

Progetto

Ingegneria del Software

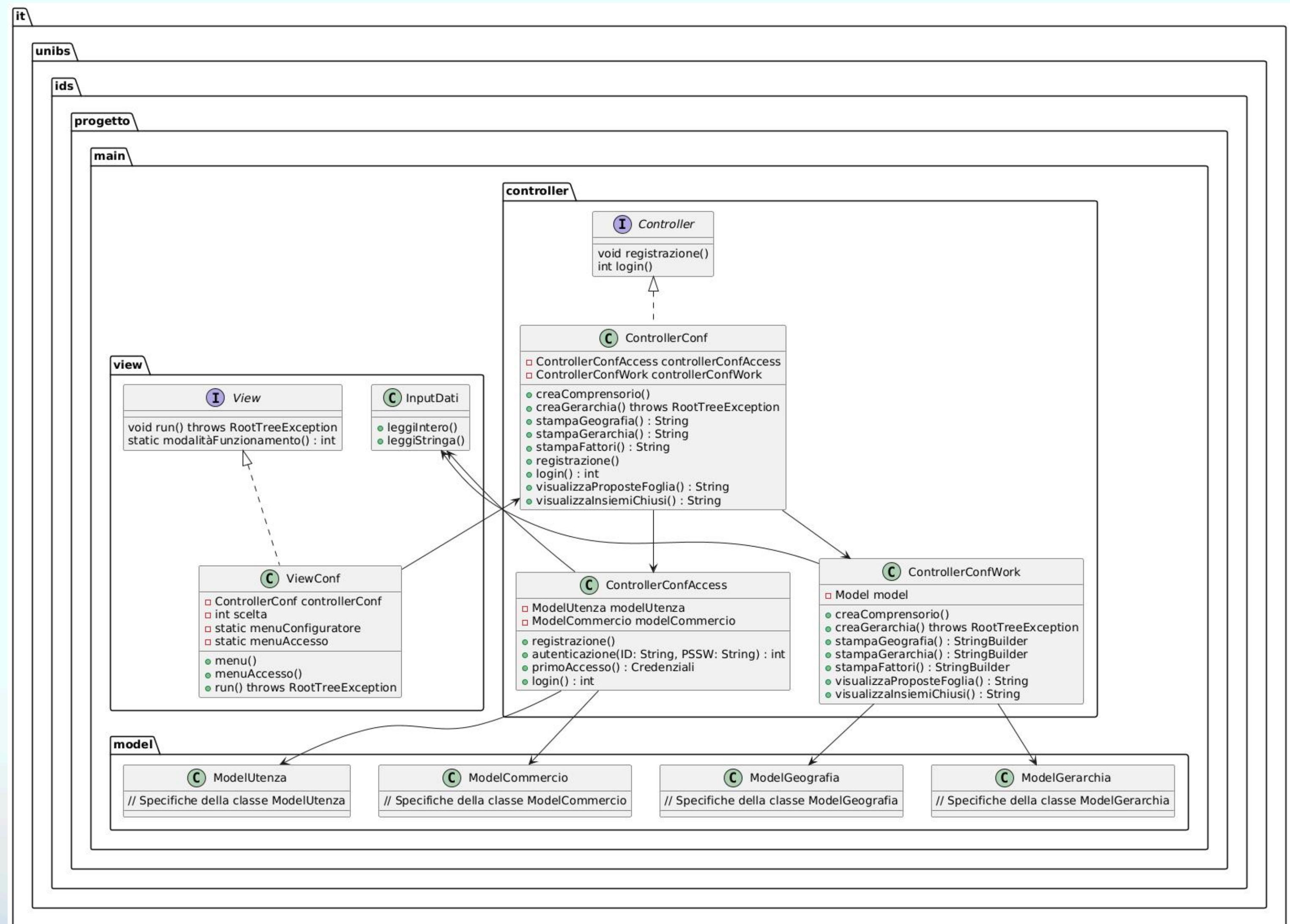
Indice

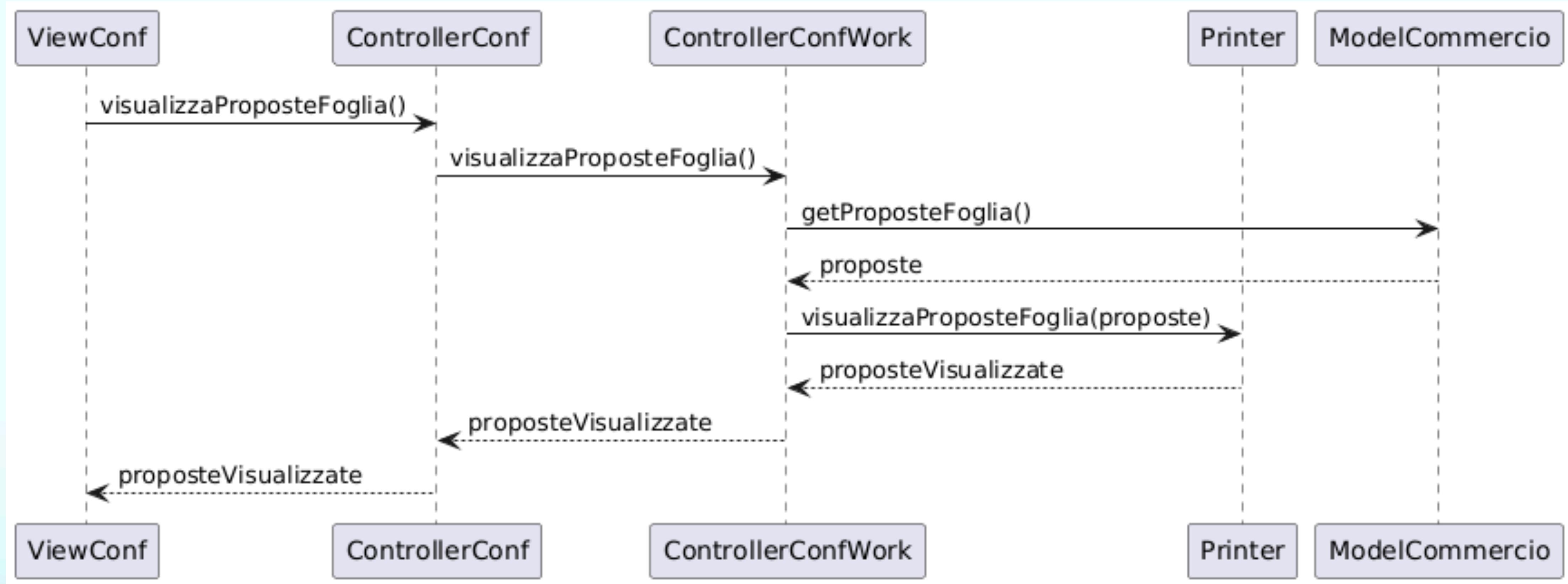
1. Modello MVC 3–5
2. Pure Fabrication (GRASP) 7–8
3. Controller (GRASP) 9–11
4. Composite (GoF) 13–15
5. Singleton(GoF) 16–17
6. Single Responsibility (SOLID) 19–20
7. Open Closed(SOLID) 21–22
8. Testing 23–29
9. Refactoring 30–36

MVC

Model—View—Controller

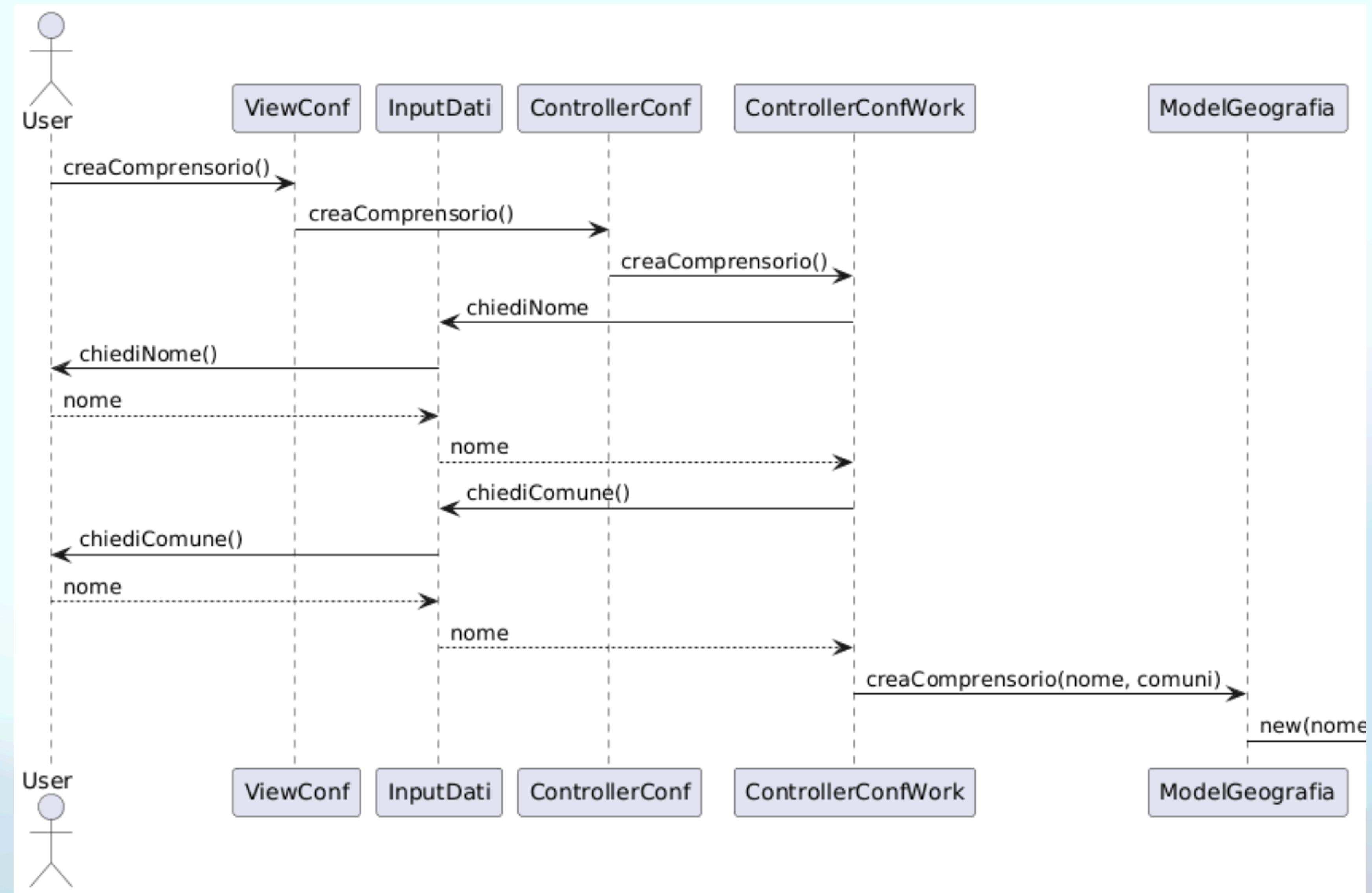
- Il *Model* rappresenta i dati e la logica di business, la *View* gestisce l’interfaccia utente e la presentazione dei dati, e il *Controller* funge da intermediario.
- Il diagramma dei package mostrato riguarda le funzionalità del solo Configuratore.





- Si noti che le dipendenze vanno dalla strato di presentazione allo strato di business.
- Le Stringhe sono formattate nella classe *Printer*.

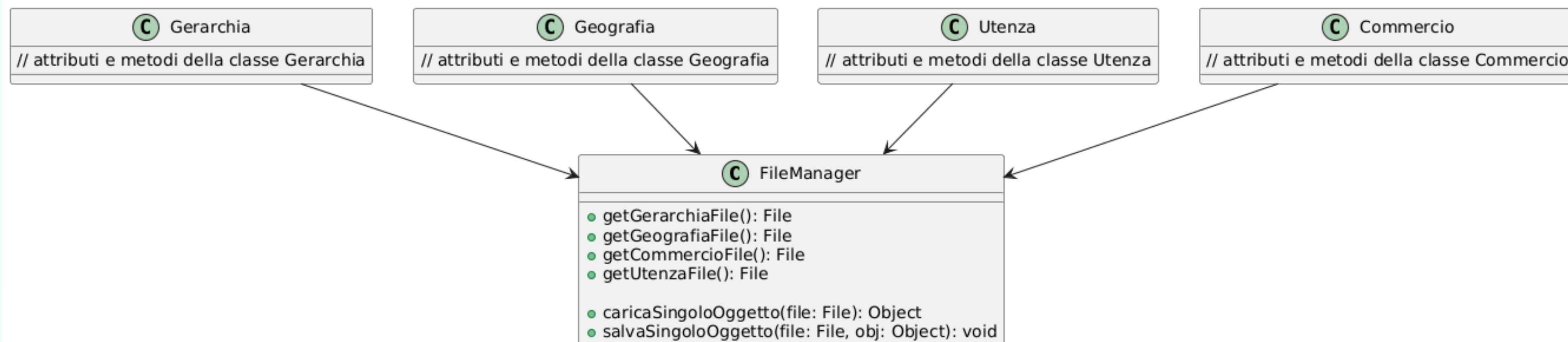
- Si noti che le dipendenze vanno dalla strato di presentazione allo strato di business.
 - La richiesta degli input è delegata alla classe *InputDati* (appartenente alla View).



Pattern GRASP

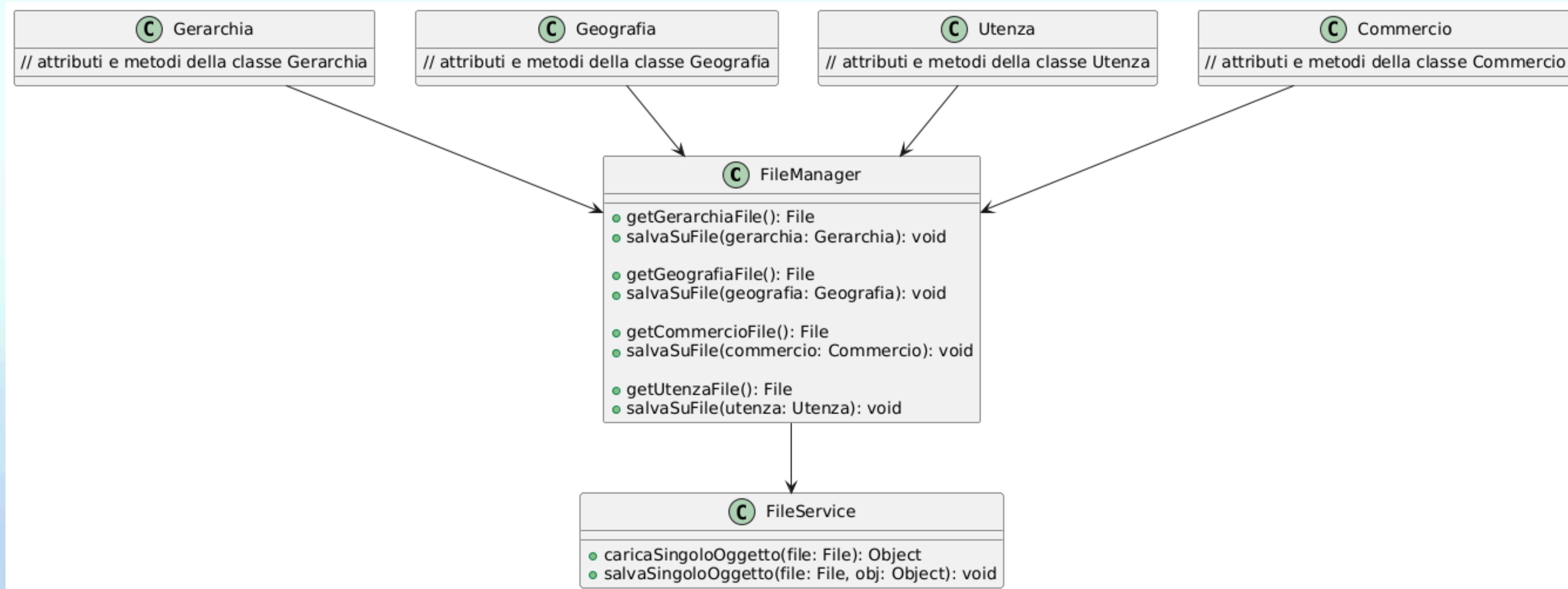
- GRASP (General Responsibility Assignment Software Patterns) è una serie di principi utilizzati per assegnare responsabilità agli oggetti in un sistema orientato agli oggetti.
- L'obiettivo di GRASP è creare un *design robusto e facilmente manutenibile*.

Pure Fabrication



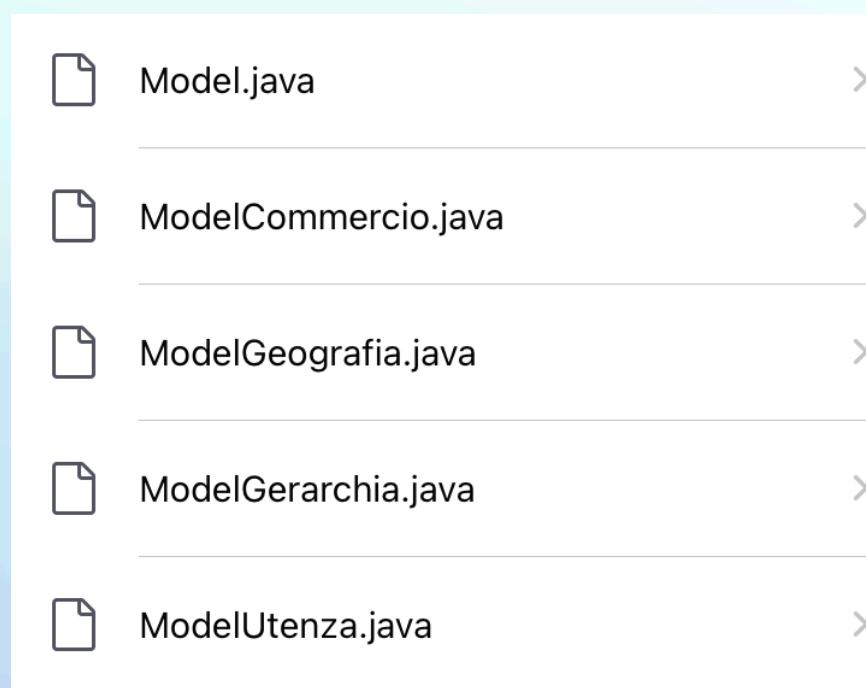
- La responsabilità di salvare/caricare un oggetto (*Gerarchia*, *Geografia*, *Utenza*, *Commercio*) dovrebbe essere delegata a una classe di servizio dedicata, come *FileManager*.
- Questa scelta segue i principi del pattern *Pure Fabrication*, evitando bassa coesione (ex. *Gerarchia* avrebbe a che fare con la persistenza dei dati) e alto accoppiamento (ex. codice duplicato).

- È stato aggiunto un ulteriore *livello di indirezione*, favorendo il riuso di una classe *FileService*, per la gestione dei dati persistenti. *FileManager* demanderà a *FileService* certe responsabilità.
- Le operazioni di salvataggio e caricamento possono essere riutilizzate facilmente per diverse classi, promuovendo la modularità e la manutenibilità del sistema.

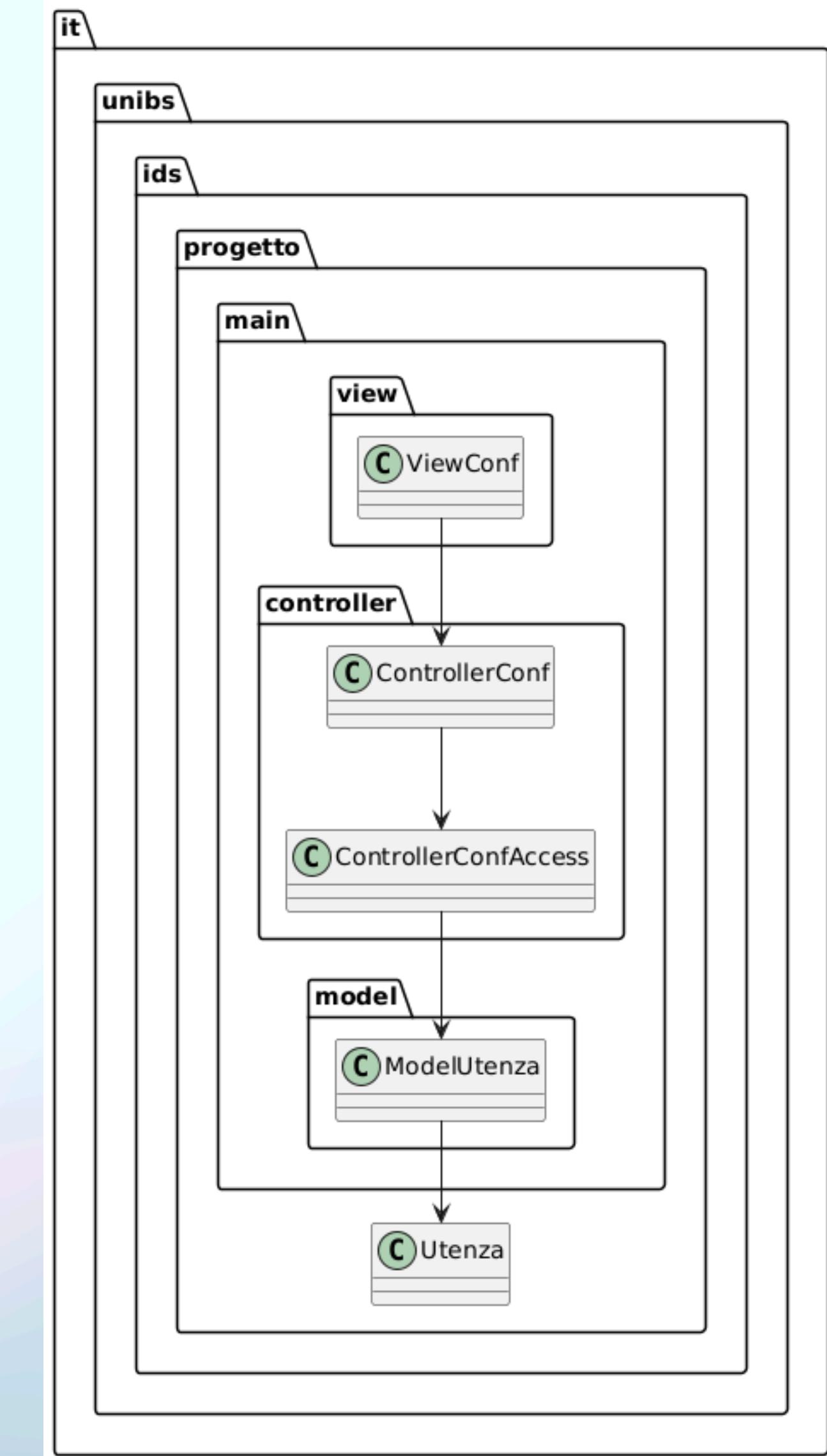


Controller

- Qual è il primo oggetto oltre lo strato UI che riceve e coordina un'operazione di sistema? Il Model, che è il controller GRASP.
- Nell'implementazione iniziale era stato realizzato un Facade Controller, rappresentante l'intero sistema (la classe *Model*).



- Con l'aggiunta di funzionalità, il *Model* è diventato troppo carico di responsabilità, preferendo così una suddivisione in base all'ambito di lavoro (Handler Controller): *ModelCommercio*, *ModelGerarchia*, *ModelGeografia* e *ModelUtenza*.



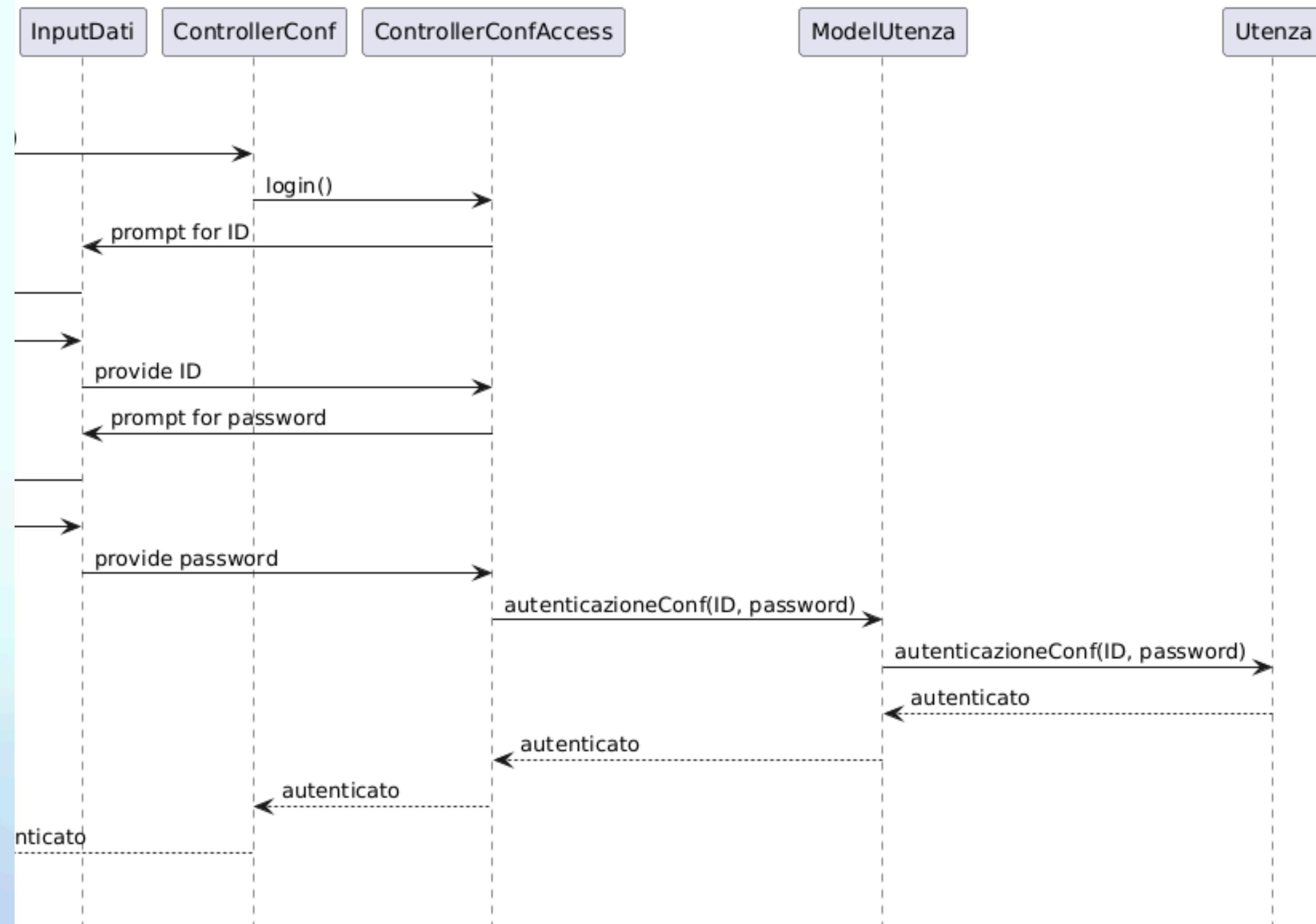
- La classe *Model* rimane utilizzata come da pattern di refactoring [“Introduce Parameter Object”](#).

```
* Costruttore di default che inizializza tutti i modelli
public Model() {
    super();
    this.modelUtenza = new ModelUtenza();
    this.modelGerarchia = new ModelGerarchia();
    this.modelGeografia = new ModelGeografia();
    this.modelCommercio = new ModelCommercio();
}
```

```
Model model = new Model();
Controller controller;
View view;

int mode = View.modalitàFunzionamento();
if (mode == 1) {
    controller = new ControllerConf(model);
    view = new ViewConf(controller);
    view.run();
}
```

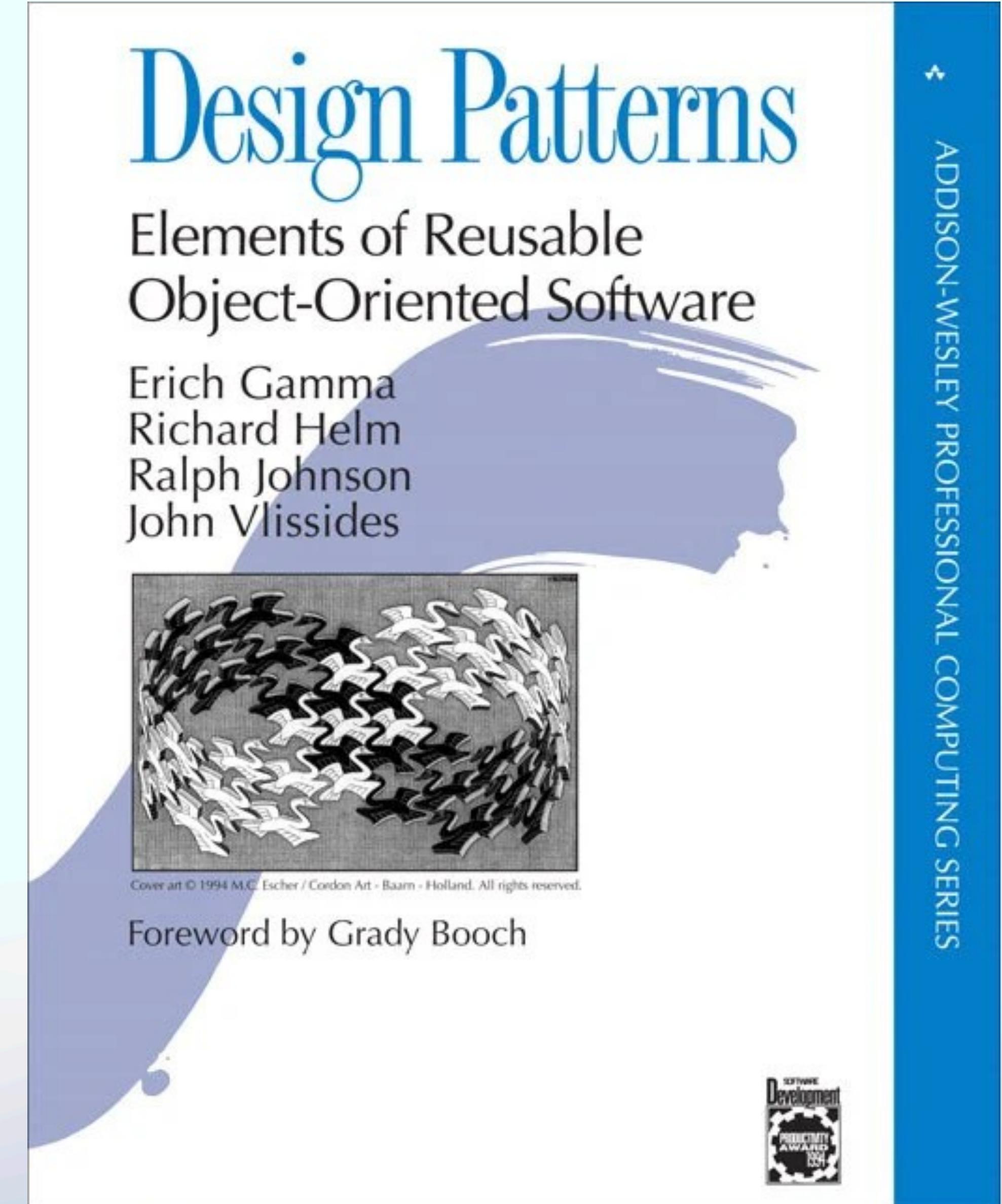
- I diversi modelli (*ModelUtenza*, *ModelGerarchia*, *ModelGeografia*, *ModelCommercio*) vengono raggruppati e creati nella classe *Model*.
- Viene utilizzato all'occorrenza il singolo oggetto *Model*, anziché i 4 Handler.



- Il Controller MVC (ControllerConf) fa parte della interfaccia utente e gestisce l'interazione con l'utente.
- Il Controller GRASP (ModelUtenza) fa parte dello strato di dominio e gestisce le operazioni di sistema (in questo esempio con Utenza).

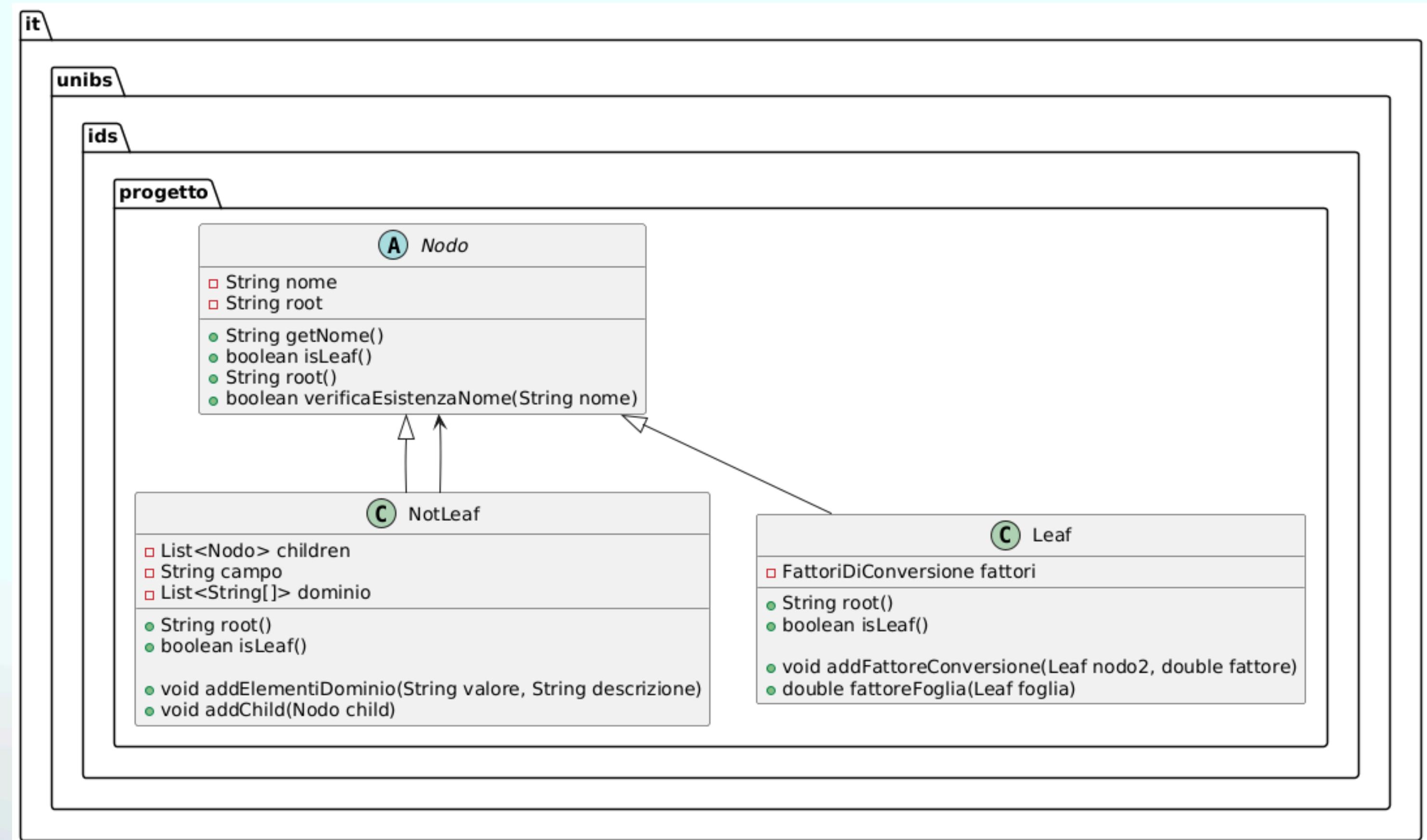
Pattern GoF

- I Pattern GoF (Gang of Four) sono 23 design pattern descritti nel libro “Design Patterns: Elements of Reusable Object-Oriented Software”.
- Questi pattern sono suddivisi in tre categorie principali: Creazionali (come il Singleton), Strutturali (come il Adapter), e Comportamentali (come il Observer).
- Forniscono soluzioni standard per problemi ricorrenti nel design del software.



Composite

- La classe astratta *Nodo* funge da componente comune (Component) che definisce l'interfaccia per tutti gli oggetti, sia *Leaf* che *NotLeaf*.
- Questo permette di trattare le leaf e le notLeaf in modo uniforme.



- Il pattern Composite applicato è di tipo type Safety.
- La sicurezza di tipo viene garantita tramite l'uso di metodi astratti e override (`isLeaf()` e costruttore).
- I costruttori di `Leaf` e `NotLeaf` sono specifici per ciascun tipo, rendendo chiaro come devono essere inizializzati.
- La classe `NotLeaf` implementa le operazioni `addChild()`, `GetChildren()`.

```
* Verifica se il nodo è una foglia.
public abstract boolean isLeaf();
```

```
public Leaf(String nome, String root) {
    this.nome = nome;
    this.root = root;
    this.fattori = new FattoriDiConversione();
    this.fattori.addFattoreConversione(this, 1.0);
}
```

```
public NotLeaf(String nome, String root, String campo) {
    this.root = root;
    this.campo = campo;
    this.nome = nome;
    this.children = new ArrayList<>();
    this.dominio = new ArrayList<>();
}
```

```
public void addChild(Nodo child) {
    this.children.add(child);
}
```

```
public List<Nodo> getChildren() {
    return children;
}
```

Uniformity?



- Nel caso si utilizzino degli alberi con molti livelli di profondità (come può accadere in futuro in un progetto di questo tipo), è preferibile utilizzare il pattern Composite nell'approccio Uniformity.
- Utilizzare il pattern Composite nella forma Uniformity introduce la necessità di gestire operazioni non applicabili a certi tipi di componenti (come `addChild()` utilizzabili su una *Leaf* dovrebbero essere gestite sollevando eccezioni).

Singleton

- E' necessaria una sola istanza della classe *DefaultInitializer*, ovvero la classe adibita all'inizializzazione del sistema.
- Per rispettare il patter Singleton si definisce un metodo statico (di classe) *getDefaultValueInitializer()* che restituisce l'oggetto singleton *defaultValueInitializer*.
- Il costruttore della classe è private.

```
private DefaultInitializer() {  
    this.geografia = defaultWorld();  
    this.utenza = defaultAccess();  
    this.gerarchia = defaultTree();  
    this.commercio = defaultCommercio();  
}  
  
private static DefaultInitializer defaultInitializer;  
//singleton  
public static DefaultInitializer getDefaultInitializer() {  
    if (defaultInitializer == null)  
        defaultInitializer= new DefaultInitializer();  
    return defaultInitializer;  
}
```

- Nonostante sia stato utilizzato, un oggetto Singleton comporta diversi possibili svantaggi:
 1. Difficoltà di testing.
 2. Difficoltà di estensione futura e flessibilità della classe nel tempo.
 3. Violazione del principio di singola responsabilità. Un Singleton può accumulare molteplici responsabilità nel corso del tempo, rendendo il codice più complesso da gestire e da evolvere.

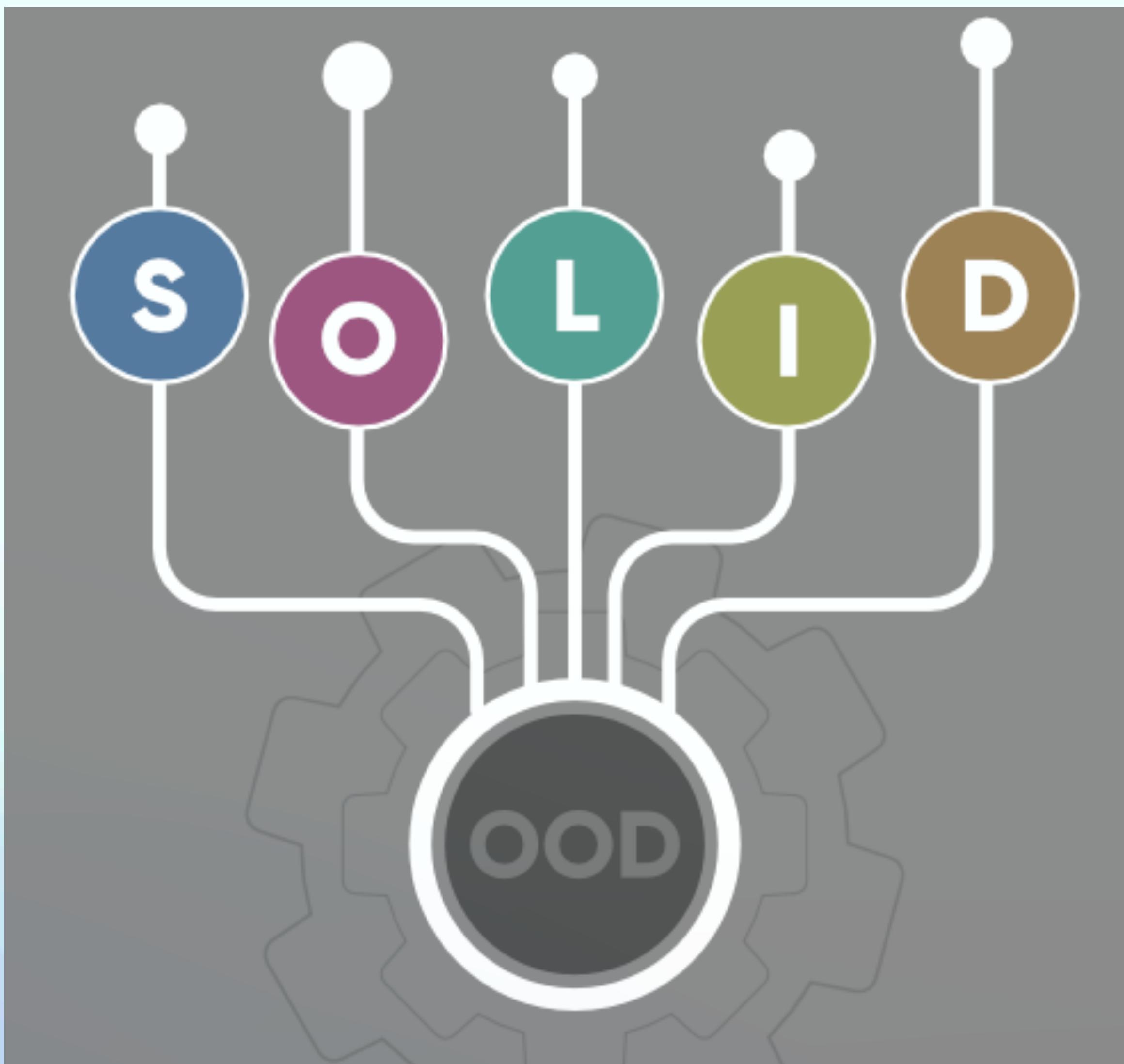
```
if (defaultInitializer == null)
    defaultInitializer= new DefaultInitializer();

return defaultInitializer;
if (defaultInitializer == null) {
    synchronized (DefaultInitializer.class) {
        if (defaultInitializer == null) {
            defaultInitializer = new DefaultInitializer();
        }
    }
}
return defaultInitializer;
```

4. Difficoltà di utilizzo in applicazioni multi-thread. È necessaria una modifica del metodo getDefaultInitializer, per evitare che più thread creino contemporaneamente un nuovo oggetto.

Principi SOLID

- SOLID è un acronimo che rappresenta cinque principi di design orientato agli oggetti:
 - A. Single Responsibility Principle (SRP);
 - B. Open/Closed Principle (OCP);
 - C. Liskov Substitution Principle (LSP);
 - D. Interface Segregation Principle (ISP);
 - E. Dependency Inversion Principle (DIP).



Single Responsibility

- L'oggetto Fruitore è caratterizzato da un comprensorio, delle credenziali e un indirizzo mail (oltre che delle proposte).
- La responsabilità di gestire le email (come String) è della classe *Fruitore*

```
+public class Fruitore extends Utente implements Serializable {  
+  
+  
+    private static final long serialVersionUID = 1L;  
+    public static final char TIPOUTENTE = 'f';  
+  
+    private String indirizzo;  
+    private Comprensorio comprensorioAppartenenza;  
+    private ArrayList<PropostaDiScambio> proposte;  
+  
+  
+    /**  
+     * Costruttore della classe Fruitore.  
+     * Crea un nuovo fruitore con le credenziali specificate.  
+     *  
+     * @param credenziali Le credenziali associate al fruitore  
+     */  
+    public Fruitore(Comprensorio comprensorioAppartenenza, Credenziali credenziali, String indirizzo) {  
+        super(TIPOUTENTE, credenziali);  
+        this.indirizzo = indirizzo;  
+        this.comprensorioAppartenenza = comprensorioAppartenenza;  
+        setIsDefinitivo(true);  
+    }
```

- Dividendo le responsabilità in due classi separate, si garantisce che ciascuna classe abbia un solo motivo per cambiare.

```

8+ * La classe MailAddress rappresenta un oggetto per gestire e validare un indirizzo email.□
10 public class MailAddress implements Serializable {
11
12     private static final long serialVersionUID = 1L;
13
14     // Pattern regex per validare un indirizzo email
15+    private static final String EMAIL_PATTERN = □
16
17     private static final Pattern pattern = Pattern.compile(EMAIL_PATTERN);
18
19     private String email; // Stringa che rappresenta l'indirizzo email
20
21     * Costruttore della classe MailAddress.□
22+    public MailAddress(String email) {□
23
24         * Metodo getter per ottenere l'indirizzo email.□
25+        public String getEmail() {□
26
27             * Metodo setter per impostare l'indirizzo email.□
28+        public void setEmail(String email) {□
29
30             * Metodo statico per verificare se una stringa rappresenta un indirizzo email valido.□
31+        public static boolean isValidEmail(String email) {□
32
33         }
34
35     }
36
37     * Metodo statico per verificare se una stringa rappresenta un indirizzo email valido.□
38+    public static boolean isValidEmail(String email) {□
39
40         }
41
42     }
43
44     * Metodo statico per verificare se una stringa rappresenta un indirizzo email valido.□
45+    public static boolean isValidEmail(String email) {□
46
47         }
48
49     }
50
51     * Metodo statico per verificare se una stringa rappresenta un indirizzo email valido.□
52+    public static boolean isValidEmail(String email) {□
53
54         }
55
56     }
57
58 }
```

- Ad esempio, se le regole di validazione dell'email cambiano, si dovrà modificare solo la classe *MailAddress* senza toccare la classe *Fruitore*.

Open Closed

- PrestazioneOpera è una classe astratta che definisce proprietà e metodi comuni per tutte le prestazioni.
- Essendo astratta, è progettata per essere estesa da altre classi, permettendo così di aggiungere nuove funzionalità senza modificare il codice della classe base.

```
4
6+ * La classe PrestazioneOpera rappresenta un'astrazione di una prestazi
7+ public abstract class PrestazioneOpera implements Serializable {
8
9
10   // Riferimento a una foglia dell'albero
11   protected Leaf foglia;
12
13   // Durata della prestazione
14   protected int durata;
15
16   * Metodo getter per ottenere la foglia dell'albero associata alla p
17+   public Leaf getFoglia() {
18
19
20+     * Metodo getter per ottenere la durata della prestazione.■
21+     public int getDurata() {
22
23
24+       * Metodo setter per impostare la durata della prestazione.■
25+       public void setDurata(int durata) {
26         this.durata = durata;
27
28
29
30+       * Metodo getter per ottenere il nome della foglia associata alla p
31+       public String getNome() {
32
33
34+     }
35
36
37
38
39
40
41+   }
42
43
44+ }
```

- La classe *Offerta* consente di creare una *PrestazioneOpera* sulla base di una leaf.

- Similmente, la classe *Richiesta* consente di creare una *PrestazioneOpera* sulla base di una leaf e una durata.

- Il comportamento di *PrestazioneOpera* potrebbe essere esteso cosicché possa comportarsi in nuovi modi a fronte di nuovi requisiti (ad esempio l'introduzione di una classe *Prenotazione*).

```

• * La classe Offerta rappresenta una specifica offerta di prestazione relativa a
public class Offerta extends PrestazioneOpera implements Serializable {

    * Costruttore della classe Offerta che inizializza la foglia dell'albero associato
    public Offerta(Leaf foglia) {}

}

```

```

• * La classe Richiesta rappresenta una specifica richiesta di prestazione relativa a
public class Richiesta extends PrestazioneOpera implements Serializable {

    * Costruttore della classe Richiesta che inizializza la foglia dell'albero associato e la durata
    public Richiesta(Leaf foglia, int durata) {}

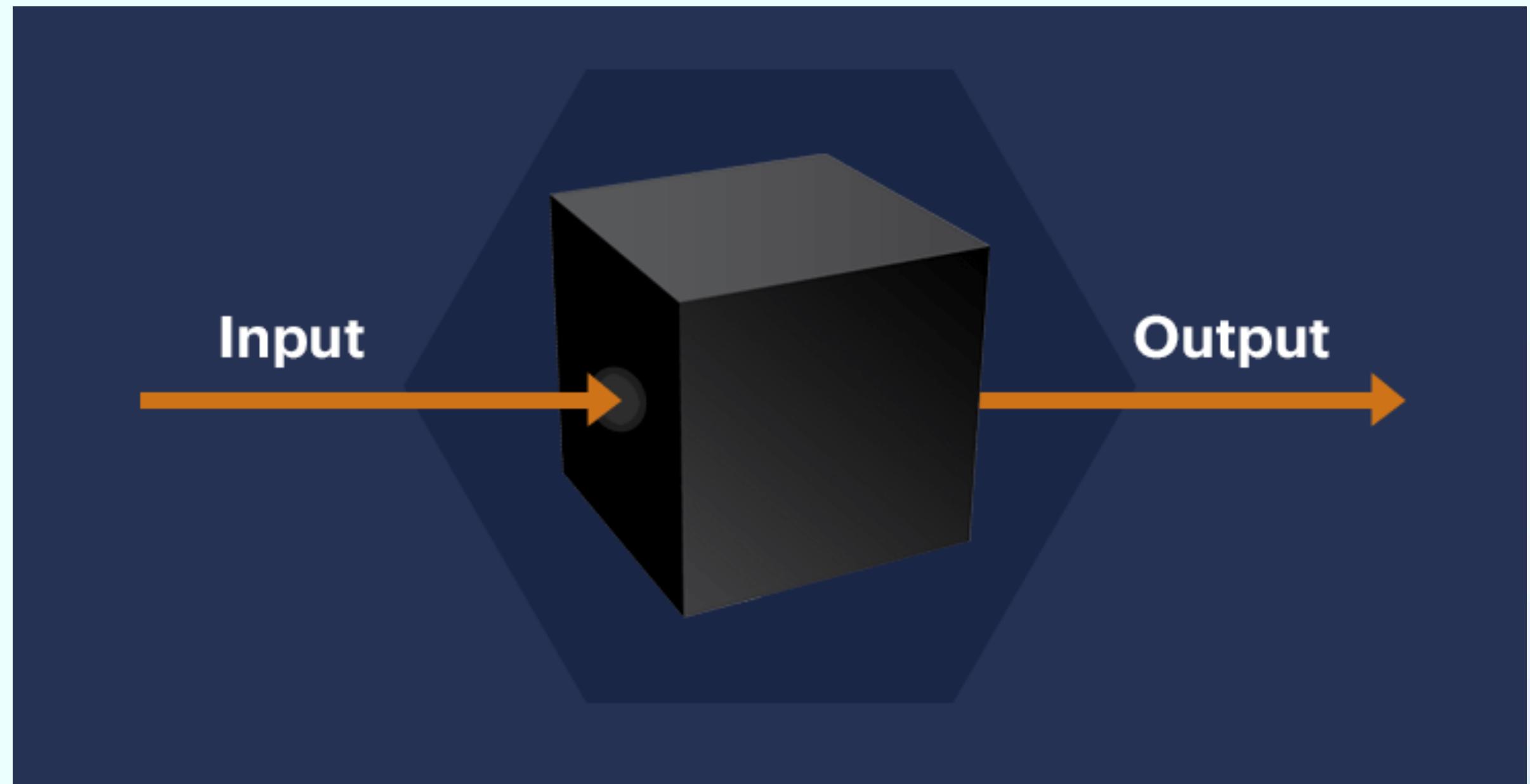
}

```

Testing

Test Black Box

- Il testing Black Box è una tecnica di test del software che esamina la funzionalità di un'applicazione senza conoscere la sua struttura interna o il codice sorgente.
- Il tester interagisce con l'applicazione attraverso le sue interfacce, fornendo input e verificando gli output contro i risultati attesi.
- L'obiettivo è verificare che il software soddisfi i requisiti.



Equivalence Partitioning Test

CommercioRegoleTest

- Le Proposte che soddisfano i criteri di chiusura (dell'algoritmo) formano una classe di equivalenza, mentre quelle che non li soddisfano (e rimangono aperte) ne formano un'altra.

```
@Test
void test_Chiudi_3Proposte() {
    // Test per verificare che sia possibile chiudere 3 proposte
    InsiemeAperto insiemeAperto0 = new InsiemeAperto(fruitore0.getComprensorioAppartenenza());

    PropostaAperta proposta00 = createProposta(
        new Richiesta(gerarchia.getFoglie().get(0), DefaultInitializer.COMMERCIO_VAL_00),
        new Offerta(gerarchia.getFoglie().get(1))
    );
    PropostaAperta proposta01 = createProposta(
        new Richiesta(gerarchia.getFoglie().get(1), DefaultInitializer.COMMERCIO_VAL_01),
        new Offerta(gerarchia.getFoglie().get(2))
    );
    PropostaAperta proposta02 = createProposta(
        new Richiesta(gerarchia.getFoglie().get(2), DefaultInitializer.COMMERCIO_VAL_02),
        new Offerta(gerarchia.getFoglie().get(0))
    );
    aggiungiProposteAperte(insiemeAperto0, proposta00, proposta01, proposta02);

    // Verifica che ci siano 3 proposte chiuse nell'insieme chiuso
    assertEquals(3, commercio.getInsiemiChiusi().get(0).getProposteChiuse().size());
}
```

- Scopo: Verifica la capacità del sistema di chiudere 3 proposte aperte.
- Implementazione: Viene creato un insieme aperto, e sono aggiunte 3 proposte chiudibili. Successivamente, si verifica che ci sono esattamente 3 proposte nell'insieme chiuso.

```
// Metodo di utilità per aggiungere ProposteAperte a un InsiemeAperto e gestire il commercio
private void aggiungiProposteAperte(InsiemeAperto insiemeAperto, PropostaAperta... proposte) {
    for (PropostaAperta proposta : proposte) {
        insiemeAperto.addPropostaAperta(proposta);
    }
    commercio.addInsiemiAperti(insiemeAperto); // Aggiunge l'insieme aperto al commercio
    commercio.cercaProposteChiudibili(insiemeAperto); // Cerca le proposte chiudibili nell'ins
}
```

- “aggiungiProposteAperte” è un metodo di utilità richiamato nel test per aggiungere proposte aperte ed effettuarne la chiusura (se chiudibili), richiamando l’algoritmo implementato tramite cercaProposteChiudibili().
- Nel test precedente, l’implementazione interna di cercaProposteChiudibili() citato è trattata come una scatola nera. Eventuali analisi del codice dello stesso riferisce test White Box.

Boundary Value Analysis

CommercioRegoleTest

- Ogni test esplora un aspetto specifico dei limiti di input e classi di equivalenza.
- Ad esempio: assenza di proposte, durate estreme (massimo valore intero), valore zero e valori negativi per le proposte.

```
@Test
void test_Chiudi_NessunaProposta() {
    // Test per verificare che non ci siano proposte da chiudere
    InsiemeAperto insiemeAperto0 = new InsiemeAperto(fruitore0.getCommercio().getInsiemiAperti());
    fruitore0.getCommercio().cercaProposteChiudibili(insiemeAperto0);
    assertEquals(0, fruitore0.getCommercio().getInsiemiChiusi().size());
}
```

- A. Scopo: Verifica che non ci siano proposte da chiudere.
- B. Implementazione: Viene creato un insieme aperto senza aggiungere proposte. Successivamente, si verifica che ci sono 0 insiemi chiusi.

A. Scopo: Verifica che sia possibile chiudere proposte con durate di valori estremi.

B. Implementazione:
Sono create due proposte aperte con valori di richiesta e offerta che includono il massimo valore intero possibile (Integer.MAX_VALUE).

```
@Test  
void test_Chiudi_ProposteConValoriEstremi() {  
    // Test per verificare che sia possibile chiudere proposte con valori estremi  
    InsiemeAperto insiemeAperto0 = new InsiemeAperto(fruitore0.getComprensorioAppartenenza());  
  
    int val = Integer.MAX_VALUE - 1;  
    PropostaAperta proposta00 = createProposta(  
        new Richiesta(gerarchia.getFoglie().get(1), val),  
        new Offerta(gerarchia.getFoglie().get(0))  
    );  
  
    PropostaAperta proposta01 = createProposta(  
        new Richiesta(gerarchia.getFoglie().get(0), proposta00.getOfferta().getDurata()),  
        new Offerta(gerarchia.getFoglie().get(1))  
    );  
  
    aggiungiProposteAperte(insiemeAperto0, proposta00, proposta01);  
  
    // Verifica che le proposte siano chiuse correttamente anche con valori estremi  
    assertEquals(2, commercio.getInsiemiChiusi().get(0).getProposteChiuse().size());  
}
```

Successivamente, si verifica che ci siano esattamente 2 proposte nell' insieme chiuso.

Equivalence Partitioning Test

FattoriDiConversioneTest

```
/*
 * @Test
void test_FattoreConversione_TraFoglieDiStessoAlbero() throws Exception {
    // Creazione della radice e dei nodi foglia
    NotLeaf nodo0 = creaNodoRadice(DefaultInitializer.ROOT_NAME_0, DefaultInitializer.ROOT_FIELD_0,
        Leaf nodo00 = creaNodoFoglia(DefaultInitializer.CHILD_NAME_00, DefaultInitializer.ROOT_NAME_0);
    Leaf nodo01 = creaNodoFoglia(DefaultInitializer.CHILD_NAME_01, DefaultInitializer.ROOT_NAME_0);
    Leaf nodo10 = creaNodoFoglia(DefaultInitializer.CHILD_NAME_10, DefaultInitializer.ROOT_NAME_0);

    // Aggiunta dei nodi foglia alla radice
    nodo0.addChild(nodo00);
    nodo0.addChild(nodo01);
    nodo0.addChild(nodo10);

    // Aggiunta dei fattori di conversione
    addFattoreConversione(nodo00, nodo01, DefaultInitializer.FACTOR_VAL_00);
    addFattoreConversione(nodo00, nodo10, DefaultInitializer.FACTOR_VAL_00);

    // Creazione dell'albero e esecuzione del test
    creaAlbero(nodo0);

    // Verifica del risultato atteso
    assertEquals(1 / DefaultInitializer.FACTOR_VAL_00 * DefaultInitializer.FACTOR_VAL_10, nodo01.fat
}
```

A. Scopo: calcolo del fattore di conversione tra Leaf che appartengono allo stesso albero, considerata l'appartenenza del fattore al range corretto.

B. Implementazione: Viene creata una radice con 3 Leaf (A, B e C), e vengono aggiunti I fattori di conversione tra A-B e A-C. Successivamente il test verifica che il fattore di conversione tra B-C sia corretto.

Boundary Value Analysis

FattoriDiConversioneTest

A. Scopo: Verifica che il fattore di conversione tra due Leaf sia 0 quando non viene aggiunto nessun valore.

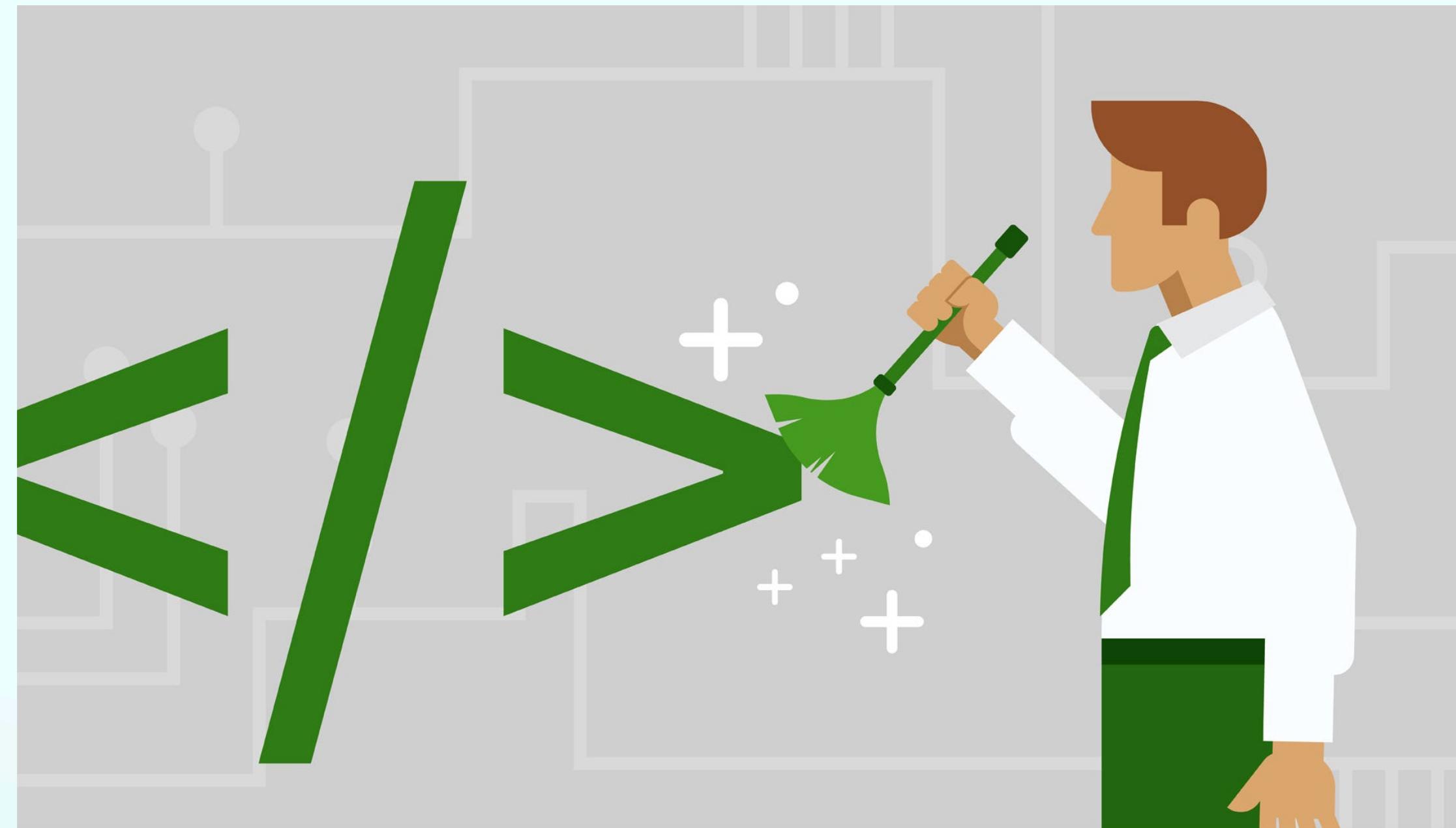
B. Implementazione: Viene creata una radice con due Leaf, ma non viene aggiunto nessun fattore di conversione tra le due. Successivamente il test verifica che il fattore di conversione sia effettivamente 0.

```
@Test
void test_FattoreConversione_Null() throws Exception {
    // Creazione della radice e dei nodi foglia
    NotLeaf nodo0 = creaNodoRadice(DefaultInitializer.ROOT_NAME_);
    Leaf nodo00 = creaNodoFoglia(DefaultInitializer.CHILD_NAME_0);
    Leaf nodo01 = creaNodoFoglia(DefaultInitializer.CHILD_NAME_0);

    // Aggiunta dei nodi foglia alla radice
    nodo0.addChild(nodo00);
    nodo0.addChild(nodo01);

    // Non aggiungiamo alcun fattore di conversione
    // Creazione dell'albero e esecuzione del test
    creaAlbero(nodo0);

    // Verifica del risultato atteso
    assertEquals(0, nodo00.fattoreFoglia(nodo01));
}
```



Refactoring

- Il refactoring è il processo di miglioramento del codice esistente senza alterarne il comportamento esterno.
- L'obiettivo è rendere il codice più leggibile, mantenibile e flessibile (a discapito di un peggioramento delle prestazioni).
- È una pratica continua che aiuta a prevenire il deterioramento del codice nel tempo.

Extract Class

- La classe *Nodo* sta lavorando per due: gestisce i *Nodi* e i *FattoriDiConversione*.
- Molti attributi e metodi devono essere spostati, la classe deve essere estratta.

```
11
12+/** 
13 * La classe Nodo rappresenta un nodo all'interno di un albero.
14 * Ogni nodo può essere una foglia o una non foglia.
15 * Se un nodo è una foglia, contiene i fattori di conversione verso altri nodi.
16 * Se un nodo è una non foglia, può avere figli e contiene informazioni sul campo e sul dominio.
17 *
18 * Autore: Daniele Martinelli e Federico Sabbadini
19 */
20 public class Nodo implements Serializable {
21
22     private static final long serialVersionUID = 1L;
23     private String nome;
24     private boolean isLeaf;
25     private String root;
26     private List<Nodo> children;
27     private String campo;
28     private List<String[]> dominio;
29     private static final double MIN_FATTORECONVERSIONE = 0.5;
30     private static final double MAX_FATTORECONVERSIONE = 2.0;
31
32     private HashMap<Nodo, Double> fattori;
33
34+    * Costruttore non foglia.□
35+    public Nodo(String nome, String root, String campo) {□
36
37+    * Costruttore foglia.□
38+    public Nodo(String nome, String root) {□
39
40
41+    public Map<Nodo, Double> getFattori() {□
42
43+        * Verifica se esiste un fattore di conversione con una data foglia.□
44+        public double fattoreFoglia(Nodo nodo) {□
45
46+            * Aggiunge un fattore di conversione associato al nodo.□
47+            public void addFattoreConversione(Nodo foglia, Double fattore) {□
48
49+                * Verifica se un dato fattore di conversione è valido.□
50+                public static boolean verificaFattoreConversione(double fattore) {□
51
52+                    * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie nella gerarchia.□
53+                    public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {□
54
55+                        * Metodo per calcolare il fattore di conversione transitivo tra due nodi.□
56+                        private static Double calcTransitivo(Nodo nodo1, Nodo nodo2, List<Nodo> visitati) {□
57
58
59
60
61
62
63
64
65
66+    public Map<Nodo, Double> getFattori() {□
67
68+        * Verifica se esiste un fattore di conversione con una data foglia.□
69+        public double fattoreFoglia(Nodo nodo) {□
70
71+            * Aggiunge un fattore di conversione associato al nodo.□
72+            public void addFattoreConversione(Nodo foglia, Double fattore) {□
73
74+                * Verifica se un dato fattore di conversione è valido.□
75+                public static boolean verificaFattoreConversione(double fattore) {□
76
77+                    * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie nella gerarchia.□
78+                    public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {□
79
80+                        * Metodo per calcolare il fattore di conversione transitivo tra due nodi.□
81+                        private static Double calcTransitivo(Nodo nodo1, Nodo nodo2, List<Nodo> visitati) {□
82
83
84
85
86
87
88
89
90
91
92
93+    public Map<Nodo, Double> getFattori() {□
94
95+        * Verifica se esiste un fattore di conversione con una data foglia.□
96+        public double fattoreFoglia(Nodo nodo) {□
97
98+            * Aggiunge un fattore di conversione associato al nodo.□
99+            public void addFattoreConversione(Nodo foglia, Double fattore) {□
100
101
102
103+        * Verifica se un dato fattore di conversione è valido.□
104+        public static boolean verificaFattoreConversione(double fattore) {□
105
106+            * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie nella gerarchia.□
107+            public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {□
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122+        * Metodo per calcolare il fattore di conversione transitivo tra due nodi.□
123+        private static Double calcTransitivo(Nodo nodo1, Nodo nodo2, List<Nodo> visitati) {□
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
```

```
5  /**
6   * La classe Nodo rappresenta un nodo all'interno di un albero.
7   * Ogni nodo può essere una foglia o una non foglia.
8   * Se un nodo è una foglia, contiene i fattori di conversione verso
9   * Se un nodo è una non foglia, può avere figli e contiene informazi
10  *
11  * Autore: Daniele Martinelli e Federico Sabbadini
12  */
13 public abstract class Nodo implements Serializable {
14
15     * Autore: Daniele Martinelli e Federico Sabbadini
16     private static final long serialVersionUID = 1L;
17     protected String nome;
18     protected String root;
19
20     * Restituisce il nome del nodo. ..
21     public String getNome() { ..
22
23     * Verifica se il nodo è una foglia. ..
24     public abstract boolean isLeaf(); ..
25
26     * Verifica se il nodo è una radice. ..
27     public abstract String root(); ..
28
29     * Verifica se esiste un nodo non radice con il dato nome sotto
30     public boolean verificaEsistenzaNome(String nome) { ..
```

```
10⊕ /**
11  * Classe per la gestione dei fattori di conversione.
12  *
13  * Autore: Daniele Martinelli e Federico Sabbadini
14  */
15 public class FattoriDiConversione implements Serializable{
16
17     private static final long serialVersionUID = 1L;
18     public static final double MIN_FATTORECONVERSIONE = 0.5;
19     public static final double MAX_FATTORECONVERSIONE = 2.0;
20
21     private HashMap<Leaf, Double> fattori;
22
23⊕     public Set<Entry<Leaf,Double>> getFattori() {..}
24
25⊕     public FattoriDiConversione() {..}
26
27⊕     * Verifica se esiste un fattore di conversione con una data foglia. ..
28⊕     public double fattoreFoglia(Leaf foglia) {..}
29
30
31⊕     * Aggiunge un fattore di conversione associato al nodo. ..
32⊕     public void addFattoreConversione(Leaf foglia, Double fattore) {..}
33
34⊕     * Verifica se un dato fattore di conversione è valido. ..
35⊕     public static boolean verificaFattoreConversione(double fattore) {..}
36
37
38⊕     * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie.
39⊕     public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {..}
```

- Si crea una nuova classe FattoriDiConversione e si spostano gli attributi e i metodi rilevanti dalla vecchia classe Nodo alla nuova.

Extract Method

- Alcuni frammenti di codice richiedono un commento. In questo caso è bene aumentare la chiarezza di ciò che accade nelle condizioni del metodo calcTransitivo.

- I frammenti sono spostati dentro metodi i cui nomi ne spiegano lo scopo, e vengono richiamati in calcTransitivo.

```
91* * Metodo per calcolare il fattore di conversione transitivo tra due nodi.□
92private static Double calcTransitivo(Leaf nodo1, Leaf nodo2, List<Leaf> visitati) {
93
94    if (nodo1.equals(nodo2)) {
95        return getFattoreIdentità();
96
97    } else if (nodo1.fattoreFoglia(nodo2) != 0) {
98        return getFattoreDiretto(nodo1, nodo2);
99
100    } else {
101        return getFattoreIndiretto(nodo1, nodo2, visitati);
102    }
103}
104
105
106
107
108
109
110
111
```

```
/*
 * Metodo per calcolare il fattore di conversione transitivo tra due nodi.
 *
 * @param nodo1 Il primo nodo
 * @param nodo2 Il secondo nodo
 * @param visitati Lista dei nodi visitati durante il calcolo
 * @return Il fattore di conversione transitivo tra i due nodi, null se non è possibile calcolarlo
 */
private static Double calcTransitivo(Leaf nodo1, Leaf nodo2, List<Leaf> visitati) {
    FattoriDiConversione fact = nodo1.getFattori();
    if (nodo1.equals(nodo2)) { ①
        return 1.0;
    } else if (fact.fattoreFoglia(nodo2) != 0) { ②
        return fact.fattoreFoglia(nodo2);
    } else {
        if (fact.isEmpty())
            return null;
        for (Map.Entry<Leaf, Double> entry : fact.getFattori()) {
            Leaf key = entry.getKey();
            if (!visitati.contains(key)) {
                visitati.add(key); ③
                Double val = calcTransitivo(key, nodo2, visitati);
                if (val != null)
                    return entry.getValue() * val;
            }
        }
    }
    return null; // Restituiamo null se non è possibile calcolare il fattore di conversione
}
```

```
.12*
.13    return 1.0; ①
.14
.15
.16*
.17    return nodo1.fattoreFoglia(nodo2); ②
.18
.19
.20*
.21    Set<Entry<Leaf,Double>> fact = nodo1.getFattori();
.22
.23    if (fact.isEmpty())
.24        return null; ③
.25
.26    for (Map.Entry<Leaf, Double> entry : fact) {
.27        Leaf key = entry.getKey();
.28        if (!visitati.contains(key)) {
.29            visitati.add(key);
.30            Double val = calcTransitivo(key, nodo2, visitati);
.31            if (val != null)
.32                return entry.getValue() * val;
.33        }
.34    }
.35
.36    return null;
.37
.38 }
```

- Alcuni frammenti di codice richiedono un commento. In questo caso è bene aumentare la chiarezza di ciò che accade nelle condizioni del metodo [addTransitivoFattoreConversione](#).

```


    /**
     * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie nella gerarchia.
     *
     * @param gerarchia La gerarchia su cui operare
     */
    public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {

        for (Leaf nodo1 : gerarchia.getFoglie()) {
            for (Leaf nodo2 : gerarchia.getFoglie()) {
                if (!nodo1.equals(nodo2) && nodo1.getFattori().fattoreFoglia(nodo2) == 0) {
                    Double fattore = calcTransitivo(nodo1, nodo2, new ArrayList<>()); ①
                    if (fattore != null) {
                        nodo1.addFattoreConversione(nodo2, fattore);
                    }
                }
            }
        }
    }


```

- I frammenti sono spostati dentro metodi i cui nomi ne spiegano lo scopo, e vengono richiamati in [addTransitivoFattoreConversione](#).

```


64⊕   * Aggiunge i fattori di conversione transitivi tra TUTTE le coppie di foglie nella gerarchia.□
68⊕ public static void addTransitivoFattoreConversione(Gerarchia gerarchia) {
69
70    for (Leaf nodo1 : gerarchia.getFoglie()) {
71        for (Leaf nodo2 : gerarchia.getFoglie()) {
72            if (shouldAddTransitivoFattore(nodo1, nodo2)) {
73                addFattoreIfCalculated(nodo1, nodo2);
74            }
75        }
76    }
77
78
79⊕ private static boolean shouldAddTransitivoFattore(Leaf nodo1, Leaf nodo2) {
80    return !nodo1.equals(nodo2) && nodo1.fattoreFoglia(nodo2) == 0; ④
81}
82
83⊕ private static void addFattoreIfCalculated(Leaf nodo1, Leaf nodo2) {
84    Double fattore = calcTransitivo(nodo1, nodo2, new ArrayList<>());
85    if (fattore != null) {
86        nodo1.addFattoreConversione(nodo2, fattore); ②
87    }
}


```

Hide Delegate

- Si sta chiamando il metodo `getFattori()` di un oggetto `FattoriDiConversione` (`fact` = oggetto delegate), ottenuto chiamando il metodo `getFattori()` su un oggetto `Leaf`.

```
    private static Double calcTransitivo(Leaf nodo1, Leaf nodo2, List<Leaf> visitati) {
        FattoriDiConversione fact = nodo1.getFattori();
        if (nodo1.equals(nodo2)) {
            return 1.0;
        } else if (fact.fattoreFoglia(nodo2) != 0) {
            return fact.fattoreFoglia(nodo2);
        } else {
            if (fact.isEmpty())
                return null;
            for (Map.Entry<Leaf, Double> entry : fact.getFattori()) {
                Leaf key = entry.getKey();
                if (!visitati.contains(key)) {
                    visitati.add(key);
                    Double val = calcTransitivo(key, nodo2, visitati);
                    if (val != null) {
                        return entry.getValue() * val;
                    }
                }
            }
        }
        return null; // Restituiamo null se non è possibile calcolare il fattore di conversione
    }
```

```
5 public class Leaf extends Nodo implements Serializable {
6
7     private static final long serialVersionUID = 1L;
8     private FattoriDiConversione fattori;
9
10    /**
11     * Restituisce il dominio associato al nodo.
12     *
13     * @return Il dominio associato al nodo.
14     */
15    public FattoriDiConversione getFattori() {
16        return fattori;
17    }
18}
```

- Il `FattoreDiConversione` è memorizzato in uno degli attributi (fattori) della classe `Leaf`.
- Il metodo `getFattori()` di `Leaf` ritorna un `FattoriDiConversione`.

```

7 public class Leaf extends Nodo implements Serializable {
8
9     private static final long serialVersionUID = 1L;
10    private FattoriDiConversione fattori;
11
12    .
13
14    public Set<Entry<Leaf,Double>> getFattori() {
15        return fattori.getFattori();
16    }
17
18

```

- Si modifica il metodo *getFattori* di *Leaf* per nascondere il delegate.

- Il return del metodo non è *FattoriDiConversione*, ma un *Set* (che poi viene utilizzato).

```

.19
.20    private static Double getFattoreIndiretto(Leaf nodo1, Leaf nodo2, List<Leaf> visitati) {
.21        Set<Entry<Leaf,Double>> fact = nodo1.getFattori();
.22
.23        if (fact.isEmpty())
.24            return null;
.25
.26        for (Map.Entry<Leaf, Double> entry : fact) {
.27            Leaf key = entry.getKey();
.28            if (!visitati.contains(key)) {
.29                visitati.add(key);
.30                Double val = calcTransitivo(key, nodo2, visitati);
.31                if (val != null)
.32                    return entry.getValue() * val;
.33            }
.34        }
.35        return null;
.36    }
.37
.38 }
.39

```