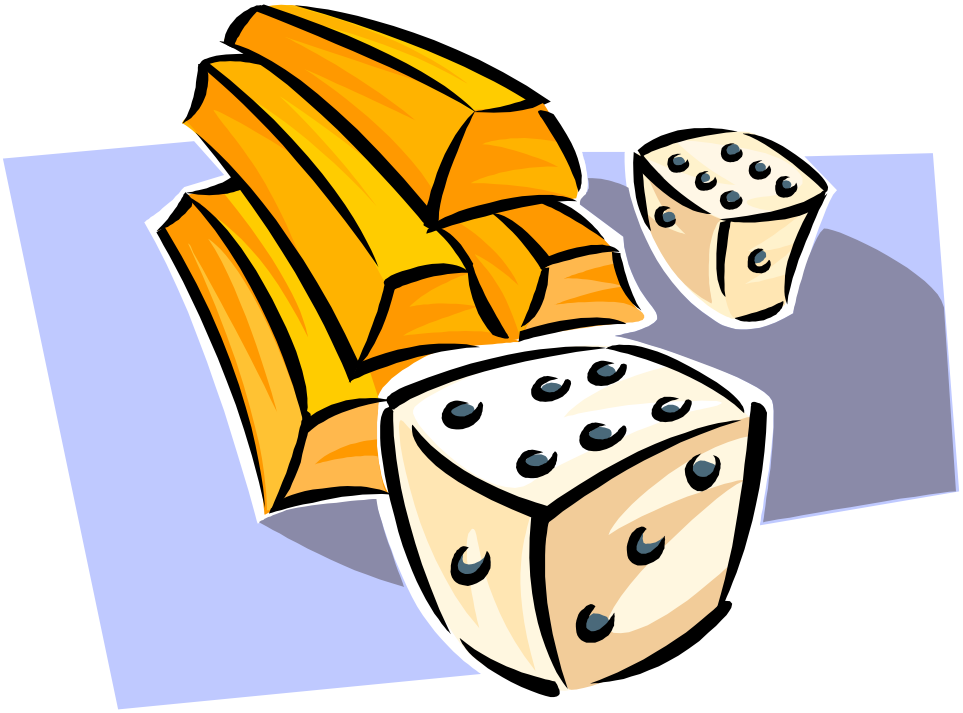
Design Pattern

Pattern-e-book: Una guida nella jungla del OO



ing. Rosario Turco

Serie e-book

I pattern forniscono una soluzione a problemi ricorrenti, in base al contesto in gioco.

I pattern sono uno strumento capace di gestire il cambiamento di un progetto.

I pattern sono degli elementi progettuali capaci di fornire maggiore flessibilità.

Terza edizione
100 pagine

Design Pattern

Pattern-e-Book: Una guida nella jungla del OO

ing. Rosario Turco

Terza edizione

L’autore si è fatto carico della preparazione del e-book, degli esempi e dei programmi didattici, non completi, in esso contenuti. L’autore non si assume responsabilità, esplicita o implicita, per i programmi didattici o il contenuto del testo e non potrà in nessun caso essere ritenuto responsabile per eventuali danni dovuti ad un uso improprio dei programmi didattici.

Terza edizione 2011

Sommario

Prefazione

L'Object Oriented è un potente paradigma informatico, fine anni '80, che ha assunto un'importanza notevole nell'analisi e nella progettazione dei sistemi software.

Nell'ultimo decennio, in particolare, sono stati fatti passi significativi in tutti i settori dell'Object Oriented:

- definizione di metodologie di modellazione dei sistemi software;
- definizione degli standard di rappresentazione (UML, OCL, etc);
- creazione di strumenti a supporto della modellazione e dello sviluppo software;
- ideazione dei processi organizzativi (Rational Unified Process, Estreme Programming, Agile, Catalysis etc).

Il presente testo si propone di colmare, almeno in piccola parte, l'assenza degli autori italiani nel settore delle pubblicazioni professionali su problematiche di modellazione con i design pattern. Si pensa, quindi, di fare cosa gradita a tutti i tecnici italiani nel presentare un e-book centrato esclusivamente sulla modellazione del software finalizzata ai design pattern.

Il tentativo è stato quello di scrivere un e-book leggibile, leggero e comprensibile, per una vasta fascia di figure professionali; ma soprattutto un e-book, di 100 pagine, destinato a far comprendere principalmente i concetti di base OO, dietro tale tipo di modellazione.

L'e-book è orientato, quindi, ad essere una breve guida concettuale nella complessa e intrigata jungla della modellazione OO con i principali Design Pattern. Non è da ritenersi, però, una guida esaustiva o completa e si invita a continui approfondimenti su vari cataloghi, poiché è sempre un settore in continua evoluzione.

Napoli, 01 dicembre 2010 L'Autore

La seconda edizione

Con la prima edizione l'autore sottoponeva al vaglio dei lettori l'opera parziale, per verificarne la validità e l'utilità ad essa attribuita; i lettori però hanno incoraggiato il completamento dell'opera, in seconda edizione, con contatti e suggerimenti all'autore.

La seconda edizione ha provveduto essenzialmente ad una revisione generale, ad introdurre le Best Practices Object Oriented propedeutiche ai Design Pattern, a completare la parte GoF dei Design Pattern, a riorganizzare gli ultimi capitoli sulle ulteriori classificazioni e ad aggiungere delle conclusioni filosofiche di natura Extreme Programming e della Metodologia Agile.

Per contatti:
rosario_turco@virgilio.it

Napoli, 24 dicembre 2010

La terza edizione

La terza edizione ha provveduto essenzialmente alla correzione di alcuni errori e ad estendere la parte de "I miei libri preferiti".

Napoli, 18 agosto 2011

Il mondo dell'Object Oriented

L'Object Oriented si è evoluto molto nel corso degli anni, al punto da rendere praticamente indistinguibili i confini tra analisi, progettazione e sviluppo.

Nel forward engineering, l'utilizzo di strumenti RAD, come Together, Rational Rose, Visual UML, Eclipse etc. fanno sì che la modellazione renda le fasi di progettazione e di sviluppo come un tutt'uno.

Il "forward engineering" ed il "reverse engineering" permettono, poi, di avere un ciclo continuo progettuale, nell'ambito di un processo, ad esempio l'Unified Process (UP), con possibilità sia di produrre software dalla modellazione, sia di riaggiornare la documentazione/modellazione partendo dal software esistente prima della prossima iterazione progettuale.

L'Object Oriented è contenuto nell'EIP e nella progettazione a componenti. In questo caso si tratta di una progettazione top-down: architettura EIP, framework, componenti e Design Pattern.

Inoltre, alcune tecnologie recenti come Java Data Object (JDO), rendono ancora più marcatamente indistinguibili la progettazione del dominio e quello delle entità del database; anzi spesso e volentieri la progettazione del database è solo una conseguenza dell'analisi Object Oriented, ovvero della modellazione del dominio e delle classi. Questo, a differenza del passato dove l'integrazione dati e delle funzioni avveniva a valle di due progettazioni parallele: database e funzionalità; il che permette di avere un immediato controllo ed integrazione delle due cose sin dall'inizio. JDO, si può affermare, permette al database relazionale di riappropriarsi del terreno perduto così come che era avvenuto in passato per le classi e i metodi rispetto alle funzioni. Tecnologie come JDO consentono, difatti, di concentrarsi solo sul dominio delle entità di business, demandando a opportune classi preposte la gestione della persistenza sul database.

In questa evoluzione del mondo OO la stessa figura del Data Base Administrator (DBA) si è evoluta in termini Object Oriented; come pure diversi tools di modellazione conducono alla produzione di modelli di database (vedi Rational Rose o altri).

La stretta dipendenza delle fasi di analisi e progettazione, inoltre, portano alla conseguenza di avere nel team di progetto figure miste come analista/progettista e progettista/sviluppatore. Lo stesso architetto è una figura che mette insieme diversi skill: analista/progettista/sviluppatore e spesso anche sistemista.

In altri termini gli skill ideali da inserire o formare in un team di lavoro sono quelli che hanno "lo stesso strato di know-how d'interfaccia", perché devono capirsi facilmente, procedendo sicuri e senza malintesi nel lavoro.

Il presente e-book è orientato, quindi, ad almeno tre figure professionali:

- architetto di sistema (System Architect)
- analista/progettista (dominio e database)
- progettista/sviluppatore

In un buon team, accanto alle figure professionali precedenti, ovviamente ne esistono anche altre di ugual importanza e necessità: Business Analyst, Usability Engineer, Tester, Test Developer, Document Writer, Project Manager, Function Point Engineer.

Il Business Analyst è l'interlocutore primario del cliente; è colui che conosce e apprende il business del cliente ed il cui compito primario è di saper trasmettere in modo concettuale, chiaro e non ambiguo tali informazioni vitali al resto del team.

La figura del "Usability Engineer" pone forte attenzione alle problematiche di usabilità ed esercibilità del colloquio uomo-macchina; quindi si occupa della progettazione, secondo concetti UX (User Experience), della User Interface o della Web Interface.

La figura del Tester si occupa di progettare ed eseguire il test del sistema software, valutandone la rispondenza ai requisiti e puntando ad una bassa difettosità residua del prodotto.

La figura del Test Developer è quella di una persone a supporto dei test, che utilizza librerie come JUnit, che sviluppa i simulatori di sistemi esterni, generatori di dati e integra il tutto con prodotti come Rational Test Manager.

Il Project Manager è preposto alla gestione del progetto secondo le tre famose variabili: tempo, numero di persone e qualità da ottenere.

Il Function Point Engineer è colui che riesce a fare, in base ai requisiti in gioco, una stima di quanto dovrà essere sviluppato esprimendo il tutto in "Function Point". Alla stima iniziale farà seguire una misura effettiva di quanto sviluppato.

La figura però del System Architect è delicata e determinante. E' un conoscitore delle tecnologie, della modellazione (Design Pattern, Components e Enterprise Integration Pattern), delle architetture e delle integrazioni.

Negli ultimi anni le architetture si sono orientate sempre di più verso i seguenti elementi metodologici:

- SOA (Service Oriented Architecture)
- ESB (Enterprise Service Bus)
- BPM (Business Process Management), in BPM-notation per la modellazione e BPEL per l'esecuzione
- JBI (Java Business Integration)

Con SOA si sono ottenuti servizi auto-consistenti e verticali, basati sui Web Services; con l'ESB la possibilità di avere un bus di comunicazione e disaccoppiamento tra i sistemi aziendali.

Col BPM l'architettura si è focalizzata sull'orchestrazione dei servizi per produrre un processo di business (in BPMN per la modellazione e BPEL per l'interpretazione del software da parte del container – vedi esempio INTALIO Community Edition), in modo da creare e riusare servizi disponibili sul Bus, assemblandoli in vari modi in un processo (workflow), e ottenere una architettura che logicamente è divisa in servizi e processi.

Il JBI ha, invece, introdotto la possibilità (vedi ServiceMix, Camel, Fuse, Mule etc.) di ottenere BUS scritti con poco software Java e maggiormente configurabili, mirando a prodotti semplici, spingendo alla modellazione architetturale e di integrazione denominata EIP (Enterprise Integration Pattern).

Un'altra grande rivoluzione proviene dalle tecnologie più light in termini di sviluppo come Ruby oppure Groovy, che è una estensione miglioramento e semplificazione di Java, interscambiabile con Java, con maggiori features, possibilità di usarlo in modalità script o compilato …

A chi è rivolto l'e-book

L'e-book, qui proposto, non ha alcuna ambizione di rappresentare un catalogo di Pattern esaustivo, compilato con tutto il rigore tecnico del caso. Ciò non è mai stato, fin dall'inizio, l'obiettivo di tale e-book.

Sul mercato mondiale esistono testi di notevole rigore formale e di grosso valore a partire dal famoso libro "Design Patterns – Elementi per il riuso di software a oggetti - Gamma, Helm, Johnson, Vlissides", consigliato da leggere e approfondire.

Alcuni di questi libri hanno il vantaggio-svantaggio di essere degli ottimi cataloghi, utili a selezionare i design pattern in base al problema ed al contesto in cui ci si trova; facendo questo però si "perde" la forza concettuale del "perché e del come" è nato il Design Pattern.

L'e-book affronta la modellazione da una prospettiva diversa, non da catalogo, mostrando quali sono i principi di base Object Oriented per ottenere la migliore soluzione possibile, in termini di flessibilità, e come in tale soluzione siano presenti proprio i Design Pattern di cui parla il catalogo.

Solo dopo una fase di maturazione dei concetti ritengo utile l'uso di un catalogo, sempre a portata di mano nel cassetto della propria scrivania.

Procedendo con entrambe le prospettive si ottiene una rapida maturazione delle motivazioni e dei principi che sono alla base dei Design Pattern.

I miei testi preferiti

Un prerequisito per la lettura del presente testo è la conoscenza dei concetti di base Object Oriented e della modellazione UML.

Mi è sembrato utile fornire anche un elenco di testi, scelti tra i miei preferiti, suddividendoli per argomenti ed in un "ordine consigliato" di apprendimento.

Tali testi sono quelli con cui ho notato dei validi progressi, sia miei che di coloro a cui ho spiegato le cose apprese e costituiscono l'approfondimento a questo e-book.

Mi scuso, chiaramente, per eventuali testi e autori qui non citati o per mia ignoranza o per il livello troppo spinto che esula dagli obiettivi del presente e-book.

La maggioranza dei testi è in lingua inglese ed è frutto di tecnici stranieri, mentre qualcun altro è una buona traduzione in italiano.

Tali testi coprono varie aree:

- Principi di base Object Oriented;
- UML e modellazione
- Progettazione a Componenti
- Design Pattern
- J2EE Pattern
- Rational Unified Process
- Architetture
- Integrazione

Object Oriented	
Titolo	**Autore - Editore**
Progettazione a oggetti con UML	Meiler Page-Jones - APOGEO

UML	
Titolo	**Autore - Editore**
UML Explained – edizione italiana	Kendal Scott – ADDISON WESLEY
The Unified Modeling Language User Guide	Grady Booch, James Rumbaugh, Ivar Jacobson – ADDISON WESLEY
UML Distilled – edizione italiana	Martin Fowler - ADDISON WESLEY
Sviluppo di sistemi informativi con UML	Leszeck A. Maciaszeck – ADDISON WESLEY
UML Components – edizione italiana	John Cheesman, John Daniels – ADDISON WESLEY

UML, Patterns, Unified Process	
Titolo	**Autore - Editore**
Applying UML and Patterns – An introduction to Object Oriented Analysis and Design and the Unified Process	Craig Larman – Prentice Hall
Design Patterns – Elementi per il riuso di software a oggetti – edizione italiana	Gamma, Helm, Johnson,Vlissides – ADDISON WESLEY
Analysis Patterns – Reusable Object Models	Martin Fowler – ADDISON WESLEY
Design Patterns Explained – A new perspective on Object Oriented Design	Allan Shalloway, James R. Trott – ADDISON WESLEY
Developing Enterprise Java Applications with J2EE and UML	Khawar Zaman Ahmed, Cary E. Umysh – ADDISON WESLEY
Core J2EE Patterns – Best Practices and Design Strategies	Deepak Alur, John Crupi, Dan Malks – PRENTICE HALL
Rational Unified Process – edizione italiana	Philippe Krutchen – ADDISON WESLEY
Patterns of Enterprise Application Architecture	Martin Fowler – ADDISON WESLEY

Enterprise Integration	
Titolo	**Autore - Editore**
Enterprise Intergration Patterns	Gregor Hohpe, Bobby Wolfe – ADDISON WESLEY

Suggerimenti e segnalazioni dei lettori

L'autore ha profuso la massima attenzione, durante tutta la stesura dell'e-book, scegliendo opportunamente tutti gli argomenti e gli esempi.

Saranno, comunque, gradite tutte le segnalazioni d'errori, d'omissioni o "passi" del e-book non completamente chiare.

Si possono inviare segnalazioni, commenti e domande, inerenti questo e-book, all'e-mail dell'autore rosario_turco@virgilio.it .

Ringraziamenti

In generale la stesura di un libro non è mai un'attività semplice, né breve. E' un processo iterativo, con fasi ben precise di studio, di selezione degli argomenti, di esperimenti d'installazione, di implementazione, di revisione e di rifacimento di interi capitoli in breve tempo. Questo e-book, anche se più in piccolo, non ha fatto differenza.

Non si è mai soddisfatti al 100% di quanto scritto e rimane sempre qualcosa di incompiuto; la stesura di un libro, ad esempio, è sempre un continuo ed utile confronto tra autore, revisore, editore e professionisti che collaborano alla realizzazione dell'opera.

Ringrazio la mia famiglia, soprattutto mia moglie Teresa e i miei figli Mario e Rosa , per la pazienza avuta, a causa del mio temporaneo "periodo di assenza intellettuale", ma soprattutto perché mi hanno incoraggiato nell'ideazione e nella realizzazione del e-book.

Biografia dell'autore

Rosario Turco è un ingegnere elettronico, laureato all'Università Federico II di Napoli, che lavora dal 1990 in società del gruppo Telecom Italia.

Le sue competenze professionali spaziano dalle architetture Object Oriented (OOA/OOP, AOP, SOA) alle problematiche "Java 2 Enterprise Edition", con progettazione e sviluppo di sistemi informatici su piattaforme Windows/Unix/Linux e con linguaggi Java, C, C++, Groovy, C#.

Attualmente si occupa di architetture Oracle BEA, TIBCO e J2EE, con componenti ESB, BPM e del mondo Open Source (ServiceMix, INTALIO, Apache Synapse, Groovy etc).

A titolo hobbistico è project leader della distribuzione Linux "LyNiX". E' autore di molti articoli su www.scribd.com/rosario_turco (Vedi Collections).

Capitolo 1 - Vecchi problemi e nuove soluzioni

A molti sarò capitato di ascoltare gli analisti senior del proprio team affermare che i tipici problemi che si devono sempre affrontare nella realizzazione di un progetto software sono:

- L'incompletezza dei requisiti;
- Una frequente erroneità dei requisiti, con inutili ricicli;
- Una non completa visibilità dell'obiettivo finale a cui partecipa il requisito stesso.

La verità fondamentale è che, in realtà, occorre saper accettare una dura regola e saperci convivere per sempre: "Un requisito tende sempre a cambiare"!

Questo accade per ovvie ragioni:

- Il punto di vista dei clienti tende a evolvere: dopo una riunione con i fornitori o dopo un aggiornamento di conoscenze o anche a causa di un confronto col mercato o con altri settori dell'azienda;
- sono nate nel frattempo nuove opportunità di business;
- Il punto di vista, la sensibilità e la maturazione del dominio del problema da parte dello stesso fornitore tende a cambiare in meglio col tempo;
- l'ambiente di sviluppo, i tool e le metodologie a supporto tendono a cambiare e ad fornire strumenti migliori.

Se il cambiamento dei requisiti lo si accetta come "fatto naturale ed inevitabile", allora l'unica cosa da fare e poter convivere con tutto ciò è di adattare il proprio processo di sviluppo, le metodologie e gli strumenti a supporto, in modo da poter gestire il tutto in maniera che avvenga nel modo più efficace ed efficiente possibile.

I nuovi orizzonti dell'OO mirano proprio a questo!

I nuovi orizzonti dell'OO

I nuovi orizzonti sono nati dal momento in cui si è riusciti a dare una risposta diversa a delle domande chiavi e interpretando le risposte in modo nuovo.

Gli oggetti sono solo contenitori di dati?

No, gli oggetti non sono soltanto dei contenitori di dati, ma sono delle entità che hanno *responsabilità*. La responsabilità dà all'oggetto un comportamento. Questa affermazione permette di focalizzarsi su cosa l'oggetto deve fare (che comportamento ha?) e non su come lo deve fare. In altri termini permette di focalizzarsi sulla "vista della specifica" e non su quella dell'implementazione. E' più facile pensare, quindi, in termini di interfaccia pubblica (prototipo).

L'incapsulamento serve solo nascondere i dati?

Tradizionalmente una classe permetteva, attraverso l'incapsulamento, di fare un "hiding data", cioè di nascondere i dati.

In realtà l'incapsulamento può essere visto anche come un modo per *"contenere le variazioni di comportamento"*.

Facciamo un esempio: supponiamo che io abbia un ombrello di colore grigio metallizzato, che ha posto per 5 persone e che nei giorni di pioggia mi ci riparo, sedendomi in esso. (Ma che razza di ombrello è? Mah!). Ovviamente è uno strano ombrello, anzi è un'auto. Anche l'auto permette di ripararmi dalla pioggia, ma definirla un ombrello è certamente una "definizione troppo restrittiva".

L'incapsulamento non è solo un concetto di "hiding data": è anche un "hide implementations", un "velare" le classi derivate (sottoclassi), cioè è anche un incapsulamento di classi. Vediamo un esempio.

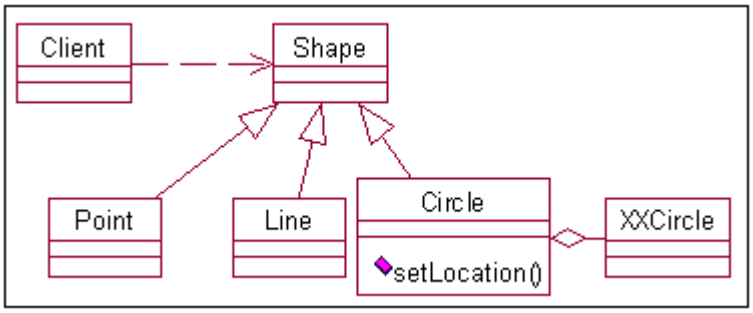


Fig 1

In figura 1 abbiamo vari tipi di incapsulamento e di strati di incapsulamento:

- Incapsulamento di dati: Point, Line e Circle sono nascosti;
- Incapsulamento di metodi: ad esempio il setLocation di Circle;
- Incapsulamento di sottoclassi: il Client di Shape non vede Point, Line, Circle;
- Incapsulamento di altri oggetti: Circle è l'unico consapevole di XXCircle.

Ma questa nuova definizione che vantaggio mi dà? Mi dà il vantaggio di splittare (decomporre!!) in modo migliore i miei programmi: gli "strati di incapsulamento" diventano degli isolanti!.

Difatti se incapsulo diversi tipi di Shape (Punti, Linee, etc) posso, poi, aggiungere in futuro altri tipi Shape, senza che il client debba subire modifiche.

Se incapsulo XXCircle, dietro Circle, posso in futuro anche cambiare la sua implementazione senza influenzare il resto. Né tantomeno Shape.

L'Ereditarietà è solo una specializzazione o è usata per il riuso?

L'ereditarietà veniva classicamente usata per la specializzazione ed il riuso; mentre oggi si può usare anche come un metodo di classificazione di oggetti. L'esempio precedente mostra tale discorso. Shape è un modo per classificare (o astrarre) Point, Line, Circle.

Il metodo di base dei Design Pattern

Da quanto visto la nuova "affermazione forte" dell'Object Oriented è: *"Cerca cosa varia e incapsulalo!!"*.

In "Design Patterns. Element of reusable Object Oriented Software" di Gamma si afferma: "Considera cosa può variare nel tuo disegno".

Questo è un approccio diverso dal dire "quali sono le cause che mi hanno condotto a ridisegnare?". In altri termini, anzicchè focalizzarmi sulle forze che mi hanno costretto a ridisegnare successivamente, cerco, invece in anticipo, di "essere capace di cambiare senza ridisegnare". E la possibilità per farlo è di incapsulare ciò che può variare! Andando oltre, viene come conseguenza l'affermazione:" Cerchiamo ciò che varia e incapsuliamolo".

L'incapsulamento va pensato, quindi, come possibilità di nascondere classi usando classi astratte (astrazione o classificazione) ed usando la composizione con un riferimento ad una classe astratta che nascondi le variazioni.

La maggior parte dei Design Pattern usano l'incapsulamento per creare layer tra oggetti, dando la possibilità ai progettisti di cambiare cose sui differenti lati del layer senza influenzare l'altro lato. Questo promuove difatti un lasco accoppiamento tra i lati.

Una superclasse **può essere astratta** *solo se* il partizionamento o classificazione delle sottoclassi è **{disjoint, complete}**. Se non è del tipo {disjoint, complete} non può essere resa astratta, altrimenti al sorgere di nuovi tipi da trattare l'unico modo di gestirli è modificare il software aggiungendo una nuova sottoclasse.

Un partizionamento è completo se non si può partizionare ulteriormente le sottoclassi. E' disjoint quando un elemento di un partizionamento non può appartenere anche all'altro.

Ulteriormente un partizionamento può essere anche overlapping, dove con overlapping si può intendere che un elemento di una partizione appartiene anche all'altra. Altra caratteristica è **dynamic** che è opposta a **static**. Un partizionamento può essere dinamico se passa da un partizionamento all'altro nel tempo, altrimenti è statico.

Esempio di partizionamento {overlapping, complete} : Superclasse Animale, sottoclassi Erbivoro e Carnivoro.

Esempio di partizionamento {disjoint, incomplete} : Superclasse VeicoloAMotore, sottoclassi Aereo, Auto, Camion.

Esempio di partizionamento {disjoint, complete, dynamic} : Superclasse Dipendente, sottoclassi Manager, Impiegato.

Un'applicazione del metodo di base

Ho un oggetto Animale. Supponiamo che i miei animali possono avere diverso numero di zampe ed essi possono volare o camminare.

Una prima soluzione per gestire la variazione del numero di zampe è: "metto una variabile su cui posso fare i metodi set e get, per settare e leggere il numero di zampe".

Variazione dati e variazione di comportamento

Contenere la variazione nei dati è sufficiente? No. Questa non è, quindi, la soluzione giusta. Dobbiamo anche pensare a gestire una variazione di comportamento.

Una seconda soluzione è di avere:

- Una classe base Animale con un attributo che mi dice il numero di zampe;
- Avere due tipi di Animale, derivati da Animale, uno col metodo walk(), l'altro con il metodo fly().

Tale soluzione ha il difetto che devo gestire due sottoclassi ma non posso gestire animali che possono sia volare che camminare!! Devo gestire la variazione di comportamento con l'incapsulamento. Ma come?

Una soluzione che mi dà la possibilità di farlo è quella in figura 2.

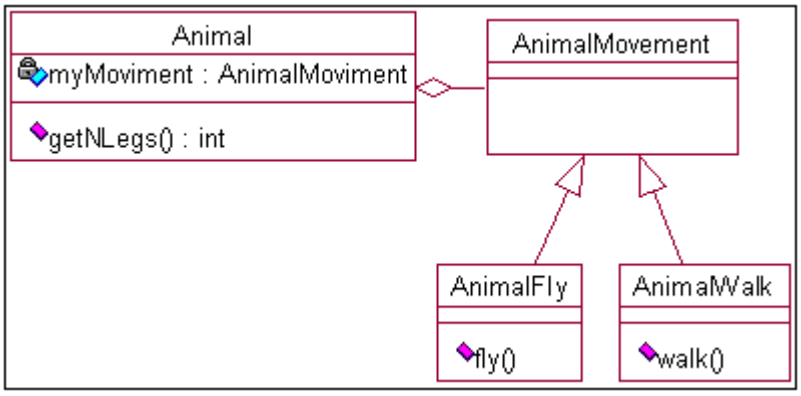


Fig 2

La soluzione mi permette dei riferimenti ad un AnimalMovement che può sia volare che camminare; mentre la variazione del numero di zampe è incapsulato nei dati.

Conclusione sul metodo alla base dei Design Pattern

Nell'esempio precedente era il problema stesso ad esprimere esplicitamente le variazioni. Quasi sempre occorre essere sensibili a capire cosa sicuramente possa variare, essere aggiunto etc.

In tal caso occorre tener presente la **regola:"Cerca le variazioni e incapsulale"**. In aiuto alla regola vengono l'incapsulamento a strati, l'astrazione (ereditarietà come classificazione) e la composizione.

E' proprio in questo modo che ci si attrezza ad accettare la variazione dei requisiti, in modo efficace ed efficiente ma soprattutto "senza ridisegnare".

L'affermazione "senza ridisegnare" sta ad indicare semplicemente la possibilità di "non stravolgere l'architettura di un gruppo di classi rispetto alla iniziale modellazione, cioè prima del nuovo requisito".

E' proprio questo è il concetto forte alla base del modellazione e del design OO, indipendentemente dalla successiva nascita dei Design Pattern.

Capitolo 2 - Perché studiare i Design Pattern?

L'idea dei pattern è nata negli anni '70, nel campo dell'Architettura e dell'Ingegneria civile (Alexander C. "A Pattern Language: Towns/Buildings/Construction" Oxford University Press, 1977). Una definizione di essi è che I pattern sono "modelli concettuali" progettuali, già collaudati in progetti reali, ricorrenti in più domini di problemi, e che possono essere riusati nella fasi di progettazione".

In realtà sono molte le motivazioni che ci portano allo studio dei Design Pattern:

- Permettono il riuso di soluzioni
  Si adottano buone soluzioni ottenute dall'esperienza di altri. Soprattutto non reinvento la ruota per i problemi ricorrenti. Una persona non molto esperta riusa l'esperienza di tecnici esperti. Si potrebbe affermare ingegneristicamente che è "il tentativo di ricondursi ad un caso noto!".

- Consentono un processo ripetibile
  Facendo uso di pattern, la modellazione diventa un processo ripetibile da più persone in un team.

- Costituiscono un vocabolario tecnico condiviso
  La comunicazione nel team è facilitata perché si parla di soluzioni note che entrano a far parte del vocabolario comune. Far riferimento a dei Design Pattern catalogati semplifica la descrizione di una soluzione nei discorsi di analisti e disegnatori. I Pattern nella comunicazione ci permettono di parlare ad un più alto livello concettuale, ci si riferisce ad un nome per indicare anche soluzioni molto complesse.

- Permettono una maggiore modificabilità e flessibilità del software
  Sono delle soluzioni testate nel tempo. Si sono evoluti in strutture facilmente modificabili rispetto a soluzioni che inizialmente vengono in mente.

- Illustrano i principi base dell'Object Oriented
  I Design Pattern sono ben fondati sui principi dell'OO: incapsulamento, ereditarietà, polimorfismo.

- Adottano strategie collaudate
  I Design Pattern si basano sul Disegno per interfacce (astrazione), la Composizione rispetto all'ereditarietà, sul principio "Cercare cosa varia per incapsularlo"

  Seguire, quindi, dei Design Pattern definiti su un catalogo significa riuscire a rispettare tutti i canoni teorici e pratici, le "Best Practices", che si possono iniettare nella realizzazione di software di buona qualità.

Ovviamente occorre tenere sempre ben presente, di ogni Design Pattern, i vantaggi e gli svantaggi e capirne il contesto di applicabilità.

La trattazione del testo, quindi, non è centrata su come individuare i Pattern catalogati, sebbene ne mostra il metodo, né segue il rigore formale di un catalogo. Questo lo si può trovare su un testo-catalogo più autorevole come quello di Gamma, Vlissides, Helm e Johnson (GoF = Gang of Four).

Il testo vuole, invece, mettere in risalto tutte le tecniche, che sono poi alla base dei Design Pattern, e che vanno usate per la ricerca della soluzione al proprio problema e nel tentativo di dare la massima flessibilità alla modellazione.

Modalità di classificazione dei pattern

I Pattern possono essere classificati in vario modo:
- Secondo lo scopo del pattern
- Secondo la fase del ciclo di produzione
- Secondo il dominio di business

Scopo del Pattern	Autore
Creazionali	[GoF95]
Strutturali	[GoF95]
Comportamentali	[GoF95]
Fondamentali	[Grand98] (Pattern in Java)
Partizionamento	[Grand98]
Concorrenza	[Grand98]
GRASP	[Larman98] (Applying UML and Patterns]

Fase del ciclo	Scopo
Analysis Pattern (M. Fowler)	Rappresentano i concetti chiave della parte business; tra cui ad esempio il pattern Inventor's Paradox
Architectural Pattern	Identificano sotto-sistemi, componenti, le loro responsabilità, ruoli etc
Design Pattern	identificano strutture ricorrenti in molti problemi
Idioms	Identificano patterns legati al linguaggio usato

Una classificazione diversa dei pattern è proposta da *Steven Johm Mester*.

Scopo del Pattern	Pattern
Pattern di interfaccia	Adapter, Facade, Composite, Bridge
Pattern di responsabilità	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
Pattern di costruzione	Builder, Factory Method, Abstract Factory, Prototipe, Memento
Pattern di operazione	Template Method, State, Strategy, Command, Interpreter
Pattern di estensione	Decorator, Iterator, Visitor

Perché fare il riuso dei pattern

Una delle domande che solitamente ci si pone nella costruzione di un nuovo sistema è: "*Che cosa possiamo riusare*?".

In generale è possibile riusare:
- l'architettura;
- il software implementato;
- il disegno;
- la documentazione;
- i test

Altra domanda: "*Perché conviene riusare*?". La risposta è:
- risparmiare tempo e avere un ROI maggiore;
- incrementare l'affidabilità del sistema;
- ridurre il "market cycle time".

Altra domanda: "*Quando i componenti sono considerabili riusabili*?". Le condizioni sono:
- devono essere sufficientemente generali;
- documentati appropriatamente;
- adeguatamente testati (meglio se in esercizio).

Altra domanda: "*Quando costruire componenti riusabili*?". La risposta è una combinazione delle seguenti:
- alla fine di un progetto;
- durante un progetto;
- permettere lo sviluppo sia di rapidi prototipi e dei componenti finali;
- avere un team di refactoring per rendere un lavoro riusabile.

L'uso combinato di patterns e frameworks comporta:
- aumento della qualità progettuale;
- riduzione della dimensione del codice e della complessità dei problemi;
- riduzione di possibili errori progettuali e architetturali;
- aumento della velocità produttiva.

Modalità di descrizione di un pattern

Secondo un metodo rigoroso da catalogo, un design pattern ha la sua "Design Pattern Description" che consiste in:

- Nome
- Intento
- Motivazione
- Conseguenze
- Esempi
- Patterns correlati

I pattern description sono, generalmente, indipendenti dal linguaggio utilizzato.

Metodologia di creazione di un pattern

L'attività di Ricerca dei Patterns (Pattern Mining) è suddivisa in almeno due fasi:

- Fase 1: Mining (Fig. 3);
- Fase 2: Polishing (Fig. 4);

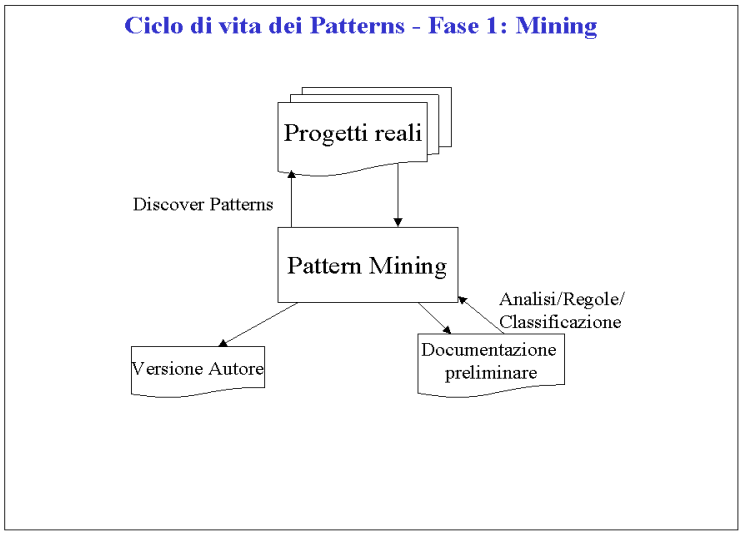


Fig. 3

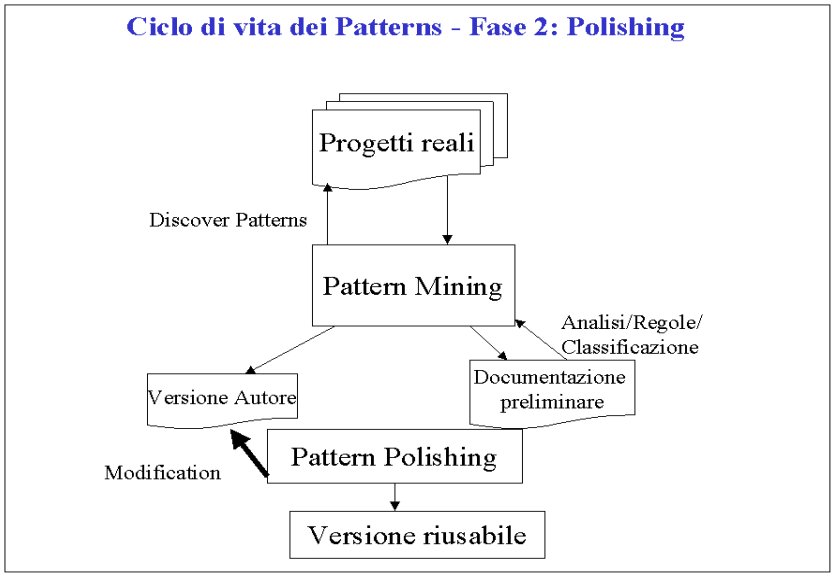


Fig. 4

Selezione di un pattern da catalogo

Ci sono differenti approcci per poter trovare il pattern adeguato alle proprie esigenze. Una modalità da me consigliata è:

- Usare un catalogo di pattern e consultare la parte "Intento", per verificare se il pattern è rilevante per il proprio problema;
- Capire come i diversi pattern interagiscono guardando le relazioni UML e selezionare quelli giusti;
- Capire il pattern anche attraverso la sua classificazione;
- Capire cosa deve essere variabile nel proprio progetto. In tal modo si va alla ricerca dei gradi di libertà del programma. Tale punto è ancora più forte nel caso di un framework.

Capitolo 3 - Best Practices Object Oriented

Principi "Object Oriented"

Open Closed Principle (OCP)

"Un modulo deve essere aperto all'estensione e chiuso alle modifiche".

Il principio OCP è tipico dell'Object Oriented: un nostro modulo deve poter essere esteso per ereditarietà senza che sia necessario modificarlo internamente. La chiave dell'OCP è l'*astrazione object oriented*.

Un esempio C++.

Supponiamo di avere del software di Logon che, purtroppo, richiede di essere modificato ogni volta a causa dell'aggiunta di un nuovo tipo di modem.

```
struct Modem {
        typedef enum Type { hayes, courrier, ernie } type;
};

struct Hayes {
        Modem::Type type;
        // etc
```

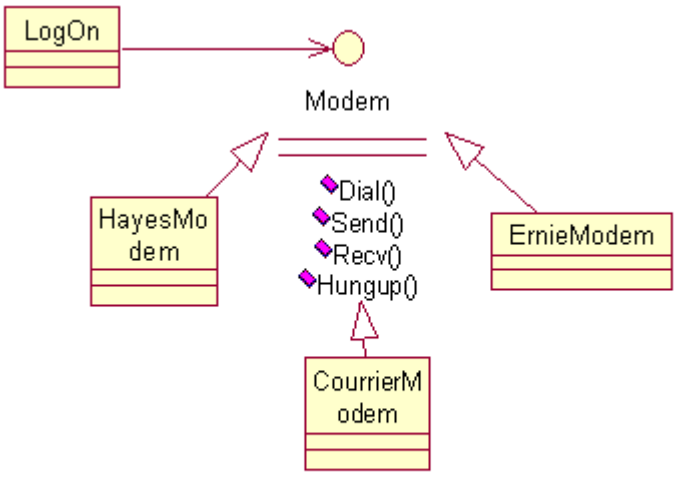
```
};

struct Courrier {
        Modem::Type type;
        // etc
};

struct Ernie {
        Modem::Type type;
        // etc
};

void Logon( Modem& m, string& pno, string& user, string& password){
        if( m.type == Modem::hayes)
           DialHayes((Hayes&)m, pno);
        else if(m.type == Modem::courrier)
           DialCourrier((Courrier&)m, pno);
        else if(m.type == Modem::ernie)
           DialErnie((Ernie&)m,pno);
}
```

Qui l'OCP aiuta a capire che sin dall'inizio occorreva progettare il tutto secondo il diagramma di seguito.



La soluzione sfrutta il "**Polimorfismo dinamico**"; la cui implementazione sarebbe:

```
class Modem{
  public:
        virtual void Dial( const string& pno) = 0;
        virtual void Send( char ) = 0;
        virtual char Recv() = 0;
        virtual void Hungup() = 0;
}
```

Il metodo della classe Logon si ridurrà a:

```
void Logon( Modem& m, string& usr, string& pwd){
  m.Dial(pno);
}
```

Esiste in C++ anche una soluzione di "**Polimorfismo statico**", tramite i template:

```
template <typename MODEM>
void Logon(MODEM& m, string& pno, string& usr, string& pwd){
  m.Dial(pno);
}
```

E' preferibile la prima soluzione, che da un maggior controllo sui tipi in gioco.

L'ereditarietà permette di aggiungere altri tipi di modem, l'astrazione permette di mettere insieme la classe del Logon e l'interface Modem, ai fini della compilazione, senza più modificare il metodo Logon ma aggiungendo solo nuovo codice.

Liskov Substitution Principle (LSP)

"Ogni sottoclasse dovrebbe essere sostituibile con la classe base".

Il principio LSP coinvolge anche il principio **Design by Contract (DBC).**

Il principio DBC insiste sul fatto che pre-condizioni e post-condizioni devono essere rispettate dal disegno per contratto. Il contratto è fra chi ha implementato le classi e chi le usa!

Vediamo perché. Di per sé un programmatore sa che il principio LSP è facile a realizzarsi.

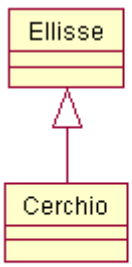
Es:

```
void User(Base&);
Derived d ;
User(d) ;
```

Ma è sempre possibile farlo? Esiste sempre un modo corretto per ottenerlo?

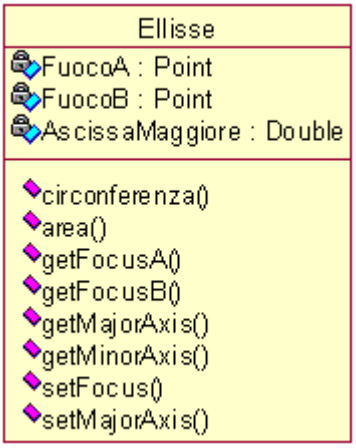
Vediamo il famoso **dilemma Ellisse/Cerchio**.

In matematica ricordiamo tutti che un cerchio è un caso particolare di ellisse. Per cui concettualmente vale il diagramma di figura.



Ma è corretto?

Pensiamo agli attributi ed ai metodi di Ellisse. Vediamo la figura successiva.



La circonferenza erediterebbe un bel po' di cose inutili!

Ma soprattutto una circonferenza è caratterizzata solo dal centro del cerchio e dalla maggiore ascissa (il raggio).

Anche se i due fuochi si fanno coincidere come centro del cerchio, forse qualche problema rimane.

Una prima implementazione "difettosa" di metodo sarebbe la seguente:

```
void Cerchio::setFocus( const Point& a, const Point& b){
    itsFocusA = a;
    itsFocusB = a;
}
```

Apparentemente uso un solo punto, ma se applico il DBC scopriamo che c'è un errore tale che il principio LCP non è rispettato:

```
void f( Ellisse& e ){
   Point a (-1, 0);
   Point b (1,0) ;
   e.setfocus(a,b) ;
   e.setMajorAxis(3);
   assert(e.getFocusA() == a);
   assert(e.getFocusB() == b);
   assert(e.getMajorAxis() == 3);
}
```

Se nell'esempio al metodo f passiamo Cerchio anzicchè Ellisse, la seconda assert non è verificata (DBC) e, quindi, l'LCP è violato!!

Per far sì che l'LCP sia valido, ovvero che sia possibile la sostituibilità, occorre che il contratto delle classe base sia onorato anche dalle classi derivate.

Le violazioni LCP sono una delle cause di errori che vengono rivelate troppo tardi, quando si è costruito molto software ed il costo per porvi riparo potrebbe essere elevato.

L'unico modo per evitare un tale errore è di segnalare in fase di disegno le pre-condizioni e le post-condizioni e poi farle implementare con delle assert, nei metodi a cui vengono passati parametri.

Una "pezza" implementativa per arginare il danno di una violazione del LCP è di inserire, poi, degli if-else nel metodo per testare che l'oggetto di input sia effettivamente la classe attesa, altrimenti viene alzata un'eccezione.

Per cui una violazione LCP conduce, come pezza, ad una latente violazione OCP.

Dependency Inversion Principle (DIP)

"Dipendere dalle astrazioni e non da concetti concreti".

L'OCP è un obiettivo architetturale dell'Object Oriented, il DIP è il concetto base che consente l'OCP.

Ci sono alcune considerazioni da fare. Usando tale tecnica è possibile anche creare degli "hot-spot" o degli "hinge-points" che permettono di agganciare altro software (rispetto dell'OCP); quindi si possono creare dei framework.

Se usiamo una interfaccia e, poi, deriviamo le altre classi, non possiamo istanziare un'interfaccia. Sembrerebbe che questo ci metta KO definitivamente!

Fortunatamente i Design Pattern ci forniscono molti meccanismi tra cui la ABSTRACT FACTORY o il FACTORY METHOD, etc.

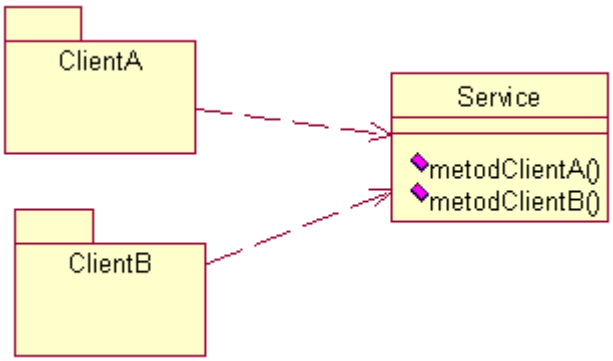
Per cui astrarre è possibile ed utile, ma poi dobbiamo usare i Design Pattern.

Il DIP ci suggerisce, cioè, che un sano Design Refactoring che porti a vantaggi di stabilità, flessibilità e riusabilità passa inevitabilmente attraverso l'uso e la scelta di Design Pattern.

Interface Segregation Principle (ISP)

"Molte interfacce, una per client, è meglio di avere un'unica interfaccia general purpose per tutti i client".

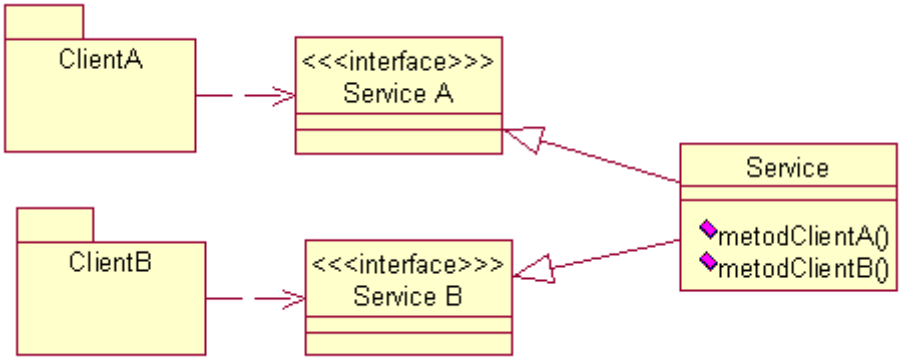
Nella figura successiva si vedono due client che dipendono dai metodi di un'unica classe Service o interfaccia che li serve tutti.



Una tale situazione è da evitare.

Se un cambiamento è fatto ad un metodo del Client A, ci può essere un'influenza sul Client B.

La soluzione al problema è la figura successiva, che rispetta l'ISP.



Principi di "Package Architecture"

Finora abbiamo visto principi generali dell'object oriented, che insistono sull'astrazione. Nell'organizzazione di un'architettura le classi da sole non sono sufficienti. Esistono i package, i pacchetti applicativi riusabili. Per i package esistono tre principi architetturali.

Release Reuse Equivalency Principle (REP)

"La granularità del riuso è la granularità della release"

In generale un componente, una classe, un package, un applicativo è sottoposto ad uno strumento di versioning del software e di tracciamento anomalie.

Se un cliente usasse un componente che continuamente un autore modifica e non vi fosse versionamento e garanzia del supporto della versione di componente precedente, allora nessun cliente accetterebbe di usare tale componente. Per cliente s'intende sia un cliente interno un cliente esterno.

Questo vuol dire che un cliente usa (o riusa) il componente all'interno di una release di prodotto; per cui la granularità del riuso è pari a quella della release del prodotto o del gruppo classi in un package.

Common Closure Principle (CCP)

"Le classi che cambiano insieme, vivono insieme"

I grandi progetti hanno una "rete di package" e la loro manutenzione, test, gestione non è banale.

Più alto è il numero di classi e package da cambiare più è alto il lavoro da fare per fare la rebuild ed i test.

Un'idea potrebbe essere di raggruppare insieme in un package le classi che pensiamo cambino insieme. Questo porta a ridurre l'impatto del cambiamento nel sistema.

Alcune considerazioni sul CCP.

Questa tecnica però va ponderata attentamente, perché potrebbe essere a discapito della logica di mettere insieme le classi correlate. Va bene, invece, quando la cosa riguarda comunque classi correlate logicamente.

Tale tecnica è orientata soprattutto a chi fa manutenzione.

Potrebbe essere una tecnica adottata momentaneamente, cioè nella fase di movimentazione del prodotto; mentre nella fase matura del prodotto si riportano le classi in package diversi usando un Implementation Refactoring.

Common Reuse Principle (CRP)

"Classi non riusate insieme non dovrebbero essere raggruppate insieme"

Ci è capitato nella vita di fare sicuramente un upgrade di sistema operativo e poi, a volte, un applicativo di nostro interesse non funzionava più AS IS; ma occorreva fare un "porting".

Ebbene se fossero raccolte insieme delle classi che non sono riusate insieme potrebbe succedere che facendo un cambiamento a parte di esse devo fare uno sforzo a testarle tutte e riverificarle tutte anche per la parte che non è d'interesse per il funzionamento di un certo applicativo.

Va evitata una situazione del genere.

I tre principi REP, CCP, CRP sono mutuamente esclusivi: non possono essere soddisfatti contemporaneamente, perché diretti a figure professionali diverse. REP e CRP interessano a chi fa riuso, CCP interessa a chi fa manutenzione.

Il CCP tende a fare i package quanto più grandi è possibile, almeno fino a comprendere tutte le classi che variano insieme. Il CRP tende a fare package piccoli.

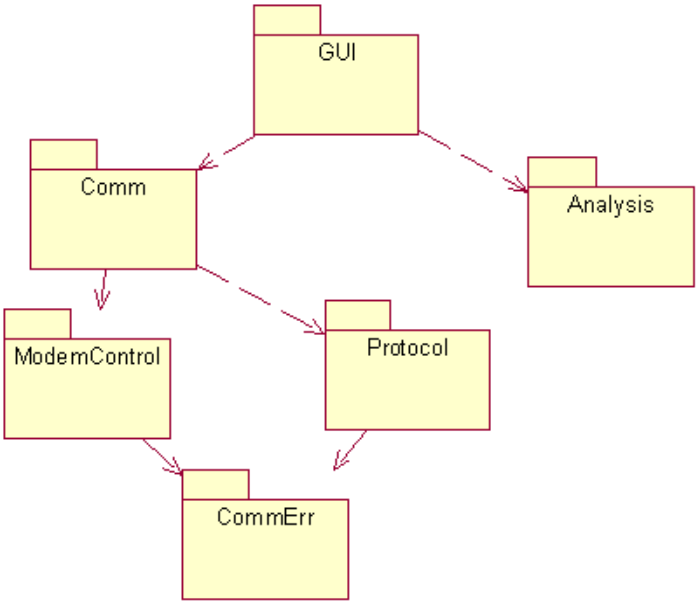
I tre principi interessano solo l'Implementation Refactoring.

Principi di "Package Coupling"

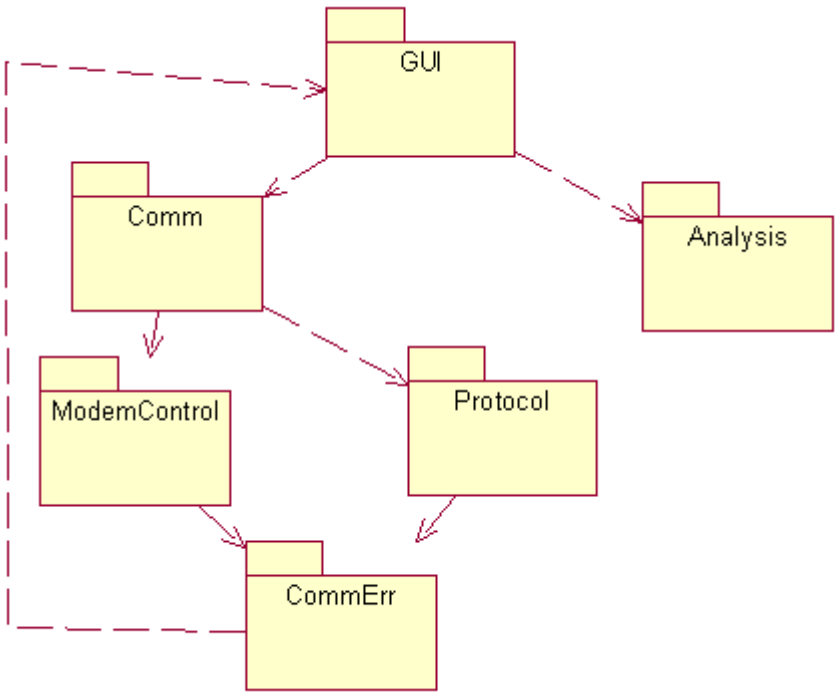
Acyclic Dependencies Principle (ADP)

"Le dipendenze tra packages non devono essere cicliche"

Vediamo il caso della figura successiva per comprendere il principio.



Supponiamo che per inviare un errore a video dal package CommErr decido di chiamare un oggetto della GUI. In questo caso le dipendenze dei package si trasformano come quelle in figura successiva.



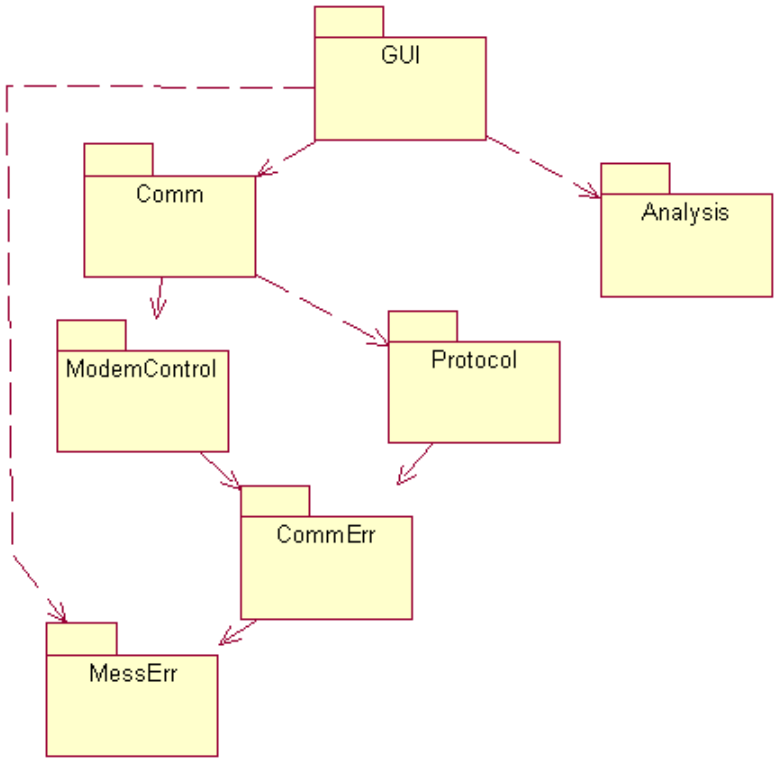
In questo caso si è introdotto un ciclo. Ma soprattutto un extra-lavoro! Infatti chi sta lavorando con Protocol per ricostruire la sua test suite ha bisogno di:

- CommErr
- GUI
- ModemControl
- Analysis

Mentre nella figura iniziale chi lavorava su Protocol avrebbe avuto bisogno solo di CommErr!!!

Le soluzioni sono:

- introdurre un altro package e cambiare il verso della dipendenza, per spezzare il ciclo (figura successiva);
- usare il DIP e l'ISP



Stable Dependencies Principle (SDP)

"Dipendere nella direzione della stabilità"

Il concetto di stabilità non ha niente a che vedere con la frequenza dei cambiamenti.

Un package X da cui dipendono altri package è detto *stabile*, perché un suo cambiamento influenza i package sottostanti. O dicendola in altro modo X ha dei buoni motivi per non cambiare. Il package X è detto anche *indipendente* ma una sua eventuale variazione influenza gli altri package.

Un package Y che dipende da tre package sopra è alquanto *instabile*.

Per capire la stabilità possono essere introdotte delle metriche semplici.

Metrica: Stabilità

Indichiamo con:

C_a incomming dependencies
C_e outgoing dependencies
I Instability

$$\mathbf{I = C_e/(C_a+C_e)}$$

L'instabilità ha un range [0,1].

Se non vi sono dipendenze uscenti ($C_e = 0$) allora I = 0.
Se non vi sono dipendenze entranti ($C_a = 0$) allora I = 1.

Riformulando SDP conviene *"Dipendere sopra package dove I è più basso del proprio"*

Ma flessibilità e stabilità vanno d'accordo? In realtà no in termini di dipendenza, sì in termini di astrazione. Vediamo perché.

Supponiamo che un package Y sia flessibile ai cambiamenti e dipenda da X; mentre X dipenda da A, B, C.

Sebbene tra Y ed X abbiamo rispettato l'SDP, una variazione di Y porta cambiamenti in X, ma anche in A,B,C. Quindi c'è una violazione dell'SDP!!

La flessibilità difatti non va ottenuta attraverso le dipendenze ma sfruttando l'astrazione come vedremo nel prossimo principio.

Stable Abstraction Principle (SAP)

"I Pacchetti stabili devono essere astratti".

E' una conclusione a cui siamo arrivati dimostrando una violazione del SDP e che discende dall'OCP.

Metrica: Astrattezza

Indichiamo con:

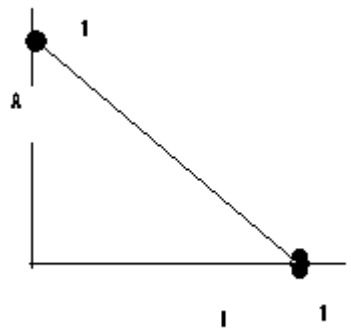
Nc numero di classi nel package
Na numero di classi astratte nel package
A astrattezza

A = Na/Nc

La metrica A ha un range [0,1].

A=0 quando non ci sono classi astratte.
A=1 quando ogni classe del package è una classe astratta.

Il grafico I-A



Dal grafico Instabilità-Astrattezza di figura si vede che al diminuire di A aumenta I. Al diminuire di I, invece A aumenta.

Nel grafico I-A il punto in alto a destra indica la massima astrattezza e nessuna dipendenza entrante (A=1, I=1) ed è una zona inutile.

Il punto in basso a sinistra è di nessuna astrattezza (package concreti) e vi sono dipendenze entranti (A=0, I=0). E' il caso peggiore per un package.

E' importante, quindi, massimizzare la distanza tra le due zone (A=1, I=1) e (A=0, I=0) con la linea chiamata *Main sequence*

Metrica: Distanza

Se conosciamo A ed I del nostro package possiamo sapere quanto il nostro package è lontano dalla Main Sequence:

D = | A+I-1|/√2 con range [0, 0.707].

D' è la distanza normalizzata = | A + I − 1|.

D' = 0 significa che il package è sulla Main Sequence.

Metriche di un modello

Abbiamo precedentemente individuate alcune metriche:
- Stabilità;
- Astrattezza;
- Distanza.

Questo significa che dato un modello è possibile valutarne tali metriche.

Capitolo 4 – GRASP Pattern

GRASP è una sigla dovuta a Craig Larman, il cui significato è: *General Responsabilities Assignment Software Pattern*s.

La caratteristica di questi patterns è quella di aiutare a capire in che modo ed a quale classe assegnare delle responsabilità e, quindi, racchiudono concetti Object Oriented di base; ed è per questo motivo che i patterns GRASP sono riconosciuti come propedeutici ai Design Pattern maggiori.

L'utilizzo dei GRASP Pattern permette di:
- Aumentare la qualità del software (metriche: correttezza, robustezza, coesione, accoppiamento, etc);
- Validare la correttezza dell'analisi/disegno UML;

I pattern GRASP più importanti sono:
- Expert [Larman98];
- Creator [Larman98];
- High Cohesion [Larman98];
- Low Coupling [Larman98];
- Controller [Larma98];
- Polymorphism [Larman98];
- Pure Fabrication [Larman98];
- Indirection [Larman98];
- Don't talk to strangers (Principio di Demetrio) [Lieberherr88]

Expert

Intento

Assegnare la responsabilità alla giusta classe.

Motivazione

In un progetto con un numero elevato di classi spesso è difficile decidere a quale classe assegnare la responsabilità. Il principio guida è di "assegnare la responsabilità all'esperto, ovvero a chi possiede tutte le informazioni per svolgere le operazioni che servono". Per soddisfare il principio è necessario suddividere la responsabilità attraverso una semplice decomposizione funzionale e poi assegnare le sotto-responsabilità alla classe che ha tutte le informazioni. Alla fine si avrà scambio di messaggi fra oggetti.

Facciamo un esempio. Supponiamo di avere una classe Vendita che contiene una lista di OggettoVenduto e una classe di SpecificaProdotto, che descrive gli oggetti venduti. Dobbiamo calcolare il totale della fattura (Fig. 5)

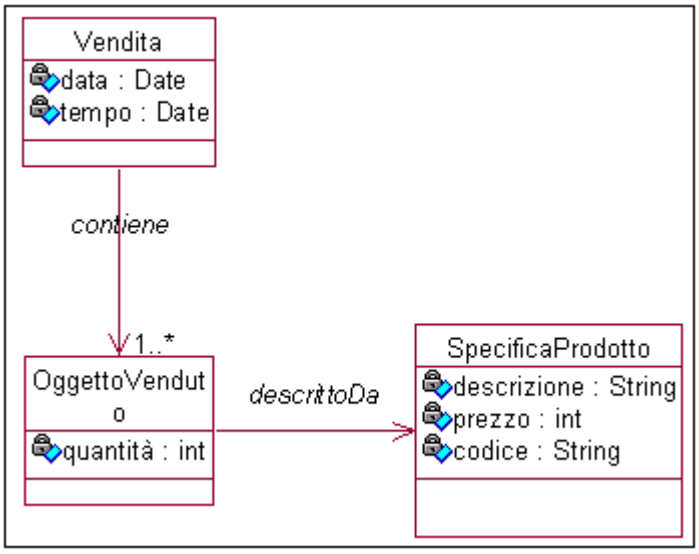


Fig. 5

Vediamo come usare la scomposizione funzionale per individuare le responsabilità (una tecnica simile a quella CRC di Kent Beck).

Per sapere quanto ho venduto m'interessano quantità e prezzo per ogni prodotto, per cui come sotto-responsabilità l'expert è OggettoVenduto, che deve inviare un messaggio a SpecificaProdotto; mentre per emettere la fattura in data e ora per il venduto la responsabilità (l'expert totale) è di Vendite.

Classe	Responsabilità
Vendita	Conosce il totale della fattura
OggettoVenduto	Conosce il subtotale di ogni item venduto
SpecificaProdotto	Conosce il prezzo di ogni prodotto

A questo punto (Fig. 6) si aggiungono i metodi necessari alla Fig. 5.

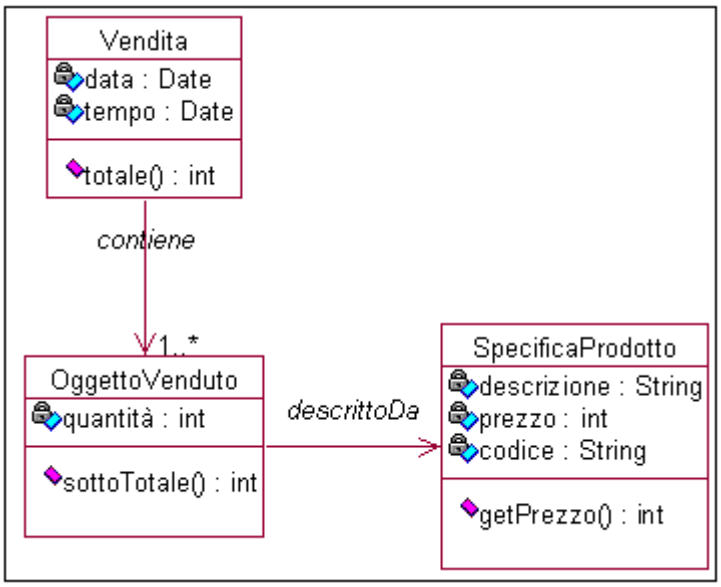


Fig. 6

Conseguenze

Viene mantenuto l'incapsulamento. L'accoppiamento è il minimo indispensabile. Le classi sono coesive, ovvero fanno solo quello che serve.

Pattern correlati

Il pattern Expert è insito in tutti i pattern maggiori GoF.

Creator

Intento

Assegnare alla giusta classe la responsabilità di creare istanze di altre classi.

Motivazione

Trovare un criterio per individuare quale classe possa avere la responsabilità di creare una o più classi.

Il criterio è: "Si assegna alla classe X la responsabilità di creare la classe Y, se sono verificate una delle seguenti affermazioni:

- X contiene oggetti di Y;
- X aggrega oggetti di Y;
- X registra istanze di oggetti di classe Y;
- X possiede tutte le informazioni (è l'expert) per creare Y".

Se si dovessero verificare molte di queste affermazioni probabilmente dovreste verificare se sia il caso di fare classi innestate (in Java le classi Inner). Riprendiamo la Fig. 5. In essa si vede che esiste un'associazione tra Vendita e OggettoVenduto. Probabilmente Vendita contiene diversi oggetti di classe OggettoVenduto; per cui il pattern Creator suggerisce che Vendita debba averne la responsabilità di creazione, ovvero un metodo che faccia questo makeOggettiVenduti() che richiama il costruttore di OggettoVenduto (Fig. 7)

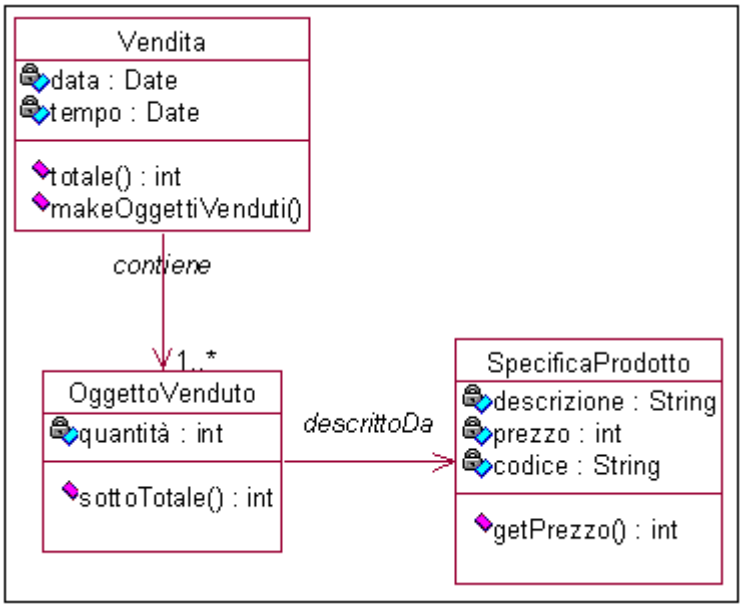


Fig. 7

Conseguenze

L'accoppiamento, sintomo di interdipendenza, non subisce stravolgimenti, perché la relazione tra le due classi era già esistente e Vendite ne è anche l'expert.

High Cohesion

Intento

Fare in modo che un modulo (metodo, oggetto, classe, package) sia concentrato su un unico obiettivo.

Motivazione

La coesione è un'altra metrica. Essa misura quanto le responsabilità di un modulo siano correlate fra loro (metodi che fanno troppe cose ad esempio).

E' meglio avere moduli che facciano un'unica cosa (metodi query o set, una classe con precise responsabilità, un package con determinati compiti etc). Una classe oberata di lavoro ha scarsa coesione, è difficilmente riusabile, scarsamente comprendibile, scarsamente manutenibile.

Il concetto di scarsa coesione è, quindi, valutabile solo attraverso dei campanelli d'allarme dimensionali, di nomenclatura, di numerosità dei parametri:

- Per i metodi la scarsa coesione potrebbe essere segnalata dal il numero alto di linee di codice, se superano le 100 loc (line of code);
- Per le classi la scarsa coesione potrebbe essere segnalata da un numero basso di metodi;
- Per i metodi e le classi la scarsa coesione potrebbe essere segnalata dalla difficoltà di trovare un nome ad essa (fa tante cose);
- Per il passaggio dei parametri siamo di fronte ad una scarsa coesione se il metodo si comporta in modo diverso a seconda del valore del parametro. Un modulo non dovrebbe demandare il suo comportamento ad un parametro. Ma quest'ultima cosa va ponderata anche a fronte di altre cose.

Conseguenze

Il pattern permette di individuare:

- Bassa coesione;
- Supporta il riuso e l'estensibilità;
- Incentiva alla comprensibilità del codice.

Pattern correlati

Low Coupling
Expert

Low Coupling

Intento

Aumentare il riuso e l'estensibilità dei "moduli".

Motivazione

L'accoppiamento è una metrica che indica la dipendenza tra metodi, oggetti, classi e packages.

Essere di fronte ad un alto accoppiamento è uno svantaggio: una modifica di un modulo può comportare una forte manutenzione e diminuisce la riusabilità e l'estensibilità.

Per comprendere quest'aspetto dovremmo esaminare due cose:
- L'accoppiamento tra metodi;
- L'accoppiamento tra moduli (classi, oggetti, packages).

ACCOPPIAMENTO TRA METODI
Per definizione "lo stato di un oggetto è la fotografia in un istante (snapshot) dei valori assunti dagli attributi dell'oggetto".

I metodi, quindi, si possono dividere in:
- Utility: non modificano lo stato dell'oggetto; es: Integer.parseInt(String), Math.cos(double); etc
- State View: ritornano lo stato dell'oggetto;
- State Change: cambiano lo stato dell'oggetto.

Occorre ancora vedere adesso come un metodo tratta l'input e l'output.

INPUT
Un metodo accetta valori da:
- parametri d'ingresso;
- attributi e metodi statici (di classe);
- attributi e metodi (d'istanza).

OUTPUT
Il metodo può:
- tornare un valore di tipo primitivo (handle);
- alterare oggetti che gli sono stati passati e che restituisce;
- alterare attributi statici direttamente o con metodi statici;
- alterare attributi direttamente o con metodi;
- lanciare eccezioni.

In base alle definizioni precedenti possiamo vedere, per ogni tipologia di metodo, come ridurre l'accoppiamento.

METODI UTILITY: hanno il minimo accoppiamento possibile; difatti un metodo utility deve essere caratterizzato da:

- INPUT: prendere solo parametri d'input e solo quelli che gli servono per produrre l'output;
- OUTPUT: ritornare un valore di tipo handle oppure alterare gli oggetti ricevuti come parametro e non richiama altri metodi.

Esempi: la classe Integer in Java.

METODI STATE VIEW: hanno un accoppiamento maggiore rispetto ai metodi utility. Per raggiungere il minimo accoppiamento possibile devono:

- INPUT: prendere i parametri d'ingresso e usare attributi e metodi propri, ad eccezione delle costanti (in Java si può definire una interfaccia di costanti);
- OUTPUT: tornare un valore, alterare parametri oppure lanciare un'eccezione.

Esempi: TextField.getText(). Questo è un caso diverso di metodo. Qui difatti il metodo lavora sullo stato dell'oggetto (la rappresentazione interna del campo editabile) sul quale viene invocato e lascia inalterato.

METODI STATE CHANGE: hanno un accoppiamento maggiore rispetto agli altri. Per raggiungere l'accoppiamento minimo possibile devono:

- INPUT: prendere parametri d'ingresso e usare attributi e metodi (di classe o di istanza) interni alla classe, ad eccezione delle costanti;
- OUTPUT: tornare un valore, alterare parametri oppure lanciare un'eccezione o assumere un altro stato (il valore ritornato come stato della classe).

ACCOPPIAMENTO TRA MODULI

Per le classi, oggetti e package è quasi simile a quanto già detto e potremmo riassumere che le forme di comune accoppiamento tra una classe X ed una classe Y è del tipo:

- La classe X è una sottoclasse della classe Y;
- La classe X implementa l'interfaccia Y;
- La classe X ha un metodo accoppiato con un metodo della classe Y;
- La classe X ha un attributo che fa riferimento ad un'istanza della classe Y o è un'istanza diretta della classe Y.

Considerazioni

Il Low Coupling incoraggia ad individuare correttamente le responsabilità e perseguire il minimo accoppiamento possibile.

Conseguenze

Supporto al riuso e alla estensibilità. Supporto alla comprensibilità del codice.

Patterns correlati

High Cohesion
Principio di Demeter
Expert

Don't talk to strangers

Il pattern è dovuto a Lieberherr. E' noto come “Non parlare con gli sconosciuti" o principio di Demeter o, come dico io, “Non bypassare l’esperto”.

Intento

Minimizzare l'accoppiamento tra classi con delle indirezioni aggiuntive.

Motivazione

Porre dei limiti agli oggetti a cui si possono inviare messaggi, per aumentare l'estensibilità del software. Il principio di Demeter aiuta ad aumentare la modularità del software. Per rispettare il pattern occorre aumentare l'indirezione. In altri termini il principio è che all'interno di un metodo si dovrebbero mandare messaggi solo ai seguenti tipi di oggetti:

- L'oggetto stesso (this);
- Un parametro del metodo;
- Un attributo di this;
- Un elemento di una collection di oggetti che è un attributo di this;
- Un oggetto creato localmente al metodo

Il nome del pattern "Non parlare con gli sconosciuti" suggerisce anche che gli oggetti che rispettano il principio di sopra sono considerati di "famiglia" mentre tutti gli altri sono "sconosciuti".

Vediamo un esempio pratico in Fig 8.

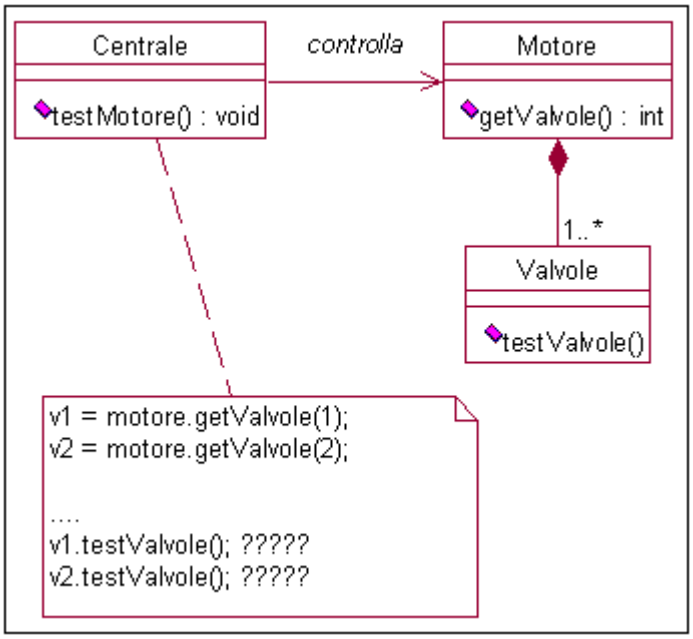


Fig. 8

Se il metodo testMotore() della classe Centrale cerca di controllare anche la classe Valvole violeremo il principio di Demeter. Ma significherebbe anche che stiamo saltando l'Esperto o che Motore non poteva essere l'Esperto! Oppure qualcosa non va!

Infatti in Fig. 8 gli oggetti v1 e v2 non rientrano nel principio visto prima. I due handle (attributi primitivi) sono sì locali ma probabilmente creati da un'altra parte (forse il costruttore di Motore) per cui non rientrano tra gli "oggetti creati localmente al metodo".

Per cui va rimodellato il tutto in modo diverso, ridistribuendo le responsabilità in modo appropriato (pattern Expert) e individuando l'expert della classe Valvole (Fig. 9).

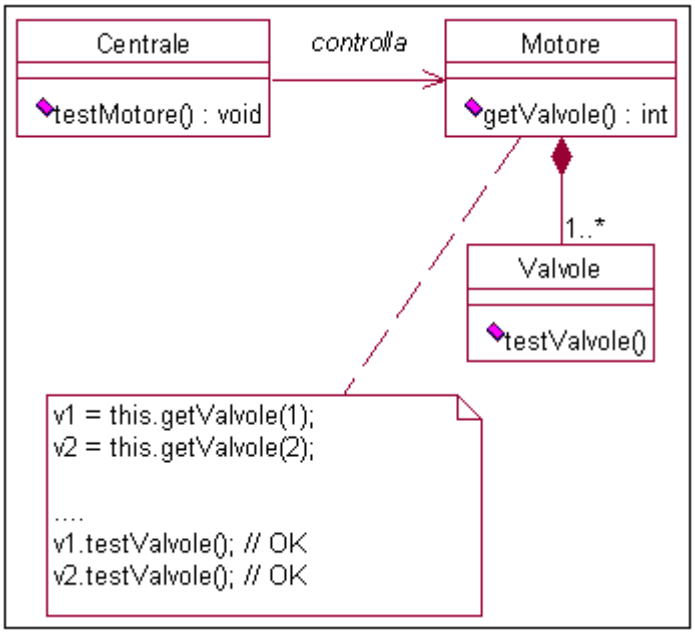


Fig. 9

Considerazioni

Spesso per rispettare tale pattern si usano dei metodi intermedi che, poi, chiamano i metodi incriminati. Questa tecnica è definita "promozione dell'interfaccia", in altri termini un metodo di una classe viene promosso nell'interfaccia dell'altra classe. Anche se questa è una soluzione, personalmente non credo che ciò migliori molte le cose. Meglio evitare, basta impiegare bene l'Expert Pattern. Anche perché si può facilmente violare il pattern High Cohesion. Ovviamente un principio, se le condizioni al contorno lo impongono, può essere violato; ma molto raramente dovrebbe essere necessario.

Conseguenze

Viene minimizzato l'accoppiamento

Sempre in ambito OO esistono diversi principi elementari che aiutano a comprendere i difetti progettuali e nel seguito le esaminiamo perché li riteniamo importanti e di valido aiuto, spesso rimedio al degrado software.

Capitolo 5 – Pattern GoF

GoF divide i pattern in tre categorie, come in tabella.

Categoria	Scopo	Pattern
Strutturali	Mettere insieme oggetti esistenti	Adapter, Facade, Bridge, Decorator
Comportamentali	Fornire una modalità per dare flessibilità al comportamento e gestire le variazioni	Strategy, Observer, Template Method, Command
Creazionali	Creare ed istanziare oggetti	Abstract Factory, Singleton, Double-Checked Locking, Factory Method.

Quando inizierete a studiarli vi verrà naturale un dubbio: “Perchè i pattern Bridge e Decorator sono strutturali? In fondo sembrano Comportamentali…”

La categoria di pattern strutturali ha lo scopo di individuare pattern che mettono insieme oggetti già esistenti come l’Adapter e il Facade.

Scopriremo che col Bridge separiamo l’astrazione dall’implementazione e che le teniamo insieme col bridge; mentre col Decorator abbiamo una classe funzionale di partenza e poi la decoriamo con funzionalità aggiuntive. Ma questi sono proprio tipici motivi strutturali.

Nel seguito non raggrupperemo i pattern secondo classificazione ma si evidenzieranno, uno dopo l’altro, tutti i concetti che sono alla base di più di uno di essi.

Facade: Strutturale

L’intento del Facade Pattern è: “fornire una interfaccia unificata ad un insieme di interfacce di un sottosistema. Il Facade definisce un più alto livello di interfaccia che rende facile l’uso del sottosistema”.

Motivi d’utilizzo

Grazie al Facade un client vede solo un unico punto di ingresso e di chiamata verso il sistema server. Il che facilita il compito di chi implementa il client.

Il Facade, da parte sua, nasconde tutte le possibili chiamate che fa verso il sottosistema e nasconde anche il numero di oggetti. Per cui il Facade nasconde la complessità del sottosistema (numero di classi e di metodi chiamati) e, quindi isola il client dal sottosistema.

In altri termini il Facade semplifica l’uso del sottosistema al client.

Il client ha una interfaccia semplificata a disposizione ed eventuali modifiche di interfacce del sottosistema sono concentrate solo nel Facade

Quando è efficace il Facade?

Quando il client deve lavorare solo con un sottoinsieme delle capacità di un sistema (quindi un sotto-sistema).

Se il client deve lavorare con tutte le classi del sistema diventa difficile creare un Facade con una interfaccia più semplice.

Esempio

Abbiamo un client:A, che deve lavorare con un oggetto Database, da cui ottenere un oggetto Models, per poi fare una query a Models per ottenere Element. Una prima soluzione è in figura 10.

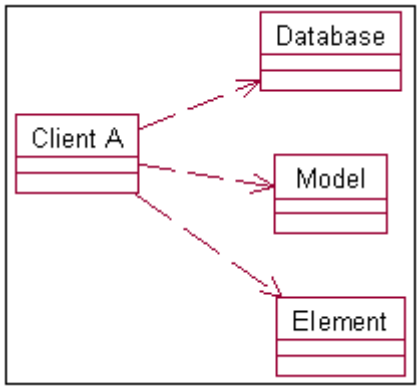


Fig. 10

Se ci si riflette meglio una migliore soluzione è mostrata in figura 11.

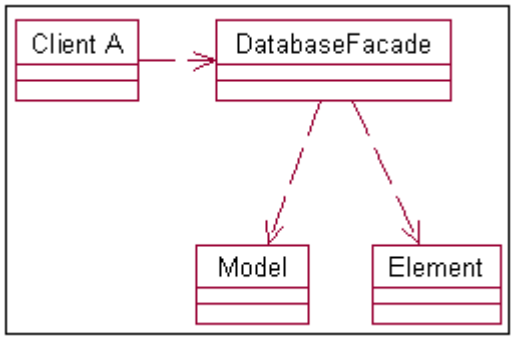


Fig. 11

Vantaggi ottenibili col Facade

La soluzione, come mostra la figura 11, riduce il numero di oggetti con cui il client deve lavorare e riduce il numero di metodi che il client deve conoscere del "sistema server".

Se dietro al Facade nascono nuovi oggetti e metodi, il client non lo saprà mai perché vede sempre le stesse interfacce, esposte dal Facade.

Concetto OO su cui si basa il Facade

Il Facade si basa sul concetto degli "strati di incapsulamento". In altri termini incapsula o nasconde il sottosistema.

Il Facade può addirittura avare tra i suoi membri privati il nome del sistema a cui sta puntando.

Quali altre ragioni ci possono essere per incapsulare il sistema?

Alcune di esse sono:

- Tracciamento dell'uso del sottosistema
- Swap di sistemi: in futuro potrebbe essere necessario puntare ad un secondo sistema, invece che al primo e può avvenire col minimo sforzo attraverso il solo Facade.

Il Facade è, quindi, un Wrapper (imballaggio, copertina, facciata). Un altro termine con cui è noto, nella terminologia J2EE SUN, è **Front-Controller**.

Il Facade è, difatti, anche un controller: nell'esempio precedente è colui che decide la giusta sequenza dei metodi del sottosistema che ha dietro.

In un sistema ovviamente possono essere presenti più Facade proprio perché il Facade semplifica l'accesso solo a parti del sistema e, quindi, possono essere presenti un Facade per sotto-sistema.

Esercizio

Un applicativo deve accettare dei dati da console, deve gestire l'input/output e fare la conversione dei tipi di dati.

Il Facade suggerisce la creazione di un oggetto che presenta una interfaccia semplificata a questa parte o sottosistema.

La figura 12 riassume come organizzare il nostro esempio.

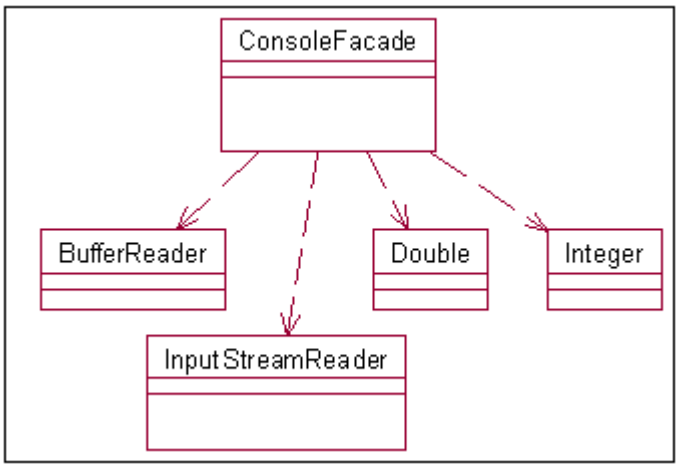


Fig. 12

Nella fig. 5 la classe ConsoleFacade conosce le interfacce (i metodi) delle classi incapsulate, cioè del "**SubSystem Class**": BufferReader, InputStreamReader, Double, Integer. Per cui ConsoleFacade *delega* alle altri classi, che incapsula, il compito di soddisfare la richiesta.

Il concetto di delega è anche una conseguenza dei Pattern GRASP di Craig Larman (Pattern elementari), cioè a chi assegnare la responsabilità di fare le cose.

Un Pattern GRASP che risolve tale punto è proprio il Pattern dell'Esperto. Il Pattern dell'Esperto punta ad una scomposizione funzionale di chi sa fare le cose: le cose vanno fatte dalla classe che sa (esperta) come farle e che ha tutte le informazioni. Ecco, quindi, che nascono le varie classi esperte al trattamento delle cose e, quindi, il SubSystem class a cui il Facade delega i compiti.

Nessuna classe del "SubSystem class" ha riferimenti al Facade ma solo il Facade verso il Subsystem class.

Adapter: Strutturale

L'intento dell'Adapter Pattern è di: "convertire l'interfaccia di una classe nell'interfaccia che un client si attende. L'Adapter consente, quindi, alle classi di poter lavorare insieme quando esse hanno interfacce incompatibili".

Esempio

Esaminiamo il seguente problema: Creare una classe in grado di mostrare (display) forme di qualsiasi genere.

Supponiamo che iniziamo a creare classi per disegnare punti (point), linee (line) e quadrati (square), e che abbiano il comportamento (metodo) display.

Il client deve avere la possibilità, usando una generica forma (shape), di poter disegnare, senza saperlo, gli oggetti di cui abbiamo parlato nella formulazione del problema.

Il client, inoltre, deve essere in grado di trattare nella stessa modalità o con lo stesso comportamento (display) qualsiasi tipo di forma, anche forme che potranno essere aggiunte in futuro (*variazione di comportamento*).

Usiamo il metodo appreso. In questo problema il "ciò che varia" è rappresentato dalle forme. Un giorno si potrebbero aggiungere (variazione del requisito) anche altre forme: cerchio, rettangolo, etc.

Inoltre quello che vede il client sono forme (astrazione).

Una prima soluzione a ciò è l'ereditarietà (come classificazione), l'incapsulamento a strati, il polimorfismo.

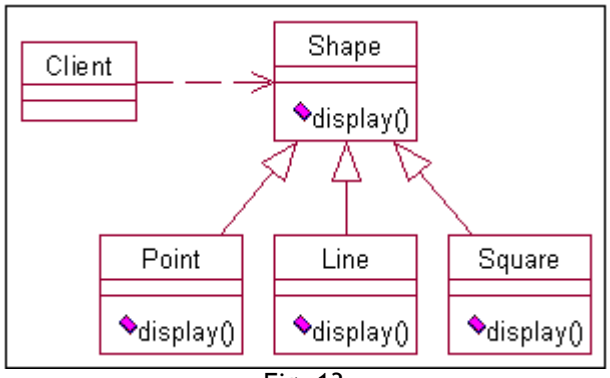


Fig. 13

Quando usare l'Adapter Pattern

I requisiti sono cambiati! Cosa succede se vogliamo aggiungere anche cerchi? Dovremmo aggiungere un'altra sotto-classe Circle!

Ma che succede se disponiamo già di una classe XXCircle, ben testata ed usata, ma che ha come metodo displayIt() ?

Purtroppo supponiamo che non ci convenga, per la quantità di software che abbiamo già sviluppato, cambiare il metodo display() in displayIt().

Né ci conviene cambiare displayIt() in display() perché esiste una marea di classi che usa XXCircle col metodo displayIt().

La soluzione è di usare l'Adapter Pattern. Creo una classe CircleAdapter che mi adatta l'interfaccia a XXCircle, come in figura 14.

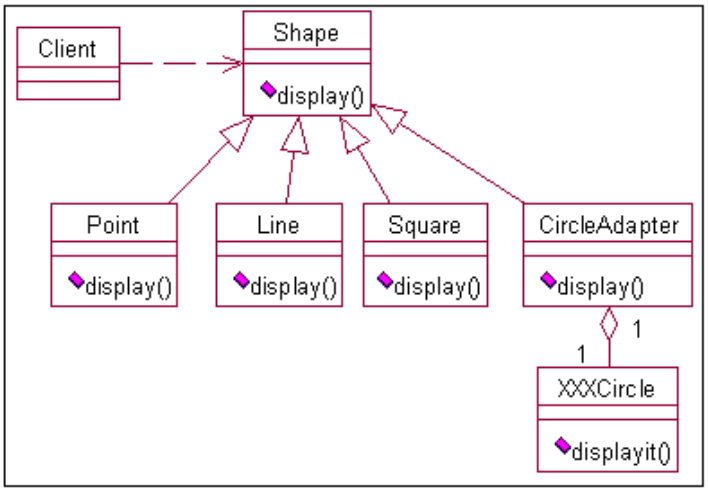


Fig. 14

Concetto OO su cui si basa l'Adapter

Il concetto è: "Piuttosto che cambiarlo, lo adatto".

L'Adapter è un Wrapper anch'esso, ma fa qualcosa di diverso rispetto ad un Facade. Il Facade semplifica l'interfaccia di un sottosistema. L'Adapter converte l'iterfaccia in un'altra.

Frammento codice Java

```
class CircleAdapter extends Shape {

  private XXCircle pxc;

  public CircleAdapter() {
    pxc = new XXCircle() ;
  }

  void public display() {
    pxc.displayIt();
  }
}
```

Frammento codice C++

```
class CircleAdapter :  public Shape {
   private:
     XXCircle *pxc;
}

CircleAdapter:: CircleAdapter() {
   pxc = new XXCircle() ;
}

void CircleAdapter::display() {
   pxc->displayIt();
}
```

Object Adapter e Class Adapter

Quello visto fin qui è un Object Adapter, perché realizzato su un oggetto (l'adattatore) che contiene un altro (quello adattato).

Un altro modo di realizzare tale Pattern è *con l'ereditarietà multipla* ed in tal caso si parla di Class Adapter.

Abstract Factory: Creazionale

L'intento dell'Abstract Factory Pattern è: "fornire un'interfaccia per creare famiglie relazionate o dipendenti di oggetti senza specificare le loro classi concrete".

Esempio

Partiamo da un esempio. Supponiamo di dover creare un sistema grafico in grado di disegnare e stampare forme, prelevate da un database, ma che il tipo di risoluzione da usare per il display e per la print dipende dalle caratteristiche del PC: velocità della CPU e la quantità di RAM disponibile.

La nostra analisi ci porta a capire di avere una matrice di possibilità come nella tabella successiva.

Funzionalità del Driver	HW low-capacity	HW high-capacity
Display	**Driver LRDD** (low resolution display driver)	**Driver HRDD** (high resolution display driver)
Print	**Driver LRPD** (low resolution print driver)	**Driver HRPD** (High resolution print driver)

Tabella 1

L'obiettivo da raggiungere è che il sistema software sia in grado di controllare i driver che sta usando, o meglio, di usare quelli disponibili.

Per semplicità, i Driver nell'esempio saranno considerati mutuamente esclusivi, ma nella realtà non sempre è così. Ci potrebbero essere casi di usi incrociati (LRDD+HRPD o HRDD+LRPD).

Quale famiglia di driver usare? Dipende dal dominio del problema. Supponiamo siano possibili solo l'accoppiata LRDD+LRPD e HRDD+HRPD per l'ipotesi semplificativa fatta.

Definire famiglie basate su un concetto unificante

L'esempio di sopra e la tabella ottenuta mette in evidenza un altro metodo: *la ricerca degli elementi comuni, rendendoli famiglie* (**Commonality Analysis**: cerca elementi che non variano).

Frammento di codice Java della prima soluzione

La prima idea istintiva è pensare ad una classe con due metodi. Ma è la soluzione peggiore: lo dimostra il frammento di codice che segue.

```
class ApControl {
public  void doDraw () {
  switch(RESOLUTION){
    case LOW:
       //use LRDD
    break;
    case HIGH:
      //use HRDD
    break;

  }
}
```

```
public void doPrint () {
  switch(RESOLUTION){
    case LOW:
        //use LRPD
    break;
    case HIGH:
       //use HRPD
     break;
  }
}

}
```

Il difetto di questa soluzione è che esiste:

- un forte accoppiamento (**Tight coupling**). Se cambia la regola di risoluzione, ad esempio cambia a MIDDLE, devo cambiare almeno due metodi che non sono nemmeno relazionati concettualmente.
- Una scarsa coesione (**Low cohesion**): i metodi hanno la responsabilità di disegnare (doDraw) o di stampare (doPrint) ma gli sto dando anche il compito di decidere in base alla risoluzione il driver da usare.

Dai GRASP Pattern e dall'OO in generale si sa che Tight coupling e Low cohesion sono una situazione da evitare assolutamente.

Switch come campanello d'allarme di soluzione non adeguata

La presenza di uno switch indica la necessità di dover introdurre un'astrazione. Indica cioè la necessità di dover introdurre un comportamento polimorfico o che c'è la presenza di responsabilità mal piazzate.

Una seconda soluzione del problema di prima è basata sull'ereditarietà.

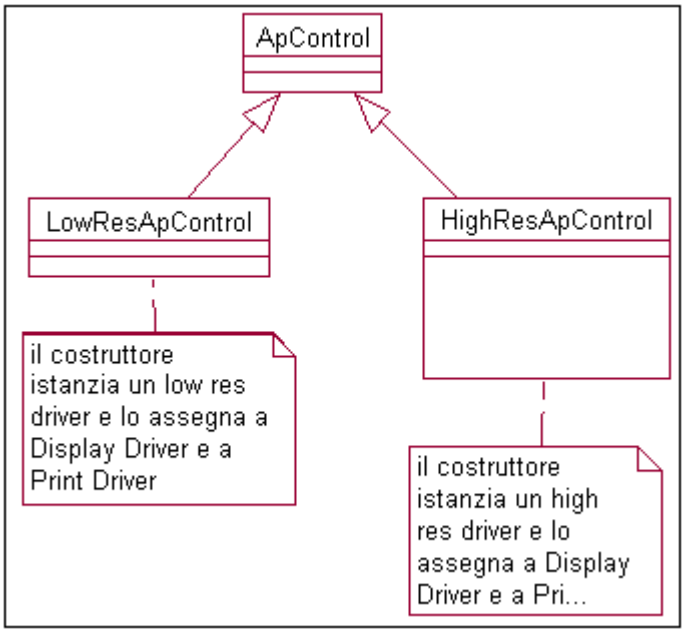


Fig. 15

Questo è comunque un semplice caso dove l'ereditarietà comunque può andar bene, ma nei casi leggermente più complessi o che possono, a tendere, diventare complessi ha degli svantaggi:

- Produce una esplosione combinatoriale: per ogni differente famiglia e per ogni nuova famiglia che nasce, devo creare una nuova classe concreta.
- Non evidenzia un chiaro significato: le classi risultanti non aiutano a comprendere che cosa si sta facendo, infatti ho dovuto specializzare le classi per ogni caso particolare. In futuro, in fase di manutenzione, sarò costretto a spendere tempo più del dovuto a (ri)capire come è stato concepito il tutto.

- Provoca una violazione della composizione: non abbiamo esaminato prima se esistesse una soluzione che privilegiasse la composizione sull'ereditarietà.

Vediamo di arrivare alla soluzione finale in vari step.

Step 1: rimpiazziamo lo switch con un'astrazione.

Ritorniamo al frammento di codice Java e allo switch e vediamo di impostare il metodo di base.

Step 2
Qual è il comportamento in gioco (Cosa devo fare)? Stampare e fare il display.
E cosa varia? La risoluzione dei driver.

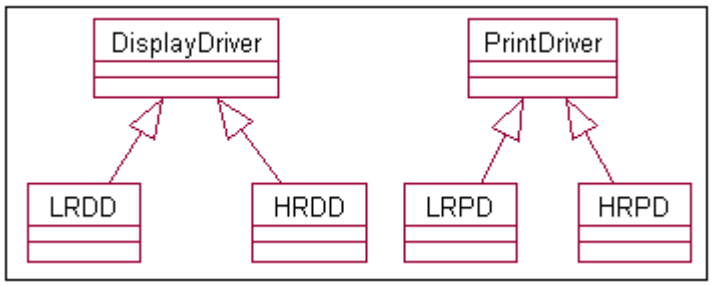


Fig. 16

Occorre, quindi, astrarre i tipi di Driver. Esistono tipi di driver per stampare e driver per disegnare (Lo stesso frammento di codice ci aveva portato a ciò), e vanno specializzati a seconda della risoluzione. D'altra parte a me interessa disegnare e stampare.

In questo caso il controllore istanza l'uno o l'altro tipo di Driver.

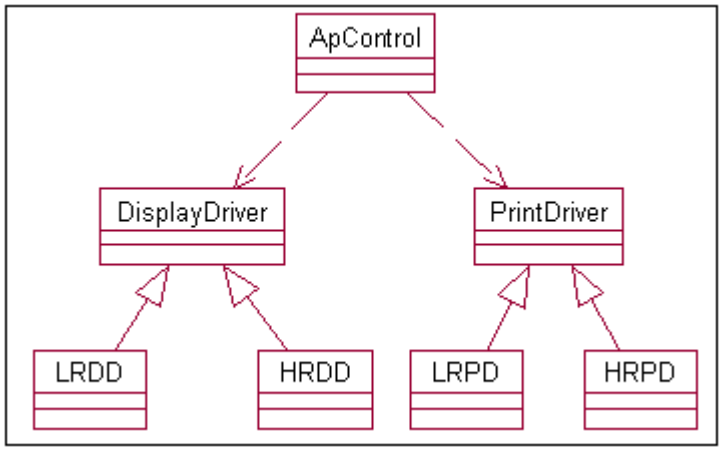


Fig. 17

In questo caso ApControl è molto più facile da capire. ApControl se deve stampare usa PrintDriver, se deve disegnare o visualizzare usa DisplayDriver, questo senza interessarsi della risoluzione!!

Seconda soluzione e frammento di codice Java

```
class ApControl {
public  void doDraw () {
  myDisplayDriver.draw();
}

public void doPrint () {
   myPrintDriver.print();
}
```

Una questione rimasta aperta nella soluzione precedente

Sebbene siamo arrivati alla soluzione giusta ci rimane un problema: come creare il giusto oggetto?

La soluzione di prima comporta che è ApControl a dover decidere che risoluzione usare: non dobbiamo far dipendere ApControl dalla risoluzione per evitare problemi futuri di manutenzione. Dobbiamo usare una factory!

ApControl deve far uso, cioè, di un Factory Object, che chiamiamo ResFactory. E' ResFactory che è responsabile di determinare il tipo di risoluzione e che deve creare il driver giusto.

ResFactory crea il giusto Driver e lo passa ad ApControl. Quindi ApControl usa ResFactory che gli restituisce il Driver creato. La soluzione è disaccoppiata e con buona coesione.

Anche in ResFactory si deve evitare lo switch (variazioni di scelte -> incapsulare la variazione).

Poiché la risoluzione è un elemento variabile (la risoluzione potrebbe diventare anche un altro valore es: MIDDLE) occorre incapsulare tale variazione in una classe.

La soluzione è astrarre anche ResFactory.

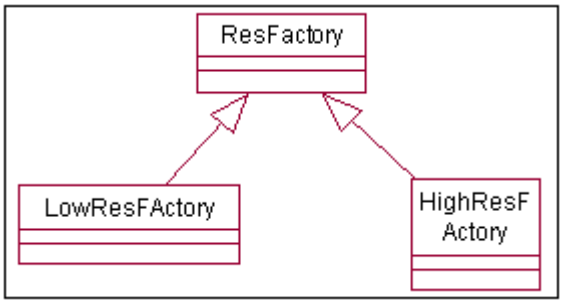


Fig- 18

Ho incapsulato la variazione con un'astrazione che rappresenta i concetti. In particolare nel caso di ResFactory io ho due differenti comportamenti (metodi):

- Ottenere il diplay driver da usare
- Ottenere il print driver da usare

La strategia finora usata

Strategia	Soluzione
Cercare cosa varia e incapsularlo	La scelta del driver poteva variare a seconda della risoluzione che poteva assumere diversi valori, di cui altri in futuro. Per cui la variazione è stata incapsulata in ResFactory.
Favorire la composizione sull'eredità	Le variazioni sono state poste in separati oggetti. Con ApControl ho un unico controllo degli oggetti.
Disegnare per interfacce e non per implementazioni	ApControl usa e sa chiamare ResFactory ma non sa come ResFactory è fatta.

Ora mettiamo tutto assieme nella figura 19.

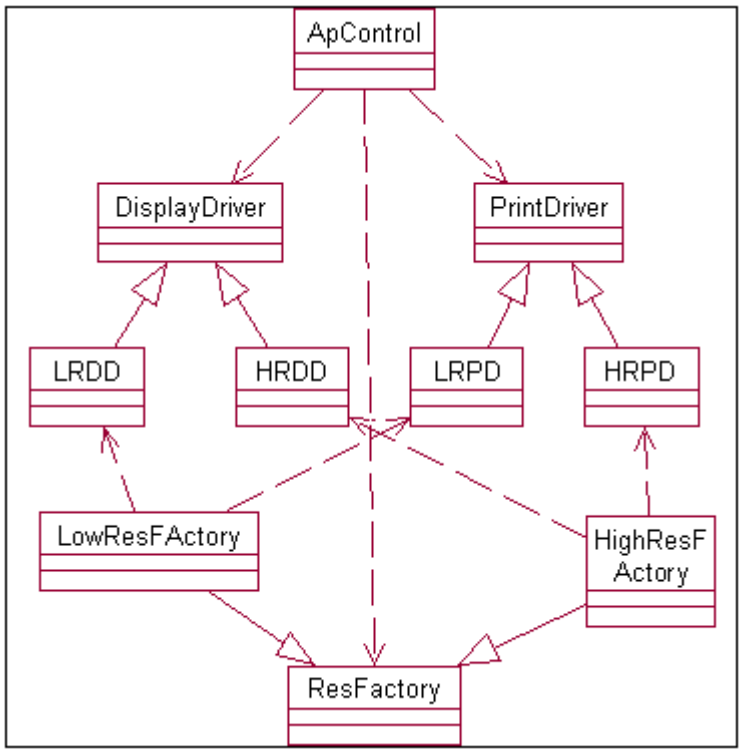


Fig. 19

ResFactory è astratta e da qui nasce il nome del Pattern: Abstract Factory.

Frammento codice Java

```
abstract class ResFactory {
   abstract public DisplayDriver getDisplayDrv();
   abstract public PrintDriver getPrintDrv();
}

class LowResFactory extends ResFactory{
   public DisplayDriver getDisplayDrv(){
     return new LRDD();
   }
   public PrintDriver getPrintDrv(){
     return new LRPD();
   }

}

class HighResFactory extends ResFactory{
   public DisplayDriver getDisplayDrv(){
     return new HRDD();
   }
   public PrintDriver getPrintDrv(){
     return new HRPD();
   }

}
```

Concetti di base ulteriori

L'Abstract Factory mette in evidenza la decomposizione per responsabilità.

Ma Come si individuavano le responsabilità? Con le domande:

- Chi usa gli oggetti per far fare le operazioni? (ApControl)
- Chi seglie i particolari oggetti? (ResFactory)

Quando usare l'Abstract Factory

Quando il dominio del problema lavora con famiglie di oggetti, dove ogni famiglia è usata per un caso particolare.

Qualche esempio:

- Differenti sistemi operativi (programmi cross-platform)
- Differenti linee guida di performance
- Differenti versioni di applicazioni in gioco
- Differenti caratteristiche per gli utenti delle applicazioni

Una volta identificato le famiglie di oggetti occorre decidere come implementare i vari casi di ogni famiglia.

Suggerimenti

A volte si possono adottare una delle seguenti soluzioni:

- File di configurazione che specificano l'oggetto da creare nella Factory.
- Ogni famiglia può avere un record in un database che specifica quale oggetto creare

In Java si sfrutta la classe Class per istanziare la giusta classe che serve.

Esercizio

HiFiMarket è un posto di vendita di prodotti HiFi. HiFiMarket, In particolare, vende due famiglie di prodotti, basati su tecnologie diverse: *tape* e *compact disk*. In entrambi i casi ognuno di tali prodotti dispone di un riproduttore (*player*) e di un masterizzatore (*recorder*).

Un cliente deve essere in grado da un'unica interfaccia (è quello che succede anche lato hardware) di usare allo stesso modo ogni prodotto della famiglia; ad esempio su un tape registra e poi può ascoltarlo, stesso dicasi per un CD.

L'Abstract Factory consente proprio di definire un'interfaccia per famiglie di oggetti relazionati senza dover esprimerne le classi concrete. Difatti ogni famiglia, come nell'esempio precedente, avrà la sua Facory che provvederà a questo.

Una soluzione è presentata nella figura 20 successiva.

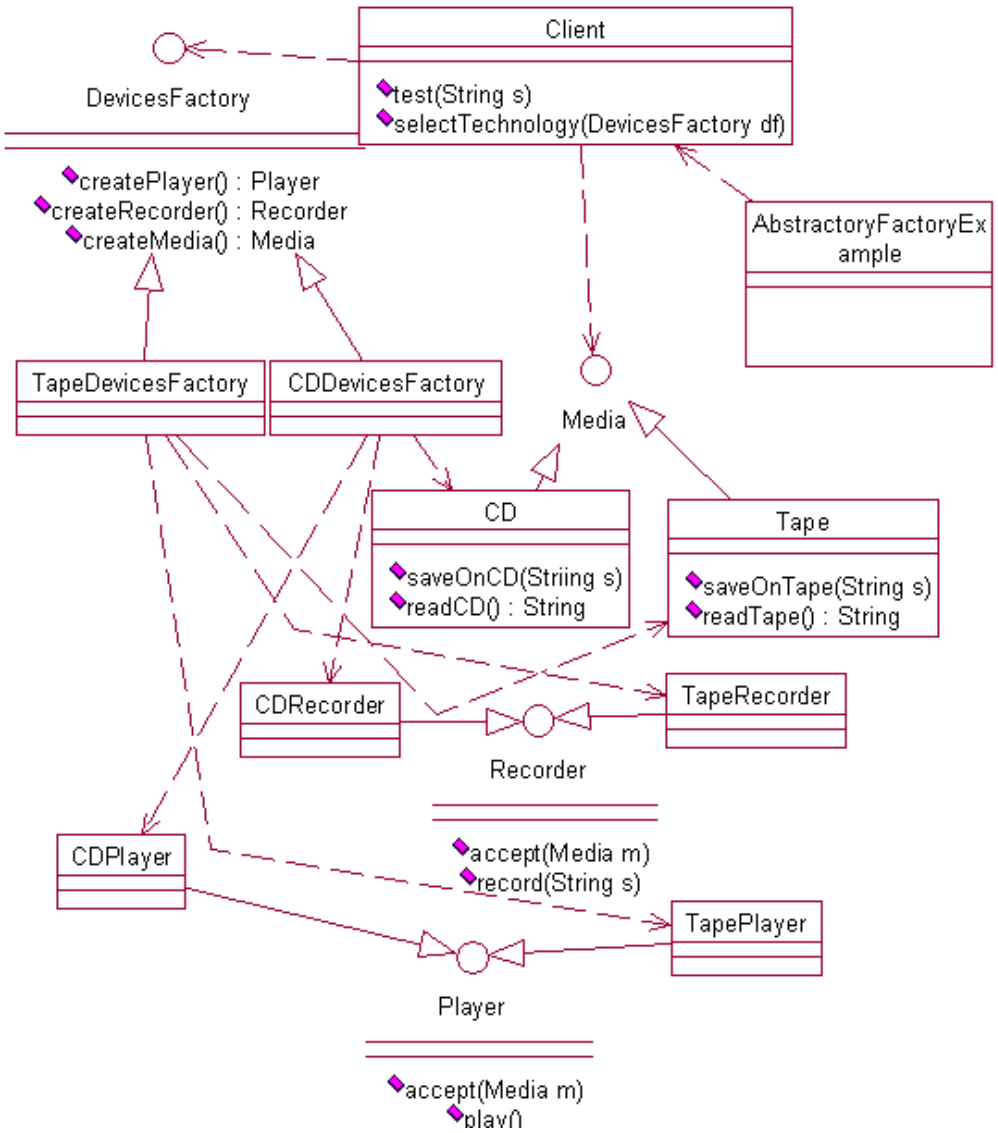


Fig. 20

Il client usa solo l'interfaccia del Media (Un cliente inserisce solo il tape o il CD per ascoltare o registrare tutto sommato).

Media è un *Marker Interface* per identificare i supporti di registrazione. Le interfacce Recorder e Player dichiarano i metodi indipendentemente dal supporto.

Le classi concrete della DevicesFactory si occupano di creare i prodotti e restituirli: Tape, CD, TapeRecorder, CDRecorder, TapePlayer, CDPlayer. A scegliere la tecnologia (la factory giusta) è il main che chiama il client.

Frammento di codice Java dell'esercizio

```
public interface Media { }

public interface Player {
  public void accept( Media med );
  public void play( );
```

```
}

public interface Recorder {
  public void accept( Media med );
  public void record( String sound );
}

public class Tape implements Media {
  private String tape= "";

  public void saveOnTape( String sound ) {
    tape = sound;
  }
  public String readTape( ) {
    return tape;
  }
}

public class TapeRecorder implements Recorder {
  Tape tapeInside;

  public void accept( Media med ) {
    tapeInside = (Tape) med;
  }

  public void record( String sound ) {
   if( tapeInside == null )
     System.out.println( "Error: Insert a tape." );
   else
     tapeInside.saveOnTape( sound );
  }
}

public class TapePlayer implements Player {
  Tape tapeInside;

  public void accept( Media med ) {
    tapeInside = (Tape) med;
  }

  public void play( ) {
    if( tapeInside == null )
      System.out.println( "Error: Insert a tape." );
    else
      System.out.println( tapeInside.readTape() );
 }
}

public class CD implements Media{
  private String track = "";

  public void writeOnDisk( String sound ) {
    track = sound;
  }

  public String readDisk( ) {
    return track;
  }
}
```

```
public class CDRecorder implements Recorder {
  CD cDInside;

  public void accept( Media med ) {
    cDInside = (CD) med;
  }

  public void record( String sound ) {
    if( cDInside == null )
      System.out.println( "Error: No CD." );
    else
      cDInside.writeOnDisk( sound );
 }
}

public class CDPlayer implements Player {
  CD cDInside;

  public void accept( Media med ) {
    cDInside = (CD) med;
  }

  public void play( ) {
   if( cDInside == null )
     System.out.println( "Error: No CD." );
   else
     System.out.println( cDInside.readDisk() );
  }
}

public interface DevicesFactory {
  public Player createPlayer();
  public Recorder createRecorder();
  public Media createMedia();
}

public class TapeDevicesFactory implements DevicesFactory {
  public Player createPlayer() {
    return new TapePlayer();
  }

  public Recorder createRecorder() {
    return new TapeRecorder();
  }

  public Media createMedia() {
    return new Tape();
  }
}

public class CDDevicesFactory implements DevicesFactory {

  public Player createPlayer() {
    return new CDPlayer();
  }
  public Recorder createRecorder() {
    return new CDRecorder();
  }
  public Media createMedia() {
    return new CD();
```

```
  }
}

class Client {
  DevicesFactory technology;

  public void selectTechnology( DevicesFactory df ) {
    technology = df;
  }

  public void test(String song) {
    Media media = technology.createMedia();
    Recorder recorder = technology.createRecorder();
    Player player = technology.createPlayer();
    recorder.accept( media );
    System.out.println( "Recording the song : " + song );
    recorder.record( song );
    System.out.println( "Listening the record:" );
    player.accept( media );
    player.play();
  }
}

public class AbstractFactoryExample {
  public static void main ( String[] arg ) {
    Client client = new Client();
    System.out.println( “**Testing tape devices” );
    client.selectTechnology( new TapeDevicesFactory() );
    client.test( "I wanna hold your hand..." );
    System.out.println( “**Testing CD devices” );
    client.selectTechnology( new CDDevicesFactory() );
    client.test( "Fly me to the moon..." );
  }
}
```

Bridge: Strutturale

L’intento del Bridge Pattern è: “Disaccoppiare un’astrazione dalla sua implementazione così che le due possano variare indipendentemente”.

Comprensione della definizione

La definizione è abbastanza complessa da capire. Dobbiamo spiegare almeno tre termini.

Disaccoppiare: significa che le cose si possono comportare indipendentemente tra loro.

Astrazione: mostra come una cosa è concettualmente relazionata ad un’altra.

Implementazione: è la parte difficile. Ci si riferisce all’oggetto che la classe astratta e le sue derivazioni usano per implementarsi. Non sto parlando delle classi derivate dalle classi astratte perché altrimenti sarebbero classi concrete.

Concetto OO su cui si basa

Il Bridge Pattern è un eccellente esempio di due concetti fondamentali:

- “Cerca cosa varia e incapsulalo”;
- “Favorisci la composizione degli oggetti sulla eredità di classi”.

Quando usarlo

Nei problemi dove si vuole:

- Una variazione nell’astrazione dei concetti
- Una variazione nel come questi concetti sono implementati.

Esempio

Supponiamo di dover scrivere un programma che deve disegnare rettangoli con due programmi di disegno DP1 e DP2 come indicato in tabella.

Utilizzo	DP1	DP2
disegnare linee	draw_a_line(x1, y1, x2, y2)	drawline(x1, y1, x2, y2)
disegnare cerchi	draw_a_circle(x, y, r)	drawcircle(x, y, r)

Al momento, quando istanzio un rettangolo, devo sapere se usare il programma DP1 o DP2.

L’obiettivo è di fare in modo che il client non abbia il fastidio circa quale programma di disegno si debba utilizzare.

Una prima soluzione è mostrata di seguito, nella figura 21.

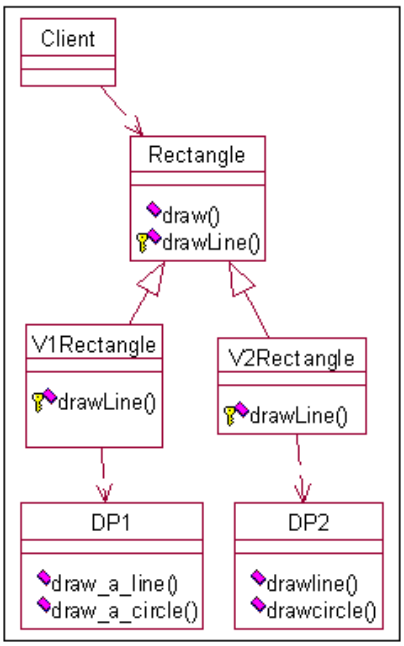


Fig. 21

Frammento codice Java

```
class Rectangle {
  public void draw() {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
  }
```

```
  abstract protected void drawLine( double x1, double y1, double x2, double y2) ;
}

class V1Rectangle extends Rectangle {
   private DP1 dp1 ;

   drawLine(double x1, double y1, double x2, double y2) {
     dp1.draw_a_line(x1, y1, x2, y2) ;
   }
}

class V2Rectangle extends Rectangle {
  public DP2 dp2 ;
  drawLine(double x1, double y1, double x2, double y2) {
    dp2.drawline(x1, x2, y1, y2) ; // vanno riarrangiati anche gli argomenti
  }
}
```

Ma i requisiti cambiano!

Ci è stato richiesto - all'ultimo momento - di supportare anche i cerchi!! Sempre con due possibili programmi di disegno.

Una facile soluzione per l'analista è quella di disegnare con l'eredità in mente. Tuttavia non sempre è la giusta soluzione perché si va verso una esplosione combinatoriale negli strati successivi che gli creerà problemi di manutenzione e riuso.

Vediamone un esempio.

Potrei pensare di usare lo stesso diagramma delle classi di prima aggiungendo:

- Un'astrazione shape
- l'astrazione rettangolo e quella cerchio

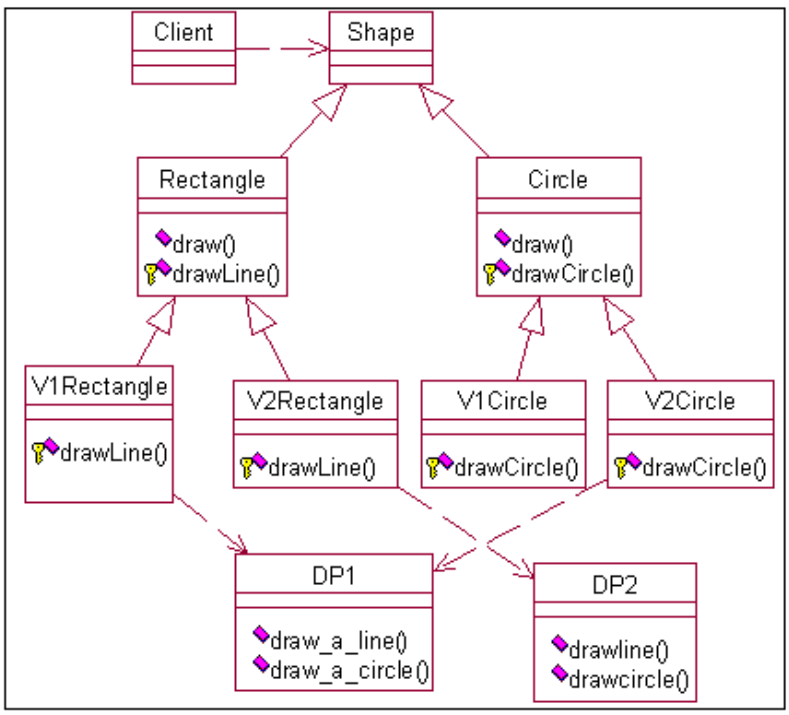


Fig. 22

In fondo sto rispettando le regole:

- incapsulamento a strati
- astrazione (ereditarietà come classificazione)

Ma vado a infrangermi contro un aumento combinatoriale di oggetti all'ultimo strato! E questo va evitato!!

Frammento codice Java

```
abstract class Shape {
  abstract public void draw();
}

abstract class Rectangle extends Shape{
   public void draw() {
     drawLine(_x1, _y1, _x2, _y1);
     drawLine(_x2, _y1, _x2, _y2);
     drawLine(_x2, _y2, _x1, _y2);
     drawLine(_x1, _y2, _x1, _y1);
   }
   abstract protected void drawLine( double x1, double y1, double x2, double y2) ;
}

class V1Rectangle extends Rectangle {
  private DP1 dp1 ;
  drawLine(double x1, double y1, double x2, double y2) {
    dp1.draw_a_line(x1, y1, x2, y2) ;
  }
}

class V2Rectangle extends Rectangle {
  private DP2 dp2 ;
  drawLine(double x1, double y1, double x2, double y2) {
    dp2.drawline(x1, x2, y1, y2) ; // vanno riarrangiati anche gli argomenti
  }
}

abstract class Circle extebds Shape{
  public void draw() {
    drawCircle(x, y, r);
  }
  abstract protected void drawCircle( double x, double y, double r) ;
}

class V1Circle extends Circle {
   private DP1 dp1 ;
   protected void drawCircle() {
     dp1.draw_a_circle (x, y, r) ;
   }
}

class V2Circle extends Circle {
   private DP2 dp2 ;
   protected void drawCircle() {
     dp2.draw_a_circle (x, y, r) ;
   }
}
```

Svantaggi

Dal diagramma delle classi si osserva che introdurre una classe al livello due, dopo shape, comporta l'aggiunta di altri due classi, cioè da 4 passo a 6 tipi di shape ed inoltre passo da due a tre tipi di programmi di disegno.

Qual è la causa dell' esplosione dei tipi di shape e dei programmi di disegno? Perché le astrazioni e le implementazioni sono fortemente accoppiate!! Infatti nella soluzione l'astrazione shape deve conoscere il tipo di programma per disegnare.

Occorre riuscire a disaccoppiare l'astrazione dall'implementazione. Il disegno precedente seguiva un "poor approach". La situazione richiede quindi un Design Pattern. Per arrivare al Pattern giusto la strategia è:

- "Individuare le variazioni ed incapsularle".
- "Favorire la composizione rispetto alla ereditarietà"

Commonality analysis/Variability analysis

La Commonality Analysis (J. Coplien) studia gli elementi comuni del problema o non variabili. In altri termini la Commonality Analysis individua le strutture che "difficilmente" variano o che molto lentamente nel tempo variano.

La Commonality Analysis, quindi, individua gli elementi stabili e dà all'architettura la longevità della soluzione.

La Variability Analysis, al contrario, individua gli elementi non stabili e che variano. Sono questi che possono influire sul ridisegno!! Ed è questo che occorre evitare.

Un approccio diverso

Primo step: individuare cosa varia

Abbiamo differenti (variazione) tipi di Shape e differenti (variazione) tipi di programmi di disegno. Adottiamo quindi la strategia adatta a gestire le variazioni. Cominciamo a disegnare gli oggetti che variano.

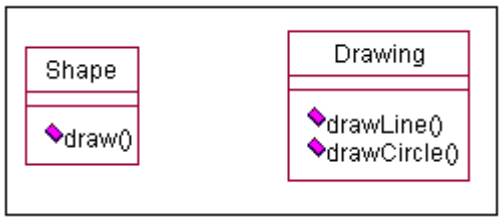


Fig. 23

Secondo step: rappresentare i tipi di variazione

Rappresentiamo le variazioni presenti rispetto alle astrazioni e incapsuliamo ciò che varia. Sotto l'astrazione Shape ho rettangoli e cerchi che hanno la responsabilità di disegnarsi. Il Drawing (disegnare) si deve basare su un V1Drawing e su un V2Drawing perché avevamo programmi diversi.

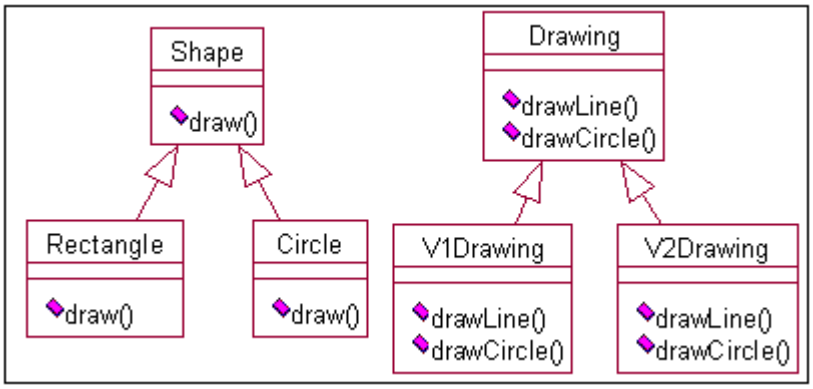


Fig. 24

Finora abbiamo usato:

- ereditarietà come classificazione (astrazione)
- l'incapsulamento

Step 3: comporre le parti che variano.

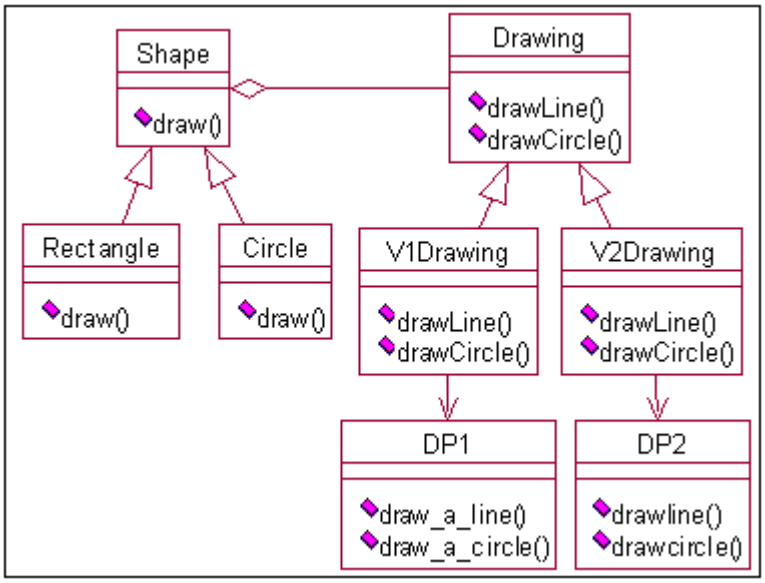


Fig. 25

Nella figura si nota che a sinistra c'è l'astrazione (Shape Abstraction) e i suoi tipi. A destra l'implementazione (Drawing Implementation).

Questo è il Bridge Pattern! La regola fondamentale del Bridge Pattern è: "One rule, one place"!!

L'implementazione difatti è in un solo posto, a vantaggio della manutenzione e prevenendo problemi futuri. Inoltre il livello due sotto Shape non esiste. Sotto Drawing i livelli non crescono in modo combinatorio.

Bridge & Adapter o altre combinazioni

Spesso il Bridge Pattern può anche contenere un Adapter Pattern; in altri casi si parla di composite design pattern.

Frammento codice Java

```
Class Client {
  public static void main(String argv[]){
```

```
    Shape r1, r2;
    Drawing dp;
    dp = new V1Drawing();
    r1 = new Rectangle(dp, 1,1,2,2);
    dp = new V2Drawing();
    r2 = new Circle (dp, 2,2,2ù3);
  }
}

abstract class Shape {
  private Drawing _dp;

  abstract public void draw();
  Shape(Drawing dp){
    _dp = dp;
  }

  protected void drawLine(double x1, double y1, double x2, double y2) {
    _dp.drawLine(x1,y1,x2,y2);
  }

  protected void drawCircle(double x, double y, double r) {
   _dp.drawCircle(x,y,r);
  }
}

abstract class Drawing {
  abstract public void drawLine( double x1, double y1, double x2, double y2);
  abstract public void drawCircle( double x, double y, double r);
}

class V1Drawing extends Drawing {
   private DP1 dp1 ;
   public void drawLine(double x1, double y1, double x2, double y2) {
     dp1.draw_a_line(x1, y1, x2, y2) ;
   }
   public void drawCircle(double x, double y, double r) {
     dp1.draw_a_line(x, y, r) ;
   }
}

class V2Drawing extends Drawing {
   private DP2 dp2 ;
   public void drawLine(double x1, double x2, double y1, double y2) {
     dp2.draw_a_line(x1, y1, x2, y2) ;
   }

   public void drawCircle(double x, double y, double r) {
     dp2.draw_a_line(x, y, r) ;
       }
}

class Rectangle extends Shape{
   private double _x1, _x2, _y1, _y2;
   public Rectangle( Drawing dp, double x1, double y1, double x2, double y2){
     super(dp) ;
     _x1= x1 ;
     _x2 = x2;
     _y1= y1;
     _y2 = y2;
```

```
  }

  public void draw(){
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
  }
}

class Circle extends Shape{
  private double _x, _y, _r;

  public Circle( Drawing dp, double x, double y, double r){
    super(dp) ;
    _x= x;
    _y= y;
    _r= r;
  }

  public void draw(){
    drawCircle(_x, _y, _r);
  }
}

class DP1 {
  static public void draw_a_line( double x1, double y1, double x2, double y2){
    // implementation
  }

  static public void draw_a_circle( double x, double y, double r){
    // implementation
  }
}

class DP2 {
  static public void drawline( double x1, double y1, double x2, double y2){
    // implementation
  }
  static public void drawcircle( double x, double y, double r){
    // implementation
  }
}
```

Esercizio

Supponiamo di dover scrivere dei programmi di matematica dove si deve trattare Matrici, complete o sparse. Una matrice sparsa è una matrice di grosse dimensione, come righe x colonne, che ha la maggior parte degli elementi nulli.

A livello di astrazione si ha una situazione come in figura 26.

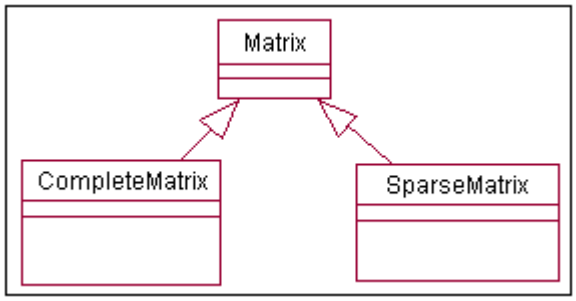


Fig. 26

ComplexMatrix ha tutti elementi costituiti de celle (oggetti) contenenti valori; mentre SparseMatrix ha alcune celle contenenti valori diversi da zero.

Le celle delle matrici sono caratterizzate da attributi come le coordinate x e y ed il valore da contenere e devono essere immagazzinate in una qualche forma di collezione di oggetti. In Java, ad esempio si può far uso di `java.util.Vector` e `java.util.ArrayList` dove la gerarchi è del tipo in figura 27.

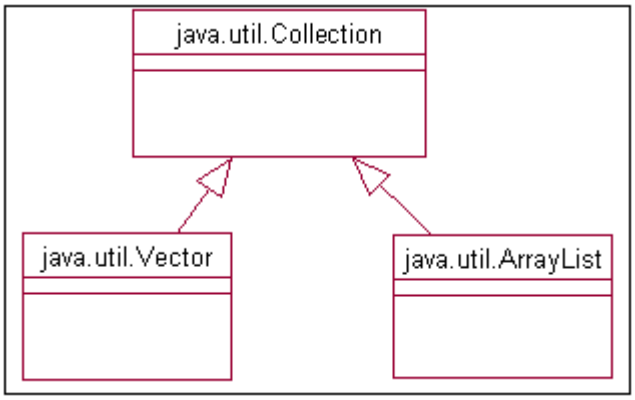


Fig. 27

Se vogliamo mantenere il concetto di matrice completa e matrice sparsa, una prima soluzione non adeguata, è quella di figura 28.

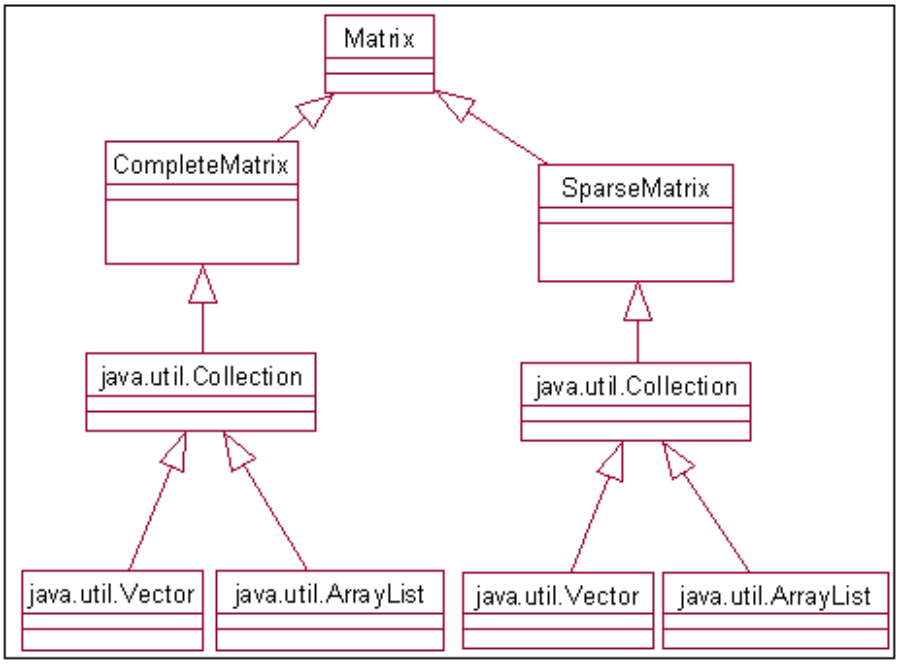


Fig. 28

In questo modo se devo aggiungere un nuovo tipo di matrice, devo aggiungere due classi (ArrayList o Vector) ogni volta.

La soluzione è di dividere la parte concettuale da quella di implementazione come in figura 29.

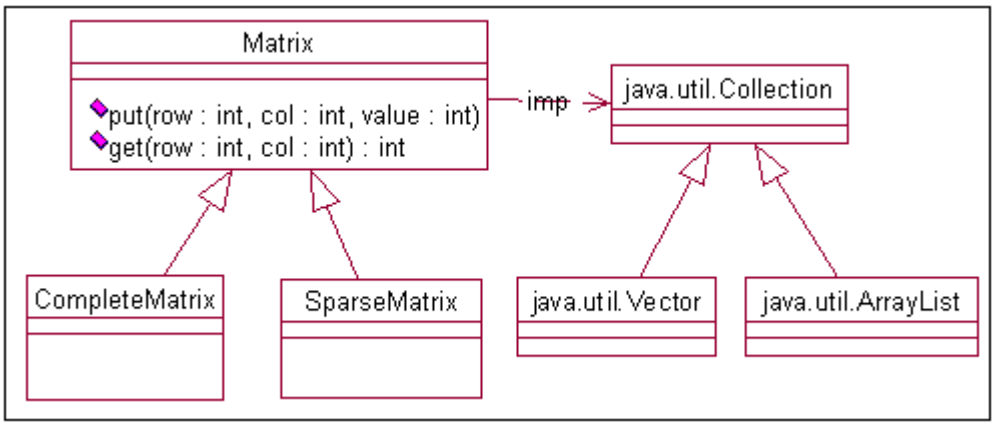


Fig. 29

Frammento di codice Java

```
import java.util.Collection;
import java.util.Iterator;

public abstract class Matrix {
  int rows, cols;
  Collection data ;
  protected Matrix( int rows, int cols, Collection collection ) {
    this.rows = rows;
    this.cols = cols;
    data = collection;
}

protected MatrixCell createPosition( int row, int col ) throws MatrixIndexOutOfBoundsException {
  MatrixCell mc = getPosition( row, col );
  if( mc == null )
    mc = new MatrixCell( row, col );
    data.add( mc );
    return mc;
}

protected void deletePosition( MatrixCell toDelete ) throws MatrixIndexOutOfBoundsException {
  data.remove( toDelete );
}

protected MatrixCell getPosition( int row, int col ) throws MatrixIndexOutOfBoundsException {

  if( row < 0 || row >= this.rows || col < 0 || col >= this.cols )
     throw new MatrixIndexOutOfBoundsException();

  Iterator it = data.iterator();
  while( it.hasNext() ) {
    MatrixCell mc = (MatrixCell) it.next();
    if( mc.row == row && mc.col == col )
       return mc;
  }
  return null;
}
```

```
public abstract void put( int row, int col, int value ) throws MatrixIndexOutOfBoundsException;
public abstract int get( int row, int col ) throws MatrixIndexOutOfBoundsException;

}


import java.util.Collection;

public class CompleteMatrix extends Matrix {
  public CompleteMatrix( int rows, int cols, Collection collection ) {
   super( rows, cols, collection );
   for(int i = 0 ; i< rows; i++ )
     for(int j = 0 ; j< cols; j++ )
       createPosition( i, j );
  }
  public void put( int row, int col, int value ) throws MatrixIndexOutOfBoundsException{
    MatrixCell cell = getPosition( row, col );
    cell.value = value;
  }

  public int get( int row, int col ) throws MatrixIndexOutOfBoundsException {
    MatrixCell cell = getPosition( row, col );
    return cell.value;
  }
}


import java.util.Collection;

public class SparseMatrix extends Matrix {
  public SparseMatrix( int rows, int cols, Collection collection ) {
    super( rows, cols, collection );
  }
  public void put( int row, int col, int value ) throws MatrixIndexOutOfBoundsException {
    MatrixCell cell = getPosition( row, col );
    if( cell != null )
      if( value == 0 ) {
        deletePosition( cell );
      } else {
          cell.value = value;
      }
    else
      if( value != 0 ) {
        cell = createPosition( row, col );
        cell.value = value;
      }
 }

public int get( int row, int col ) throws MatrixIndexOutOfBoundsException {
   MatrixCell cell = getPosition( row, col );
   if( cell == null)
     return 0;
   else
     return cell.value;
}
}

public class MatrixCell {
  public int row, col, value;
```

```
  public MatrixCell( int r, int c ) {
    row = r;
    col = c;
    value = 0;
  }
}


import java.util.Vector;
import java.util.ArrayList;
public class BridgeExample {
   public static final int ROWS = 3;
   public static final int COLS = 4;
   public static void main( String[] arg ) {
   CompleteMatrix matrixCV = new CompleteMatrix( ROWS, COLS, new Vector() );
   System.out.println( "Complete Matrix with Vector:");
   matrixCV.put( 1, 2, 1 );
   matrixCV.put( 2, 1, 2 );
   matrixCV.put( 0, 3, 3 );
   matrixCV.put( 1, 2, 0 );
   for(int i = 0 ; i< ROWS; i++ ) {
     for(int j = 0 ; j< COLS; j++ )
       System.out.print( matrixCV.get( i, j )+" " );
     System.out.println();
   }

   SparseMatrix matrixSV = new SparseMatrix( ROWS, COLS, new Vector() );
   System.out.println( "Sparse Matrix with Vector:");
   matrixSV.put( 1, 2, 1 );
   matrixSV.put( 2, 1, 2 );
   matrixSV.put( 0, 3, 3 );
   matrixSV.put( 1, 2, 0 );
   for(int i = 0 ; i< ROWS; i++ ) {
     for(int j = 0 ; j< COLS; j++ )
       System.out.print( matrixSV.get( i, j )+" " );
     System.out.println();
}

CompleteMatrix matrixCA = new   CompleteMatrix( ROWS, COLS, new ArrayList() );
System.out.println( "Complete Matrix with ArrayList:");
matrixCA.put( 1, 2, 1 );
matrixCA.put( 2, 1, 2 );
matrixCA.put( 0, 3, 3 );
matrixCA.put( 1, 2, 0 );
for(int i = 0 ; i< ROWS; i++ ) {
  for(int j = 0 ; j< COLS; j++ )
    System.out.print( matrixCA.get( i, j )+" " );
  System.out.println();
}

SparseMatrix matrixSA = new SparseMatrix( ROWS, COLS, new ArrayList() );
System.out.println( "Sparse Matrix with ArrayList:");
matrixSA.put( 1, 2, 1 );
matrixSA.put( 2, 1, 2 );
matrixSA.put( 0, 3, 3 );
matrixSA.put( 1, 2, 0 );
for(int i = 0 ; i< ROWS; i++ ) {
  for(int j = 0 ; j< COLS; j++ )
   System.out.print( matrixSA.get( i, j )+" " );
  System.out.println();
```

```
}
}
}
```

Strategy: Comportamentale

Iniziamo con un esempio. Insisteremo sempre su un concetto: i Design Pattern permettono di "gestire le variazioni di requisito" se nella progettazione si tiene conto di voler ottenere tale flessibilità.

Esempio

Vediamo un esempio. Supponiamo di dover progettare un sistema di e-commerce che è destinato al mercato USA; esso deve gestire richieste di vendite provenienti da INTERNET.

Nell'architettura del sistema è conveniente che esista un oggetto controller (TaskController) che gestisca le richieste di vendita e che rediriga le stesse, offline, verso un altro oggetto (OrdineDiVendita) come mostrato in figura 30.

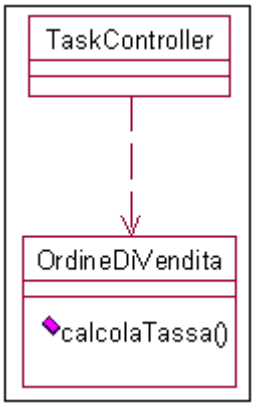


Fig. 30

Supponiamo che OrdineDiVendita deve consentire:

- riempire una form di una GUI per avere le informazioni di vendita
- gestire il calcolo delle tasse di spedizioni
- processare l'ordine e stampare una ricevuta

Ohibò, il requisito è variato!

Abbiamo appena modellato, che qualcuno ci avverte che c'è una variazione sul requisito. Ora occorre calcolare le tasse di spedizione anche per il Canada e forse anche per Cuba!! Siamo costretti a gestire le nuove regole. Una prima idea di soluzione è estendere la classe OrdineDiVendita con una classe CanadianOrdineDiVendita dove il metodo calcolaTassa() è sottoposto a overload.

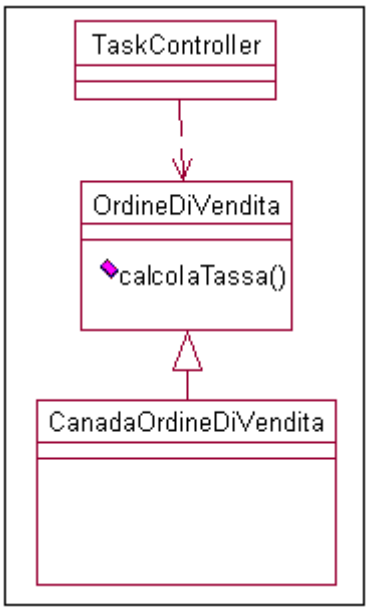


Fig. 31

E' una soluzione che mi consente di estendere con altri classi eventualmente, ma ha qualche difetto.

Il principio da cui sarei dovuto partire è sempre:"Valuta cosa deve essere variabile nel disegno e incapsula i concetti che variano".

Inoltre tra i principi dei Design Pattern c'è anche: "Favorisci la composizione sull'ereditarietà" e la soluzione di sopra non è in tale direzione!

Applichiamo il metodo

Step 1 Analizziamo cosa varia

Nell'esempio è il calcolo della tassa che varia e per incapsulare ciò dovremmo creare una classe astratta che esprime il concetto del calcolo della tassa e da essa far ereditare le variazioni.

Step 2 Incapsuliamo la variazione con una astrazione

OrdineVendita conterrà le variazioni con la composizione.

Vediamo la soluzione della figura 32 successiva.

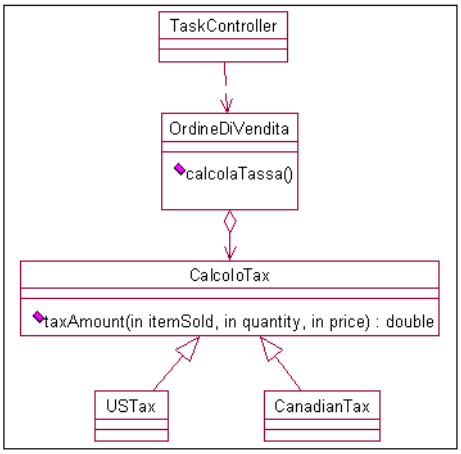


Fig. 32

I vantaggi della soluzione

Nella iniziale soluzione di figura 31, l'oggetto TaskController, usando il polimorfismo, doveva conoscere obbligatoriamente che tipo istanziare.

In altri termini, con tale soluzione, il client è dipendente dalla parte server e deve conoscere le sottoclassi. Questo è un'errata strategia per un client!

Nella soluzione di figura 32, TaskController è indipendente da questo. In questo caso è OrdineDiVendita a decidere come farlo (ad esempio in base ad una configurazione).

Questo approccio permette di far variare le regole indipendentemente dall'oggetto che OrdineDiVendita utilizza. E' proprio questo che cerca di risolvere lo Strategy Pattern.

L'intento dello Strategy

Lo Strategy Pattern "definisce una famiglia di algoritmi, incapsula ognuno, e li rende intercambiabili. In altri termini consente di variare gli algoritmi indipendentemente dal client che lo utilizza".

Quando usare lo Strategy

Il "sintomo" del sistema, che manifesta la necessità di utilizzare uno Strategy Pattern, si nota spesso con una o più delle seguenti esigenze:

- oggetti con responsabilità
- implementazione di responsabilità differenti attraverso l'uso del polimorfismo
- gestione diversa di un qualcosa che concettualmente è lo stesso algoritmo

Esercizio

Occorre progettare una applicazione che offra funzionalità matematiche e che offre una opportuna classe ArrayFormat (array di formati numerici) per la rappresentazione di vettori di numeri. Tra i metodi della classe deve esisterne uno che permette la stampa del vettore MathFormat nelle seguenti due modalità:

$$\{1,-2,5,...\}$$
$$Arr[0]=1 Arr[1]=-2 Arr[2]=5...$$

Potrebbe succedere che tali formati, in futuro, possano essere sostituiti da altre regole di formattazione.

A questo punto, poiché la parte che è soggetta a variazioni è proprio l'algoritmo di formattazione e stampa dell'array va incapsulato con un'astrazione. In pratica l'algoritmo va isolato, così si può farlo variare in modo indipendente dal resto!

Abbiamo visto precedentemente che lo Strategy Pattern aiuta a fare questo. La logica o le regole vanno incapsulate in classi concrete (**ConcreteStrategy**) che implementano l'interfaccia e consentono agli oggetti ArrayFormat (Context) di interagire con loro. L'interfaccia deve fornire un accesso efficiente ai dati del Context, richiesti da ogni ConcreteStrategy e viceversa.

La figura 33 successiva mostra la soluzione.

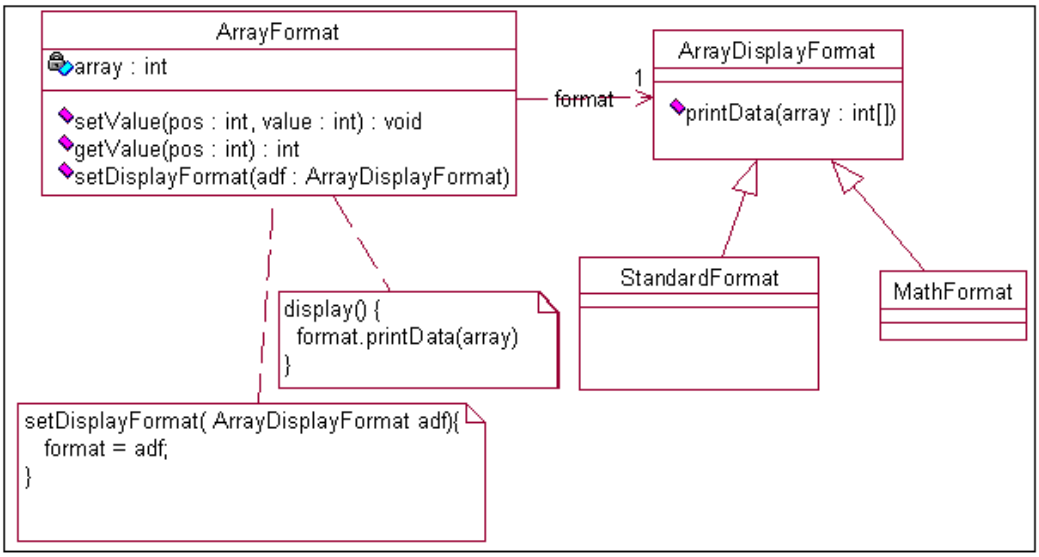


Fig. 33

Frammento di codice Java dell'esercizio

```
public class ArrayFormat {
  private int[] array;
  private int size;
  ArrayDisplayFormat format;

  public ArrayFormat( int size ) {
    array = new int[ size ];
  }

  public void setValue( int pos, int value ) {
    array[pos] = value;
  }

  public int getValue( int pos ) {
    return array[pos];
  }

  public int getLength( int pos ) {
    return array.length;
  }
  public void setDisplayFormat( ArrayDisplayFormat adf ) {
   format = adf;
  }

  public void display() {
    format.printData( array );
  }
}

public class StandardFormat implements ArrayDisplayFormat {
  public void printData( int[] arr ) {
  System.out.print( "{ " );
  for(int i=0; i < arr.length-1 ; i++ )
    System.out.print( arr[i] + ", " );
    System.out.println( arr[arr.length-1] + " }" );
  }
}
```

```
public class MathFormat implements ArrayDisplayFormat {
  public void printData( int[] arr ) {
    for(int i=0; i < arr.length ; i++ )
      System.out.println( "Arr[ " + i + " ] = " + arr[i] );
 }
}
```

Un esempio d'uso potrebbe essere

```
public class MyStrategyExample {
  public static void main (String[] arg) {
    ArrayFormat m = new ArrayFormat( 10 );
    m.setValue( 1 , 6 );
    m.setValue( 0 , 8 );
    m.setValue( 3 , 1 );
    m.setValue( 8 , 4 );
    System.out.println("This is the array in 'standard' format");
    m.setDisplayFormat( new StandardFormat() );
    m.display();
    System.out.println("This is the array in 'math' format:");
    m.setDisplayFormat( new MathFormat() );
    m.display();
  }
}
```

Decorator: Strutturale

Per spiegare questo pattern, riprendiamo lo stesso esempio con cui abbiamo introdotto lo Strategy Pattern. Supponiamo che OrdineDiVendita deleghi la responsabilità di stampare una ricevuta a RicevutaDiVendita.

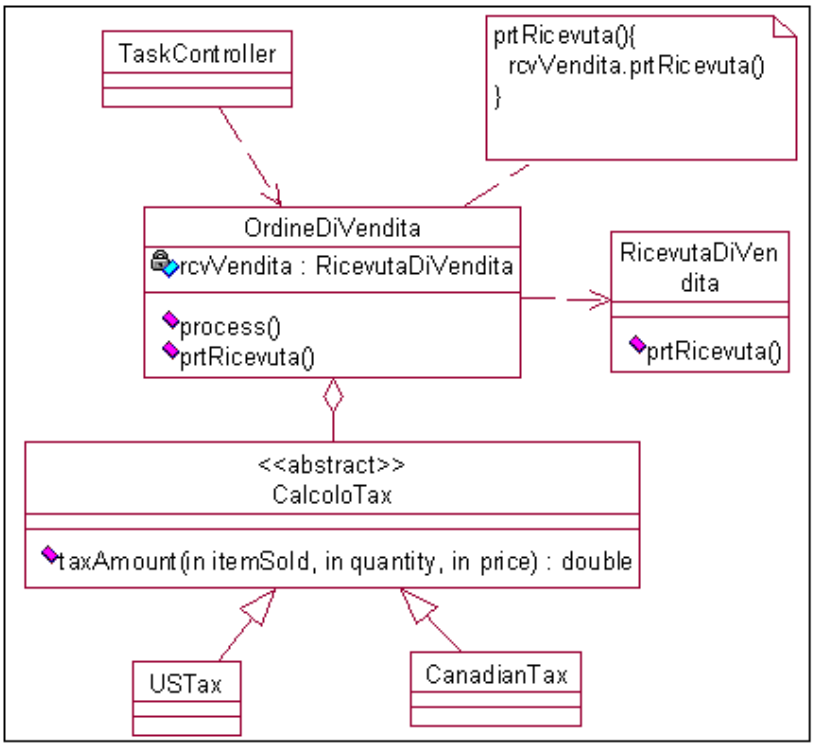


Fig. 34

Variazione del requisito

Supponiamo che il requisito possa variare; ad esempio si prevede già che occorrerà aggiungere un header alle informazioni e un piè pagina alla stampa (ptrRicevute())che fa la classe RicevutaDiVendita!!

Una prima idea per gestire la variazione è di far in modo che RicevutaDiVendita faccia uso di due altri oggetti Header e Footer ognuno con i propri metodi per stampare la ricevuta.

In questo caso il controllo è nelle mani di RicevutaDiVendita che sceglie chi chiamare: prtHeader() o prtFooter().

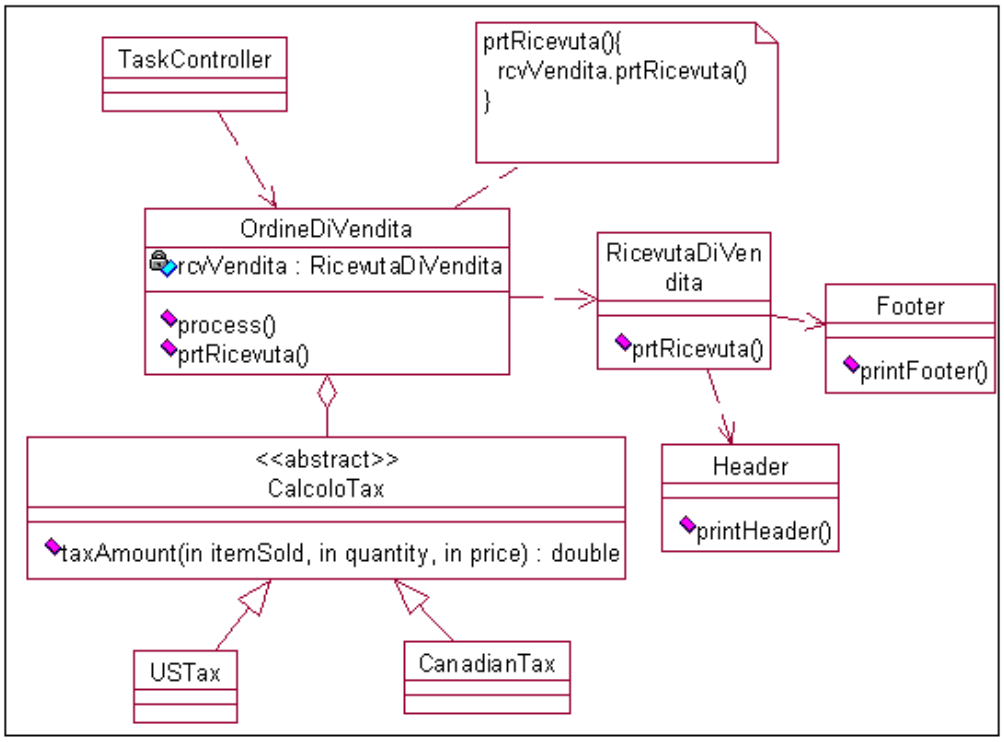


Fig. 35

La soluzione di figura 35 funziona, ma è poco flessibile: va bene fin quando le scelte sono basse di numero e se l'header o il piè pagina sono fissi e non cambiano mai. In altri termini posso stampare un header fisso e il piè pagina fisso.

Ma se ho bisogno di più header? Cioè posso voler scegliere tra N header diversi o tra N Footer diversi, stampandone uno solo per volta?

In questo caso potrei adottare lo Strategy Pattern e me la caverei. In fondo vado a scegliere uno tra gli N algoritmi a disposizione!

Ma se invece devo stampare più di un header e/o di un footer alla volta? Ecco in aiuto il Decorator Pattern.

Il Decorator Pattern anzicchè controllare le funzionalità aggiunte, come fa lo Strategy Pattern, controlla il "chaining" delle funzionalità desiderate nell'ordine che ci serve.

L'intento del Decorator

GoF definisce il Decorator Pattern nel seguente modo:"Attacca responsabilità addizionali ad un oggetto dinamicamente. Il Decorator fornisce una flessibile alternativa al subclassing per stendere delle funzionalità".

In altri termini il Decorator Pattern è una catena di oggetti che parte con un Component (ConcreteComponent o Decorator). Poi ogni Decorator è seguito da un altro Decorator o dall'originale ConcreteComponent. Alla fine della catena vi è sempre un ConcreteComponent.

Esaminiamo la figura 36 successiva.

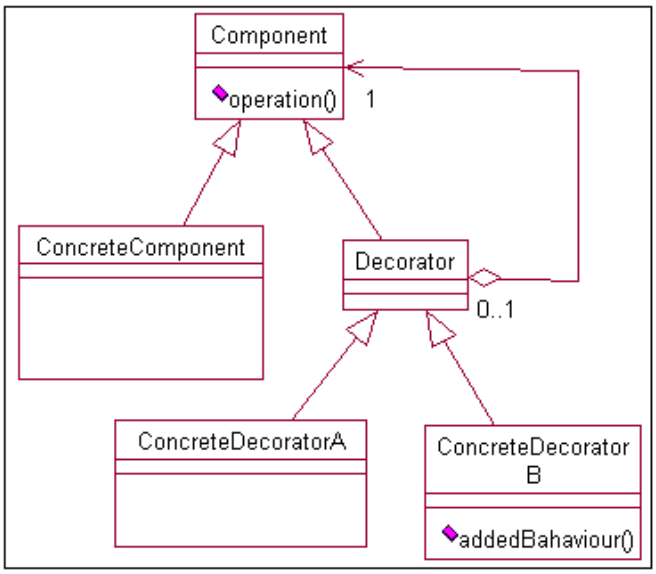


Fig. 36

La soluzione del problema

Sfruttiamo il Decorator Pattern per giungere alla nostra soluzione. Il ConcreteComponent è RicevutaOrdine, il Decorator è RicevutaDecorator, il ConcreteDecoratorA è HeaderDecorator, il ConcreteDecoratorB è FooterDecorator.

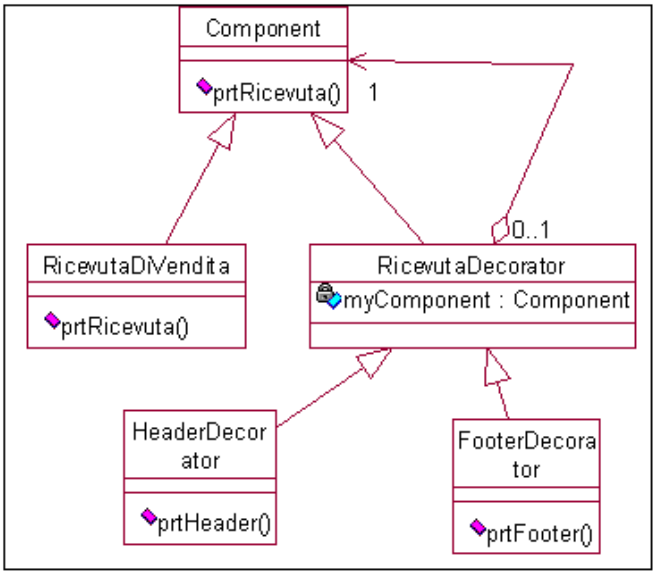


Fig. 37

Come lavora il Decorator

Ogni Decorator va a wrappare le nuove funzionalità attraverso una scia di altri oggetti.

Il Decorator aiuta a decomporre il problema in due parti:

- Come implementare gli oggetti che devono fornire le nuove funzionalità
- Come organizzare gli oggetti per ogni caso particolare

Esercizio #1

Supponiamo di dover modellare gli impiegati di una azienda (Employee). Tra tali impiegati esistono gli Ingegneri (Engineer) che implementano le operazioni definite per gli impiegati. Il sistema deve prevedere la possibilità di investire gli impiegati di responsabilità aggiuntive; ad esempio possono assumere ruolo di Administrative Manager e/o di Project Manager. I due ruoli non si escutono a vicenda. Tali ruoli per ogni impiegato possono variare dinamicamente, con possibilità non solo di aggiungerli ma anche di toglierli ad ogni impiegato. La figura 38 mostra la soluzione al problema.

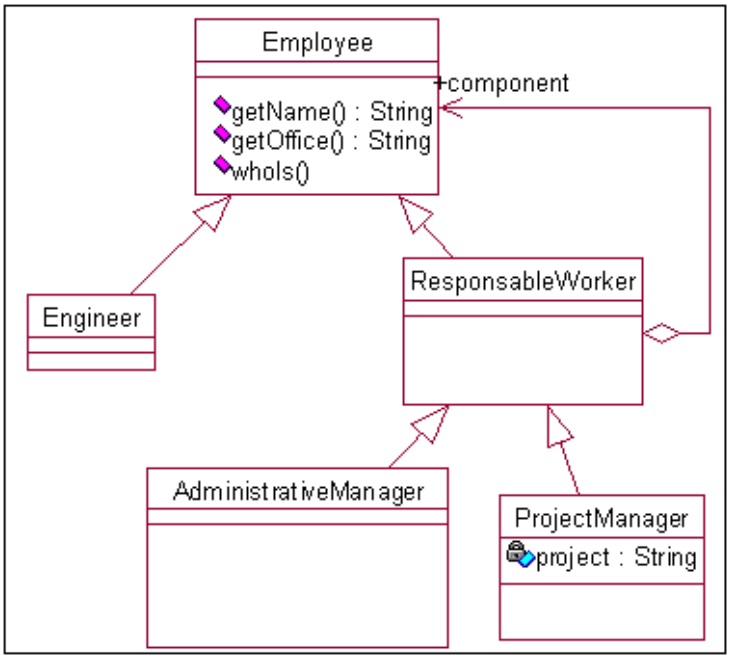


Fig. 38

Frammento di codice Java dell'esercizio

```
public interface Employee {
  public String getName();
  public String getOffice();
  public int getId();
}

public class Engineer implements Employee {
  private String name, office;
  private int     key;
  public Engineer( String nam, String off) {
    name = nam;
    office = off;
  }

  public String getName() {
    return name ;
  }
```

```
  public String getOffice() {
    return office ;
  }
  public void whoIs () {
    System.out.println( "I am " + getName() + ", and I am with the " + getOffice() +".");
  }
}

abstract class ResponsibleWorker implements Employee {
  protected Employee responsible;

  public ResponsibleWorker(Employee employee) {
    responsible = employee;
  }

  public String getName() {
    return responsible.getName();
  }

  public String getOffice() {
    return responsible.getOffice();
  }

  public void whoIs () {
    responsible.whoIs();
  }
}


public class AdministrativeManager extends ResponsibleWorker {
   public AdministrativeManager( Employee empl ) {
     super( empl );
   }

   public void whoIs() {
     sayIamBoss();
     super.whoIs();
   }

private void sayIamBoss(){
   System.out.print( "I am a boss. " );
}
}


public class ProjectManager extends ResponsibleWorker {
  private String project;
  public ProjectManager( Employee empl, String proj ) {
    super( empl );
    project = proj;
 }

public void whoIs() {
  super.whoIs();
  System.out.println( "I am the Manager of the Project:" + project );
}
}
```

Esercizio #2

Un altro uso utile del Decorator è ei sincronizzare un oggetto costruito originalmente per essere utilizzato in un ambiente single-threading, e lo si vuole portare in un ambiente d'esecuzione multi-threading.

Supponiamo di avere l'interfaccia DiagonalDraggablePoint (Component) che fornisce la firma dei metodi per muovere un punto diagonalmente una distanza specificata, e per scrivere la posizione attuale.

Vediamo il codice Java.

Frammento di codice Java

```
public interface DiagonalDraggablePoint {
  public void moveDiagonal( int distance, String draggerName );
  public void currentPosition( );
}
public class SequentialPoint implements DiagonalDraggablePoint {
  private int x, y;

  public SequentialPoint( ) {
    this.x = 0;
    this.y = 0;
  }

  public void moveDiagonal( int distance, String draggerName ) {
    int aux = x + distance ;
    System.out.println( "Moved by " + draggerName + " - Origin x=" + x + " y=" + y );
    x = aux;
    y = y + distance;
 }

 public void currentPosition( ) {
   System.out.println( "Current position : x=" + x + " y=" + y );
 }
}
```

In una esecuzione multi-threaded il codice di sopra ha il problema che non blocca l'oggetto nel processo di movimento (moveDiagonal), così che se il thread in esecuzione è interrotto in questa fase, alcune delle variabili potrebbero essere aggiornate da un altro thread, prima che il controllo ritorna, ottenendosi come risultato valori diversi di x e y. Il problema si può risolvere creando un Decorator (SynchronizedPoint) che implementi l'interfaccia DiagonalDraggablePoint e che sincronizzi il codice rischioso.

```
public class SynchronizedPoint implements DiagonalDraggablePoint {
  DiagonalDraggablePoint theSequentialPoint;

  public SynchronizedPoint(DiagonalDraggablePoint np) {
    theSequentialPoint = np;
  }

  public void moveDiagonal( int distance, String draggerName ) {
    synchronized(theSequentialPoint) {
    theSequentialPoint.moveDiagonal( distance, draggerName );
    }
  }
  public void currentPosition( ) {
   theSequentialPoint.currentPosition();
  }
}
```

```
public class DecoratorExample2 {
  public static void main( String[] arg ) {
    System.out.println( "Non synchronized point:" );
    DiagonalDraggablePoint p = new SequentialPoint();
    PointDragger mp1 = new PointDragger( p, "Thread 1" );
    PointDragger mp2 = new PointDragger( p, "Thread 2" );
    Thread t1 = new Thread( mp1 );
    Thread t2 = new Thread( mp2 );
    t1.start();
    t2.start();
    while( t1.isAlive() || t2.isAlive() );
    p.currentPosition();
    System.out.println( "Synchronized point:" );
    p = new SynchronizedPoint( new SequentialPoint() );
    mp1 = new PointDragger( p, "Thread 1" );
    mp2 = new PointDragger( p, "Thread 2" );
    t1 = new Thread( mp1 );
    t2 = new Thread( mp2 );
    t1.start();
    t2.start();
    while( t1.isAlive() || t2.isAlive() );
    p.currentPosition();
 }
}

class PointDragger implements Runnable {
  DiagonalDraggablePoint point;
  String name ;
  public PointDragger( DiagonalDraggablePoint p, String nom ) {
    point = p;
    name = nom;
  }

  public void run() {
    for( int i=1; i < 5; i++ ) {
      point.moveDiagonal( 1, name );
    }
  }
}
```

Singleton: Comportamentale

GoF definisce l'intento del Singleton Pattern nel seguente modo: "Esso assicura che la classe abbia una ed una sola istanza e fornisce un punto globale di accesso ad essa".

Come lavora il Singleton

Esso procede nel seguente modo:

- Quando il metodo è chiamato, esso verifica se l'oggetto è già stato istanziato.Se esso lo è, ritorna solo un reference all'oggetto, altrimenti lo istanzia e ritorna il reference all'oggetto.
- Per assicurare che questa sia l'unica strada per istanziare un oggetto di tal tipo, definisco un costruttore di questa classe protetto o privato.

Esempio

Riprendiamo l'esempio CalcoloTax dello Strategy Pattern. Poiché CalcoloTax è sempre lo stesso calcolo per tutti coloro che lo richiameranno, allora, per ragioni di performance e per risparmio risorse, è

candidato ad essere istanziabile una ed una sola volta. Questo per evitare che cresca smisuratamente il numero di Strategy Pattern in gioco.

Frammento codice Java

```
class USTax {

   private static USTax istance;   // variabile statica

   private USTax();    // Costruttore privato

   public static USTax getInstance () {
      if( istance == null){
         istance = new USTax();
      }
      return istance;
   }
}
```

Altri esempi di Singleton

Ecco altri due esempi dove può trovare applicazione il Singleton Pattern:

- La gestione della connessione ad un database, in modo da averla sempre aperta;
- La gestione della concorrenza su una risorsa condivisa sul filesystem, in modo che una sola richiesta per volta possa essere fatta ({sequential}).

Double-Checked Locking: Comportamentale

Il Singleton Pattern è in grado di risolvermi quella proprietà che è applicabile ai metodi che in UML vanno segnalati con {sequential}, cioè il Singleton Pattern va bene per una situazione di progettazione non concorrente.

Non è in grado, invece, di affrontare proprietà del tipo {concurrent} o meglio ancora {guarded} che sono tipiche di ambienti concorrenti, multi-thread per l'esecuzione parallela.

Il Singleton Pattern presenta difetti in un ambiente multi-thread:

- Due chiamate parallele ad esso se entrambe nello stesso istante stabiliscono che la variabile istance è null, entrambe creeranno un oggetto USTax.
- Se il Singleton è stateless non crea problemi, ma se deve essere statefull (pensiamo al contante su un account bancario o ad un contatore) è già problematico. In tal caso dovrebbe essere garantita una proprietà {guarded} sul metodo che agisce sullo stato.
- La creazione di oggetti in più non è un problema in Java: viene occupata più memoria ma la JVM, sempre che non avvenga un crash, provvederà a liberare successivamente le parti non usate. In C++ tale occupazione di memoria conduce a situazioni di memory leak.

Il Singleton Pattern in un ambiente multi-thread può condurre all'errore sottile di duplicazioni di stato: ad esempio avremo due contatori su cui sono suddivisi i calcoli. Non si saprà mai chi si incrementa e chi si legge, né è detto che i calcoli sono equamente suddivisi. Un tale errore è difficile da scovare.

La soluzione potrebbe sembrare quella di sincronizzare i thread sul test. Questa soluzione semplice è effettuabile nella misura in cui tutto ciò non diventa un collo di bottiglia (bottleneck), cioè quando il numero di thread concorrenti sul test è basso. In Java tutto ciò è implementabile con la keyword synchronized.

Vediamo la soluzione.

Frammento codice Java

```
class USTax {

  private static USTax instance;   // variabile statica

    private USTax();    // Costruttore privato

    private synchronized static void doSync(){
      if(istance == null) instance = new USTax();
    }

    public static USTax getInstance () {
      if( ustance == null){
         doSync();
      }
      return istance;
    }
}
```

Observer: Comportamentale

Per il sistema di e-commerce precedentemente visto, supponiamo di avere l'esigenza che, una volta un cliente si sia registrato, di dover:

- inviargli una e-mail di welcome quando ha acquistato qualcosa;
- verificare l'indirizzo del cliente con l'ufficio postale.

Se fossi assolutamente certo che il requisito non variasse potrei risolvere il problema nel modo di figura 39.

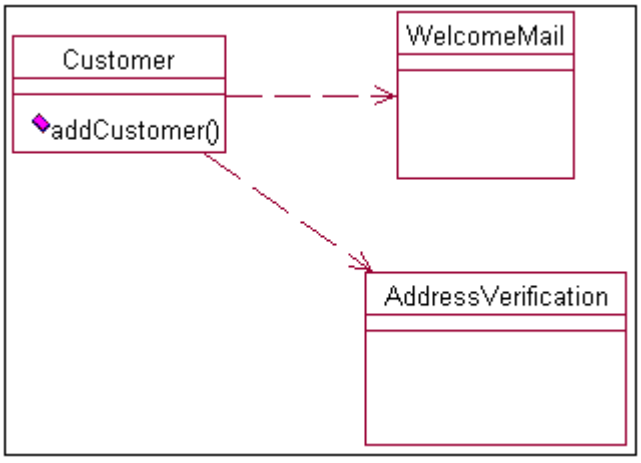


Fig. 39

Purtroppo il requisito cambia prima o poi!!

Come sempre, la certezza assoluta che il requisito non possa cambiare non esiste e non esisterà mai.

Nell'esempio proposto la cosa molto probabile è che ci potrà essere una variazione di indirizzo del cliente nel tempo, o dell'email etc.

L'intento dell'Observer

GoF dice che: "Definisce dipendenze (dinamiche) una-a-molti tra oggetti, così che quando un oggetto cambia stato tutti gli altri oggetti dipendenti sono notificati e updatati automaticamente".

In altri termini permette il Pattern noto anche come Publish/Subscribe.

In Java è già disponibile l'interface Observer e la Observable class.

Step 1: Identifico gli observer e faccio in modo che abbiano la stessa interfaccia

Gli oggetti che, al verificarsi di un evento, devono ricevere le notifiche sono detti *observer* (osservatori); mentre l'oggetto che triggera l'evento è il *subject*. Per cui in generale posso avere più observer che faranno capo ad uno stesso subject.

Bisogna fare in modo però che gli observer abbiano la stessa interfaccia di notifica, altrimenti saremo costretti a modificare il subject, aggiungendo metodi per ognuno degli observer e ciò sarebbe sfavorevole.

Se gli observer hanno la stessa interfaccia il subject riesce a notificare l'evento facilmente a tutti e non sarà più modificato. Eventi diversi, in generale, dovrebbero richiedere subject diversi.

Step 2: Registro tutti gli Observer al Subject

Il Subject deve avere due metodi:

- attach(Observer), che permette ad un Observer di sottoscriversi al Subject;
- detach(Observer), che permette di staccare l'Observer dal Subject

Step 3: Invio la notifica agli Observer quanto l'evento si verifica

L'Observer deve avere un metodo update. Il Subject ha il metodo notify che usa i metodi update degli observer. Il metodo update deve contenere il codice per gestire l'evento.

Step 4: Ottenere informazioni sull'evento

Notificare l'evento spesso non basta. Potrebbe essere necessario avere le informazioni dal Subject. Questo significa che il Subject deve avere ulteriori metodi di getXXX; ad esempio getState(), getAddress(), etc.

La soluzione al problema

Il Customer, per come abbiamo formulato il problema, è un subject da tenere sotto osservazione, sia per verificare quando un utente si collega la prima volta (e solo la prima volta), sia se varia il suo stato nel tempo. Teniamo presenti i quattro step, di cui prima e vediamo il modello che nasce.

Implementiamo il Customer (subjet) con un Observer Pattern.

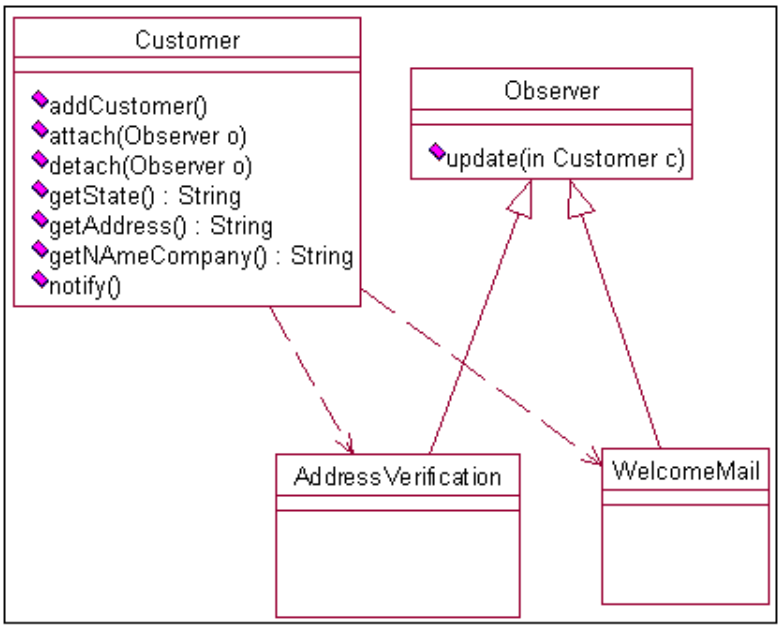


Fig. 40

Le sottoclassi dell'Observer sono, fondamentalmente, gli eventi del subject a cui si è interessati.

Metodi statici

I metodi *attach e detach vanno resi statici* perché gli Observer sono interessati a tutti i Customer in gioco.

Durante la notifica viene attivato il metodo update che prende in input il Customer.

Il vantaggio della soluzione è che posso aggiungere flessibilmente Observer senza influire su esistenti classi. In altri termini il grado di disaccoppiamento è alto.

Se cambia ancora requisito ? Se devo ad esempio inviare anche dei coupon? Aggiungo sotto l'Observer le nuove classi, come in figura 41.

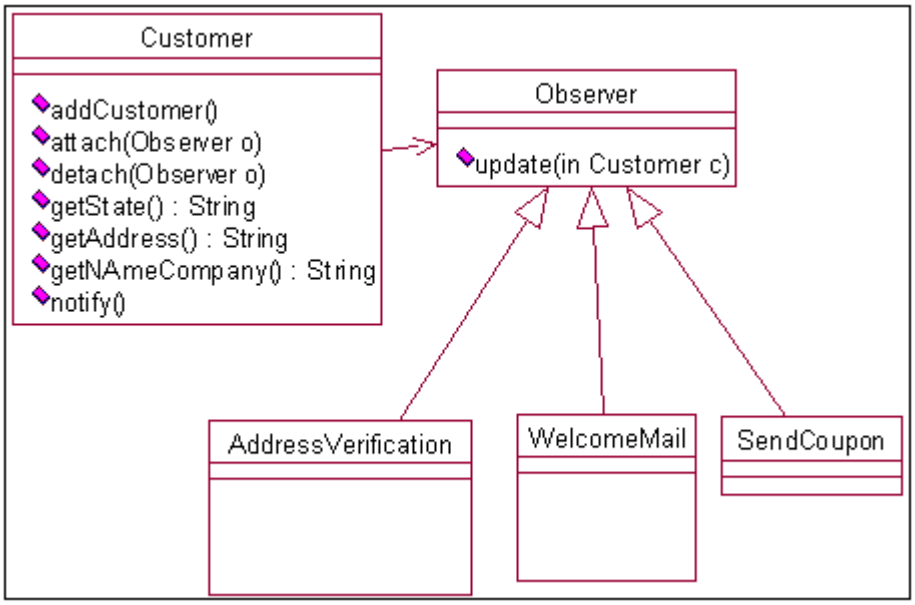


Fig. 41

Il simbolo di dipendenza tra Customer e le sottoclassi di Observer si può omettere perché la dipendenza, come vedremo nel frammento di codice Java, avviene attraverso il metodo update.

L'Observer nella realtà

Non sempre tutti gli oggetti hanno la stessa interfaccia, per cui è facile vedere l'Observer Pattern accoppiato anche all'Adapter Pattern come in figura 42.

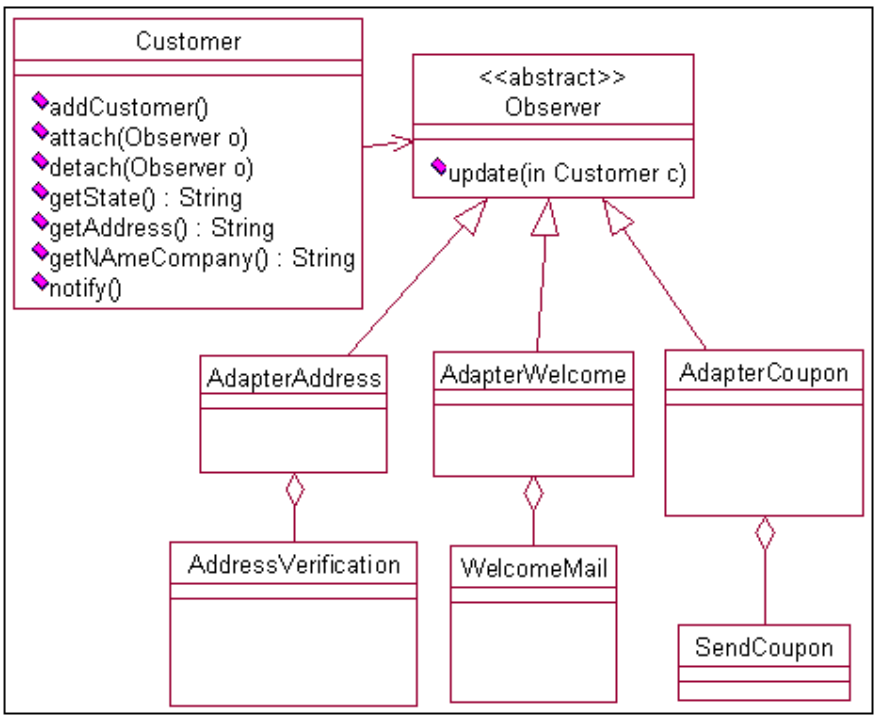


Fig. 42

Frammento codice Java

```
class Customer {

    private static Vector myObs;

    private USTax();     // Costruttore privato

      static {   // esiste un solo vettore per tutti i Customer
        myObs = new Vector();
      }
      public static void attach(Observer o) {
        myObs.addElement(o);
      }
      public static void detach(Observer o) {
        myObs.remove(o);
      }

      public String getState() {
        // viene ritornato il valore dell'attributo
      }
      public void notify() {
        for (Enumeration e = myObs.elements(); e.hasMoreElements(); ){
          ((Observer) e).update(this);
        }
      }
}

abstract class Observer {
```

```
   public Observer(){
      Customer.attach(this);
   }
   abstract public void update(Customer myCust);
}

class AddressVerification extends Observer{
   public AddressVerification(){
      super();
   }
   public void update(){
      // magari va a consultare l'address del Customer e lo setta sul DB
   }
}

class WelcomeMail extends Observer{
   public WelcomeMail(){
      super();
   }
   public void update(){
      // magari va a consultare la e-mail del Customer e la varia sul DB
      //
   }
}
```

Esercizio #1

Un oggetto riceve dei numeri e casualmente cambia il suo stato interno, memorizzandovi il numero.

Altri due oggetti sono incaricati del monitoraggio dell'oggetto di cui sopra e sono interessati alla variazione di stato di esso.

Una soluzione è offerta dall'Observer Pattern (In Java ad esempio è offerto da `java.util.Observable`).

In figura 43 la soluzione. Provata a scrivere voi il frammento di codice Java.

Il suggerimento è che:

- quando viene chiamato il metodo receiveValue() della sottoclasse in esso avviene sia il setChanged() che il notify()
- quando viene chiamato il notify() avviene l'update()
- quando avviene l'update() avviene poi il returnValue().

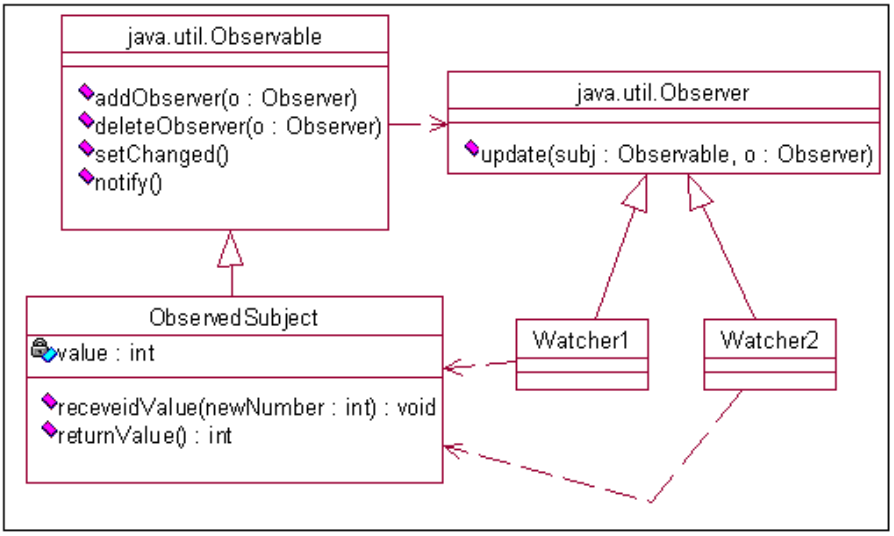


Fig. 43

Esercizio #2

Ecco un esercizio lasciato al lettore. Supponiamo che una banca abbia più sportelli bancomat (ATM). Ogni sportello ATM ha una tastiera, un video/display, una stampante. Ogni ATM ha un menù permette il prelievo di contanti, il trasferimento di soldi da un account ad un altro, il saldo, la stampa dello scontrino. Gli ATM sono connessi via WAN ad un server.

Un cliente deve inserire nell'ATM la carta bancomat e autenticarsi con una password, che viene associata dall'ATM al PIN della carta bancomat. La carta bancomat è caratterizzata, infatti, da un PIN e da una data di data di scadenza.

Il generico sportello ATM dispone di una somma in contanti pari a 10.000 euro.

L'operatore della banca deve essere avvisato dal sistema software quando:
- si verifica che il contante dello sportello ATM diventa inferiore a 1000 euro
- si verifica che una carta è stata trattenuta perché la data è scaduta
- si verifica che il cliente ha digitato 3 volte una password errata e la carta è stata trattenuta

Occorre, poi, tener presente che i trasferimenti di somme non producono diminuzione di contante nel bancomat e che l'ATM non permette prelievi se la somma disponibile è inferiore a quella richiesta dal cliente.

Poiché si tratta di una WAN e c'è anche l'acceso da INTERNET alla banca, gli account dei clienti possono essere non solo movimentati da bancomat ma anche da remoto via INTERNET.

Suggerimento 1: l'operatore è interessato a tre eventi (Observer Pattern)
Suggerimento 2: occorre garantirsi dalla concorrenza sugli account, possibile da ATM e INTERNET (Double Checked Locking Pattern) e sulla somma totale disponibile nell'ATM.

Template Method: Creazionale

Riprendiamo l'esempio dell'e-commerce, visto prima. Supponiamo che per motivi architetturali dobbiamo supportare sia un database Oracle che MySQL (o qualsiasi altra cosa per l'esempio).

In generale dovremo fare diversi step:
1. formare la stringa di comando della CONNECT
2. inviare la CONNECT al database
3. formare la stringa della select
4. inviare la select al Database
5. ritornare i dati selezionati

L'intento del Template Method

La definizione data da GoF è: "Definisce lo scheletro di un algoritmo in una operazione, differendo alcuni step alle sottoclassi. Ridefinisce gli step in un algoritmo senza cambiare la struttura dell'algoritmo".

In altri termini nel nostro esempio vi sono *diversi metodi* per connettersi ai database, ma concettualmente i metodi sono lo stesso processo (fanno la stessa cosa in modo diverso); difatti i metodi possono essere diversi per la modalità di fare CONNECT e per la SELECT ad esempio.

Per cui il Template Method Pattern ci dà una via per catturare le parti comuni non variabile (Commonality Analysis) attraverso una classe astratta; mentre le differenze saranno presenti nelle sottoclassi.

La soluzione

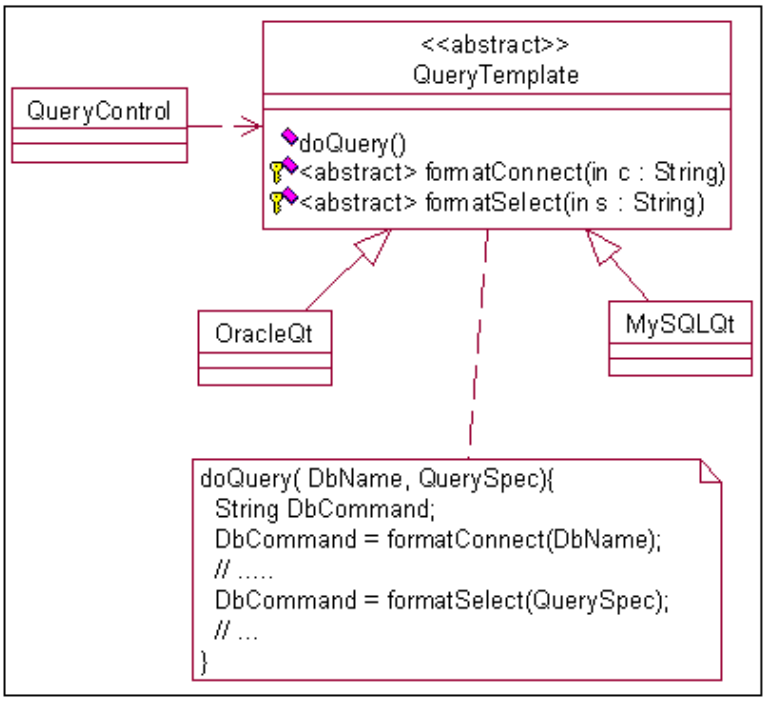


Fig. 44

Il Template Method è, quindi, applicabile quando vi sono differenze di metodo ma processi concettuali simili.

Altri esempi

- Creazione di framework
- Creazione e gestione di Task diversi in Workflow

Factory Method: Creazionale

Supponiamo di continuare l'esempio del Template Method Pattern. Nel Template Method Pattern QueryControl deve conoscere le sottoclassi da usare. Questa non è una buona strategia per un client. Basta usare una Factory per evitare questo!

Introduco nel QueryTemplate un nuovo metodo makeDB() che si legge da configurazione il tipo di oggetto da istanziare.

L'intento del Factory Method

GoF dice:"Definisce una interfaccia per creare un oggetto, ma lascia alle sottoclassi il compito di decidere quale classe istanziare. Il Factory Method lascia ad una classe di differire la creazione alle sottoclassi".

La soluzione

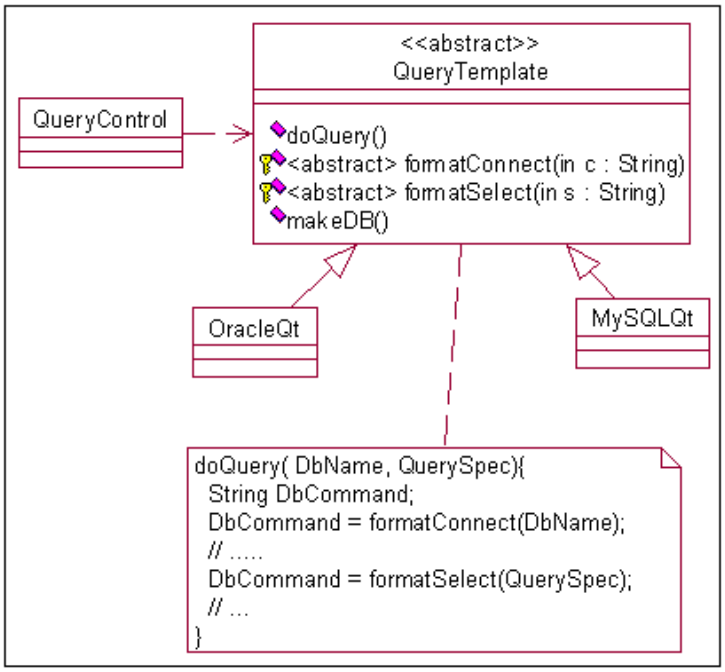


Fig. 45

Builder: Creazionale

L'intento del Builder

Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione sia adatto a costruire diverse rappresentazioni.

Esempio

Si debba sviluppare un tool in grado di fare la rappresentazione ER (Entity Relationship) di un database.

Oggi nel settore dei database esistono varie notazioni, tra cui anche quella UML. Ad esempio per una relazione del tipo: "Uno studente studia in 0 o 1 università" si possono usare: notazione ER classica o UML. La prima notazione dà luogo ad un *modello non orientato* e la seconda ad un *modello orientato*.

In entrambe le rappresentazioni, i concetti sono gli stessi. Supponiamo che si vogliano gestire entrambe le tipologie di diagrammi e di produrre anche i modelli in supporti di natura diversa (ad esempio testo o XML).

Il problema presenta una rappresentazione di base comune (il processo di rappresentazione) di entrambi i modelli, ma una costruzione diversa perché entrambi i modelli usano notazioni diverse.

In altri termini siamo di fronte ad una situazione per la quale dobbiamo ottenere prodotti diversi ma con un processo che è lo stesso.

Lo schema

Il Builder consente di separare la logica del processo di costruzione dalla costruzione stessa. Per ottenere questo si utilizza un oggetto *Director* che si occupa della logica di costruzione e che invia ad un oggetto *Builder* le istruzioni per la creazione del prodotto, come in figura 46.

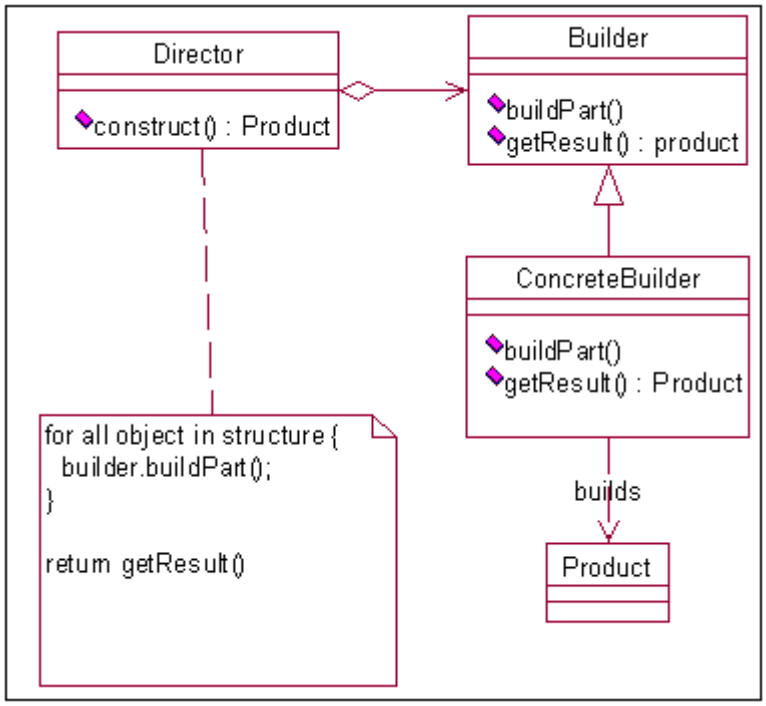


Fig. 46

Poiché i prodotti da costruire sono diversi e di diversa natura, esisteranno più Builder ma un solo Director.

La soluzione dell'esempio

Il problema richiede di costruire modelli diversi partendo concettualmente da uno stesso processo. Nella costruzione del modello, in uno dei due modi (non orientato o orientato), si può omettere qualcosa. Ad esempio nel modello ER non orientato (l'ER classico) NON devo riportare la molteplicità e posso omettere il nome della relazione. La soluzione è quindi in figura 47.

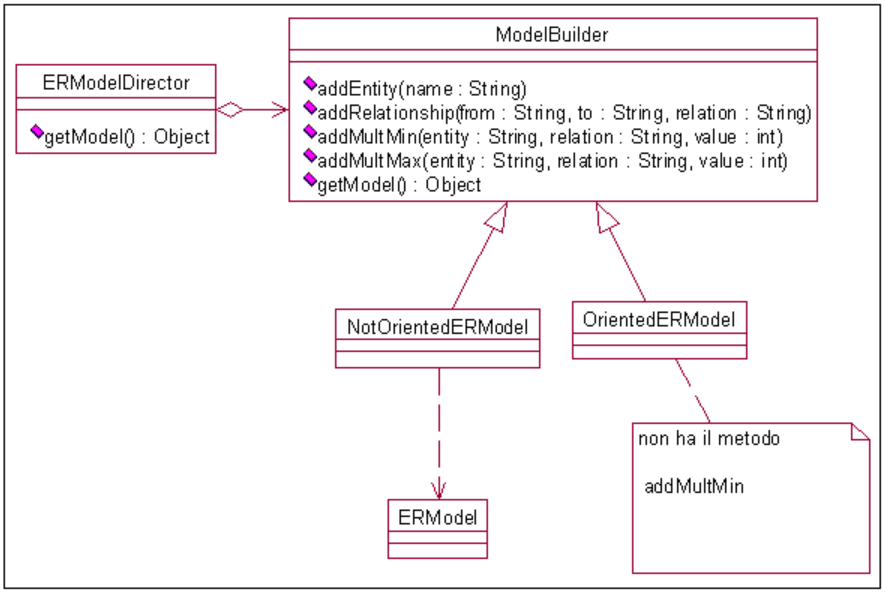


Fig. 47

Composite: Strutturale

Consente la costruzione di gerarchia di oggetti composti.

Esempio

Supponiamo di dover rappresentare una gerarchia tipo albero, dove un computer è costituito da: Main System, Monitor, Keyboard; mentre il Main System è costituito da Processor, Hard disk e RAM.

Il pattern Composite offre una classe astratta Component che deve essere estesa da due sottoclassi: una che rappresenta i singoli componenti (Leaf) e l'altra per i componenti composti (Composite), che si presenta come contenitore di altri componenti, sia singoli che contenitori.

In questo caso il modello del pattern Composite si presenta come in figura 48.

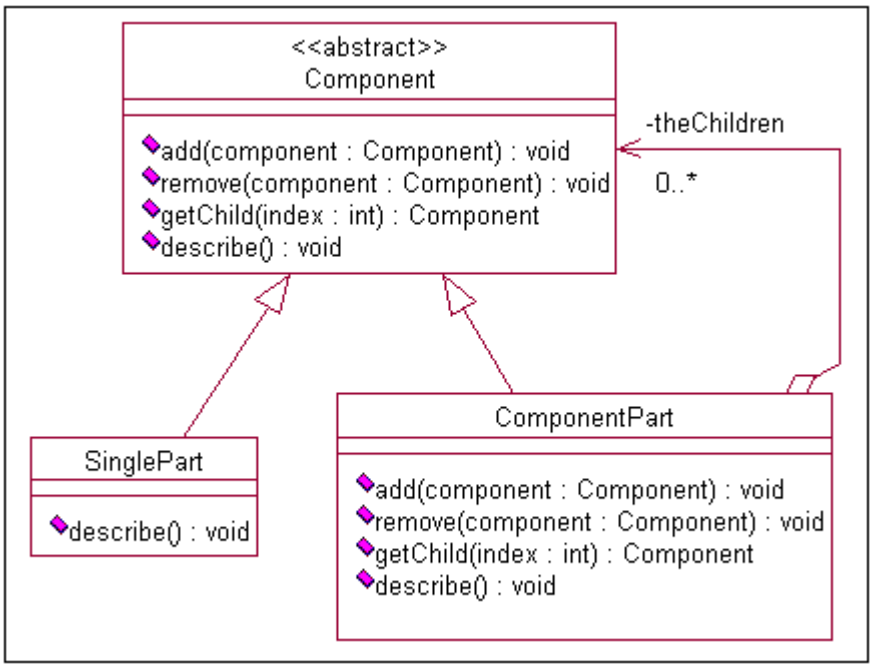


Fig. 48

Una soluzione software del problema col Composite

```
public abstract class Component {
public String name;
public Component(String aName){
name = aName;
}
public abstract void describe();

public void add(Component c) throws SinglePartException {
if (this instanceof SinglePart)
throw new SinglePartException( );
}
public void remove(Component c) throws SinglePartException{
if (this instanceof SinglePart)
throw new SinglePartException( );
}
public Component getChild(int n){
return null;
}
}
```

```
public class SinglePart extends Component {
public SinglePart(String aName) {
super(aName);
}
public void describe(){
System.out.println( "Component: " + name );
}
}

import java.util.Vector;
import java.util.Enumeration;
public class CompoundPart extends Component {
private Vector children ;
public CompoundPart(String aName) {
super(aName);
children = new Vector();
}
public void describe(){
System.out.println("Component: " + name);
System.out.println("Composed by:");
System.out.println("{");
int vLength = children.size();
for( int i=0; i< vLength ; i ++ ) {
Component c = (Component) children.get( i );
c.describe();
}
System.out.println("}");
}
public void add(Component c) throws SinglePartException {
children.addElement(c);
}

public void remove(Component c) throws SinglePartException{
children.removeElement(c);
}
public Component getChild(int n) {
return (Component)children.elementAt(n);
}
}
```

Aggiungiamo un'exception da alzare nel caso di richiamo metodi di gestione dei componenti vengano, invece, invocati su parti singole, cioè non ulteriormente composte.

```
class SinglePartException extends Exception {
public SinglePartException( ){
super( "Not supported method" );
}
}
```

Aggiungiamo un main di prova.

```
public class CompositeExample {
public static void main(String[] args) {
// Creates single parts
Component monitor = new SinglePart("LCD Monitor");
Component keyboard = new SinglePart("Italian Keyboard");
Component processor = new SinglePart("Pentium III Processor");
Component ram = new SinglePart("256 KB RAM");
Component hardDisk = new SinglePart("40 Gb Hard Disk");
// A composite with 3 leaves
```

```
Component mainSystem = new CompoundPart( "Main System" );
try {
mainSystem.add( processor );
mainSystem.add( ram );
mainSystem.add( hardDisk );
}
catch (SinglePartException e){
e.printStackTrace();
}
// A Composite compound by another Composite and one Leaf
Component computer = new CompoundPart("Computer");
try{
computer.add( monitor );
computer.add( keyboard );
computer.add( mainSystem );
}
catch (SinglePartException e){
e.printStackTrace();
}
System.out.println("**Tries to describe the 'monitor' component");
monitor.describe();
System.out.println("**Tries to describe the 'main system' component"
);
mainSystem.describe();
System.out.println("**Tries to describe the 'computer' component" );
computer.describe();
// Wrong: invocation of add() on a Leaf
System.out.println( "**Tries to add a component to a single part
(leaf) " );
try{
monitor.add( mainSystem );
}
catch (SinglePartException e){
e.printStackTrace();
}
}
}
```

Prototype: Creazionale

L'intento del Prototype

Specifica i tipi di oggetti da creare, utilizzando un'istanza prototipo e crea nuove istanze tramite la copia di tale prototipo.

Esempio

Supponiamo che si debba realizzare un editor per spartiti musicali, capace di scrivere e modificare note.

Si può pensare di realizzarlo con un editor già disponibile, capace di rappresentare elementi grafici, ma a cui si vuole aggiungere la possibilità di lavorare con le note ed i pentagrammi.

Supponiamo che esista una classe astratta Graphic che serve alla realizzazione di mezze note e note intere; mentre Tools ha una sottoclasse GraphicTool che crea istanze di elementi grafici. Il Prototype allora può aiutare in questo caso.

L'unica cosa che deve fare GraphicTool è che quando si devono creare "elementi grafici di tipo nota" vada a copiare o clonare un'istanza di una sottoclasse Graphic, che lo sa fare.

La soluzione

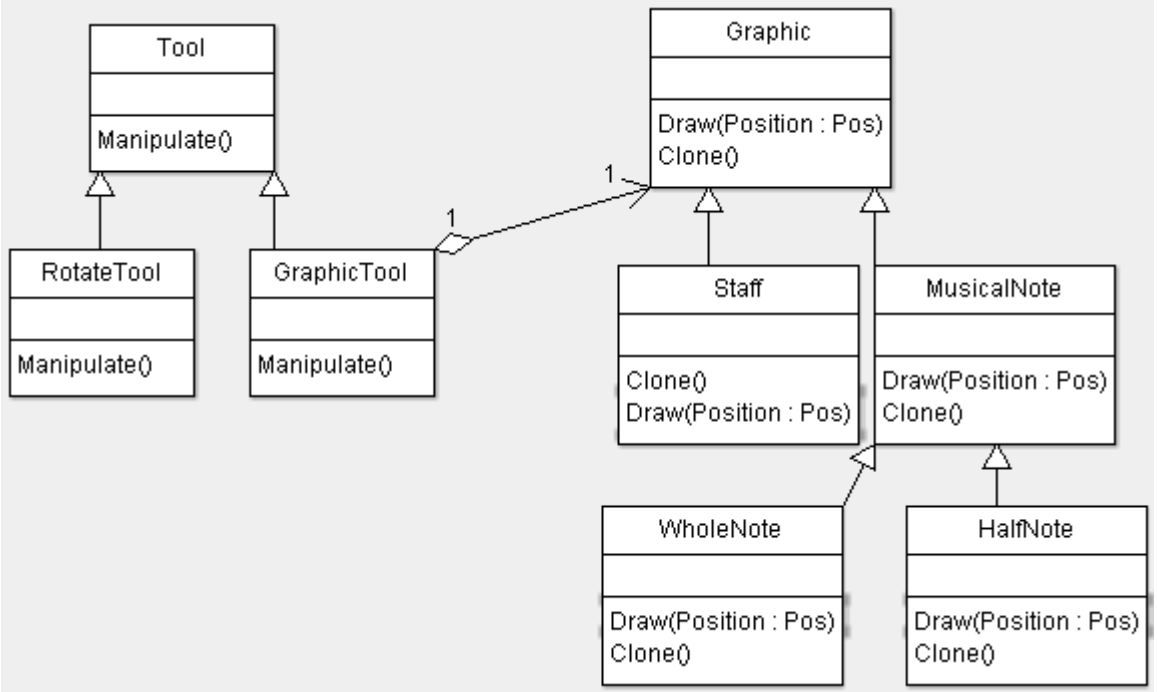


Fig. 49

Vanno sempre viste su un catalogo GoF le conseguenze a cui si va incontro nell'uso di un Prototype.

L'uso di un Prototype è consigliabile:

- per rendere un sistema indipendente dal modo in cui i suoi prodotti sono creati
- quando le istanze da istanziare sono note durante l'esecuzione
- per evitare una gerarchia di factory parallela a quella dei prodotti

Flyweight: Strutturale

L'intento del Flyweight

Definisce un meccanismo per condividere oggetti, in modo di fare un uso più efficiente delle risorse, particolarmente, l'utilizzo della memoria. In sostanza punta alla condivisione per supportare in modo efficiente un gran numero di oggetti a granularità fine.

Esempio

Supponiamo di dover progettare un editor di documenti a oggetti. Un oggetto testo è costituito da vari oggetti più piccoli: caratteri, righe, colonne. Una riga è un insieme di caratteri. In ogni caso un oggetto testo è costituito da un numero elevato di questi oggetti a granularità fine (basta pensare ai caratteri). Il Flyweight permette di condividere oggetti in modo da consentire il loro uso a granularità fine ma senza costo eccessivo. Vediamo come.

Un Flyweight è un oggetto condiviso che può essere usato in più contesti; in altri termini ha un significato o meglio uno stato diverso a seconda del contesto. Occorre distinguere tra uno stato interno (intrinseco) e uno stato esterno (estrinseco). Lo stato interno del flyweight è costituito da informazioni indipendenti dal contesto e, quindi, condivisibili; mentre lo stato esterno del flyweight dipende dal contesto e le informazioni non si possono condividere. E' il client responsabile a passare lo stato esterno al flyweight se necessario.

Con un flyweight possiamo rappresentare ogni carattere del documento dell'esempio dell'editor, oppure una sua riga che è costituita da un insieme di caratteri o anche una colonna che è un insieme di righe. Esiste un flyweight per ogni carattere, ma il flyweight appare anche nel contesto riga o nel contesto colonna.

La soluzione

Nel seguito supponiamo di operare con un editor di oggetti grafici/testuali. Glyph è una classe astratta per oggetti grafici, alcuni dei quali possono essere flyweight per l'analisi fatta precedentemente. Le operazioni che dipendono dallo stato esterno devono ricevere Glyph come parametro; cioè Draw e Intersects devono sapere in che contesto si trova il glyph prima di svolgere il loro lavoro.

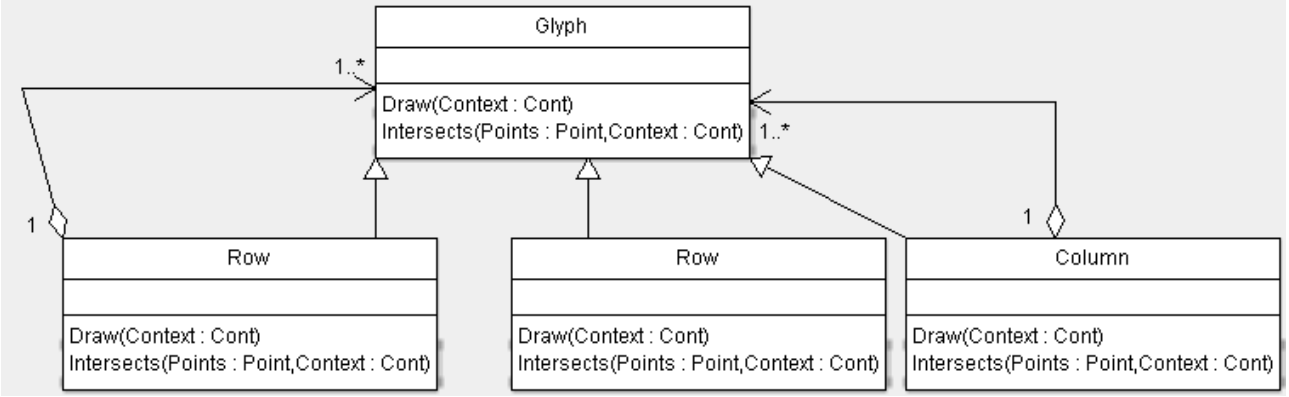


Fig. 50

Un flyweight che memorizza un carattere, memorizza solo il carattere e non la posizione o il font. I client danno invece le informazioni di contesto. Ad esempio un glyph di tipo riga sa le posizioni dei caratteri nella riga, per cui le passa ai figli carattere nella chiamata Draw. Se supponiamo che i font e la dimensione dei caratteri è sempre la stessa ed esistono ad esempio al massimo 100 tipologie di oggetti diversi (a volte molte di meno) qualunque sia la lunghezza del testo, allora tratteremo solo 100 oggetti flyweight.

Proxy: Strutturale

L'intento del Proxy

Fornisce una rappresentazione di un oggetto, un "surrogato" di accesso difficoltoso o che richiede un tempo importante per l'accesso o la creazione.

Il Proxy fornisce, cioè, un "oggetto surrogato" o un "segnaposto" in modo da consentire una apparente rapidità, poiché posticipa l'accesso o la creazione dell'oggetto solo al momento in cui è realmente necessaria.

Esempio

Pensate all'accesso ad un sito, dove viene rapidamente visualizzata la sua struttura con delle "immagini leggere riempitive", che hanno lo stesso comportamento o finalità di quelle reali, senza far attendere l'utente, per poi nel tempo sostituirle con le immagini reali definitive e più pesanti.

Consideriamo un programma di visualizzazione di testi e immagini e deve gestire informazioni che riguardano i file e dove file e testi vengono archiviati.

La soluzione

La soluzione dell'esempio è mostrata in figura 51.

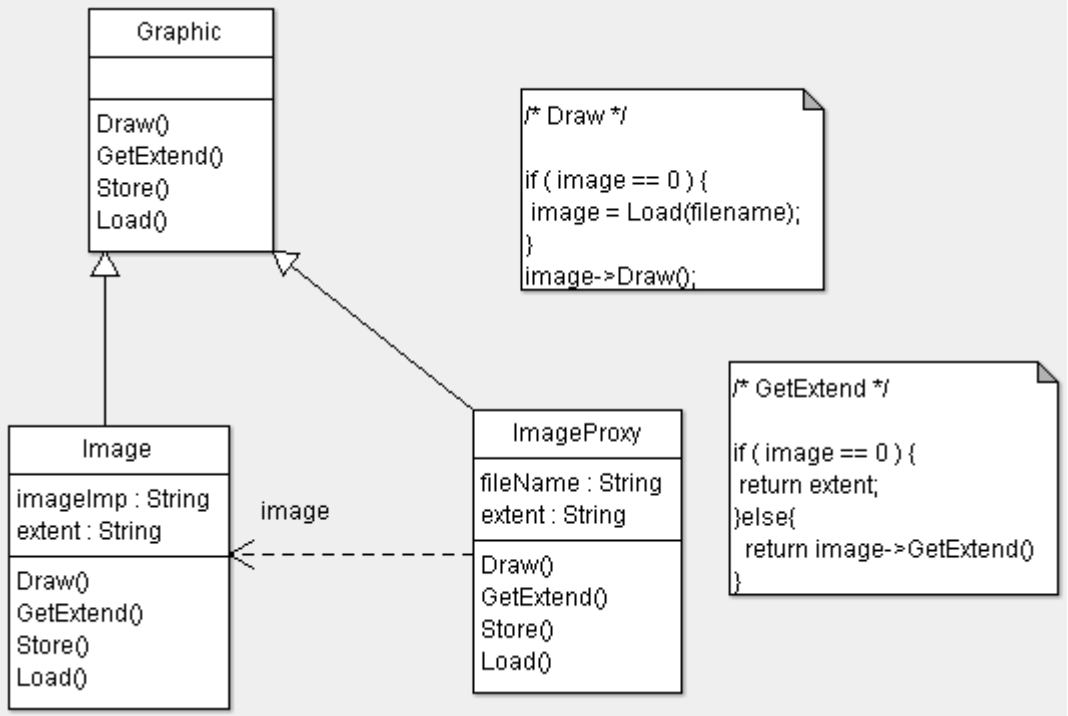


Fig. 51

Possiamo usare una classe astratta Graphic, padre di ImageProxy. ImageProxy ha il ruolo di Proxy e crea le immagini su richiesta ed ha il riferimento ai file immagine sul file system (filename è comprensivo di path ad esempio). Il riferimento può essere passato al costruttore di ImageProxy; inoltre mantiene anche un riferimento alle estensioni o riquadro dell'immagine. Il riferimento non è valido finché il Proxy non crea l'immagine vera e propria. L'operazione Draw in ImageProxy assicura che venga istanziata Image, prima che ne venga fatta richiesta, mentre GetExtend trasferisce la chiamata all'Image soltanto se è stata già istanziata, altrimenti restituisce l'estensione memorizzata in ImageProxy.

Alcune applicabilità del Proxy sono:

- Proxy per accesso a risorse costose (Proxy virtuale), come nell'esempio.
- Proxy per accesso di risorse da proteggere (Proxy di protezione)
- Proxy remoto, tipico di comunicazioni remote RMI. Coplen lo definisce "Proxy Ambasciatore".

Chain of Responsibility: Comportamentale

L'intento del Chain of Responsability

Evita l'accoppiamento tra la richiesta di un mittente ed un destinatario, consentendo che più di un oggetto possa soddisfare la richiesta.

Esempio

Tipicamente gli Help contestuali: otteniamo un help a seconda del contesto. Il primo oggetto della catena che riceve la richiesta la gestisce o la passa all'oggetto successivo, fino a che la richiesta non è soddisfatta.

La soluzione

Ad esempio potremmo avere una classe HelpHandler con un'operazione, che fa da superclasse di tutte le classi che potrebbero fornire l'help.

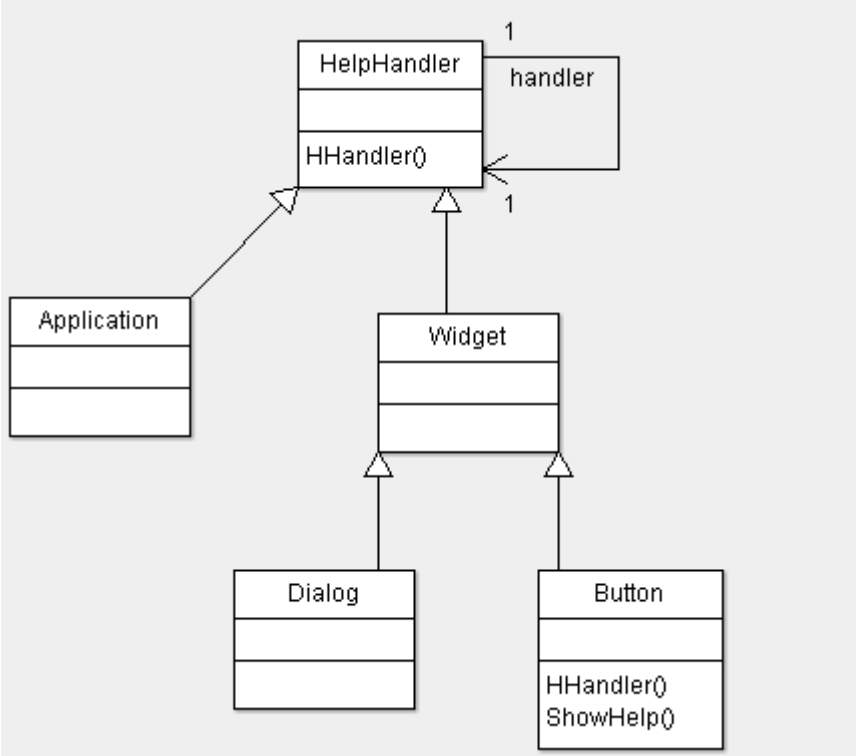


Fig. 52

Command: Comportamentale

L'intento del Command

Incapsula una richiesta in un oggetto e consente di parametrizzare i client con richieste diverse. In altri termini consente ai client di inoltrare richieste a oggetti sconosciuti dell'applicazione trasformando una richiesta in un oggetto.

Esempio

Supponiamo di avere una applicazione grafica (GUI), costituita da un document, menù e menuitem. I menu si possono implementare col Command. Un'applicazione, ad esempio, crea tali menu, i componenti del menu ed il resto della GUI. Il Command ha un metodo astratto Execute() che invoca la richiesta desiderata, mentre le sottoclassi concrete del Command destinatarie della richiesta mantengono un riferimento al mittente con un attributo e implementano il metodo Execute() in modo che invochi la richiesta desiderata. In altri termini sono i destinatari ad avere la conoscenza per portare a termine la richiesta. Ad ogni click sul menu, il Command fa eseguire l'azione all'oggetto giusto. L'applicazione, menu e sottomenu non conoscono nulla di cosa c'è dietro il Command o che cosa fa fare, mentre l'unica responsabilità dell'applicazione è di mantenere un riferimento ai documenti lavorati o lavorabili. Le sottoclassi di Command, possono fare, ad esempio, operazioni di past & cut su un documento, quindi, hanno un riferimento a Document. L'operazione Clicked esegue un command->Execute().

Nell'esempio di Fig. 53 abbiamo riportato solo Il PastCommand, ma si possono aggiungere tutte le sottoclassi necessarie al trattamento di tutti i click del mouse.

La soluzione

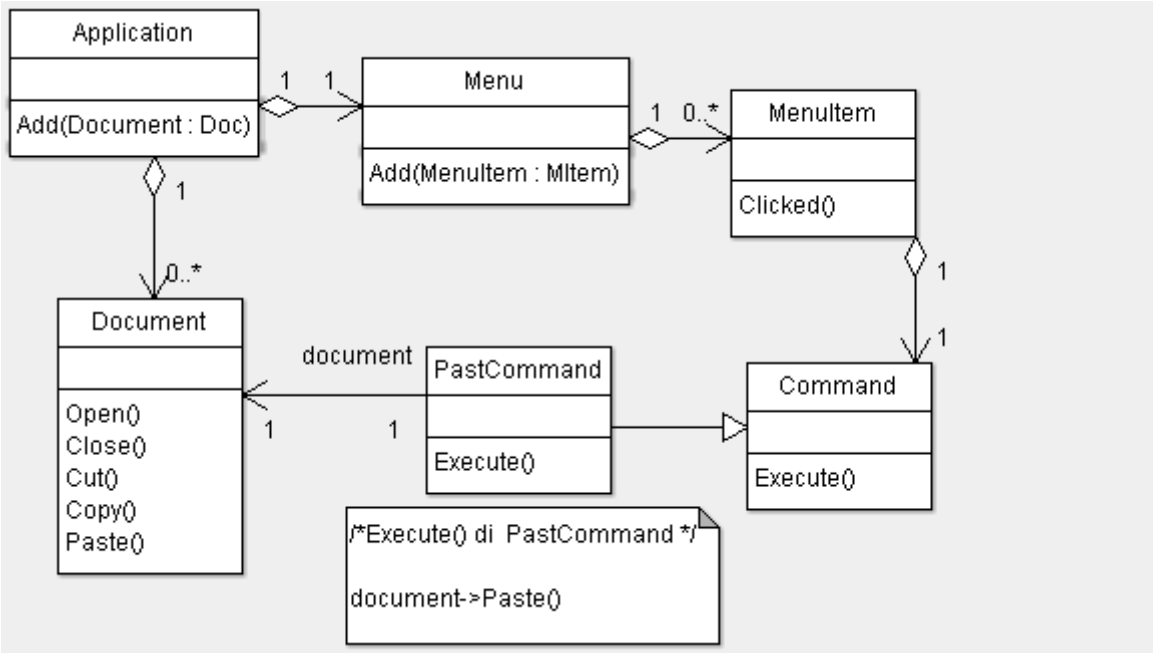


Fig. 53

Memento: Comportamentale

L'intento del Memento

Cattura e rende esterno lo stato interno di un oggetto, senza violarne l'incapsulamento, in modo tale da ripristinarne lo stato in un secondo momento. In altri termini permette di effettuare una istantanea di un oggetto.

Esempio

Utile ad esempio per creare dei sistemi di ripristino di una applicazione, dei savepoint in caso di crash o di recupero. In questo caso l'oggetto di cui si vuole salvare lo stato deve avere un riferimento al Memento (si dice originatore del Memento), lo deve creare, settare lo stato al momento giusto o in più momenti per un possibile recupero; in latri termini è l'originatore che gestisce e manipola il Memento.

Il Memento può essere usato anche per ricordare determinate situazioni esistenti tra due oggetti; ad esempio in un editor grafico, se l'utente disegna due rettangoli tra loro legati da una linea, essi devono continuare a rimanere legati anche se vengono spostati diversamente sul piano grafico.

Sempre in ambito grafico, lo stato precedente potrebbe servire per poter implementare le operazioni di undo. Ovviamente occorre porre molta attenzione nella implementazione delle operazioni di recupero, sul cosa si intende e l'effetto che si deve ottenere.

Spesso si usa in accoppiata con un Mediator per memorizzare un'istantanea (parziale o totale) e per poterla ripristinare successivamente; oppure per evitare di dover intervenire sui dettagli interni del Memento con una GUI.

La soluzione

Nella soluzione si considera un Oggetto Caretaker, che vede una interfaccia ridotta del Memento e la cui responsabilità è di passare Memento ad altri oggetti che lo possano gestire. In altri termini Caretaker è il meccanismo che consentirà il recupero in assenza dell'originatore e , quindi, il ripristino di quest'ultimo. Nell'esempio state è un intero, però potrebbe essere una informazione strutturata o una classe o una interfaccia di costanti.

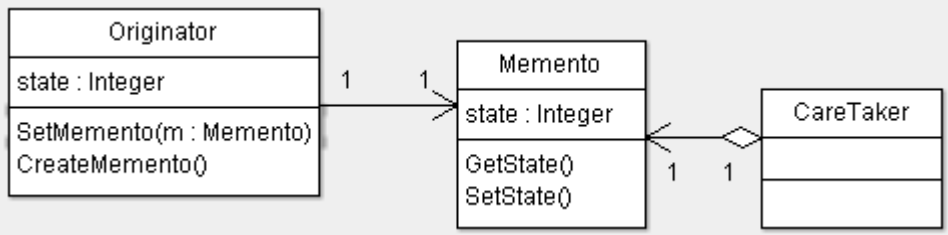


Fig. 54

State: Comportamentale

L'intento dello State

Permette ad un oggetto di cambiare il suo comportamento al cambiare del suo stato interno. L'oggetto si comporterà come se avesse cambiato la sua classe.

Esempio

Supponiamo di considerare una classe TCPConnection, che deve rappresentare la connessione alla rete, e quest'ultima si può trovare nello stato established, listening, closed. Quando un oggetto TCPConnection riceve una richiesta da un oggetto si comporterà in modo diverso a seconda dello stato della rete; infatti gli effetti del metodo Open() invocato su TCPState sarà diverso a secondo dello stato. TCPState è una interfaccia o una classe astratta superclasse delle sottoclassi che rappresentano stati diversi.

La soluzione

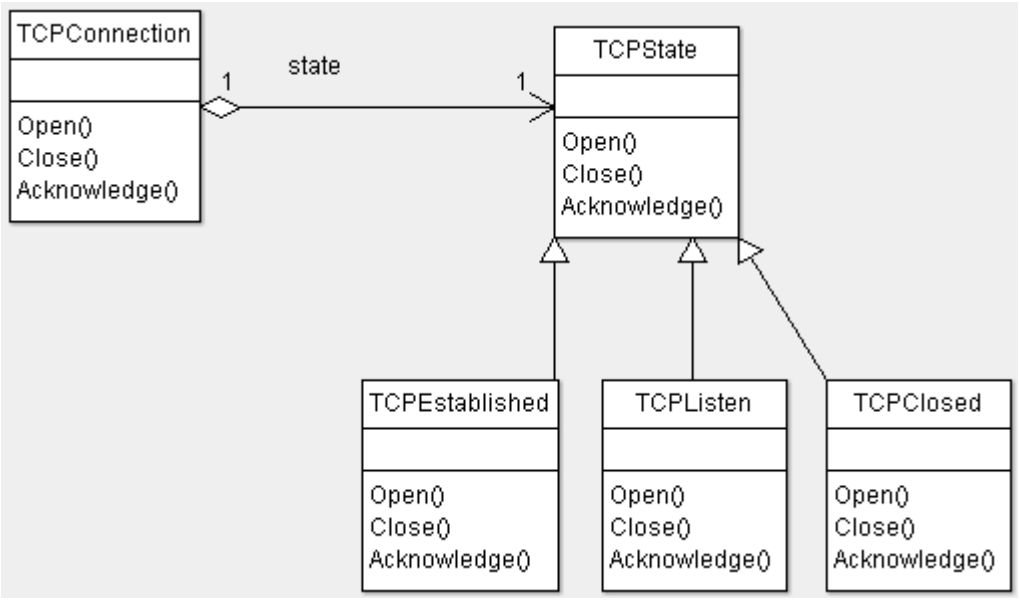


Fig. 55

Esiste una somiglianza col Command, ma mentre il Command permette di eseguire metodi a seconda dell'oggetto che richiama il Pattern, lo State esegue il metodo in base allo stato assunto da TCPConnection.

Interpreter: Comportamentale

L'intento dell'Interpreter

Dato un linguaggio definisce una rappresentazione della grammatica del linguaggio ed un interprete che usi la grammatica per interpretare le proposizioni del linguaggio.

Esempio

Questo pattern viene usato tipicamente nei compilatori e negli strumenti che trattano grammatiche. Supponiamo che la grammatica in gioco per le "espressioni regolari" sia del tipo:

expression ::= literal | alternation | sequence | repetition | '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | … {'a' | 'b'| …}*

La soluzione

La traduzione di quanto detto sopra è la Figura successiva.

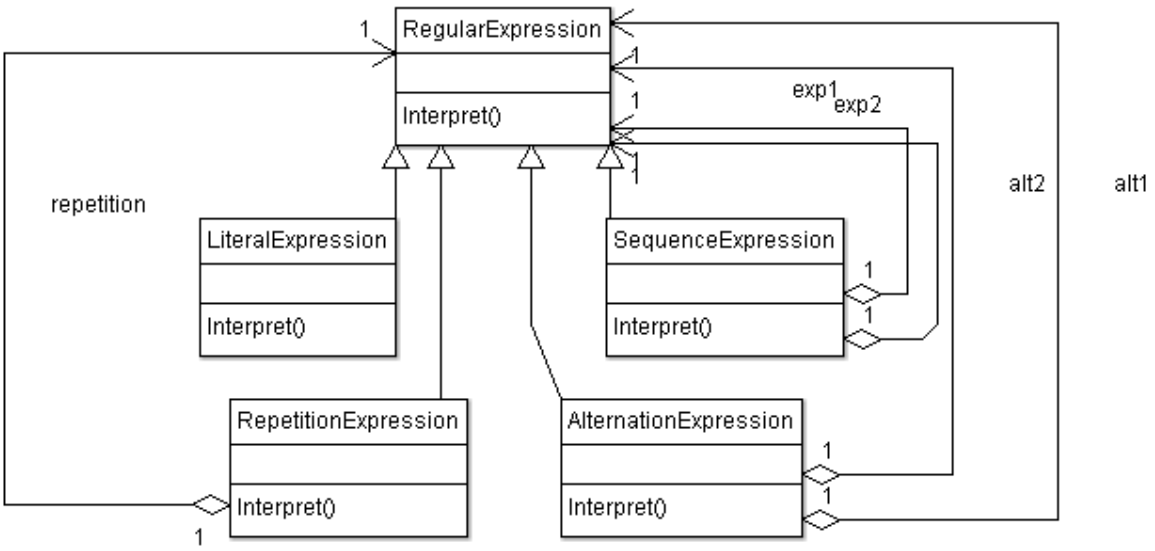


Fig. 56

Visitor: Comportamentale

L'intento del Visitor

E' applicabile ad oggetti compositi; il suo scopo ambizioso è di definire nuove operazioni senza modificare le classi degli elementi su cui opera.

Esempio

I compilatori sono capaci di esaminare i programmi scritti da noi secondo un *abstract syntax tree*, un albero sintattico; ma il loro scopo è molteplice:

- analisi sintattica del linguaggio
- analisi semantica (ad esempio che tutte le variabili siano state definite e inizializzate)
- generazione e ottimizzazione del codice
- indentazione

- stampa

Spesso si sfruttano gli alberi sintattici anche per ottenere:
- instrumentazione del codice e analisi statica e dinamica del software per il calcolo delle metriche
- debugging

Per ottenere una cosa del genere a partire da un albero sintattico, significa che, fissato un linguaggio, ogni nodo dell'albero - e le classi che lo rappresentano - va trattato diversamente a seconda se si tratta di assegnazione di variabili, di variabili o espressioni aritmetiche.

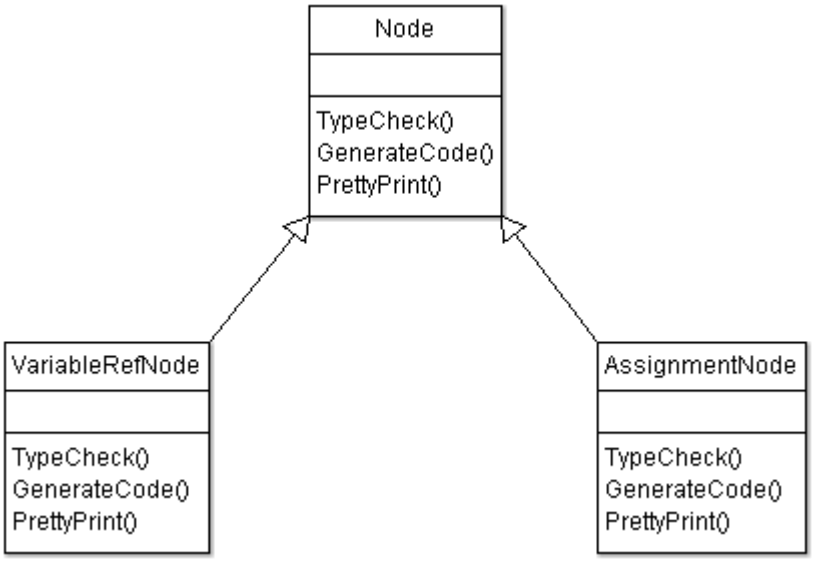


Fig. 57

La soluzione

La modellazione di Fig. 57 sebbene risolve la necessità ha però l'indiscutibile problema di mescolare metodi di fatti completamenti diversi: generazione del codice, la verifica del flusso e la stampa dell'albero ad esempio etc. rendendo complessa la comprensione del progettino. Mescola cioè sia gli aspetti funzionali che non funzionali. Ad esempio l'aspetto funzionale poteva essere per il compilatore fare il check e la generazione di codice; mentre la stampa dell'albero, il debugging etc potevano essere aspetti non funzionali.

Inoltre l'aggiunta di un solo metodo, ad esempio per l'instrumentazione, richiede una ricompilazione di tutte le classi della gerarchia.

Si può evitare? In realtà sì, col Design Pattern Visitor.

Basta raccogliere tutti i metodi correlati in una classe che chiameremo Visitor, che passeremo agli elementi dell'abstract syntax tree. Ogni elemento dell'albero deve accettare il Visitor e quando ne ha bisogno inoltra una richiesta al Visitor con l'elemento passato come parametro al Visitor. In questo caso il Visitor va ad eseguire l'operazione giusta per l'elemento, cosa che evita di mettere l'operazione nell'elemento stesso come era nella Fig. 57. In pratica rimane la gerarchia e viene introdotto il Visitor che viene passato agli elementi dell'albero per poter eseguire i metodi aggiuntivi. In questo caso è il Visitor che può cambiare con l'aggiunta di metodi ma non si ricompila tutta la gerarchia delle classi del progetto base. Nel Visitor si concentrano cioè i metodi che sono prettamente "non funzionali" cioè che non risolvono il problema funzionale.

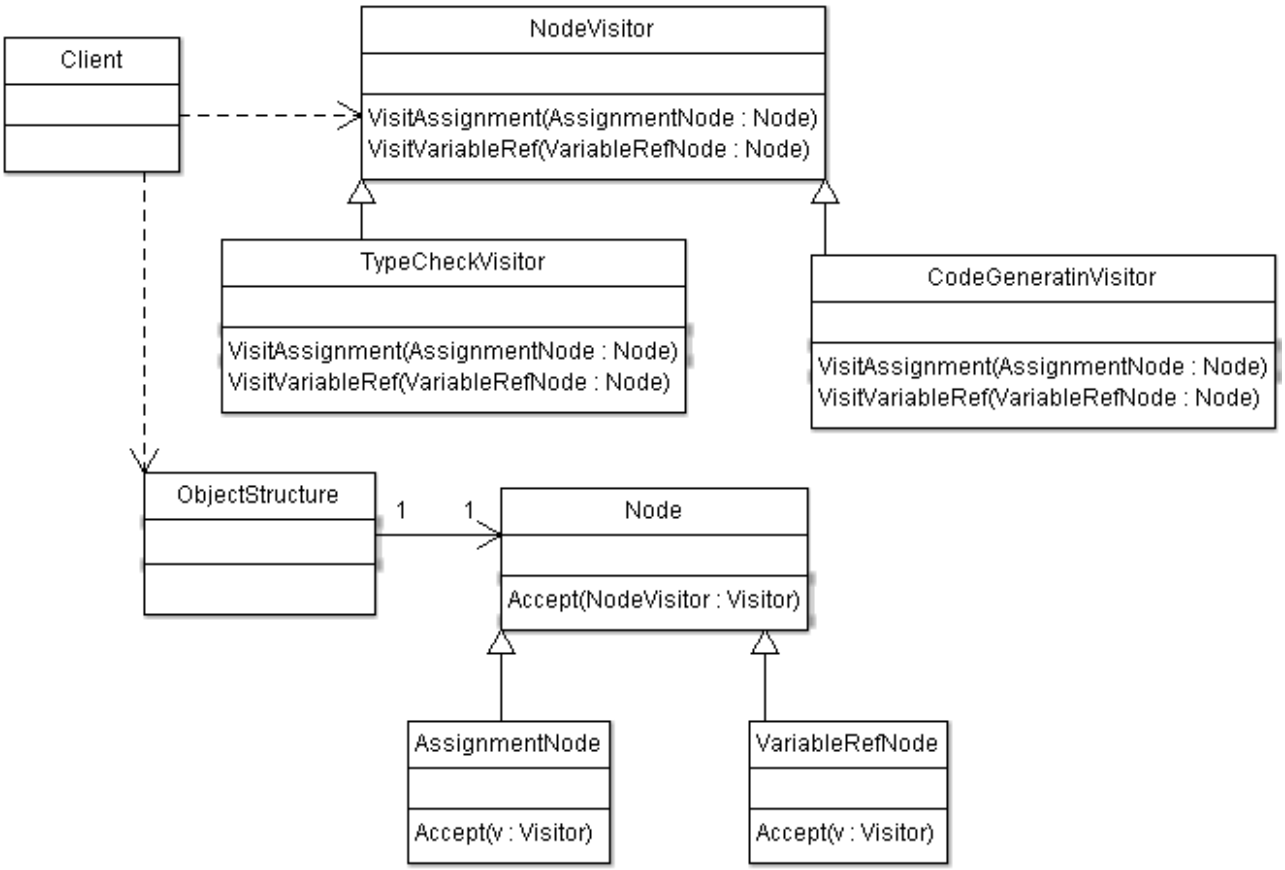


Fig. 58

Iterator: Comportamentale

L'intento dell'Iterator

L'iterator fornisce una modalità di accesso sequenziale agli elementi che formano un oggetto composito, senza esporre all'esterno la struttura dell'oggetto composito e tutelandone l'incapsulamento.

Esempio

Oggi in vari linguaggi come Java, JDBC, C#, C++ l'iterator è già presente. L'esigenza di un iterator, al di fuori di quello già offerto può nascere solo con la creazione di una struttura dati composita che risolve il dominio del problema: una Lista dinamica, una coda FIFO, uno stack, etc. Generalmente sono sufficienti le API del linguaggio in gioco.

In ogni caso supponiamo di dover trattare una Lista. In una Lista ci interessa contare gli elementi, aggiungere o rimuovere un elemento, attraversare la lista in modalità diverse: il primo elemento, l'ultimo item, l'item corrente etc.

Il Design Pattern che stiamo analizzando da la responsabilità di accesso e di attraversamento della Lista ad un Iterator ListIterator al di fuori di essa.

Poiché potremmo avere Liste diverse ci conviene lavorare con interfacce o classi astratte da cui derivare le nostre Liste e farle accedere da altrettante classe derivate di ListIterator. La lista però deve creare l'Iterator.

La soluzione

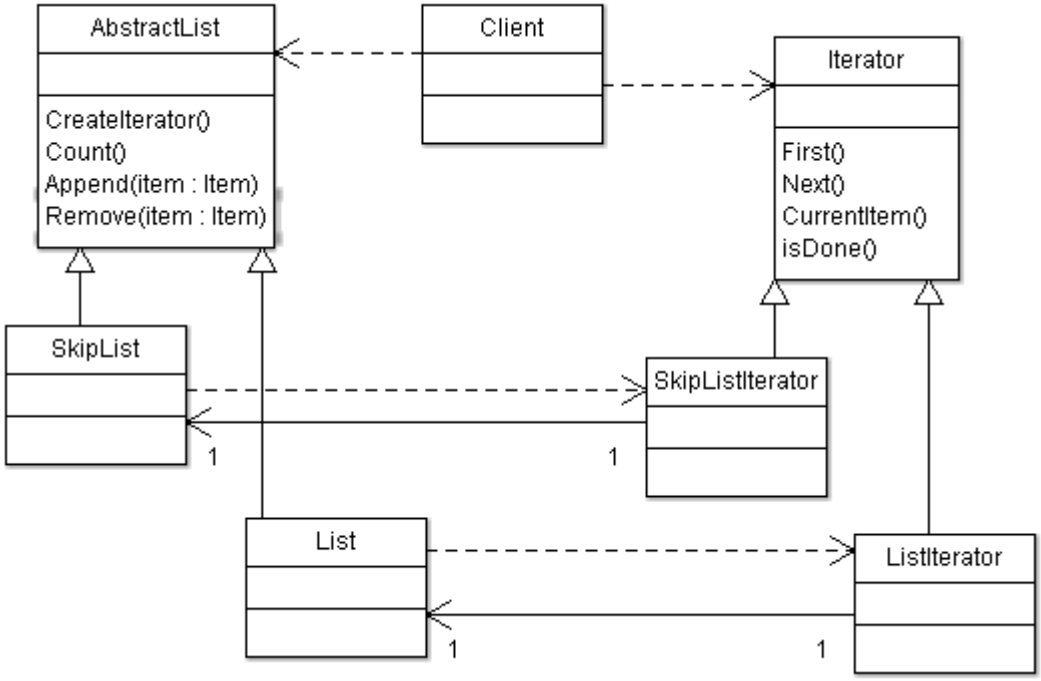


Fig. 59

Capitolo 6 – Ulteriori classificazioni di Pattern

Esistono molte altre classificazioni di Pattern, tali da individuare ed esaltare determinate caratteristiche progettuali; ma spesso sono la rivisitazione di Pattern già visti e derivabili da quelli di GoF.

Pattern Fondamentali

Sono definiti fondamentali poiché sono ricorrenti negli altri o sono di un livello di astrazione più alto. Alcuni di essi sono già stati esaminati.

Fra essi vi sono:

- Delegation [Grand98];
- Interface [Grand98];
- Immutable [Grand98];
- Marker Interface [Grand98];
- Proxy [GoF95] rivisto da [Grand98]

Pattern di Partizionamento

Questi pattern aiutano a "spezzettare" concetti complessi in più classi.

Fra essi vi sono:

- Layered Initialization [Grand98];
- Filter [BMRSS96];
- Composite [GoF95] rivisto da Grand;

Pattern di Concorrenza

Sono pattern che s'interessano di come gestire operazioni concorrenti, per gestire, ad esempio, risorse condivise.

Fra essi vi sono:

- Single Threaded Execution [Grand98];
- Guarded Suspension [Lea97];
- Balking [Lea97];
- Scheduler [Lea97];
- Read/Write Lock [Lea97];
- Producer/Consumer
- Two Phase Termination [Grand98]

Pattern J2EE

Anche in ambito J2EE, ci sono una rivisitazione dei Pattern GoF, secondo le architetture e le tecnologie J2EE: EJB session stateless, EJB session statefull, entity bean, Message driven bean (MDB), componenti JMX, etc, ma sempre come derivazione dei pattern già visti e spesso con nomi differenti o aggiuntivi.

Consigliamo, a chi dovesse lavorare in ambito J2EE, di fare approfondimenti anche con i Pattern J2EE.

Glossario

Come in ogni progetto Object Oriented è previsto anche in questo lavoro un glossario dei termini.

Termine	**Descrizione**
Analisi	L'analisi (OOA) definisce il problema in gioco e risponde alla domanda "Che cosa realizzare".
Audit	Suggerimenti prodotti a seguito di una "code ispection" umana o forniti da uno strumento (es: Together, Fortify, etc.) e finalizzati a migliorare la chiarezza, la manutenibilità del software e i requisiti non funzionali.
Forward Engeenering	E' l'attività che dall'analisi porta all'implementazione del software.
GoF	Gang of four: Gamma, Vlissides, Helm, Johnson
GRASP	General Responsabilities Assignement Software Pattern
Incrementale	In un Unified Process (UP), il processo incrementale prevede di affrontare più volte, con vari raffinamenti implementativi (refactoring), lo stesso gruppo di requisiti (storie) previsti in una iterazione. Questo per migliorare la qualità del software.
Iterativo	In un Unified Process (UP), il processo iterativo prevede di affrontare solo gruppi di requisiti (storie) per volta, in modo che il progetto evolva ad incremento e consolidamento, in termini di requisiti analizzati, progettati e implementati.
Metriche	Parametri oggettivi rispetto ai quali confrontare il software implementato (sw o modellazione del progetto è la stessa cosa in OO) e individuarne il grado di qualità. Tali metriche sono ottenibili con strumenti (es: Together) che esaminano il software.
OO	Object Oriented
OOA	Object Oriented Analysis
OOD	Object Oriented Design
Progettazione	La progettazione (OOD) porta dal dominio del problema al dominio della soluzione e soddisfa la domanda "Come realizzare, ciò che è stato definito nell'analisi". Nella modellazione di questo tipo sono importanti i Design Pattern.
Refactoring	Un rifacimento implementativi del software teso a migliorarne la qualità, fermo restando il requisito e il comportamento finale del software.
Reverse Engeenering	E' l'attività che a partire dal software risale al modello OO di progetto.
Sviluppo o Development	L'implementazione dovrebbe essere la parte di minor peso del ciclo, mentre Modellazione di analisi e di progettazione hanno definito il 99% delle cose. L'implementazione si deve occupare solo dello sviluppo dei metodi e degli algoritmi migliori da utilizzare in termini di tempo e efficienza (spazio, memoria, etc).
UML	Unified Modeling Language – Un linguaggio unificato e standard per la rappresentazione e lettura del modello OO.

Conclusioni

Terminiamo l'e-book con pillole di saggezza filosofica cinese, insita nell'Extreme Programming o nella Metodologia Agile.

Un prodotto senza una adeguata progettazione è una nave in balia delle onde dell'oceano, senza rotta e senza mezzi per prevenire e ridurre al massimo i danni.

L'Object Oriented è una metodologia ed una disciplina che esalta ed esige un'organizzazione sia umana che progettuale, in grado di fondere strettamente i team di analisi/progettazione/sviluppo/test; capace di realizzare sinergie e gruppi amalgamati in breve, con un apparente costo iniziale. Consente di lavorare in cicli ristretti e rapidi, migliorando ad ogni riciclo sia uomini che prodotti.

Se si entra in tale ottica il lavoro diventa una professionalità divertente e produttiva; ma molto dipende dall'uomo: le metodologie da sole non pensano, né camminano né producono prodotti di qualità.

I prodotti da soli non fanno un'azienda. Il know-how tecnologico, quello dei dati e dei processi di business aziendali sono un tesoro interno all'azienda a cui non si deve rinunciare.

Riferimenti

[DR1] Martin Fowler – UML Distilled – Prima edizione italiana
[DR2] Rosario Turco – Concetti di base Object Oriented
[DR3] Rosario Turco – Principi di Disegno
[DR4] Rosario Turco – Usabilità e ripetibilità dei processi produttivi software
[DR5] Rosario Turco – Modellare con l'UML ed i colori
[DR6] Rosario Turco – Risoluzione di problemi con UML
[DR7] Rosario Turco – Tradurre le relazioni UML in C++
[DR8] Rosario Turco - Refactoring: la teoria in pratica
[DR9] Rosario Turco – Disegno per il riuso e con il riuso
[DR10] Robert C. Martin – Design Principles and Design Patterns
[DR11] Gamma, Helm, Johnson,Vlissides – Design Patterns – Elementi per il riuso di software a oggetti – Prima edizione italiana
[DR12] Rosario Turco – Framework e UML
[DR13] Rosario Turco – Il Business AS IS Modeling con UML
[DR14] Kent Beck – Programmazione estrema – Introduzione
[DR15] Rosario Turco – Extreme Programming
[DR16] Rosario Turco – Rational Unified Process
[DR17] Rosario Turco – BPM, SOA e Business Rules Engine, l'ultima frontiera
[DR18] John Cheesman, John Daniels – UML Components
[DR19] Rosario Turco – Progettazione a componenti
[DR20] Catalysis http://www.catalysis.org/
[DR21] Rosario Turco - Metodologia Agile