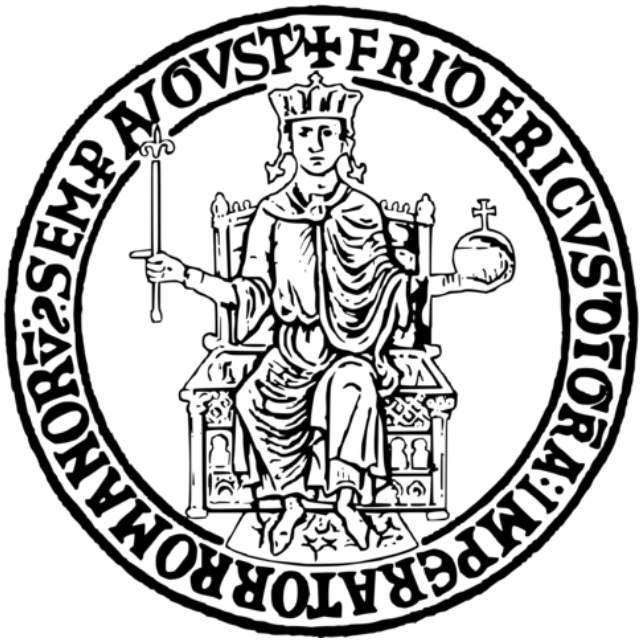
Università degli Studi di Napoli Federico II



Corso di Laurea Magistrale in *Informatica*

RELAZIONE DEL PROGETTO DI SISTEMI OPERATIVI PER DISPOSITIVI MOBILI

INFLUENZA SUL CONSUMO ENERGETICO DI PRATICHE DI PROGRAMMAZIONE E DESIGN PATTERN

Prof. Porfirio Tramontana

Dott. Francesco Calcopietro (Matr. N97000432)

A.A. 2023/2024

Sommario

Abstract

Negli ultimi decenni, la programmazione orientata agli oggetti (OOP) è diventata lo standard de facto per la progettazione di software commerciale. È facile convincersi che, con miliardi di righe di codice distribuite ed eseguite su telefoni cellulari, dispositivi IoT, PC… l'efficienza energetica del software svolge e svolgerà un ruolo sempre più importante nell'ambito del Green IT.
Con questa relazione si intende effettuare uno studio approfondito sull'impatto che hanno alcune tipiche tecniche di OOP e design pattern sull'efficienza energetica. Alcune di esse hanno un impatto trascurabile, altre invece riducono l'efficienza energetica.
Da questo studio si potrà capire quando è opportuno utilizzare tali tecniche e quando, invece, non è bene sfruttarle.
Tale relazione è disponibile, insieme ai programmi allegati, al seguente link GitHub: https://github.com/kekkokalko/Influenza-sul-consumo-energetico-di-pratiche-di-programmazione-e-design-pattern.

Introduzione

Da uno studio condotto nel 2021 da parte della compagnia Evans, ci sono circa 24 milioni di sviluppatori in tutto il mondo e questo aumenterà del 20% fino a quasi 30 milioni entro il 2025. Con circa 4,4 milioni di sviluppatori di software, gli Stati Uniti hanno attualmente il maggior numero, ma si prevede che l'India li supererà entro il 2025. (lowcodeitalia, 2021)

Vengono scritte miliardi di righe di codice ogni anno con la crescente domanda di software. L'esecuzione di questi codici su vari dispositivi hardware (ad esempio telefoni cellulari, sistemi embedded, PC e server) consuma una notevole quantità di energia.

Tuttavia, gli sviluppatori di software potrebbero sapere molto poco su come migliorare l'efficienza energetica del software sviluppato. In effetti, molti di loro potrebbero non sapere nemmeno quanta energia viene consumata dal loro software. Di conseguenza, abbondano software eccessivi e inefficienti sotto questo punto di vista.

Dall'invenzione del concetto di OOP all'inizio degli anni '60, la programmazione è stata notevolmente migliorata sia dal punto di vista teorico che dal punto di vista dell'implementazione. Molti linguaggi di programmazione (es Java, C++, ecc.) forniscono un forte supporto ai concetti di OOP. Attualmente, l'OOP è diventato l'approccio de facto nella progettazione di software commerciale su larga scala. Il grande successo dell'OOP deriva dalla modularità e riusabilità del codice, che permettono di definire tempi di rilascio ridotti del software sviluppato. Tuttavia, l'approccio OOP può ridurre le prestazioni e aumentare il consumo energetico del software (rispetto alla procedura classica di programmazione) a causa degli ulteriori livelli e meccanismi di astrazione e incapsulamento che vengono introdotti nel software.

Siccome sempre più spesso, negli ultimi anni, anche per i software dedicati ai dispositivi mobili è fortemente richiamata la programmazione Object-Oriented, è facile convincersi che, per evitare consumi smisurati della batteria di questi ultimi, risulta fondamentale sviluppare dei software che siano in grado di limitare i consumi energetici.

Questa relazione contribuisce allo stato dell'arte della programmazione orientata agli oggetti presentando uno studio empirico che valuta quantitativamente l'impatto di tre tecniche di OOP (ereditarietà, polimorfismo e overload di operatori) e sei design pattern (decorator, builder, adapter, composite, chain of responsability e state) sulle prestazioni e sull'efficienza energetica del software.

La relazione presenta un'ampia sezione dedicata alle caratteristiche di ogni tecnica e ogni design pattern, sottolineando i vantaggi che si hanno nell'applicarli, e un'altra sezione dedicata specificamente alle prestazioni e ai consumi energetici che ognuno di loro promette nel momento in cui vengono usufruiti. Verrà fatta un'analisi sia nel momento in cui la tecnica e/o il design pattern usufruito viene richiamato una sola volta, sia un'analisi empirica dei casi in cui tali pratiche vengono richiamate più volte nello stesso programma. Tutto ciò per avere un feedback completo dal punto di vista energetico di ogni singolo scenario.

Tecniche di programmazione

In questa sezione si vuole descrivere in maniera dettagliata le varie tecniche di programmazione che si intende sottoporre ad un'analisi specifica dal punto di vista dell'efficienza energetica per poi generare risultati sperimentali.

Ereditarietà

L'ereditarietà è la caratteristica dei linguaggi Object Oriented che consente di utilizzare definizioni di classe come base per la definizione di nuove che ne specializzano il concetto. Nell'ereditarietà si devono avere bene a mente le seguenti definizioni:

- Classe base / Superclasse: si riferisce ad una classe utilizzata come fondamenta per una nuova definizione di classe;
- Classe derivata / Sottoclasse: si riferisce ad una classe che eredita tutte le funzionalità dalla classe base o superclasse;

L'ereditarietà avviene grazie al comando *Extends* che viene posto di seguito alla dichiarazione iniziale della classe.

L'estensione di classe può avvenire solo per una classe; infatti, non è possibile l'ereditarietà multipla se non per quanto riguarda le interfacce (che non hanno limiti di implementazione). (skuola.net, 2016)

L'ereditarietà offre vari vantaggi:

1. **Riutilizzabilità del codice:** il codice scritto nella super classe è comune a tutte le sottoclassi. Le classi figlie possono utilizzare direttamente il codice della classe genitore.
2. **Astrazione:** il concetto di astratto in cui non si deve fornire tutti i dettagli si ottiene attraverso l'ereditarietà. L'astrazione mostra solo la funzionalità all'utente. (geeksforgeeks.org, 2023)

Per gli scopi che si intende raggiungere, vengono sviluppati due programmi differenti: uno senza ereditarietà e uno con ereditarietà.
Nel primo programma vengono create 10 classi senza ereditarietà. Tutte le classi hanno gli stessi metodi della precedente ma si crea ì un'istanza di ciascuna classe e vengono richiamati i loro metodi separatamente. Di seguito un screen del codice di tale programma:

```
class Parent{
    public Parent(){
    }
    public void showP() { System.out.println("Ciao! Sono la classe Parent!"); }
}

class Child{

    public Child(){}
    public void showC(){
        System.out.println("Ciao! Sono la classe Child!");

    }
}

public class BeforeInheritance{
    public static void main(String[]args){
        Parent p = new Parent();
        Child c = new Child();
    }
}
```

Il secondo programma contiene 10 classi (Classi A, B, C…, L). La 10° classe (classe L) eredita da tutte le 9 classi precedenti. Ogni classe ha un costruttore, che viene chiamato ogni volta che viene creata una nuova istanza di quella classe.
Poiché la classe L eredita da tutte le classi precedenti, ogni volta che un oggetto viene creato da L, verrà creata implicitamente anche un'istanza di tutte le altre classi. Pertanto, la maggior parte del tempo di esecuzione viene dedicato alla creazione di oggetti, che richiamano i costruttori corrispondenti. Poiché L eredita da altre classi, possono essere chiamati i loro metodi (solo pubblici e protetti) con un'istanza della classe L poiché l'ereditarietà consente di creare nuove classi che ereditano automaticamente i codici dalle classi esistenti. Di seguito uno screen del codice di tale programma:

```java
class Parent{
    public Parent(){
    }
    public void showP() { System.out.println("Ciao! Sono la classe Parent!"); }
}

class Child extends Parent{

    public Child(){}
    public void showC(){
        System.out.println("Ciao! Sono la classe Child!");

    }
}

public class AfterInheritance{
    public static void main(String[]args){
        Parent p = new Parent();
        Parent c = new Child();
    }
}
```

Polimorfismo

Dal greco “molte forme”. si riferisce in generale alla possibilità, data ad una determinata espressione, di assumere valori diversi in relazione ai tipi di dato a cui viene applicata. (html.it, 2015)

Esistono due tipologie di polimorfismo nell’ambito OOP: polimorfismo per metodi e polimorfismo per dati. Ci si vuole concentrare, più nello specifico, alla prima categoria. Essa consiste nel cambiare la firma dei metodi. Può avvenire in due modi:

1. **Overload**: chiamato anche sovraccarico, cioè definire un metodo che sostituisce un altro. Il nuovo metodo avrà parametri diversi (quindi una *signature* differente) rispetto al vecchio metodo.
2. **Override**: chiamato anche riscrittura, cioè modificare le operazioni di un metodo specifico.

L’analisi che si intende effettuare per gli scopi di questo studio è riferita al solo override.

Tale tecnica viene molto sfruttata “a braccetto” con la tecnica dell’ereditarietà. Infatti, le sottoclassi possono, avendo delle loro caratteristiche e funzionalità, ridefinire un metodo o una serie di metodi che ereditano dalla loro superclasse. Tali metodi riscritti nelle sottoclassi dovranno avere la stessa firma dei metodi ereditati. Il tipo di dato di ritorno di tali metodi riscritti dovrà essere lo stesso del metodo originale o deve essere di un tipo che estende il tipo di dato di ritorno del metodo originale. Inoltre, il metodo ridefinito nella sottoclasse non deve essere meno accessibile del metodo originale.

Per gli scopi di tale studio, vengono definiti due programmi: il primo non sfrutta l’override, il secondo definisce tale tecnica. In particolare, il primo presenta due classi (Parent e Child) che definiscono due metodi che hanno lo stesso scopo (una semplice stampa a video) ma con nome differente: saranno, in questo modo, duemetodi distinti e separati. Di seguito uno screen del codice sorgente di tale programma:

```java
class Parent{
    public Parent(){
    }
    public void showP() { System.out.println("Ciao! Sono la classe Parent!"); }
}

class Child{

    public Child(){}
    public void showC(){
        System.out.println("Ciao! Sono la classe Child!");

    }
}

public class BeforeOverride {
    public static void main(String[]args){
        Parent p = new Parent();
        p.showP();
        Child c = new Child();
        c.showC();
    }
}
```

Il secondo programma, invece, lega le due classi da ereditarietà e sfrutta l'override del metodo *show().* In questo modo sarà possibile, nel momento in cui verrà richiamato il nome di tale metodo, richiamare, di conseguenza, il metodo *show()* della classe specifica di interesse. Sarà, quindi, il tipo di oggetto a cui si farà riferimento che determinerà quale versione del metodo, sottoposto a override, verrà eseguito. Di seguito uno screen del codice sorgente:

```java
class Parent{
    public Parent(){}
    public void show() { System.out.println("Ciao! Sono la classe Parent!"); }
}

class Child extends Parent{
    public Child(){}
    @Override
    public void show() { System.out.println("Ciao! Sono la classe Child!"); }
}


public class AfterOverride {
    public static void main(String[]args){
        Parent p = new Parent();
        p.show();
        Parent p1 = new Child();
        p1.show();
    }
}
```

Overload di operatori

L'overload degli operatori è una funzionalità della programmazione orientata agli oggetti che consente a un programmatore di ridefinire un operatore integrato per lavorare con tipi di dati definiti dall'utente.
Per verificare le performance e l'efficienza energetica di tale tecnica, si vuole analizzare una coppia di programmi. Entrambi hanno come obiettivo la risoluzione della seguente espressione matematica:

$$\frac{(i_1+i_2)}{(i_3+i_4)} * \frac{(i_1-i_2)}{(i_3-i_4)}$$

Il primo programma fa uso della libreria Math richiamando i suoi metodi *addExact()*, *subctractExtact()*, *multiplyExtact()* e *divideExact().* Di seguito il codice sorgente di tale programma:

```java
import java.util.*;
import java.io.*;
import java.math.BigDecimal;


class BeforeOverloadOperatori {
    public static void main(String[] args){
        int sum1= Math.addExact(1,2);
        int sum2=Math.addExact(3,4);
        double divisione1= Math.divideExact(sum1,sum2);
        int sottrazione1= (int) Math.subtractExact(1,2);
        int sottrazione2= (int) Math.subtractExact(3,4);
        int divisione2=Math.divideExact(sottrazione1,sottrazione2);
        int moltiplicazione = Math.multiplyExact((int) divisione1, divisione2);
        System.out.println(moltiplicazione);
    }
}
```

Il secondo programma fa uso, semplicemente degli operatori +,-,*,/ sovraccaricati, per risolvere l'espressione. DI seguito il codice sorgente di tale programma:

```java
public class AfterOverloadOperatori {
    public static void main(String[] args){
        System.out.println(((1+2)/(3+4))*((1-2)/(3-4)));
    }
}
```

Design Pattern

A questo punto della relazione si vuole fare la stessa presentazione, fatta per le tecniche OOP, ai Design Pattern che sono stati scelti per poter condurre questo studio. Per tutti i pattern considerati verranno implementati due programmi: il primo che non fa uso di pattern e il secondo, invece, che ne fa uso.

Decorator

Nella programmazione orientata agli oggetti, il pattern decoratore è un Design Pattern che consente di aggiungere un comportamento o una caratteristica a un singolo oggetto, in modo dinamico, senza influenzare il comportamento di altre istanze della stessa classe.
L'uso del decoratore può essere più efficiente della creazione di sottoclassi, perché il comportamento di un oggetto può essere aumentato senza definire un oggetto completamente nuovo. Di seguito il diagramma UML di tale pattern.

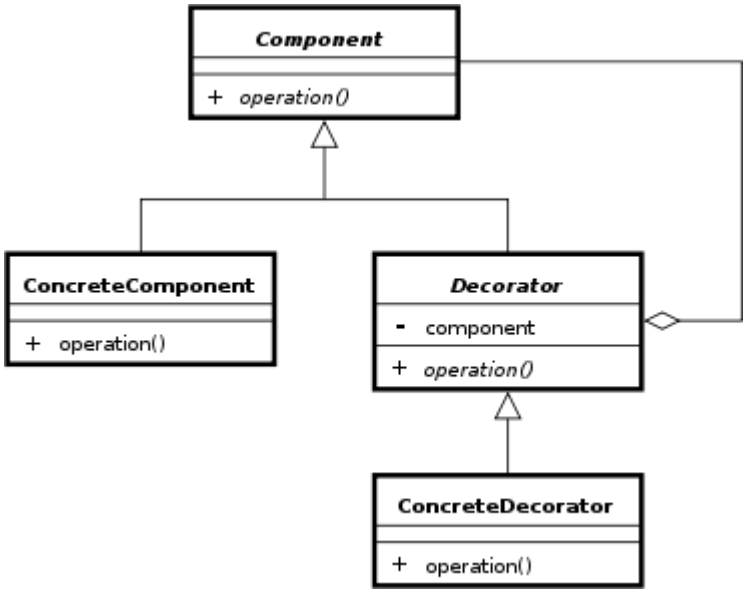


Figura 1 Diagramma UML di Decorator

Il primo programma che è stato analizzato non fa uso del pattern Decorator. Vengono definite diverse classi che estendono una classe base *A*, ciascuna con comportamenti aggiuntivi. Infatti, le classi *AwithX*, *aWithY*, *aWithZ*, *AwithXY*, e *AwithXYZ* estendono la classe *A* e aggiungono funzionalità come *X*, *Y*, e *Z*.
Nel metodo *main()*, vengono creati oggetti di diverse classi e chiamato il metodo *doIt()* su ognuno di essi. Di seguito una coppia di screen del codice sorgente:

```java
class A {

    public void doIt() { System.out.print('A'); }
}


class AwithX extends A {

    public  void doIt() {
        super.doIt();
        doX();
    }


    private void doX() { System.out.print('X'); }
}


class aWithY extends A {

    public void doIt() {
        super.doIt();
        doY();
    }


    public void doY() { System.out.print('Y'); }
}


class aWithZ extends A {

    public void doIt() {
        super.doIt();
        doZ();
    }


    public void doZ() { System.out.print('Z'); }
}


class AwithXY extends AwithX {

    private aWithY obj = new aWithY();

    public void doIt() {
        super.doIt();
        obj.doY();
    }
}
```

```java
class AwithXYZ extends AwithX {

    private aWithY obj1 = new aWithY();

    private aWithZ obj2 = new aWithZ();


    public void doIt() {
        super.doIt();
        obj1.doY();
        obj2.doZ();
    }
}


public class BeforeDecorator {

    public static void main(String[] args) {
        A[] array = {new AwithX(), new AwithXY(), new AwithXYZ()};
        for (A a : array) {
            a.doIt();
            System.out.print("  ");
        }
    }
}
```

Il secondo programma fa uso del pattern Decorator. Infatti:

- viene definita un'interfaccia *I* con un metodo doIt().
- viene definita una classe base *A* che implementa l'interfaccia *I*.
- viene definita una classe astratta *D* che implementa l'interfaccia *I* e contiene un riferimento a un oggetto di tipo *I*.
- le classi *X, Y,* e *Z* estendono la classe *D* e aggiungono funzionalità specifiche.
- nel metodo *main()*, vengono creati oggetti di diverse classi e chiamato il metodo *doIt()* su ognuno di essi.

Di seguito una coppia di screen che illustrano il codice sorgente:

```
interface I {

    void doIt();
}


class A implements I {

    public void doIt() { System.out.print('A'); }
}


abstract class D implements I {

    private I core;

    public D(I inner) { core = inner; }


    public void doIt() { core.doIt(); }
}


class X extends D {

    public X(I inner) { super(inner); }


    public void doIt() {
        super.doIt();
        doX();
    }


    private void doX() { System.out.print('X'); }
}
```

```
class Y extends D {

    public Y(I inner) { super(inner); }


    public void doIt()  {
        super.doIt();
        doY();
    }


    private void doY() { System.out.print('Y'); }
}


class Z extends D {

    public Z(I inner) { super(inner); }


    public void doIt()  {
        super.doIt();
        doZ();
    }


    private void doZ() { System.out.print('Z'); }
}


public class AfterDecorator {

    public static void main( String[] args ) {
        I[] array = {new X(new A()), new Y(new X(new A())),
                new Z(new Y(new X(new A())))};
        for (I anArray : array) {
            anArray.doIt();
            System.out.print("  ");
        }
    }
}
```

Entrambi i programmi raggiungono lo stesso risultato finale, che è quello di fornire un modo flessibile per aggiungere comportamenti aggiuntivi a un oggetto senza dover modificare il suo codice sorgente. Tuttavia, il secondo programma utilizza un'implementazione più modulare e orientata agli oggetti utilizzando l'ereditarietà e il principio di composizione.

Builder

Il design pattern Builder, nella programmazione ad oggetti, separa la costruzione di un oggetto complesso dalla sua rappresentazione cosicché il processo di costruzione stesso possa creare diverse rappresentazioni.
L'algoritmo per la creazione di un oggetto complesso è indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate.

Ciò ha l'effetto immediato di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla corretta costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti. Questo è particolarmente utile quando si vuole assicurare che un oggetto sia valido prima di istanziarlo, e non si vuole che la logica di controllo appaia nei costruttori degli oggetti. Un builder permette anche di costruire un oggetto passo-passo, cosa che si può verificare quando si fa il parsing di un testo o si ottengono i parametri da un'interfaccia interattiva.

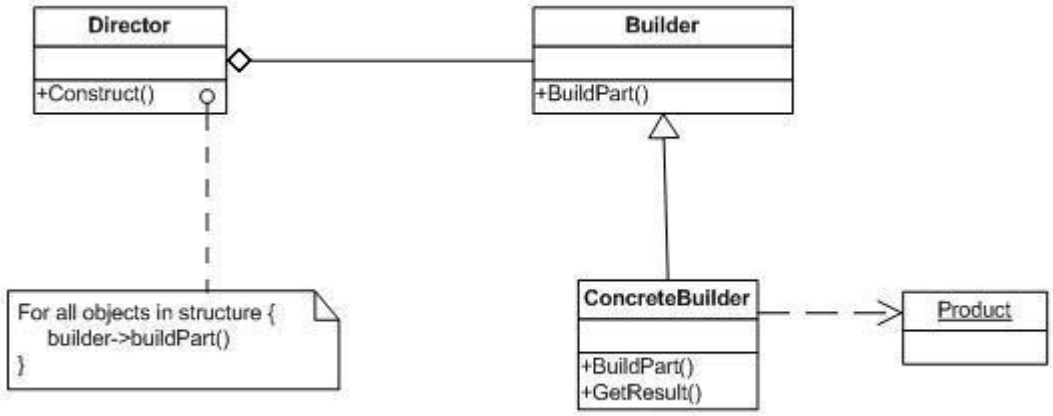


Figura 2 Diagramma UML di Builder

Il primo programma non fa uso del pattern suddetto.
Esso crea un'istanza della classe *Automobile* direttamente nel metodo *main*(). La classe *Automobile* ha un costruttore che richiede diversi parametri per creare un'istanza dell'oggetto. Nel metodo *main()*, viene istanziata un'automobile con un modello, una marca, un anno e un colore specificati. Questi valori vengono passati direttamente al costruttore della classe *Automobile*. Successivamente, vengono utilizzati i metodi getter per ottenere i valori degli attributi dell'oggetto Automobile creato e stamparli a schermo.
Di seguito uno screen che definisce il codice sorgente:

```java
import java.util.*;

// Senza l'uso del pattern Builder
class Automobile {
    private String modello;

    public String getModello() {
        return modello;
    }

    public String getMarca() {
        return marca;
    }

    public int getAnno() {
        return anno;
    }

    public String getColore() {
        return colore;
    }

    private String marca;
    private int anno;
    private String colore;

    public Automobile(String modello, String marca, int anno, String colore) {
        this.modello = modello;
        this.marca = marca;
        this.anno = anno;
        this.colore = colore;
    }


}

public class BeforeBuilder {
    public static void main(String[] args) {
        Automobile auto = new Automobile("Fiesta", "Ford", 2022, "Blu");
        System.out.println("Automobile: " + auto.getModello() + ", " + auto.getMarca() + ", " + auto.getAnno() + ", " + auto.getColore());
    }
}
```

Il secondo programma utilizza il pattern Builder per creare un'istanza della classe *Automobile*. La classe *Automobile* ha un costruttore privato che prende un oggetto di tipo Builder come parametro. Il Builder ha i suoi metodi per impostare i valori degli attributi dell'automobile, come modello, marca, anno e colore. Nel metodo *main()*, viene istanziato un oggetto *Builder* e i suoi metodi vengono chiamati per impostare i valori desiderati per l'automobile. Infine, il metodo *build()* del *Builder* viene chiamato per ottenere un'istanza dell'oggetto *Automobile* con i valori desiderati. Questo approccio rende più flessibile la costruzione dell'oggetto e facilita la lettura del codice, specialmente quando ci sono molti attributi da impostare. Di seguito il codice sorgente del programma appena descritto:

```java
import java.util.*;

// Con il pattern Builder
class Automobile {
    private String modello;
    private String marca;
    private int anno;
    private String colore;

    private Automobile(Builder builder) {
        this.modello = builder.modello;
        this.marca = builder.marca;
        this.anno = builder.anno;
        this.colore = builder.colore;
    }

    public String getModello() {
        return modello;
    }

    public String getMarca() {
        return marca;
    }

    public int getAnno() {
        return anno;
    }

    public String getColore() {
        return colore;
    }

    public static class Builder {
        private String modello;
        private String marca;
        private int anno;
        private String colore;

        public Builder(String modello, String marca) {
            this.modello = modello;
            this.marca = marca;
        }

        public Builder setAnno(int anno) {
            this.anno = anno;
            return this;
        }

        public String getModello() {
            return modello;
        }

        public String getMarca() {
            return marca;
        }

        public int getAnno() {
            return anno;
        }

        public String getColore() {
            return colore;
        }

        public Builder setColore(String colore) {
            this.colore = colore;
            return this;
        }

        public Automobile build() { return new Automobile(this); }
    }

    // Getters
}

public class AfterBuilder {
    public static void main(String[] args) {
        Automobile auto = new Automobile.Builder("Fiesta", "Ford")
                .setAnno(2022)
                .setColore("Blu")
                .build();
        System.out.println("Automobile: " + auto.getModello() + ", " + auto.getMarca() + ", " + auto.getAnno() + ", " + auto.getColore());
    }
}
```

Nonostante l'output dei due programmi sia lo stesso, il secondo programma utilizza il pattern Builder definendo un programma più strutturato e flessibile, consentendo una migliore separazione delle responsabilità e una maggiore estendibilità del codice.

Adapter

Con il nome adapter, o adattatore, si denota un design pattern nella programmazione orientata agli oggetti. A volte viene chiamato wrapper (ovvero involucro) per il suo schema di funzionamento.
Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il problema si presenta ogni qual volta nel progetto di un software si debbano utilizzare sistemi di supporto (come, per esempio, librerie) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti. Invece di dover riscrivere parte del sistema, compito oneroso e non sempre possibile se non si ha a disposizione il codice sorgente, può essere comodo scrivere un adapter che faccia da tramite.

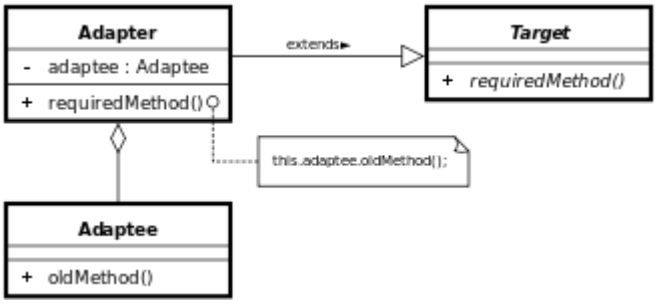


Figura 3 Diagramma UML di Adapter

Il primo programma non fa uso di tale pattern. Presenta la seguente logica:

- le classi *Line* e *Rectangle* sono implementate con i propri metodi *draw()*.
- nel metodo *main()*, viene creato un array di oggetti *Object* che contiene istanze di *Line* e *Rectangle*.
- Durante l'iterazione dell'array, viene verificato il tipo di ciascun oggetto utilizzando la reflection (*getClass().getSimpleName()*), e quindi viene eseguito un cast esplicito all'interno del blocco *if / else* per chiamare il metodo *draw()* corretto per ciascun tipo di forma.

Di seguito il codice sorgente di tale programma:

```java
class Line {

    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "), to point B(" + x2 + ";" + y2 + ")");
    }
}


class Rectangle {

    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" + x + ";" + y + "), width: " + width
                + ", height: " + height);
    }
}


public class BeforeAdapter {

    public static void main(String[] args) {
        Object[] shapes = {new Line(), new Rectangle()};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        int width = 40, height = 40;
        for (Object shape : shapes) {
            if (shape.getClass().getSimpleName().equals("Line")) {
                ((Line)shape).draw(x1, y1, x2, y2);
            } else if (shape.getClass().getSimpleName().equals("Rectangle")) {
                ((Rectangle)shape).draw(x2, y2, width, height);
            }
        }
    }
}
```

Il secondo programma fa uso del pattern Adapter.

- In questo caso, vengono definita un'interfaccia comune (*Shape*) e le classi specifiche (*Line* e *Rectangle*) implementano questa interfaccia.
- Vengono anche implementate le classi adattatore (*LineAdapter* e *RectangleAdapter*) che consentono alle classi specifiche di conformarsi all'interfaccia comune *Shape*.
- Nel metodo *main()*, viene creato un array di oggetti *Shape* che contiene istanze di *LineAdapter* e *RectangleAdapter*.
- Durante l'iterazione dell'array, viene chiamato il metodo *draw()* su ciascun oggetto *Shape*, senza dover controllare esplicitamente il tipo di oggetto. L'adattatore si occupa di convertire i parametri passati al formato richiesto dalle classi specifiche (*Line* e *Rectangle*).

Di seguito il codice sorgente di tale programma:

```java
interface Shape {

    void draw(int x, int y, int z, int j);
}


class Line {

    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "), to point B(" + x2 + ";" + y2 + ")");
    }
}


class Rectangle {

    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" + x + ";" + y + "), width: " + width
                + ", height: " + height);
    }
}


class LineAdapter implements Shape {

    private Line adaptee;


    public LineAdapter(Line line) { this.adaptee = line; }


    @Override
    public void draw(int x1, int y1, int x2, int y2) { adaptee.draw(x1, y1, x2, y2); }
}


class RectangleAdapter implements Shape {

    private Rectangle adaptee;


    public RectangleAdapter(Rectangle rectangle) { this.adaptee = rectangle; }


    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
}


public class AfterAdapter {

    public static void main(String[] args) {
        Shape[] shapes = {new RectangleAdapter(new Rectangle()),
                new LineAdapter(new Line())};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (Shape shape : shapes) {
            shape.draw(x1, y1, x2, y2);
        }
    }
}
```

Questi due programmi sono entrambi esempi di disegno di forme geometriche (linee e rettangoli), ma utilizzano due approcci diversi per gestire la rappresentazione e il disegno delle forme: il primo programma porta a codice meno flessibile e più soggetto a errori, poiché richiede il controllo esplicito del tipo di oggetto e la gestione manuale

delle diverse implementazioni di disegno; di contro, il secondo programma è più flessibile e facilmente sottoponibile a manutenzione, poiché consente l'aggiunta di nuove forme senza modificare il codice principale e sfrutta il principio dell'incapsulamento per gestire le differenze di implementazione.

Composite

Nella programmazione ad oggetti, il Composite è uno dei pattern fondamentali. Questo pattern permette di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. Il design pattern Composite organizza gli oggetti in una struttura ad albero, nella quale i nodi sono delle composite e le foglie sono oggetti semplici.
È utilizzato per dare la possibilità ai client di manipolare oggetti singoli e composizioni in modo uniforme.

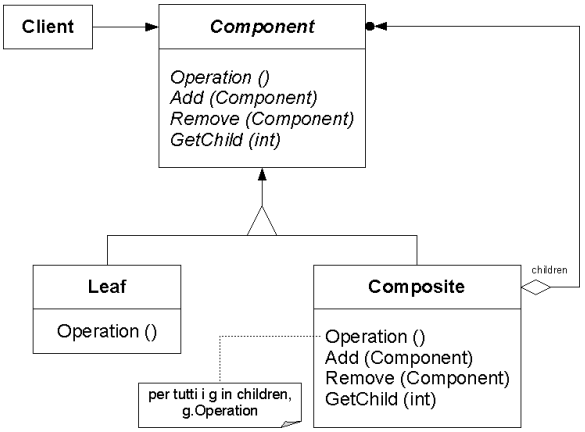


Figura 4 Diagramma UML di Composite

Entrambi i programmi cercano di affrontare il problema della gestione di un insieme di file e directory. Il primo programma non fa uso del pattern Composite. Agisce direttamente sui file e sulle directory senza un'astrazione comune. Viene utilizzato *StringBuilder* per costruire la rappresentazione della struttura delle directory e dei file. Di seguito il codice sorgente di tale programma:

```java
import java.util.ArrayList;

class File {
    private String name;

    public File(String name) { this.name = name; }

    public void ls() { System.out.println(BeforeComposite.compositeBuilder + name); }
}

class Directory {
    private String name;
    private ArrayList includedFiles = new ArrayList();

    public Directory(String name) { this.name = name; }

    public void add(Object obj) { includedFiles.add(obj); }

    public void ls() {
        System.out.println(BeforeComposite.compositeBuilder + name);
        BeforeComposite.compositeBuilder.append("   ");
        for (Object obj : includedFiles) {
            // Recover the type of this object
            String name = obj.getClass().getSimpleName();
            if (name.equals("Directory")) {
                ((Directory)obj).ls();
            } else {
                ((File)obj).ls();
            }
        }
        BeforeComposite.compositeBuilder.setLength(BeforeComposite.compositeBuilder.length() - 3);
    }
}

public class BeforeComposite {
    public static StringBuilder compositeBuilder = new StringBuilder();

    public static void main(String[] args) {
        Directory music = new Directory("MUSIC");
        Directory scorpions = new Directory("SCORPIONS");
        Directory dio = new Directory("DIO");
        File track1 = new File("Don't wary, be happy.mp3");
        File track2 = new File("track2.m3u");
        File track3 = new File("Wind of change.mp3");
        File track4 = new File("Big city night.mp3");
        File track5 = new File("Rainbow in the dark.mp3");
        music.add(track1);
        music.add(scorpions);
        music.add(track2);
        scorpions.add(track3);
        scorpions.add(track4);
        scorpions.add(dio);
        dio.add(track5);
        music.ls();
    }
}
```

Il secondo programma presenta molti più aspetti interessanti:

- è stata introdotta un'interfaccia chiamata *AbstractFile* che rappresenta il "lowest common denominator" tra file e directory. Entrambe le classi File e Directory implementano questa interfaccia. Questo consente loro di essere trattate in modo uniforme all'interno della directory e semplifica l'aggiunta di nuovi tipi di file o directory in futuro;
- la funzione *ls()* all'interno della classe Directory utilizza la ricorsione per attraversare la struttura delle directory e dei file inclusi. Questo approccio rende il codice più semplice ed elegante rispetto al primo programma, dove era necessario controllare manualmente il tipo di oggetto e invocare il metodo l*s()* corretto;
- è presente *StringBuffer* per costruire la rappresentazione della struttura delle directory e dei file. La differenza principale tra esso e *StringBuilder* è che

StringBuffer è thread-safe, mentre *StringBuilder* non lo è. Tuttavia, poiché il codice è eseguito in un contesto non concorrente, la scelta tra i due non ha un impatto significativo.

Di seguito il codice sorgente di tale programma:

```java
import java.util.ArrayList;
// Define a "lowest common denominator"
interface AbstractFile {
    void ls();
}

// File implements the "lowest common denominator"
class File implements AbstractFile {
    private String name;

    public File(String name) { this.name = name; }

    public void ls() { System.out.println(AfterComposite.compositeBuilder + name); }
}

// Directory implements the "lowest common denominator"
class Directory implements AbstractFile {
    private String name;
    private ArrayList includedFiles = new ArrayList();

    public Directory(String name) { this.name = name; }

    public void add(Object obj) { includedFiles.add(obj); }

    public void ls() {
        System.out.println(AfterComposite.compositeBuilder + name);
        AfterComposite.compositeBuilder.append("   ");
        for (Object includedFile : includedFiles) {
            // Leverage the "lowest common denominator"
            AbstractFile obj = (AbstractFile) includedFile;
            obj.ls();
        }
        AfterComposite.compositeBuilder.setLength(AfterComposite.compositeBuilder.length() - 3);
    }
}

public class AfterComposite {
    public static StringBuffer compositeBuilder = new StringBuffer();

    public static void main(String[] args) {
        Directory music = new Directory("MUSIC");
        Directory scorpions = new Directory("SCORPIONS");
        Directory dio = new Directory("DIO");
        File track1 = new File("Don't wary, be happy.mp3");
        File track2 = new File("track2.m3u");
        File track3 = new File("Wind of change.mp3");
        File track4 = new File("Big city night.mp3");
        File track5 = new File("Rainbow in the dark.mp3");
        music.add(track1);
        music.add(scorpions);
        music.add(track2);
        scorpions.add(track3);
        scorpions.add(track4);
        scorpions.add(dio);
        dio.add(track5);
        music.ls();
    }
}
```

La differenza da sottolineare è che il secondo programma (*AfterComposite*) implementa il design pattern Composite. Questo design pattern permette di trattare oggetti singoli (come file) e collezioni di oggetti (come directory) uniformemente. Ciò si riflette nell'uso dell'interfaccia *AbstractFile* e nell'implementazione ricorsiva

della funzione *ls()* nella classe *Directory*. Il primo programma (*BeforeComposite*) non utilizza questo design pattern e affronta direttamente i file e le directory senza un'astrazione comune.

Chain of Responsability

Tale pattern permette di separare gli oggetti che invocano richieste, dagli oggetti che le gestiscono, dando ad ognuno la possibilità di gestire queste richieste. Viene utilizzato il termine catena perché di fatto la richiesta viene inviata e "segue la catena" di oggetti, passando da uno all'altro, finché non trova quello che la gestisce. Tale pattern è comodo quando non si conosce a priori quale oggetto è in grado di gestire una determinata richiesta, sia perché effettivamente è sconosciuto staticamente o sia perché l'insieme degli oggetti in grado di gestire richieste cambia dinamicamente a runtime.

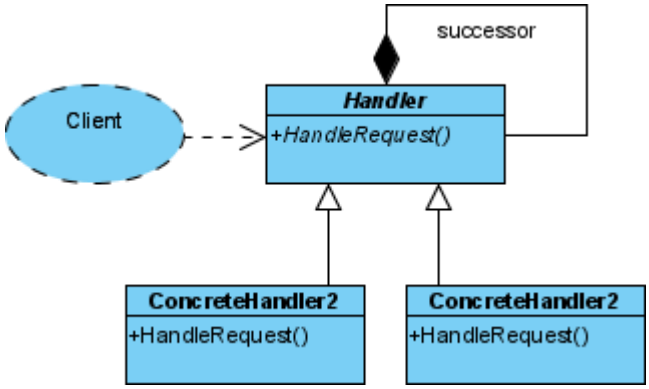


Figura 5 Diagramma UML di Chain-of-Responsability

Il primo programma implementa il design pattern Chain of Responsibility utilizzando una serie di oggetti *Handler* collegati tra loro. Ogni oggetto *Handler* ha la responsabilità di gestire un'operazione, e se non è in grado di farlo, passa la richiesta al successivo *Handler* nella catena. Questa implementazione è più rigida e meno flessibile, poiché i nodi della catena devono essere collegati manualmente nel metodo *main()*.
Di seguito il codice del programma appena descritto:

```java
import java.util.Random;

class Handler {
    private static final Random RANDOM = new Random();
    private static int nextID = 1;
    private int id = nextID++;

    public boolean execute(int num) {
        if (RANDOM.nextInt(4) != 0) {
            System.out.println("    " + id + "-busy  ");
            return false;
        }
        System.out.println(id + "-handled-" + num);
        return true;
    }
}

public class BeforeChain {
    public static void main(String[] args) {
        Handler[] nodes = {new Handler(), new Handler(),
                new Handler(), new Handler()};
        for (int i = 1, j; i < 6; i++) {
            System.out.println("Operation #" + i + ":");
            j = 0;
            while (!nodes[j].execute(i)) {
                j = (j + 1) % nodes.length;
            }
            System.out.println();
        }
    }
}
```

Il secondo programma migliora la flessibilità e la manutenibilità dell'implementazione del design pattern Chain of Responsibility introducendo i concetti di aggiunta dinamica di nuovi gestori e wrapping della catena. In questo caso, ogni oggetto *Handler* mantiene un riferimento al prossimo *Handler* nella catena e offre metodi per aggiungere un nuovo *Handler* alla catena o per avvolgere la catena attuale attorno a un nuovo gestore. Questo rende l'aggiunta di nuovi gestori più semplice e dinamico, senza la necessità di modificare il codice sorgente esistente. Di seguito il codice:

```java
import java.util.Random;

class Handler {
    private final static Random RANDOM = new Random();
    private static int nextID = 1;
    private int id = nextID++;
    private Handler nextInChain;

    public void add(Handler next) {
        if (nextInChain == null) {
            nextInChain = next;
        } else {
            nextInChain.add(next);
        }
    }

    public void wrapAround(Handler root) {
        if (nextInChain == null) {
            nextInChain = root;
        } else {
            nextInChain.wrapAround(root);
        }
    }

    public void execute(int num) {
        if (RANDOM.nextInt(4) != 0) {
            System.out.println("    " + id + "-busy  ");
            nextInChain.execute(num);
        } else {
            System.out.println(id + "-handled-" + num);
        }
    }
}

public class AfterChain {
    public static void main(String[] args) {
        Handler rootChain = new Handler();
        rootChain.add(new Handler());
        rootChain.add(new Handler());
        rootChain.add(new Handler());
        rootChain.wrapAround(rootChain);
        for (int i = 1; i < 6; i++) {
            System.out.println("Operation #" + i + ":");
            rootChain.execute(i);
            System.out.println();
        }
    }
}
```

State

Nella programmazione orientata agli oggetti, lo State è un design pattern comportamentale. Esso consente ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello stato in cui si trova.

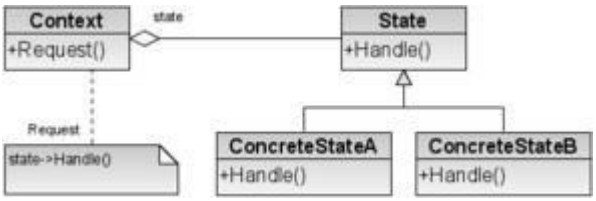


Figura 6 Diagramma UML di State

Il primo programma non fa uso di tale pattern. Ha una classe *CeilingFanPullChain* che controlla lo stato del ventilatore tramite un intero *currentState*. Questo stato viene aggiornato quando il metodo *pull()* viene chiamato. L'implementazione è sequenziale: il ventilatore si sposta attraverso gli stati in un ordine fisso (da spento a basso, medio, alto e poi di nuovo spento). Di seguito il codice sorgente:

```
import java.util.*;
import java.io.*;

class CeilingFanPullChain {
    private int currentState;

    public CeilingFanPullChain() { currentState = 0; }

    public void pull() {
        if (currentState == 0) {
            currentState = 1;
            System.out.println("low speed");
        } else if (currentState == 1) {
            currentState = 2;
            System.out.println("medium speed");
        } else if (currentState == 2) {
            currentState = 3;
            System.out.println("high speed");
        } else {
            currentState = 0;
            System.out.println("turning off");
        }
    }
}

class BeforeState {
    public static void main(String[] args) {
        CeilingFanPullChain chain = new CeilingFanPullChain();
        while (true) {
            System.out.print("Press ENTER");
            getLine();
            chain.pull();
        }
    }

    static String getLine() {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = null;
        try {
            line = in.readLine();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        return line;
    }
}
```

Il secondo programma utilizza un'interfaccia *State* che viene implementata da diverse classi, ciascuna rappresentante uno stato del ventilatore (*Off*, *Low*, *Medium*, *High*). Ogni stato ha il proprio comportamento quando il metodo *pull()* viene chiamato, e può cambiare lo stato del *CeilingFanPullChain* chiamando il metodo *set_state().* Di seguito uno screen del codice sorgente:

```
import java.util.*;
import java.io.*;

interface State {
    void pull(CeilingFanPullChain wrapper);
}

class CeilingFanPullChain {
    private State currentState;

    public CeilingFanPullChain() { currentState = new Off(); }

    public void set_state(State s) { currentState = s; }

    public void pull() { currentState.pull(this); }
}

class Off implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Low());
        System.out.println("low speed");
    }
}

class Low implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Medium());
        System.out.println("medium speed");
    }
}

class Medium implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new High());
        System.out.println("high speed");
    }
}

class High implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Off());
        System.out.println("turning off");
    }
}

class AfterState {
    public static void main(String[] args) {
        CeilingFanPullChain chain = new CeilingFanPullChain();
        while (true) {
            System.out.print("Press ENTER");
            getLine();
            chain.pull();
        }
    }

    static String getLine() {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = null;
        try {
            line = in.readLine();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        return line;
    }
}
```

Entrambi i programmi implementano lo stesso concetto di gestione degli stati, ma il secondo programma utilizza una struttura più flessibile grazie all'uso delle classi separate per rappresentare gli stati, seguendo così il principio di separazione delle responsabilità e la facilità di estensione.

Risultati sperimentali

A questo punto della relazione si vuole presentare i risultati sperimentali ottenuti dai vari programmi proposti per dimostrare l'impatto delle varie tecniche di OOP e dei vari Design Pattern su prestazioni ed efficienza energetica.
Gli esperimenti sono stati effettuati su un calcolatore prettamente di uso personale e domestico. Tale calcolatore è dotato delle seguenti componenti:

- Processore: Intel(R) Core (TM) i5-8400 CPU @ 2.80GHz, 2808 Mhz, 6 core, 6 processori logici
- RAM: 16GB, 3200 MHz
- GPU: Radeon RX 580 Series

PowerJoular e JoularJX

Per la misurazione dell'efficienza energetica è stato sfruttato **JoularJX**. Esso è un agente basato su Java per il monitoraggio energetico a livello di codice sorgente con supporto per le moderne versioni Java e multi-OS per monitorare il consumo energetico di hardware e software. È stato emanato nel 2022 da parte del Dott. Adel Noureddine e il suo Team in Francia.
Tale software è stato sviluppato insieme a **PowerJoular**. Quest'ultimo monitora il consumo energetico di CPU e GPU per PC, server e computer a scheda singola. JoularJX, basandosi, pertanto, su PowerJoular, analizza i suoi dati forniti e monitora il consumo energetico dei metodi e del codice sorgente delle applicazioni Java. PowerJoular è scritto in Ada per fornire uno strumento a basso impatto poiché Ada è costantemente classificato tra i linguaggi di programmazione più efficienti dal punto di vista energetico, migliorando allo stesso tempo la manutenibilità e la sicurezza del codice perché si mira anche al monitoraggio dei computer a scheda singola e dei dispositivi embedded. PowerJoular è rivolto a sviluppatori di software e ad amministratori di sistema con l'obiettivo di aiutare questi utenti a comprendere il consumo energetico dei loro dispositivi e software e di creare strumenti più approfonditi utilizzando tale piattaforma. Il monitoraggio della potenza di PowerJoular si basa su due moduli:

1. per PC/server: Intel RAPL tramite Linux Power Capping Framework e, facoltativamente, l'interfaccia di gestione di sistema di NVIDIA
2. per Raspberry Pi (computer a singola scheda): modelli di potenza di regressione empirica sviluppati dal team del Dott. Noureddine.

PowerJoular rileva automaticamente la configurazione del computer e i moduli supportati e fornisce i dati di alimentazione di conseguenza.
Per la GPU, PowerJoular controlla se NVIDIA SMI è installato, quindi lo utilizza per verificare se il monitoraggio della potenza della GPU è supportato dalla scheda grafica specifica e per leggere il consumo energetico della GPU ogni secondo.

Per la CPU, PowerJoular utilizza i dati di alimentazione Intel RAPL tramite l'interfaccia Linux Powercap leggendo i file di sistema appropriati.
Innanzitutto, rileva quali domini di potenza sono supportati dalla CPU:

- Pkg: supportato a partire dalle CPU Intel Sandy Bridge e fornisce il consumo energetico dei core della CPU, la grafica integrata, il controller di memoria e le cache di ultimo livello. PowerJoular controlla anche se il dominio di potenza DRAM è supportato (RAM collegata al controller di memoria) e aggiunge le sue letture di potenza al totale.
- Psys: è supportato a partire dalle CPU Intel Skylake e fornisce il consumo energetico per l'intero SOC (incluso Pkg insieme ad altri componenti, come eDRAM, PCH, System Agent). Se Psys è supportato, verrà utilizzato esclusivamente da PowerJoular per il consumo energetico della CPU anziché Pkg e DRAM, in quanto fornisce una lettura più completa della potenza del SOC della CPU.

Alla fine, PowerJoular aggrega le letture di potenza di tutti i componenti supportati per fornire un consumo energetico complessivo.
Oltre a monitorare il consumo energetico dei componenti hardware (CPU, GPU), PowerJoular può monitorare il consumo energetico della CPU di un singolo processo fornendo il suo PID in fase di esecuzione.
PowerJoular è stato progettato per essere efficiente, con poche risorse, flessibile e intuitivo da usare. L'interazione con lo strumento avviene tramite un'interfaccia a riga di comando. Tale interfaccia offre flessibilità per sperimentazioni scientifiche, monitoraggio headless in ambienti server e può essere facilmente incorporata in framework o dashboard esterni.
Il monitoraggio della potenza in fase di esecuzione può essere visualizzato sul terminale e/o scritto su file CSV.
La figura seguente mostra l'interfaccia della riga di comando di PowerJoular.

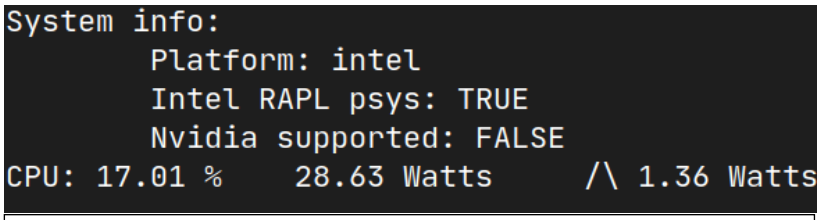


Figura 7 Output dell'interfaccia della riga di comando di PowerJoular

L'opzione dell'uso di un file CSV memorizza i dati di potenza ogni secondo e può quindi essere letta per ricostruire i dati dello storico del consumo energetico di un dispositivo o di un processo specifico.
Se un PID viene monitorato, i suoi dati di potenza verranno visualizzati sul terminale e verranno archiviati in un file CSV distinto, mentre la potenza del dispositivo verrà salvata in modo indipendente.

Al termine di ogni sessione di monitoraggio, PowerJoular visualizza il consumo energetico totale della sessione (e per il PID monitorato).

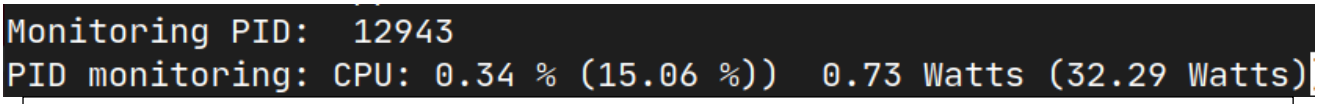


Figura 8 Output dell'interfaccia della riga di comando di PowerJoular durante il monitoraggio di un PID specidfico

La grande flessibilità di PowerJoular consente l'integrazione e l'utilizzo da parte di altri strumenti. Un esempio è JoularJX, uno strumento di monitoraggio dell'alimentazione del codice sorgente per i programmi Java.
Esso è un agente Java che si aggancia alla Java Virtual Machine per monitorare il consumo energetico, in fase di esecuzione, di ogni metodo del software monitorato. L'agente, nel momento in cui viene eseguito, avvia automaticamente PowerJoular con i parametri appropriati (monitoraggio del PID del programma Java e scrittura dei dati di potenza in un file CSV). L'utilizzo della CPU viene quindi monitorato ogni secondo per ogni thread Java e il consumo energetico viene allocato di conseguenza. Un secondo ciclo di monitoraggio rileva, per ogni thread, quale metodo viene attualmente eseguito ogni 10 millisecondi (osservando il primo metodo nello stacktrace del thread); pertanto, il consumo energetico viene allocato statisticamente a ciascun metodo. Poiché questo monitoraggio energetico include anche i metodi propri della JDK, c'è anche un'opzione per monitorare metodi specifici in base ai loro nomi completi. Ad esempio, possono essere monitorati tutti i metodi appartenenti a un determinato pacchetto allocando la potenza consumata dal primo metodo di uno stacktrace del thread al metodo che lo ha chiamato (analizzando l'albero dello stacktrace). In entrambi i casi, JoularJX genererà sempre i dati di potenza per tutti i metodi in un file separato e i metodi specifici in un altro file.
La documentazione completa è disponibile al seguente link.

Uso dello strumento

Il codice sorgente e i file binari di JoularJX sono disponibili al seguente link GitHub.
Attualmente tale tool supporta ambienti Windows e GNU/Linux, su PC/Server e dispositivi Raspberry Pi.
Una volta che si è stati indirizzati alla repository, effettuare il download dell'archivio contenente lo strumento. Una volta fatto ciò, effettuare l'operazione di unzip dell'archivio. Dopodiché, per effettuare l'installazione, eseguire lo script opportuno presente nella cartella */install.* A questo punto:

1. Su Windows, eseguire da riga di comando il file *windows-install.bat*. Questo installerà JoularJX nella cartella *C:\joularjx*
2. Su GNU/Linux, eseguire da terminale *sh linux-install.sh.* Questo installerà JoularJX nella cartella */opt/joularjx.*

JoularJx richiede una versione minima di Java 11.

JoularJX dipende dai seguenti software o pacchetti per ottenere la lettura della potenza:

- Su Windows, JoularJX utilizza un programma di monitoraggio dell'alimentazione personalizzato che sfrutta l'API Intel Power e pertanto richiede l'installazione dello strumento Intel Power Gadget e l'utilizzo di una CPU Intel supportata.
- Su PC/server GNU/Linux, JoularJX utilizza l'interfaccia Intel RAPL tramite powercap e pertanto richiede l'esecuzione di una CPU Intel o una CPU AMD Ryzen.
- Su macOS, JoularJX utilizza powermetrics, uno strumento in bundle con macOS che richiede l'esecuzione con l'accesso al comando *sudo*. Si consiglia di autorizzare gli utenti correnti a eseguire */usr/bin/powermetrics* senza richiedere una password apportando la modifica appropriata al file *sudoers*.

Avendo effettuato l'installazione completa, si può provvedere all'effettivo utilizzo dello strumento.
La prima operazione è la compilazione del file *.java* contenente il *main()* del programma da analizzare. Questo lo si effettua con il comando *javac*. L'istruzione completa è la seguente:

javac nomefile.java

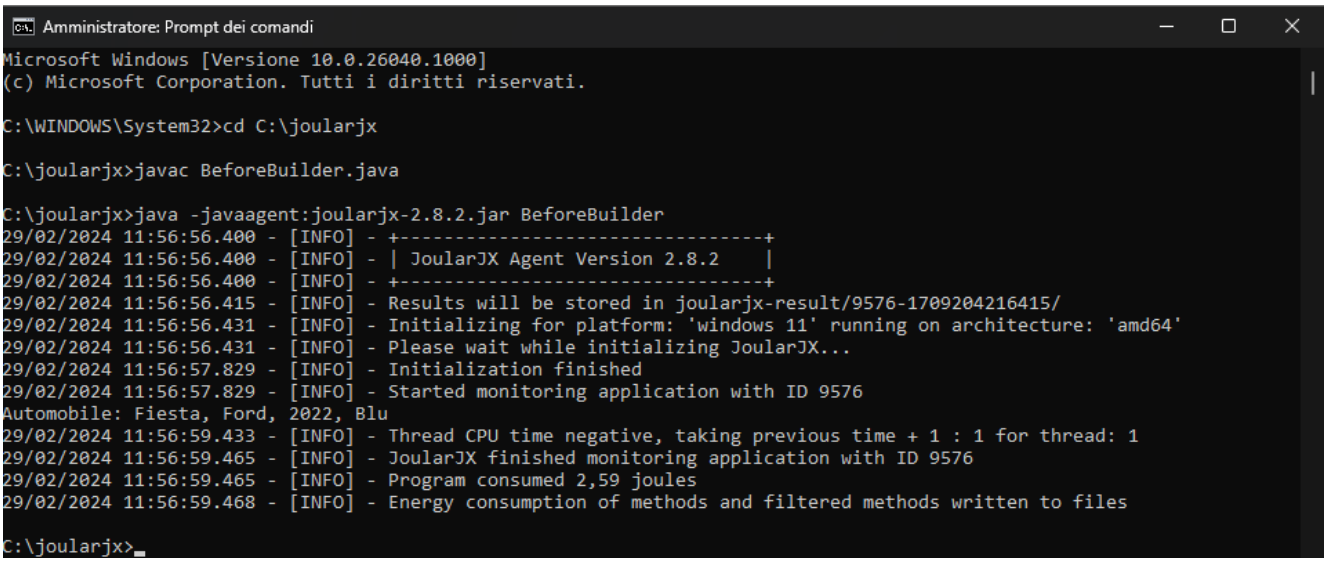
Avendo ottenuto il corrispondente file class si può provvedere al collegamento di JoularJX con la Java Virtual Machine all'avvio della classe principale del programma Java:

java -javaagent:joularjx-$version.jar nomefile

Da precisare che al posto di *$version* deve essere scritta la versione che si sta usufruendo di JoularJX.
Se il programma che si sta analizzando è un file JAR, lo si può eseguire ed analizzare con JoularJX aggiungendo un flag nell'ultima istruzione esplicitata:

java -javaagent:joularjx-$version.jar -jar nomefile

Un esempio pratico dell'esecuzione di tali istruzioni sono riportati nel seguente screen:



Lo screen illustra bene la sequenza delle varie operazioni. In particolare, ci si vuole soffermare sull'output dell'ultima istruzione: viene avviato JoularJX, viene mostrato il pathname della directory all'interno della quale verranno salvati i valori di potenza dei metodi del programma esaminato, si visualizza l'architettura della macchina su cui si sta lavorando, il PID del programma Java in esecuzione analizzato, il suo ouput e, infine, il consumo enrgetico in Joule complessivo.

Risultati sperimentali delle tecniche di OOP

In questa sezione vengono mostrati i risultati sperimentali delle sole tecniche OOP considerate nelle sezioni precedenti. Si osserva che, si effettuerà un'analisi sia di quando la tecnica (e successivamente anche per i Design Pattern) viene richiamata una sola volta nel programma, sia quando tutto il codice nel *main()* viene eseguito per 10,100,1000,10000,100000 e 1000000 volte. Tutto ciò per avere un feedback completo sulle performance energetiche delle varie pratiche di programmazione analizzate.

Ereditarietà

Una prima tecnica esaminata è l'ereditarietà. Il primo programma consuma 1,25 Joule di energia non sfruttando l'ereditarietà.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeInheritance
18/02/2024 08:22:21.502 - [INFO] - +---------------------------------+
18/02/2024 08:22:21.502 - [INFO] - | JoularJX Agent Version 2.8.2    |
18/02/2024 08:22:21.502 - [INFO] - +---------------------------------+
18/02/2024 08:22:21.517 - [INFO] - Results will be stored in joularjx-result/10416-1708240941517/
18/02/2024 08:22:21.533 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
18/02/2024 08:22:21.533 - [INFO] - Please wait while initializing JoularJX...
18/02/2024 08:22:22.981 - [INFO] - Initialization finished
18/02/2024 08:22:22.981 - [INFO] - Started monitoring application with ID 10416
18/02/2024 08:22:24.626 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
18/02/2024 08:22:24.659 - [INFO] - JoularJX finished monitoring application with ID 10416
18/02/2024 08:22:24.659 - [INFO] - Program consumed 1,09 joules
18/02/2024 08:22:24.659 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Il secondo programma, invece, consuma più energia (4.58 Joule).

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterInheritance
23/02/2024 12:21:23.480 - [INFO] - +---------------------------------+
23/02/2024 12:21:23.480 - [INFO] - | JoularJX Agent Version 2.8.2    |
23/02/2024 12:21:23.480 - [INFO] - +---------------------------------+
23/02/2024 12:21:23.496 - [INFO] - Results will be stored in joularjx-result/32616-1708687283496/
23/02/2024 12:21:23.496 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
23/02/2024 12:21:23.496 - [INFO] - Please wait while initializing JoularJX...
23/02/2024 12:21:24.916 - [INFO] - Initialization finished
23/02/2024 12:21:24.916 - [INFO] - Started monitoring application with ID 32616
23/02/2024 12:21:26.557 - [INFO] - JoularJX finished monitoring application with ID 32616
23/02/2024 12:21:26.557 - [INFO] - Program consumed 4,58 joules
23/02/2024 12:21:26.557 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Facendo un'analisi anche sui casi in cui vengono iterati i due programmi per più volte, si ottengono le seguenti due tabelle di riferimento. La prima riguarda il programma *BeforeInheritance.java:*

	Consumo Energetico (Joule)
1 Iterazione	**1.25**
10 Iterazioni	**2.81**
100 Iterazioni	**0.75**
1000 Iterazioni	**2.04**
10000 Iterazioni	**1.02**
100000 Iterazioni	**1.51**
10000000 Iterazioni	**2.08**

Dai risultati acquisiti si ottiene come media 1.6 Joule e come deviazione standard (data dalla radice quadrata con segno positivo della varianza) 0.7 Joule.
Per il programma *AfterInheritance.java* si ha la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**4.58**
10 Iterazioni	**2.28**
100 Iterazioni	**0.36**
1000 Iterazioni	**1.18**
10000 Iterazioni	**2.35**
100000 Iterazioni	**1.55**
10000000 Iterazioni	**1.45**

Facendo la media si ottiene come valore 1,9 Joule e, invece, la deviazione standard è pari a 1.4 Joule.

Polimorfismo

Un'altra tecnica che è stata considerata è l'override. Il primo programma, che non ne fa uso, consuma 2.85 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeOverride
18/02/2024 07:54:30.344 - [INFO] - +---------------------------------+
18/02/2024 07:54:30.344 - [INFO] - | JoularJX Agent Version 2.8.2    |
18/02/2024 07:54:30.345 - [INFO] - +---------------------------------+
18/02/2024 07:54:30.359 - [INFO] - Results will be stored in joularjx-result/11200-1708239270357/
18/02/2024 07:54:30.373 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
18/02/2024 07:54:30.374 - [INFO] - Please wait while initializing JoularJX...
18/02/2024 07:54:31.804 - [INFO] - Initialization finished
18/02/2024 07:54:31.805 - [INFO] - Started monitoring application with ID 11200
Ciao! Sono la classe Parent!
Ciao! Sono la classe Child!
18/02/2024 07:54:33.441 - [INFO] - JoularJX finished monitoring application with ID 11200
18/02/2024 07:54:33.442 - [INFO] - Program consumed 2,85 joules
18/02/2024 07:54:33.448 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Il secondo programma, usufruendo dell'override, non solo sfrutta i vantaggi di tale tecnica dal punto di vista funzionale e implementativo ma risparmia anche dal puntodi vista energetico. Infatti, vengono consumati 1,8 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterOverride
18/02/2024 07:59:36.967 - [INFO] - +---------------------------------+
18/02/2024 07:59:36.967 - [INFO] - | JoularJX Agent Version 2.8.2    |
18/02/2024 07:59:36.968 - [INFO] - +---------------------------------+
18/02/2024 07:59:36.982 - [INFO] - Results will be stored in joularjx-result/9172-1708239576980/
18/02/2024 07:59:36.996 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
18/02/2024 07:59:36.997 - [INFO] - Please wait while initializing JoularJX...
18/02/2024 07:59:38.419 - [INFO] - Initialization finished
18/02/2024 07:59:38.422 - [INFO] - Started monitoring application with ID 9172
Ciao! Sono la classe Parent!
Ciao! Sono la classe Child!
18/02/2024 07:59:40.064 - [INFO] - JoularJX finished monitoring application with ID 9172
18/02/2024 07:59:40.064 - [INFO] - Program consumed 1,8 joules
18/02/2024 07:59:40.071 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Analizzando gli stessi programmi iterati si ottengono i seguenti risultati.
Per quanto riguarda il programma *BeforeOverride.java* si ha la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**2.85**
10 Iterazioni	**1.35**
100 Iterazioni	**3.12**
1000 Iterazioni	**1.04**
10000 Iterazioni	**7.15**
100000 Iterazioni	**22.09**
10000000 Iterazioni	**247.05**

Dati i seguenti valori, la media risulta essere pari a 40.6 Joule, mentre la deviazione standard è pari a 91.3 Joule.

Per quanto riguarda il secondo programma (*AfterOverride.java*) iterato si ottengono i seguenti esiti:

	Consumo Energetico (Joule)
1 Iterazione	**1.8**
10 Iterazioni	**1.96**
100 Iterazioni	**1.42**
1000 Iterazioni	**1.75**
10000 Iterazioni	**6.11**
100000 Iterazioni	**19.47**
10000000 Iterazioni	**153.74**

Dai seguenti valori si ottiene una media pari a 26.6 Joule e una deviazione standard pari a 56.43 Joule.

Overload degli operatori

Il primo programma, come si diceva nella sezione dedicata a tale tecnica di OOP, non sfrutta l'overload degli operatori; richiama, infatti, metodi della libreria Math per poter effettuare le singole operazioni semplici matematiche per poi risolvere l'intera espressione. Il programma consuma 4,31 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeOverloadOperatori
17/02/2024 04:35:43.561 - [INFO] - +---------------------------------+
17/02/2024 04:35:43.561 - [INFO] - | JoularJX Agent Version 2.8.2    |
17/02/2024 04:35:43.561 - [INFO] - +---------------------------------+
17/02/2024 04:35:43.577 - [INFO] - Results will be stored in joularjx-result/11512-1708184143577/
17/02/2024 04:35:43.593 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
17/02/2024 04:35:43.593 - [INFO] - Please wait while initializing JoularJX...
17/02/2024 04:35:44.991 - [INFO] - Initialization finished
17/02/2024 04:35:44.991 - [INFO] - Started monitoring application with ID 11512
0
17/02/2024 04:35:46.654 - [INFO] - JoularJX finished monitoring application with ID 11512
17/02/2024 04:35:46.655 - [INFO] - Program consumed 4,31 joules
17/02/2024 04:35:46.660 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Il secondo programma, che sfrutta tale tecnica, è molto più compatto e consuma 2,42 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterOverloadOperatori
18/02/2024 07:45:01.921 - [INFO] - +--------------------------------+
18/02/2024 07:45:01.922 - [INFO] - | JoularJX Agent Version 2.8.2    |
18/02/2024 07:45:01.923 - [INFO] - +--------------------------------+
18/02/2024 07:45:01.938 - [INFO] - Results will be stored in joularjx-result/17920-1708238701937/
18/02/2024 07:45:01.950 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
18/02/2024 07:45:01.951 - [INFO] - Please wait while initializing JoularJX...
18/02/2024 07:45:03.490 - [INFO] - Initialization finished
18/02/2024 07:45:03.492 - [INFO] - Started monitoring application with ID 17920
0
18/02/2024 07:45:05.138 - [INFO] - JoularJX finished monitoring application with ID 17920
18/02/2024 07:45:05.139 - [INFO] - Program consumed 2,42 joules
18/02/2024 07:45:05.146 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Usufruendo degli stessi programmi ma iterati si ottengono i seguenti esiti.
Per il primo programma (*BeforeOverloadOperatori.java*) si ottiene la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**0.36**
10 Iterazioni	**1.73**
100 Iterazioni	**2.87**
1000 Iterazioni	**5.86**
10000 Iterazioni	**3.89**
100000 Iterazioni	**8.5**
10000000 Iterazioni	**75.6**

Da tali valori si ottiene una media pari a 14.1 Joule e una deviazione standard pari a 27.24 Joule.

Per quanto riguarda il secondo programma (che fa uso della tecnica dell'overload degli operatori) chiamato *AfterOverloadOperatori.java* si ha la seguente tabella dei valori:

	Consumo Energetico (Joule)
1 Iterazione	**2.42**
10 Iterazioni	**0.65**
100 Iterazioni	**0.19**
1000 Iterazioni	**1.27**
10000 Iterazioni	**2.77**
100000 Iterazioni	**6.42**
10000000 Iterazioni	**42.55**

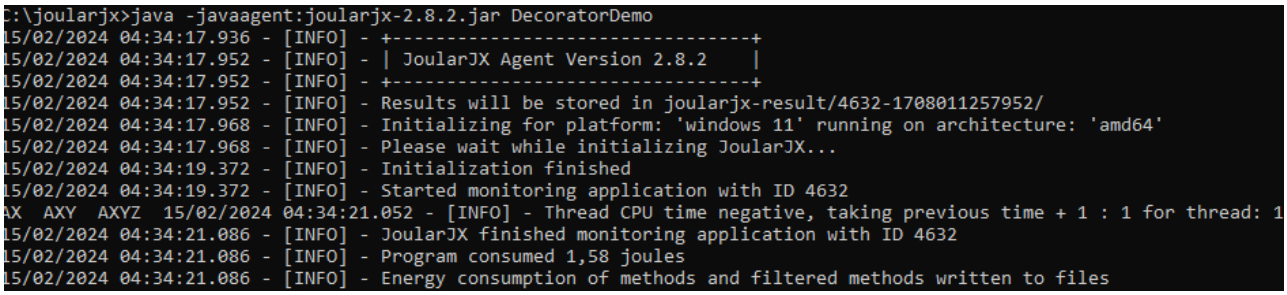
Da tali valori si ottiene una media pari a 8.03 Joule e una deviazione standard pari a 15.35 Joule.

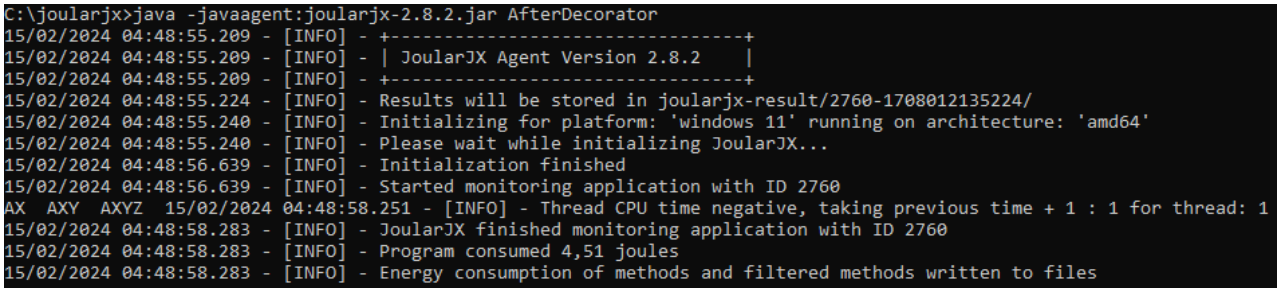
Risultati sperimentali dei Design Pattern

Di seguito i risultati ottenuti dai vari programmi usufruiti per l'analisi sui Design Pattern sopra citati.

Decorator

Come si può notare dalle immagini seguenti, il programma che non utilizza tale Design Pattern sfrutta 1,59 Joule mentre il programma che sfrutta tale pattern fa uso di 4,51 Joule di energia. Come si può intuire, tale Design Pattern consuma 2,8 volte più energia. Una possibile ragione è l'uso della modalità decorativa, che aumenta il numero di classi decorative specifiche e genera tanti piccoli oggetti.





Situazione differente la si ha nel momento in cui si itera i *main()* per più volte.
Per quanto riguarda il primo programma (*BeforeDecorator.java*) si ha la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**1.58**
10 Iterazioni	**1.29**
100 Iterazioni	**2.71**
1000 Iterazioni	**4.31**
10000 Iterazioni	**5.77**
100000 Iterazioni	**74.15**
10000000 Iterazioni	**736.67**

Da tali risultati sperimentali si ottiene una media complessiva di 195.04 Joule e una

deviazione standard pari a 274.06 Joule.
Per quanto riguarda il secondo programma (*AfterDecorator.java*) si hanno i seguenti risultati sperimentali:

	Consumo Energetico (Joule)
1 Iterazione	**4.51**
10 Iterazioni	**1.14**
100 Iterazioni	**1.73**
1000 Iterazioni	**4.14**
10000 Iterazioni	**5.93**
100000 Iterazioni	**87.49**
10000000 Iterazioni	**674.22**

Dai risultati poc'anzi menzionati si ottiene una media pari a 201.25 Joule e la deviazione standard pari a 250.19 Joule.

Builder

Il primo programma, non facendo uso del pattern, consuma 3.79 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeBuilder
24/02/2024 08:40:51.632 - [INFO] - +---------------------------------+
24/02/2024 08:40:51.633 - [INFO] - | JoularJX Agent Version 2.8.2    |
24/02/2024 08:40:51.633 - [INFO] - +---------------------------------+
24/02/2024 08:40:51.648 - [INFO] - Results will be stored in joularjx-result/14268-1708760451645/
24/02/2024 08:40:51.658 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
24/02/2024 08:40:51.659 - [INFO] - Please wait while initializing JoularJX...
24/02/2024 08:40:53.070 - [INFO] - Initialization finished
24/02/2024 08:40:53.072 - [INFO] - Started monitoring application with ID 14268
Automobile: Fiesta, Ford, 2022, Blu
24/02/2024 08:40:54.665 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
24/02/2024 08:40:54.691 - [INFO] - JoularJX finished monitoring application with ID 14268
24/02/2024 08:40:54.692 - [INFO] - Program consumed 3,79 joules
24/02/2024 08:40:54.700 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Il secondo programma consuma 3.39 Joule: una differenza assolutamente minima rispetto al consumo del primo programma.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterBuilder
24/02/2024 08:51:24.478 - [INFO] - +---------------------------------+
24/02/2024 08:51:24.479 - [INFO] - | JoularJX Agent Version 2.8.2    |
24/02/2024 08:51:24.479 - [INFO] - +---------------------------------+
24/02/2024 08:51:24.494 - [INFO] - Results will be stored in joularjx-result/14364-1708761084493/
24/02/2024 08:51:24.505 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
24/02/2024 08:51:24.507 - [INFO] - Please wait while initializing JoularJX...
24/02/2024 08:51:25.910 - [INFO] - Initialization finished
24/02/2024 08:51:25.911 - [INFO] - Started monitoring application with ID 14364
Automobile: Fiesta, Ford, 2022, Blu
24/02/2024 08:51:27.490 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
24/02/2024 08:51:27.518 - [INFO] - JoularJX finished monitoring application with ID 14364
24/02/2024 08:51:27.519 - [INFO] - Program consumed 3,39 joules
24/02/2024 08:51:27.526 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Facendo la stessa analisi fatta per tutti gli altri pattern si ottengono i seguenti risultati: per quanto riguarda il programma *BeforeBuilder.java* si ha la seguente tabella…

	Consumo Energetico (Joule)
1 Iterazione	**3.79**
10 Iterazioni	**0.93**
100 Iterazioni	**1.11**
1000 Iterazioni	**1.58**
10000 Iterazioni	**4.02**
100000 Iterazioni	**7.09**
10000000 Iterazioni	**67.6**

Dai seguenti esiti sperimentali si ottiene una media complessiva pari a 12.3 Joule e una deviazione standard pari a 24.48 Joule.
La stessa analisi la si effettua per il secondo programma (*AfterBuilder.java*). Si ottiene la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**3.39**
10 Iterazioni	**1.58**
100 Iterazioni	**1.28**
1000 Iterazioni	**3.08**
10000 Iterazioni	**3.05**
100000 Iterazioni	**9.62**
10000000 Iterazioni	**72.74**

Da tali esiti si ottiene una media di 13.53 Joule e una deviazione standard pari a 26.25 Joule.

Adapter

Il primo programma necessita di un adattatore in quanto l'interfaccia tra gli oggetti Linea e Rettangolo è incompatibile e l'utente dovrebbe recuperare il tipo di ciascuna forma e fornire manualmente gli argomenti corretti. Tale programma sfrutta 1,61 Joule di energia.

Il secondo programma elimina le incongruenze. Infatti, il "livello extra di reindirizzamento indiretto" dell'adattatore si occupa di mappare un'interfacciacomune di facile utilizzo su interfacce peculiari specifiche dell'ereditarietà.
Nonostante ciò, l'utilizzo di tale pattern richiede più energia del primo programma, usufruendo di 3,63 Joule.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeAdapter
15/02/2024 05:07:54.683 - [INFO] - +---------------------------------+
15/02/2024 05:07:54.683 - [INFO] - | JoularJX Agent Version 2.8.2    |
15/02/2024 05:07:54.683 - [INFO] - +---------------------------------+
15/02/2024 05:07:54.699 - [INFO] - Results will be stored in joularjx-result/7524-1708013274699/
15/02/2024 05:07:54.715 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
15/02/2024 05:07:54.715 - [INFO] - Please wait while initializing JoularJX...
15/02/2024 05:07:56.119 - [INFO] - Initialization finished
15/02/2024 05:07:56.119 - [INFO] - Started monitoring application with ID 7524
Line from point A(10;20), to point B(30;60)
Rectangle with coordinate left-down point (30;60), width: 40, height: 40
15/02/2024 05:07:57.730 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
15/02/2024 05:07:57.762 - [INFO] - JoularJX finished monitoring application with ID 7524
15/02/2024 05:07:57.762 - [INFO] - Program consumed 1,61 joules
15/02/2024 05:07:57.762 - [INFO] - Energy consumption of methods and filtered methods written to files
```

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterAdapter
15/02/2024 05:14:50.941 - [INFO] - +---------------------------------+
15/02/2024 05:14:50.941 - [INFO] - | JoularJX Agent Version 2.8.2    |
15/02/2024 05:14:50.942 - [INFO] - +---------------------------------+
15/02/2024 05:14:50.956 - [INFO] - Results will be stored in joularjx-result/15876-1708013690954/
15/02/2024 05:14:50.968 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
15/02/2024 05:14:50.969 - [INFO] - Please wait while initializing JoularJX...
15/02/2024 05:14:52.379 - [INFO] - Initialization finished
15/02/2024 05:14:52.380 - [INFO] - Started monitoring application with ID 15876
Rectangle with coordinate left-down point (10;20), width: 20, height: 40
Line from point A(10;20), to point B(30;60)
15/02/2024 05:14:53.980 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
15/02/2024 05:14:54.007 - [INFO] - JoularJX finished monitoring application with ID 15876
15/02/2024 05:14:54.009 - [INFO] - Program consumed 3,63 joules
15/02/2024 05:14:54.017 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Come fatto per gli altri pattern, si effettua un'analisi anche sui programmi che c possiedono più iterazioni. Per quanto riguarda il primo programma (*BeforeAdapter.java*) si ottiene la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**0.75**
10 Iterazioni	**1.18**
100 Iterazioni	**1.53**
1000 Iterazioni	**2.86**
10000 Iterazioni	**4.71**
100000 Iterazioni	**13.98**
10000000 Iterazioni	**114.66**

Da tali esiti si ottiene una media pari a 19.95 Joule e una deviazione standard pari a 42.01 Joule.

Per quanto riguarda il secondo programma (*AfterAdapter.java*) si ha la seguente tabella di risultati:

	Consumo Energetico (Joule)
1 Iterazione	**3.63**
10 Iterazioni	**2.8**
100 Iterazioni	**1.87**
1000 Iterazioni	**1.47**
10000 Iterazioni	**6.43**
100000 Iterazioni	**12.24**
10000000 Iterazioni	**125.05**

Dagli esperimenti effettuati si ottiene una media pari a 21,9 Joule e una deviazione standard pari a 42.24 Joule.

Composite

Il primo programma, che non fa uso di tale pattern, sfrutta 3,1 Joule di energia.

Il secondo programma, che ne fa uso, sfrutta, invece, 5,83 Joule di energia, 1.8 di più. La motivazione potrebbe essere dettata dal fatto che si vuole gestire una raccolta eterogenea di oggetti in modo atomico (o trasparente) attraverso delle aggregazioni. Dovendo, poi, effettuare delle operazioni specifiche su tale aggregazione, per raggiungere il risultato finale, il tutto costa in termini energetici.

```
PS C:\joularjx> java -javaagent:joularjx-2.8.2.jar BeforeComposite
15/02/2024 05:50:57.087 - [INFO] - +---------------------------------+
15/02/2024 05:50:57.087 - [INFO] - | JoularJX Agent Version 2.8.2    |
15/02/2024 05:50:57.087 - [INFO] - +---------------------------------+
15/02/2024 05:50:57.102 - [INFO] - Results will be stored in joularjx-result/11408-1708015857102/
15/02/2024 05:50:57.118 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'a
md64'
15/02/2024 05:50:57.118 - [INFO] - Please wait while initializing JoularJX...
15/02/2024 05:50:58.637 - [INFO] - Initialization finished
15/02/2024 05:50:58.637 - [INFO] - Started monitoring application with ID 11408
MUSIC
   Don't wary, be happy.mp3
   SCORPIONS
      Wind of change.mp3
      Big city night.mp3
      DIO
         Rainbow in the dark.mp3
   track2.m3u
15/02/2024 05:51:00.291 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread:
1
15/02/2024 05:51:00.327 - [INFO] - JoularJX finished monitoring application with ID 11408
15/02/2024 05:51:00.327 - [INFO] - Program consumed 3,1 joules
15/02/2024 05:51:00.327 - [INFO] - Energy consumption of methods and filtered methods written to files
```

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterComposite
21/02/2024 04:08:52.670 - [INFO] - +---------------------------------+
21/02/2024 04:08:52.670 - [INFO] - | JoularJX Agent Version 2.8.2    |
21/02/2024 04:08:52.671 - [INFO] - +---------------------------------+
21/02/2024 04:08:52.695 - [INFO] - Results will be stored in joularjx-result/15080-1708528132693/
21/02/2024 04:08:52.707 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
21/02/2024 04:08:52.708 - [INFO] - Please wait while initializing JoularJX...
21/02/2024 04:08:54.161 - [INFO] - Initialization finished
21/02/2024 04:08:54.162 - [INFO] - Started monitoring application with ID 15080
MUSIC
   Don't wary, be happy.mp3
   SCORPIONS
      Wind of change.mp3
      Big city night.mp3
      DIO
         Rainbow in the dark.mp3
   track2.m3u
21/02/2024 04:08:55.758 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
21/02/2024 04:08:55.795 - [INFO] - JoularJX finished monitoring application with ID 15080
21/02/2024 04:08:55.796 - [INFO] - Program consumed 2,77 joules
21/02/2024 04:08:55.805 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Facendo la stessa analisi dei pattern precedenti, si ottiene per il primo programma (*BeforeComposite.java*) la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**3.1**
10 Iterazioni	**1.42**
100 Iterazioni	**1.27**
1000 Iterazioni	**4.06**
10000 Iterazioni	**6.76**
100000 Iterazioni	**62.25**
10000000 Iterazioni	**536.68**

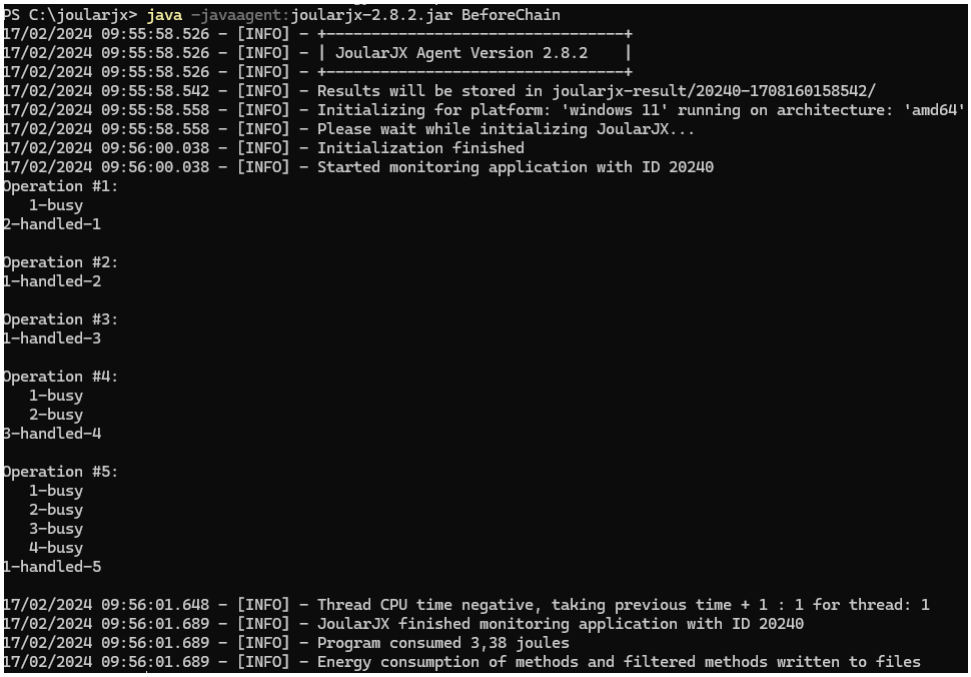
Dai risultati ottenuti si ha una media di 87.72 Joule e una deviazione standard pari a 199.20 Joule.
Per il secondo programma (*AfterComposite.java*) si ha il seguente scenario:

	Consumo Energetico (Joule)
1 Iterazione	**2.77**
10 Iterazioni	**0.64**
100 Iterazioni	**1.22**
1000 Iterazioni	**6.16**
10000 Iterazioni	**6.85**
100000 Iterazioni	**69.16**
10000000 Iterazioni	**528.09**

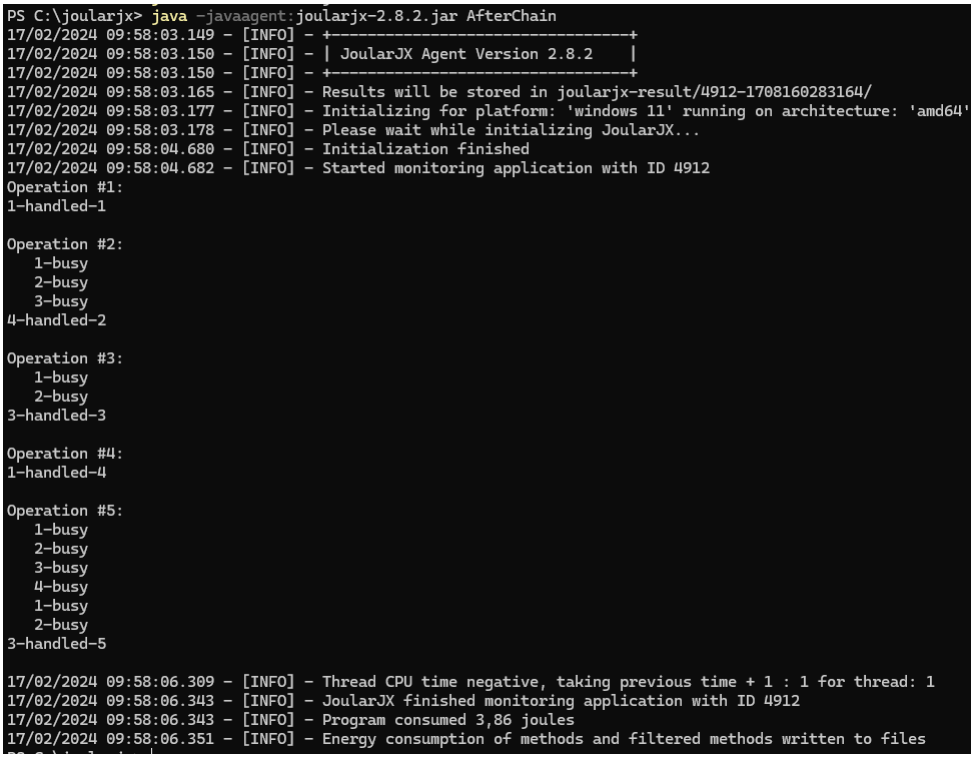
Da tali esiti si ottiene una media complessiva di 87.84 Joule e una deviazione standard pari a 195.67.

Chain of Responsability

Nel primo programma il Client è responsabile dello scorrimento di oggetti Handler edi determinare quando la richiesta è stata effettivamente gestita. Tale programma sfrutta 3,38 Joule di energia.



Nel secondo programma, invece, il client sottopone ogni richiesta all'astrazione "Chain" ed è disaccoppiato da tutte le elaborazioni. Questo fa sì che non ci siano troppi sprechi di energia. Infatti, il secondo programma fa uso di 3,86 Joule di energia. Si può dire che la differenza energetica sia del tutto trascurabile.



Facendo gli stessi esperimenti sugli stessi programmi si ottengono le seguenti tabelle:

	Consumo Energetico (Joule)
1 Iterazione	**3.38**
10 Iterazioni	**0.45**
100 Iterazioni	**1.81**
1000 Iterazioni	**3.75**
10000 Iterazioni	**12.08**
100000 Iterazioni	**179.03**
10000000 Iterazioni	**1996.44**

Da tali dati si ottiene una media pari a 313.84 Joule e una deviazione standard pari a 744.81.

Per quanto riguarda la versione ‘After’ si ha la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**3.86**
10 Iterazioni	**1.98**
100 Iterazioni	**2.47**
1000 Iterazioni	**6.74**
10000 Iterazioni	**21.67**
100000 Iterazioni	**215.03**
10000000 Iterazioni	**2131.72**

Da tali risultati si ottiene una media pari a 340.49 Joule e una deviazione standard pari a 793.66.

State

Il primo programma, non usufruendo del Pattern State, sfrutta 2.07 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar BeforeState
02/03/2024 12:31:31.283 - [INFO] - +---------------------------------+
02/03/2024 12:31:31.284 - [INFO] - | JoularJX Agent Version 2.8.2    |
02/03/2024 12:31:31.284 - [INFO] - +---------------------------------+
02/03/2024 12:31:31.299 - [INFO] - Results will be stored in joularjx-result/23224-1709379091296/
02/03/2024 12:31:31.310 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
02/03/2024 12:31:31.310 - [INFO] - Please wait while initializing JoularJX...
02/03/2024 12:31:32.746 - [INFO] - Initialization finished
02/03/2024 12:31:32.748 - [INFO] - Started monitoring application with ID 23224
Press ENTER









low speed
02/03/2024 12:31:34.340 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
02/03/2024 12:31:34.379 - [INFO] - JoularJX finished monitoring application with ID 23224
02/03/2024 12:31:34.379 - [INFO] - Program consumed 2,07 joules
02/03/2024 12:31:34.388 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Il secondo programma, usufruendo del pattern State, sfrutta un quantitativo di energia pari a 3.7 Joule di energia.

```
C:\joularjx>java -javaagent:joularjx-2.8.2.jar AfterState
02/03/2024 03:22:12.512 - [INFO] - +---------------------------------+
02/03/2024 03:22:12.514 - [INFO] - | JoularJX Agent Version 2.8.2    |
02/03/2024 03:22:12.514 - [INFO] - +---------------------------------+
02/03/2024 03:22:12.527 - [INFO] - Results will be stored in joularjx-result/6976-1709389332526/
02/03/2024 03:22:12.539 - [INFO] - Initializing for platform: 'windows 11' running on architecture: 'amd64'
02/03/2024 03:22:12.540 - [INFO] - Please wait while initializing JoularJX...
02/03/2024 03:22:13.953 - [INFO] - Initialization finished
02/03/2024 03:22:13.954 - [INFO] - Started monitoring application with ID 6976
Press ENTER
low speed
02/03/2024 03:22:15.572 - [INFO] - Thread CPU time negative, taking previous time + 1 : 1 for thread: 1
02/03/2024 03:22:15.609 - [INFO] - JoularJX finished monitoring application with ID 6976
02/03/2024 03:22:15.609 - [INFO] - Program consumed 3,7 joules
02/03/2024 03:22:15.617 - [INFO] - Energy consumption of methods and filtered methods written to files
```

Anche qui si cerca di fare un'analisi completa anche sulle versioni iterate dei due programmi. In particolare, per il primo programma si ottiene la seguente tabella:

	Consumo Energetico (Joule)
1 Iterazione	**2.07**
10 Iterazioni	**3.81**
100 Iterazioni	**6.88**
1000 Iterazioni	**34.18**
10000 Iterazioni	**241.99**
100000 Iterazioni	**547.56**
10000000 Iterazioni	**3333.02**

Da tali esiti si ottiene una media pari a 595.64 Joule e una deviazione standard pari a 1223.73 Joule.
Per quanto riguarda il secondo programma si ottiene:

	Consumo Energetico (Joule)
1 Iterazione	**3.7**
10 Iterazioni	**2.02**
100 Iterazioni	**8.71**
1000 Iterazioni	**46.16**
10000 Iterazioni	**300.33**
100000 Iterazioni	**950.05**
10000000 Iterazioni	**4625.21**

Da questi ultimi esiti di ottiene una media pari a 848,02 Joule e una deviazione standard pari a 1700.66 Joule.

Conclusione e Sviluppi Futuri

Con i linguaggi di programmazione orientati agli oggetti (OOP) che dominano la progettazione di software commerciale e di app mobili, diventa sempre più importante avere una solida conoscenza di come l'OOP influisce sulle prestazioni e sull'efficienza energetica. Questa relazione conduce un'analisi empirica sull'impatto di diverse tecniche di programmazione orientate agli oggetti e di Design Pattern sulle prestazioni del software e sull'efficienza energetica.

Si notano diversi esiti sotto il punto di vista energetico delle varie pratiche di OOP e dei Design Pattern.

Dagli esiti delle sperimentazioni effettuate si nota come il fenomeno considerato non sia così banale da trattare e scontato. Da notare che, nella grande maggioranza dei casi considerati, il consumo energetico dipende abbondantemente dal numero di iterazioni effettuate.

In generale, analizzando e confrontando le medie e le deviazioni standard calcolate nelle varie sezioni, si osserva che:

- l'ereditarietà degrada le prestazioni; si nota un aumento di 2 volte la deviazione standard;
- l'override risulta essere una tecnica performante dal punto di vista energetico risparmiando, mediamente, circa la metà dell'energia consumata con il corrispondente programma che non fa uso di tale tecnica;
- anche l'overload degli operatori è una tecnica molto efficiente; risparmia anch'essa circa la metà del consumo energetico del programma in versione "Before";
- Il pattern Decorator degrada le perfomance rispetto alla versione "Before". C'è una differenza media di 72,02 Joule di energia. Una possibile ragione è l'uso della modalità decorativa, che aumenta il numero di classi decorative specifiche e genera tanti piccoli oggetti.
- Il pattern Builder degrada leggermente le performance. Si ha, nello specifico, un degrado medio di 1,23 Joule. Tale differenza può essere considerata come trascurabile per gli studi condotti.
- Il pattern Adapter degrada, anch'esso, leggermente le perfomance, con un aumento medio di 1.95 Joule.
- Anche il pattern Composite ha un consumo energetico trascurabile rispetto alla versione "Before" non possedente tale tecnica. C'è un consumo medio leggermente superiore di circa 0,14 Joule.
- Il pattern Chain of Responsability aumenta il consumo energetico arrivando a

consumare, mediamente, circa 27 Joule in più.

- Il pattern State, secondo le analisi effettuate, ha un fortissimo impatto sulle perfomance energetiche, consumando, mediamente, più di 250 Joule di energia in più.

C'è da sottolineare un ulteriore aspetto: osservando in maniera scrupolosa i risultati ottenuti nelle varie tabelle, si nota spesso che, il consumo di energia avuto nel momento in cui si esegue con una sola iterazione un programma, si ha un consumo anomalo maggiore rispetto a quando, invece, lo stesso programma viene eseguito 10 o, addirittura, 100 volte attraverso le iterazioni dei cicli for. Questo aspetto porta alla conclusione che, seppur tale studio può contribuire allo stato dell'arte della programmazione orientata agli oggetti riguardo gli impatti che varie tecniche hanno dal punto di vista energetico, si deve, comunque, sottolineare il fatto che:

- il tool utilizzato potrebbe non essere estremamente accurato nell'analisi empirica descritta;
- i risultati potrebbero dipendere dal caso o da altri elementi che non si è riusciti a controllare;
- le relazioni ondivaghe tra i valori misurati con e senza pattern e tecniche di OOP non permettono di affermare, con la massima certezza, che alcuno di tali pratiche produca miglioramenti o peggioramenti in generale dal punto di vista energetico.

Tra gli sviluppi futuri, che si intende sottolineare, dello studio proposto da tale relazione, c'è sicuramente quello di estendere tale analisi empirica:

- cercando, innanzitutto, altri tool che permettano di migliorare i risultati sperimentali proposti da tale tesina;
- considerando anche gli altri Design Pattern esistenti e le altre tecniche maggiormente diffuse nell'ambito OOP.

Bibliografia

(s.d.). Tratto da https://www.lowcodeitalia.it/articoli/quando-vanno-in-pensione-i-programmatori-i-35-anni-sono-la-fine#:~:text=Secondo%20i%20dati%20di%20Evans,li%20superer%C3%A0%20entro%20il%202024.

developer.android. (2023). *developer.android*. Tratto da developer.android.com.

geeksforgeeks.org. (2023). Tratto da geeksforgeeks.org.

html.it. (2015). Tratto da www.html.it: https://www.html.it/pag/51817/polimorfismo-in-java/

lowcodeitalia. (s.d.). Tratto da lowcodeitalia.it: https://www.lowcodeitalia.it/articoli/quando-vanno-in-pensione-i-programmatori-i-35-anni-sono-la-fine#:~:text=Secondo%20i%20dati%20di%20Evans,li%20superer%C3%A0%20entro%20il%202024.

lowcodeitalia. (2021). *lowcodeitalia*. Tratto da lowcodeitalia.it: https://www.lowcodeitalia.it/articoli/quando-vanno-in-pensione-i-programmatori-i-35-anni-sono-la-fine#:~:text=Secondo%20i%20dati%20di%20Evans,li%20superer%C3%A0%20entro%20il%202024.

skuola.net. (2016). Tratto da skuola.net: https://www.skuola.net/informatica/java-ereditarieta.html#:~:text=L'ereditariet%C3%A0%20avviene%20grazie%20al,non%20hanno%20limiti%20di%20implementazione).