APS - Analisi e Progettazione del Software

Elia Ronchetti
@ulerich

2022/2023

Contents

Chapter 1

Introduzione

Che cosa sono l'analisi e la Progettazione

- Analisi - Enfatizza l'investigazione di un problema e dei suoi requisiti, anzichè di una soluzione
- La progettazione enfatizza una soluzione concettuale che soddisfa i requisiti del problema

Fare la cosa giusta (analisi) e fare la cosa bene (progettazione)

1.1 Analisi e Progettazione orientata agli oggetti

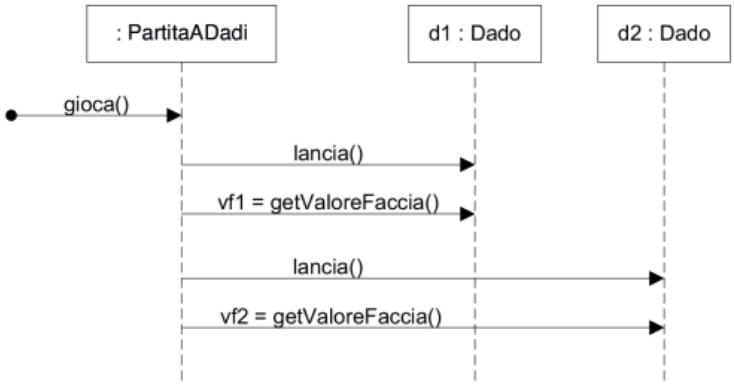
L'analisi orientata agli oggetti enfatizza sull'identificazione dei concetti o degli oggetti, nel dominio del problema.
La progettazione orientata agli oggetti enfatizza sulla definizione di oggetti software che collaborano per soddisfare i requisiti.

Esempio Oggetti - Aereo, Volo, Pilota ognuno con i propri attributi (tipo basi di dati). Analisi e progettazione hanno obiettivi diversi che venogno perseguiti in modi diversi, sono comunque attività sinergiche.
L'OO (Object Oriented) enfatizza la rappresentazione di oggetti

Definizione dei casi d'uso I casi d'uso sono delle storie scritte relative al mondo in cui il sistema viene utilizzato.

Definizione di un modello di dominio Un modello di dominio mostra i concetti o gli oggetti significativi del dominio, con i relativi attributi e associazioni.

Definizione dei diagrammi di interazione Un diagramma di interazione mostra le collaborazioni tra oggetti software.



Definizioni dei diagramma delle classi di progetto Un diagramma delle classi di progetto mostra una vista statica delle definizioni delle classi software, con i loro attributi e metodi.

Lo scopo di tutti questi diagrammi è facilitare la progettazione e il passaggio da idea a codice.

1.2 Che cosa è UML

Unified Modelling Language (UML) è un linguaggio visuale di modellazione dei sistemi e non solo. Rappresenta una collezione di best practice di ingegneria, dimostratesi vincenti nella modellazione di sistemi vasti e complessi. Dato che è lo standard de facto favorisce la divulgazione di informazioni nella comunità di ingegneri del software.
UML NON è una metodologia

- UML è un linguaggio visuale
- UP (Unified Process) è una metodologia

UML modella i sistemi come insiemi di oggetti che collaborano tra loro

- Struttura Statica
  - Quali tipo di oggetti sono necessari
  - Come sono tra loro correlati
- Struttura Dinamica
  - Ciclo di vita di questi oggetti
  - Come collaborano per fornire la funzionalità richieste

1.2.1 Tre modi per applicare UML

- UML come abbozzo
- UML come progetto
- UML come linguaggio di programmazione

UML come Abbozzo Diagrammi informali e incompleti (spesso abbozzati a mano) che vengono creati per esplorare parti difficili dello spazio del problema o della soluzione, sfruttando l'espressività dei linguaggi visuali.

UML come Progetto Diagrammi di progetto relativamente dettagliati, utilizzati per il reverse engineering, per la documentazione e per la comunicazione, utilizzati quindi per visualizzare e comprendere meglio il codice esistente mediante diagrammi UML.

UML come linguaggio di programmazione In questo caso il codice viene generato direttamente e automaticamente da UML (approccio ancora in fase di sviluppo). **La modellazione agile enfatizza l'uso di UML come abbozzo**

Due punti di vista per applicare UML

- Punto di vista concettuale - I diagrammi descrivono oggetti del mondo reale o in un dominio di interesse
- Punto di vista software - I diagrammi descrivono atrsazioni o componenti software

Entrambi usano la stessa notazione UML.

Il significato di classe nei diversi punti di vista

Nell'UML grezzo, i rettangoli illustrati sono chiamati classi, questo termine racchiude una varietà di casi: oggetti fisici, concetti astratti, elementi software, ecc.
Il significato varia in base al diagramma di utilizzo, per fare chiarezza:

- Classe concettuale - Oggetto o concetto del mondo reale - Significato attribuito nel **Modello di Dominio di UP**
- Classe software - Classe intesa come componente software (es. classe Java) - Significato attribuito nel **Modello di Progetto di UP**

1.2.2 Vantaggi della modellazione visuale

Disegnare o leggere UML implica che si sta lavorando in modo visuale e il nostro cervello è più rapido a comprendere simboli, unità e relazioni rappresentati con una notazione grafica.

Chapter 2

Processi per lo sviluppo del software

Un processo per lo sviluppo del software (o processo software) definisce un approccio disciplinato per la costruzione, il rilascio e la manutenzione del software.
Definisce chi fa che cosa, quando e come per raggiungere un certo obiettivo.

- Cosa - Sono le attività
- Chi - Sono i ruoli
- Come - Sono le metodologie
- Quando - Riguarda l'organizzazione temporale delle attività

2.1 Che cos'è UP

Un processo per lo sviluppo software descrive un approccio alla costruzione, al rilascio ed eventualmente alla manutenzione del software. **Unified Process (UP)** è un processo iterativo diffuso per lo sviluppo del software per la costruzione di sistemi orientati ad oggetti.
UP è molto flessibile e aperto e incoraggia l'utilizzo di pratiche tratte da altri metodi iterativi come Extreme Programming (XP), Scrum, ecc.

Riassumendo

- UP è un processo iterativo
- Le pratiche UP forniscono una struttura di esempio

- UP è flessibile cioè iterativo, incrementale ed evolutivo
- Pilotato dai casi d'uso (requisiti) e dai fattori di rischio
- Incentrato sull'architettura

2.2 Il processo a cascata

Il processo a cascata è il più vecchio tra quelli utilizzati oggi (definito tra gli anni 60-70). Definito così per il suo ciclo di vita a cascata (o sequenziale), basato sullo svolgimento sequenziale delle diverse attività dello sviluppo del software.
Richiede che i requisiti siano chiari sin dall'inizio e spesso non è così dato che durante la progettazione emergono altri requisiti da parte del cliente.

Perchè il processo a cascata è soggeto a frequenti fallimenti É stata rilevata un'alta percentuale di fallimenti di progetti che utilizzano questo processo e ci sono diversi motivi legati a questo alto tasso di fallimenti.
Il motivo principale è che il processo a cascata presuppone che i requisiti siano prevedibili e stabili e che possano essere definiti all'inizio, ma ciò è falso. É stato dimostrato che ogni progetto software subisce mediamente il 25% di cambiamenti nei requisiti. Addirittura altri studi hanno evidenziato tassi dal 35 fino al 50%, per quanto riguarda i progetti più grandi. Nello sviluppo software il presupposto che i requisiti siano stabili e prevedibili nel tempo è fondamentalmente sbagliato. Nei progetti software il cambiamento è piuttosto una costante.

2.3 Sviluppo iterativo ed evolutivo

Lo sviluppo iterativo ed evolutivo è una pratica fondamentale in molti processi software moderni come UP e Scrum. In questo approccio al ciclo di vita, lo sviluppo è organizzato in una serie di mini progetti brevi di lunghezza fissa chiamati **iterazioni**. Ad ogni fase c'è un raffinamento del progetto a seguito di feedback, per questo viene chiamato **sviluppo iterativo e incrementale** o anche **sviluppo iterativo ed evolutivo**.
L'instabilità dei requisiti e del progetto tende a diminuire nel tempo, nelle iterazioni finali è difficile (ma non impossibile), che si verifichi un cambiamento significativo dei requisiti.

2.3.1 Vantaggi dello sviluppo iterativo

- Minor probablità di fallimento del progetto
- Miglior produttività
- Percentuali più basse di difetti
- Riduzione precoce anzichè tardiva dei rischi maggiori
- Progesso visibile fin dall'inizio
- Feedback preoce e conseguente coinvolgimento dell'utente e adattamento
- Gestione della complessità

2.3.2 Feedback e adattamento

Attività fondamentale per il successo, come si struttura? E da dove proviene?

- Feedback proveniente dalle attività di sviluppo
- Feedback proveniente dai test e dagli sviluppatori che raffinano il progetto e i modelli
- Feedback circa l'avanzamento del tema nell'affrontare i requisiti, per raffinare le stime dei tempi e dei costi
- Feedback proveniente dal cliente e dal mercato

2.3.3 Durata delle iterazioni e timeboxing

Una buona pratica dello sviluppo iterativo è il timeboxing: le iterazioni hanno una lunghezza fissata (Molti processi iterativi raccomandano una lunghezza da 2 a 6 settimane).
Passi piccoli, feedback rapido e adattamento sono le idee centrali dello sviluppo iterativo. La durata di un'iterazione, una volta fissata non può cambiare, quando viene fissata viene detta **timeboxed**.

Non permettere al pensiero a cascata di invadere un progetto iterativo

2.3.4 Sviluppo iterativo e flessibilità del codice e del progetto

L'adozione dello sviluppo iterativo richiede che il softare venga realizzato in modo flessibile affinchè l'impatto dei cambiamenti sia basso. Il codice sorgente deve quindi essere facilmente modificabile e comprensibile (per facilitarne la modifica). **flessibile**

2.4 Come eseguire l'analisi e progettazione in modo iterativo ed evoluto

Un'attività critica nello sviluppo iterativo è la pianificazione delle iterazioni, non bisogna tentare di pianificare tutto il progetto in modo dettagliato fin dall'inizio. I processi iterativi promuovono la pianificazione guidata dal rischio e guidata dal cliente.

2.4.1 Non cambiare gli obiettivi dell'iterazione

Durante un'iterazione non è possibile cambiare i requisiti, dato che sono stati precedentemente fissati durante la pianificazione iterativa e poi bloccati. Questo perchè così facendo il Team di sviluppo può lavorare al suo meglio.

2.4.2 Iterazioni - Perchè sono la chiave per UP

Le iterazioni sono la chaive per UP, dato che ogni iterazione è come un mini-progetto che include:

- Pianificazione
- Analisi e progettazione
- Costruzione
- Integrazione e test
- Un rilascio

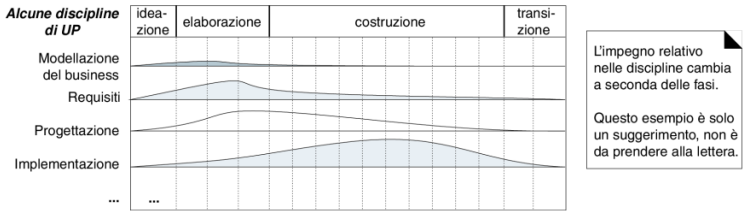
Si arriva al rilascio finale attraverso una sequenza di iterazioni. Le iterazioni possono sovrapporsi e questo è importante, dato che consente lo sviluppo parallelo e il lavoro flessibile in grandi squadre. Ad ogni iterazione si svolge una parte del lavoro totale di ogni disciplina.

Esempio Si svolgono in parallelo sia modellazione, Progettazione e implementazione, chiaramente a intensità diverse in base all'iterazione (e quindi alla fase del progetto), per esempio il testing sarà maggiore dopo qualche iterazione, dato che ci saranno più cose da testare.
Ogni iterazione genera una **release** che costituisce un insieme di manufatti previsti e approvati. Un incremento è la differenza tra una release e quella successiva. Costituisce un passo in avanti verso il rilascio finale del sistema, per questo UP si chiama iterativo e incrementale.

2.5 Le fasi di UP

- Ideazione - Avvio del progetto
- Elaborazione - Realizzazione del nucleo dell'architettura
- Costruzione - Realizzazione delle capacità operative iniziali
- Transizione - Completamento del prodotto

2.5.1 Fasi e dicipline (o flussi di lavoro)



Per ogni fase si considera:

- L'attenzione in termini di flussi di lavoro
- L'obiettivo per la fase
- Il milestone alla fine della fase

Il processo con lo scenario di sviluppo di UP è nato per essere personalizzato, per questo tra gli elaborati e le pratiche di UP, quasi tutto è opzionale.
Alcune pratiche e principi sono fissi, come lo sviluppo iterativo e guidato dal rischio e la verifica continua della qualità.
Tutti gli elaborati (modelli, diagrammi, documenti) sono opzionali, con l'ovvia esclusione del codice.

La scelta delle pratiche degli elaborati UP per un progetto può essere scritta in un breve documento chiamato scenario di sviluppo.

Scenario di sviluppo

Disciplina	Pratica	Elaborato	Ideazione	Elaboraz	Costr	Transiz
		Iterazione	I1	E1..En	C1..Cn	T1..T2
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio		i		
Requisiti	workshop requisiti	Modello dei Casi d'Uso	i	r		
	esercizio sulla visione votazione a punti	Visione	i	r		
		Specifica Supplementare	i	r		
		Glossario	i	r		
Progettazione	modellazione agile	Modello di Progetto		i	r	
	sviluppo guidato dai test	Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	

i=Inizio, r=raffinamento

Non si è capito lo sviluppo iterativo o UP se

- Si cerca di definire la maggior parte dei requisiti prima di iniziare la progettazione o l'implementazione
- Si pensa che i diagrammi UML e le attività di progettazione devono definire il progetto in dettaglio e che la programmazione è una mera traduzione di questi diagrammi
- Si ritiene che una durata adeguata per un'interazione sia tre mesi anzichè tre settimane
- Si prova a pianificare il progetto in dettaglio dall'inizio alla fine, in modo speculativo

Risulta quindi fondamentale ricordarsi che è un metodo per visualizzare il progetto, per poter poi flessibilmente apportare modifiche, deve rimanere quindi agile. Per dettagli vedere il Manifesto Agile.

2.6 SCRUM

Scrum è un metodo agile che consente di sviluppare e rilasciare prodotti software con il più alto valore per i clienti ma nel più breve tempo possibile. Scrum si occupa principalmente dell'organizzazione del lavoro e della gestione di progetti.

Terminologia

- Daily Scrum - Incontro giornaliero del Team SCRUM che esamina i progressi e definisce le priorità del lavoro da svolgere in quel giorno. Incontro breve faccia a faccia con tutto il team.
- ScrumMaster - Responsabile di assicurare che il processo di Scrum sia seguito e guida il team nell'uso efficace di Scrum. Responsabile dell'interfacciamento con il resto dell'azienda e di assicurare che il team non venga deviato da interferenze esterne
- Sprint - Un'iterazione di sviluppo. Le sprint sono solitamente di 2-4 settimane
- Velocity - Una stima di quanto lavoro rimanente un team può fare in un singolo sprint. Fondamentale per capire le prestazioni di una squadra e così migliorarle.
- Team di sviluppo - Un gruppo auto-organizzatore di sviluppatori di software, che non dovrebbe superare le 7 persone. Sono responsabili dello sviluppo del software e di altri documenti essenziali del progetto.
- Incremento potenzialmente rilasciabile - L'incremento software fornito da uno sprint
- Product Backlog - Questo è un elenco di elementi da fare (to do) che il team Scrum deve affrontare
- Product Owner - Può essere un cliente, ma potrebbe essere anceh un product manager in una società di software o un rappresentante di altri stakeholder. Identifica le caratteristiche o i requisiti del prodotto.

Chapter 3

Analisi dei requisiti

3.1 Ideazione

La maggior parte dei progetto richiede un breve passo iniziale in cui si esaminano i seguenti tipi di domande:

- Il progetto è fattibile?
- Comprare e/o costruire?
- Stima approssimativa e non affidabile dei costi
- Dovremmo procedere o fermarci?

L'ideazione non è la fase dei requisiti.
Il problema principale che risolve l'ideazione è il seguente:

Le parti interessate hanno un accordo di base, sulla visione del progetto, e vale la pena di investire un'indagine seria?

Lo scopo è quindi quello di stabilire una visione comune per gli obiettivi del progetto e capire se questo è fattibile.
In questa fase non si utilizza molto UML (verrà utilizzato soprattutto durante l'elaborazione).

Non hai capito l'ideazione se

- Dura più di qualche settimana
- Provi a definire molti requisiti
- Ci si aspetta che i piani e le stime siano affidabili

- I nomi di molti attori e casi d'uso non sono stati identificati
- Troppi casi d'uso sono stati scritti nel dettaglio
- Nessun caso d'uso è stato scritto in dettaglio

3.2 Che sono sono i requisiti

Un requisito è una capacità o una condizione a cui il sistema e più in generale il progetto, deve essere conforme.
I requisiti sono un aspetto veramente molto importante, si evidenzia addirittura che 34 % delle cause dei fallimenti dei progetti software riguardano l'attività dei requisiti.

Ci sono due tipi principali di requisiti

- Requisiti funzionali (comportamentali) - Descrivono il comportamento del sistema, in termini di funzionalità fornite ai suoi utenti e informazioni che il sistema deve gestire
- Requisiti non funzionali (tutti gli altri requisiti) - Sono relativi a proprietà del sistema nel suo complesso come per esempio sicurezza, presetazioni, scalabilità, usabilità...

3.2.1 Requisiti funzionali

Sono funzionalità o servizi che il sistema deve fornire, risposte che l'utente aspetta dal software in determinate condizioni, risultati che il software deve produrre in risposta a specifici input.

Il problema dell'imprecisione nella specifica dei requisiti Requisiti ambigui possono portare a diverse interpretazioni da sviluppatori e utenti. In linea di principio i requisiti dovrebbero essere completi e coerenti, quindi includere la definizione di tutti i servizi richiesti e non essere ambigui.

3.2.2 Requisiti non funzionali

Questi definiscono le proprietà e i vincoli del sistema, ad esempio affidabilità, tempi di risposta, oppure vincoli come la capacità dei dispositivi I/O, le rappresentazioni dei dati nelle interfacce di sistema, ecc.
I requisiti non funzionali possono essere più critici dei requisiti funzionali,

dato che in caso se non definiti correttamente il sistema potrebbe risultare inutilizzabile.

Obiettivi e requisiti I requisiti non funzionali possono essere molto difficili da stabilire con precisione e requisiti imprecisi possono essere difficili da verificare.
Un obiettivo può essere un'intenzione generale dell'utente come la facilità d'uso, mentre un requisito non funzionale verificabile è una dichiarazione che utilizza alcune misure oggettivamente verificabili. Gli obiettivi sono utili agli sviluppatori in quanto trasmettono le intenzioni degli utenti del sistema.

Metriche per specificare i requisiti non funzionali

Proprietà	Misura
Velocità	Transazioni elaborate al secondo Tempi di risposta a utenti/eventi Tempo di refresh dello schermo
Dimensione	Mbytes Numero di chip ROM
Facilità d'uso	Tempo di addestramento Numero di maschere di aiuto
Affidabilità	Tempo medio di malfunzionamento Probabilità di indisponibilità Tasso di malfunzionamento Disponibilità
Robustezza	Tempo per il riavvio dopo malfunzionamento Percentuali di eventi causanti malfunzionamento Probabilità di corruzione dei dati dopo malfunzionamento
Portabilità	Percentuali di dichiarazioni dipendenti dall'architettura di destinazione Numero di architetture di destinazione

3.2.3 Quali sono i modi validi per troavare requisiti

- Interviste con i clienti
- Scrivere casi d'uso con i clienti
- Workshop dei requisiti a cui partecipano sia sviluppatori che clienti
- Gruppi di lavoro con rappresentanti dei clienti
- Sollecitare feedback clienti alla fine di ogni iterazione

3.3 Requisiti e principali elaborati di UP

- Modello dei casi d'uso
- Specifiche supplementari

- Glossario
- Visione
- Regole di Business

Linee guida per la scrittura dei requisiti

- Ideare un formato standard e utilizzare per tutti i requisiti
- Utilizzo del linguaggio in modo consistente. Utilizzare DEVE per requisiti obbligatori, DOVREBBE per requisiti desiderabili
- Evidenziare porzioni di testo per identificare le parti più importanti dei requisiti
- Evitare gergo informatico

Chapter 4

Casi d'uso

I casi d'uso sono storie scritte, ampiamente utilizzati per scoprire e registrare i requisiti. Un caso d'uso è un dialogo tra un attore e un sistema che svolge un compito.
I casi d'uso non sono elaborati orientati agli oggetti, influenzano però molti aspetti di un progetto, compresa l'analisi e la progettazione orientata agli oggetti. **I casi d'uso sono testo**.

4.1 Attori, scenari e casi d'uso

- Un attore è qualcosa o qualcuno dotato di comportamento - Es. cassiere o sistema di pagamento
- Uno scenario (o istanza del caso d'uso) è una sequenza specifica di azioni e iterazioni tra il sistema e alcuni attori - Descrive una particolare storia nell'uso del sistema, si possono dividere in scenari di successo e di fallimento.
- Un caso d'uso è una collezione di scenari correlati, sia di successo che di fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo specifico

Il **Modello dei Casi d'Uso** è l'insieme di tutti i casi d'uso scritti.

4.1.1 Perchè i casi d'uso

Si tratta di un metodo semplice per descrivere i requisiti funzionali ed è direttamente comprensibile dai clienti. Inoltre mettono in risalto obiettivi degli utenti e il loro punto di vista. Molto utile per produrre la guida utente e per i test di sistema.

I casi d'uso sono requisiti funzionali Dato che essi indicano cosa deve fare il sistema, un caso definisce un contratto relativo al comportamento di un sistema.

4.2 Tipi di Attore

SuD = Sistema in discussione

- Attore primario: raggiunge obiettivi usando il SuD
- Attore finale: vuole che il SuD sia utilizzato affinchè vengano raggiunti i suoi obiettivi (es cliente)
- Attore di supporto: es. servizio pagamento
- Attore fuori scena: ha un interessa nel comportamento del caso d'uso SuD, ma non è attore primario, finale o di supporto (es Governo interessanto al pagamento delle imposte)

4.3 Tre formati comuni per i casi d'uso

- Formato breve - Riepilogo conciso di un solo paragrafo, normalmente relativo al solo scenario principale di successo
- Formato informale - Più paragrafi relativi anche agli altri scenari
- Formato dettagliato - Tutti i passi e le variazioni sono scritti nel dettaglio

Formato Dettagliato

Formato dettagliato

SELEZIONE DEL CASO D'USO	DESCRIZIONE
Nome del Caso d'Uso	Inizia con un verbo
Portata	Il sistema che si sta progettando
Livello	"Obiettivo utente" o "sottofunzione"
Attore Primario	Nome dell'attore primario
Parti Interessate e Interessi	A chi interessa questo caso d'uso e che cosa desidera
Pre-condizioni	Che cosa deve essere vero all'inizio del caso d'uso (e vale la pena di dire al lettore)
Garanzia di successo	Che cosa deve essere vero se il caso d'uso viene completato con successo (e vale la pena di dire al lettore)
Scenario Principale di Successo	Uno scenario comune di attraversamento del caso d'uso, di successo e incondizionato
Estensioni	Scenari alternativi, di successo e di fallimento
Requisiti speciali	Requisiti non funzionali correlati
Elenco delle variabili tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati
Frequenza di ripetizione	Frequenza prevista di esecuzione del caso d'uso
Varie	Altri aspetti, ad esempio i problemi aperti

Scrivere in uno stile essenziale, come per esempio

- L'amministratore si identifica
- Il sistema autentica l'identità

Scrivere casi d'uso in modo conciso e completo.
Scrivere casi d'uso a scatola nera, ovvero speciricare che cosa deve fare il sistema, senza decidere come lo farà. Concentrarsi sulla comprensione di ciò che l'attore considera un risultato di valore.

4.4 Come trovare i casi d'uso

- Scegliere il confine di sistema (Es identificando gli attori esterni)
- Identificare gli attori primari
- Identificare gli obiettivi per ogni attore primario
- Definire i casi d'uso che soddisfano questi obiettivi

Per dettagli leggere Capitolo 7 pag 88 del Larman Link Bookshelf.

4.4.1 Verificare l'utilità dei casi d'uso

Ci sono diversi metodi

- Test del capo
- Test EBP (Elementary Business Process) - Capire se si tratta di un processo elementare e non troppo complesso
- Test della Dimensione - Un buon caso d'uso non dovrebbe essere troppo breve

4.4.2 Livello dei casi d'uso

I casi d'uso possono essere scritti a livelli diversi

- Livello di obiettivo utente
- Livello di sotto-funzione
- Livello di sommario

4.5 Diagramma dei Casi d'uso

usare notazioni diverse per gli attori umani e per quelli che sono sistemi informatici. Unire i vari attori e casi d'uso con associazioni rappresentate da una linea continua. La direzione viene utilizzata nel verso di chi dà inizio all'interazione.
Non viene associata una direzione se entrambe le parti possono dare inizio all'interazione.

Relazioni tra Casi d'Uso

- Include - Relazione tra un caso d'uso base ed un caso d'uso incluso nel caso base
- Extend - Connette un caso d'uso esteso ad un caso d'uso base, aggiunge varianti ad un caso d'uso base e viene inserito solo se la condizione d'estensione è vera.
- Generalization - Un caso d'uso genitore è una generalizzazione di un caso d'uso figlio. Viene eseguito se la condizione di generalizzazione è vera. Possono ereditare, aggiungere sovrascrivere le funzioni del loro genitore. Si può avere anche tra attori questa relazione.

Chapter 5

Modellazione di dominio

Un caso d'uso o una caratteristica sono spesso troppo complessi per poter essere completati in una sola breve iterazione pertante le varie parti o scenari possono essere distribuiti su diverse iterazioni.

5.1 L'elaborazione

L'elaborazione è la serie iniziale di iterazioni durante le quali, in un progetto normale:

- Viene programmato e verificato il nucleo, rischioso, dell'architettura software
- Viene scoperta e stabilizzata la maggior parte dei requisiti
- I rischi maggiori sono attenuati o rientrano

Alcune idee e best practice

- Eseguire iterazioni guidate dal rischio, brevi e timeboxed
- Iniziare presto la programmazione
- Progettare, implementare e testare in modo attivo le parti principali
- Effettuare test presto, spesso e in modo realistico
- Adattare in base al feedback proveniente da test, utenti e sviluppatori
- Scrivere la maggior parte dei casi d'uso e degli altri requisiti nel dettaglio

É importante definire le priorità in base al rischio, copertura e criticità.

- Rischio - Comprende tanto la complessità tecnica quanto altri fattori, come l'incertezza dello sforzo o l'usabilità
- Copertura - Indica che le iterazioni iniziali prendono in considerazione tutte le parti principali del sistema
- Criticità (Valore) - Le funzioni che il cliente considera di elevato valore di business

Prima di ogni iterazione viene aggiornata la classifica.

5.2 L'analisi a oggetti - Modellazione di Dominio

Nell'analisi orientata agli oggetti vengono modellati i seguenti aspetti

- Il Dominio informativo, ovvero le tipologie di informazioni che il sistema deve rappresentare e gestire
- Le interazioni fra attori e sistema, ovvero le funzioni
- Il comportamento del sistema, ovvero i cambiamenti nelle informazioni associati a ciascuna funzione

Vengono modellati nella come segue:

- Il dominio informativo è rappresentato mediante un modello oggetti (modello di dominio)
- Le funzioni del sistema sono rappresentate in termini delle operazioni che il Sistema è chiamato a svolgere (**operazioni di sistema**), insieme a una descrizione dell'ordine relativo in cui si possono richiedere queste operazioni **diagrammi di sequenza di sistema**
- Il comportamento è descritto come l'effetto prodotto dall'esecuzione di ciascuna operazione di sistema (**contratti** delle operazioni di sistema)

Modellazione di Dominio

Un modello di dominio è il modello più importante dell'OOA dato che:

- Descrive le classi concettuali e le relazioni tra esse
- Fonte di ispirazione per classi di progetto e di implementazione
- Sviluppato in modo iterativo ed incrementale

- Limitato dai requisiti dell'iterazione corrente

Si tratta di una rappresentazione visuale di classi concettuali o di oggetti del mondo reale, nonchè delle relazioni tra di essi, in un dominio di interesse. Applicando UML, un modello di dominio può essere realizzato come uno o più diagrammi delle classi in cui non sono definite operazioni e mostrano:

- Classi concettuali o oggetti di dominio
- Associazioni tra classi concettuali
- Attributi di classi concettuali

Il modello di dominio non è una raffigurazione di oggetti software, si tratta di un dizionario visuale delle astrazioni significative, della terminologia del dominio e del contenuto informativo del dominio di interesse.
Un modello dei dati mostra dati che devono essere memorizzati in modo persistente.
Un modello di dominio descrive informazoni che devono essere gestite nel sistema in discussione e può contenere

- Classi concettuali senza attributi
- Classi concettuali che hanno un ruolo puramente comportamentale e non un ruolo informativo

Classi Concettuali

Una classe concettuale è un'idea, una cosa o un oggetto che può essere considerata in termini di:

- Simbolo - Un aparola o immagine usata per rappresentare la classe concettuale
- Intenzione - La definizione della classe concettuale in linguaggio naturale
- Estensione - L'insieme degli oggetti descritti dalla classe concettuale

In UML, una classe è il descrittore per un insieme di oggetti che possiedono le stesse caratteristiche (attributi, operazioni, metodi, relazioni e comportamento).

Associazioni In UML, un'associazione è la relazione tra due o più classificatori che comporta connessioni tra le rispettive istanze.

Attributi In UML, un attributo è la descrizione di una proprietà di una classe. Creare un modello di dominio è molto utile per comprendere il dominio del sistema da realizzare e il suo vocabolario (analisi), è inoltre una fonte di ispirazione per lo strato del dominio.

5.3 Come creare un modello di dominio

- Trovare le classi concettuali
- Disegnarle come classi in un diagramma delle classi UML
- Aggiungere associazioni e attributi

Identificare le classi concettuali

- Riusare o modificare dei modelli esistenti
- Utilizzare un elenco di categorie comuni
- Identificare nomi e locuzioni nominali

Associazione Un'associazione è una relazione tra classi (più precisamente, tra istanze di queste classi) che indica una connessione significativa e interessante. In UML, una relazione semantica tra due o più classificatori che coinvolge connessioni tra le loro istanze.

Quando mostrare un'associazione Considerare l'inclusione delle seguenti associazioni in un modello di dominio

- Associazioni per cui la conoscenza della relazione deve essere conservata per qualche durata
- Associazioni identificate mediante l'elenco di associazioni comuni

Evitare di inserire troppe associazioni in un modello di dominio.
Considera che le associazioni del modello di progetto andranno implementate nel software, ma considera che un modello di dominio non è un modello di progetto, infatti una associazione descrive una relazione significativa tra oggetti del mondo reale, non descrive caratteristiche di oggetti software.

Molteplicità

La molteplicità di un ruolo indica quante istanze di una classe possono essere associate a una istanze dell'altra classe. Essa dipende dal contesto

Aggregazione e composizione

Aggregazione Una associazione che rappresenta una relazione intero-parte

Composizione (aggregazione composta) Si tratta di una forma forte di aggregazione in cui:

- Una parte appartiene a un composto alla volta
- Ciascuna parte appartiene sempre a un composto
- Il composto è responsabile della creazione e cancellazione delle sue parti

Per esempio un computer e una stampante sono una aggregazione, dato che sono debolmente collegati. Mentre un albero e le sue foglie sono una composizione, dato che sono fortemente collegati.

- L'aggregato può in alcuni casi esistere indipendentemente dalle parti, ma in altri casi no
- Le parti possono esistere indipendentemente dall'aggregato
- L'aggregato è in qualche modo incompleto se mancano alcune delle sue parti
- É possibile che più aggregati condividano una stessa parte

Per la composizione

- Ogni parte può appartenere a un solo composto per volta
- Il composto è l'unico responsabile di tutte le sue parti: questo vuol dire che è responsabile della loro creazione e distruzione
- Il composto può rilasciare una sua parte, a patto che un altro oggett si prenda la relativa responsabilità
- Se il composto viene distrutto, deve distruggere tutte le sue parti o cederne la responsabilità a qualche altro oggetto
- La composizione è transitiva e asimmetrica

Come identificare la composizione

Si consideri di mostrare una composizione quando:

- C'è un ovvio gruppo fisico o logico intero-parte
- La vita della parte è limitata dalla vita del composto
- Alcune proprietà dell'intero si propagano anche alle parti (es. cancellazione/creazione)

5.4 Attributi

Si tratta di una proprietà elementare degli oggetti di una classe a cui viene associato un valore.

Tipi di attributi appropriati nel modello di dominio

Gli attributi devono avere tipo semplice, elementare, corrispondente a un tipo di dati primitivo. Non devono avere tipo corrispondente a un concetto complesso del dominio.
Il tipo degli attributi deve essere un tipo di dato e in UML con tipo di dato si intende un insieme di valori in cui l'identità univoca non è significativa.

Esempio Numero di telefono è un valore, Persona non è un tipo di dato.

5.5 Considerazioni finali

Esiste un modello di dominio univocamente corretto? Non esiste un solo modello di dominio corretto, essendo tutti i modelli approssimazioni del dominio che si sta tentando di capire. Meglio chiedersi se il modello di dominio è utile.
É meglio evitare un grosso sfrozo in valutazione secondo l'approccio a cascata per creare un modello di dominio completo e corretto. Non utilizzare più di alcune ore per ogni iterazione.

Chapter 6

Diagrammi di Sequenza di Sistema (SSD)

Un diagramma di sequenza di sistema (SSD) mostra gli eventi di input e output dei sistemi in discussione. Per un particolare corso di eventi all'interno di un caso d'uso mostra:

- Gli attori esterni che interagiscono direttamente con il sistema
- Il sistema (a scatola nera)
- Gli eventi di sistema generati dagli attori

6.1 Che cosa sono gli eventi e le operazioni di sistema

Casi d'uso descrivono il modo in cui gli attori esterni interagiscono con il sistema. Un attore genera degli **eventi di sistema** per richiedere l'esecuzione di alcune operazioni di sistema. Le operazione di sistema sono operazioni che il sistema deve definire per gestire gli eventi di sistema.
In UML un evento è qualcosa di importante o degno di nota che avviene durante l'esecuzione di un sistema. Un evento di sistema è un evento esterno al sistema, di input, di solito generato da un attore per interagire con il sistema.
In UML un'operazione rappresenta una trasformazione oppure un'interrogazione che un oggetto o componente può essere chiamato eseguire.
Un'operazione di sistema è una trasformazione oppure un'interrogazione che il sistema può essere chiamato a eseguire.

Fondamentale per riassumere le interazioni tra attori e le operazioni iniziate da essi Disegnare un SSD per uno schenario principale di successo di ciascun caso chiuso, nonchè per gli scenari alternativi più frequenti o complessi.

Applicazione UML; Eventi, operazioni e SSD

UML non definisce ne eventi di sistema o perazioni di sistema ne diagrammi di sequenza di sistema. UML definisce degli elementi o diagrammi chiamati semplicemente evento, operazione e diagrammi di sequenza. Sono considerati a scatola nera.

Perchè disegnare un SSD

Perchè gli eventi di sistema sono importanti:

- Il software deve essere progettato per gestire questi eventi
- Un sistema software reagisce a tre cose: Eventi esterni, temporali o guasti (eccezioni)

Risulta quindi fondamentale descrivere il comportamento in funzione di queste operazioni tramite

- I casi d'uso
- Diagrammi di sequenza di sistema
- Contratti delle operazioni di sistema

Servono proprio a speigare **Che cosa fa** il sistema, senza spiegare come lo fa. L'SSD è fortemente in relazione con i casi d'uso.

Assegnare il nome a eventi e operazioni di sistema

Eventi di sistema e operazioni corrispondenti devono essere espressi a un livello astratto di **Intenzione** anzichè di azione.
Iniziare il nome con un verbo.

Esempio addItem, enterItem, make... Vedremo che l'SSD coinvolgerà più sistemi (anche esterni).

6.2 SSD e Glossario

Dato che utilizzeremo termini concisi per rappresentare le operazioni e gli eventi sarà fondamentale aggiungere questi termini al Glossario e dettagliarne il significato.

6.3 Lo scopo di un SSD

Lo scopo è comprendere e non è adatto a tutti gli scenari. É adatto solo per quelli dell'iterazione corrente.
Gli SSD fanno parte del Modello dei casi d'uso e sono maggiormente creati durante l'elaborazione. Non sono un elaborato ufficiale di UP, anche se i creatori di Up ne riconoscono l'utilità.

Chapter 7

Contratti delle operazioni di sistema

I contratti delle operazioni di sistema sono un modo per specificare il comportamento delle operazioni di un sistema. I contratti sono scritti in un linguaggio formale e preciso e contengono informazioni sulla precondizione, la postcondizione e l'effetto dell'operazione sul sistema. I contratti servono per descrivere il comportamento del sistema in maniera più dettagliata.

7.1 Le sezioni di un contratto

- Operazione - Nome e parametri (firma) dell'operazione
- Riferimenti - Casi d'uso in cui può verificarsi questa operazione
- Pre-condizioni - Ipotesi significative sullo stato del sistema o degli oggetti nel modello di dominio prima dell'esecuzione dell'operazione. Si tratta di ipotesi non banali che dovrebbero essere comunicate al lettore
- Post-condizioni - É la sezione più importante. Descrive i cambiamenti di stato degli oggetti nel modello di dominio dopo il completamento dell'oerazione

Firma Viene definita come il Nome dell'operazione e i parametri che essa accetta. Esempio → addItem(item, quantity).

7.2 Che cos'è un'operazione di sistema

Si tratta di un'operazione pubblica del sistema la cui esecuzione è richiesta da un evento di sistema. L'esecuzione di una operazione di sistema cambia lo stato del sistema stesso. L'interfaccia del sistema è l'insieme di tutte le operazioni di sistema.
Un contratto è relativo a una operazione di sistema e descrive il cambiamento dello stato di oggetti del modello del dominio causato dall'esecuzione dell'operazione di sistema.

7.3 Post-condizioni

- Descrivono i cambiamenti nello stato degli oggetti del Modello di dominio.
- Non descrivono azioni eseguite durante l'esecuzione

Sono dichiarazioni circa lo stato degli oggetti dopo l'esecuzione dell'operazione e sono scritte al passato.

7.3.1 Possibili cambiamenti di stato

- Creazione o cancellazione di oggetto (istanza di una classe)
- Cambiamento di valore di attributo
- Formazione o rottura di collegamento (istanza di un'associazione)

Come già detto in precedenza sono super importanti dato che descrivono i cambiamenti richiesti dall'esecuzione del'operazione di sistema, senza però descrivere come ottenere questi cambiamenti. **Le post condizioni vanno espresse con un verbo al passato**.
Il cambiamento dello stato del sistema causato dall'esecuzione dell'operazione di sistema va espresso in termini di oggetti, collegamenti e attributi del dominio di interessa che il sistema ha necessità di conoscere e ricordare.

7.3.2 Aggiornamento del modello di dominio

Durante la creazione dei contratti è normale che emerga la necessità di registrare nuove classi concettuali, attributi o associazioni nel modello di dominio.

Ricorda Nei metodi iterativi e evolutivi, tutti gli elaborati dell'analisi e della progettazione sono considerati parziali e imperfetti, ed evolvono in risposta a nuove scoperte.

7.4 Utilità e scrittura contratti

Non sempre è necessario scrivere dei contratti, a volte i casi d'uso sono sufficienti. I contratti sono adatti alla rappresentazione di situazioni dettagliate o complesse, che non è opportuno descrivere nei casi d'uso. I contratti sono appunto complementari ai casi d'uso.

Operazioni per la scrittura

1. Identificare le operazioni di sistema dagli SSD
2. Creare i contratti per le operazioni più complesse, o che non sono chiare dai casi d'uso
3. Per descrivere le post-condizioni utilizzare le seguenti categorie
    - Creazione o cancellazione di oggetto
    - Modifica di attributo
    - Formazione o rottura di collegamento
4. Le post-condizioni descrivono cambiamenti di stato al passato
5. Ricordare di formare collegamenti necessari tra oggetti esistenti e quelli appena creati
6. Le pre-condizioni sono normalmente implicate dall'esecuzione di operazioni di sistema precedenti

Chapter 8

Progettazione orientata agli oggetti

Fino ad ora attraverso l'analisi dei requisiti e l'analisi a oggetti ci siamo interessati all'imparare a fare la cosa giusta, mentre il lavoro successivo è fare la cosa bene, attraverso **la progettazione**.
Ricordiamoci sempre che non dobbiamo prevedere tutto all'inizio (anche perchè spesso è impossibile), ma dobbiamo necessariamente provocare il cambiamento all'inizio del progetto per poter scoprire e modificare alcuni requisiti di progettazione e implementazione. I **metodi iterativi ed evolutivi** esistono per questo, abbracciano il cambiamento. Si cerca però di provocare i cambiamenti maggiori nelle fasi iniziali per avere obiettivi più stabili per le iterazioni successive.

Chapter 9

Architettura Logica

Siamo passati da un lavoro di analisi ad un lavoro di progettazione, si inizia quindi a pensare su larga scala. La progettazione di un tipico sistema orientato agli oggetti è basata su diversi **strati architetturali**, come per esempio uno strato per l'interfaccia utente, uno per la logica applicativa, ecc.

9.1 Che cos'è l'architettura logica

L'**architettura logica** di un sistema software è l'organizzazione su larga scala delle classi software in package (o namespace), sottoinsiemi e strati.
Uno stile comune per la l'architettura logica è l'**l'architettura a strati**.

9.1.1 Architettura a strati

Uno strato è un gruppo a grana molto grossa di classi, package o sottosistemi, che ha delle responsabilità coese rispetto a un aspetto importante del sistema.
Gli strati sono organizzati in modo gerarchico, quelli più alti ricorrono a servizi degli strati più bassi (normalmente non avviene il contrario).
Gli strati normalmente comprendono

- Presentazione - interfaccia utente
- Logica applicativa o strato del dominio - elementi che rappresentano concetti del dominio
- Servizi tecnici

Due tipi di architettura a strati

- Stretta - uno strato può richiamare solo i servizi dello strato immediatamente sottostante
- Rilassata - uno strato può richiamare i servizi di strati più bassi di diversi livelli

Noi ci concentreremo sullo **strato della logica applicativa (o strato del dominio)**.

9.2 Che cos'è un'architettura software

- L'insieme delle decisioni significative sull'organizzazione di un sistema software
- La scelta degli elementi strutturali da cui è composto il sistema e delle relative interfacce
- La specifica della collaborazione tra gli elementi strutturali
- La composizione di questi elementi strutturali e comportamentali in sottosistemi via via più ampi
- Lo stile architetturale che guida questa implementazione

L'architettura software ha a che fare con la larga scala.

9.3 Diagrammi dei Package

L'architettura logica può essere illustrata mediante un diagramma dei package di UML, dove uno strato può essere modellato come un package UML. Un package UML può ragguppare qualunque cosa (classi, altri package, casi d'uso).

- É molto comune l'annidamento di package
- Per mostrare la dipendenza tra i package viene utilizzata una dipendenza UML
- Un package UML rappresenta un namespace, in questo modo è possibile definire due classi con lo stesso nome su package diversi

9.3.1 Progettzione degli strati

Organizzare l astruttura logica di un sistema in strati separati con responsabilità distine e correlate, con una separazione netta e coesa degli interessi: Come detto in precedenza:

- Gli strati inferiori sono servizi generali e di basso livello
- Gli strati superiori sono più specifici per l'applicazione

Collaborazione e accoppiamenti vanno dagli strati più alti a quelli più bassi. L'obiettivo è suddividere un sistema in un insieme di elementi software che per quanto possibile, possano essere sviluppati e modificati ciascuno **indipendentemente dagli altri**. Nella slide 16 viene riportato un esempio di scelta comune per gli strati, qui di seguito il link per la slide: Link slide

Vantaggi dell'uso a strati

La sperazione provocata dall'utilizzo degli strati crea diversi vantaggi fra cui:

- riduce l'accoppiamento e le dipendenze, migliora la coesione, aumenta la possibilità di riuso e aumenta la chiarezza.
- La complessità relativa a questi aspetti è incapsulata e può essere decomposta
- alcuni strati possono essere sostituiti da nuove implementazioni
- Gli strati più bassi contengono funzioni riusabili
- Alcuni strati possono essere distribuiti
- Lo sviluppo del team è favorito dalla segmentazione logica

Responsabilità coese e seperazione degli interessi

In uno strato le responsabilità degli oggetti devono essere fortemente coese l'uno all'altro e non devono essere mescolate con le responsabilità degli altri strati.

9.4 Oggetti e logica applicativa

Un oggetto software è un oggetto con nomi e informazioni simili al dominio del mondo reale e assegnare ad esso responsabilità della logica applicativa, un oggetto di questo tipo è chiamato un oggetto di dominio.
Rappresenta una cosa nello spazio del dominio del problema e ha una logica applicativa o di business correalata.
Progettando gli oggetti in questo modo si arriva a uno strato della logica applicativa che può essere chiamato **strato del dominio** dell'architettura

9.4.1 Definizione livelli, trati e partizioni

- Livello (tier): solitamente indica un nodo fisica di elaborazione
- Strato (layer): una sezione verticale dell'architettura
- Partizione (partition): una divisione orizzontale di sottosistema di uno strato

9.4.2 Principio di separazione Modello-Vista

Gli oggetti di dominio (modello) non devono essere connessi o accoppiati direttamente agli oggetti UI (vista).
Un rilassamento legittimo di questo principio è il **Pattern Observer**, qua gli oggetti del dominio inviano messaggi a oggetti della UI, visti però solo indirettamente, in termini di un'interfaccia come PropertyListener. L'oggetto di dominio non sa che quell'oggetto della UI è un oggetto UI, non conosce la sua classe UI concreta, sa solo che l'oggetto implementa l'interfaccia (non UI) PropertyListener.

9.5 Legame tra SSD, operazioni di sistema e strati

I messaggi inviati dallo strato UI allo strato del dominio sono i messaggi mostrati negli SSD.

9.6 Verso la progettazione a oggetti

Gli sviluppatori hanno 3 modalità per progettare gli oggetti:

- Codifica - La progettazione avviene durante la codifica
- Disegno, poi codifica - Disegnare alcuni diagrammi UML e poi passare al punto 1
- Solo disegno - Lo strumento genererà il codice a partire dai diagrammi

Noi tratteremo il disegno leggero di UML

Agile Modeling e il disegno leggero di UML

Gli obiettivi sono ridurre il costo aggiuntivo del disegno e modellare per comprendere e comunicare, anzichè documentare. La modellazione agile comprende anche le seguenti pratiche:

- Modellare insieme agli altri
- Creare diversi modelli in parallelo

Ci sono due tipi di modelli per gli oggetti:

- Statici
- Dinamici

Creare questi modelli in parallelo

Modelli dinamici

- Diagrammi di sequenza e comunicazione
- I più importanti e difficili da creare
- Si applicano
  - La progettazione guidata delle responsabilità
  - I principi di GRASP

Modelli statici

- Diagrammi delle classi
- Utili come sintesi e come base per la struttura del codice

Chapter 10

Diagrammi di interazione

10.1 Che sono sono gli oggetti

Un oggetto è un pacchetto coeso di dati e funzioni incapsulate in una unità riusabile. La parte dei dati è composta da attributi, mentre il comportamento è definito dalle operazioni. Tutti gli oggetti hanno:

- Identità: ogni oggetto ha il suo identificativo univoco
- Stato: viene stabilito dai valori effettivi dei dati memorizzati in un oggetto ad un certo istante
- Comportamento: l'insieme delle operazioni che l'oggetto può eseguire

I dati sono nascosti all'interno dell'oggetto (insapsulamento) e sono accessibili solo attraverso le operazioni, questo favorisce software più robusto e codice riusabile. Gli oggetti collaborano tra loro attraverso messaggi, essi causano l'invocazione delle operazioni da parte dell'oggetto.
In UML i **diagrammi di interazione** illustrano il modo in cui gli oggetti interagiscono attraverso lo scambio di messaggi. Sono utilizzati per la **modellazoone dinamica degli oggetti**.

10.2 Interazioni

Un'interazione è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto. I diagrammi di interazione catturano un'interazione come:

- Linee di vita - partecipanti nell'interazione
- Messaggi - Comunicazioni tra linee di vita

10.2.1 Linee di vita

Una linea di vita rappresenta un singolo partecipante a un'interazione: raffigura come un'istanza del classificatore partecipa all'interazione. Le linee vita hanno:

- Nome: usato per far riferimento alla linea vita nell'interazione
- Tipo: nome del classificatore di cui rappresenta un'istanza
- Selettore: una condizione booleana che seleziona una specifica istanza

10.2.2 Messaggi

Un messaggio rappresenta la comunicazione tra due linee vita e può essere di diverse tipologie:

- Sincrono
- Asincrono
- Di ritono
- Creazione dell'oggetto
- Distruzione dell'oggetto
- Messaggio trovato
- Messaggio perso

10.3 Diagrammi di interazione

- Diagrammi di sequenza
- Diagrammi di comunicazione
- Diagrammi di interazione generale
- Diagrammi di temporizzazione

10.3.1 Diagrammi di sequenza

- Mostra l'interazione tra un insieme di oggetti.
- Enfatizza la sequenza temporale degli scambi di messaggi
- Mostra interazioni ordinate in una sequenza temporale
- Non mostra le relazioni degli oggetti (possono essere dedotte dagli invii di messaggi)

Formato a steccato.

10.3.2 Diagramma di comunicazione

Un diagramma che mostra l'interazione tra un insieme di oggetti (formato a grafo o rete). Mostra gli aspetti strutturali di un'interazione - mostrando come si collegano le linee di vita. Il collegamento indica l'esistenza di qualche forma di navigabilità tra gli oggetti. Nella side 51 troviamo il dettaglio di come scorrono i messaggi e di come numerare una sequenza.

Messaggi condizionali Anche in questo caso è possibile avere delle condizioni di diverso tipo, possono essere per esempio mutuamente esclusive o iterazioni.

- Enfatizza le relazioni strutturali tra gli oggetti
- Sono utilizzati per esplicitare le relazioni degli oggetti

É consigliato creare i digrammai delle classi e di interazione in parallelo per dare risalto non solo agli aspetti statici (diagramma delle classi), ma anche a quelli dinamici (diagrammi di interazone).

Auto-delegazione è quando una line vita invia un messaggio a se stessa, generando un'attivazione annidata.

Distruzione Per indicare la distruzione di un oggetto si termina la linea vita con una grossa croce.

Frammenti combinati (Frame)

I diagrammi di sequenza possono essere divisi in aree chiamate frammenti combinati. I frammenti combinati hanno uno o più operandi, l'operatore determina come verranno eseguiti gli operandi. Le condizioni di guardia stabiliscono se i loro operandi devono essere esegiti. L'esecuzione avveniene solo se la condizioni di guardia è valutata vera.

Operatori più usati

- opt - Option - Il suo unico operando viene eseguito se e solo se la condizione è vera
- alt - Alternatives - Messaggi condizionali mutuamente esclusivi (2 o più) operandi (aka if else)
- loop - loop
- break - break
- ref - Reference

Iterazioni con loop e break

Sintass del loop:

- Ciclo senza max o min equivale a una condizione di ciclo infinito
- Se viene specificato solo min allora si considera max=min
- La condizione può essere booleana o un testo (purchè il significato sia chiaro)

Il break indica in quale condizione il loop viene interrotto e che cosa succede in seguito, il resto del ciclo non viene eseguito dopo il break. Nelle slide vengono riportati diversi esempi di iterazioni (da slide 33 in poi) Link Slide

Reference Fondamentali per correlare i diagrammi fra di loro (slide 44).

10.4 Diagrammi di Interazione Generale

Modellano il flusso di controllo tra interazioni ad alto livello, mostrano le interazioni e le occorrenze di interazioni, hanno la sintassi dei diagrammi di attività.

Chapter 11

Diagramma delle classi

Cosa sono le classi Una classe modella un insieme di oggetti omogenei (le istanze della classe) ai quali sono associate proprietà statiche (attributi) e dinamiche (operazioni).
La classificazione è uno dei più importanti modi che noi abbiamo per organizzare la nostra vista del mondo.
Per identificare un oggetto che è istanza di una classe si traccia un arco *instantiate*.
Per descrivere una classe in notazione UML è consigliabile usare nomi descrittivi ed evitare abbreviazioni.

Classificatore Un classificatore si definisce come elemento di modello che descrive caratteristiche comportamentali e strutturali, sono una generalizzazione di molti degli elementi di UML, come classi, interfacce, casi d'uso e attori.
Nei diagrammi delle classi i due classificatori più comuni sono le classi e le interfacce.

11.1 Proprietà Strutturali

In UML le proprietà strutturali di un classificatore comprendono:

- Gli attributi
- Le estremità di associazioni

11.1.1 Attributi

Un attributo modella una propeità locale della classe ed è caratterizzato da un nome e dal tipo dei valori associati. Ogni attributo stabilisce una proprietà

locale valida per tutte le istanze (per locale si intende che è indipendente dagli altri oggetti).

UML In UML negli attributi si indica obbligatoriamente un nome, opzionalmente si può indicare la visibilità (di default si ipotizza che siano privati).

Identificatore di oggetti Due oggetti con identificatori distinti sono comunque distinti, anche se hanno i valori di tutti gli attributi uguali.

Visibilità Sono i 4 classici tipi già studiati in programmazione:

- Pubblica
- Privata
- Protetta - Solo operazioni appartenenti alla classe o ai suoi discendenti possono accedere ai membri con visibilità protetta
- Package - Ogni elemento nello stesso package della classe può accedere ai membri della classe con visibilità package

Molteplicità Consideriamo un attributo B di tipo T di una classe C. Se la molteplicità non viene indicata si intende 1:1, altrimenti si indica per esempio x..y per dire che B associa ad ogni istanza di C al minimo x e al massimo y valori di tipo T. Nelle istanza il multivalore si indica mediante un insieme (tipo array).

11.1.2 Associazioni in UML

Per il momento, ci limitamo a discutere associazioni tra due classi (ma le associazioni possono coinvolgere N classi).
Un'associazione tra una classe C1 e una classe C2 modella una relazione matematica tra l'insieme delle istanze di C1 e l'insieme delle istanze di C2. Gli attributi modellano proprietà locali di una classe, le associazioni modellano proprietà che coinvolgono altre classi.

Collegamenti Le istanze delle associazioni si chiamano collegamenti. Come gli oggetti sono istanze delle classi, così i collegamenti sono istanze delle associazioni. Al contrario degli oggetti però i collegamenti non hanno identificatori espliciti: un collegamento è implicitamente identificato dalla coppia (o in generale dalla ennupla) di oggetti che esso rappresenta. Ciò non toglie che tra due classi possono essere definite più associazioni.

Sintassi Le associazioni possono essere indicate o con il nome dell'associazione o con il nome dei ruoli (es. Azienda - Persona : Datore - Dipendente) e la relativa cardinalità. A differenza della cardinalità però il ruolo non aggiunge nulla al significato pratico dell'associazione. A volte è utile specificare un verso, ma questo non significa che l'associazione ha un verso, è solo relativo al verso che il nome dell'associazione evoca, ma anche in questo caso non ha un significato in termini pratici. L'unico caso in cui il ruolo è obbligatorio è nel caso di associazioni che insistono più volte sulla stessa classe e rappresentano una relazione non simmetrica.

Perchè usare i ruoli Risolvono alcune ambiguità a livello schematico, rendendo più chiaro cosa si intende rappresentare. In alcuni casi però i ruoli non sono significativi, come nei casi di relazioni simmetriche (esempio Stato confina con).

Navigabilità

La navigabilità indica che è possibile spostarsi da n qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione. Gli oggetti della classe di orgine possono fare riferimento a oggetti della classe di destinazione utilizzando il nome del ruolo.

Molteplicità

La molteplicità limita il numero di oggetti di una classe che possono partecipare in una relazione in un dato insieme. Se la molteplicità non viene indicata esplicitamente allora è indefinita.

Attributi di associazioni

Analogamente alle classi, anche le associazioni possono avere attributi. Formalmente, un attributo di una associazione è una funzione che associa ad ogni collegamento che è istanza dell'associazione un valore di riferimento.

Parallelo con Basi di Dati Praticamente è un attributo di relazione, dato che è un valore che ha senso di esistere solo se esiste la relazione (in questo caso collegamento).

Classi associazioni nella progettazione Nessuno dei linguaggi OO comunemente utilizzati supporto le classi associazione, per questo vengono reificate tramite

l'uso di classi di progettazione, collegandole alle classi con l'aggregazione e la composizione (da utilizzare opportunamente in base alla cardinalità decidendo chi sia il tutto e chi sia la parte). Come in basi di dati, anche qua esistono le associazioni n-arie che modellano una relazione matematica tra n insiemi.

Associazioni e Attributi

Utilizzare le associazioni quando la classe di destinazione è una parte importante del modello.
Utilizzare gli attributi quando:

- La classe di destinazione non è una parte importante del modello (ad esempio un tipo primitivo come stringa)
- La classe di destinazione è solo un dettaglio di implementazione, ad esempio una componente di libreria

11.1.3 Semantica della collezione : Stringhe di proprietà

UML mette a disposizione delle proprietà standard che possono essere applicate alle molteplicità per indicare la semantica richiesta dalla collezione:

- ordered - gli elementi dell'insieme sono ordinati
- unordered - non c'è un ordine definito
- unique - gli elementi dell'insieme sono tutti univoci
- nounique - la collezione può contenere elementi duplicati

É consentito l'utilizzo di parole chiavi definite dall'utente (esempio List).

11.1.4 Operazioni

Un'operazione di UML è una dichiarazione, con un nome, dei parametri, un tipo di ritorno, un elenco di eccezioni e magari un insieme di vincoli di precodizioni e post condizioni.

title *Definizione di un'operazione I nomi delle operazioni sono solitamente in minuscolo (evitare simboli speciali). UML consente che le firme delle operazioni siano scritte in un qualunque linguaggio, purchè sia notificato al lettore o allo strumento.

Metodi In UML un metodo è l'implementazione di un'operazione.

Parole chiave

Un decoratore testuale è una parola chiave utilizzata per classificare un elemento di modello. Qui di seguito alcuni esempi:

- actor - classifica un attore
- interface - un'interfaccia
- abstract - una classe astratta (che quindi non può essere istanziata)
- ordered - insieme di oggetti con un ordine predefinito

11.2 Generalizzazione, Ereditarietà, Polimorfismo, Overriding

Generalizzazione

In UML la relazione is-a si modella mediante la nozione di generalizzazione (ancora una volta parallelo con basi di dati). La generalizzazione coinvolge una superclasse ed una o più sottoclassi dette classi derivate, il significato è che ogni istanza di ciascuna sottoclasse è anche istanza della superclasse. Quando la sottoclasse è una, la generalizzazione modella appunto la relazione is-a tra sottoclase e la superclasse.
In UML si usa una freccia dal basso verso l'alto.
La superclasse può generalizzare più sottoclassi rispetto ad un unico criterio (Persona - Maschio/Femmina) Oppure si possono creare più generalizzazioni diverse (secondo criteri diversi). Una generalizzazione può essere disgiunta oppure no (Maschio e Femmina è disgiunta, mentre per esempio Studente Lavoratore no). Onverlapping (non disgiunta) è default.

Generalizzazione complete Una generalizzazione può essere completa o no (Studente Lavoratore è incompleta, Maschio Femmina è completa). Incompleta è default.

Ereditarietà

Principio di Ereditarietà, ogni proprietà della superclasse è anche una proprietà della sottoclasse e non si riporta esplicitamente nel diagramma.

Ereditarietà Multipla UML consente l'ereditarietà multipla

Overriding

L'overriding è la possibilità di ridefinire un'operazione ereditata da una superclasse La firma dell'operazione è data dal nome dell'oerazione, il tipo di ritono e tutti i parametri. I nomi dei parametri non vengono considerati parte della firma.

Polimoorfismo

Un'operazione poliformica ha molte implementazioni (forme). Le sottoclassi avranno l'esigenza di ridefinire alcune funzioni ereditate per poterle rendere specifiche per le lore necessità.

11.3 Dipendenza

In UML una dipendenza è una relazione di dipendenza. Si utilizza quando un elemento cliente (classi, package, casi d'uso) è a conoscenza di un elemento fornitore e un cambiamento nel fornitore potrebbe influire sul cliente (rompendone il funzionamento).
Un elemento è **accoppiato** con un altro elemento o dipendenza da un altro elemento.

11.4 Interfaccia

Un'interfaccia è un insieme di funzionalità pubbliche identificate da un nome e separano le specifiche di una funzionalità dall'implementazione, un'interfaccia definisce un contratto e tutti i classificatori che la realizzano devono rispettare.

Come individuare le interfacce Esaminando ogni associazione chiediamoci, dovrebbe questa associazione essere associata ad una classe particolare o dovrebbe essere più flessibile?
Lo stesso per i messaggi. Lo scopo è raggruppare più operazione che potrebbero essere riusabili altrove o attributi.

Interfacce per stabilire protocolli

Le interfacce sono molto utili perchè tramite il loro utilizzo è possibile stabilire protocolli comuni che potrebbero essere realizzati da molte classi e molti

componenti.
Le interfacce facilitano l'inserimento di nuove classi nel sistema. Permette anche di nascondere dettagli implementativi di sottosistemi complessi.

Architettura L'insieme dei sottosistemi e delle interfacce di progetto costituisce l'architettura di alto livello di un sistema. Per capire e mantenere questa architettura, l'insieme dovrebbe essere organizzato in modo coerente, applicando un architettura a strati.

Vantaggi e Svantaggi interfacce

Vantaggi

- Quando progettiamo con le classi stiamo progettando specifiche implementazioni
- Quando progettiamo con interfacce stiamo invece progettando dei contratti che possono essere realizzati da molte implementazioni diverse (classi)
- La progettazione per interfacce svincola il modello da dipendenze dell'implementazione e ne aumenta quindi la flessibilità e estendibilità

Svantaggi

- Maggiore flessibilità significa anche più complessità. In teoria ogni operazione di ogni classe potrebbe essere un'interfaccia, ma in questo modo il sistema diventerebbe incomprensibile
- Maggior costo a livello di prestazioni

11.5 Aggregazione e Composizione

In UML un'aggregazione è un'associazione che rappresenta una relazione intero-parte
Una composizione (aggregazione composta) è una forma forte di aggregazione in cui:

- Una parte appartiene a un composto alla volta
- Ciascuna parte appartiene sempre a un composto
- Il composto è responsabile della creazione e cancellazione delle sue parti

Chapter 12

Diagramma di macchina a stati

Le macchine a stati possono essere utilizzate per modellare il comportamento dinamico di classificatori quali classi, casi d'uso, sottosistemi e interi sistemi. Le macchine a stati esistono nel contesto di un particolare classificatore che:

- Risponde a eventi esterni
- Ha un ciclo di vita definito, che può essere modellato come una successione di stati, transizioni ed eventi
- Può avere un comportamento corrente che dipende dai comportamenti precedenti

Le macchine a stati di solito vengono utilizzate per modellare il comportamento dinamico degli oggetti.

12.1 Tipi di oggetti

- Indipendenti dallo stato - Oggeti che rispondono sempre nello statto modo a un determinato evento
- Dipendenti dallo stato - Rispondono in modi diversi ad un determinato evento a seconda dello stato in cui si trovano

Modellazione oggetti dipendenti dallo stato

Modellano il comportamento di un oggetto reattivo complesso in risposta agli eventi oppure modellano le sequenze valide delle operazioni ovvero specifiche di protocollo o linguaggio.
Esempi di oggetti complessi sono dispositivi fisici come un telefono o un'auto, oppure transazioni e oggetti di business (ordine, vendita, pagamenti, ecc.).

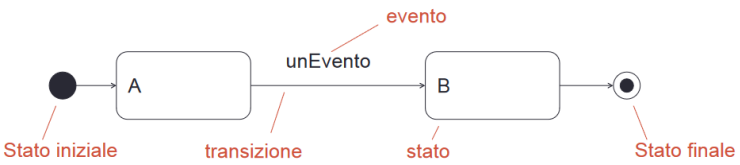
Altri esempi

- Protocolli di comunicazione - TCP si comporta in maniera diversa a seconda dello stato in cui la comunicazione si trova
- Flusso e navigazione delle pagine/finestre UI
- Controlle di flusso UI
- Operazioni di sistema dei casi d'uso - endSale deve arrivare solo dopo una o più operazioni enterItem

12.2 Macchina a stati - Sintassi e rappresentazione

Noterete dalla seguente immagine che si tratta praticamente di un DFA (Deterministic Finite Automa - Automa a stati finiti deterministico) dove abbiamo:

- Uno stato iniziale
- Stati intermedi
- Transizioni (etichettate da eventi)
- Stato finale



Stati

Una condizione o una situazione della vita di un oggetto durante la quale tale oggetto soddisfa una condizione, esegue un'attività o aspetta un evento. Lo stato di un oggetto in qualsiasi momento è determinato da

- I valori dei suoi attributi
- Le relazioni che ha con altri oggetti
- Le attività che sta eseguendo

Le azioni sono istantanee e non interrompibili, le transizioni interne occorrono dentro lo stato e non causano la transizione in un nuovo stato, mentre le attività richiedono un intervallo di tempo finito e sono interrompibili.
Le transizioni possono essere collegate tramite uno pseudo stato di giunzione o tramite uno pseudo stato di selezione (tipo if else).

Eventi

Gli **eventi** attivano le transizioni nelle macchine a stati.
Ci sono eventi di diversa tipologia:

Eventi di chiamata Una chiamata per una specifica operazione. L'evento dovrebbe avere la stessa segnatura di un'operazione della classe di contesto.

Eventi di segnale Un segnalte è un pacchetto di informazioni inviato in modo asincrono tra oggetti. Non ha operazioni perchè il suo scopo è trasportare informazioni.

Ricezione del segnale indicata da un pentagono concavo.

Evento di Variazione Un evento di variazione è un'espressione booleana: L'azione viene eseguita quando il valore dell'espressione passa da falso a vero. Da un punto di vista implementativo, l'espressione viene valutata in modo periodico (ciclo di test continuo).

Evento temporale Un evento temporale è un evento che si verifica dopo un certo periodo di tempo, quindi quando un'espressione di tempo diventa vera.

Stati composti

Hanno una o più regioni ognuna delle quali contiene una sottomacchina annidata.
Il più semplice contiene una sola regione, mentre quelli compositi ortogonali hanno due o più regioni e quando si entra nello stato composito tutte le macchine iniziano la loro esecuzione in modo concorrente. L'uiscta può essere sincronizzata, cioè si esce dallo stato quando tutte le regioni sono terminate, oppure asincrona, cioè si esce dallo stato composito quando una regioen termina e l'altra sottomacchine viene terminata.

Comunicazione tra sotto macchine

Capita spesso di avere l'esigenza di far comunicare due sotto-macchine, biforcazioni e ricongiuizioni possono essere usate per creare sotto-macchine concorrenti e per ri-sincronizzarle. La comunicazione asincrona è ottenuta da una sotto-macchina configurando un flag per un'altra sotto-macchina.

Stato di sync Come alternativa all'utilizzo degli attributi possiamo utilizzare uno stato di sync il cui compito è quello di tenere traccia di ogni singola attivazione della sua unica transizione di input. Lo stato di sync è come se fosse una coda dove si aggiunge un elemento alla coda ogni volta che viene attiva la transizione di input.

Stati con memoria semplice

Lo pseudo-stto con memoria semplice ricorda in quale sottostato si era quando si è lasciato il super stato, in seguito quando si ritorna da uno stato esterno allo stato con memoria, l'indicatore ridirezione la transizione sull'ultimo sottostato memorizzato.

Memoria multilivello Uno stato con memoria multilivello può ricordarsi di più sottostati.

12.3 I diagrammi di macchina in UP

Non esiste nessun modello in UP chiamato "modello a stati", tuttavia qualsiasi elemento in qualsiasi modello può avere una macchina a stati per comprendere o comunicare meglio il proprio comportamento dinamico.

Esempio Macchina a stati per rappresentare un processo di vendita, dalla selezione dell'item, al pagamento.

Chapter 13

Diagramma di attività

I diagrammi di attività spesso vengono chiamati "diagrammi di flusso OO" e consentono di modellare un processo come un'attività costituita da un insieme di nodi connessi da archi
Le attività vengono di solito associate a:

- Casi d'uso
- Classi
- Interfacce componenti
- Collaborazioni
- Operazioni
- Processi di Business

13.1 Attività

Le attività sono reti di nodi connessi ad archi, esistono tre categorie di nodi:

- Nodi azione - Rappresentano unità discrete di lavoro atocmiche all'interno dell'attività
- Nodi controllo - Controllano il flusso attraverso l'attività
- Nodi oggetto - Rappresentano oggetti usati nell'attività

Gli archi rappresentano il flusso attraverso le attività, ne esistono due categorie:

- Flussi di controllo: rappresentano il flusso di controllo attraverso le attività
- Flussi di oggetti: rappresentano il flusso di oggetti attraverso l'attività

13.1.1 Sintassi dell'attività

L'attività sono reti di nodi connessi da archi, il flusso di controllo è un tipo di arco che iniziano spesso con un nodo iniziale. Ci possono Essere pre e post condizioni. Quando un nodo azione finisce esso emette un token che potrebbe attraversare un arco per dare inizio alla prossima azione.

Il token game Consiste nel descrivere il flusso di token attorno a una rete di noti e archi. Il token è un oggetto, alcuni dati o un flusso di controllo che si spsota da un nodo sorgente a un nodo di destinazione attraverso un arco, ci possono essere dei vincoli che controllano il flusso dei token. Il movimento può quindi verificarsi quando tutte le condizioni sono soddisfatte.
Un nodo inizia la sua esecuzione quando ci sono tutti i token su tutti i suoi archi d'entrata.

Nodi Azione e token

L'esecuzione dei nodi azione avviene quando esiste un token simultaneamente su tutto gli archi entranti e i tken in ingresso soddisfano tutte le precondizoni locali del nodo.
Eseguono quindi un AND logico sui loro token di entrata e una fork implicita su tutti i suoi archi uscenti quando l'esecuzione è terminata.
subsection*Nodo azione di chiamata Il nodo azione di chiamata può invocare:

- Un'attività
- Un comportamento
- Un'operazione

Nodo di decisione

Nodo di controllo che ha un arco entrate e due o più archi alternativi uscenti, ogni arco uscente è protetto da una condizione di guardia, la condizione di guardia deve essere mutuamente esclusiva. L'arco di uscita viene percorso solo se la relativa condizione di guardia è vera, altrimenti la parola chiave specifica cosa fare in caso nessuna condizione di guarda venisse soddisfatta.

Nodi di biforcazione e ricongiuizione

Modellano flussi concorrenti duplicando i token in arrivo su tutti gli archi uscenti, quelli di ricongiunzione invece sincronizzano i flussi entranti offrendo un token in uscita quando c'è un token su tutti i loro archi entranti.

Nodo Oggetto

Indicano che sono disponibili istanze di un particolare classificatore, ogni nodo oggetto può tenere un numero infinito di token oggetto.
I Token sono offerti agli archi uscenti secondo un ordine FIFO o LIFO, che viene specificato (comportamento di selezione). Questi nodi possono essere parametri di input e output per le attività.

Pin

Un Pin è un nodo oggetto che rappresenta un input in un'azione o output da un'azione.

Partizione delle attività

Ogni partizione rappresenta un raggruppamento ad alto livello di azioni correlate:

- Partizioni possono essere gerachie
- Partizioni possono essere verticali, orizzontali o entrambe

Chapter 14

Pattern GRASP

UML è un linguaggio di modellazione visuale estremamente utile per l'attività di progettazione, ma la sua conoscenza NON implica saper progettare ad oggetti. La progettazione risulta utile per comprendere e comunicare.
In questo capitolo ci concentreremo sulla progettazione a oggetti.
Un modo comune di pensare alla progettazione di oggetti è in termini di **responsabilità, ruoli e collaborazioni** o **RDD (Responsability-Driver Development)**.

- Responsabilità - Astrazione di ciò che si deve saper fare, gli oggetti hanno responsabilità
- Ruoli - L'obiettivo o la capacità che un oggetto o una classe ha di partecipare ad una relazione con un altro oggetti. Obblighi e comportamenti di un oggetto sono assegnati in base al ruolo.
- Collaborazioni - Gli oggetti collaborano per raggiungere un obiettivo

2 Tipi di responsabilità

- Responsabilità di fare - Eseguire un calcolo, creare modificare eliminare oggetti, iniziare un'azione o controllare e coordinare altri oggetti.
- Responsabilità di conoscere - Conoscere i propri dati privati, conoscere informazioni di oggetti correalti, conoscere cose che può derivare o calcolare.

Le responsabilità si assegnano durante la modellazione e durante la codifica, nel contesto UML sono individuate mentre si creano modelli statici e dinamici del sistema.

14.1 Definizione di GRASP

I principi **GRASP (General Responsibility Assignment Software Patterns)** sono un aiuto per l'apprendimento degli aspetti essenziali per la progettazione a oggetti e per l'applicazione di ragionamenti di progettazione in maniera metodica.

Che cos'è un pattern?

Un pattern è una descrzione, con un nome, di un problema di progettazione ricorrente e di una sua soluzione ben provata che può essere aplicata a nuovi contesti. Idealmente, un pattern dà consigli su come applicare la sua soluzione in circostanze diverse e considera le forze e i compromessi. I pattern restano comunque delle linee guida e non delle regole ferree, chiaramente seguirli può portare a risultati migliori.
Di seguito tratteremo i seguenti pattern:

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller

14.2 Creator

Uno dei problemi comuni nella programmazione è: chi deve essere responsabile della creazione di una nuova istanza di una classe? Assegnare a B la responsabilità di creare istanze di tipo A sulla base delle seguenti condizioni:

- B contiene o aggrega con una composizione oggetti di tipo A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A

Più condizioni possono essere vere contemporaneamente.

14.3 Information Expert

Nominato anche semplicemente come Expert risponde al seguente problema:

Problema Qual è un principio di base per assegnare responsabilità agli oggetti? In termini pratici: Chi conosce un oggetto Square, dato un suo identificatore?
Si tratta di una responsabilità di conoscere, ma Expert si applica anche a responsabilità di fare. Il criterio (cioè la soluzione) consiste nell'assegnare le responsabilità all'esperto delle informazioni, ovvero alla classe che possiede informazioni necessarie per soddisfare le responsabilità.
Spesso risulta utile il modello del comportamento dinamico del sistema per identificare correttamente l'assegnazione delle responsabilità.

14.4 Low Coupling

Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e un riuso elevato?
L'accoppiamento indica quanto fortemente un elemento è connesso ad altri elementi, ha conoscenza di altri elementi e dipende da altri elementi. Classi con un alto accoppiamento comportano:

- I cambiamenti in classi correlate obbligano a cambiamenti locali
- Sono difficili da comprendere in isolamento
- Sono difficili da riusare perchè il loro riuso richiede anche il riuso delle classi da cui dipendono

Soluzione Assegnare responsabilità in modo da mantenere l'accoppiamento basso.

Forme comuni di accoppiamento

A livello implementativo, l'accoppiamento è spesso causato da:

- Un oggetto A ha un aattributo o referenzia un oggetto B
- Un oggetto A invoca servizi di B
- Un oggetto A implementa un metodo che riutilizza un parametro, una variabile locale o ritorna un oggetto B
- Un oggetto è istasnza di una classe C1 che a sua volta è sottoclasse di una classe C2
- Un oggetto è istasnza di una clase C1 che implementa una interfaccia I1

Disaccooppiamento Estremo

Un sistema OO è prima di tutto un sistema di oggetti che comunicano attraverso messaggi, il Low Coupling estremizzato può portare alla creazione di oggetti molto grande e copmplessi che eseguono tutto il lavoro da soli e quanto è una male perchè porta ad avere:

- Progetto pessimo a bassa coesione
- Progetto diffiicle da comprendere e mantenere

Un grado moderato di accoppiamento è necessario per la creazione di un sistema OO.

14.5 High Cohesion

Come mantere gli oggetti focalizzati, comprensibili e gestibili? La coesione indica quanto siano correlate e concentrate le responsabilità di un elemento. Un elemento con responsabilità altamente correlate che conon esegue una quantità di lavoro eccessiva ha coesione alta.
Una classe con coesione bassa fa molte cose non correlate tra loro o svolge troppo lavoro e questo le porta ad essere:

- Difficili da comprendere
- Difficili da riusare
- Difficili da mantenere
- Continuamente soggette a cambiamenti

Una classe con coesione alta ha un numero di metodi relativamente basso con delle funzionalità altamente correlate e non fa troppo lavoro. Essa collabora con altri oggetti per condividere lo sforzo se il compito è grande.

14.6 Controller

Come si integrano si integrano Interfaccia Utente e Servizi?

Servizi Sono operazioni eventualmente composte come per esempio operazioni di sistema: le operazioni di sistema sono gli eventi principali di input al sistema.

Interfaccia Utente L'interfaccia utente viene utilizzat per eseguire una sequenza (un workflow) di operazioni (producendo eventi).

Problema Come possiamo integrare questi due elementi?

Soluzione Utilizzando un controller, un oggetto oltre lo strato di UI che è responsabile di ricevere e gestire gli eventi come i messaggi di operazioni di sistema.
Ci sono due alternative in base alla responsabilità assegnata:

- Scegliere/Creare un oggetto che rappresenta il sistema complessivo (FacadeController)
- Un oggetto controller può rappresentare uno scenario di un caso d'uso (UseCaseController)

Il controller implementa le operazioni a livello utente: conosce quali servizi devono essere eseguiti in corrispondenza di una certa operazione utente. Nello specifico il controller ha le seguenti responsabilità

- Validare l'input
- Tradurre l'oerazione richiesta dall'utente nell'invocazione dei servizi
- Scegliere la schermata successiva in base al risultato prodotto dall'operazione
- Preparare i dati per l'interfaccia

Vantaggi di un controller

La logica applicativa non è gestita nello strato di interfaccia e si può inoltre riusare la logica essendo svincolata dalla UI. Si possono usare interfacce diverse e se si usano controller basati sui casiu d'uso è possibile:

- Verificare che le operazioni si susseguano in una sequenza legale
- Ragionare sullo stato corrente dell'attività

Tipici Errori di implementazione

- Controller "gonfi"
- Esiste una unica classe controller che riceve tutti i numerosi eventi di sistema

- Il controller svolge parte del lavoro prima di delegarlo
- Il controller ha numerosi attributi e conserva informazioni sul sistema e sul dominio

14.7 Pure Fabrication

Problema É sempre corretto creare rappresentazioni dei concetti del mondo reale nel dominio? Non poichè potrebbero violare altri principi (low coupling, high cohesion, ecc.)

Soluzione Creaiamo una classe astratta ma che sostenga i principi GRASP principali, come low coupling e high cohesion. Così facendo andiamo a inventare una classe immaginaria.

Vantaggi

- Sostiene high cohesion
- Potenzialità di riuso aumentata

Svantaggi

- Un utilizzo eccessivo può portare ad una decomposizione del codice: ovvero si potrebbe avere un eccesso di oggetti comportamentali che non possiedono informazioni richieste per soddisfarli

14.8 Polymporphism

Problema Come gestire l'introduzione di un nuovo tipo? Come inserire nuovi componenti che non richiedono la modifica di gran parte del codice?

Soluzione Introduciamo delle operazioni di polimorfismo.

Vantaggi

- Le estensioni sono facili da aggiungere
- Si possono introdurre nuove implementazioni senza influire sui client

Svantaggi

- Progettare interfacce e polimorfismi che ancora non si stanno gestendo, non conoscendo quanta probabilità si ha di gestire quella casistica

14.9 Indirection

Problema Come posso assegnare una responsabilità per evitare l'accoppiamento diretto tra più elementi? E di conseguenza, come posso separare gli oggetti in modo da mantenere un alto riuso e un accoppiamento bassso?

Soluzione Si assegna la responsabilità ad un oggetto intermediario che media tra gli elementi, in modo che non ci sia un accoppiamento diretto tra di essi. L'intermediario crea una indirizzione (indirection) tra le componenti.

Vantaggi Favorisce il Low Coupling.

14.10 Protected Variations

Problema Come posso progettare oggetti, sottosistemi e sistemi in modo tale che le modifiche (o l'eventuale instabilità) di questi elementi non si riversino su altri elementi?

Soluzione Si identificano i punti in cui sono previste delle variazioni, poi si assegnano le responsabilità per creare una "interfaccia" stabile attorno a questi punti.
É conosciuto anche con Information Hiding.

Vantaggi

- Le estensioni richieste per nuove variazioni sono facili da aggiungere
- è possibile introdurre nuove implementazioni senza influire sui client
- favorisce Low Coupling
- è possibile ridurre l'impatto o il costo dei cambiamenti

Svantaggi

- è inutile e costoso utilizzare PV per delle variazioni che non si verificheranno mai
- in alcuni momenti conviene di più riprogettare che realizzare la protezione da tutte le possibili variazioni

14.11 Principi OO

Tenere presente i principi precedentemente spiegati tenendo conto però che è non bisogna mai prendere decisioni ragionando singolarmente su un principio e mai attenersi strettamente alle regole senza usare il buon senso, la logica e l'esperienza. Chiedersi sempre quali sono le soluzioni possibilie e le conseguenza di ogni soluzione.

14.12 Granularità delle responsabilità

Caso semplice - Granularità fine - Creazione di un metodo.
Caso difficile - Granularità grossa - Refactoring + creazione metodi e attributi.

Chapter 15

Design Patterns

Un Design pattern è un modo di riutilizzare la conoscenza astratta di un problema e la sua soluzione. Un pattern è una desrizione del problema e l'essenza della sua soluzione. Dovrebbe essere sufficientemente astratto per essere riutilizzato in diversi contesti. Le descrizioni dei pattern di solito fanno uso di caratteristiche orientate all'oggetto, come l'eredità e il polimorfismo.
I design pattern sono emergi da alcuni anni come una delle tecniche più effiaci per trasferire la conoscenza e l'esperienza dei progettisti, rendendola disponibile a tutti in modo comprensibile e concreto.

I DP aiutano a:

- Costruire del software che sia risuabile
- Evitare scelte che compromettano il riutilizzo del software
- Migliorare la documentazione e la manutenzione di sistemi esistenti

In pratica un design pattern è una regola che esprime una relazione tra un contesto, un problema ed una soluzione.
Un design pattern nomina, astrae e identifica gli aspetti chiave di una struttura di design comune, che la rendono utile per creare un design OO riusabile.
Identificano:

- Le classi e le istanze che vi partecipano
- I loro ruoli
- Come collaborano
- La distribuzione delle responsabilità

15.1 Come sono fatti i Design Patterns?

Un pattern è formato da quattro elementi essenziali:

- Il **nome** del pattern, per riassumere la sua funzionalità in una o due parole
- Il **problema** nel quale il pattern è applicabile. Spiega il problema e il contesto, a volte descrive dei problemi specifici del design, mentre a volte può descrivere strutture di classi e oggetti.
- La **soluzione** descrive in modo astratto come il pattern risolve il problema Descrive gli elementi che compongono il design, le loro responsabilità e le collaborazioni
- Le **conseguenze** portate dall'applicazione del pattern, i costi-benefici

15.2 Descrizione DP

Il formato che si sceglie per la descrizione di un pattern non è rilevante, purchè la descrizione sia completa, coerente e accurata.
Di seguito un esempio:

- Nome e classificaizone pattern
- Sinonimi
- Scopo
- Motivazione: scenario che illustra un design problem
- Applicabilità
- Struttura
- Partecipanti: classi e oggi inclusi nel pattern
- Collaborazioni
- Conseguenze
- Implementazione
- Codice di esempio
- Usi noti

- Pattern correlati

L'obiettivo è sempre quello di aiutare gli sviluppatori a risolvere problemi già studiati, mostrando che la soluzione descritta è:

- Utile
- Utilizzabile
- Usata

15.3 Classificazione DP

Scope

Un primo criterio riguarda il raggion di azione (scope)

Classi Pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sul concetto di ereditierietà e sono quindi statiche.

Oggetti Pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più dinamiche.

Purpose

Un secondo criterio riguarda lo scopo (purpose):

- Creazionali: i pattern di questo tipo sono relativi alle operazioni di creazione oggetti
- Strutturali: sono utilizzati per definire la struttura del sistema in termini di composizione di classi ed oggetti. Si basano sui concetti OO di ereditarietà e polimorfismo
- Comportamentali: permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti e definendo le modalità di interazione.

15.3.1 Creational

Questa categoria raccoglie i pattern che forniscono un'astrazione del processo di istanziazione degli oggetti. Questi pattern permettono di rendere un sistema indipendente da come gli oggetti sono creati, rappresentati e composti al suo interno.

15.3.2 Structural

I pattern di questa categoria sono dedicati alla composizione di classi e oggetti per formare strutture complesse. É possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti.
Particolarmente utili per fare in modo che librerie di classi svilkuppate indipendentemente possano operare insieme.

15.3.3 Behavioral

Questi pattern sono dedicati all'assegnamento di responsabilità tra gli oggetti e alla creazione di algoritmi. Una caratteristica comune di questi pattern è data dal supporto fornito per seguire le comunicazioni che avvengono tra gli oggetti.
L'utilizzo di questi pattern permette di dedicarsi principalmente alle connessioni tra oggetti, tralasciando la gestione dei flussi di controllo.

15.4 I Framework

Un **Framework** non è una semplice libreria. Un Framework rappresenta il design riusabile di un sistema (o di una sua parte), definito da un insieme di classi astratte. Un Framework è lo scheletro di un'applicazione che viene personalizzato da uno sviluppatore. Il programmatore di applicazioni implementa le interfacce e classi astratte ed ottiene automaticamente la gestione delle funzionalità richieste.
I Framework permettono quindi di definire la struttura e lo scopo di un sistema, si tratta di un buon esempio di progettazione orientata agli oggetti. Permettono di raggiungere due obiettivi:

- Riuso del design
- Riuso del codice

Classe Astratta Una classe astratta è una classe che possiede almeno un metodo non implementato, definito "astratto". Si tratta di un template per le sottoclassi da cui deriveranno le specifiche applicazioni.
Un Framework è rappresentato da un'insieme di classi astratte e dalle loro interrelazioni. I Design Pattern sono spesso i mattoni per la costruzione di un Framework.

15.5 DP: Adapter (Structural)

- Nome: Adpater
- Sinonimo: Wrapper
- Scopo: Converte l'interfaccia di una classe in un'altra interfaccia richiesta dal client (quindi permette la cooperazione di classi che altrimenti avrebbero interfacce incompatibili)
- Motivazione: in alcune circostanze non si può utilizzare una classe già esistente solo perchè quest'ultima non comunica più con una interfaccia specializzata richiesta da un'applicazione.

Si tratta letteralmente di un adattatore per rendere due compatibili due oggetti che altrimenti non sarebbero compatibili (come una shuko e una presa italiana).

Partecipanti

- Target: definisce l'interfaccia specializzata del client
- Client: collabora con oggetti conformi all'interfaccia target
- Adaptee: definisce una interfaccia che deve essere resa conforme
- Adaptor: adatta l'interfaccia di Adaptee all'interfaccia Target

Applicabilità Adapter può essere utilizzato:

- Se si vuole utilizzare una classe esistente la cui interfaccia si incompatibile
- Se si vuole creare una classe riusabile che dovrà collaborare con classi non prevedibili al momento della sua creazione
- Se si vuole riusare un insieme di sottoclassi esistenti, ma non è facile adattare l'interfaccia creando una sottoclasse per ciascuna: meglio creare un adattatatore per la classe genitore

Conseguenze sulla classe Adapter ridefinisce (overrides) parte del comportamenti di Adaptee.
Una clase particolare adatta Adaptee a Target concretizzando un Adapter, quindi non è possibile adattare una classe e tutte le sue sottoclassi.

Conseguenze sull'oggetto Gli Adapter possono funzionare in molti modi, dalla conversione di interfaccia alla completa ridefinizione e integrazione dei metodi, la quantità di lavoro svolta da un Adapter dipenderà da quanto simili sono il Target voluto e l'Adaptee disponibile.

Pattern correlati

- Bridge
- Decorator
- Proxy

Correlazione con GRASP É una specie di Indirection e Pure Fabrication, che utilizza Polymorphism.

15.6 DP: Strategy (Behavioral)

- Nome: Strategy
- Sinonimo: Policy
- Scopo: Definisce una famiglia di algoritmi, incapsula ognino in una classe e li rende intercambiabili. Permette di variare gli algoritmi indipendentemente dai client che ne fanno uso.
- Motivazione: In alcune circostante sono necessari algoritmi diversi in situazioni diverse per risolvere lo stesso problema. Per dare più flessibilità all'implementazione, tali algoritmi non sono specificati in una sola classe, ma in classi diverse. In questo modo, algoritmi diversi possono essere utilizzati in momenti diversi. Inoltre, nuovi algoritmi sono facilemente inseribili.

Partecipanti

- Stretegy (ArrayDisplayFormat): dichiara un'interfaccia comune per tutti gli algoritmi supportati. Context utilizza tale interfaccia per invocare gli algoritmi definiti in ogni ConcreteStrategy.
- ConcreteStrategy (StandardFormat, MathFormat): implementa l'algoritmo che usa l'interfaccia stretegy

- Context (MyArray): è configurata con un oggetto ConcreteStrategy e mantiene un riferimento verso questo; può specificare una interfaccia che permette alle Strategy di accedere ai propri dati.

Applicabilità

- Più classi correlate variano soltanto per il loro comportamente. Il pattern permette la configurazione di una classe con più comportamenti.
- Per modellare la differenza fra diverse varianti dello stesso algoritmo. Esempio: calcolo dell'ora per varie zone geografiche
- Per nascondere all'utente strutture dati complesse e specifiche dell'algoritmo
- Per evitare l'utilizzo di tanti if in un algoritmo. Si crea una classe divers per ogni possibile ramo.

Conseguenze

Vantaggi

- La definzione di famiglie di algoritmi
- Un'alternativa alla gerarchia di classi
- Eliminazione di tanti if
- Scelta fra varie implementazioni

Svantaggi

- Gli utenti diventano consapevoli dell'esistenza delle strategie
- Overhead di comunicazione fra Strategy e Context
- Incremento del numero di oggetti

Pattern correlati Flyweight.

15.7 Composite (Structural)

Problema Come comporre oggetti in strutture ad albero? Utilizzando **Composite**

- Definisce gerarchie di classi
- Il Clien tratta foglie e nodi allo stesso modo
- L'aggiunta di nuovi tipi di nodi e foglie è molto semplice

Descrizione Composite

- Name - Composite
- Intent - Comporre oggetti in strutture ad albero per rappresentare gerachie tutto-parti trattare uniformemente oggetti individuali e composti, catturare l'essenza ricorsiva della composizione
- Motivation - In questo modo si semplifica il client dato che si possono trattare strutture singole e oggetti in modo uniforme

Si rende anche più semplice l'aggiunta di nuove tipologie di componenti (nuove sottoclassi Leaf o Composite potranno essere utilizzate automaticamente nelle strutture esistenti e operre con il codice dei client). Può rendere il progetto troppo generico per la facile aggiunta di nuove componenti.

15.8 DP: Singleton (Creational)

Problema Assicurare che una classe abbia una sola istanza con accesso globale.
Per esempio voglio utilizzare un'unica connessione verso il database ed impedire che un oggetto possa autonomamente creane di nuove.

Soluzione Definire un metodo statico della classe che restituisce il singleton.

Conseguenze

Pro

- Accesso controllato ad un'unica istanza
- Incapsula la creazione dell'oggetto

- In alcune implementazioni possiamo gestire un pool di istanze invece che una singola istanza

Contro

- In alcune implementazioni ogni volta che l'oggetto viene richiesto viene anche controllata la sua esistenza
- Singleton non tratta distruzione dell'oggetto

15.9 DP: Facade (Structural)

Problema Un client accede direttamente a numerose classi che fanno parte di un altro modulo.
Il pattern Facade nasconde ai client il sottosistema. Questo risulta molto utile per migliorare l'integrazione fra componenti. Questo pattern ha lo scopo di fornire un'intefaccia unificata per un dato insieme di interfacce presenti in un sottosistema.

Applicabilità Per ottenere interfacce semplici per sottosistemi complessi o per i casi in cui vi sono troppe dipendenze fra i client e le classi.
Per ottenere una struttura a livelli in cui il Facade funge da entrata e uscita di ogni livello.

Partecipanti

- Facade: Conosce la struttura del sottosistema e delega agli oggetti interni più appropriati le richieste provenienti dall'esterno.
- Classi di Subsystem: Forniscono le funzionalità interne adatte a rispondere alle richieste provenienti da Facade. Esse non hanno conoscenza dell'esistenza di Facade e non dipendono da esso.

Conseguenze

La classe Facade nasconde il sottosistema al Client, la complessità del sottosistema viene spostata all'interno della classe Facade. Si disaccoppia il codice del Client dagli oggetti che si trovano dietro la Facade (Low coupling pattern GRASP).
Risulta inoltre più facile modifica il sottosistema.

Vantaggi

- Nasconde ai client i componenti del sottosistema, riducendone così la complessità
- Consente di modificare i componenti del sottosistema senza il coinvolgimento del client

Il Facade si interpone tra classi client e il sottosistema fungendo da intermediario fra queste 2 entità. (Indirection e Protected Variations).

Alcuni usi noti

Java Virtual Machine e varie librerie standard come AWT e Swing, forniscono un seti di wrapper Facade che incapsulano molte delle chiamate native di basso livello del SO e GUI APIs.
Il pattern Facade è spesso usato anche per fornire un'interfaccia unificata per un insieme di funzionalità che altrimenti andrebbero ad appesantire il client. Si nascondono inoltre le ie complessità delle interazioni e delle dipendenze.

15.10 DP: Observer (Behavioral)

Il pattern Observer definisce una relazione uno-a-molti tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da esso siano notificati e aggiornati automaticamente.
In pratica si definisce uno schema 1:N fra due oggetti chiamati Observer e Subject: i primi osservano un evento specifico del soggetto e non appena quest'ultimo lo genera gli osservatori vengono informati del cmabiamento di stato e aggiornano automaticamente le informazioni.

Applicabilità

Il design Observr è indicato in particolare:

- Quando un'astrazione ha due aspetti, uno dipendente dall'altro. Implementare questi aspetti in oggetti separati permette al programmatore di modificarli e di riutilizzarli indipendentemente
- Quando il cambiamento di stato di un oggetti implica il cambiamento di stati di altri oggetti e non è noto in anticipo il numero degli oggetti che dovranno essere cambiati
- Quando un oggetto deve essere in gradi di notificar eil proprio cambiamento di stato ad altri oggetti senza conoscere quali siano tali oggetti

Partecipanti

- Subject
  - Conosce i suoi osservatori che possono essere in un numero indefinito
  - Fornisce un'interfaccia per aggiungere o eliminare oggetti Observer
- Observer
  - Definisce un'interfaccia di aggiornamento che deve essere avvisata dei cambiamenti di un Subject
- ConcreteObserver
  - Fa riferimento a un oggetto ConcreteSbject
  - Il suo stato deve restare coerente con quello del soggetto
  - Implementa l'interfaccia per l'aggiornamento della classe Observer per mantenere il suo stato coerente con quello del soggetto
- ConcreteSubject
  - Contiene valori di interesse per gli oggetti ConcreteObserver
  - Invia un avviso ai suoi osservatori quando il suo stato cambia

Conseguenze

Il pattern Observer permette di variare i Subject e gli Observer indipendentemente, è possibile riutilizzare i soggetti senza il riutilizzo dei loro osservatori e viceversa.
Inoltre:

- Astrazione della relazione esistente tra Subject e Observer: il soggetto infatti sa soltano id avere una lista di osservatori, ognuno derivato dalla semplice interfaccia della classe astratta Observer.
- Supporto per comunicazioni in broadcast: a differenza di una normale richiesta, la notifica di cambiamento inviata da un soggetto non ha bisogno di specificare il proprio destinatario. La notifica è trasmessa automaticamente a tutti gli osservatori interessati.
- Update imprevisti: dal momento che gli osservatori non sono a conoscenza della presenza degli altri osservatori, una apparentemente innocua operazione sul soggetto può provocare una cascata di aggiornamenti degli osservatori

e dei loro oggetti dipendenti. Questo problema è aggravato dal fatto che la semplice operazione di notifica definita nel DP non prevede dettagli su cosa è cambiato nel soggetto.

Pattern correlati

- Mediator: definisce un oggetto che incapsula il modo in cu un insieme di oggetti intergiscono mantenendo la loro indipendenza
- Singleton: si assicura del fatto che una classe abbia una sola istanza e fornisce un punto di accesso globale ad essa

15.11 Factory Method (Creational)

- Nome - Factory Method
- Altri nomi: Virtual Constructor
- Classificazione - Creational
- Scopo - Definire un'interfaccia per la creazione di un oggetto, lasciando alle sottoclassi la decisione su quale classe istanziare. Il Factory Method permette di posticipare l'istanziazione e permetterla a runtime
- Motivazione - Si vuole che un oggetto istanzi un altro oggetto, ma non si sa a priori quale tipo di oggetto deve essere creato.

Si tratta di un pattern ampiamente usato nei Framework dove le classi astratte definiscono le relazioni tra gli elementi del dominio e sono responsabili per la creazione degli oggetti concreti.

Applicabilità

Il Factory Method è applicabile quando:

- Una classe non può prevedere il tipo di oggetti che deve creare
- Una classe vuole che siano le sue sottoclassi a specificare gli oggetti che crea
- Si vuole conoscere la classe alla quale è stata delegata la reponsabilità della creazione

La classe **Creator** affida alle sue sottoclassi la definizione del Factory Method in modo da ritornare l'istanza appropriata del ConcreteProduct

Conseguenze

Utlizzare un metodo Factory per la creazione di oggetti in una classe fornisce sempre una flessibilità maggiore rispetto alla creazione diretta dell'oggetto. Fornisce inoltre alle sottoclassi un punto di aggancio per la produzione di una versione specializzata di un oggetto.

15.12 Problemi di progettazione

Per utilizzare i pattern nel vostro progetto è necessario tenere in mente che qualsiasi problema di progettazione che si sta affrontando può avere un pattern associato che può essere applicato.

- Dire a diversi oggetti che lo stato di qualche altro oggetto è cambiato (pattern Observer)
- Mettere in ordine le interfacce con un numero di oggetti correlati che spesso devono essere sviluppati in modo incrementale (pattern Facade)

15.13 Pattern Dati

Pattern per la manipolazione di dati di DBMS, noi trattaremo il modello relazionale dato che è il più diffuso a livello commerciale. L'esigenza di definire dei pattern per questa casistica deriva dal fatto che i modelli ER e i modello OO (Object Oriented) sono strutturalmente molto diversi.

15.13.1 Active Record

Adatto nel caso in cui i record del DB corrispondono alle entità usate nella logica di business. Un Active Record include la parte della logica applicativa. In applicazioni semplici si ha una classe di dominio per una tabella del database e quindi ogni oggetto di dominio è responsabile di caricare e salvare nel database (le classi corrispondono alle struttura dei record nel db-schema isomorfico, cioè alle tabelle). Un oggetto incapsula dati e comportamento la maggior parte dei dati sono persistenti e devono essere memorizzati in un DB.
Active Record usa l'approccio più ovvio: mettere la logica di accesso ai dati nell'oggetto di dominio: in questo modo tutti conoscono come leggere e scrivere i dati a/dal DB.

Classificazione tipica dei metodi

- Load
  - Costruzione di un'istanza dell'oggetto a partire dai risultati ritornati dalle query SQL
  - Se non costruiamo interamento l'oggeto al primo caricamento i metodi devono caricare le istanze quando necessario (lazy-load)
  - Nel caso di riferimenti ad oggetti può essere necessario caricare altri oggetti e quindi navigare altre tabelle.
- Costruttuore classico - Costuire nuove istanze da inserire successivamente nel DB
- Metodi finder (statici) - Incapsulano le Query SQL e ritornano istanze degli oggetti
- Metodi di aggiornamento del DB
  - Update, aggiorna un record esistente con i valori degli attributi
  - Insert, aggiunge un record utilizzando i valori degli attributi
  - Delete, elimina il record corrispondente all'oggetto corrente
- Metodi accessors - Classici set e get per accedere ai campi
- Metodi di logica di business

Vantaggi

- Semplicità implementativa e d'uso nel caso di logica di business OO

Svantaggi

- L'acoppiamento fra logica applicativa e DB rende difficile il refactoring separato dei due progetti (progetto applicazion e del DB). Ci sono modi di realizzare il mapping su DB che disaccoppiano maggiormente la logica applicativa e meccanismi di persistenza
- Forza o quasi l'isomorfismo fra schema del DB e entità dle modello OO

15.13.2 Data Mapper

Consiste in uno strato separato di componenti dedicati a trasferire i dati fra applicazione e DB.

- Tratta indipendentemente i due schemi, tenendo opportunamente conto delle differenze
- Bussiness logic non deve conoscere l'esistenza del DB
- Nessuna interfaccia SQL nel programma
- Nessuna conoscenza dello schema del DB
- Inverte le dipendenze rispetto agli altri gateway
- Generlalmente coinvolce altri design pattern per la gesitone della sincronizzazione fra i due schemi

Quando usarlo Imprescindibile per gestire un mapping complesso fra DB e logica business.

Vantaggi

- Isola i due strati (logica di business e DB mapping)
- La logica di business non usa le API SQL
- La logica di business non dipende da alcuna conoscenza del DB
- La logica di business non deve gestire la sincronizzazione dei dati persistenti
- Il layter Data Mapper può essere sostituito intergralmente senza bisogno di modificare alcun dettaglio dello strato applicativo (es. se cambio il DB, il modello dell'applicazione o per test)

Svantaggi Difficoltà implementiativa.

Chapter 16

Dal progetto al codice

Gli elaborati creati fin'ora saranno utilizzati come input per il processo di generazione del codice. Anche se la creazione di codice non fa parte dell'OOA/D (Object Oriented Analysis/Design), è comunque l'obiettivo finale della progettazione di un software. Durante la programmazione ci si devono aspettare e si devono pianificare numerosi cambiamenti e deviazioni rispetto al progetto e questo ha diverse implicazioni (per esempio aggiornare documentazione).

16.1 Creazione di Classi e Interfacce

I diagrammi di classe servono proprio a questo, dato che contengono

- Nomi delle classi, interfacce e superclassi
- attributi
- firme delle operazioni
- associazioni tra classi

Queste informazioni sono sufficienti per creare un'implementazione di base delle classi, alcuni strumenti effettuano automaticamente queste operazioni (come RSA).

16.1.1 Gestione delle Eccezioni

Il comportamento del sistema quando sono generate delle eccezioni può essere descritto con i modelli dinamici, se la gestione delle eccezioni non è opportunamente progettata il loro effetto diventa preso un fenomeno globale e incontrollato.

Alcune linee guida

- Definire un numero finito di eccezioni che possono essere rilasciate da un componente e che siano interpretabili dal client (re-mapping delle eccezioni)
- Nascondere infomrazioni inutili ai client e diminuire il set di eccezioni (es. ThreadInterruptedException è scarsamente utile al client)
- Attenzione a non nascondere informazioni utili
- Le eccezioni sono appropriate quando si hanno fallimenti di risorse (disco, memoria, accesso rete)
- Il valore dell'eccezioni può essere utilizzare per fornire info aggiuntive
- In UML le eccezioni possono essere indicate nelle stringhe di proprietà dei messaggi e delle dichiarazioni delle operazioni

16.1.2 Ordine di implementazione delle classi

Esistono 3 stili principali, di cui l'ultimo è un mix dei primi due:

1. Guidato dagli scenari descritti nei modelli dinamici (si implementano più parti di diverse classi)
2. Basato sulle classi - Ordine definito in base alle dipendenze
3. Misto - Si implementano prima le classi che rappresentano le entità principali del sistema e si implementano iterativamente le operazioni di sistema

16.2 Progettare la visibilità

La visibilità viene definita come la capacità di un oggetto di "vedere" o di avere un riferimento ad un altro oggetto. Affinchè un oggetto mittente invii un messaggio a un oggetto destinatario, il destinatario deve essere visibile al mittente - il mittente deve avere qualche tipo di riferimento (o puntatore) all'oggetto destinatario.
Risulta fondamentale garantire le visibilità necessarie per consentire le interazioni fra gli oggetti.

16.2.1 Visibilità tra oggetti

Come possiamo determinare se una risorsa (ad esempio un'istanza) rientra nella portata di un'altra?
La visibilità può essere ottenuta dall'oggetto A all'oggetto B in quattro modi comuni:

- Visibilità per attributo - B è un attributo di A
- Visibilità per parametro - B è un parametro di un metodo di A
- Visibilità locale - B è un oggetto locale (non parametro) di A
- Visibilità globale - B è in qualche modo visibile globalmente

16.2.2 Visibilità per attributo

La visibilità per attributo da A a B esiste quando B è un attributo di A.
Si tratta di una visibilità relativamente permanente perchè persiste finchè A e B esistono, è molto diffusa nei sistemi OO.

16.2.3 Visibilità per parametro

La visibilità per parametro da A e B esiste quando B viene passato come parametro ad un metodo di A. Si tratta di una visibilità relativamente temporanea perchè persiste solo nell'ambito del metodo.

16.2.4 Visibilità locale

La visibilità dichiarata localmente da A a B esiste quando B è dichiarato come oggetto locale nell'interno di un metodo di A. Si tratta di una visibilità relativamente temporanea perchè persiste solo nell'ambito del metodo.

- Creare una nuova istanza locale e assegnarla ad una variabile locale
- Assegnare l'oggetto di ritorno da un'invocazione di metodo ad una variabile locale

Trasformare la visibilità è possible trasformare la visibilità locale in visibilità per parametro oppure la visibilità per parametro in visibilità per attributo.

16.2.5 Visibilità globale

La visibilità globaile da A a B esiste quando B è globale per A. Si tratta di una visibilità relativamente permanente perchè persiste finchè A e B esistono. la forma meno comune, puiò favorire alto accoppiamento. Un modo per ottenere questo risultato è quello di assegnare un'istanza a una variabile globale (possibile in C++, ma non in Java).
Il metodo migliore per ottenere la visibilità globale è quello di utilizzare il pattern Singleton.

16.3 Test Driven Development

Il TDD (Test Driven Development) è una delle pratiche introdotte da Extreme Programming (XP) ed è diventata una pratica molto diffusa anche in metodologie non agili. Applicabile anche a UP e Scrum, consente lo sviluppo iterativo ed evolutivo. Prevede Refactoring continuo del codice.

16.3.1 Benefici del TDD

Alla fine dell'attività di sviluppo è stata ottenuta sia l'implementazione della classe che i suoi casi di test d'unità/integrazione e quindi ci siamo tolti un bel po' di lavoro, dato che altrimenti andrebbe fatto tutto dopo lo sviluppo delle classe.
Interfaccia e comportamenti saranno dettagliati dato che si dovrà ragionare sui metodi necessari ai test e sui possibili comportamenti. Il testing è oltretutto ripetibile automaticamente. Ci sarà anche maggior confidenza nel cambiare il sistema e il codice sarà migliore.

Tipologie di test

- Test di unità
- Test di integrazione
- Test di sistema
- Test di accettazione
- Test di regressione

16.3.2 Test di unità

Il test di unità si traduce in:

- Scrivere il codice di un test prima della classe da testare
- Implmenetare parte della classe in modo che questa superi il test
- Scrivere un nuovo caso di test
- Reiterare

Dopo aver scritto i casi di test procediamo con l'implementazione. Una volta che i casi di test sono stati superati dall'implementazione attuale procediamo progettando nuovi casi di test oppure passando ad una nuova classe/metodo.

Schema dei metodi di test di unità

- Preparazione - Crea l'oggetto da verificare (detto anche fixture)
- Esecuzione - Far fare qualcosa alla fixture al fine di eseguire alcune operazioni che si desidera testare
- Verifica - Verifica che i risultati ottenuti corrispondono a quelli previsti
- Rilascio - Opzionalmente rilascia o ripulisce gli oggetti e le risorse utilizzate nel test

Esempi di Framework di test xUnit è un Framework per test di unità per molti linguaggi, per esempio NUnit è per .NET, mentre JUnit è per Java.

16.3.3 Ciclo del TDD

- Scrivere un caso di test che fallisce
  - Non scrivere codice di produzione fino a quando non hai scritto un test unitario
  - Non scrivere più di un test di quanto è sufficiente per far fallire la compilazione o esecuzione del test
- Scrivi il codice più semplice possibile per far passare il test
- Riscrivi o ristruttura il codice migliorandolo, oppure passa a scrivere il prossimo test unitario

Il Refactoring può essere applicato dopo aver effettuato ciascun test, oppure dopo diversi test.

Chapter 17

Refactoring

17.1 Varie definizioni

Il refactoring è il processo di modifica di un sistema software che non altera il suo comportamento esterno e migliora la sua struttura interna (più semplice da capire e modificare).
Un'altra definizione è che il refactoring è un cambiamneto al sistema che lascia inalterato il suo comportamento, ma che migliora alcune qualità funzionali come semplicità, flessibilità, chiarezza, performance, riusabilità, manutenibilità.

17.1.1 Quando fare refactoring

- Quando aggiugi una funzione
- Quando hai bisogno di fixare un bug
- Mentre fai revisione del codice
- Quando il codice è difficile da capire

17.1.2 Quando NON fare refactoring

- Quando il codice non funzione
- Quando sei troppo vicino a una deadline
- QUando il design è così pessimo che è necessario riscrivere il codice
- Prima di effettuare il refactoring, assicurati di avere un buon set di casi di test

17.1.3 Come effettuare refactoring

- Essere sicuro che i testi passini
- Trova del codice che "smells" (puzza)
- Determina come semplificare il codice
- Effettuare le semplificazioni
- Esegui Test per verificare che non siano stati introdotti bug
- Ripeti il processo finchè la "smell" (puzza) non se nè andata

17.1.4 Perchè fare refactoring

- Migliorare il design del test (che decade nel tempo)
- Ridurre la quantità di codice richiesta per completare un task (rimuovendo codice inutile)
- Migliorare la leggibilità del codice
- Ti aiuta a trovare Bug (codice più complesso è più soggetto ad avere bug)

17.1.5 Problemi con il refactoring

- Database (richiede di modificare i dati per adattarsi al nuovo design, la migrazione più essere molto lungha e costosa)
- Personal Ownership of Code
- Cambiare le interfacce (interfacce pubbliche non possono essere cambiate)

17.1.6 Esempi di refactoring

- Extract Method - Trasforma un metodo lungo in uno più breve, estraendone una parte in un metodo di supporto
- Extract Class - Crea una nuova classe e vi muove alcuni campi e metodi di un'altra classe
- Move Method - Crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più

17.2 Code Smells

La nozione di code smells è molto importante per il refactoring.
Si tratta sostanzialmente di caratteristiche che sono forti indicatori di una struttura cattiva che dovrebbe essere ristrutturata.

> If it stinks, change it. (M.Fowler, Refactoring)

I code smelss sono regole empiriche che indicano che il codice potrebbe essere migliorato. Risulta sempre importante valutare il caso specifico, perchè non sempre un code smell porta necessariamente a un refactoring. Comunque un codice che presenta code smells è probabile che significhi che dovresti cambiare qualcosa.

17.3 Bad Smells in Code

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field

- Message Chain
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

Uno smells è un sintonmo, il refactoring è la cura. Usa il relativo refactoring per il relativo smells. Anche se stessi sintomi possono necessitare di cure diverse, così come ogni smelss può suggerire più di un refactoring.

17.3.1 Duplicated Code

Due frammenti di codice che sembrano uguali.
I duplicati si verificano solitamente quando più programmatori stanno lavorando su differenti parti di uno stesso programma simultaneamente e dato che stanno lavorando su differenti task, potrebbero non accorgersi che stanno creando codice che è già stato creato da altri colleghi.
Si tratta di duplicazione anche quando specifiche parti di codice sembrano diverse, ma fanno la stessa cosa. Questo tipi di codice duplicato è difficile da trovare e sistemare.

17.3.2 Long Method

Un metodo contiene troppe righe di codice. Generalmente qualsiasi metodo lungo più di 10 righe dovrebbe iniziare a far scattare l'allarme.
Questo accade perchè spesso per pigrizia è più difficile scrivere un nuovo metodo, risulta più facile aggiungere qualche linea di codice in più a un metodo già esistente, così facendo mano a mano si aggiunge codice e si arrivare ad avere un metodo enorme contenente molte righe.

Risoluzione Come regola generale, se senti che qualcosa all'interno di un metodo necessita di un commento per essere spiegato probabilmente dovresti creare un metodo separato.
Come Refactoring è buona norma usare Extract Method per ridurre la lunghezza del corpo del metodo.

17.3.3 Feature Envy

Un metodo in una classe accede dati di un altro oggetti più spesso che ai dati della sua stessa classe. Questo può accadere dopo aver spostato attributi in una classe dati. Se è questo il caso potresti voler spostare anche le operazioni da effettuare sui dati in questa classe.

Risoluzione Come regola base, se le cose cambiano insieme, mantienile insieme nello stesso posto. Solitamente dati e funzioni che usano quei dati sono modificati insieme.
Utilizza qundi Move Method per spostare i metodi nella classe corretta, se una sola parte del metodo acede ai dati di un'altro oggietto usa Extract Method per spostare le parti in questione.

Vantaggi

- Riduzione code duplication
- Miglior organizzazione del codice

17.3.4 Large Class

Una classe che contiene molti metodi, molte variabili istanziate è una clase large class, una classe che prova a fare troppo.
Questo indica un problema di astrazione, probabilmente nel codice c'è più di un problema o qualche metodo appartiene ad altre classi.
Le classi solitamente vengono create piccole, ma nel tempo vengono gonfiate più il programma cresce. Questo accade per la stessa motivazione dei Long Methods, i programmatori solitamente trovano mentalmente più facile aggiungere codice ad una classe già esistente piuttosto che crearne una nuova.

Risoluzione Dividi la classe in più classi, usando Extract Class per separare il comportamento della classe in componenti dedicate, Move Method per spostare i metodi nelle nuove classi e Extract Subclass per creare sottoclassi per un subset di features se sono usate solo da alcune istanze.

Vantaggi

- Effettuare il refactoring di queste classi rende la vita degli sviluppatori più semplice, dato che non devono ricordarsi un gran numero di attributi per ogni classe

- In molti casi dividere le classi in classi più piccole evita duplicazione di codice e rende più funzionale lo sviluppo

17.3.5 Switch Statements

Classica situazione dove si ha uno switch complesso, più switch identifici in posti diversi o una cascata di if statements.
Fortunatamente nella programmazione object-oriented, questo tipo di codice è raro, dato che gli switch e case sono poco usati, il problema del loro utilizzo è che quando una condizione viene aggiunta è necessario andare a ricercare tutti gli switch e case per aggiungere la nuova condizione.
Come regola generale se vedi switch dovresti pensare al polimoriismo.

Risoluzione Per isolare lo switch e metterlo nella classe giusta potresti aver bisogno di Extract Method e poi Move Method. Dopo aver specificato la struttura di ereditarietà utilizza Replace Condtional with Polymporphism per sostituire lo switch con polimorfismo.

17.3.6 Data Class

Una classe che contiene solo variabili di clase, getter/setter e nient'altro. Sta semplicemente operando come data holder.
La vera potenza degli oggetti è che possono contenere sia dati che operazioni e operare sui dati contenuti in essa.

Risoluzione Esaminare i metodi che utilizzano dati e usare Move Method per spostarli nella data class.

17.3.7 Long Parameter List

Un metodo che richiede troppi parametri.

Risoluzione Rimpiazza i parametri con metodi, introduci Oggetti parametro e utilizzao Preserve Whole Object, cioè incapsula tutti i valori che passeresti come parametro in un unico oggetto.

```
//Prima
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
//Dopo
boolean withinPlan = plan.withinRange(daysTempRange);
```

17.3.8 Shotgun Surgery

Si parla di Shotgun Surgery quando la modifica di una classe è causa di moltri altri cambiamenti ù in altre classi, il cambiamento di un'operazione implica il tanti piccoli cambiamenti in diverse operazioni e classi. Viene definita in questo modo perchè è come se si effettuasse un'operazione chirurgica sparando con un fucile a pompa, quindi colpendo molte parti del corpo, richiama un approccio goffo e soggetto a errori, piuttosto che preciso.

Cause Questo accade quando si programma copiando e incollando codice, ottenendo quindi codice duplicato e non utilizzando i corretti Design Pattern.

Problemi Aumenta il tempo richiesto per effettuare una modifica, dato che è necessario effettuare più parti del codice. Aumenta la probabilità di introdurre bug dato che è necessario ricordarsi di modificare più punti del codice. Oltretutto si produce più codice di quanto richiesto.

Risoluzione Move Method e Move Field. Per ridurre ridondanze e sviluppare funzioni e procedure apposite.

Esempio Un metodo che implementa una funzionalità utilizzata da un gran numero di classi e metodi. Cambiare questo metodo implicherebbe cambiamenti a cascata su tutti i metodi in un gran numero di classi.

17.3.9 Refused Bequest

Una classe che non usa ciò che ha ereditato dalla sua superclasse.
Se una sottoclasse utilizza solo alcuni dei metodi e proprietà ereditate dai suoi parenti l'ereditarietà è fuori luogo. I metodi non richiesti potrebbero essere non utilizzati o essere ridefiniti e restituire eccezioni.
Questo accade perchè si è motivati a creare ereditarietà solo per riutilizzare codice, ma la superclasse e la sottoclasse sono completamente diverse.

Risoluzione Replace Inheritance with Delegation per eliminare l'ereditarietà, in questo caso inutile e forzata.

17.3.10 Comments (called sweet smeel)

I commenti nel codice dovrebbero essere pochi e brevi, perchè il codice dovrebbe spiegarsi da solo, se necessita di lunghi commenti significa che non è chiaro. Dovresti effettuare refactoring per rendere il codice più chiaro e ridurre così i commenti.
Quando senti il bisogno di scrivere un commento prima di tutto ristruttura il codice così che qualsiasi commento sia superfluo.

Refactoring

- Extract Method
- Rename Method: rinomina il metodo in modo tale da rendere più chiaro il suo scopo

17.3.11 Lazy Class

Capire e mantenere una classe è sempre un costo in termini di tempo e denaro,
quindi se una clase non fa abbastanza per guadagnarsi la tua attenzione dovrebbe essere cancellata.

Cause del problema Probabilmente una classe è stata progettatata per essere completamente funzionale, ma dopo alcuni refactoring è diventata molto piccola oppure è stata creata per supportare sviluppi futuri che non sono mai stati effettivamente fatti.

Risoluzione Componenti che non sono richiesti dovrebbero essere cancellati, le poche responsabilità che aveva quella classe vengono spostate in un'altra classe.

Vantaggi

- Riduzione delle dimensioni del codice
- Mantenimento più semplice

17.3.12 Middle Man

Se una classe effettua una sola operazione, delegando il lavoro ad altre classi, perchè esiste?

Cause del problema Questi smells possono essere il risultato di una eliminazione troppo aggressiva dei messaggi a catena. In altri casi possono essere il risultato del refactoring di una classe che era troppo grande e quindi dello spostamento dei metodi in altre classi. La classe di partenza resta diventa quindi un guscio vuoto che non fa altro che delegare il lavoro ad altre classi.
In alcuni casi è da ignorare perchè potrebbe essere il risultato dell'utilizzo di Design Pattern come Proxy o Decorator.

17.3.13 Data Clumps

Qualche volta differenti parti di codice contengono gruppi identici di variabili (come parametri per connettersi a un database). Questi gruppi dovrebbero essere spostati nelle loro classi di appartenenza.

Cause A volte questi data groups sono il risultato di un'operazione di copia e incolla o di una struttura povera del programma.

Risoluzione Usa Extract Class per creare una classe che contenga il gruppo di dati ripetuto. Verificare se sono tutti essenziali.

17.3.14 Message Chains

Una catena di chiamate di metodi su oggetti restituiti da altri metodi. Questa catena può risultare molto difficile da gestire e mantenere perchè anche un cambiamento minore della struttura degli oggetti può richiedere modifiche in molti punti del codice.

Risoluzione Usare Hide Delegate, tecnica utilizzate per nascondere la complessità di un oggetto delegato. Meno il client conosce i dettagli delle relazioni tra oggetti più è facile apportare modifiche al programma. Può causare **Middle-man**.
Un'altra soluzione è usare Extract Method per creare un metodo che prenda in input l'oggetto su cui vengono chiamati i metodi della atena e che restituisca il valore finale desiderato. In questo modo si chiama una sola funzione, semplificando lettura e manutenzione del codice.
Terzo metodo è utilizzare Move Methodo per spostare il metodo nella classe che viene richiamata maggiormente dalla catena.

17.3.15 Speculative Generality

Ci sono classi, metodi o parametri che non sono utilizzati.

Cause A volte il codice è creato per supportare in anticipo funzionalità future che non verrano mai implementate, il risultato è che si ottiene un codice difficile da capire e supportare.

Risoluzione Per rimuovere classi inutilizzate usare Collapse Hierarchy per effettuare il merge delle sottoclassi nelle superclassi, oppure Inline Class per eliminare le classi inutilizzate e spostare i pochi attributi e metodi rimasti in altre classi.

17.3.16 Incomplete Library Class

Una libreria che stai utilizzando non ha le funzionalità che ti servono.

Cause Questo si verifica perchè l'autore si rifiuta di implementare il metodo perchè non conviene a livello computazionale o perchè è troppo costoso in temrinidi tempo e denaro. Può anche essere che l'autore è inconsapevole che il metodo sia necessario.

Risoluzione Se ci sono pochi metodi nella libreria possiamo implementarli con riferimenti esterni, ma questo può causare Feature Envy.
Possiamo altrimenti creare sottoclassi della libreria in locale e implementare i metodi mancanti, introducendo una maschera per nascondere la libreria (wrapper).

Vantaggi Riduce i duplicati, ma se i più progetti usano queste soluzioni risulta più difficile mantenere la compatibilità.

17.3.17 Fonti utili

Fonti consigliate dai prof (e anche da me dato che sono molto utili e ben fatte) sono i seguenti link:

- Per il refactoring - `https://sourcemaking.com/refactoring/refactorings`
- Per i code smells - `https://sourcemaking.com/refactoring/smells`