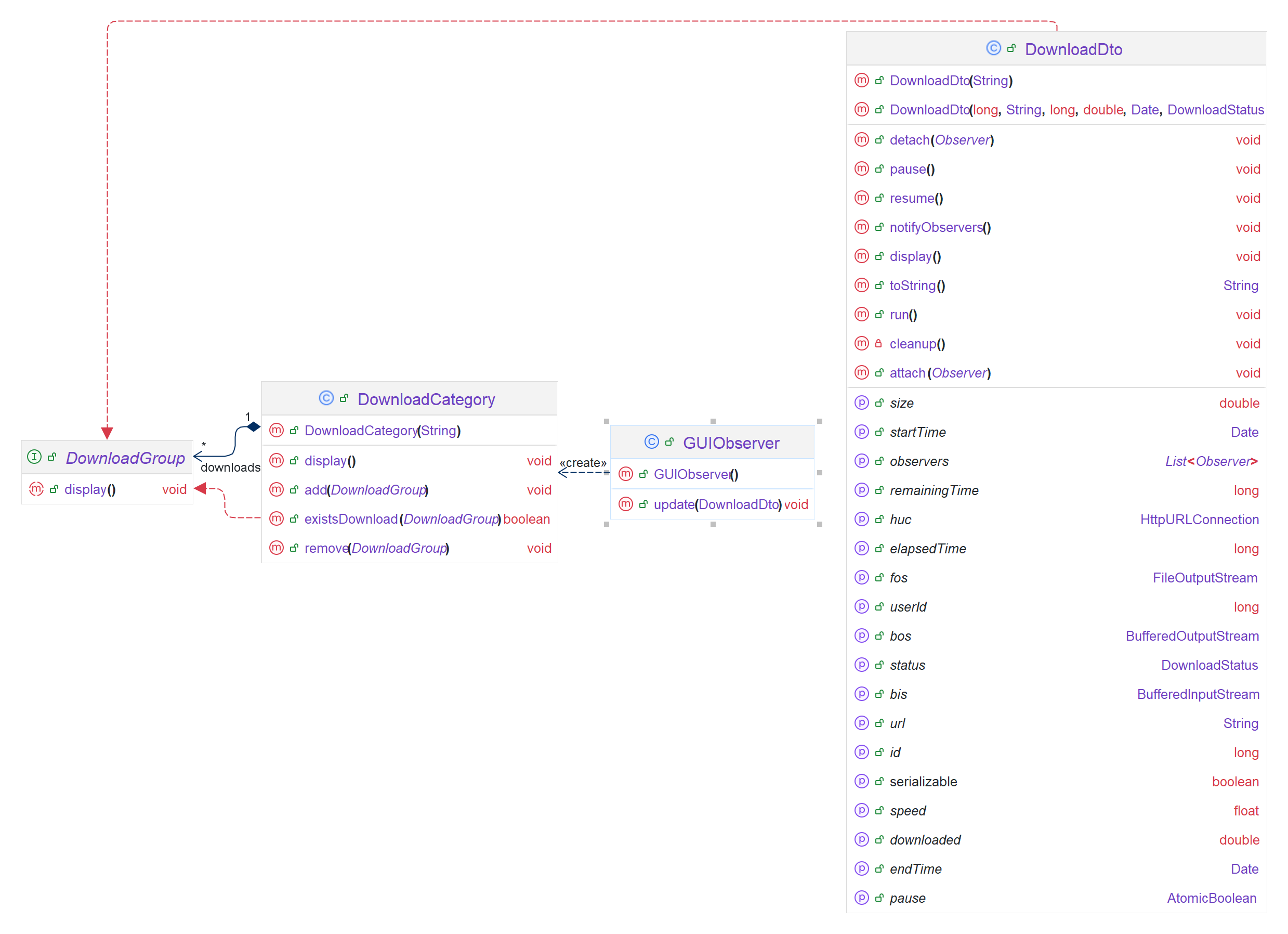


Рисунок №1 – Діаграма класів , згенерована IDE, реалізації шаблону Observer

Структура Composite

* **Компоненти (Component):** Інтерфейси та методи, які є спільними для всіх об’єктів у структурі Composite, забезпечують уніфіковану роботу з різними об'єктами в дереві. У цій діаграмі DownloadCategory є основним "компонентом", який визначає загальні методи для роботи з групами завантажень (add, remove, display тощо).
* **Листові вузли (Leaf):** Це об'єкти, які не містять дочірніх елементів і реалізують базові операції, визначені в компоненті. У цій структурі DownloadGroup представляє "лист", що виконує конкретну дію (наприклад, display) і може бути частиною більшої категорії.
* **Композитні вузли (Composite):** Це об’єкти, які можуть містити інші компоненти (включаючи листи й інші композитні вузли). У даній діаграмі DownloadCategory виконує роль Composite, що може містити об'єкти типу DownloadGroup і працювати з ними через єдиний інтерфейс.

Як це працює

1. DownloadCategory **керує групами завантажень (**DownloadGroup**)**:
   * Вона може додавати нові групи через метод addDownloadGroup.
   * Видаляти групи за допомогою remove.
   * Перевіряти наявність конкретної групи через метод existsDownloadGroup.
2. DownloadGroup **виконує специфічну дію**:
   * Метод display відповідає за відображення інформації про групу завантажень.
3. **Універсальність структури**: Завдяки Composite клієнтський код може працювати як із групами (DownloadGroup), так і з категоріями (DownloadCategory), використовуючи спільний інтерфейс. Це забезпечує гнучкість і полегшує розширення структури.

Код реалізації паттерну можна переглянути у GitHub репозиторії у папці DownloadManager або у Додатку А.

# **Робота паттерну.**

Для демонстрації роботи паттерну будемо сортувати завантаження по групам: Активні завантаження, Зупинені та Завершені. (Код для тестування є у GitHub репозиторії або у Додатку А)

Результат виконання коду:

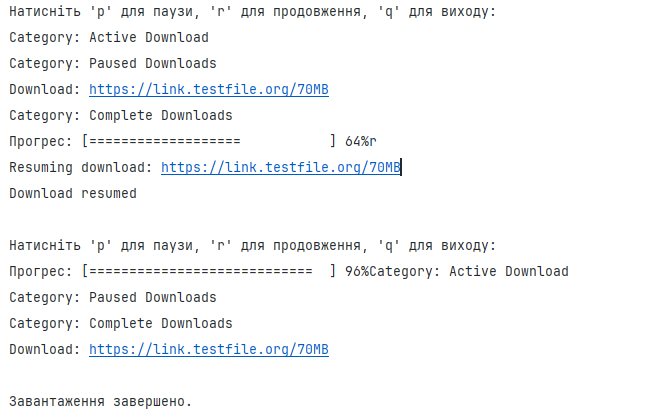


Рисунок №2 – Сортування завантажень по групам

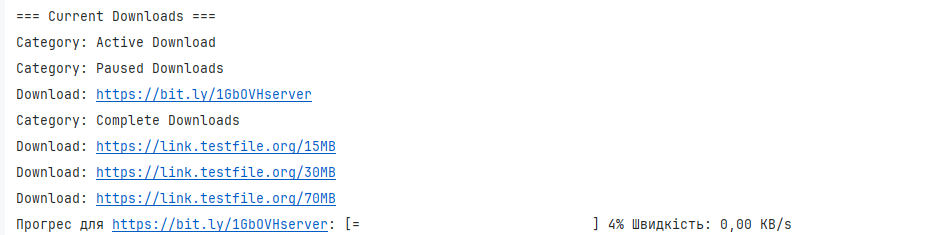


Рисунок №3 – Сортування декількох завантажень

**Висновки.**

За результатами виконання лабораторної роботи успішно реалізовано патерн Composite для структуризації завантажень у Download manager. Розроблено механізм групування завантажень за статусами: Активні, Зупинені та Завершені. Набуто практичних навичок застосування шаблонів проєктування в розробці програмного забезпечення

# **Додаток А.**

**DownloadCategory.java**

package com.project.downloadmanager.util.composite;  
  
  
import java.util.ArrayList;  
import java.util.*List*;  
  
public class DownloadCategory implements *DownloadGroup*{  
 private String name;  
 private *List*<*DownloadGroup*> downloads = new ArrayList<>();  
  
 public DownloadCategory(String name) {  
 this.name = name;  
 }  
  
 public void add(*DownloadGroup* component) {  
 downloads.add(component);  
 }  
  
 public void remove(*DownloadGroup* component) {  
 downloads.remove(component);  
 }  
  
 public boolean existsDownload(*DownloadGroup* component) {  
 return downloads.contains(component);  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("Category: " + name);  
 for (*DownloadGroup* download : downloads) {  
 download.display();  
 }  
 }  
}

**DownloadGroup.java**

package com.project.downloadmanager.util.composite;  
  
public interface *DownloadGroup* {  
 void display();  
}

**DownloadDto.java (Основний код для реалізації методу)**

package com.project.downloadmanager.model;  
  
import com.project.downloadmanager.config.ConfigLoader;  
import com.project.downloadmanager.model.enums.DownloadStatus;  
import com.project.downloadmanager.util.composite.*DownloadGroup*;  
import com.project.downloadmanager.util.observer.*Observer*;  
import com.project.downloadmanager.util.observer.*Subject*;  
import lombok.Getter;  
import lombok.Setter;  
  
import java.io.\*;  
import java.net.HttpURLConnection;  
import java.net.URL;  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.*List*;  
import java.util.Objects;  
import java.util.concurrent.atomic.AtomicBoolean;  
  
  
@Getter  
@Setter  
public class DownloadDto implements *Runnable*, *Subject*, *Serializable*, *DownloadGroup* {

@Override  
public void display() {  
 System.*out*.println("Download: " + url);  
}

**GUIObserver (Виконує роль розміщення завантажень по групам)**

package com.project.downloadmanager.util.observer.impl;  
  
import com.project.downloadmanager.model.DownloadDto;  
import com.project.downloadmanager.model.enums.DownloadStatus;  
import com.project.downloadmanager.util.composite.DownloadCategory;  
import com.project.downloadmanager.util.observer.*Observer*;  
  
public class GUIObserver implements *Observer* {  
 DownloadCategory activeDownload = new DownloadCategory("Active Download");  
 DownloadCategory completeDownloads = new DownloadCategory("Complete Downloads");  
 DownloadCategory pausedDownloads = new DownloadCategory("Paused Downloads");  
  
 @Override  
 public void update(DownloadDto dto) {  
 activeDownload.remove(dto);  
 completeDownloads.remove(dto);  
 pausedDownloads.remove(dto);  
  
 switch (dto.getStatus()) {  
 case *DOWNLOADING*:  
 if(!activeDownload.existsDownload(dto)){  
 activeDownload.add(dto);  
 }  
 break;  
 case *PAUSED*:  
 if(!pausedDownloads.existsDownload(dto)){  
 pausedDownloads.add(dto);  
 }  
 break;  
 case *COMPLETED*:  
 if(!completeDownloads.existsDownload(dto)){  
 completeDownloads.add(dto);  
 }  
 break;  
 }  
 System.*out*.println("=== Current Downloads ===");  
 activeDownload.display();  
 pausedDownloads.display();  
 completeDownloads.display();  
 }  
}