Scuola di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

Ingegneria del Software

Relazione Progetto:

Copia virtuale di un sistema Hardware

Pagliocca Marco

Anno Accademico 2018/2019

1. Introduzione

1.1. Intento

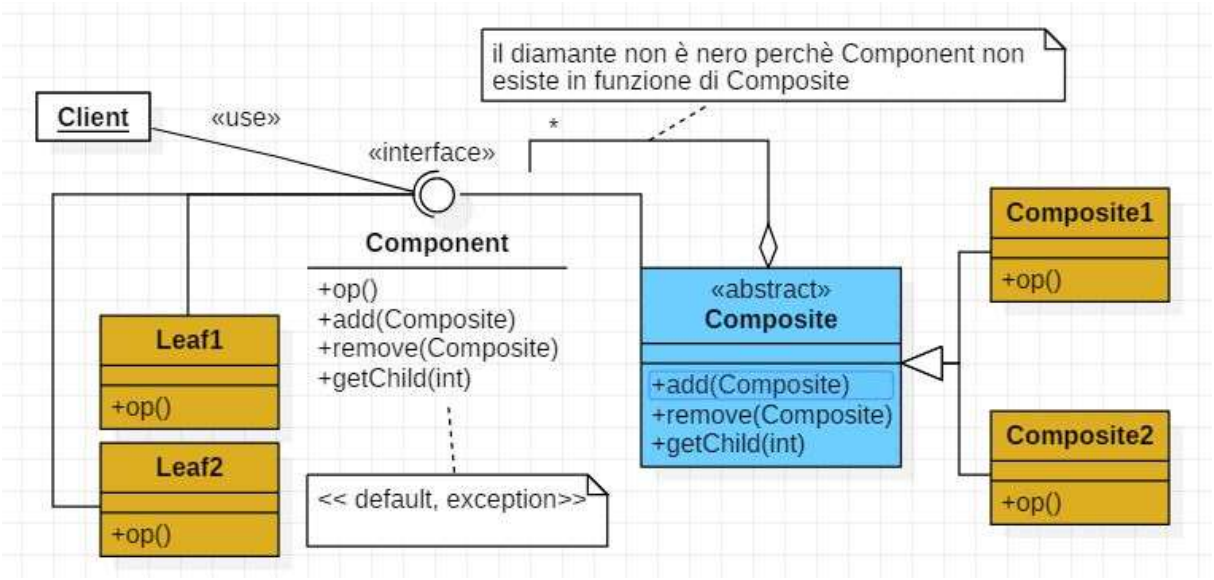
L'intento è di realizzare una copia virtuale di un sistema complesso, senza che il Client abbia conoscenza di come esso sia strutturato, ma solamente di quali parti sia composto.

A tal proposito si vuole costruire il sistema complesso, e potenzialmente dinamico, mediante il pattern *Composite* e lo si vuole adattare alla vista del Client mediante il pattern Adapter, nella versione *Object Adapter*. Poiché il Client è all'oscuro della complessità del sistema, si rende necessario delegare la sua costruzione ad un *Builder* pattern.

1.2. Composite pattern

L'intento di questo pattern strutturale è di comporre oggetti per rappresentare gerarchie, in modo tale che i client non riescano a distinguere gli oggetti concreti, cioè quelli terminali, dagli oggetti astratta utilizzati per tenere insieme la composizione.

Il problema del Composite nasce non appena si considerano le operazioni dei due tipo di oggetti. Ipotizziamo che le foglie abbiano una operazione *op()*, mentre Composite definiscano op() dei figli ad esso connessi e abbiano anche operazioni per modificare la struttura gerarchica: *add(Component), remove(Component), getChild(int),* etc. Poiché entrambi implementano l'interfaccia, tutte le operazioni vengono inserite in Component, e dunque anche in Leaf in modo da essere indistinguibile da un Component. Tuttavia Leaf non può utilizzare queste operazioni, pertanto Component verrà definita come classe astratta dove ai metodi verrà fornita una fake implementation. Da Java 8 il problema della single-inheritance è risolto grazie al default implementation: Component torna ad essere un'interfaccia dove le *children related operation* hanno una implementazione di default.



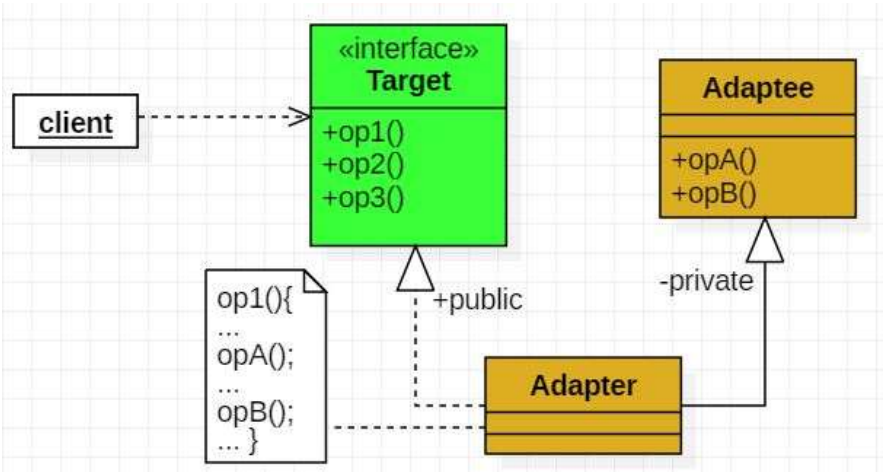
Durante la programmazione si vedono le classi e non gli oggetti, pertanto tenere bene a mente la struttura gerarchica degli oggetti è fondamentale per conoscere il comportamento, il debugging e i test. Per facilitare l'attraversamento nella struttura ad albero si può aggiungere nell'interfaccia un metodo *getParent()* che restituisce il riferimento all'unico padre di ogni Component. In tal caso è altrettanto necessario definire un comportamento speciale per la radice.

1.3. Adapter pattern

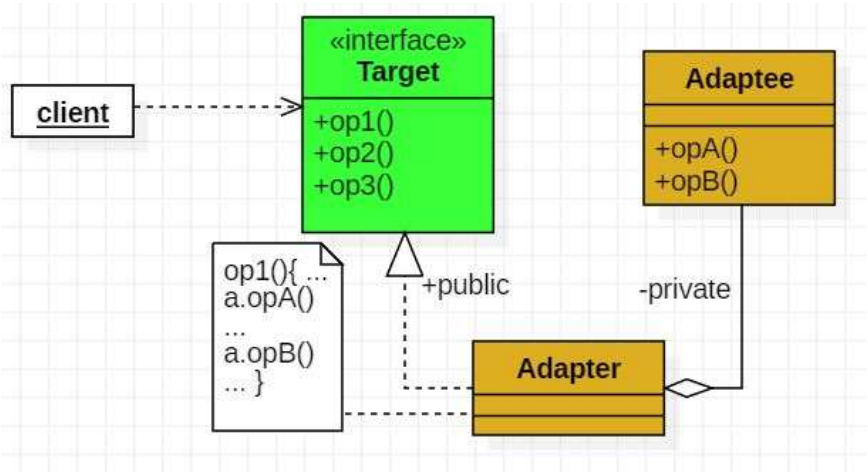
L'intento è di convertire l'interfaccia di una classe già esistente, detta *Adaptee*, che espone dei metodi, in un'altra interfaccia *Target* che un Client si aspetta. Per un qualche motivo il Client non può usare direttamente le operazioni di Adaptee (per esempio non corrisponde la signature o le operazioni aggregate), e quest'ultimo non si può modificare. È pertanto un pattern strutturale.

Ne esistono di due tipologie che differiscono nel modo in cui riusano la classe esistente:

1. **Class Adapter** è basata sull'ereditarietà, cioè implementa l'interfaccia del target estendendo Adaptee. Di conseguenza:
    a. Utilizza una singola istanza come oggetto, semplice da creare e testare;
    b. Richiede di sparare l'unica cartuccia dell'ereditarietà;
    c. Utilizza i metodi direttamente, senza farne override.



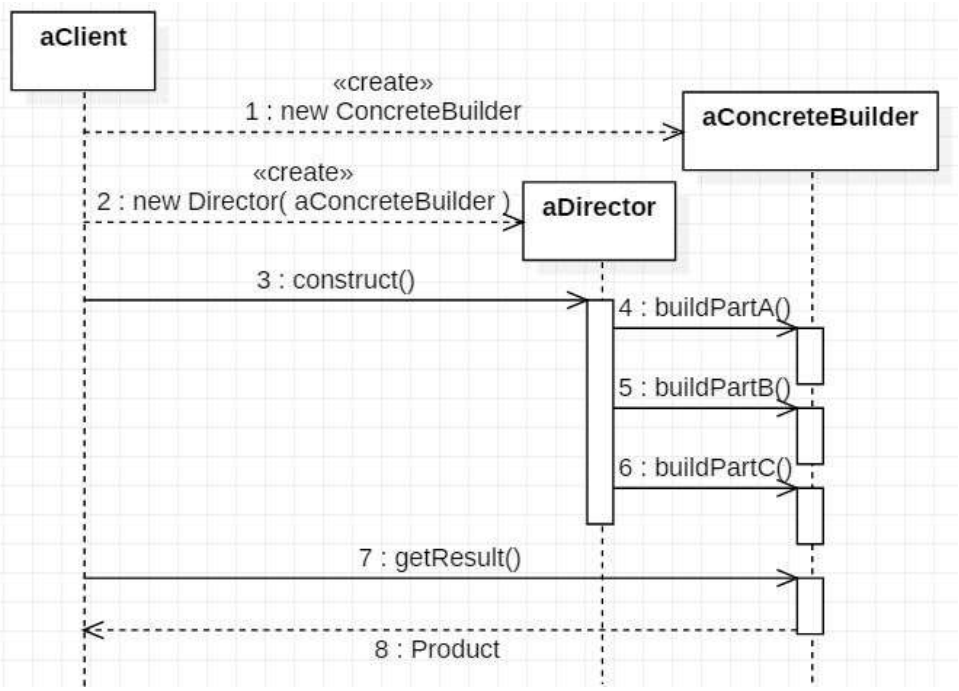
2. **Object Adapter** è basato sulla composizione, cioè contiene un riferimento ad un oggetto Adaptee. Di conseguenza:
    a. Utilizza due oggetti istanziati, dunque ci sono più test;
    b. Permette di aggregare una classe astratta di Adaptee in modo da poter essere sostituita dinamicamente, il che è un bene se si vuole usarlo, ma un male, perché più complesso, se non lo si utilizza;
    c. Utilizza i metodi per forwarding;
    d. Presenta il problema del self.



1.4. Builder pattern

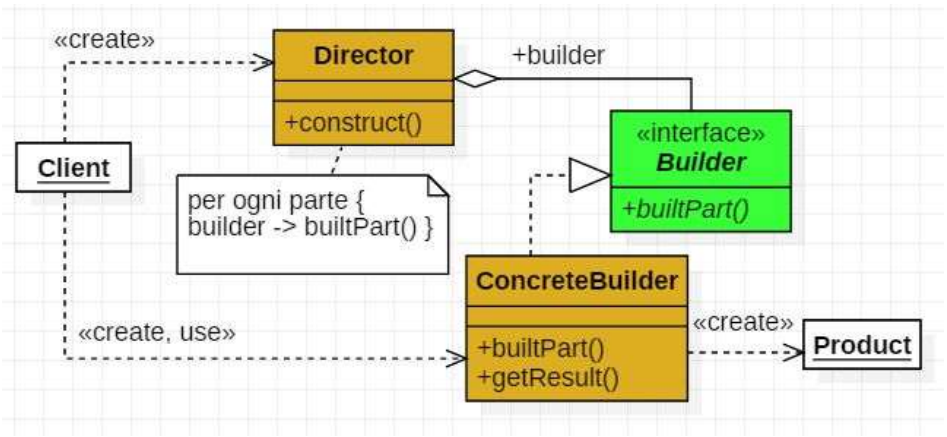
L'intento di questo pattern creazionale è separare la costruzione di un oggetto complesso dalla sua rappresentazione così che si possano mettere in vita complesse configurazioni differenti.

La **motivazione** sta nella separazione delle responsabilità. Ad esempio assemblare un Composite con differenti tipi di Foglie.



Tra i partecipanti vi sono:

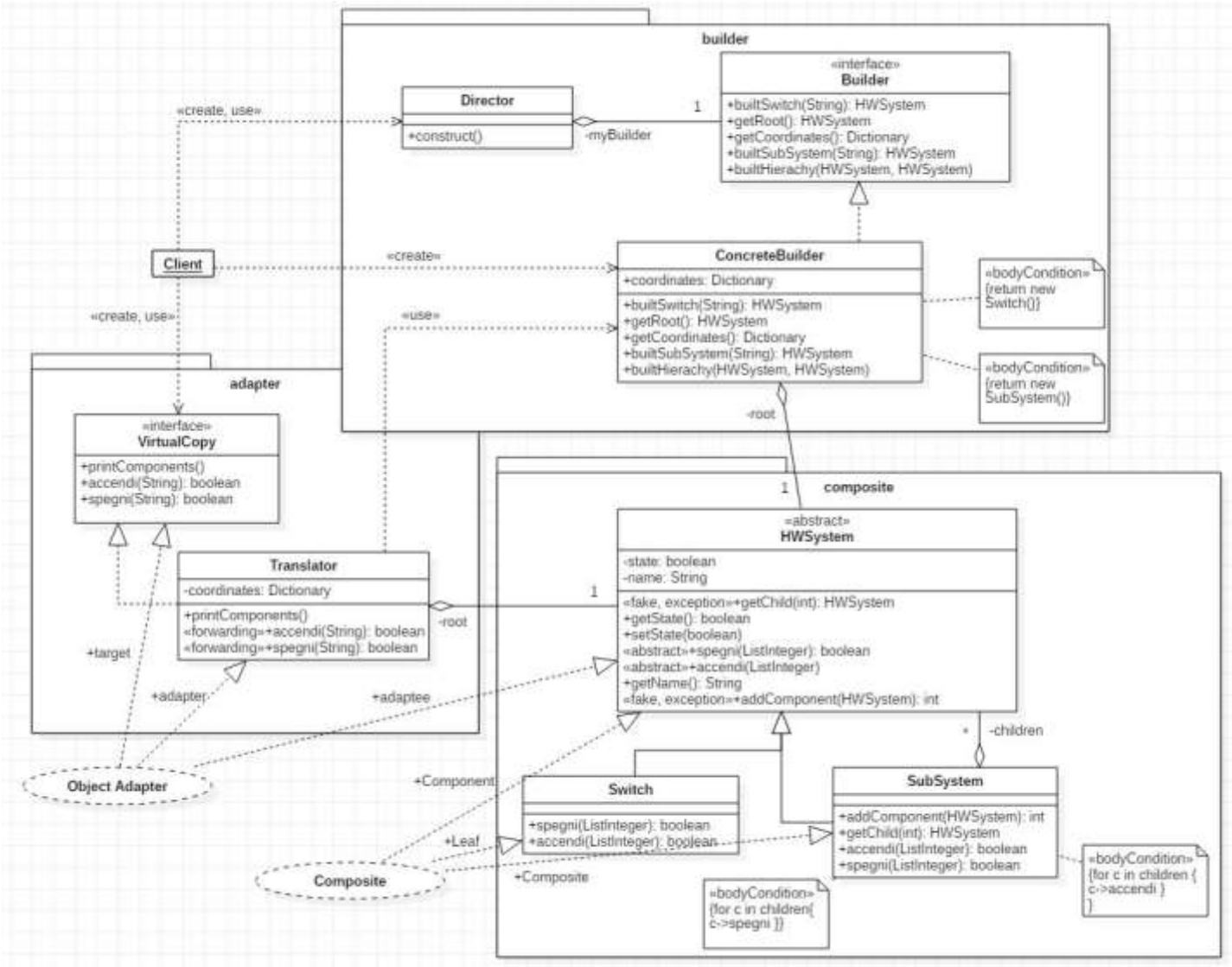
- *Builder*: specifica l'interfaccia per la creazione delle parti di *Product*;
- *ConcreteBuilder*: implementa l'interfaccia del Builder e mantiene lo stato dei Product che sta costruendo; inoltre provvede ad un metodo per cambiare il tipo di prodotto;
- *Director*: determina la sequenza delle parti da costruire e come disporle mediante invocazione di *construct()*;
- *Client*: mette in vita il Director e gli fornisce un ConcreteBuilder. Alla fine delle costruzione recupera il Product dal ConcreteBuilder attraverso *getResult()*.



1.5. Scenario

La motivazione principale è la costruzione di un insieme di impostazioni che possono essere accese oppure spente, come se fossero degli interruttori, i quali agiscono direttamente su un sistema hardware, convertendo la richiesta ad un (o più) segnale elettrico. Seguendo l'intento, lo scenario presenta un complesso sistema gerarchico, per il quale un'operazione ad alto livello si propaga a tutte le sue componenti. Ad esempio, se viene disabilitato l'audio di un personal computer, automaticamente vengono disabilitate anche le notifiche. In pratica si vuole far collassare una struttura gerarchica in una struttura linearizzata.

2. Implementazione



2.1. Partecipanti

I partecipanti sono:

HWSystem rappresenta la classe astratta del sistema fisico, costituita dagli attributi:

- *state*, un booleano che specifica se la componente è accesa (true) o spenta (false);
- *name*, una stringa che dettaglia il nome della componente, così come verrà presentata al Client.

E dai metodi:

- *getState()*, restituisce il valore di state;
- *setState(boolean)*, imposta un nuovo valore per state;
- *getName()*, restituisce il valore di name;
- *accendi(ArrayList <Integer>)*, metodo astratto per accendere una componente;
- *spegni(ArrayList <Integer>)*, metodo astratto per spegnere una componente;
- *addComponent(HWSystem)*, definisce una fake implementation che lancia una *Exception*;
- *getChild(int)*, definisce una fake implementation che lancia una *Exception*;

Il progetto non contempla la possibilità di rimuovere una componente.

Switch estende *HWSystem* lasciando le fake implementation delle children-related operation e implementando i metodi astratti:

- *accendi(ArrayList <Integer>),* controlla che la lista di interi sia vuota e imposta il proprio state a true. Nel caso la lista non fosse vuota, vuol dire che l'indirizzo è sbagliato e tenta di accedere ad un figlio di una foglia, per cui viene lanciata una *Exception*;
- *spegni(ArrayList <Integer>),* controlla che la lista di interi sia vuota e imposta il proprio state a false. Nel caso la lista non fosse vuota, vuol dire che l'indirizzo è sbagliato e tenta di accedere ad un figlio di una foglia, per cui viene lanciata una *Exception*.

SubSystem estende *HWSystem* e ne implementa sia i metodi astratti che quelli con fake implementation:

- *accendi(ArrayList <Integer>)*, controlla che la lista di interi sia vuota e invoca il metodo *accendi()* per ogni sua sotto-componente, per poi impostare il proprio state a true. Nel caso la lista non fosse vuota, elimina il primo valore della lista per inoltrare la chiamata ad *accendi()* della medesima sotto-componente;
- *spegni(ArrayList <Integer>)*, controlla che la lista di interi sia vuota e invoca il metodo *spegni()* per ogni sua sotto-componente, per poi impostare il proprio state a false. Nel caso la lista non fosse vuota, elimina il primo valore della lista per inoltrare la chiamata ad *spegni()* della medesima sotto-componente;
- *addComponent(HWSystem)*, aggiunge il parametro fornito come sua sotto-componente e ne restituisce la posizione;
- *getChild(int)*, restituisce la sotto-componente della posizione specificata dall'intero.

Presenta l'attributo *children*, un ArrayList <HWSystem> per contenere le sotto-componenti.

Builder definisce l'interfaccia per la costruzione di nuove componenti hardware, in particolare fornisce i metodi astratti *builtSwitch(String), builtSubSystem(String)*, *getRoot(), getCoordinates(),* per recuperare i prodotti, e *builtHierachy(HWSystem, HWSystem).*

ConcreteBuilder viene messo in vita dal Client e implementa l'interfaccia Builder:

- *builtSwitch(String),* mette in vita e restituisce un Switch, a cui è associato un nome;
- *builtSubSystem(String),* mette in vita e restituisce un SubSystem, a cui è associato un nome;
- *getRoot(),* restituisce il riferimento alla radice del sistema hardware;
- *getCoordinates(),* restituisce una copia del suo attributo *coordinates*, un Hashtable<String, ArrayList<Integer>>, che etichetta ad ogni componente inseriva il percorso da seguire per raggiungerla, partendo dalla radice;
- *builtHierachy(HWSystem, HWSystem),* inoltra la chiamata a *addComponent()* del primo HWSystem e, attraverso il valore intero tornato, recupera e inserisce il percorso da seguire per raggiungere il secondo HWSystem partendo dalla radice.

Director si occupa di dirigere la costruzione definendo quali oggetti devono essere creati e a chi devono essere connessi. Contiene un *myBuilder*, ovvero un Builder al quale delegare il lavoro del suo unico metodo *construct().*

VirtualCopy è l'interfaccia richiesta dal Client per effettuare le operazioni di accensione e spegnimento. Contiene anche il metodo *printComponents()* per vedere su quali componenti sono applicabili le operazioni.

Translator definisce la classe adattatrice del sistema hardware per il target voluto dal Client. Contiene gli attributi:

- *root*, la radice del sistema hardware, recuperato tramite *getRoot()* del Builder fornito dal Client durante l'inizializzazione;
- *coordinates*, il sistema di coordinate per tradurre un nome in un cammino, anch'esso ricevuto dal Builder tramite *getCoordinates()* in fase di inizializzazione.

E implementa le operazioni della VirtualCopy invocando per forwarding quelle del root, ma non prima di aver recuperato il cammino giusto per ogni componente e fornendone una copia come parametro di *accendi()* e *spegni()*. Il metodo *printComponents()* stampa a video tutte le componenti.

2.2. Il Codice

```
package composite;

import java.util.ArrayList;

public abstract class HWSystem {
    @Override
    public boolean spegni(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            String whatHappened;
            if(this.getState() == false)
                whatHappened = " era già spento.";
            else whatHappened = " è stato spento.";
            this.setState(false);
            System.out.println(this.getName()+whatHappened);
            return true;
        }
        else throw new Exception("L'indirizzo per "+ this.getName()+ " è sbagliato.");
    }

    @Override
    public boolean accendi(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            String whatHappened;
            if(this.getState() == true)
                whatHappened = " era già acceso.";
            else whatHappened = " è stato acceso.";
            this.setState(true);
            System.out.println(this.getName()+whatHappened);
            return true;
        }
        else throw new Exception("L'indirizzo per "+ this.getName() + " è sbagliato.");
    }
}
    public abstract boolean accendi(ArrayList <Integer> order) throws Exception;
}
```

```
package composite;

import java.util.ArrayList;

public class Switch extends HWSystem {

    public Switch(String name) {
        super(false, name);
    }
```

```
    @Override
    public boolean spegni(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            String whatHappened;
            if(this.getState() == false)
                whatHappened = " era già spento.";
            else whatHappened = " è stato spento.";
            this.setState(false);
            System.out.println(this.getName()+whatHappened);
            return true;
        }
        else throw new Exception("L'indirizzo per "+ this.getName()+ " è sbagliato.");
    }

    @Override
    public boolean accendi(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            String whatHappened;
            if(this.getState() == true)
                whatHappened = " era già acceso.";
            else whatHappened = " è stato acceso.";
            this.setState(true);
            System.out.println(this.getName()+whatHappened);
            return true;
        }
        else throw new Exception("L'indirizzo per "+ this.getName() + " è sbagliato.");
    }
}
import java.util.ArrayList;

public class SubSystem extends HWSystem{
    private ArrayList <HWSystem> children;

    public SubSystem(String name) {
        super(false, name);
        this.children = new ArrayList <>();
    }

    @Override
    public int addComponent(HWSystem newComp){
        children.add(newComp);
        return children.indexOf(newComp);
    }

    @Override
    public HWSystem getChild(int num) {
        return children.get(num);
    }


    @Override
    public boolean spegni(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            boolean all = true;
            for (HWSystem c : children) {
                boolean response = c.spegni(order);
                all = all || response;
            }
            if (all == true) {              //solo se tutte solo spente allora si può considerare spento
                this.setState(false);
            }
            return all;
        } else {
            try {
                HWSystem newChild = getChild(order.get(0));
                ArrayList<Integer> newOrder = new ArrayList<>(order);
                newOrder.remove(0);
                return newChild.spegni(newOrder);
            }catch(Exception e) {
                System.out.println(e.getMessage());
                return false;
```

```
            }
        }
    }

    @Override
    public boolean accendi(ArrayList <Integer> order) throws Exception{
        if (order.isEmpty()) {
            boolean all = false;
            for (HWSystem c : children) {
                boolean response = c.accendi(order);
                all = all || response;
            }
            if (all == true) {              //solo se almeno uno è acceso allora si può considerare acceso
                this.setState(true);
            }
            return all;
        } else {
            try {
                HWSystem newChild = getChild(order.get(0));
                ArrayList<Integer> newOrder = new ArrayList<>(order);
                newOrder.remove(0);
                return newChild.accendi(newOrder);
            } catch(Exception e) {
                System.out.println(e.getMessage());
                return false;
            }
        }
    }
}
```

```
package builder;

import java.util.ArrayList;
import java.util.Hashtable;

import composite.HWSystem;

public interface Builder {
    public HWSystem builtSubSystem(String name) throws Exception;
    public HWSystem builtSwitch(String name) throws Exception;
    public void builtHierachy(HWSystem parent, HWSystem child) throws Exception;
    public HWSystem getRoot();
    public Hashtable<String, ArrayList<Integer>> getCoordinates();
}
```

```
package builder;

import java.util.ArrayList;
import java.util.Hashtable;

import composite.*;

public class ConcreteBuilder implements Builder{
    private final HWSystem root;
    private Hashtable<String, ArrayList<Integer>> coordinates;

    public ConcreteBuilder() {
        root = new SubSystem("PC");
        this.coordinates = new Hashtable<String, ArrayList<Integer>>();
        this.coordinates.put(root.getName(), new ArrayList<>());
    }

    @Override
    public HWSystem builtSubSystem(String name) throws Exception{
        return new SubSystem(name);
    }

    @Override
    public HWSystem builtSwitch(String name) throws Exception{
        return new Switch(name);
    }
```

```
    @Override
    public void builtHierachy(HWSystem parent, HWSystem child) throws Exception{
        ArrayList<Integer> newCoordinate;
        int next;
        if(parent == null) {
            newCoordinate = new ArrayList<>(coordinates.get(root.getName())); //recupera le coordinate del padre
            next = root.addComponent(child);
        }
        else {
            newCoordinate = new ArrayList<>(coordinates.get(parent.getName()));
            next = parent.addComponent(child);
        }
        newCoordinate.add(next);
        this.coordinates.put(child.getName(), newCoordinate);
    }
    @Override
    public HWSystem getRoot() {
        return this.root;
    }

    @Override
    public Hashtable<String, ArrayList<Integer>> getCoordinates(){
        return new Hashtable<String, ArrayList<Integer>>(coordinates);
    }
}
```

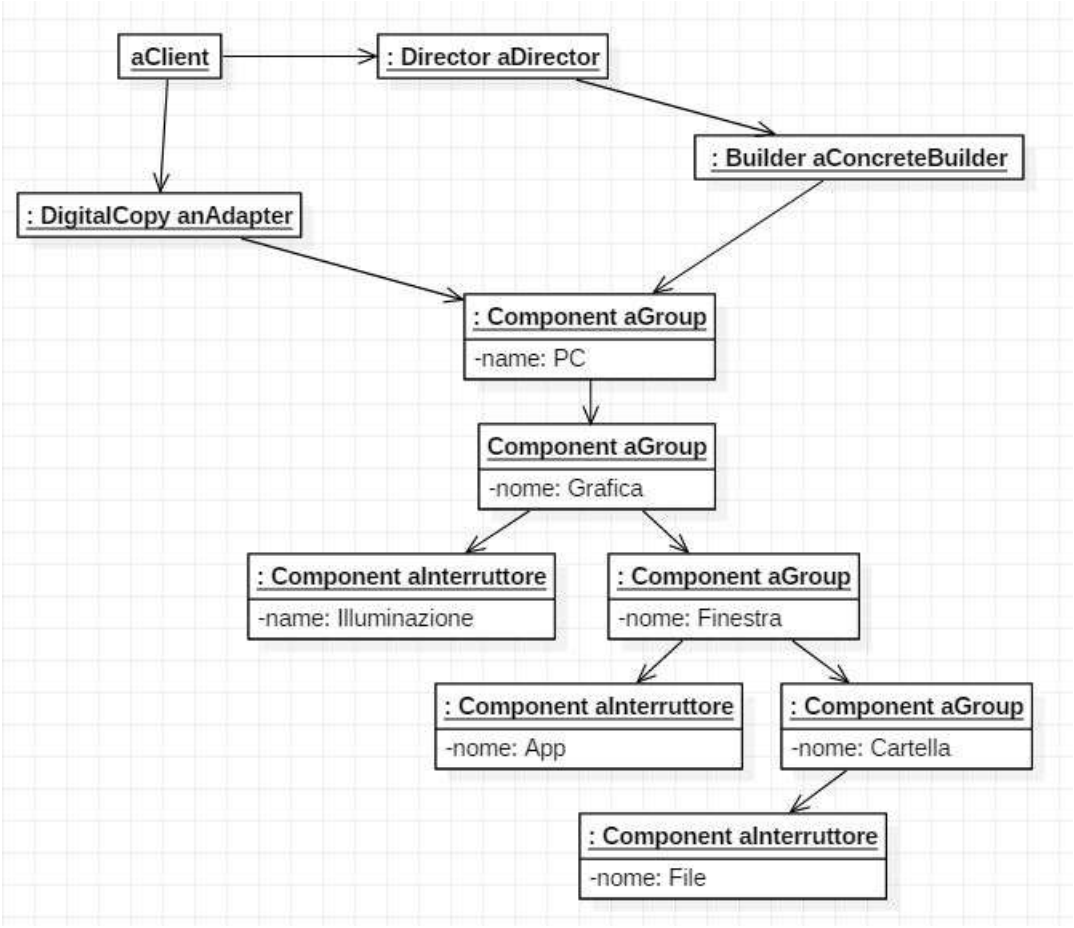
```
package builder;

import composite.HWSystem;

public class Director {
    private final Builder myBuilder;

    public Director(Builder builder) {
        this.myBuilder = builder;
    }

    public void construct() throws Exception{

        HWSystem grafica = myBuilder.builtSubSystem("Grafica");
        myBuilder.builtHierachy(null, grafica);

        HWSystem illuminazione = myBuilder.builtSwitch("Illuminazione");
        myBuilder.builtHierachy(grafica, illuminazione);

        HWSystem finestra = myBuilder.builtSubSystem("Finestra");
        myBuilder.builtHierachy(grafica, finestra);

        HWSystem app = myBuilder.builtSwitch("App");
        myBuilder.builtHierachy(finestra, app);

        HWSystem cartella = myBuilder.builtSubSystem("Cartella");
        myBuilder.builtHierachy(finestra, cartella);

        HWSystem file = myBuilder.builtSwitch("File");
        myBuilder.builtHierachy(cartella, file);

    }
}
```

```
package adapater;

public interface VirtualCopy {
    public void printComponents();
    public boolean accendi(String system) throws Exception;
    public boolean spegni(String system) throws Exception;
}
```

```java
package adapater;

import java.util.ArrayList;
import java.util.Hashtable;

import builder.Builder;
import composite.HWSystem;

public class Translator implements VirtualCopy {
    private Hashtable<String, ArrayList<Integer>> coordinates;
    private HWSystem root;

    public Translator(Builder cb) {
        this.coordinates = cb.getCoordinates();
        this.root = cb.getRoot();
    }

    @Override
    public void printComponents() {
        System.out.println(this.coordinates.keySet());
    }

    @Override
    public boolean accendi(String system) throws Exception {
        ArrayList <Integer> order = this.coordinates.get(system);
        //System.out.println(order);
        return this.root.accendi(new ArrayList<Integer>(order));
    }

    @Override
    public boolean spegni(String system) throws Exception {
        return this.root.spegni(new ArrayList<Integer>(this.coordinates.get(system)));
    }
}
```

3. Test

Precedentemente è stato detto che tenere bene a mente la struttura gerarchica degli oggetti è fondamentali per molte occasioni, tra le quali il debugging e i test. Di fatti durante la costruzione dei test per le classi costruite si è reso fondamentale conoscere quale fosse il Object diagram ad esso associato.

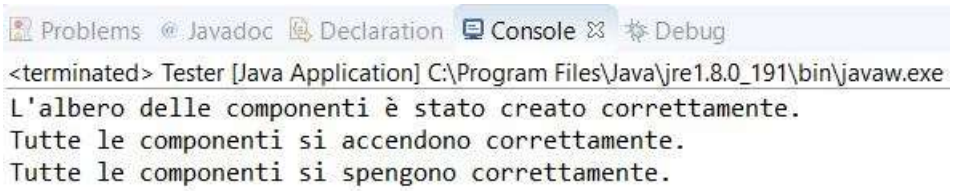


.

Nel particolare è interessante l'albero gerarchico costruito dal ConcreteBuilder sotto istruzione del Director. Tuttavia il diagramma è stato realizzato in via preventiva, come ci si aspetta che sia il sistema nel suo complesso. Per questo motivo è necessario verificarne la composizione.

I test che seguono sono definiti come metodi statici di classi testatrici e del package tester. Vengono invocati dal main e stampano su console una frase riassuntiva riguardante l'esito del test. Il primo test verifica che la struttura costruita dal Builder sia effettivamente quella supposta. In caso contrario i test successivi saranno inaffidabili. Il secondo e terzo insieme di test, invece, riguardano il controllo delle due operazioni principali del progetto: accendi() e spegni(). Viene controllato se il valore di state di ogni componente è concorde con l'operazione eseguita (true = acceso, false = spento).

3.1. Output



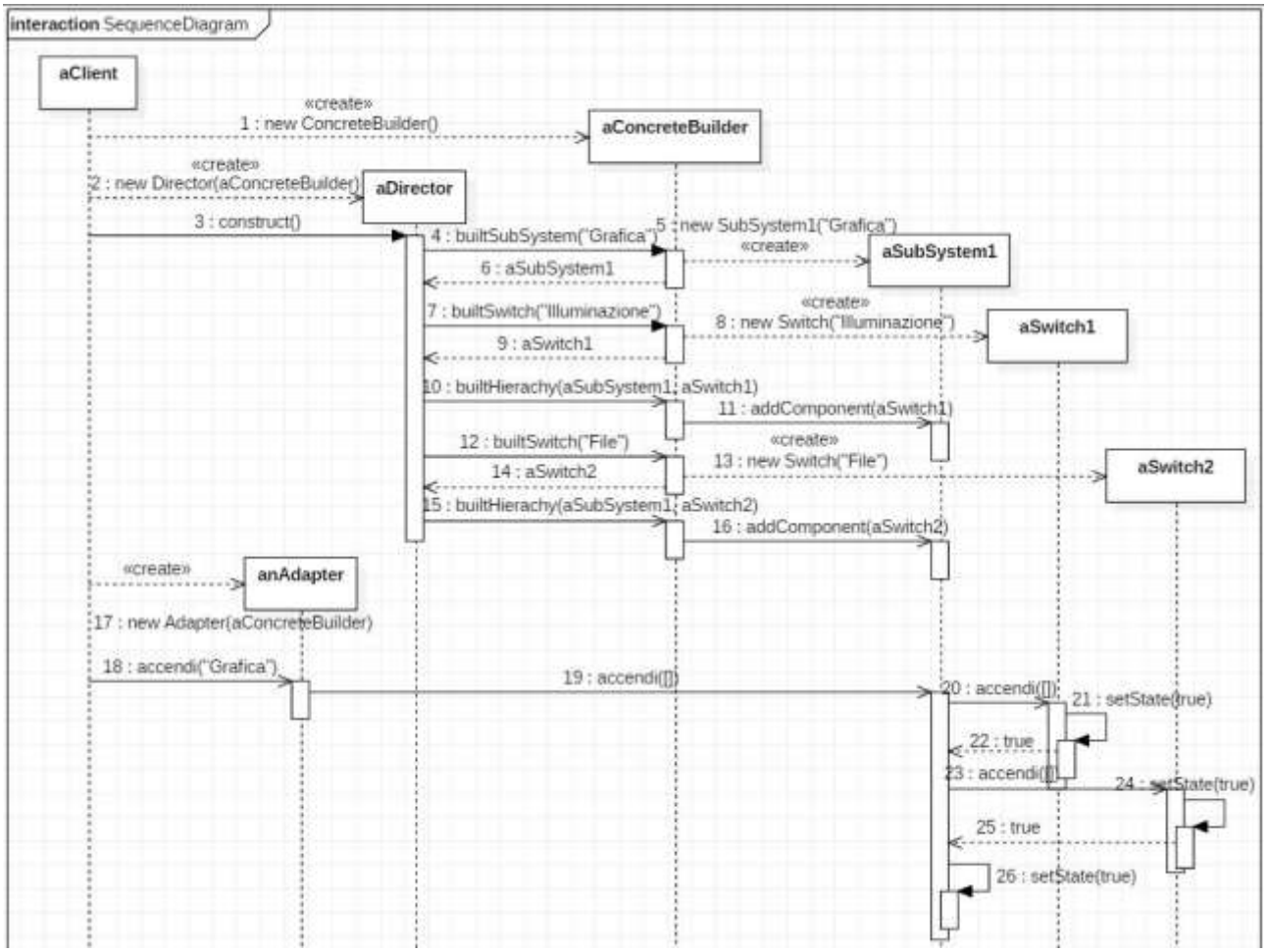
```java
package tester;
import composite.*;

public final class TestComposite {
    public static final void testComposite(HWSystem root) throws Exception {
        boolean response = true;

        HWSystem grafica = root.getChild(0);
        if(grafica.getName() == "Grafica" && grafica.getState() == false)
            response = response && true;
        else response = response && false;

        HWSystem illuminazione = grafica.getChild(0);
        if(illuminazione.getName() == "Illuminazione" && illuminazione.getState() == false)
            response = response && true;
        else response = response && false;

        HWSystem finestra = grafica.getChild(1);
        if(finestra.getName() == "Finestra" && finestra.getState() == false)
            response = response && true;
        else response = response && false;

        HWSystem app = finestra.getChild(0);
        if(app.getName() == "App" && app.getState() == false)
            response = response && true;
        else response = response && false;

        HWSystem cartella = finestra.getChild(1);
        if(cartella.getName() == "Cartella" && cartella.getState() == false)
            response = response && true;
        else response = response && false;

        HWSystem file = cartella.getChild(0);
        if(file.getName() == "File" && file.getState() == false)
            response = response && true;
        else response = response && false;

        if (response)
            System.out.println("L'albero delle componenti è stato creato correttamente.");
        else System.out.println("L'albero delle componenti NON è stato creato correttamente.");
    }
}
```

```java
package tester;

import adapter.*;
import composite.*;

public final class TestAccensione {

    public static final void testAccensione(VirtualCopy vc, HWSystem root) throws Exception{
        boolean response = true;
        HWSystem file = root.getChild(0).getChild(1).getChild(1).getChild(0);
        HWSystem cartella = root.getChild(0).getChild(1).getChild(1);
        HWSystem app = root.getChild(0).getChild(1).getChild(0);
        HWSystem finestra = root.getChild(0).getChild(1);
        HWSystem illuminazione = root.getChild(0).getChild(0);
        HWSystem grafica = root.getChild(0);

        vc.accendi("Cartella");
        if ((file.getState() && cartella.getState()) == true)
            response = response && true;
        else response = response && false;

        vc.accendi("Finestra");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState()) == true)
            response = response && true;
        else response = response && false;

        vc.accendi("Grafica");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState() && illuminazione.getState() && grafica.getState()) == true)
            response = response && true;
        else response = response && false;

        vc.accendi("PC");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState() && illuminazione.getState() && grafica.getState() && root.getState()) == true)
            response = response && true;
        else response = response && false;

        if (response)
            System.out.println("Tutte le componenti si accendono correttamente.");
        else System.out.println("Alcune componenti NON si accendono correttamente.");

    }
}
```

```java
package tester;

import adapter.*;
import composite.*;

public final class TestSpegnimento {

    public static final void testSpegnimento(VirtualCopy vc, HWSystem root) throws Exception{
        boolean response = true;
        HWSystem file = root.getChild(0).getChild(1).getChild(1).getChild(0);
        HWSystem cartella = root.getChild(0).getChild(1).getChild(1);
        HWSystem app = root.getChild(0).getChild(1).getChild(0);
        HWSystem finestra = root.getChild(0).getChild(1);
        HWSystem illuminazione = root.getChild(0).getChild(0);
        HWSystem grafica = root.getChild(0);

        vc.spegni("Cartella");
        if ((file.getState() && cartella.getState()) == false)
            response = response && true;
        else response = response && false;

        vc.spegni("Finestra");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState()) == false)
            response = response && true;
        else response = response && false;

        vc.spegni("Grafica");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState() && illuminazione.getState() && grafica.getState()) == false)
            response = response && true;
        else response = response && false;

        vc.spegni("PC");
        if ((file.getState() && cartella.getState() && app.getState() && finestra.getState() && illuminazione.getState() && grafica.getState() && root.getState()) == false)
            response = response && true;
        else response = response && false;

        if (response)
            System.out.println("Tutte le componenti si spengono correttamente.");
        else System.out.println("Alcune componenti NON si spengono correttamente.");

    }
}
```

3.2. Sequence diagram

Per evitare di appesantire troppo il Sequence diagram è stata limitata la sequenza di operazioni per la costruzione di sole tre componenti: un SubSystem e due Switch ad esso connessi. Ovviamente si tratta di un caso singolo, ma del tutto generale.



4. Possibili sviluppi

Così come dall'idea iniziale di utilizzare solo i pattern Composite e Adapter si è reso necessario l'ausilio del Builder, nel corso d'opera sono nate tante altre possibili idee su come estendere il sistema costruito mediante altri pattern. Di seguito viene dato un breve resoconto.

4.1. Proxy

Un sistema hardware spesso e volentieri potrebbe richiedere delle autorizzazioni per procedere nella modifica di una qualche componente. Poter aggiungere un criterio di sicurezza sarebbe importante anche per rendere il sistema più prossimo alla realtà e rendere le sue interazioni più dinamiche.

Per esempio, si consideri il caso in cui spegnere un interruttore File richiedesse una password di autorizzazione. Qualora venisse invocato il metodo spegni("Finestra"), questa spengerebbe tutte le componenti e arrivata a File si fermerebbe in attesa della parola chiave. Nel caso venisse inserita correttamente il procedimento continuerebbe col suo corso normale; mentre in caso il Client non avesse tale autorizzazione verrebbe negata l'operazione, impedendo anche lo spegnimento di Finestra, poiché una componente viene spenta solo se lo sono tutte le sue sotto-componenti.

Un sistema di questo genere si potrebbe sviluppare mediante un SecurityProxy che implementa l'interfaccia (o estende la classe astratta) di HWSystem e detiene un riferimento ad uno Switch. Ogni interruttore avrebbe una variabile che indica se è protetto o meno da requisiti di amministratore, e a seconda di tale valore, il Proxy agirebbe o meno come intermediario.

```
public class SecurityProxy implements Component {
    private Interruttore myInt;

    public SecurityProxy(boolean admin) {
        myInt = new Interruttore(admin);
    }

    @Override
    public boolean getState() {
        return myInt.getState();
    }
    @Override
    public void setState(boolean s) {
        myInt.setState(s);
    }
    @Override
    public boolean spegni(ArrayList <Integer> order) throws Exception{
        if(myInt.getAdmin()) {
            if (this.security())
                return false;
        }
        return myInt.spegni(order);
    }
    @Override
    public boolean accendi(ArrayList <Integer> order) throws Exception{
        if(myInt.getAdmin()) {
            if (this.security())
                return false;
        }
        return myInt.accendi(order);
    }

    private boolean security() {
        System.out.println("L'interruttore "+ this.myInt.toString() +" è protetto.");
        Scanner input = new Scanner(System.in);
        String response = input.nextLine();
        input.close();
        if (!(response.equals("marcopaglio")))
            return false;
        else return true;
    }
}
```

4.2. Observer

Altro scenario interessante riguarda la possibilità di modificare (aggiungere, rimuovere, spostare) le componenti del sistema hardware. Sarebbe un'operazione molto onerosa, in quanto sarebbe necessario notificare sia le sotto-componenti sia il Translator dei cambiamenti avvenuti. Di fatto deve rimanere sempre coerente il cammino per arrivare a ciascuna componente.

A tal proposito sarebbe necessario implementare la classe che si occupa di modificare il sistema come Subject e il Translator come Observer. Quest'ultimo, una volta agganciato al Subject, sarebbe in grado di ricevere aggiornamenti. La modalità migliore sarebbe quella push, in quanto le modifiche sono conosciute dalla classe modificatrice, per cui le renderebbe disponibili anche all'Observer, che sicuramente utilizzerà il dato per effettuare l'aggiornamento. Talvolta però renderebbe complesso l'utilizzo di una seconda tipologia di osservatori.

4.3. Mediator

Un problema aggiuntivo è la consistenza dello stato: è possibile che il Client faccia uso delle operazioni della VirtualCopy per una componente che è stata modificata e per la quale deve avvenire l'aggiornamento delle informazioni relative al cammino. Pertanto la richiesta deve essere messa in pausa da quando una qualche componente all'interno del sistema è stata modificata fino all'aggiornamento delle informazioni. La coordinazione è responsabilità del Mediator pattern.