Design Patterns

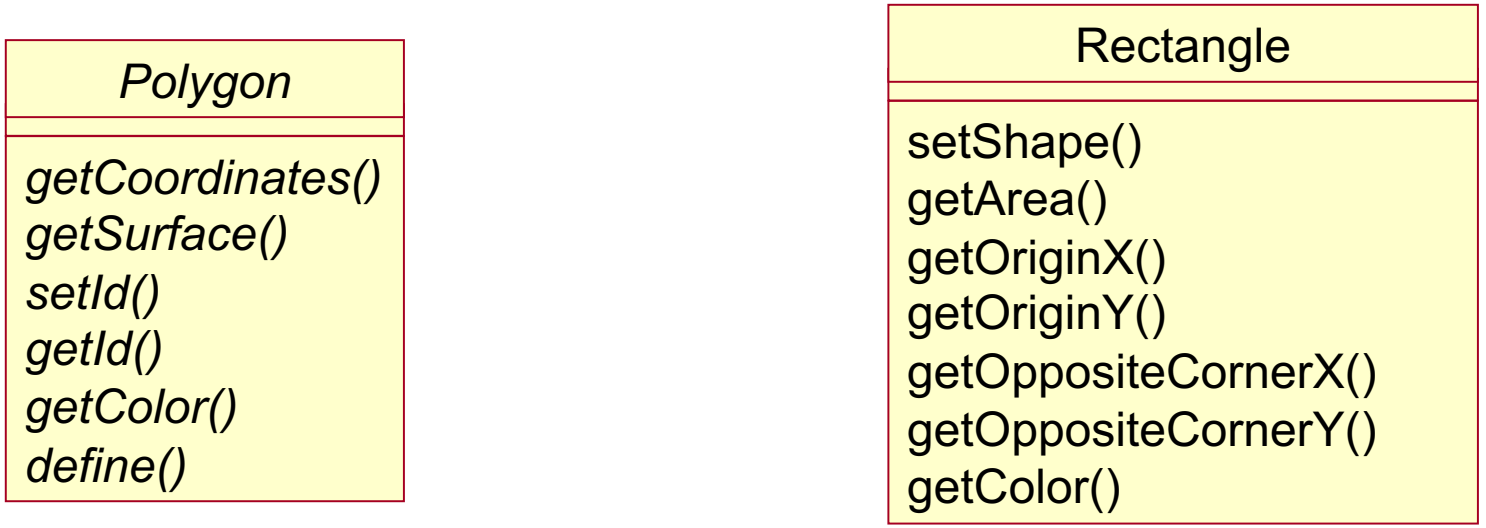
Esempi di applicazione

Adapter: esempio

Scenario: consideriamo un'applicazione per lavorare con oggetti geometrici

Questi oggetti saranno gestiti dall'applicazione tramite un'interfaccia particolare (*Polygon*), che offre un insieme di metodi che gli oggetti grafici devono implementare.
Si ha a disposizione una classe (*Rectangle*) che si vorrebbe riutilizzare, ma tale classe ha un'interfaccia diversa che non si vuole modificare.

Adapter: esempio (2)



Adapter: esempio (3)

```
public class Rectangle {
    private float x0, y0;
    private float height, width;
    private String color;

    public void setShape(float x, float y, float a, float l, String c) {
        x0 = x;
        y0 = y;
        height = a;
        width = l;
        color = c;
    }

    public float getArea() {
        return x0 * y0;
    }

    public float getOriginX() {
        return x0;
    }

    public float getOriginY() {
        return y0;
    }

    public float getOppositeCornerX() {
        return x0 + height;
    }

    public float getOppositeCornerY() {
        return y0 + width;
    }

    public String getColor() {
        return color;
    }
}
```

Adapter: esempio (4)

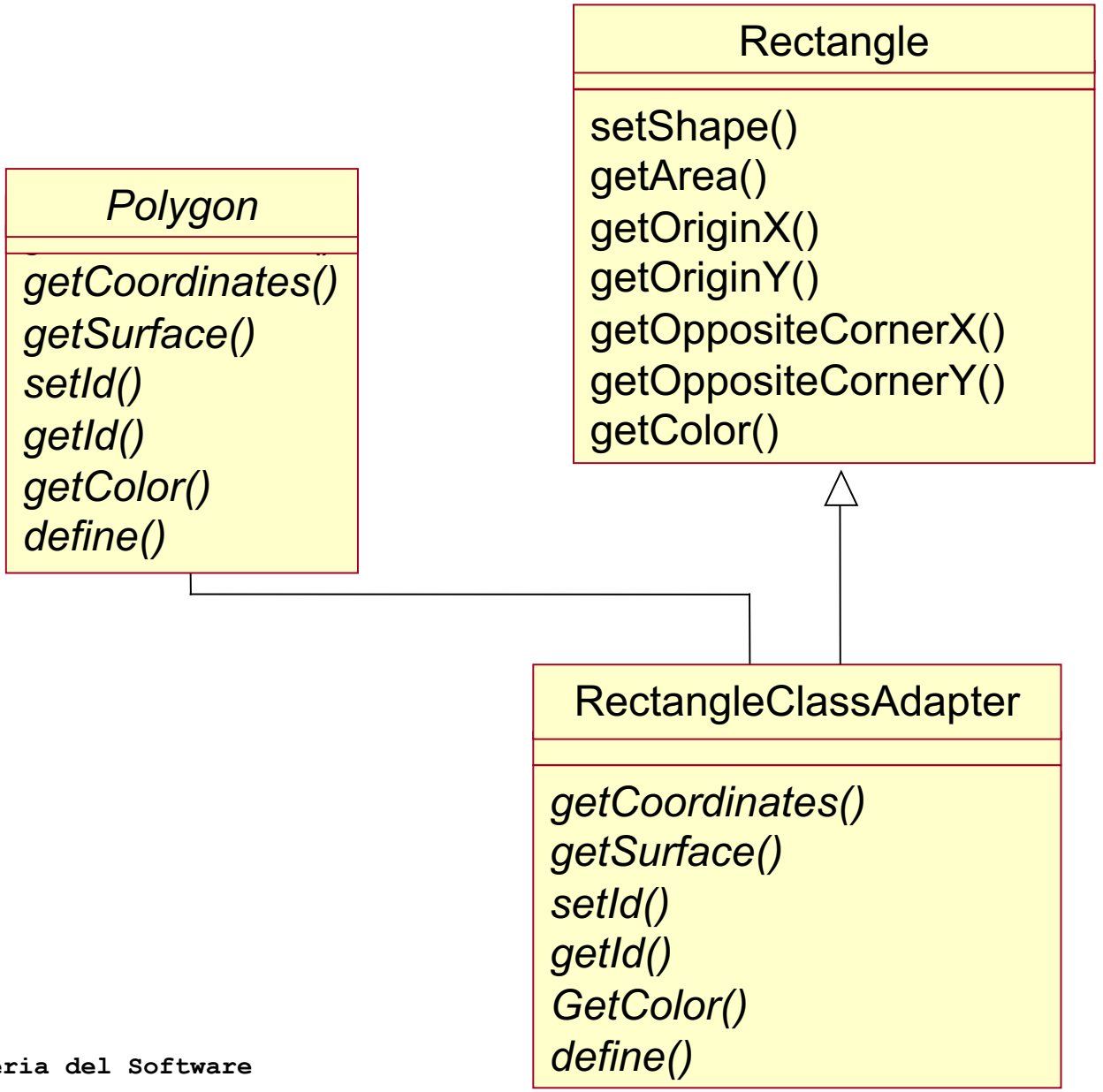
```
public interface Polygon {
    public void define(float x0, float y0, float x1, float y1, String color);

    public float[] getCoordinates();

    public float getSurface();

    public void setId(String id);

    public String getId();

    public String getColor();
}
```

Adapter: esempio (5)

Caso 1 - Class Adapter

La costruzione del Class Adapter per il Rectangle è basato sulla sua estensione. Per questo obbiettivo viene creata la classe *RectangleClassAdapter* che estende *Rectangle* e implementa l'interfaccia *Polygon*

Adapter: esempio (6)



Adapter: esempio (7)

```
public class RectangleClassAdapter extends Rectangle implements Polygon {
    private String name = "NO NAME";

    public void define(float x0, float y0, float x1, float y1, String color) {
        float a = x1 - x0;
        float l = y1 - y0;
        setShape(x0, y0, a, l, color);
    }

    public float getSurface() {
        return getArea();
    }

    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = getOriginX();
        aux[1] = getOriginY();
        aux[2] = getOppositeCornerX();
        aux[3] = getOppositeCornerY();
        return aux;
    }

    public void setId(String id) {
        name = id;
    }

    public String getId() {
        return name;
    }
}
```

Ereditato da Rectangle

Adapter: esempio (8)

Ed ecco un possibile client...

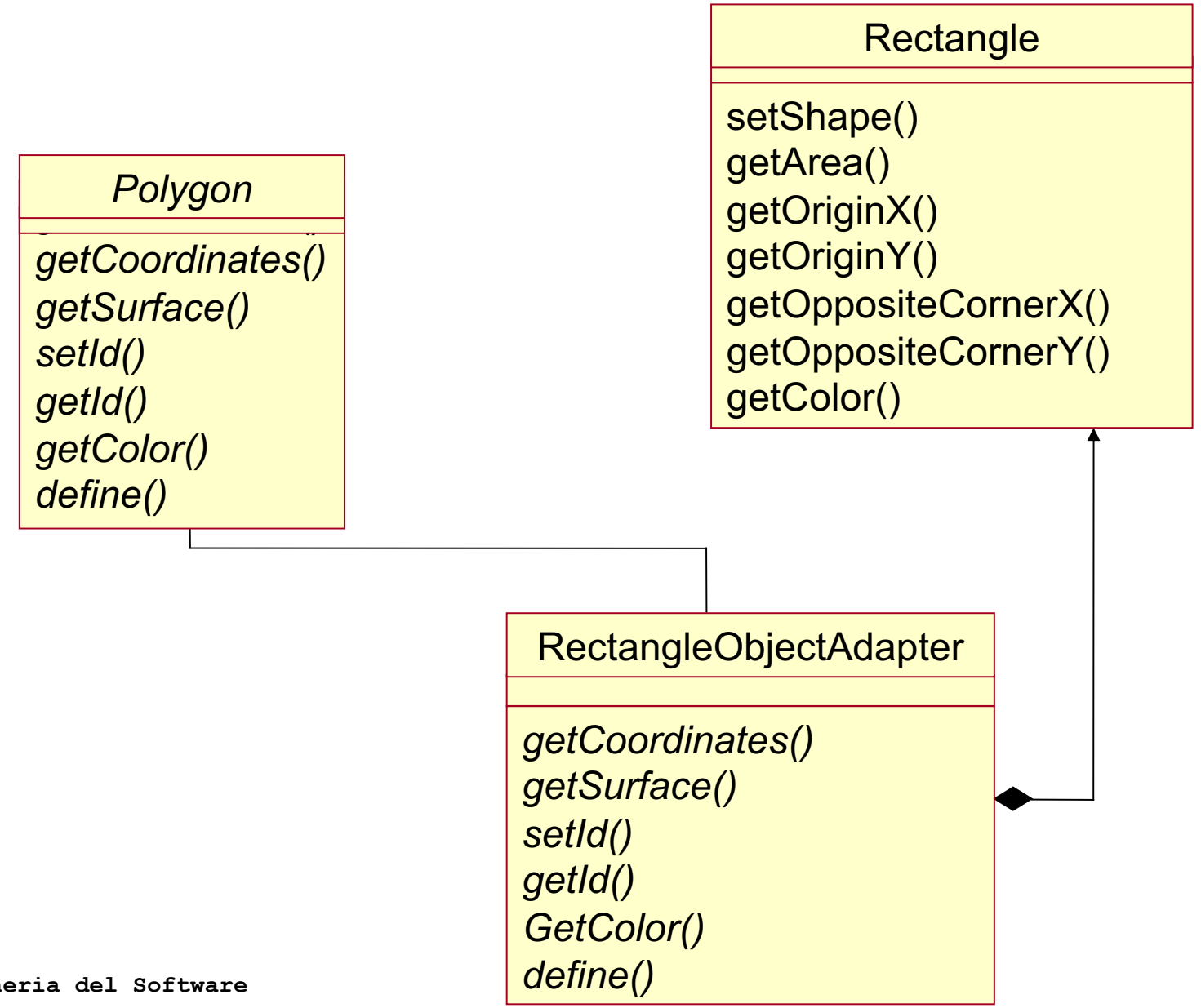
```
public class ClassAdapterExample {
    public static void main(String[] arg) {
        Polygon block = new RectangleClassAdapter();
        block.setId("Demo");
        block.define(3, 4, 10, 20, "RED");
        System.out.println("The area of " + block.getId() + " is "
                + block.getSurface() + ", and it's " + block.getColor());
    }
}
```

Adapter: esempio (9)

Caso 2 – Object Adapter

La costruzione dell'Object Adapter per il Rectangle, si basa nella creazione di una nuova classe (*RectangleObjectAdapter*) che avrà al suo interno un'oggetto della classe *Rectangle*, e che implementa l'interfaccia *Polygon*

Adapter: esempio (9)



Adapter: esempio (10)

```
public class RectangleObjectAdapter implements Polygon {
    Rectangle adaptee;
    private String name = "NO NAME";

    public RectangleObjectAdapter() {
        adaptee = new Rectangle();
    }

    public void define(float x0, float y0, float x1, float y1, String col) {
        float a = x1 - x0;
        float l = y1 - y0;
        adaptee.setShape(x0, y0, a, l, col);
    }

    public float getSurface() {
        return adaptee.getArea();
    }

    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = adaptee.getOriginX();
        aux[1] = adaptee.getOriginY();
        aux[2] = adaptee.getOppositeCornerX();
        aux[3] = adaptee.getOppositeCornerY();
        return aux;
    }

    public void setId(String id) {
        name = id;
    }

    public String getId() {
        return name;
    }

    public String getColor() {
        return adaptee.getColor();
    }
}
```

Istanzia un oggetto Rectangle "adaptee" e invoca i suoi metodi

Adapter: esempio (11)

Ed ecco un possibile client...

```
public class ObjectAdapterExample {
    public static void main(String[] arg) {
        Polygon block = new RectangleObjectAdapter();
        block.setId("Demo");
        block.define(3, 4, 10, 20, "RED");
        System.out.println("The area of " + block.getId() + " is "
                + block.getSurface() + ", and it's " + block.getColor());
    }
}
```

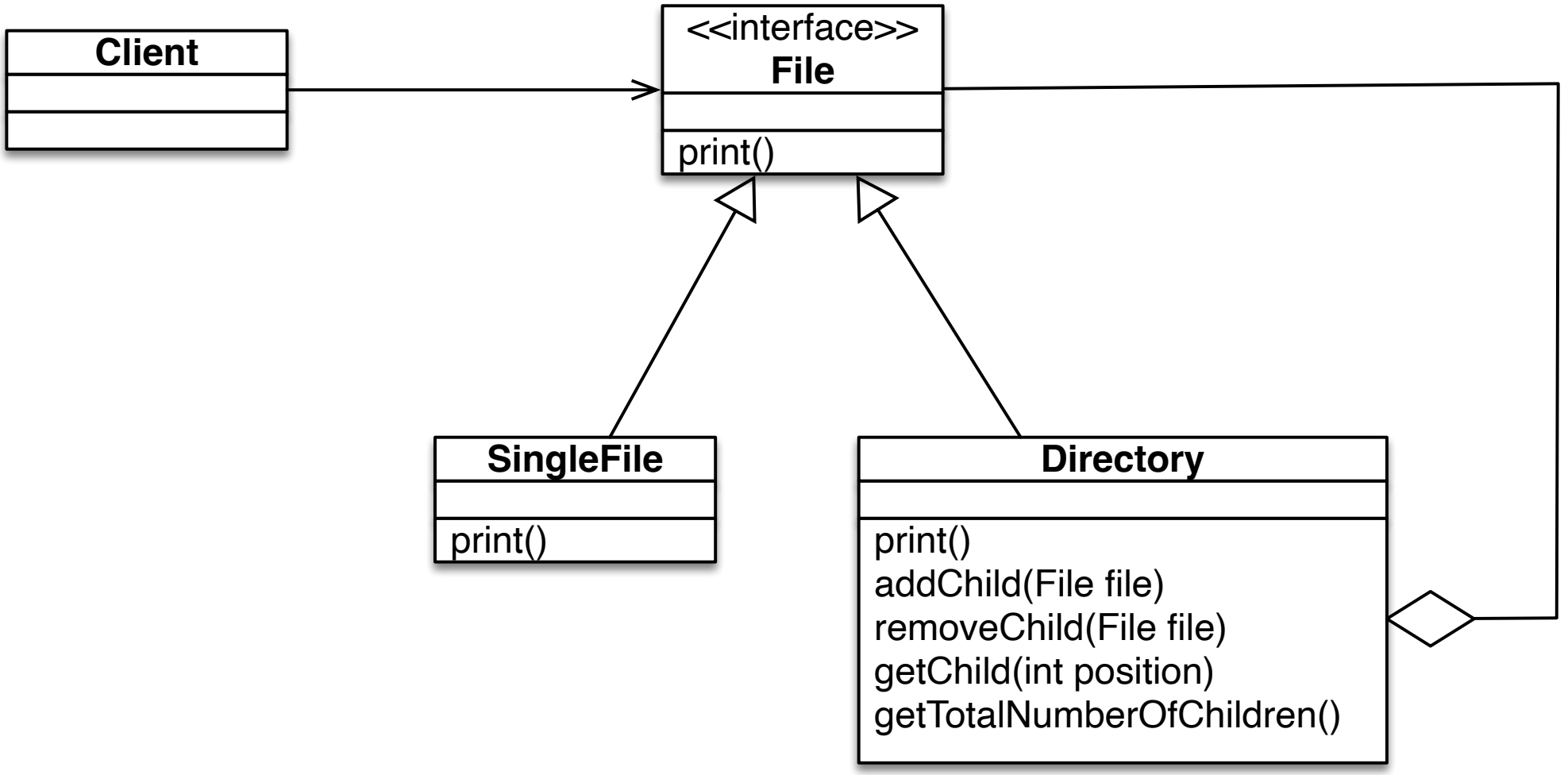
Adapter: esempio (12)

- Partecipanti
  - Target Interface: interfaccia Polygon.
    - Specifica l'interfaccia che il Client utilizza.
  - Adaptee: classe Rectangle.
    - Implementa una interfaccia che deve essere adattata.
  - Adapter: classi RectangleClassAdapter e RectangleObjectAdapter.
    - Adatta l'interfaccia dell'Adaptee verso la Target Interface.

Composite: esempio

Scenario: pensiamo al FileSystem che presenta una struttura ad albero e che può essere composto da elementi semplici (*files*) e da contenitori (*cartelle*). L'obiettivo è quello di permettere al Client di accedere e navigare il File System senza conoscere la natura degli elementi che lo compongono in modo da trattare tutti gli elementi nello stesso modo.
Per fare questo il Client userà la stessa interfaccia per l'accesso mentre l'implementazione nasconderà la gestione degli oggetti a seconda della loro reale natura.

Composite: esempio (2)



Composite: esempio (3)

- Creiamo l'interfaccia di interrogazione per l'accesso a file e directory.

```
public interface File {
    void print();
}
```

Composite: esempio (4)

Implementiamo la classe che gestisce i File.

```
public class SingleFile implements File {
    private final String fileName;

    public SingleFile(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void print() {
        System.out.println(fileName);
    }
}
```

Composite: esempio (5)

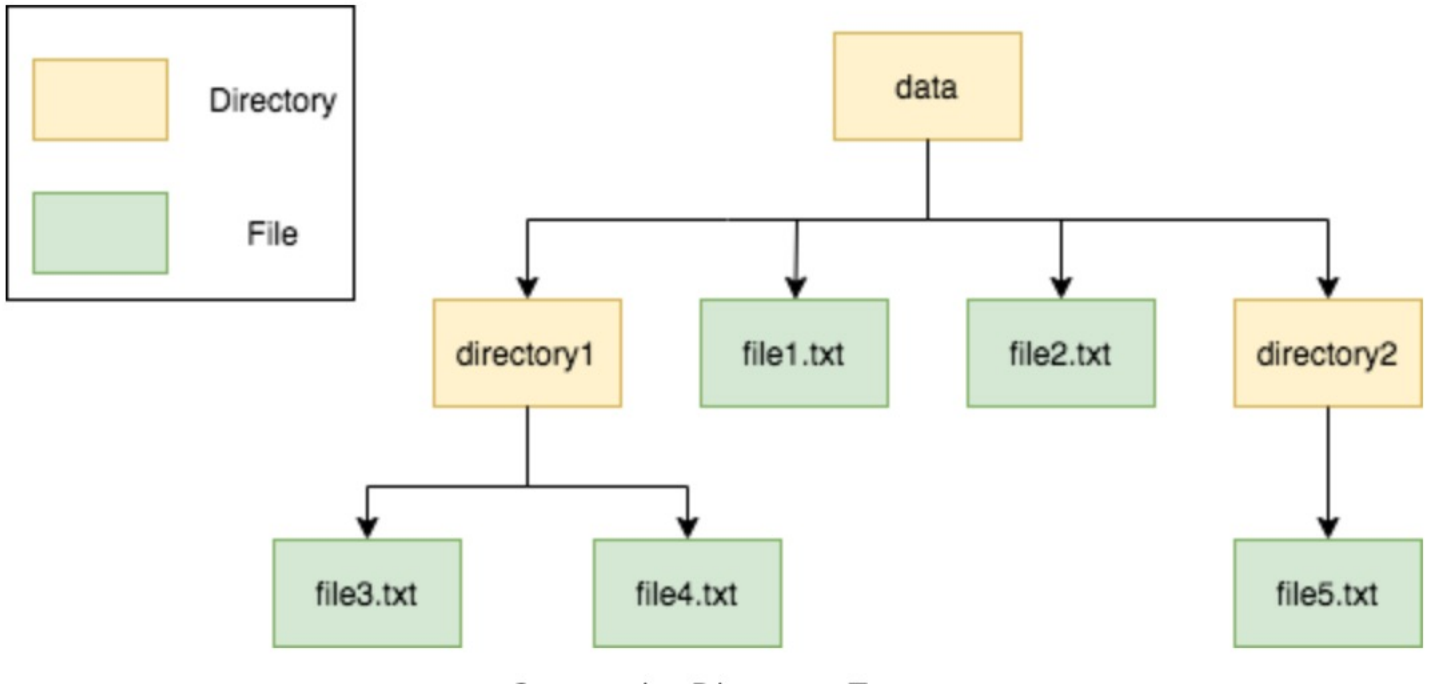
```java
public class Directory implements File {
    private final String directoryName;
    private final List<File> children;

    public Directory(String directoryName, List<File> children) {
        this.directoryName = directoryName;
        this.children = new ArrayList<>(children);
    }

    public void addChild(File file) {
        this.children.add(file);
    }

    public void removeChild(File file) {
        this.children.add(file);
    }

    public File getChild(int position) {
        if (position < 0 || position >= children.size()) {
            throw new RuntimeException("Invalid position " + position);
        }
        return children.get(position);
    }

    private int getTotalNumberOfChildren() {
        return children.size();
    }

    @Override
    public void print() {
        System.out.println("Printing the contents of the directory - " +
directoryName);
        children.forEach(File::print);
        System.out.println("Done printing the contents of the directory - " +
directoryName);
    }
}
```

Composite: esempio (6)

- Il metodo print viene invocato su tutti gli oggetti dell'albero, siano essi files o cartelle. Il comportamento sarà diverso nei 2 casi.
- Nella classe SingleFile si limita a stampare il nome del file.
- Nella classe Directory, oltre a stampare il nome della cartella, presenta un loop utilizzato per invocare file/directory per consentire la ricorsione su tutta l'alberatura del file system.

Composite: esempio (7)

Consideriamo il seguente file system



Composite: esempio (8)

La classe Client crea l'alberatura del File System e poi visualizza il suo contenuto.

```
File file1 = new SingleFile("file1.txt");
File file2 = new SingleFile("file2.txt");
File file3 = new SingleFile("file3.txt");
File file4 = new SingleFile("file4.txt");
File file5 = new SingleFile("file5.txt");


Directory directory1 = new Directory("directory1", List.of(file3, file4));
Directory directory2 = new Directory("directory2", List.of(file5));
Directory data = new Directory("data", List.of(directory1, file1, file2,
directory2));

data.print();
```

Composite: esempio (9)

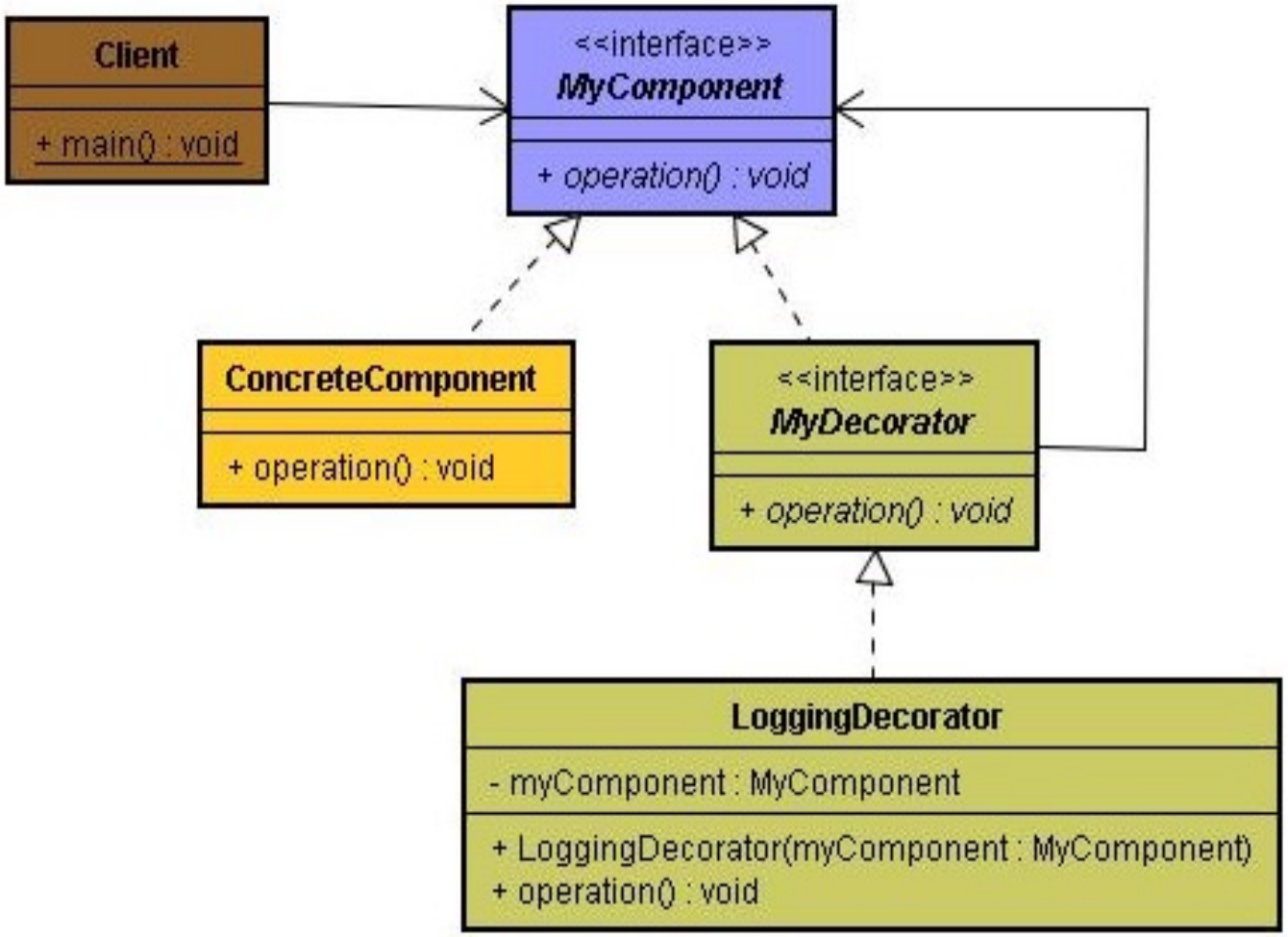
Quale sarà l'output?

```
Printing the contents of the directory - data
Printing the contents of the directory - directory1
file3.txt
file4.txt
Done printing the contents of the directory - directory1
file1.txt
file2.txt
Printing the contents of the directory - directory2
file5.txt
Done printing the contents of the directory - directory2
Done printing the contents of the directory - data
```

Decorator: esempio

Scenario: abbiamo l'esigenza di monitorare l'invocazione di un metodo ma non abbiamo la possibilità di modificare il codice. Utilizziamo il pattern Decorator per esigenze di debug pertanto "wrappiamo" un metodo con delle semplici istruzioni print-screen.

Decorator: esempio (2)



Decorator: esempio (3)

```
package patterns.decorator;
public interface MyComponent {
    public void operation();
}
```

Creiamo l'interfaccia Component
che dichiara il metodo interessato.

Decorator: esempio (3)

```
package patterns.decorator;
public class ConcreteComponent implements MyComponent {
    public void operation(){
        System.out.println("Hello World");
    }
}
```

Implementiamo il metodo dichiarato nell'interfaccia MyComponent creando la classe ConcreteComponent.

Decorator: esempio (4)

```
package patterns.decorator;
interface MyDecorator extends MyComponent {
}
```

Definiamo l'interfaccia MyDecorator che si occupa di ereditare il metodo interessato da MyComponent e di interporsi con le classi di decorazione concrete.

Decorator: esempio (5)

```
public class LoggingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public LoggingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }

    public void operation() {
        System.out.println("First Logging");
        myComponent.operation();
        System.out.println("Last Logging");
    }
}
```

Creiamo la classe LoggingDecorator che implementa l'interfaccia MyDecorator ed aggiunge le informazioni di debug prima e dopo l'esecuzione del metodo interessato.

Decorator: esempio (6)

```
class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new ConcreteComponent());
        myComponent.operation();
    }
}
```

La classe Client invoca la classe concreta LoggingDecorator passando al costruttore il componente concreto, successivamente invoca il metodo operation().

Decorator: esempio (7)

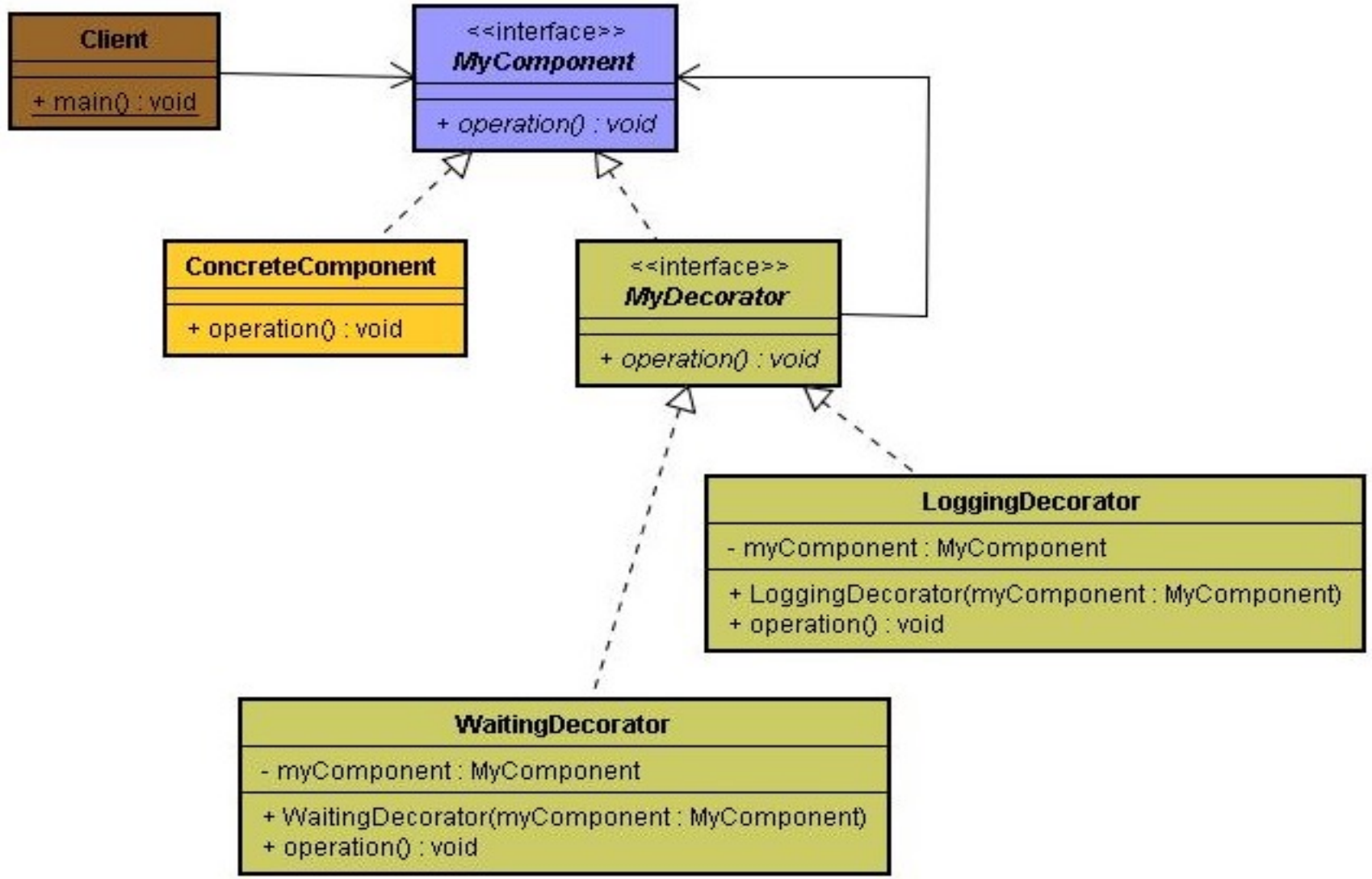
Quale sarà l'output?

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Hello World
Last Logging
```

Decorator: esempio (8)

- Partendo dall'esempio, possiamo creare una moltitudine di classi concrete Decorator che aggiungono nuove funzionalità.
- Per esempio possiamo creare una classe WaitingDecorator che preveda una pausa durante l'esecuzione.
- Vediamo come diventa il Class Diagram a seguito dell'inseriemento di questa nuova classe.

Decorator: esempio (9)



Decorator: esempio (10)

```
public class WaitingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public WaitingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }

    public void operation() {
        try {
            System.out.println("Waiting...");
            Thread.sleep( 1000 );
        }
        catch (Exception e) {}

        myComponent.operation();
    }

}
```

Decorator: esempio (11)

Il Client invoca in modo annidato i Decorator tramite il loro costruttore, come segue:

```
class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new WaitingDecorator(new
ConcreteComponent()));
        myComponent.operation();
    }
}
```

Decorator: esempio (12)

Quale sarà l'output?

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Waiting...
Hello World
Last Logging
```

Factory Method: esempio

Scenario: supponiamo di avere una applicazione che legge dei dati da un file di testo contenente le informazioni relative a delle rilevazioni di letture di contatori per acqua e gas.

Nel nostro codice abbiamo una classe dedicata a questo scopo, che legge i vari formati dei file, la classe *AcquisizioneLetture*:

Factory Method: esempio (2)

```
public class AcquisizioneLetture {
   public AcquisizioneLetture() {
   }

   public void parseFile(String fileName, String dataType) {
      ArrayList lettureArray = new ArrayList();
      FileLettureReader fileLettureReader; //questa è una interfaccia
      Lettura lettura;              //questa è una interfaccia

      if (dataType.equals("gas")) {
        fileLettureReader = new GasLettureReader(fileName);
      }
      if (dataType.equals("H2O")) {
        fileLettureReader = new H2OLettureReader(fileName);
      }

      while (fileLettureReader.hasNextLettura()) {
        lettura = fileLettureReader.getNextLettura();

        if (lettura.verifica()) {
           lettura.calcolaconsumo();
           lettura.registra();
        } else {
           lettura.scarta();
        }
      }

   }
}
```

classi specializzate nella lettura di file in formato testo che implementano l'interfaccia *FileLettureReader*

Ritornano un oggetto di tipo Lettura

Factory Method: esempio (3)

- Succede però che il nostro cliente inizia a vendere anche energia elettrica e deve di conseguenza acquisire anche i file con le relative letture.
- E' quindi necessario gestire un tipo aggiuntivo di file, e modifichiamo la nostra classe:

Factory Method: esempio (4)

```
public class AcquisizioneLetture {
   public AcquisizioneLetture() {
   }

   public void parseFile(String fileName, String dataType) {
      ArrayList lettureArray = new ArrayList();
      FileLettureReader fileLettureReader; //questa è una interfaccia
      Lettura lettura; //questa è una interfaccia

      if (dataType.equals('gas')) {
        fileLettureReader = new GasLettureReader(fileName);
      }
      if (dataType.equals('H2O')) {
        fileLettureReader = new H2OLettureReader(fileName);
      }

      if(dataType.equals('EE')){
         fileLettureReader = new EELettureReader(fileName);
      }

      while (fileLettureReader.hasNextLettura()) {
        lettura = fileLettureReader.getNextLettura();

        if (lettura.verifica()) {
           lettura.calcolaconsumo();
           lettura.registra();
        } else {
           lettura.scarta();
        }
      }

   }
}
```

E' stato necessario aprire la classe e introdurre la modifica

Factory Method: esempio (5)

- Operazione non pratica, soprattutto se prevediamo di ripeterla in futuro.
- Dovremmo separare il codice soggetto a modifiche da quello sempre uguale, come fare?
- Ad esempio incapsulando la creazione di FileLettureReader all'interno di una nuova classe FileReaderFactory:

Factory Method: esempio (6)

```
public class ReaderFactory {
   private FileLettureReader fileLettureReader;
   public ReaderFactory() {
   }

   public FileLettureReader getFileLettureReader(String fileName, String dataType) {
      if (dataType.equals("gas")) {
        fileLettureReader = new GasLettureReader(fileName);
      }
      if (dataType.equals("H2O")) {
        fileLettureReader = new H2OLettureReader(fileName);
      }
      if (dataType.equals("EE")) {
        fileLettureReader = new EELettureReader(fileName);
      }
      return fileLettureReader;
   }
}
```

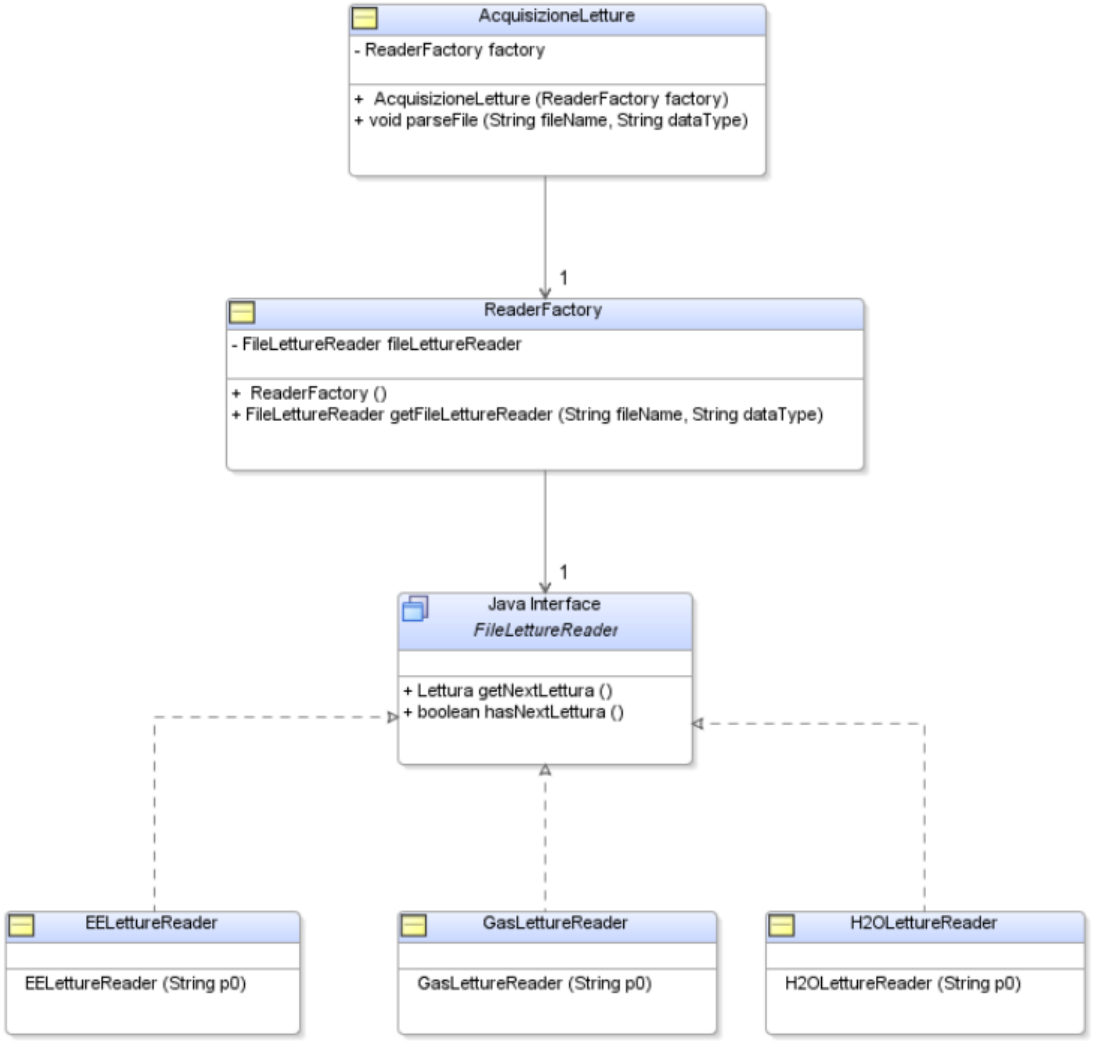
Factory Method: esempio (7)

```
public class AcquisizioneLetture {
   private ReaderFactory factory;
   public AcquisizioneLetture(ReaderFactory factory) {
   this.factory = factory;
   }

   public void parseFile(String fileName, String dataType) {
      ArrayList lettureArray = new ArrayList();
      FileLettureReader fileLettureReader; //questa è una interfaccia
      Lettura lettura; //questa è una interfaccia
      // meglio, no?
      fileLettureReader = factory.getFileLettureReader(fileName,dataType);
      //
      while (fileLettureReader.hasNextLettura()) {
        lettura = fileLettureReader.getNextLettura();

        if (lettura.verifica()) {
           lettura.calcolaconsumo();
           lettura.registra();
        } else {
           lettura.scarta();
        }
      }
   }
}
```

Factory Method: esempio (8)

- Abbiamo ottenuto:
  - La chiusura di AcquisizioneLetture alle modifiche
  - La possibilità di riutilizzare ReaderFactory anche altrove, isolando in essa le modifiche

Factory Method: esempio (9)



Factory Method: esempio (10)

- Poi accade che acquisiamo due importanti clienti, che vendono acqua e gas ed utilizzano un proprio formato XML di interscambio dati
- Adeguiamo il nostro codice alle nuove esigenze, scrivendo due nuove factory class ad hoc per loro,Cliente1ReaderFactory e Cliente2ReaderFactory che derivano dalla nostra ReaderFactory

Factory Method: esempio (11)

```
public class Cliente1ReaderFactory extends ReaderFactory{
   public Cliente1ReaderFactory() {
   }

   public FileLettureReader getFileLettureReader(String fileName, String dataType){
      if (dataType.equals("gas")) {
        fileLettureReader = new Cliente1GasLettureReader(fileName);
      }
      if (dataType.equals("H2O")) {
        fileLettureReader = new Cliente1H2OLettureReader(fileName);
      }
      return fileLettureReader;
   }
}

public class Cliente2ReaderFactory extends ReaderFactory {
   public Cliente2ReaderFactory() {
   }

   public FileLettureReader getFileLettureReader(String fileName,
                                String dataType) {
      if (dataType.equals("gas")) {
        fileLettureReader = new Cliente2GasLettureReader(fileName);
      }
      if (dataType.equals("H2O")) {
        fileLettureReader = new Cliente2H2OLettureReader(fileName);
      }
      return fileLettureReader;
   }
}
```

Factory Method: esempio (12)

- Corretto, ma notiamo due cose:
  - AcquisizioneLetture ha sempre bisogno che gli venga passato un factory per funzionare, tanto che viene passato nel costruttore.
  - Le operazioni fatte sulla lettura, come abbiamo visto all'inizio, sono sempre quelle.
- Sarebbe utile quindi portare il factory direttamente dentro AcquisizioneLetture per rendere la classe autosufficiente, ma senza perdere la flessibilità ottenuta fino ad ora.

Factory Method: esempio (13)

- Torniamo alla prima versione di AcquisizioneLetture, ma questa volta incapsuliamo la creazione delle classi FileLettureReader all'interno di un metodo astratto e rendiamo quindi astratta tutta la classe.
- Da questa deriviamo le varie versioni per i vari clienti, implementando in ognuna di esse il metodo che incapsula la creazione delle classi FileLettureReader.
- Ora abbiamo una AcquisizioneLetture per ogni cliente, ognuna contenente la propria logica di lettura dei file.

Factory Method: esempio (14)

```
public abstract class AcquisizioneLetture {
   public AcquisizioneLetture() {

   }

   public void parseFile(String fileName, String dataType) {
      ArrayList lettureArray = new ArrayList();
      FileLettureReader fileLettureReader; //questa è una interfaccia
      Lettura lettura; //questa è una interfaccia

      fileLettureReader =getFileLettureReader(fileName,dataType);
      //
      while (fileLettureReader.hasNextLettura()) {
        lettura = fileLettureReader.getNextLettura();

        if (lettura.verifica()) {
           lettura.calcolaconsumo();
           lettura.registra();
        } else {
           lettura.scarta();
        }
      }
   }
   protected abstract FileLettureReader getFileLettureReader(String fileName, String fileType);
}
```

La classe base non conosce il *FileLettureReader* su cui itera e da cui ricava le letture, perché questo dipende dalle classi derivate.
Abbiamo conservato disaccoppiamento e flessibilità.

Factory Method: esempio (15)

```
public class AcquisizioneLettureConcreta extends AcquisizioneLetture {
   public AcquisizioneLettureConcreta() {
   }

   protected FileLettureReader getFileLettureReader(String fileName,
                                    String fileType) {
      FileLettureReader fileLettureReader;
      if (fileType.equals("gas")) {
        fileLettureReader = new GasLettureReader(fileName);
      }
      if (fileType.equals("H2O")) {
        fileLettureReader = new H2OLettureReader(fileName);
      }
      if (fileType.equals("EE")) {
        fileLettureReader = new EELettureReader(fileName);
      }
      return fileLettureReader;
   }
}
```

Factory Method: esempio (16)

```
public class AcquisizioneLettureCliente1 extends AcquisizioneLetture {
   public AcquisizioneLettureCliente1() {
   }

   protected FileLettureReader getFileLettureReader(String fileName,
                                    String fileType) {
      FileLettureReader fileLettureReader;
      if (fileType.equals("gas")) {
        fileLettureReader = new Cliente1GasLettureReader(fileName);
      }
      if (fileType.equals("H2O")) {
        fileLettureReader = new Cliente1H2OLettureReader(fileName);
      }
      return fileLettureReader;

   }
}
```

Factory Method: esempio (17)

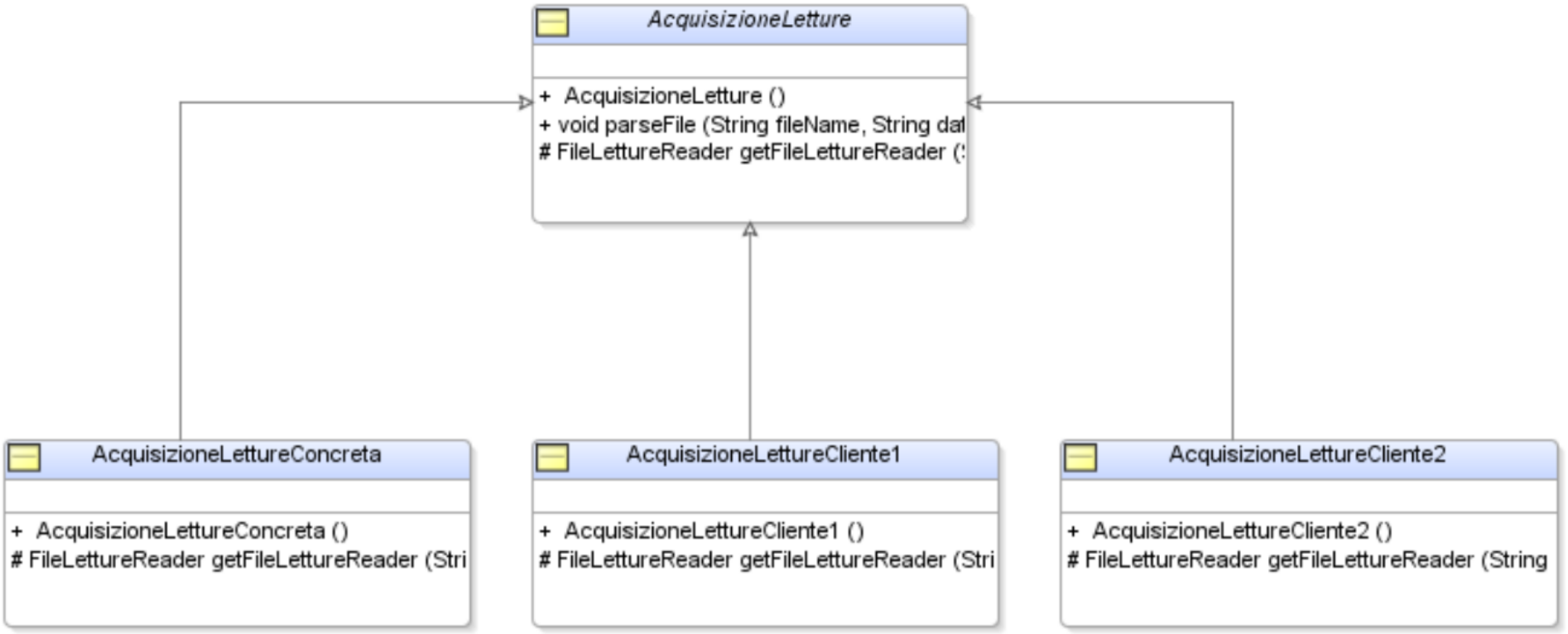
```
public class AcquisizioneLettureCliente2 extends AcquisizioneLetture {
    public AcquisizioneLettureCliente2() {
    }

    protected FileLettureReader getFileLettureReader(String fileName,
                                        String fileType) {
        FileLettureReader fileLettureReader;
        if (fileType.equals("gas")) {
          fileLettureReader = new Cliente2GasLettureReader(fileName);
        }
        if (fileType.equals("H2O")) {
          fileLettureReader = new Cliente2H2OLettureReader(fileName);
        }
        return fileLettureReader;
    }
}

}
```

Factory Method: esempio (17)

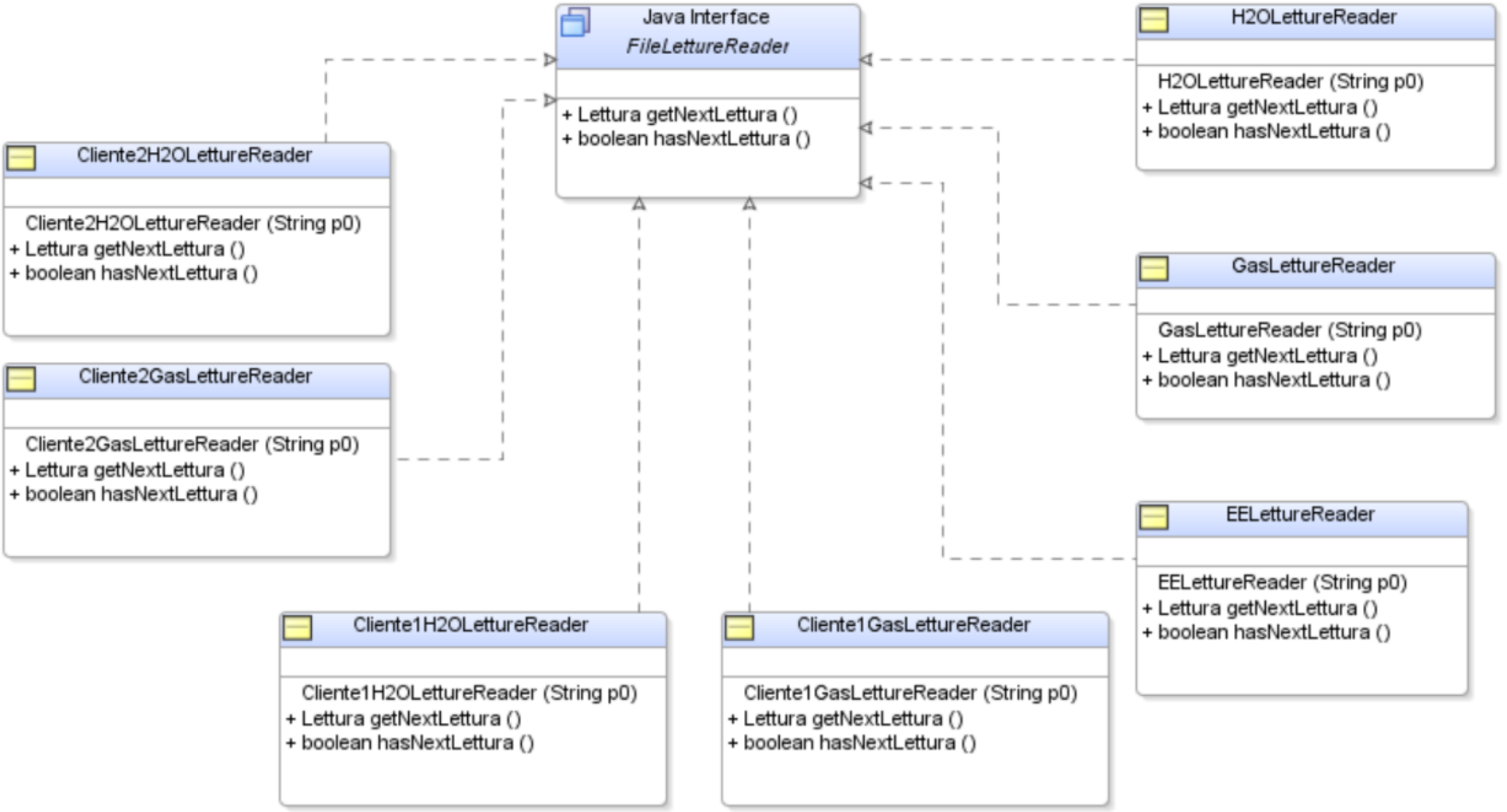
Ricordando il diagramma UML del Factory Method, abbiamo costruito:

Creator	*AcquisizioneLetture*
Product	*FileLettureReader*
ConcreteCreator	*AcquisizioneLettureConcreta* *AcquisizioneLettureCliente1* *AcquisizioneLettureCliente2*
ConcreteProduct	*GasLettureReader* *H2OLettureReader* *EELettureReader* *Cliente1GasLettureReader* *Cliente1H2OLettureReader* *Cliente2GasLettureReader* *Cliente2H2OLettureReader*

Factory Method: esempio (18)



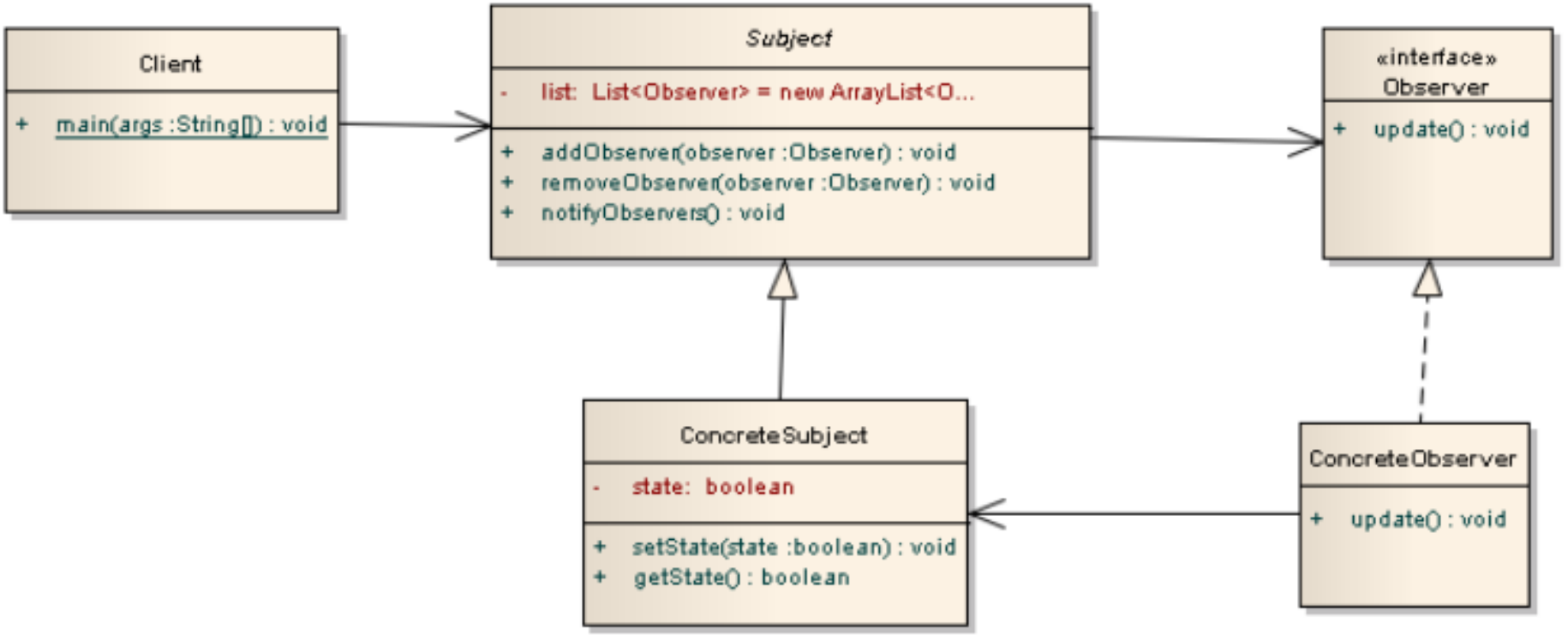
Factory Method: esempio (19)



Observer: esempio

Scenario: due osservatori sono interessati al cambio di stato di un soggetto osservato. Successivamente uno di essi cancella la sottoscrizione e non riceve più notifiche. In questo caso dobbiamo creare il *Subject* e l'*Observer* e le classi concrete *ConcreteSubject* e *ConcreteObserver*.

Observer: esempio (2)



Observer: esempio (3)

```
public interface Observer {

    public void update();

}
```

L'interfaccia dell'Observer definisce il metodo che dovrà essere implementato dagli osservatori.
Pertanto quando interverranno delle modifiche al soggetto osservato, verrà invocato il metodo update() di tutti gli osservatori.

Observer: esempio (3)

```
public abstract class Subject {

    private List<Observer> list = new ArrayList<Observer>();

    public void addObserver(Observer observer) {
        list.add( observer );
    }

    public void removeObserver(Observer observer) {
        list.remove( observer );
    }

    public void notifyObservers() {
        for(Observer observer: list) {
            observer.update();
        }
    }

}
```

Il Subject include una lista degli osservatori che si registrano presso il soggetto osservato tramite i metodi *addObserver()* e si cancellano tramite il metodo *removeObserver()*.
Mentre invece il metodo *notifyObservers()* viene invocato
dalla classe concreta ConcreteSubject quando interviene un cambio di stato.

Observer: esempio (4)

```java
public class ConcreteObserver implements Observer {

    @Override
    public void update() {
        System.out.println("Sono " + this + ": il Subject e' stato modificato!");
    }

}
```

Il *ConcreteObserver* implementa il metodo *update()* per definire l'azione da intraprendere quando interviene un cambio di stato del *Subject.*

Observer: esempio (5)

```
public class ConcreteSubject extends Subject {

    private boolean state;

    public void setState(boolean state) {
        this.state = state;
        notifyObservers();
    }

    public boolean getState() {
        return this.state;
    }

}
```

Il *ConcreteSubject* definisce lo stato del Subject concreto e l'invocazione degli osservatori in caso di cambio di stato.

Observer: esempio (6)

```
public class Client {

    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();
        Observer observer1 = new ConcreteObserver();
        Observer observer2 = new ConcreteObserver();

        //aggiungo 2 observer che saranno notificati
        subject.addObserver(observer1);
        subject.addObserver(observer2);

        //modifico lo stato
        subject.setState( true );

        //rimuovo il primo observer che non sarà + notificato
        subject.removeObserver(observer1);

        //modifico lo stato
        subject.setState( false );

    }

}
```

Creiamo la classe *Client* che si occupa di creare il soggetto da osservare e due osservatori che si registrano per essere notificati in caso di cambio di stato del soggetto osservato. Successivamente rimuoviamo un osservatore e cambiamo lo stato del soggetto osservato per notare che non riceverà più notifiche.

Observer: esempio (7)

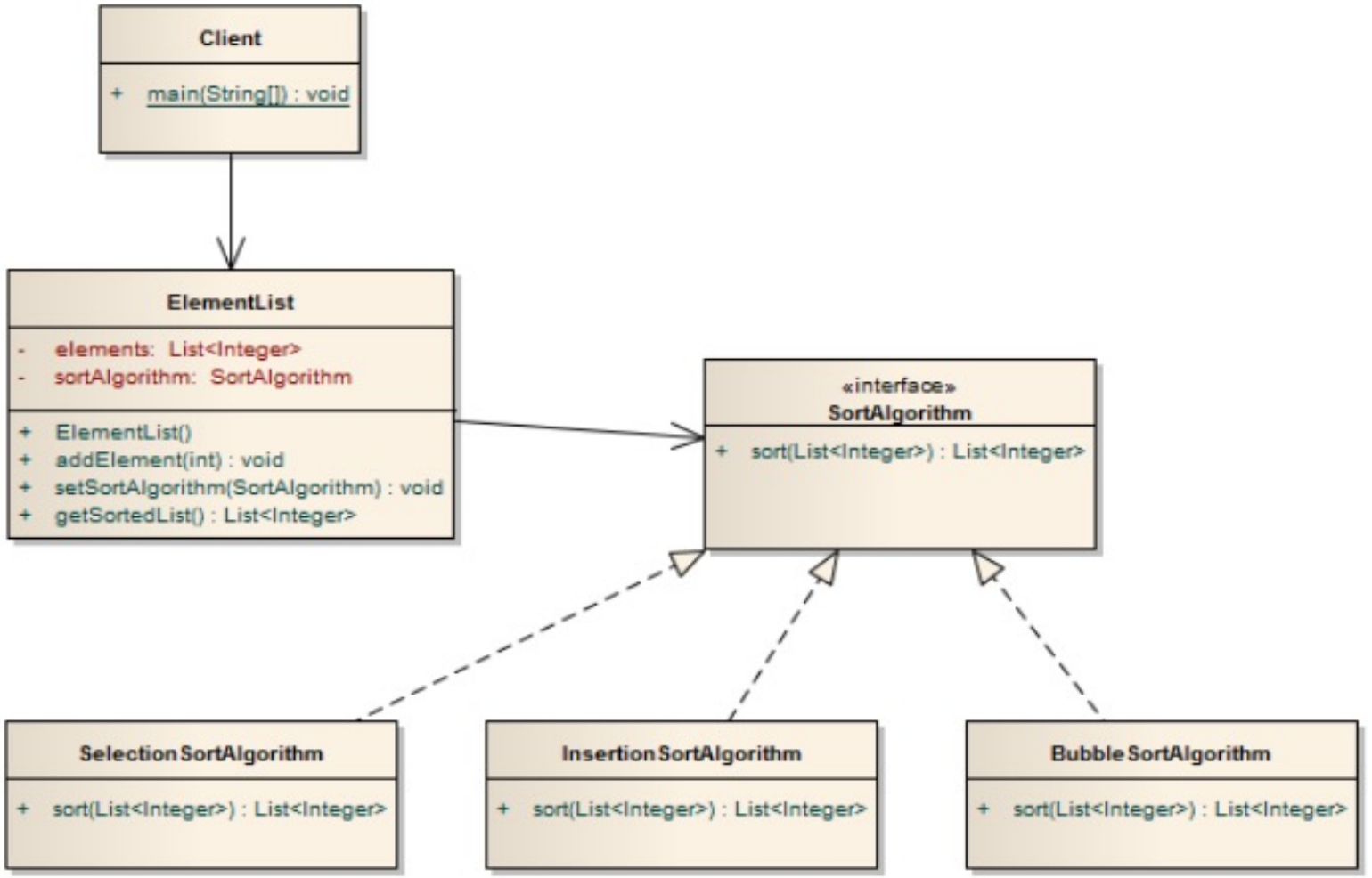
Quale sarà l'output?

```
$JAVA_HOME/bin/java patterns.Client
Sono pattern.observer.ConcreteObserver@4a5ab2: il Subject e' stato modificato!
Sono pattern.observer.ConcreteObserver@1888759: il Subject e' stato modificato!
Sono pattern.observer.ConcreteObserver@1888759: il Subject e' stato modificato!
```

Strategy: esempio

Scenario: Implementiamo il caso di un algoritmo di ordinamento. Possiamo utilizzare diversi algoritmi, dai più semplici ai più complessi: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort ecc. Lasciamo all'utente la libertà di scelta dell'algoritmo da utilizzare ed inoltre lui definirà i dati da essere oggetto dell'ordinamento.

Strategy: esempio (2)



Strategy: esempio (3)

Classe di contesto:

Costruttore *ElementList()*: definisce una lista vuota che ospiterà i valori oggetto dell'ordinamento.

Successivamente verranno caricati i valori da ordinare tramite il metodo *addElement(int… values)*

Infine verrà definito l'algoritmo di ordinamento da utilizzare tramite il metodo setSortAlgorithm(SortAlgorithm sortAlgorithm).

Per ultimo il metodo getSortedList() che invoca l'algoritmo concreto passandogli i dati da ordinare.

Strategy: esempio (3)

```
public class ElementList {

    private List elements;
    private SortAlgorithm sortAlgorithm;

    public ElementList(){
        elements = new ArrayList();
    }

    public void addElement(int... values){
        for(int value: values)
            elements.add( value );
    }

    public void setSortAlgorithm( SortAlgorithm sortAlgorithm){
        this.sortAlgorithm = sortAlgorithm;
    }

    public List getSortedList(){
        return sortAlgorithm.sort( elements );
    }
}
```

Strategy: esempio (4)

```
public interface SortAlgorithm {

    public List sort(List unSortedList);

}
```

L'interfaccia *SortAlgorithm* dichiara il metodo *sort()* che prenderà in ingresso una lista non ordinata per restituirne una ordinata.

Strategy: esempio (5)

```
public class BubbleSortAlgorithm implements SortAlgorithm {

    @Override
    public List sort(List unSortedList) {
        System.out.println("BubbleSortAlgorithm");
        //TODO algoritmo da implementare, fuori contesto
        return null;
    }

}
```

Classe che implementa l'algoritmo BubbleSort (in maniera fittizia)

Strategy: esempio (6)

```
public class SelectionSortAlgorithm implements SortAlgorithm {

    @Override
    public List sort(List unSortedList) {
        System.out.println("SelectionSortAlgorithm");
        //TODO algoritmo da implementare, fuori contesto
        return null;
    }

}
```

Classe che implementa l'algoritmo SelectionSort (in maniera fittizia)

Strategy: esempio (7)

```
public class InsertionSortAlgorithm implements SortAlgorithm {

    @Override
    public List sort(List unSortedList) {
        System.out.println("InsertionSortAlgorithm");
        //TODO algoritmo da implementare, fuori contesto
        return null;
    }

}
```

Classe che implementa l'algoritmo InsertionSort (in maniera fittizia)

Strategy: esempio (8)

```
public class Client {

    public static void main(String[] args) {
        ElementList contextElements = new ElementList();
        contextElements.addElement(new int[]{3,2,4,3,6,5});
        contextElements.setSortAlgorithm(new BubbleSortAlgorithm());
        List sortedList = contextElements.getSortedList();
    }
}
```

Client che utilizza la classe di contesto per:
inizializzare il vettore,
aggiungere gli elementi,
settare l'algoritmo di ordinamento ed infine recuperare la lista ordinata.

Strategy: esempio (9)

Quale sarà l'output?

```
$JAVA_HOME/bin patterns.strategy.sort.Client
BubbleSortAlgorithm
```