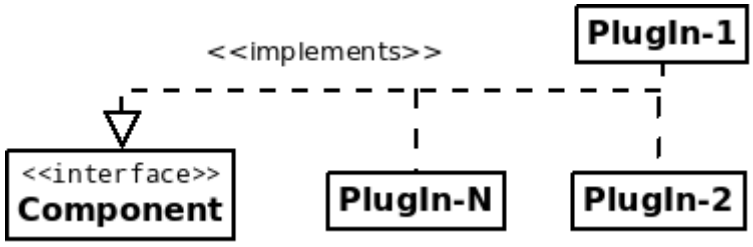
DESIGN PATTERNS – Parte 4

Product Trader
Service Manager
Type Object
Manager
Adapter

PRODUCT TRADER

Il Product Trader è un design pattern per la gestione delle estensioni in modo "componibile", ovvero a plug-in.
Se per esempio avessimo una qualche applicazione che riceve questi plug-in, dovremmo poterli identificare in qualche maniera, magari per esempio attraveso un indice identificativo. La prima idea che ci viene è quella di usare un approccio non orientato ad oggetti, e quindi gestire il tutto con delle istruzioni condizionali come gli if o le switch. Purtroppo questo ci fa perdere la "dinamicità" degli oggetti, e saremmo costretti a riscrivere il codice per ogni plug-in aggiunto. Proviamo quindi a pensare in maniera orientata ad oggetti.
Il primo passo è sicuramente quello di applicare il Polimorfismo e quindi avere uno schema simile a questo:



Fatto questo dovremmo applicare il Pattern Factory per poter generare i vari componenti, quindi avremmo una classe ComponentFactory con un metodo create() per la generazione degli oggetti PlugIn, attraverso uno swich... sembra la soluzione ai nostri problemi, ma purtroppo non è così: mettiamo il caso che la nostra classe Component appartenga ad una libreria, come possiamo fare a generare gli oggetti? Non potremmo farlo, perché il Factory è già stato implementato da altri.

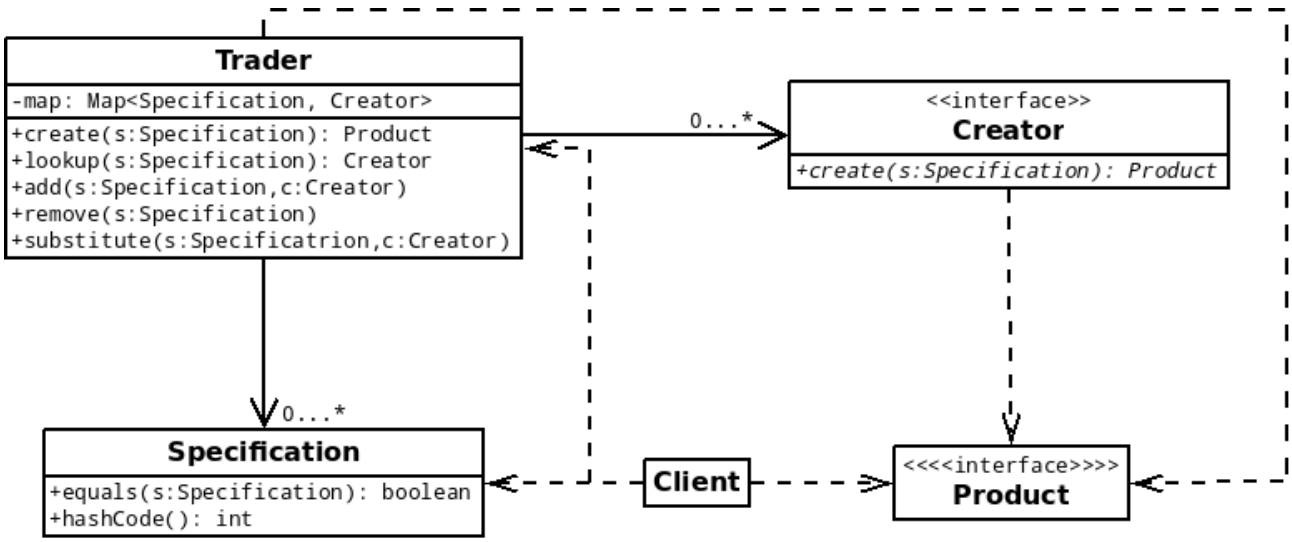
La soluzione della Factory è quindi inapplicabile. Per comprendere meglio il problema, proviamo a vedere chi sono gli "attori" in gioco:

- Chi ha progetto l'API, ovvero la nostra libreria
- Chi vuole usare la libreria
- Chi vuole aggiungere qualcos'altro all'API mantenendo segreti i propri dettagli implementativi

Possiamo dunque dividere il lavoro in due parti: compilazione e runtime.

1) Compilazione:
    1. Il Client può accedere all'interfaccia utilizzando una Specifica del Trader di un'opportuna implementazione/sottoclasse.
    2. Successivamente si potranno scrivere o lasciar scrivere a terzi le varie implementazioni/sottoclassi.
2) Run-time.
    1. Le implementazioni/sottoclassi saranno registrate nel Trader.
    2. Successivamente il Client otterrà un'istanza dell'implementazione/ di una sottoclasse opportuna

In breve questo sarà il nostro Product Trader, dove il nostro Component non sarà nient'altro che il Product, e le varie istanze di esso i vari PlugIn.
Vediamolo in dettaglio nello schema UML:

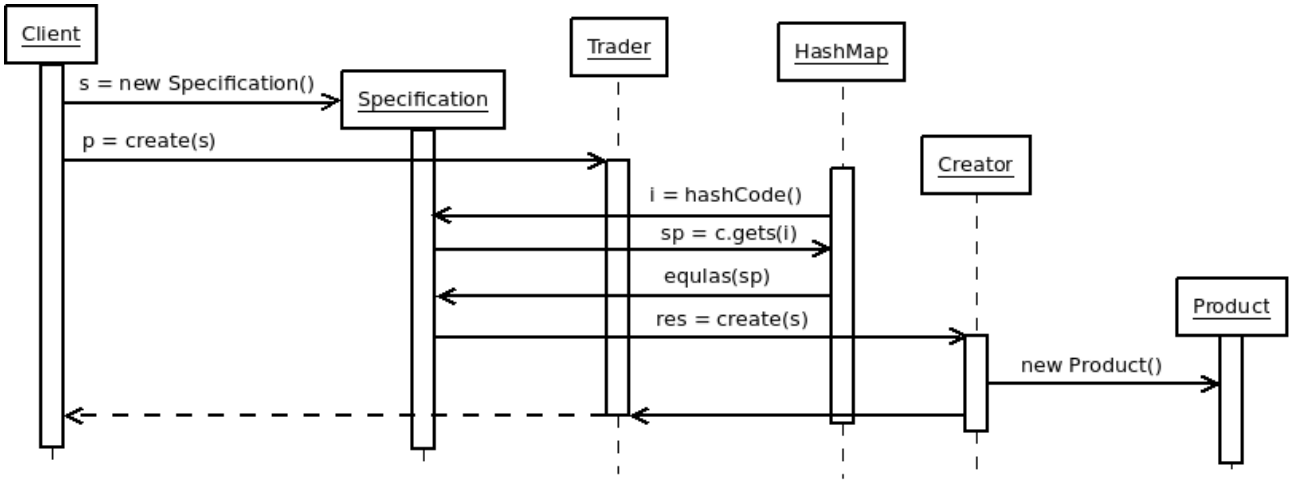


In questo pattern Creator non è che una classe che applica il pattern Factory, essendo però il ruolo di questa classe “speciale” in questo specifico pattern, è stato scelto il nome di Creator.

Come abbiamo detto in precedenza, e che ora possiamo vedere più in dettaglio, abbiamo tutto ciò che ci serve.
Possiamo dunque notare che Client per ottenere il Prodcut di cui ha necessità dovrà chiederlo a Trader attraverso una Specification, la quale sarà usata per identificare il Creator che produrrà il Product di nostra necessità.

Rimane ancora un problema, ovvero chi è che crea la classe Creator. Prima di dare risposta a questo dubbio vediamo di vedere come avverrà il flusso delle azioni e come è strutturato il codice.

Vediamo dapprima il flusso delle azioni con uno diagramma di sequenza:



Vediamo in dettaglio cosa facciamo:

il Client crea una nuova Specification quindi chiede al Trader di creare il Product desiderato. Il Trader per determinare il Product desiderato richiama la funzione hashCode della Specification che gli permetterà di ottenere il Creator desiderato dalla HashMap. HashMap farà dei controlli di equivalenza per tornare l'oggetto giusto (per questo dobbiamo definire i metodi equals() ed hashCode() nella classe Specification). Ottenuto il Creator desiderato, invochiamo il suo metodo create per ottenere il Product di nostro piacimento. Ritorniamo quindi il Product al Client.
Ora che abbiamo visto com'è il flusso di azioni possiamo scrivere il codice. Non serve che creiamo la classe HashMap visto che è già implementata all'interno del package java.util di Java.
In questo esempio non useremo implementazioni, ma faremo direttamente uso delle classi base; al posto di queste implementazioni mancate metteremo dei commenti che indicano la presenza di effettive implementazioni che l'utente stesso o terzi sviluppatori possono creare. La classe Trader inoltre sarà implementata come Singleton.

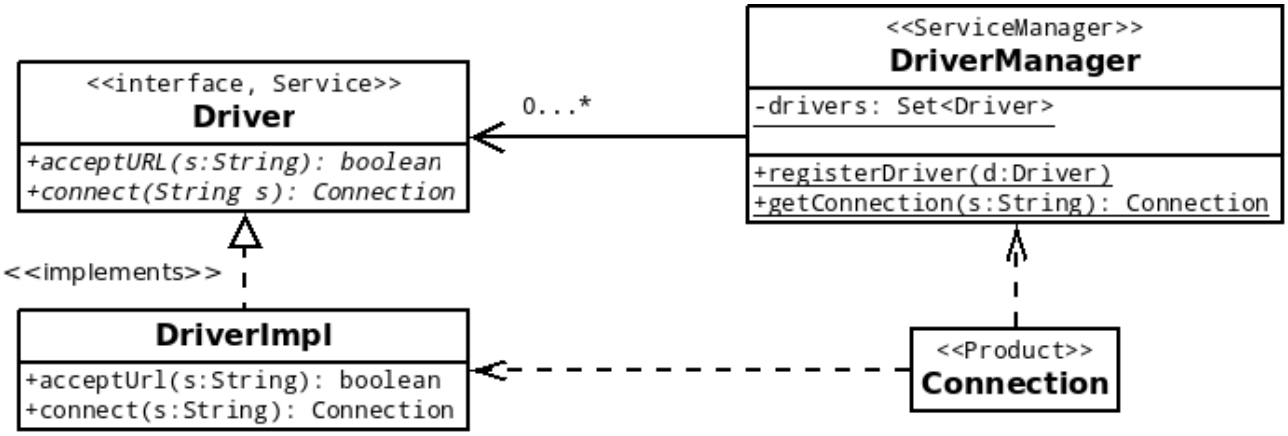
```
//Specification

public class Specification {
     private int codice;

     public Specification(int t) { codice=t; }
     public boolean equals(Object o) {
          if(o instanceof Specification) {
               Specification s = (Specification)o;
               return s.codice==this.codice;
          } else return false;
     }
     public int hashCode() { return codice; }
}
```

```
//Product

interface Product {
     Type argument(Object o);
}
// Classi varie che implementano l'interfaccia Product
// …

//Creator

interface Creator {
     Product create(Specification s);
}
// Classi varie che implementano l'interfaccia Creator
// …

//Trader

public class Trader {
     private static Trader singleton = new Trader();
     private static Map<Specification, Creator> =
                    new HashMap<Specification,Creator>();

     private Trader() { }

     public static Trader getInstance() { return singleton; }

     public Product create(Specification s) {
          Creator c = map.get(s);
          return c.create(t);
     }

     public Creator lookup(Specification s) { return map.get(s); }

     public void add(Specification s, Creator c) { map.put(s,c); }

     public void remove(Specification s) { map.remove(s); }

     public void substitute(Specification s, Creator c) {
          map.put(s, c);
     }
}

//Client

public class Client {
     //Costruttore ed eventuali metodi di Client
     public void doSomething(int code) {
          //...
          Trader t = Trader.getInstance();
          Product p = t.create( new Specification(code));
          //...
}
```

Ora che abbiamo visto il codice resta ancora un quesito irrisolto. Chi è che crea i Creator? Una soluzione può essere utilizzare i file Properties visti col pattern Reflection.

SERVICE MANAGER

Il Service Manager è un pattern derivante dal Product Trader, ed è per l'appunto, una versione semplificata di quest'ultimo. Un esempio di Service Manager sono i driver JDBC. Per comprenderli meglio vediamone la struttura in UML e poi il codice Java. In questo pattern i Creator saranno chiamati Service, mentre il Product Trader sarà chiamato Service Manager.



Il problema sembra essere lo stesso, chi è che crea il DriverManager o lo stesso DriverImpl? Java grazie alla possibilità di eseguire codice al momento del caricamento della classe stessa dal ClassLoader ci permette di risolvere questo angoscioso problema. Vediamo dunque il codice:

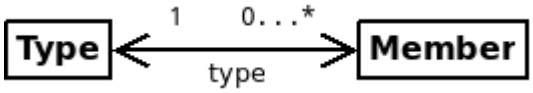
```
//ServiceManager
public class DriverManager {

     static {
          drives = System.getProperty("jdbc");
          String[] driverClass = jdbcDrivers.split(":");
          for(String className: driverClass)
               Class.forName(className);
     }

     public static Set<Driver> drives;

     public static void registerDriver(Driver d){ drives.add(d); }
     public static Connection getConnection(String s) {
          for(Driver d: drives)
               if(d.acceptUrl(s)) return d.connect(url);
     }
}
```

```
//Service
interface Driver {
     boolean acceptURL(String s);
     Connection connect(String s); //Connection è il Product
}

public class DriverImpl implements Driver {

     static {
          Driver d = new DriverImpl();
          DriverManager.registerDriver(d);
     }

     public boolean acceptURL(String s) {
          return s.startsWith("jdbc");
     }

     public Connection connect(String s) {
          //... codice per la connessione
     }
}
```

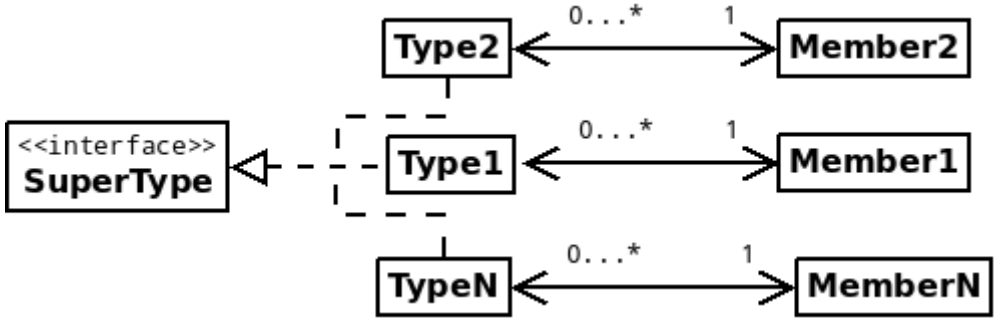
La parte di codice rinchiusa da static vuol dire che verrà eseguita al caricamento della classe, questo significa che quando la nostra applicazione viene lanciata essa già creerà i creatori (il Service) e anche le loro implementazioni (il DriverImpl).

TYPE OBJECT

Questo Pattern è usato quando si tratta di gestire dei tipi/sottotipi ma non è conveniente usare il polimorfismo.
Prendiamo un esempio generale:



abbiamo un certo tipo Type, del quale più oggetti possono essere di quel tipo, e quindi sono dei membri di esso (Member).
La nostra struttura dei dati però è un po' più complicata di così, perché scopriamo che non c'è un solo tipo Type, ma ce ne sono tanti. La prima soluzione che ci verrebbe sicuramente in mente è l'applicare il polimorfismo. Applicando il polimorfismo otterremo ciò:



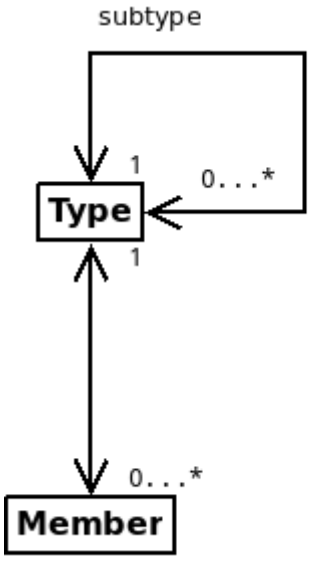
Questo come possiamo vedere causa la creazione di un eccessivo numero di classi, magari per l'appunto, inutili.

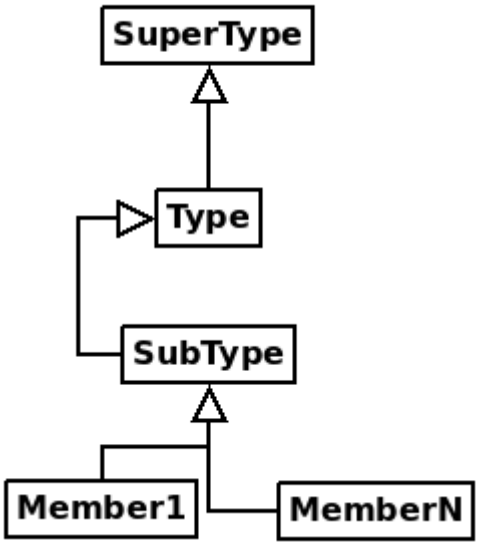
Se usassimo invece il modello descritto precedentemente, che per ora non sapevate ma è il pattern Type Object, possiamo ridurre il tutto in poche righe di codice:

```
public class Type {
     //attributi del tipo
     private Set<Member> membri; //Attributo obbligatorio per
                          //applicare questo pattern efficentemente

     public Set<Member> getMembri() { return membri; }
}

public class Member {
     //attributi del membro
     private Type type; //Attributo obbligatorio per applicare
                    //questo pattern efficientemente

     // metodi del membro
}
```

Come vediamo ogni tipo avrà una lista dei suoi membri, ed ogni membro avrà un riferimento al suo tipo.

Però può capitare di avere anche delle strutture gerarchiche a multi-livello, e in molti casi, non si sa come gestire tutti questi livelli. Anche in questo caso, si può usare il Type Object. Vediamolo applicato in confronto al normale polimorfismo:





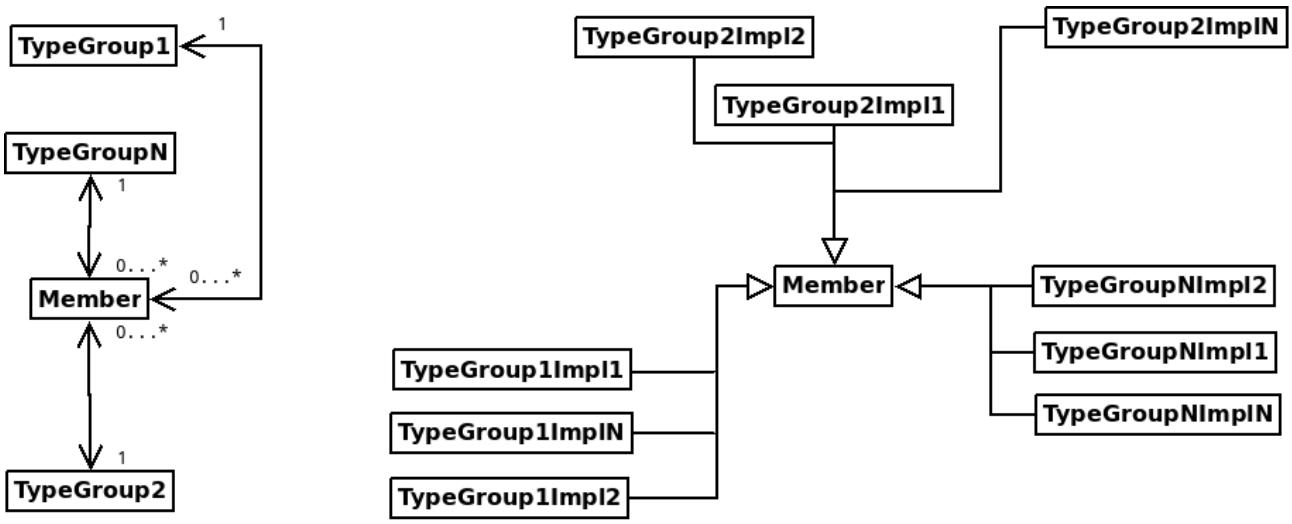
Come possiamo subito vedere anche qui finiamo con creare una marea di classi pressoché inutili. Vediamo dunque il codice delle classi Type e Member

per il caso ricorsivo.

```
public class Member {
     private Type type;

     //metodi e attributi del membro
}

public class Type {
     private Type supertype;
     private Set<Type> subtypes;
     private Set<Member> members;

     //metodi e attributi del tipo
}
```

Una nota di sicurezza ci sta: attenzione quando assegnate ad un tipo il suo superiore o il suo inferiore, assicuratevi che non sia se stesso! Altrimenti rischiamo di finire in un loop ricorsivo infinito!

Un altro caso in cui possiamo applicare il Type Object è quando abbiamo un oggetto che appartiene a più tipi multipli. Vediamo come si presenterebbero i due diagrammi applicando il Type Object e il Polimorfismo:



Si vede immediatamente che l'implementazione con il polimorfismo, come già prima dimostrato crea classi in più decisamente inutili.. e non solo, se il linguaggio da noi usato fosse poco tipizzato, si sarebbe possibile (forse) implementare questa scelta, ma resterebbe comunque di difficile implementazione dovuta in molti casi alla mancanza dell'ereditarietà multipla, che applicandola attraverso il pattern già visto con il polimorfismo, diverrebbe estremamente complicata da attuare.

Per poter applicare il Type Object ad un Member con più tipi multipli, sarà come applicarlo ad un singolo tipo, però pensando che ora bisgona farlo per

ognuno di essi. Vediamo dunque il codice dell'applicazione del Type Object.

```
public class TypeGroup1 {
     private Set<Member> members;

     //metodi e attributi della classe
}

public class TypeGroup2 {
     private Set<Member> members;

     //metodi e attributi della classe
}

public class TypeGroupN {
     private Set<Member> members;

     //metodi e attributi della classe
}

public class Member {
     private TypeGroup1 tg1;
     private TypeGroup2 tg2;
     private TypeGroupN tgN;

     //metodi e attributi della classe
}
```

Vediamo dunque di tirare le somme del Type Object; lo usiamo quando:

- dobbiamo raggruppare per tipo/categoria
- se il numero delle sottoclassi è sconosciuto
- le istanze possono cambiare dopo la creazione
- le istanze sono raggruppate in maniera ricorsiva

come tutti i pattern, ha anch'esso i suoi PRO e CONTRO:
PRO:

- Creazione dei tipi a runtime
- Riduce il numero delle sottoclassi
- Nasconde la separazione dei tipi dall'oggetto stesso. Sfruttando un'interfaccia applicando le regole di Demetra
- Cambiamento dinamico del tipo
- Estensione indipendente tra l'oggetto e il tipo
- Classificazione multipla

CONTRO:

- Bisogna mantenere un riferimento al tipo
- Non risolve il problema di oggetti con comportamento differente (risolvibile con un parser per esempio).

MANAGER

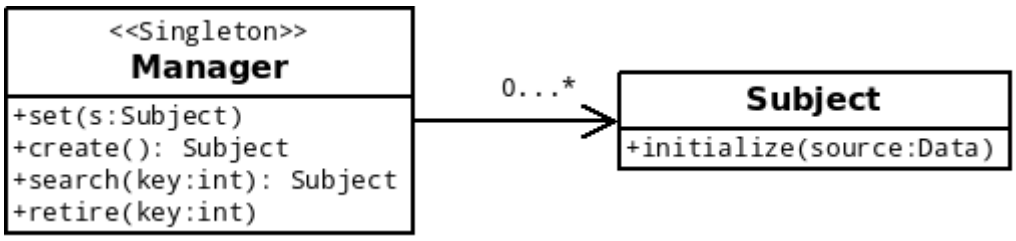
Un pattern per gestire il ciclo di vita di un oggetto, che nasce come caso speciale del Pure Fabrication ed implementa più metodi del normale Factory. In particolare il Manager gestendo il ciclo di vita dell'oggetto, ne gestisce anche gli accessi e le transazioni delle modifiche ad esso.. in più ritorna la stessa istanza dell'oggetto se esso esiste già.
Va dunque usato questo pattern soprattutto quando l'implementazione degli oggetti non devono riflettersi nella gestione del loro ciclo di vita.
Questo Pattern ha quindi i seguenti punti forti:

- Indipendenza degli oggetti dalla gestione del ciclo di vita di essi (possibilità di cambiare la persistenza senza modificare altro)
- Riuso del codice del Manager con più oggetti
- Visione di più oggetti come unica istanza
- Subclassing del Manage

e in seguenti punti deboli:

- Divisione dei compiti tra Manager e Subject spezzano l'idea di incapsulamento
- Se il Manager gestisce sottoclassi della Subject siamo costretti all'uso del pattern Reflection

Vediamo dunque il diagramma UML per comprendere meglio la sua struttura:



e il suo codice:

```
//Manager
public class Manager {
     public static final Manager instance = new Manager();
     private Map<Integer, Subject> subs;
     private Manager() { /* … */ }

     public Subject search(int key) {
          Subject s = subs.get(key);
          if(s == null) {
               s = new Subject();
               s.initialize();
               subs.put(key, s);
          }
          return s;
     }
```

```
    public Subject create(int key) throws Exception {
        Subject s = subs.get(key);
        if(s!=null) throw new AlreadyDefinedException();
        s = new Subject();
        s.put(key, s);
        return s;
    }

    public void retire(int key) throws Excepiton {
        Subject s = subs.get(key);
        if(s==null) throw new NoElementInMapException();
        else subs.remove(s);
    }
}
```

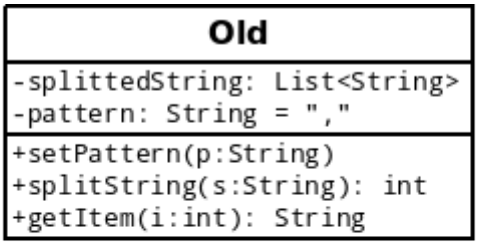
Per migliorare l'uso dei metodi di questo pattern si può pensare di definirlI synchronized sfruttando quindi il mutex interno dei Monitor in Java.

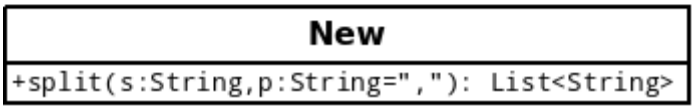
ADAPTER

Questo pattern appartiene ad un gruppo di pattern chiamati Indirection, i quali hanno la seguente caratteristica comune: evitae l'accoppiamento diretto tra due o più oggetti. Un esempio di Indirection è il pattern Observer visto in precedenza.
Ma vediamo in dettaglio i punti chiave dell'Adapter:

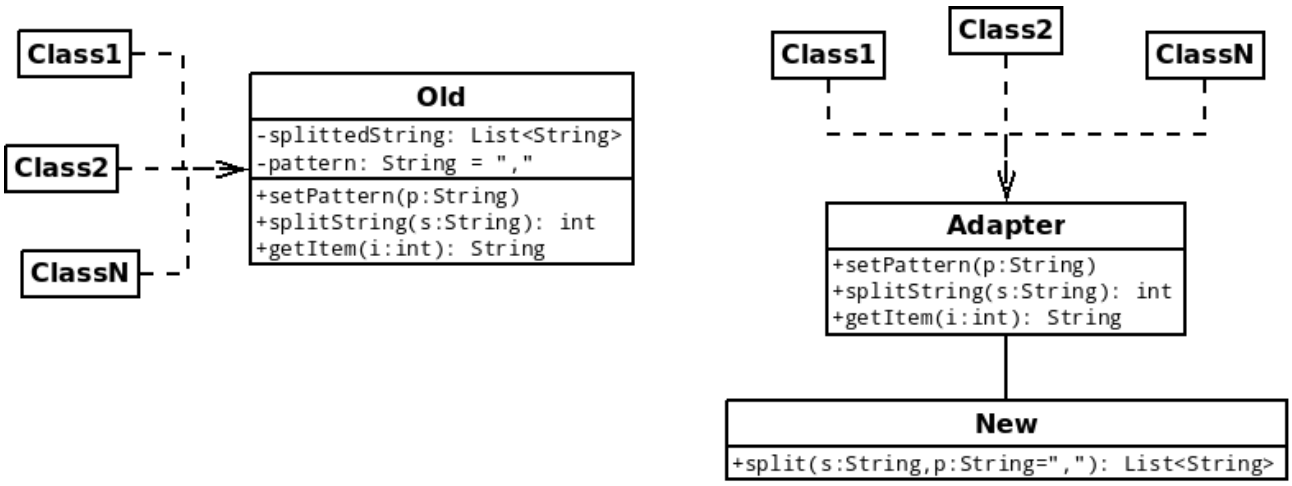
- Gestire interfacce incompatibili o fornire un'interfaccia stabile a componenti simili con interfacce diverse
- Converte l'interfaccia di un componente in un adattatore intermedio.

Un esempio di uso dell'Adapter è quando abbiamo una libreria da sostituire. E la nuova libreria ha metodi simili ma diversi. Prendiamo il seguente caso:





noi vorremmo passare dalla vecchia libreria (Old) a quella nuova (New). Entrambe fanno la stessa cosa, purtroppo però lo fanno con metodi differenti. Vediamo dunque con uno schema UML come le dipendenze della alla classe Old verranno shiftate verso un Adapter che si interfaccerà con la classe New.



Come vediamo la classe Adapter lo stesso nome dei metodi della classe Old, così che nel codice delle classi ci basterà cambiare il tipo dell'oggetto che invoca quei metodi. Vediamo del codice esemplificativo per capire meglio ciò:

```
// --- CODICE VECCHIO --- //
// Chiamate alla classe Old in un metodo della classe ClassN
public class ClassN {
     public void elaborate(String s, String p) {
          Old splitter = new Old();
          splitter.setPattern(p);
          int l = splitter.splitString(s);
          for(int i = 0; i < l; i++)
               System.out.println(splitter.getItem(i));
     }
}

// --- CODICE NUOVO --- //
//Classe Adapter
public class Adapter {
     private String pattern=”,”;
     private List<String> splitstr;
     public void setPattern(String p) { pattern = p; }
     public int splitString(String s) {
          splitstr = (new New()).split(s, pattern);
          return splitstr.size();
     }
     public String getItem(int i) { return splitstr.get(i); }
}
// ClassN modificata in funzione all'applicazione dell'Adapter
public class ClassN {
     public void elaborate(String s, String p) {
          Adapter splitter = new Adapter();
          splitter.setPattern(p);
          int l = splitter.splitString(s);
          for(int i = 0; i < l; i++)
               System.out.println(splitter.getItem(i));
     }
}
```

Come detto, con l'aggiunta della classe Adapter abbiamo potuto facilmente modificare il metodo elaborate() modificando una sola riga di codice mantenendo comunque una completa compatibilità con tutto il resto del codice.
Comunque, mostriamo qui sotto anche il diagramama sequenziale, per una maggiore comprensione del codice vecchio e nuovo: