

Domotic System Management

Autori:

- Andrea Penazzi
- Pavneesh Singh



Indice

- Principio di separazione M-V 3-5
- GRASP 6-7
 - ◊ Information expert 8-9
 - ◊ Creator 10-11
- SOLID 12
 - ◊ Interface Segregation Principle 13-14
 - ◊ Open Closed Principle 15-16
- GoF 17-19
 - ◊ DAO 20
 - ◊ Simple factory 21-22
- Testing 23-24
- Refactoring 25
 - ◊ Extract method 26-27

Principio Model-View

Principio per cui «gli oggetti non UI non devono essere accoppiati direttamente agli oggetti UI», ovvero non si deve mettere la logica applicativa nei metodi di un oggetto dell’interfaccia utente.

Applicazione in Domotic-system-management: eliminazione dei metodi `toString()` all’interno delle classi contenenti la logica del programma; creazione di classi adibite alla presentazione (stampa) delle classi già create.

Esempio: Manutentore non ha più un metodo `toString()`, ma esiste una classe `ClasseDiServizioManutentore` adibita alla sua presentazione.

```

/**
 * Visualizza la descrizione dei sensori delle categorie sensori
 *
 * @param listaCategorie scelta
 * @return la descrizione delle categorie sensori
 */
public static String visualizzaDescrizioneCatergorieSensori(ListaCategorie listaCategorie)
    return ClasseDiServizioListaCategorie.descrizioneCategorieSensori(listaCategorie);
}

/**
 * Visualizza la descrizione degli attuatori delle categorie attuatori
 *
 * @param listaCategorie scelta
 * @return la descrizione delle categorie attuatori
 */
public static String visualizzaDescrizioneCatergorieAttuatori(ListaCategorie listaCategorie)
    return ClasseDiServizioListaCategorie.descrizioneCategorieAttuatori(listaCategorie);
}

/**
 * Descrizione stanze
 *
 * @param unitaImmobiliare da cui prendere le stanze
 */
public static void descrizioneStanze(UnitaImmobiliare unitaImmobiliare) {
    ClasseDiServizioUnitaImmobiliare.descrizioneStanze(unitaImmobiliare);
}

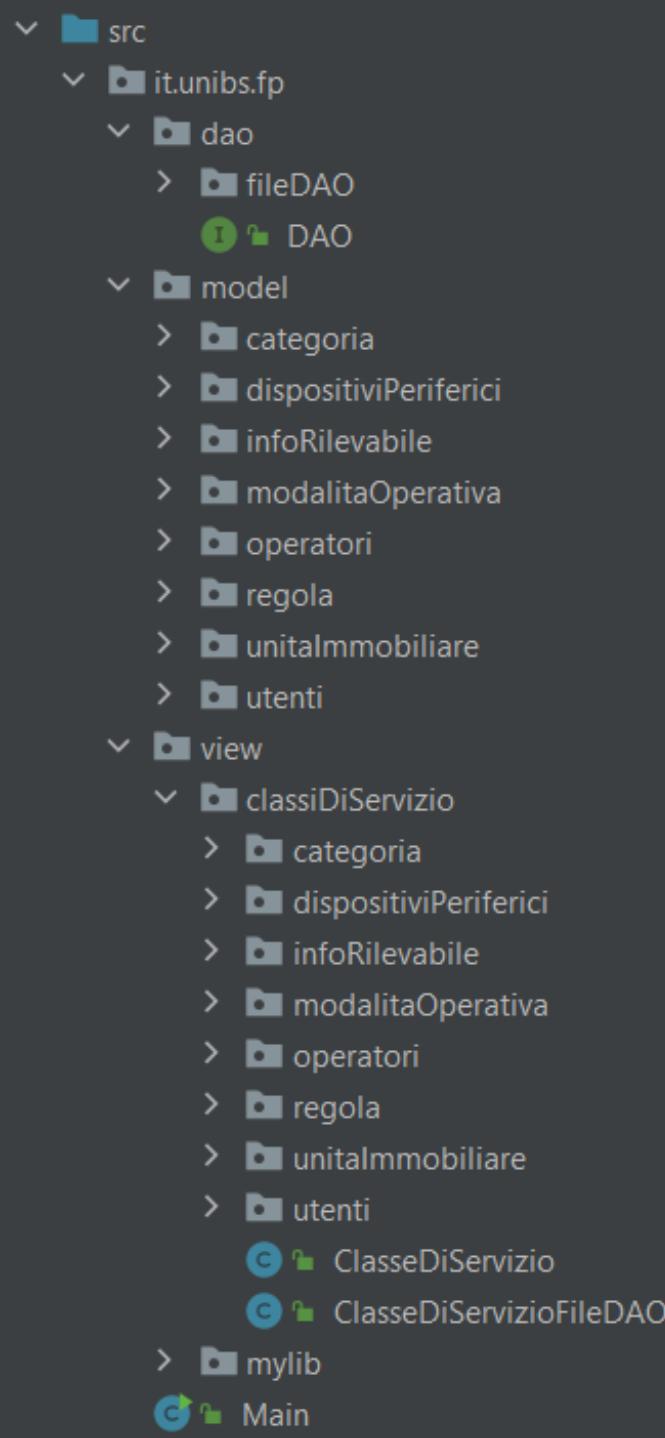
/**
 * Descrizione Artefatti
 *
 * @param unitaImmobiliare per visualizzare i suoi artefatti
 */
public static void descrizioneArtefatti(UnitaImmobiliare unitaImmobiliare) {
    ClasseDiServizioUnitaImmobiliare.descrizioneArtefatti(unitaImmobiliare);
}

/**
 * inserisciESalvaNuovaCategoriaDiSensori
 *
 * @param listaCategorie nel quale salvare
 */
public static void inserisciESalvaNuovaCategoriaDiSensori(ListaCategorie listaCategorie) {
    CategoriaSensori categoriaSensori = ClasseDiServizioListaCategorie.creaCategoriaSensori();
    listaCategorie.inserisciESalvaCategoriaSensori(categoriaSensori);
}

```

Principio Model-View

Esempio ClasseDiServizioManutentore



Architettura software

Anche la suddivisione in package è stata modificata e ottimizzata per rendere il sistema più leggibile e intuitivo. Riprogettazione da 1 package a diversi packages, per esempio:

- DAO
- Model
- View

Che sono divisi in ulteriori package per avere un'organizzazione ottimale.

GRASP

Ideati da Craig Larman, questi pattern supportano il design by responsibility: forniscono un nome e descrivono principi per assegnare responsabilità.

- General
- Responsibility
- Assignment
- Software
- Patterns

Pattern GRASP

- Low coupling
- High cohesion
- Information Expert
- Creator Controller
- Polymorphism
- Pure fabrication
- Indirect
- Protected variation (Legge di Demetra)

Information Expert

Pattern che risolve il problema di assegnare la responsabilità all'oggetto corretto, utilizzando come principio di scelta l'assegnamento all'oggetto con le informazioni necessarie.

Applicazione in Domotic-system-management: la classe ListaCategorie per la creazione di una nuova categoria, assegna la responsabilità di controllare l'omonimia del nome (per evitare categorie ripetute) alla classe con le informazioni necessarie ovvero CategorieSensori o CategorieAttuatori.

ListaCategorie

```
public boolean controlloOmonimia(CategoriaSensori catSen) {  
    for (CategoriaSensori s : categorieSensori)  
        if (catSen.controlloOmonimia(s)) {  
            return true;  
        }  
    return false;  
}
```

CategorieSensori

```
public void inserisciESalvaCategoriaSensori(CategoriaSensori categoriaSensori) {  
    if (!controlloOmonimiaCatSen(categoriaSensori))  
        categorieSensori.inserisciCategoriaSensori(categoriaSensori);  
}
```

Creator

Il pattern **Creator** risolve il problema di chi debba essere responsabile per la creazione di una nuova istanza di una classe.

Un sistema che impiega il pattern Creator, può ottenere anche basso accoppiamento, migliore intelligibilità, encapsulamento e inoltre l'oggetto in esame sarà in grado di sostenere il riuso.

Date due classi A e B, B potrebbe essere responsabile per la creazione di A nel caso in cui B contenga (o nel complesso aggreghi, registri, usi strettamente e contenga) le informazioni iniziali per A.

```
public class UnitaImmobiliare implements Serializable {  
    private final Stanze stanze;  
    private final Artefatti artefatti;  
    private String nome;  
    private Regole regole;  
  
    /**  
     * Costruttore unita' immobiliare  
     *  
     * @param nome dell'unita' immobiliare  
     */  
    public UnitaImmobiliare(String nome) {  
        this.nome = nome;  
        this.stanze = new Stanze();  
        this.artefatti = new Artefatti();  
        this.regole = new Regole();  
    }  
}
```

```
public class Stanze implements Serializable {  
    private List<Stanza> stanze;  
  
    /**  
     * Costruttore Stanze  
     */  
    public Stanze() { stanze = new ArrayList<>(); }  
  
    /**  
     * Getter  
     *  
     * @return stanze  
     */  
    public List<Stanza> getStanze() { return stanze; }  
  
    public void setStanze(List<Stanza> stanze) { this.stanze = stanze; }  
}
```

```
public class Stanza implements Serializable {  
    private String nome;  
    private Artefatti artefatti;  
    private Sensori sensori;  
    private Attuatori attuatori;  
  
    /**  
     * Costruttore della stanza.  
     *  
     * @param nome della stanza.  
     */  
    public Stanza(String nome) {  
        sensori = new Sensori();  
        attuatori = new Attuatori();  
        artefatti = new Artefatti();  
        this.nome = nome;  
    }  
}
```

Creator

SOLID

- Single responsibility: ogni classe dovrebbe avere una sola responsabilità
- Open-closed: una classe dovrebbe essere aperta all'estensione, ma chiusa alle modifiche
- Liskov substitution: ogni superclasse dovrebbe essere rimpiazzabile da una sua sottoclasse, senza modificare il significato del programma
- Interface segregation: meglio tante interfacce, una per ciascun cliente, che un'unica interfaccia buona per tutti
- Dependency inversion: meglio dipendere dalle astrazioni che dalle implementazioni

Interface segregation principle

Questo principio afferma che una classe client non dovrebbe dipendere da metodi che non usa, e che pertanto è preferibile che le interfacce siano molte, specifiche e piccole (composte da pochi metodi) piuttosto che poche, generali e grandi.

Questo consente a ciascun client di dipendere da un insieme minimo di metodi, ovvero quelli appartenenti alle interfacce che effettivamente usa.

Di conseguenza, un oggetto dovrebbe tipicamente implementare numerose interfacce, una per ciascun ruolo che l'oggetto stesso gioca nei diversi contesti o diverse interazioni con altri oggetti.

Interface segregation principle

Anziché utilizzare ArrayList o LinkedList abbiamo optato nell'utilizzo List così da minimizzare la dipendenza da metodi.

```
public class CategorieSensori implements Serializable {  
    private List<CategoriaSensori> categorieSensori;  
  
    public CategorieSensori() { categorieSensori = new ArrayList<>(); }  
  
    public List<CategoriaSensori> getCategorieSensori() { return categorieSensori; }  
  
    public void setCategorieSensori(List<CategoriaSensori> categorieSensori) {  
        this.categorieSensori = categorieSensori;  
    }  
  
    public CategoriaSensori getCategoriaSensori(int i) { return categorieSensori.get(i); }  
}
```

Open-Closed Principle

Pattern utile per estendere il comportamento di un modulo, affinché quest'ultimo possa comportarsi in nuovi modi a fronte di nuovi requisiti.

Applicazione in Domotic-system-management: variante del pattern chain of responsibility in modo da rispettare la chiusura alla modifica e l'aperura all'estensione.

```
public class CompositeRendererInfoRilevabile implements RendererInfoRilevabile {  
  
    List<SelectableRendererInfoRilevabile> renderers = new ArrayList<>() {  
        {  
            add(new InfoNumericaRendererInfoRilevabile());  
            add(new InfoNonNumericaRendererInfoRilevabile());  
        }  
    };  
  
    private RendererInfoRilevabile getRendererFor(InfoRilevabile info) {  
        for(SelectableRendererInfoRilevabile renderer : this.renderers) {  
            if(renderer.canHandle(info))  
                return renderer;  
        }  
        return null;  
    }  
  
    @Override  
    public String renderDescriviValoreRilevato(InfoRilevabile info) {  
        return getRendererFor(info).renderDescriviValoreRilevato(info);  
    }  
  
    @Override  
    public String renderDescriviSoloValoreRilevato(InfoRilevabile info) {  
        return getRendererFor(info).renderDescriviSoloValoreRilevato(info);  
    }  
  
    @Override  
    public String renderToString(InfoRilevabile info) { return getRendererFor(info).renderToString(info); }  
}
```

Open-Closed Principle

Abbiamo trasformato InfoRilevabile da classe astratta a interfaccia inoltre abbiamo scelto di inserire la classe SelectableRendererInfoRilevabile e CompositeRendererInfoRilevabile in modo da rendere il tutto chiuso alla modifica e aperto all'estensione.

```
public class CompositeRendererInfoRilevabile implements RendererInfoRilevabile {  
  
    List<SelectableRendererInfoRilevabile> renderers = new ArrayList<>(){  
        {  
            add(new InfoNumericaRendererInfoRilevabile());  
            add(new InfoNonNumericaRendererInfoRilevabile());  
        }  
    };  
  
    private RendererInfoRilevabile getRendererFor(InfoRilevabile info) {  
        for(SelectableRendererInfoRilevabile renderer : this.renderers) {  
            if(renderer.canHandle(info))  
                return renderer;  
        }  
        return null;  
    }  
  
    @Override  
    public String renderDescriviValoreRilevato(InfoRilevabile info) {  
        return getRendererFor(info).renderDescriviValoreRilevato(info);  
    }  
  
    @Override  
    public String renderDescriviSoloValoreRilevato(InfoRilevabile info) {  
        return getRendererFor(info).renderDescriviSoloValoreRilevato(info);  
    }  
  
    @Override  
    public String renderToString(InfoRilevabile info) { return getRendererFor(info).renderToString(info); }  
}
```

GoF (Gang of four)

23 pattern presentati nel libro «Design Pattern» della Gang of Four.

Questi principi sono organizzati e classificati in base a diversi criteri:

In base allo scopo:

- Creazionale: riguarda la creazione di oggetti
- Strutturale: riguarda la composizione di classi/oggetti
- Comportamentale: riguarda l'interazione tra classi/oggetti e la distribuzione di responsabilità

In base al raggio d'azione:

- Class pattern: riguarda le relazioni tra classi e sotto-classi (statiche)
- Object pattern: riguarda le relazioni tra oggetti (dinamiche)

Design Pattern

I Design pattern hanno molte funzioni, ad esempio aiutano a:

- Identificare astrazioni non ovvie e oggetti che possono rappresentarle
- Definire le interfacce e definiscono relazioni tra esse
- Rendere il sistema più adatto a sopportare un particolare tipo di cambiamento.

GoF (Gang of four)

		PURPOSE		
SCOPE	CLASS	CREATIONAL	STRUCTURAL	BEHAVIORAL
		Factory method	Adapter	Interpreter Template method
OBJECT	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor	Visitor Strategy State Observer

DAO

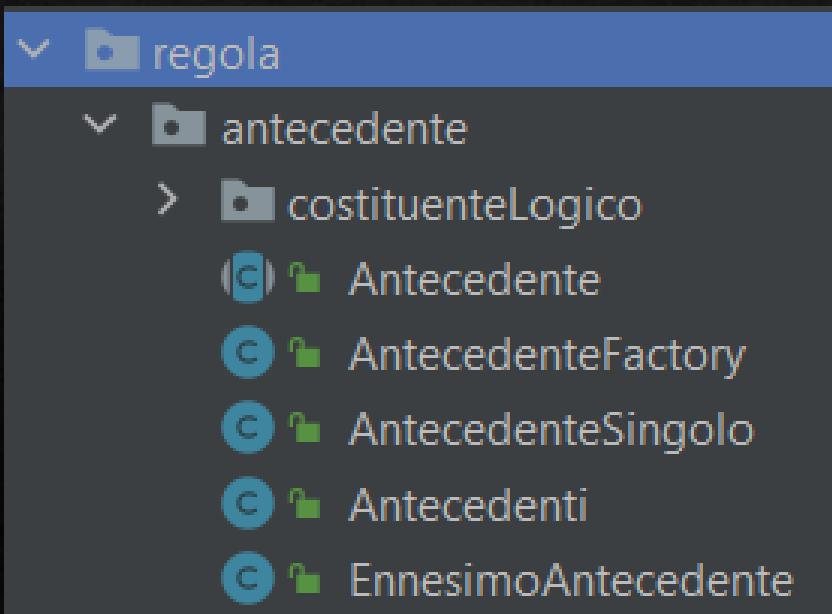
Il DAO è un pattern utilizzato per dividere la responsabilità della gestione dei dati persistenti utilizzando una classe specifica per le varie operazioni in modo da astrarre.

Applicazione in Domotic-system-management: si utilizza una classe esterna per gestire le letture e i salvataggi su disco, in modo da astrarre la procedura, (I salvataggi vengono effettuati su diversi contenitori in base al tipo di oggetto) ed è possibile avere anche diversi tipi di salvataggio.

Simple factory

Pattern utile per assegnare la responsabilità ad una classe per la creazione di oggetti adattatori, attraverso la creazione di un oggetto Pure Fabrication che gestisce la creazione.

Applicazione in Domotic-system-management: Utilizzo della classe AntecedenteFactory per la creazione degli oggetti



Simple Factory

Classe utilizzata per la creazione degli oggetti

```
public class AntecedenteFactory implements Serializable {  
    /**  
     * Crea antecedente singolo  
     *  
     * @param costituenteLogico per la creazione dell'antecedente  
     * @return antecedente  
     */  
    public static Antecedente creaAntecedente(CostituenteLogico costituenteLogico) {  
        return new AntecedenteSingolo(costituenteLogico);  
    }  
  
    /**  
     * Crea ennesimo antecedente  
     *  
     * @param opBooleano per il confronto  
     * @param costituenteLogico per la creazione dell'antecedente  
     * @return antecedente  
     */  
    public static Antecedente creaAntecedente(OperatoriBooleani opBooleano, CostituenteLogico costituenteLogico) {  
        return new EnnesimoAntecedente(opBooleano, costituenteLogico);  
    }  
}
```

Testing

Procedura di controllo e collaudo del progetto prima della consegna al cliente.

Applicazione in Domotic-System-Management:

- Test dello Sviluppo (verifica)
- Test delle unità: vengono testate le singole unità di programma o classi di oggetti per verificare le funzionalità di oggetti e metodi



Test della classe CategoriaSensori

Testing: classe «Invito»

```
class CategoriaSensoriTest {  
    private static final String NOME_CATEGORIA_SENSORE="Termometro";  
    private static final String TESTO_LIBERO="Termometro che misura una variabile fisica";  
    private static final ArrayList<InfoRilevabile> infoRilevabili = creaInfoRilevabili();  
    private static final String NOME_INFORMAZIONE_RILEVABILE="Temperatura";  
    private static final double MIN=0.0;  
    private static final double MAX=40.0;  
    private static final InfoRilevabile infoRilevabile = new InfoRilevabileNumerica(NOME_INFORMAZIONE_RILEVABILE,MIN,MAX);  
    private CategoriaSensori categoriaNuova;  
  
    private static ArrayList<InfoRilevabile> creaInfoRilevabili(){  
        ArrayList<InfoRilevabile> infoRilevabilit = new ArrayList<>();  
        infoRilevabilit.add(infoRilevabile);  
        return infoRilevabilit;  
    }  
  
    @Test  
    public void inizializzazioneTest(){  
        categoriaNuova = new CategoriaSensori(NOME_CATEGORIA_SENSORE,TESTO_LIBERO,infoRilevabili);  
        assertEquals(NOME_CATEGORIA_SENSORE,categoriaNuova.getNome());  
        assertEquals(TESTO_LIBERO,categoriaNuova.getTestoLibero());  
        assert categoriaNuova.getInformazioniRilevabili().size()==1;  
    }  
  
    @Test  
    void getTestoLibero() {  
        categoriaNuova = new CategoriaSensori(NOME_CATEGORIA_SENSORE,TESTO_LIBERO,infoRilevabili);  
        assertEquals(TESTO_LIBERO,categoriaNuova.getTestoLibero());  
    }  
  
    @Test  
    void getInformazioniRilevabili() {  
        categoriaNuova = new CategoriaSensori(NOME_CATEGORIA_SENSORE,TESTO_LIBERO,infoRilevabili);  
        assertEquals(infoRilevabili,categoriaNuova.getInformazioniRilevabili());  
    }  
  
    @Test  
    void getName() {  
        categoriaNuova = new CategoriaSensori(NOME_CATEGORIA_SENSORE,TESTO_LIBERO,infoRilevabili);  
        assertEquals(NOME_CATEGORIA_SENSORE,categoriaNuova.getName());  
    }  
}
```

Refactoring

«Un cambiamento fatto alla struttura interna del software per renderlo più facile da capire e più semplice da modificare senza cambiarne il comportamento»

Esempio in Domotic-system-management:

Pattern utilizzato: Extract Method

Prima

```
/*
 * Associa un sensore ad uno o più artefatti
 *
 * @param contenitore per ottenere gli oggetti necessari
 */
static void associaSensoreAdArtefatti(Contenitore contenitore) {
    ListaCategorie listaCategorie = contenitore.getListeCategorie();
    Manutentore manutentore = contenitore.getManutentore();
    UnitaImmobiliare unitaImmobiliare = ClasseDiServizioInserimenti.scegliUnitaImmobiliare(manutentore);
    if (!unitaImmobiliare.getArtefatti().isEmpty() & !listaCategorie.getCategorieSensori().isEmpty()) {
        do {
            Sensore nuovoSensore = ClasseDiServizioInserimenti.creaSensore(listaCategorie);
            ArrayList<Artefatto> artefatti = ClasseDiServizioInserimenti.scegliArtefatti(unitaImmobiliare);
            for (Artefatto a : artefatti)
                for (Sensore s : a.getSensori())
                    if (nuovoSensore.getCategoriaSensori() == s.getCategoriaSensori())
                        System.out.println("Esiste già un sensore con la stessa categoria in uno degli artefatti");

            manutentore.associaSensoreAdArtefatti(nuovoSensore, artefatti, unitaImmobiliare);

            contenitore.setListeCategorie(listaCategorie);
            contenitore.setManutentore(manutentore);
            ServizioFile.salvaSingoloOggetto(new File( pathname: "contenitore.dat"), contenitore);
        } while (InputDati.yesOrNo( messaggio: "Vuoi associare un'altro sensore ad artefatti?"));
    } else {
        if (listaCategorie.getCategorieSensori().isEmpty())
            System.out.println("Bisogna prima creare una categoria sensori");
        if (unitaImmobiliare.getArtefatti().isEmpty())
            System.out.println("Bisogna prima avere un artefatto nell'unità immobiliare");
    }
}
```

Dopo

```
/**  
 * Associa Sensore ad artefatti  
 *  
 * @param unitaImmobiliare nel quale scegliere gli artefatti  
 * @param listaCategorie per la creazione di sensori  
 */  
public static void associaSensoreAdArtefatti(UnitaImmobiliare unitaImmobiliare, ListaCategorie listaCategorie) {  
    if (!unitaImmobiliare.artefattiInUnitaImmobiliareIsEmpty() && !listaCategorie.categorieSensoriIsEmpty())  
        do  
            associaSensoreArtefatto(unitaImmobiliare, listaCategorie);  
        while (InputDati.yesOrNo( messaggio: "Vuoi associare un'altro sensore ad artefatti?"));  
  
    else {  
        gestisciCategoriaSensoriMancante(listaCategorie);  
        gestisciArtefattiMancanti(unitaImmobiliare);  
    }  
}
```