Rispetto ai pattern GRASP, che sono dei principi progettistici, pattern GoF sono più degli “schemi di “ùprogettazione avanzata” inerenti all’istanziazione degli oggetti, essi sono classificati in base al loro scopo

* Creazionale
* Strutturale
* Comportamentale

GoF predilige la composizione rispetto all’eredittarietà di classi ⇒ le funzionalità delle classi sono ottenute tramite composizione di altri oggetti, riutilizzo di classi black box di cui non conosciamo i dettagli.

Questo approccio aiuta a mantenere le classi incapsulate e coese.

Se con l’ereditarietà, la superclasse è definita staticamente e non può essere cambiata a runtime (ergo una modifica alla superclasse può avere ripercrussioni indesiderate), con la composizione si possono usare classi referenziate tramite interfaccia, può esserci quindi una qualsiasi classe che implemente l’interfaccia a runtime e il codice non subisce ripercussiono se non viene alterata l’interfaccia.

Pattern creazionali

**NON HO CAPITO, CHIEDERE INFO**

**Abstract factory:**

* Problema: come creaare famiglie di classi che implementano un’interfaccia comune?
* Soluzione: definire un interfaccia “factory” per ciascuna famiglia di elementi da creare.

**Singleton:**

* Problema: è consentita/richiesta una sola istanza di una classe, come rendo questo oggetto globalmente accessibile?
* Soluzione: definizione di un metodo statico della classe che restituisce l’oggetto singleton, implementato in 3 modi diversi:
  1. classe statica: non è un vero singleton, si lavora con una classe statica (non necessita di essere istanziata) anzichè con oggetti. La classe statica offre tutti i servizi necessari.
  2. creazione del singleton da un metodo statico: la classe ha un metodo che restituisce l’istanza del singleton. L’oggetto viene quindi istanziata solo una volta, dopodichè viene sempre restituita la referenza a quell’oggetto. Se l’oggetto viene istanziato quando il metodo viene chiamato la prima volta, si dice “inizializzazione pigra”, contrapposta “all’inizializzazione golosa” dove la classe viene istanziata quando la classe viene caricara, se poi non viene mai usata è uno spreco di risorse.
  3. Singleton multi-thread: versione multi-thread della soluzioen precedente
* Ossrvazioni: si preferisce la soluzione con metodo statico perchè facilitano il raffinamento di sottoclassi, perchè molti meccanismi di comunicazione remote OO supportano solo l’accesso a metodi d’istanza e perchè un singleton non è sempre tale in tutti i contesti applicativi, la versione con classe statica è meno flessibile insomma

Pattern struttrali

**Adapter**:

* Problema: fornire un interfaccia stabile per comportamenti simili ma interfacce non compatibili
* Soluzione: utilizzo di un oggetto adattatore intermedio: converti l’interfaccia originale in un’altra interfaccia

**Composite**

* Problema: come trattare un gruppo o una struttura composta di oggetto allo stesso modo di un oggetto atomico?
* Soluzione: definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

**Decorator:**

* Problema: assegnare responsabilità addizionali ad un oggetto dinamicamente, evitando il problmea della relazione statica, provvedere ad un alternativa flessibile al meccanismo di sottclasse evitando gerarchie di classi complesse
* Soluzione: inglobare l’oggetto all’interno di un altro che aggiunge nuove funzionalità (detto “wrapper”)

**Composite VS Decorator:**

composite fornisce un interfaccia comune a elementi atomici (foglie) e composti, mentre decorator fornisce caratterische addizionali alle foglie mantenendo l’interfaccia comune originale.

Pattern compostamentali

**Observer:**

* Problema: una serie di oggetti “subscriber” vogliono reagire, ognuno a proprio modo, al cambiamento di stato/eventi di un oggetto “publisher”. Come fare, mantenendo allo stesso tempo un accoppiamento basso?
* Soluzione: definizione di un’interfaccia subscriber o listener, che viene implementata dagli oggetti subscriber. Il publisher registra dinamicamente gli oggetti interessati ai suoi eventi e li avvisa quando questo si verifica.
* Osservazioni:
  + La relazione di dipendenza è 1 a N
  + Il publisher è disinteressato della natura dei subscribers se non l’interfaccia che implementano ⇒ disaccoppiamento

**State:**

* Problema: il comportamento di alcuni oggetti dipende dallo stato dell’oggetto stesso, riflettendo azioni condizionali in funzione dello stato. Come evito la logica condizionale?
* Soluzione: creazione di classi si stato con interfaccia comune. Le operazioni che dipendono dallo stato vengono delegate all’oggetto dello stato corrispondente. Importnate: assicurarsi che l’oggetto contesto referenzi sempre un oggetto stato che riflette lo stato corrente

**Strategy:**

* Problema: come progettare per gestire algoritmi e politiche variabili ma correlati? Come consento queste modifiche?
* Soluzione: definizione di ciascun algoritmo/politica in una classe separata, con interfaccia comune. L’algoritmo viene applicato all’oggetto contesto, associato ad un oggetto “strategia” che implementa l’algoritmo
* Osservazioni:
  + Consente la definizione di famiglie di algoritmi intercambiabili tra loro
  + Indipendenza degli algoritmi rispetto al clienti che lo usano
  + Diaccoppiamento algoritmi - clienti
  + Consente modifiche runtime
  + Usa la composizione, non l’ereditarietà

**NON HO CAPITO, CHIEDERE INFO**

**Visitor:**

* Problema: come separare l’operazione applicata su un contenitore complesso dalla struttura dati a cui è applicata?

Io cane