# Esame PO MOD B – introduzione ai pattern

Di Marco Baratella, contiene sicuramente ORRORI.

# Information Expert

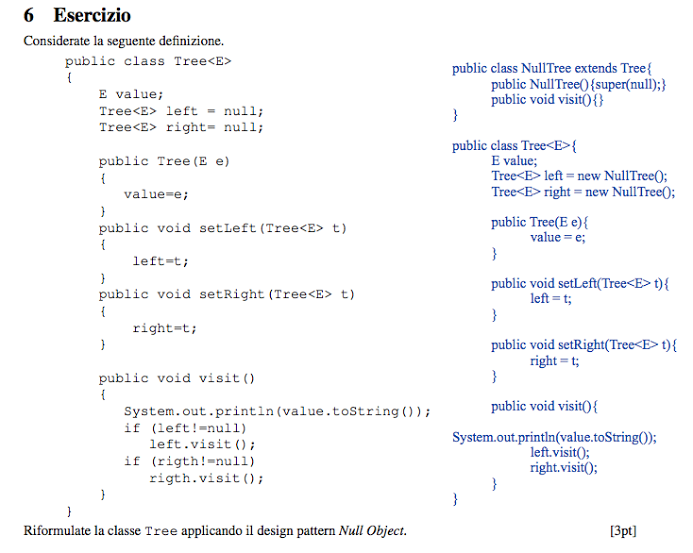
Assegni le mansioni adatte all’oggetto che lo può portare a termine, ad esempio esercizio studente monitor, dentro alla classe studente c’erano metodi di Monitor: sposti dentro a monitor quei metodi.

# Creator

E’ applicabile ad un oggetto da parte di un altro oggetto se : B contiene A (e A è accessibile solo tramite B e A è creato SOLO da B) , oppure B aggrega A con A accessibile esternamente, B memorizza A e B contiene un attributo di tipo A.

# Null Object

Noi utilizzeremo solo la soluzione 2: si crea una classe che estende la classe con molteplicita 0..\*.   
Esempio di applicazione:



# Singleton

Risolve il problema dell’unica istanza ed esistono due tipi di singleton: quello normale e quello Lazy.  
Il singleton preventiva (normale) si applica così:  
**1) Costruttore privato  
2) Variabile privata statica e final Istanza  
3) Getter statico e pubblico che ritorna l’istanza  
4) Tutti i campi e i metodi che in precedenza erano statici ora non lo sono più (si toglie static)**

**Lazy singleton:** si inizializza la variabile private static final istanza a null, e dentro al getter si controlla se questa è nulla o meno, se nulla si crea e si ritorna, altrimenti si ritorna e basta. Ovviamente costruttore privato.

# Pattern Polimorfismo

E’ la base della programmazione e si basa su due utilizzi: ereditarietà e l’implementazione. Deve essere utilizzato secondo questi due punti:  
1) ogni comportamento diverso va gestito da un oggetto diverso  
2) Tante sotto-classi quanti sono i comportamenti  
3) Ogni classe implementa solo i metodi con comportamento diverso

In pratica, si tratta di utilizzare interfacce, sottoclassi ecc per risparmiare codice e per aumentare coesione.  
Va un po’ in contrapposizione con il pattern type-object

# Pure fabrication

Nasce per soddisfare le necessità di divedere la gestione di alcune funzionalità (come la ricerca) che non competono ai comportamenti di una classe  
In poche parole si tratta della creazione di una nuova classe che svolge i compiti a lui pensati, l’uso di pure fabrication è usato per la persistenza dei dati e la visualizzazione di essi. Ha degli svantaggi:  
- necessità di creare una classe in più  
- difficoltà di capire chi è la classe artificiosa  
- visione procedurale anziché ad oggetti (raggruppo per funzioni e non per oggetti).

Utilizzato ad esempio per una GUI o una Session per la WebApp.

# Simple factory

Il simple factory risolve un problema concreto che non è risolto da pure fabrication: chi crea i dati? La gestione l’abbiamo vista risolta con il pure fabrication, ma la creazione dei dati chi la fa? Simple factori: si crea una nuova classe artificiosa chiamata NomeclasseFactory, come nel fabrication, la classe ad esempio datafactory potrebbe aver necessità che i metodi siano statici o che il datafactory sia un singleton, dipende dall’utilizzo di esso.



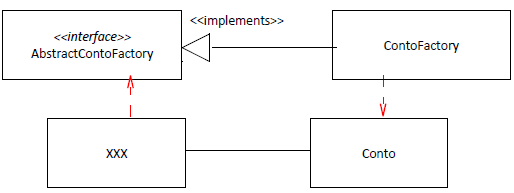
# Factory method

Il factory simple risolve molti problemi ha ancora un difetto: è limitato a un solo metodo. Se volessimo più metodi che creano il nostro oggetto in modo differente ci verrebbe incontro il factory method. Bisogna però rispettare 3 regole:  
**- unica classe (no nuova)**  
**- costruttore unico e privato  
- metodi statici**

# Abstract method

I factory fino ad adesso abbiamo visto che possono essere fatti tramite l’uso di una classe artificiale (simple), ma si può fare anche con un’interfaccia: c’è bisogno di un’interfaccia e di una classe, l'interfaccia definisce almeno un factory

method che poi verrà implementato da una o più classi.



# Protect variations + reflection

Si utilizza il protect variations quando si vuole proteggere un oggetto/sistema dai cambiamenti indotti da altri oggetti. Come? Mascherando i cambiamenti degli altri oggetti tramite un’interfaccia stabile o tramite l’uso dei file di configurazione. Con l’uso dei file di configurazione ridurremmo tutte le dipendenze a solo due dipendenze. I file di conf usati avranno estensione .properties.

# Model view controller

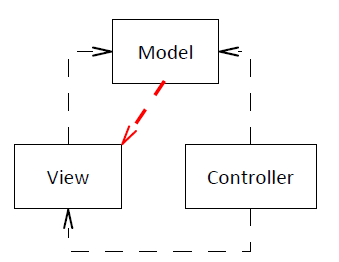
Questo patern nasce per la gestione delle finestre, dove l’app viene divisa in 3 gruppi che sono :  
**- Controller  
- Model  
- View**

**Controller** : rappresenta un oggetto artificioso che coordina le operazioni di sistema delle applicazioni, questa classe può essere divisa in due specifici gruppi:  
- **caso d’uso :** per ogni caso d’uso esiste uno specifico controller  
**- sessione :** per ogni utente esiste uno specifico controller chiamato Session.  
Il compito del controller è quello di collegare le classi del Model al resto dell’applicazione, ovvero alla View.

**Model** : rappresenta tutte le classi che hanno a che fare con il campo d’utilizzo e non con l’interazione dell’applicazione.

**View:** è l’interfaccia grafica dell’app, nel caso di un app a video si riduce alle istruzioni di stampa a video.

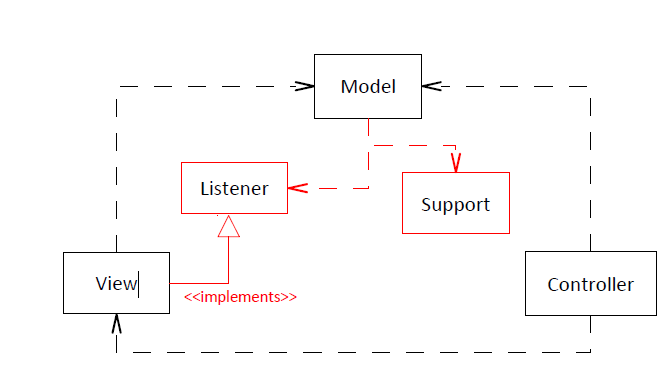
Unendo questi tre elementi si ottiene MVC.



# Observer

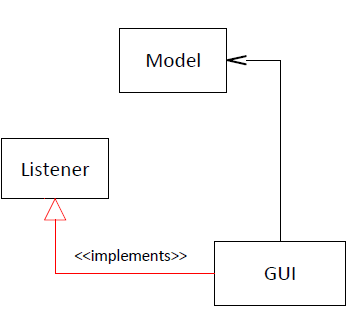
Il pattern observer si basa sull’idea di Subscriber e Publisher, abbiamo più Subscriber interessati ad un certo evento di un oggetto detto Publisher. Ricapitolando, ci sono uno o più oggetti (i subscribers) che sono interessati agli eventi o cambi di stato di un altro oggetto Publisher, il publisher vuole essere quanto più indipendenten dai subscriber.  
  
Per applicare il pattern Observer la soluzione è :  
**1 – si definisce un’interfaccia Listener  
2 – i subscriver implementano Listener  
3 – il publisher registra dinamicamente i subscribers  
4- il publisher avvisa i subscriver registrati quando si verifica l’evento**

Diagramma:

****

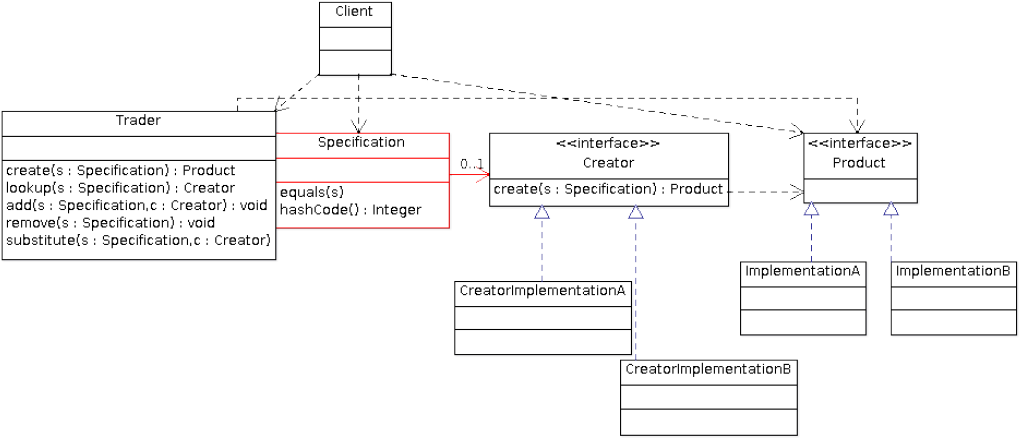
# GUI

Nel pattern GUI Controller e View fanno parte della stessa classe, detta appunto GUI. Per applicare GUI, sposti tutto in un’unica classe, in cui il controller fa parte della classe View. La classe View assorbe i metodi della classe controller.



# Product trader

E’ un pattern per la gestione delle estensioni in modo “componibile”, ovvero i plug-in. Con il PT è possibile rendere il codice pienamente componibile senza dover riscrivere il codice già scritto e eliminare tutti gli switch (o if then else).



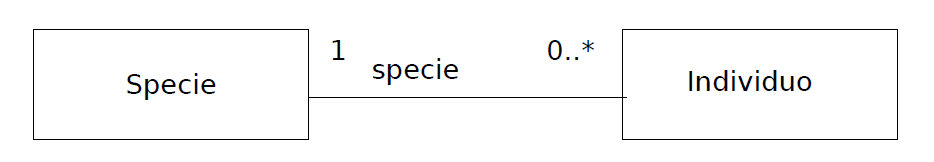
Imparalo a memoria!

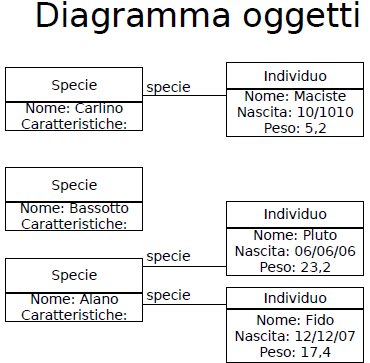
# Service Manager

E’ un pattern derivante da product trader, è una versione semplificata di esso. Un esempio di Service Manager sono i driver JDBC. In questo diagramma UML i creator sono chiamati Service, mentre il Porduct Trader sarà chiamato Service Manager.

# Type object

E’ usato quando si tratta di gestire dei tipi/sottotipi illimitati ma non è conveniente usare il polimorfismo.



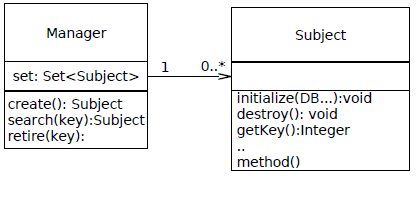


Ogni sottoclasse di Animale diventa una istanza di Specie.

|  |  |
| --- | --- |
| public class Specie { // Type  String nome;  String caratteristiche;  …  Set<Individuo> individui;//specie-1  public Set<Individuo> getIndividui(){  return individui;  }  } | public class Individuo { // Object  Specie specie; //type!!!  String nome;  Date nascita;  double peso;  ...  public String getCaratteristiche(){  //delega;  return specie.getCaratteristiche();  }  ... |

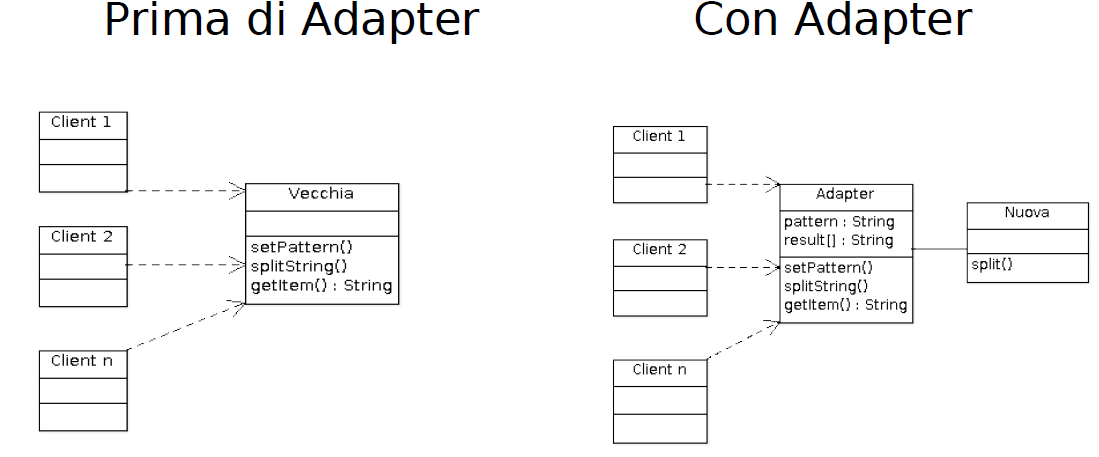
# Manager

E’ un pattern per gestire il ciclo di vita di un oggetto (creazione,distruzione,ricerca …) . E’ un caso speciale di pure fabrication ed implementa più metodi di esso. In particolare il Manager gestendo il ciclo di vita dell’oggetto ne gestisce anche gli accessi e le transazioni delle modifiche ad esso e in più ritorna la stessa istanza dell’oggetto se esso esiste già.   
Quando si usa manager? Quando tutti gli oggetti sono accessibili insieme, o o delle modifiche all’implementazione degli oggetti non devono riflettersi nella gestione del ciclo di vita, oppure quando tante classi diverse hanno la stessa gestione del ciclo di vita.



# Adapter

Questo pattern appartiene al gruppo di pattern chiamati **Indirection**, i quali serono per evitare l’accoppiamento diretto tra due o più oggetti. Anche observer è indirection.   
Adapter si usa quando: devo gestire interfacce incompatibili, o l’adapter server per convertire l’interfaccia di un componente in un adattatore intermedio.   
Un esempio di utilizzo è quando abbiamo una libreria da sostituire, la nuova libreria ha metodi SIMILI ma DIVERSI. Per applicare il pattern, si crea una classe adapter con i metodi della classe vecchia: tali metodi vengono riscritti con i metodi della classe nuova.



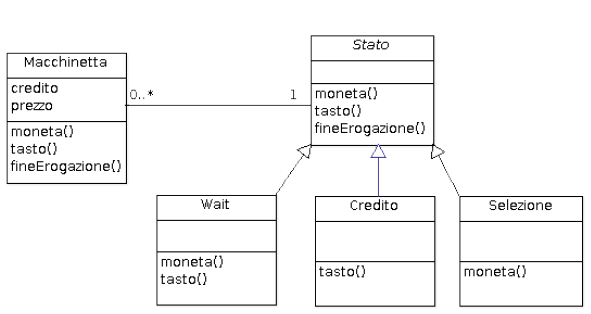
# Facade

Fornisce Protect Variations usando Indirection. Molto simile ad Adapter ma lo scopo è diverso:  
• Adapter per client e server esistenti;  
• Facade è preventivo e/o per sperimentare implementazione diverse;

Favorisce Low Coupling all'esterno del sottosistema, ma non I.E. E L.C. nel sottosistema stesso!  
Di solito si implementa con un Singleton L'interfaccia definisce almeno un factory method che poi verrà implementato da una o più classi.   
Per la gestione dei prodotti non è ancora stato deciso se usare Polimorfismo o Type Obejct. Come facciamo a sviluppare il resto dell'applicazione senza dover dipendere da questa scelta? Definisco un oggetto Facade che nasconde l'interazione con la parte ancora da definire. Nota che li oggetti nascosti sono più di uno e con molteplicità diversa a seconda della scelta.

# State

Pattern state nasce dall’esigenza di poter gestire gli stati di una classe senza dover usare i costrutti come enum e switch.   
Esempio di applicazione di state:

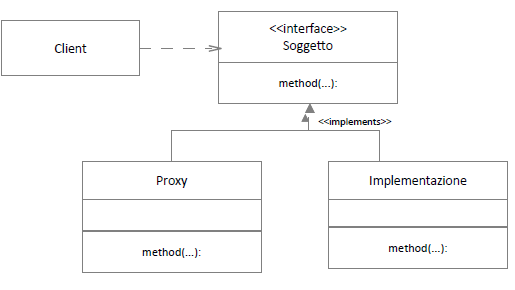


E la state table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SORGENTE** | **DESTINAZIONE** | **EVENTO** | **GUARDIA** | **ATTIVITà** |
| Wait | Wait | On() |  | Accensione |

# Proxy

Quando è un problema accedere direttamente ad un oggetto si usa proxy, questo pattern va a sostituire l’oggetto originale facendo credere all’applicazione di accedere all’oggetto voluto quando invece non l osta facendo.



# RemoteProxy

E’ un caso speciale di proxy, nel quale l’oggetto si trova i un’altra rete. Per rendere possibile l’accesso a questo oggetto si crea un livello intermedio che fa da ponte tra i due mondi.  
Un esempio di RemoteProxy sono gli RMI in Java.

# Failover proxy

E’ usato quando l’accessibilità al dato non è sempre garantino, si usa quindi un oggetto fittizio per sostituirlo momentaneamente finchè non è possibile di nuovo accedere al servizio.