Design pattern 1

Introduzione

Obiettivi:

- Applicare propriamente i pattern del modello GoF

Prima di introdurre i particolari dei pattern del modello GoF, si definiscono le lucidazioni necessarie per comprendere l'etimologia del termine.

Definizione informale
Ogni pattern descrive uno specifico problema e la sua soluzione associata, per fare in modo che la soluzione possa essere ripetuta anche in nuovi contesti del modello.

Riassumendo il contesto, l'intento consiste nella decodifica dell'esperienza vissuta, affinchè ciò che è stato riscontrato possa essere adeguato in altri contesti. Esistono tante altre definizioni del termine, ma basti pensare alla correlazione diretta presente tra la denominazione del problema e della soluzione, pur di ottenere l'idea su cui fonda un pattern.

Spesso i *pattern* sono raggruppati in strutture coese, ossia accomunate da stesse caratteristiche, le quali danno vita a vocabolari capaci di esprimerne la *sintassi* e la *grammatica*, oppure, a titolo semplificativo, il *contesto* e il *caso d'uso*. Tuttavia, sono presenti un numero elevatissimo di cataloghi inerenti ai pattern applicati in ingegneria del software, ma in questa sezione verrà adoperato il *vocabolario* posto dalla *Gang of Four*.

GoF

Stabilita un'introduzione al contesto cardine della sezione, si approccia nel dettaglio ciò che contraddistingue maggiormente l'insieme dei pattern del *catalogo GoF*. I pattern caratterizzati si suddividono in tre macro areee, poste come segue:

- *Creational*, ...
- *Structural*, ...
- *Behavioral*, ...

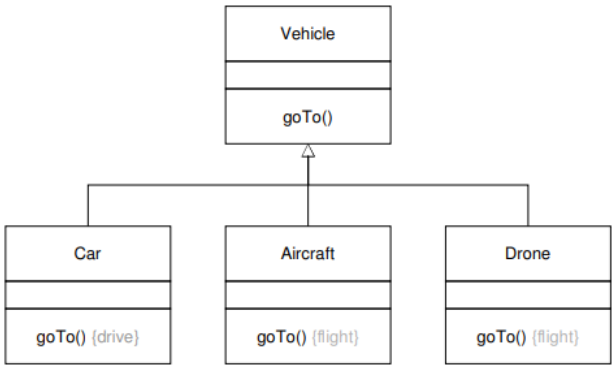
Tuttavia, prima di illustrare nel dettaglio tutti i pattern di riferimento alle classificazioni precedenti, occorre descrivere una tematica su cui fonda la propria logica il meccanismo *GoF*.

Analizzando nuovamente linguaggi legati al paradigma degli oggetti, per adeguare la soluzione software ad un'elevata *reusability* occorre ottimizzare l'utilizzo di caratteristiche particolari, quali *ereditarietà* e *poliformismo*. Per cui l'obiettivo di un qualsiasi team di sviluppo, consiste nella possibilità di riutilizzare porzioni di codice già attutate per risolvere problemi passati. Ciò è possibile tramite il corretto uso di meccanismi legati alle caratteristiche del linguaggio di programmazione adottato, ma non rappresenta l'unica soluzione.

In questo ambito, inerente al catalogo GoF, è posta la preferenza nell'utilizzo di meccanismi legati alla **delega**, o meglio definiti di **composizione**, piuttosto che strumenti correlati ad

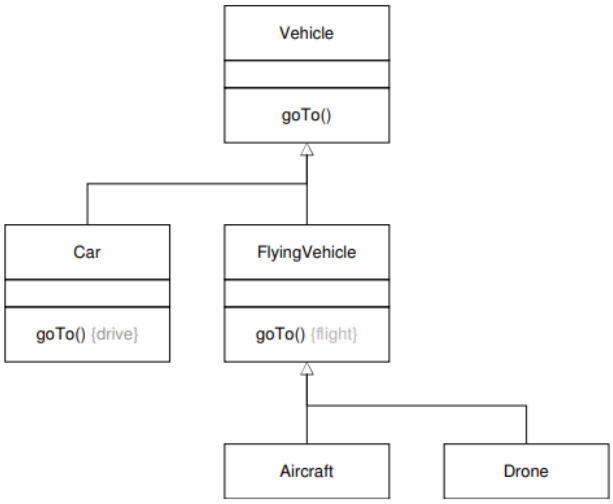
inheritance, pur di aumentare la qualità di riutilizzo. La ragione principale è dovuta dalla maggiore conformità del modello proposto rispetto ad un elevato livello di incapsulamento, in cui oggetti di ogni singola classe possano svolgere un unico compito, ossia siano associati ad un solo *comportamento*. Tuttavia di seguito è proposto un caso di studio che possa dimostrare la maggiore efficacia.

Caso di studio
All'interno dell'esempio è illustrato un insieme di step che possano evidenziare come l'ereditarietà in certe casistiche non risulti essere una soluzione efficace; proponendo una via alternativa, di maggiore correttezza, dovuta al meccanismo di *delega*.

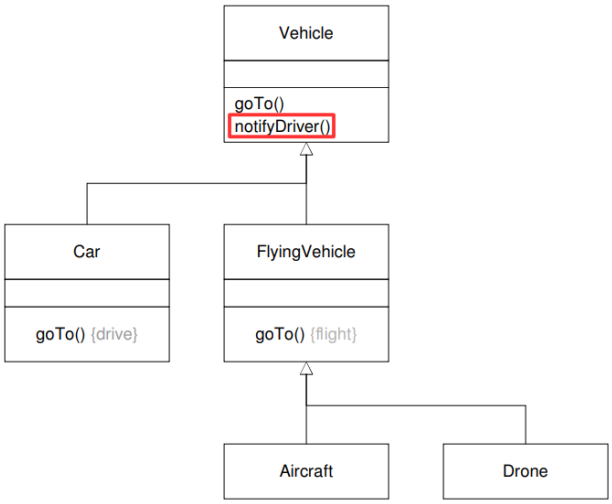


Come da raffigurazione, sono presenti un totale di quattro classi software, di cui *Vehicle* risulta essere la superclasse. In tale domain model l'implementazione adeguata propone un metodo generale posto nella classe padre denominato *goTo()*, *ereditato* anche dalle classi figlie le quali attuano uno sviluppo personale a seconda del comportamento che debba essere compiuto, per cui adeguando *poliformismo*.

Nonostante possa apparire una soluzione valida, viola il principio di *Liskov*, il quale ammette che se si dovesse verificare nella soluzione software *needless repetition*, ossia l'azione *copia-incolla* tra classi, come avviene per *AirCraft* e *Drone*, allora rappresenta una mancata visualizzazione di un livello di astrazione nella progettazione adeguata fino ad ora. Ciò comporta all'adozione di un'interfaccia che sia sovrapposta tra *Vehicle* e le classi figlie.

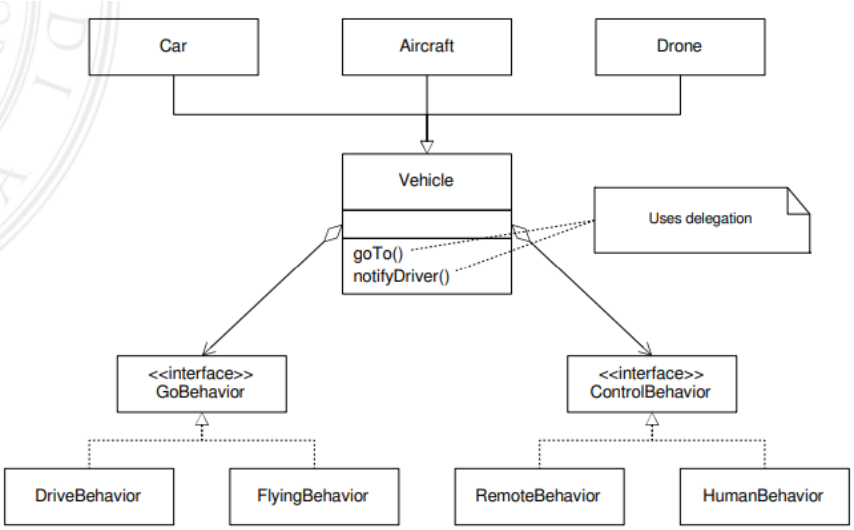


Un meccanismo simile risulta essere solido fino a quando non sia proposta un'ulteriore funzionalità da sviluppare. Infatti, qualora all'interno della superclasse sia posto un ulteriore metodo che dovrà essere ereditato dalle sottoclassi, quale *notifyDriver()*, dimostrandosi totalmente differente per ogni elemento del modello, qualsiasi livello di astrazione non sarebbe sufficiente per poter rispettare i principi legati al modello GRASP, diminuendo la qualità del codice.



Concludendo, rispetto a quanto detto prima, sarebbe violato il principio *Open Closed*, il quale ammette che in una qualsiasi classe software possano essere aggiunti metodi ma non modificate funzionalità già esistenti, ciò che non rispetterebbe l'implementazione di *notifyDriver()* in ogni singola sottoclasse.

Osservati i limiti che si potrebbero verificare in un meccanismo legato all'*ereditarietà*, una via risolutiva comprende la nozione di *composizione*. Tramite lo strumento della *delega*, è introdotto un livello di astrazione capace di suddividere le classi legate alla logica algoritmica da classi *concrete*.



La raffigurazione ritrae la classe *Vehicle* strettamente connessa all'interfacce *GoBehavior* e

ControlBehavior, poichè è parte del layer di astrazione; tale scelta è dovuta alla necessità di porre un elemento del modello come mediatore tra l'implementazione dei comportamenti rispetto alle richieste delle classsi derivate. Per cui saranno incapsulate le due istanze delle interfacce all'interno della classe padre, affinchè classi figlie possano reclamare all'interno del costruttore quale implementazione dei metodi *goTo()* e *notifyDriver()* siano necessarie. Si osserva un vero e proprio meccanismo di delega che contraddistingue la *composizione*.

In conclusione, sono riassunti i due punti critici di *inheritance* che comportano alla scelta della *composizione* e del meccanismo della *delega*, quali:

- *Poliformismo*, non accetta overriding dei metodi da parte di classi figlie, ciò che è stato già riscontrato all'interno di *Liskov Substitution Principle*. Si ricorda l'affinità con il principio di sotituibilità, il quale ammette che ogni istanza di una sottoclasse sia interpellabile da una superclasse; tuttavia questo non è sufficiente, poichè si richiede anche affinità comportamentale, in cui dovrebbe essere atteso lo stesso comportamento descritto a livelli soprastanti, ossia specifica alla logica logaritmica
- *Condivisione comportamentale*, strettamente collegato al punto precedente, in cui pur di implementare metodi comportamentali si aggiungono layer architetturali tra classi padre e classi figlie, illustrando una crescente dissonanza con *behavioral compatibility*

Tutto ciò si traduce in azioni mirate che comportino lo spostamento del codice implementato per la rappresentazione di comportamenti al di fuori di contesti dipendenti, inserendo l'implementazione attuata in classi esterne.

Template Method (Behavioral Pattern)

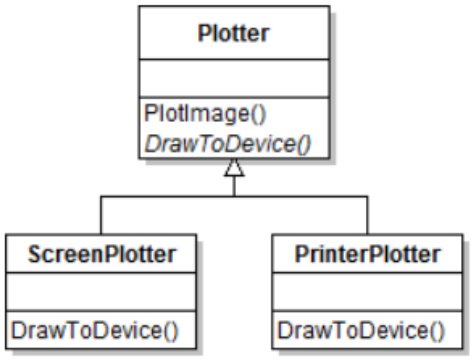
Problema
Come condividere metodi parzialmente definiti in una gerarchia ereditaria?

Soluzione
Definire lo schema dell'algoritmo di certe funzionalità, le quali saranno poi specializzate, in base alle necessità, dalle sottoclassi.

Template Method rappresenta uno dei *pattern comportamentali* del *catalogo GoF*, il quale permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono. In questo modo è possibile ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune.

E' bene sottolineare che tramite questo meccanismo non avviene overriding dei metodi, anzi la definizione generale della struttura dell'algoritmo permette a classi figlie di implementare solamente step specifici che dovranno essere contraddistinti a seconda del comportamento richiesto.

Caso di studio
Di seguito è proposto un esempio che prende in considerazione l'effettività di un plotter, suddivisa da due comportamenti specifici adeguati in due sottoclassi sottostanti.



Come da raffigurazione è rappresentata la classe astratta *Plotter*, la quale definisce lo schema generale dei metodi incapsulati al suo interno. In tale contesto le sottoclassi *ScreenPlotter* e *PrinterPlotter* adeguano il metodo *DrawToDevice()*, specializzando mirate funzionalità affinchè il comportamento sia conforme con le finalità comportamentali, ovviando all'overriding dell'intero metodo.

Tramite tale approccio è ovviato il design smell *needless repetition*, per cui rispettando appieno il principio di *Liskov*, rendendo l'architettura resistente ai cambiamenti.

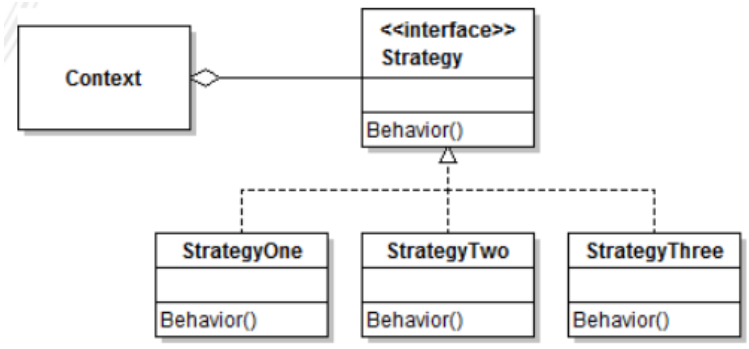
Strategy (Behavioral Pattern)

Problema
Come separare un oggetto dal proprio comportamento e variarlo durante il run time?

Soluzione
Definire un insieme di algoritmi, i quali risultino incapsulati, affinchè siano intercambiali durante il run time.

Strategy per certi aspetti è molto simile al pattern presentato nella sezione precedente, in cui vige una differenza sostanziale. In *Method Template* è definita la struttura generale dell'algoritmo affinchè classi figlie possano specializzare il comportamento a seconda della necessità, mentre in *Strategy* non avviene un meccanismo di suddivisione comportamentale, pone un livello astratto, tipicamente un'interfaccia, da cui saranno estesi *behavior* specifici i quali potranno essere richiamati dalla classe concreta.

Casi di studio



In questo esempio la classe *Context* mantenendo al suo interno istanze dell'interfaccia che implementa i comportamenti specifici, è in grado di richiamare il *behavior* che più si addice al contesto in cui è posta. Grazie a *Strategy*, è preferita la *composizione* rispetto all'*ereditarietà*, favorendo dinamicità anche durante fasi di run time; dato che i comportamenti derivano dalla stessa interfaccia, la classe *Context*, imponendo attributi differenti, è in grado di intercambiare i metodi messi a disposizione.

In conclusione un caso di studio simile rispetta il principio *Open Closed*, ossia non promuove la modifica di funzionalità già esistenti ma introduce nuove classi che possano implementare i comportamenti desiderati.