Design pattern

Introduzione

Obiettivi:

- Comprendere qualora sia possibile applicare pattern del modello GRASP e GoF (Gang of Four)

Prima di introdurre il *catalogo dei pattern* inerenti alla *banda dei quattro*, si definiscono le lucidazioni necessarie per comprendere il termine *pattern*.

Definizione
Ogni pattern descrive un problema che avviene frequentemente e la soluzione associata alla problematica, 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 questo contesto si farà riferimento al *vocabolario* posto dalla *Gang of Four*. Rappresenta un caso di studio popolare e di estrema importanza, il quale include un totale di 23 pattern ritenuti utili durante la progettazione e implementazione del sistema software di riferimento.

Documentazione

In relazione al catalogo di pattern proposto, sono presenti alcune tematiche focalizzate principalmente sulla documentazione correlata, in cui sono proposte alcune chiavi di lettura, suddivise come segue:

- *Pattern name e classification*, indica un nome informativo ed unico utilizzato per l'identificazione del pattern desiderato
- *Intent*, è attuata una descrizione dell'obiettivo del pattern e la ragione che porterebbe ad una sua implementazione
- *Motivation*, indica una raffigurazione in cui è esplicitato il problema e il contesto in cui il pattern possa essere adeguato
- *Structure*, uso di un diagramma per l'illustrazione dei pattern contenuti, come un class diagram oppure un interaction diagram, simili ai charts presentati nella sezione UML, ma non del tutto equivalenti, dove solitamente variano le interazioni e associazioni tra gli elementi del modello
- *Consequences*, rappresenta i risultati, gli effetti e i compromessi causati dall'uso del pattern in questione

- *Related patterns*, indica le dipendenze con altri elementi del catalogo derivanti dall'uso del pattern di riferimento ed è accennata una breve discussione che contraddistingue particolarità sottili rispetto a possibili corrisposti

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*.

Le due tecniche principali per adeguare la qualità *reusability*, in linguaggi legati al paradigma degli oggetti, consistono nell'implementazione di ereditarietà e composizione. Per cui, è proposto l'intento che comporti al riutilizzo di porzioni oppure di intere soluzioni software già attuate per decifrare un problema sorto; ciò può avvennire tramite meccanismi di inheritance, ereditarietà, o meccanismi di delega. Tuttavia, in questo ambito, è imposta la preferenza nell'attuazione di meccanismi di composizione piuttosto che meccanismi di ereditarietà, poichè la volontà ricade nel mantenimento di un certo livello di incapsulamento per ogni classe del dominio e improntare ognuna di esse su un unico compito da svolgere.

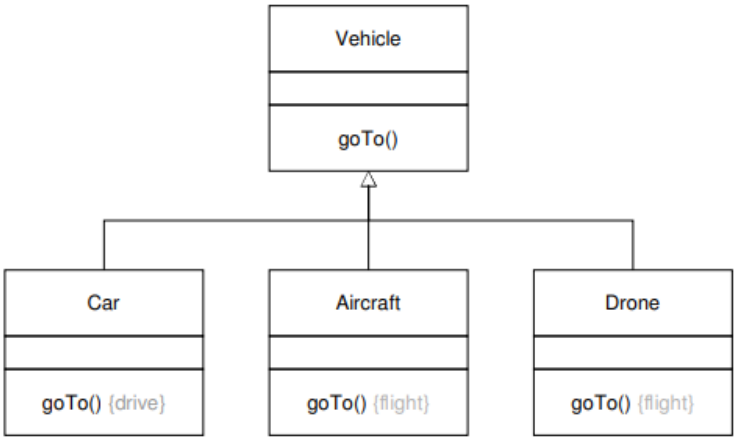
E' bene sottolineare che tra i due meccanismi proposti non vige un'ampia diversificazione, anzi la *delega* promuove la creazione di una composizione potente quanto possa essere stabilito tramite *ereditarietà*, ma con l'unica contraddizione della sintassi adeguata nei confronti di classi derivate, i quali comportamenti non sono derivanti dalla classe padre, ma essa ottiene propri comportamenti in base alle specifiche, ossia possiede propri atteggiamenti.

La scelta che preclude totalmente l'ereditarietà è dovuta a due problematiche strettamente collegate:

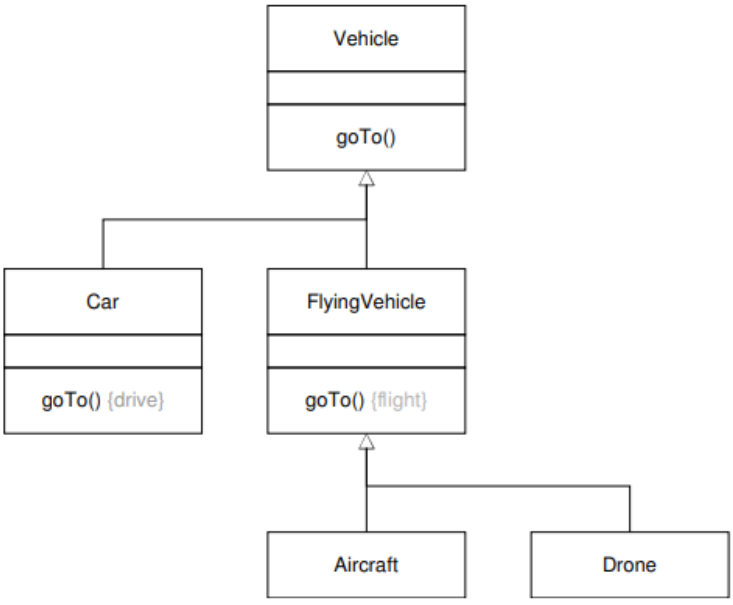
- *Poliformismo*, non accetta overriding dei metodi da parte di classi figlie, ciò che è stato già riscontrato all'interno di *Liskov Substitution Principle*. Si ricorda l'affinità con il principio di sotituibilità, il quale ammette che ogni istanza di una sottoclasse sia interpellabile da una superclasse; 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*

Caso di studio
Di seguito è proposto un caso di studio che possa evidenziare problematiche simili a quanto

riportato, in cui prima si illustrano avversità legate all'uso di ereditarietà per poi contraporre la risoluzione data dalla composizione.



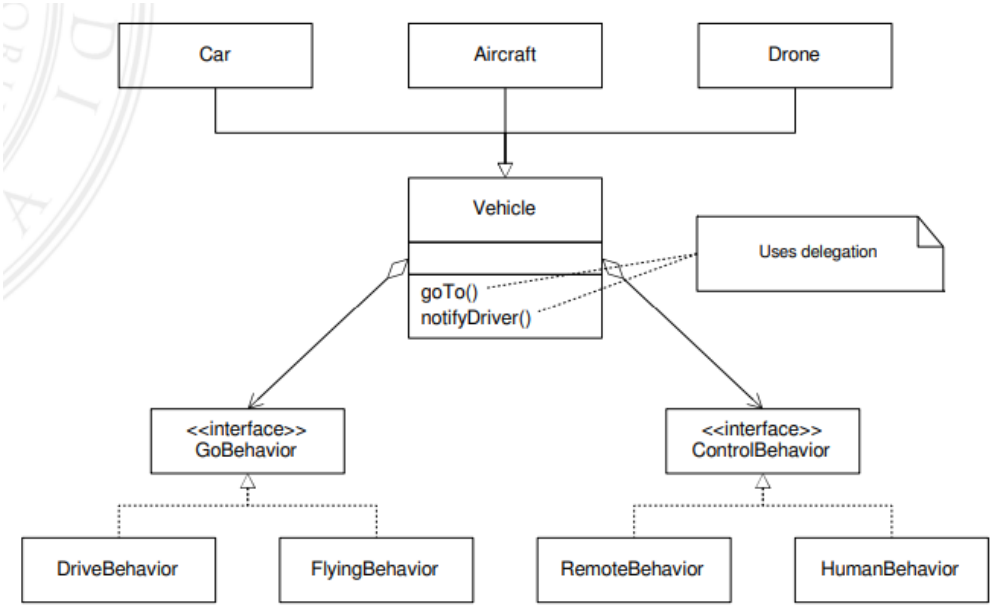
Come da raffigurazione, sono poste un totale di quattro classi, di cui *Vehicle* risulta il padre. In tale domain model si osserva la volontà di voler implementare per ogni classe figlia un certo comportamento, *goTo()*; tuttavia è violato il principio di *Liskov*, non è mantenuta unità comportamentale, data la violazione che avviene tra *Car*, *Drone* e *AirCraft*, inoltre è presente un evidente *design smell*, il quale simboleggia chiaramente la mancata qualità della progettazione, denominato *needless repetition*, posta dall'implementazione di *goTo()* in elementi caratterizzati dallo spostamento aereo piuttosto che terreno. Dato il segnale di errore, si evince come una ripetizione dello stesso codice in differenti elementi del dominio, anche se fossero contraddistinti da lievi modifiche, indica un mancato livello di astrazione nella progettazione. (goTo() varia a seconda della classe, a cui è applicato consecutivamente poliformismo pur di ottenere un'implementazione specifica). Necessita una soluzione, la quale se applicato il meccanismo di ereditarietà consiste nella figura seguente.



La soluzione consiste nell'uso nell'elemento di eccellenza dei linguaggi orientati al paradigma

degli oggetti, ossia le interfacce, grazie alle quali si evita *needless repetition* ed è possibile mantenere un grado elevato di *behavioral compatibility*. Tuttavia, l'introduzione di un'ulteriore funzionalità nella classe padre potrebbe provocare l'intera rottura del discorso imbastito fino ad ora; se dovesse essere implementato un metodo simile a *notifyDriver()* il quale si dimostra differente per ogni classe posta a livelli sottostanti, pur di rispettare i principi proposti prima, sarebbe impossibile adeguare layer di astrazioni legate ad interfacce poichè provocherebbero una complessità architetturale troppo elevata, violando il principio *Interface Segregation*. Concludendo, sarebbe violato anche il vincolo posto da *Open Closed Principle*, a causa della modifica che dovrebbe subire il metodo *notifyDriver()* in ogni sottoclasse.

Per poter ovviare alle problematiche descritte si potrebbe fare uso di *meccanismi di delega*, come la composizione, in grado di stabilire il livello di astrazione necessario, strumento molto più flessibile e in grado di variare in *runtime*.



Tramite il meccanismo della composizione si visualizza un'illustrazione totalmente differente rispetto alla precedenti, in cui coloro che giocano un ruolo fondamentale sono le classi *GoBeahvior* e *ControlBehavior*, oltre agli elementi da cui derivano. Mediante una semplificazione simili è mantenuto un livello di flessiibilità estremamente elevato, in cui *Vehicle* implementando le due interfacce raffigurate garantisce alle classi come *Car*, ... e ... di poter implementare i metodi elencati in base alle necessità presentati; saranno create due istanze di riferimento nella classe principale, dove le classi che derivano da *Vehicle* specificheranno nel proprio costruttore quali siano le caratteristiche sufficienti per ottennere un certo livello comportamentale, per poi successivamente la classe padre provvederà alla richiesta dei comportamenti attessi in base alle specifiche ottenute. L'implementazione delle interfacce prevede l'uso della *realizzazione*, in cui esse sono definite dalle singole classi riportate, legate alla logica algoritmica, le quali provvedono alla creazione dei comportamenti specifici richiesti prima e adeguati erroneamente prima, come toGo() per veicoli terrestri oppure veicoli aerei.