Object oriented Analisys and design

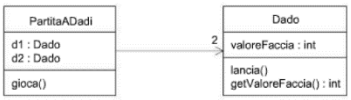
L'OOD, progettazioen basata sulla responsabilità.
Si basa sull'analisi dei requisiti ovvero casi d'uso e user story. Serve a tradurre i concetti nel dominio del problema.
La progettazioen orientata agli oggetti pone l'enfasi sulla definizione di oggetti software e del modo in cui questi collaborano per soddisfare i requisiti
L'analisi requisiti e l'OOA/D vanno svolte nel contesto di qualche processo di sviluppo: iterativo/agile/UP

UML

L'UML è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software

-Punto di vista concettuale(Modello di dominio) la notazione dei diagrammi è utilizzata per visualizzare concetti del mondo reale

-Punto di vista software (diagramma delle classi di progetto) La notazione dei diagrammi delle classi UML è utilizzata per visualizzare elementi software



Processi per lo sviluppo software

Un processo per sviluppo software/processo software descrive un approccio disciplinato alla costruzione, al rilascio e eventualmente alla manutenzione di software

Processo a cascata (o sequenziale)

E' basato sullo svolgimento sequenziale delle attività:
-definizione dei requisiti
-definizione di un piano temporale dettagliato per le attività da svolgere
-modellazione (analisi e progettazione)
-Creazione di un progetto completo del software
-Programmazione del sistema software
-Verifica rilascio e manutenzione del prodotto

E' inefficiente e le stime dei costi non sono sempre veritiere in quanto le specifiche non sono prevedibili dall'inizio

Sviluppo iterativo ed evolutivo (o incrementale)

Lo sviluppo è organizzato in piccoli progetti detti iterazioni, il risultato di ogni iterazione è un sistema eseguibile testato che farà parte del sistema reale
Ogni iterazione ha la propria parte di analisi dei requisiti e progettazione
-Comporta fin dall'inizio la programmazione e il test di un sistema software
-Comporta che lo sviluppo inizi prima che tutti i requisiti siano stati definiti in modo dettagliato
-Viene usato il feedback per chiarire e migliorare le specifiche in evoluzione del sistema
INCREMENTALE: il sistema cresce nel tempo EVOLUTIVO: Il feedback fa evolvere le specifiche
La pianificazione è guidata dal rischio del cliente, in ogni iterazione si stabilisce il piano di lavoro per la sola iterazione e non si cambia in corso d'opera,
in UP si stabilisce alla fine di ciascuna iterazione per la successiva
Le iterazioni iniziali vengono scelte per attivare i rischi maggiori, per costruire le caratteristiche principali e per stabilizzare il nucleo dell'architettura software
Vantaggi: Riduzione precoce dei rischi maggiori / Progresso visibile dall'inizio / Feedback precoce e coinvolgimento dell'utente / Gestione della complessità
ITERAZIONE TIMEBOXED = di lunghezza fissata

Unified Process UP

E' un processo di sviluppo ITERATIVO ed EVOLUTIVO, le iterazioni sono guidate dal rischio del cliente e si basa sull'Agile modeling
Agile modeling: Lo scopo della modellazione è principalmente quello di comprendere, di agevolare la comunicazione, NON di documentare
Il progetto si sviluppa in fasi sequenziali:
-IDEAZIONE: Visione approssimata del progetto, studio economico, portata, stime dei costi (Milestone: obiettivi)
NB non è una fase di requisiti ma di fattibilità, si esegue un'indagine sul progetto
-ELABORAZIONE: Visione raffinata, implementazione iterativa del nucleo dell'architettura, risoluzione dei maggiori Rischi, identificazione della maggior parte dei requisiti (Milestone: architetturale)
NB Non è una fase di requisiti o di progettazione ma una fase in cui si implementa iterativamente l'architettura del sistema e vengono mitigati i rischi maggiori
-COSTRUZIONE: Implementazione iterativa degli elementi rimanenti, più facili e a rischio minore, preparazione al rilascio (Milestone: capacità operazionale)
-TRANSIZIONE: beta test e rilascio (Milestone: rilascio prodotto)

Discipline di UP Una disciplina è un insieme di attività e relativi elaborati in una determinata area (es relativa all'analisi requisiti)

Discipline ingegneristiche
-Modellazione del business: attività che modellano il dominio del problema e il suo ambito
-Requisiti: Attività di raccolta dei requisiti del sistema
-Progettazione: Attività di analisi dei requisiti e progetto architetturale del sistema
-Implementazione: Attività di progetto dettagliato e codifica del sistema, test su componenti
-Test: Attività di controllo qualità, test di integrazione e di sistema.
-Rilascio: Attività di consegna e messa in opera

Discipline di supporto
-Gestione delle configurazioni e del cambiamento: Attività di manutenzione durante il progetto
-Gestione progetto: Attività di pianificazione e governo del progetto
-Infrastruttura: Attività che supportano il team di progetto riguardo ai processi e strumenti utilizzati

NB Le fasi sono sequenziali e quando si concludono portano ad un milestone, le discipline NON sono sequenziali e si seguono per ogni iterazione

Non si è capito lo sviluppo iterativo UP se

- Si cerca di definire tutti i requisiti del software prima di iniziare la progettazione o l'implementazione
- Si dedicano giorni o settimane a modellare con UML prima di iniziare a programmare
- Si pensa: ideazione = requisiti, elaborazione = progettazione, costruzione =implementazione (cioè, si adotta l'approccio `a cascata')
- Si pensa che l'obiettivo dell'elaborazione sia quello di definire in maniera completa e dettagliata i modelli, che verranno tradoti in codice durante la costruzione
- Si pensa che la durata adeguata per una iterazione siano 3 mesi al posto di 3 settimane
- Si cerca di pianficare il progetto nei dettagli dall'inizio _no alla _ne, e di prevedere inmaniera speculativa tutte le iterazioni e cosa deve accadere in ognuna di esse

REQUISITI EVOLUTIVI

Requisito: Un requisito è una capacità o condizione che il sistema deve soddisfare
Sorgente dei requisiti: I requisiti derivano da richieste degli utenti del sistema, per risolvere dei problemi e raggiungere obiettivi
Tipi di requisiti:
-Requisiti funzionali: I requisiti comportamentali descrivono il comportamento del sistema in termini di funzionalità fornite agli utenti
-Requisiti non funzionali: Le proprietà del sistema nel suo complesso, sicurezza scalabilità usabilità ecc

In UP si inizia a programmare già da quando sono stati specificati il 10/20% dei requisiti in quanto questi cambiano nel tempo

Modello FURPS+ dei requisiti
F Funzionale Requisiti funzionali e sicurezza
U Usabilità Facilità d'uso del sistema, documentazione e aiuto per l'utente
R Affidabilità (Reliability) La disponibilità del sistema, la capacità di tollerare guasti o di essere ripristinato a seguito di fallimenti
P Prestazioni Tempi di risposta, throughput, capacità e uso delle risorse
S Sostenibilità Facilita di modifica per riparazioni e miglioramenti, adattibilità ecc
+ Altre Vincoli di progetto (software hardware ecc) interoperabilità, ecc

Requisiti ed elaborati di UP Modello dei casi d'uso: scenari tipici dell'utilizzo del sistema (requisiti funzionali, comportamento)
Specifiche supplementari: cio che non rientra nei casi d'uso, requisiti non funzionali
Glossario: termini significativi, dizionario dei dati
Visione: Riassume i requisiti di alto livello, un documento sintetico per apprendere rapidamente le idee principali del progetto
Regole di business: Regole di dominio, i requisiti o le politiche che trascendono un unico progetto software

IDEAZIONE

L'ideazione permette di stabilire una visione comune e la portata del progetto (studio di fattibilità)

Durante l'ideazione si analizzano circa il 10% dei casi d'uso in dettaglio
si analizzano i requisiti non funzionali più critici
si realizza uno studio economico per stabilire l'ordine di grandezza del progetto e la stima dei costi
si prepara l'ambiente di sviluppo
durata: normalmente breve

Scopo dell'ideazione NON di definire tutti i requisiti, né di generare una stima o un piano di progetto affidabile, si tratta di decidere se il progetto merita un'indagine più seria durante la fase di elaborazione (NON di effettuare questa indagine)

Artefatti nell'ideazione: **Visione e studio economico**: descrive gli **obiettivi del sistema** e i vincoli di alto livello
Modello dei casi d'uso: **Descrive i requisiti funzionali**, si identificano i nomi della maggior parte degli use case, e il 10% nel dettaglio
Specifiche supplementari: descrivono altri requisiti, per lo più **i requisiti non funzionali**
Glossario: Terminologie chiave del dominio e dizionario dei dati (dizionario dei dati)
Scenario di sviluppo: Una descrizione della personalizzazione dei passi e degli elaborati di UP per questo progetto
… slide
Tempi di elaborazione degli elaborati: durante l'ideazione i documenti vengono abbozzati in maniera leggera, nell'elaborazione i documenti vengono raffinati anche grazie al feedback, durante la costruzione i requisiti principali, funzionali o meno dovrebbero essere stabilizzati

CASI D'USO

Siamo sempre nella disciplina dei requisiti

Requisiti di sistema: capacità e condizioni alle quali il sistema deve essere conforme, scritti nel "linguaggio" del committente

Disciplina dei requisiti: processo per scoprire cosa deve essere costruito ed orientare lo sviluppo verso il sistema corretto

Flusso delle attività in UP: Produrre una lista dei **requisiti potenziali** (candidati)
Capire il **contesto del sistema**
Catturare i **requisiti funzionali** (di comportamento)
Catturare i **requisiti non funzionali**

Modello di dominio: descrive i concetti importanti del sistema come oggetti di dominio e relaziona i concetti con associazioni

Modello di business: è un super insieme del modello di dominio, descrive i processi di business, è un prodotto dell'ingegneria del business, serve a migliorare i processi di business

Un Caso D'uso è un modo di usare il sistema da parte dell'utente. UP è case-driven. Sono delle descrizioni testuali degli scenari d'uso.
L'enfasi è sull'UTENTE non sul sistema.

CASO D'USO: Una collezione di scenari correlati (successo e fallimento) che descrivono un attore che usa il sistema per raggiungere un obiettivo specifico

Composto da ATTORI: qualcosa o qualcuno dotato di comportamento Attore primario (raggiunge obiettivi utente usando i servizi del sistema) Attore di Supporto (offre un servizio al sistema) Attore fuori scena (ha un interesse nel comportamento del caso d'uso)
SCENARIO: sequenza specifica di azioni ed interazioni tra il sistema e alcuni attori. Descrive una particolare storia nell'uso del sistema, un percorso attraverso l'UC

Formati dei casi d'uso **Breve**: riepilogo conciso di un solo paragrafo, relativo al solo scenario principale di successo, serve a capire rapidamente argomento e portata
Informale: Più paragrafi scritti informalmente relativi a vari scenari, stessa funzione del formato breve ma più dettagliato
Dettagliato: tutti i passi e variazioni scritte in dettaglio, include pre-condizioni e garanzie di successo.

Come scrivere un caso d'uso

Preambolo E' tutto cio che precede lo scenario principale e le estensioni
Portata: descrive i confini del sistema in progettazioni
Livello: tipicamente livello di obiettivo utente o livello di sottofunzione
Attore finale, attore primario: l'attore finale è l'attore che vuole raggiungere un obiettivo, il primario è colui che usa direttamente il sistema
Parti interessate: elenco delle parti interessate
Precondizioni/Post condizioni

Scenario principale di successo E' il percorso felice, descrive un percorso di successo, costituito da una sequenza di passi, la gestione del comportamento condizionale
È solitamente descritta da ESTENSIONI
Un'interazione tra attori (anche il sistema _e un attore):
• Un attore interagisce con il sistema, inserendo dei dati o effettuando una richiesta
• Il sistema interagisce con un attore, comunicandogli dei dati o fornendogli una risposta
• Il sistema interagisce con altri sistemi

Estensioni Le estensioni descrivono tutti gli altri scenari possibili a partire da quello principale, sia di successo che di fallimento
Le estensioni possono essere usate per gestire almeno tre tipi di situazioni:
• L'attore vuole che l'esecuzione del caso d'uso proceda in modo diverso da quanto previsto nello scenario principale
• Il caso d'uso deve procedere diversamente da quanto previsto nello scenario principale, ed e il sistema che se ne accorge, mentre esegue un'azione effettua una validazione
• Un passo dello scenario principale descrive un'azione \generica" o \astratta", mentre le estensioni relative a questo passo descrivono le possibili azioni "specifiche" o "concrete" per eseguire il passo

Stile essenziale e A SCATOLA NERA: ovvero il sistema viene descritto come dotato di responsabilità, si descrive COSA non COME

Trovare i casi d'uso Scegliere i confini del sistema, identificare gli attori primari, identificare gli obiettivi per ciascun attore, definire UC che soddisfano gli obiettivi

Verificare l'utilità dei casi d'uso Per verificare l'utilità dei casi d'uso ci sono i seguenti test:
Test del capo: "ho fatto questo, il capo sarà felice?"
Test EBP un processo di business elementare è un'attività svolta da una persona in un determinato tempo e luogo, in risposta a un evento di business che aggiunge valore di business misurabile e lascia i dati in uno stato consistente
Test della dimensione Un UC è raramente costituito da una sola azione, nel suo formato dettagliato comprende da 3 a 10 pagine di testo

Livelli del caso d'uso Livello obiettivo utente: Nell'analisi dei requisiti è utile concentrarsi sui casi d'uso EBP
Livello di sotto-funzione: Rappresenta una funzionalità nell'uso del sistema

	Identificazione UC	*Descrizione dettagliata UC*	*Realizzazione UC*
Ideazione	50%–70%	10%	5%
Elaborazione	Quasi 100%	40%–80%	Meno del 10%
Costruzione	100%	100%	100%
Transizione			

ELABORAZIONE

L'elaborazione è la serie iniziale di iterazioni durante le quali

- 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

Pianificazione dell'iterazione successiva I requisiti e le iterazioni sono organizzate in base al rischio, alla copertura e alla criticità

Iterazione 1 Nella prima iterazione i requisiti sono sottoinsiemi dei requisiti completi, si inizia la programmazione di questo sottoinsieme

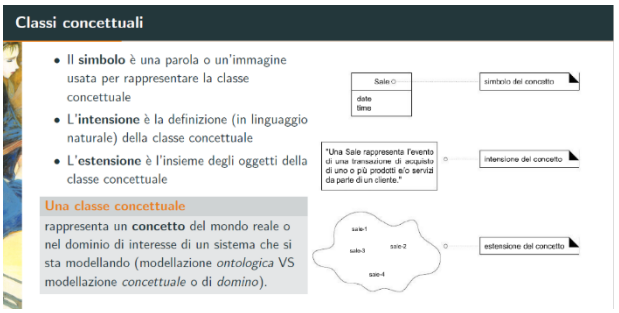
Artefatti dell'elaborazione Modello di dominio Visualizzazione dei concetti del dominio
Modello di progetto E' l'insieme dei diagrammi che descrivono la progettazione logica (classi software, diagrammi di interazione ecc)
Documento dell'architettura software
Modello dei dati
Storyboard dei casi d'uso, prototipi UI

Non si è capita l'elaborazione se - Ha una durata superiore ad "alcuni" mesi per la maggior parte dei progetti
- Ha una sola iterazione
- La maggior parte dei requisiti _e stata definita prima dell'elaborazione
- Gli elementi rischiosi e l'architettura non vengono affrontati
- Non si produce un'architettura eseguibile con progettazione e codice di qualitita-produzione
- E considerata una fase di requisiti o di progettazione che precede una fase di implementazione nella costruzione
- Il feedback e l'adattamento sono minimi, gli utenti non vengono coinvolti continuamente
- Non vengono fatti test realistici fin dall'inizio
- L'architettura viene finalizzata in modo speculativo prima della programmazione
- E considerata un passo per fare programmazione proof-of-concept

I REQUISITI E LE ITERAZIONI SONO ORGANIZZATE IN BASE ->UP E' ORIENTATO AI RISCI, ALL UTENTE E ALL ARCHITETTURA

MODELLO DI DOMINIO

Il modello di dominio è dinamico e cambia durante le varie iterazioni, è rappresentato in classi UML
E' una rappresentazione visuale delle classi concettuali, ovvero oggetti REALI e non Software
E' sviluppato nell'ambito della disciplina di Modellazione del Business
E' un insieme di classi UML che includono Oggetti del dominio, classi concettuali
Associazioni tra classi concettuali, relazioni, connessioni
Attributi di classi concettuali, valori logici degli oggetti
-non contiene operazioni
E' un dizionario visuale, NON un modello dei dati

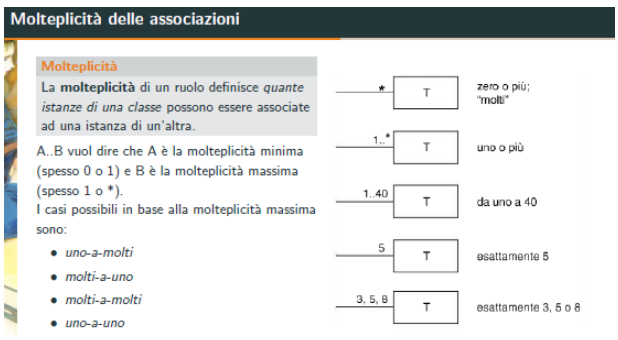


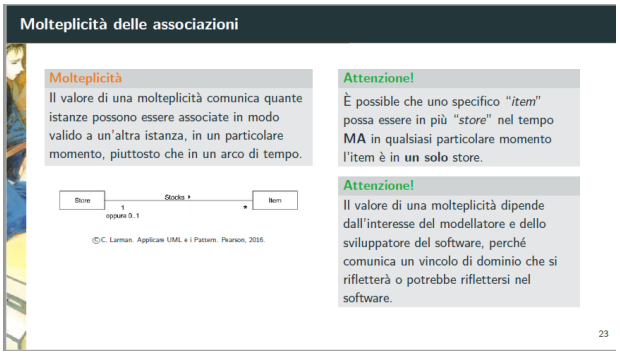
Riduzione del GAP di rappresentazione: Comprendere i concetti chiave e la terminologia
Comprendere il dominio del sistema da realizzare e il suo vocabolario
Definire un **linguaggio comune** che abiliti la comunicazione tra le parti interessate del sistema
Come fonte di ispirazione per la **progettazione** dello strato di dominio

Creare un modello di dominio

Per creare un modello di dominio si restringono i requisiti scelti per la progettazione dell'iterazione corrente
Si trovano le **classi concettuali** (ad esempio con i seguenti metodi: riuso-modifica di modelli esistenti, utilizzo di elenchi di categorie, analisi linguistiche)
Si disegnano come classi in UML, si aggiungono le associazioni, si aggiungono gli attributi

Associazioni Un'associazione rappresenta una relazione tra due o piu classi che indica una connessione significativa tra le istanze di quella classe
Ruoli nelle associazioni: Ciascuna estremità di una associazione è chiamata RUOLO, i ruoli possono avere molteplicità, nome, navigabilità
//ci possono essere anche associazioni multiple o riflessive, ovvero su se stessa





Composizione (Aggregazione)

L'aggregazione è una associazione che suggerisce una relazione intero-parte. (auto-ruote)
La Composizione è un tipo di FORTE AGGRECAZIONE: ciascuna istanza della parte appartiene **a una sola** istanza del composto alla volta
Ciascuna parte deve sempre appartenere **a un** composto
La vita delle parti è limitata a quella dei composti, le parti devono essere create dopo il composto e distrutte prima

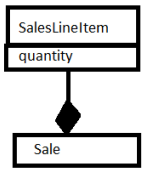
NB UNA ISTANZA DI SALESLINEITEM APPARTIENE A UNA SOLA ISTANZA DI SALE ALLA VOLTA

Attributi Un attributo di una classe rappresenta un valore degli oggetti della classe, un attributo derivato è calcolato tramite oggetti associati

Generalizzazione La generalizzazione è un'astrazione basata sull'identificazione di caratteristiche comuni tra alcuni concetti, che porta a definire una relazione tra
Tra un concetto generale (**superclasse**) e altri specifici (**sottoclasse**). Vale il principio di SOSTITUIBILITA'

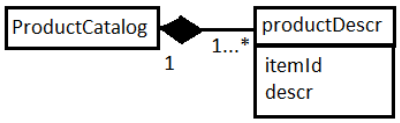
ESERCITAZIONE

1)



Un'istanza di Sale deve essere distrutta DOPO un'istanza di SaleLineItem (prima si distruggono le parti e poi il composto)
Una istanza di SalesLineItem APPARTIENE A UNA SOLA ISTANZA DI SALE ALLA VOLTA

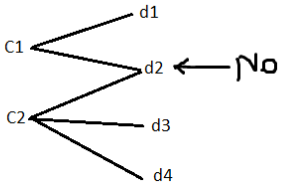
2)



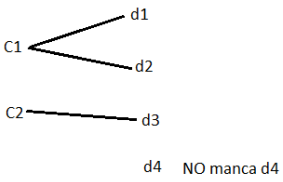
//A un ProductCatalog si possono associare 1 o + elementi di ProductDescription
//A un ProductDescription è associato esattamente un ProductCatalog

Supponiamo ProductCatalog = { c1, c2 }
ProductDescription = { d1, d2, d3, d4 }

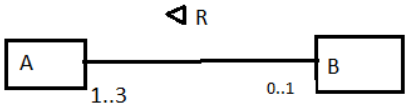
Contains (la relazione) puo essere { (c1,d1) (c1,d2) (c2,d2) (c2,d3) (c2,d4) }



Contains (la relazione) puo essere { (c1,d1) (c1,d2) (c2,d3) }

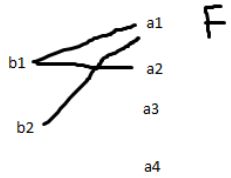


3) Considerando il seguente diagramma



Supponiamo A = { a1, a2, a3, a4 }
B = { b1, b2 }

R puo essere { (b1,a1) (b1,a2) (b2,a1) }
← vado nella direzione della freccia



FALSO ad a1 sono associati piu di un elemento B
In quanto B deve essere associata a 1..3 elementi di A, ma A deve essere associata ad al piu un elemento di B

((NB tenersi l'esercizio sulla costruzione del modello di dominio))

DIAGRAMMI DI SEQUENZA DI SISTEMA SSD (modello di progetto)

//Siamo sempre nella disciplina dei requisiti, nell'elaborazione

L' SSD è un elaborato della disciplina dei requisiti che illustra eventi di input e di output relativi ai sistemi in discussione, espressi tramite diagrammi di sequenza UML

L' SSD mostra per un particolare scenario di un caso d'uso gli eventi generati dagli attori del sistema, il loro ordine e gli eventi inter-sitema

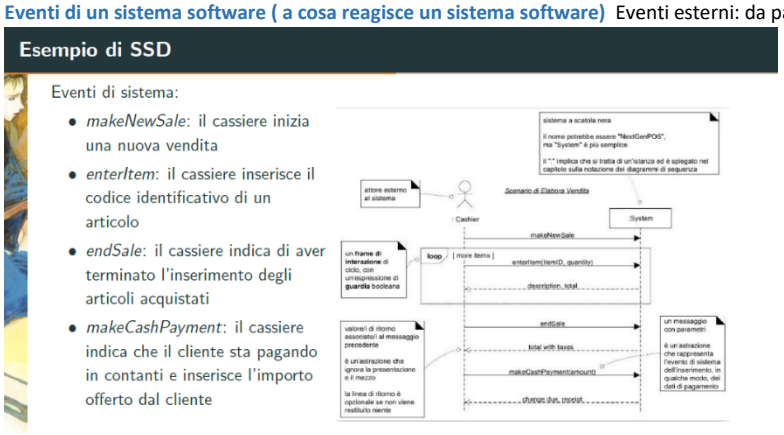
Il sistema è modellato come una "scatola nera", L'SSD è l'input dei contratti delle operazioni e per la progettazione a oggetti.

Mostra l'ordine degli eventi generati dagli attori del sistema / Mostrano gli eventi generati dagli attori esterni al sistema

Eventi Durante un'interazione con il sistema software un attore genera eventi di sistema, che costituiscono un input per il sistema, di solito per richiedere l'esecuzione di alcune operazioni di sistema.

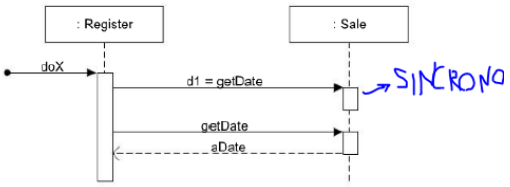
Le operazioni di sistema sono operazioni che il sistema deve definire per gestire tali eventi, un evento è qualcosa di importante che avviene durante l'esecuzione del sistema.

Eventi di un sistema software (a cosa reagisce un sistema software) Eventi esterni: da parte di attori umani o informatici / Eventi temporali / Guasti o eccezioni



Solitamente un SSD mostra gli eventi di sistema per un solo scenario
E puo essere generato per ispezione di tale scenario.

Un SSD mostra: L'attore primario del caso d'uso
Il sistema in discussione
I passi che rappresentano le interazioni tra sistema e attore



Operatori per i frame UML

Alt	Frammento alternativo per logica espressa nella guardia (if-else)
Opt	Frammento opzionale in base alla guardia (if)
Loop	Frammento ripetuto in base alla guardia (while/for)
Par	Frammenti eseguiti in parallelo
Region	Regione critica all'interno della quale si puo eseguire un solo thread

CONTRATTI

//Siamo sempre nella disciplina dei requisiti, nell ELABORAZIONE

Operazioni di sistema Le operazioni di sistema sono operazioni che il sistema offre nella sua interfaccia pubblica, possono essere identificate mentre si abbozzano gli SSD, gli SSD mostrano eventi di sistema

Un evento di sistema implica che il sistema definisca un'operazione di sistema per gestire quell'evento

Contratti delle operazioni I contratti usano pre e post condizioni per descrivere nel dettaglio i cambiamenti agli oggetti (concettuali) in un modello di dominio, possono Essere considerati parte del Modello dei casi d'uso perché forniscono maggiori dettagli dell'analisi sull'effetto delle operazioni di sistema
Sono strumenti dell'analisi dei requisiti per descrivere i cambiamenti richiesti durante l'esecuzione dell'operazione SENZA DESCRIVERE pero COME devono essere ottenuti questi cambiamenti

Sezioni di un contratto
Operazione Nome e parametri (firma) dell'operazione
Riferimenti Casi d'uso in cui puo verificarsi questa operazione
Pre-condizioni Ipotesi significative sullo stato del sistema o degli oggetti nel modello di dominio prima dell'esecuzione dell'operazione.
Post-condizioni Descrive il cambiamento degli oggetti nel Modello di Dominio dopo il completamento dell'operazione. I cambiamenti dello stato del modello di dominio comprendono oggetti creati, collegamenti formati o rotti, attributi modificati
NB NON sono azioni da eseguire nel corso dell'operazione, ma osservazioni sugli oggetti al termine dell'operazione

Come scrivere i contratti
1.Identificare le operazioni di sistema dagli SSD
2.Creare un contratto per le operazioni di sistema complesse, o i cui effetti sono sottili, o che non sono chiare dai casi d'uso
3.Per descrivere le post-condizioni si usano le seguenti sotto categorie: Creazione o cancellazione di oggetto (istanza)
Formazione o rottura di un collegamento
Modifica di un attributo

Trasformazione e interrogazione Ogni operazione di sistema può avere una componente di TRASFORMAZIONE: il sistema cambia il proprio stato
INTERROGAZIONE: Il sistema calcola e restituisce valori
Un'operazione di sistema HA POST CONDIZIONI SOLO se implica una trasformazione
NON HA POST CONDIZIONI se si tratta solo di una interrogazione

NB Non serve scrivere un contratto per ogni evento di sistema dell'SSD, solo per i più complessi
NB Se si scoprono nuove classi o attributi DEVONO essere aggiunte al modello di dominio in quanto UP è incrementale
NB Le post condizioni non è necessario siano in ogni momento super complete perché UP è iterativo e incrementale

ARCHITETTURA LOGICA E ORGANIZZAZIONE IN LAYER

Architettura la progettazione di un sistema orientato agli oggetti è suddiviso in strati architetturali (interfaccia, login …)

Architettura logica E' la macro organizzazione su larga scala delle classi software in package, sottoinsiemi e strati. Si chiama logica perché non vengono prese decisioni Su come gli elementi siano distribuiti sui processi o sui computer fisici della rete

Layer/Strato E' un gruppo di classi software, packages, sottoinsiemi con responsabilità condivisa su un aspetto importante del sistema.
Gli strati solitamente comprendono -*User interface* oggetti software per gestire l'interazione con l'utente e la gestione degli eventi
-*Application logic* o domain objects : logica applicativa o oggetti del dominio che rappresentano concetti del dominio
-*Technical services* Oggetti e sottosistemi d'uso generale che forniscono servizi tecnici di supporto

Architettura a strati E' la suddivisione di un sistema complesso in un insieme di elementi software che possono essere sviluppati e modificati in modo indipendente

Separation of concerns E' la separazione degli interessi, riduce l'accoppiamento e le dipendenze, aumenta la possibilità di riuso, manutenzione e chiarezza

Alta coesione In uno strato le responsabilità devono essere correlate, non mischiate, alta coesione porta a possibilità di riuso, manutenzione e chiarezza

Strato del dominio

Lo strato di dominio fa riferimento al modello di dominio per trarre ispirazione per i nomi delle classi dello strato del dominio

Si dice che gli strati di un'architettura rappresentano le sezioni verticali, mentre le partizioni rappresentano una divisione orizzontale di sottosistemi relativamente paralleli di uno strato.

Oggetto di dominio Oggetto software con nome e informazioni simili al dominio del mondo reale

Principio di separazione modello vista

- Non relazionare o accoppiare oggetti non UI con oggetti UI: gli oggetti non UI non devono essere connessi o accoppiato direttamente agli oggetti UI
- Non incapsulare la logica dell'applicazione in metodi di UI: non mettere la logica applicativa nei metodi di un oggetto dell'interfaccia utente

Modello E un sinonimo di STRATO DEGLI OGGETTI DEL DOMINIO

Vista E' un sinonimo per gli OGGETTI DELL'INTERFACCIA UTENTE
Principio di separazione modello-vista Gli oggetti del modello (dominio) non devono avere conoscenza diretta degli oggetti della vista (UI)
Le classi di dominio incapsulano le informazioni e il comportamento della logica applicativa
Le classi della vista sono responsabili dell'input e dell'output e di catturare eventi della GUI, ma NON MANTENGONO i dati dell'applicazione
Vantaggi del principio di separazione modello vista: Favorire la definizione coesa di modelli
Consentire lo sviluppo separato degli strati del modello e dell'interfaccia utente
Minimizzare l'impatto sullo strato del dominio dei cambiamenti dei requisiti relativi all'interfaccia
Consente di connettere facilmente nuove viste a uno strato del dominio esistente
Consente viste multiple, simultanee sugli stessi oggetti modello
Consente l'esecuzione dello strato di modello indipendente da quella dello strato dell'interfaccia utente
Consente un porting facile dello strato di modello ad un altro framework per l'interfaccia utente

MODELLAZIONE DINAMICA E STATICA CON UML

Metodi per progettare ad oggetti
Codifica Progettare mentre si codifica
Disegno, poi codifica Disegnare alcuni diagrammi UML, poi passare alla codifica
Solo disegno Lo strumento genera ogni cosa dai diagrammi

Tipi di modelli per gli oggetti
Modellazione dinamica: I modelli dinamici rappresentano il comportamento del sistema, la collaborazione tra oggetti SW per realizzare un caso d'uso
DIAGRAMMI DI INTERAZIONE UML (tipo i DSD)
Modellazione statica: I modelli statici servono per definire i package, i nomi delle classi, gli attributi e le firme delle operazioni
DIAGRAMMI DI CLASSE UML (Tipo il DCD)
La progettazione a oggetti richiede soprattutto la conoscenza di principi di assegnazione di responsabilità e design pattern

Diagrammi di interazione (modellazione dinamica)
Un'interazione è una specifica di come alcuni oggetti si scambiano messaggi per eseguire un compito nell'ambio di un contesto, l'iterazione è motivata dalla necessità di eseguire un determinato compito
Un compito è rappresentato da un messaggio che da inizio all'interazione (messaggio trovato)
Un messaggio è inviato a un oggetto designato come responsabile per questo compito, è la richiesta di eseguire una OPERAZIONE
L'oggetto responsabile collabora/interagisce con altri oggetti (partecipanti) per svolgere il suo compito
I partecipanti svolgono un ruolo nell'ambito della collaborazione
La collaborazione avviene mediante scambio di messaggi (interazioni)
Per le interazioni di usano i **DIAGRAMMI DI SEQUENZA DSD** che mostrano le interazioni
vantaggi: mostrano chiaramente la sequenza dell'ordinamento temporale dei messaggi
svantaggi: costringono a estendersi verso destra quando si aggiungono nuovi oggetti

Diagrammi di sequenza DSD
Mostrano le interazioni tra oggetti
I rettangoli sono **Linee di vita** , rappresentano i partecipanti all'interazione
Singleton Pattern nel quale da una sola casse viene istanziata una sola istanza, in un DSD un singleton si identifica con un 1 nel rettangolo
Messaggi sincroni: freccia piena bloccano l'esecuzione finchè non hanno finito la barra di esecuzione **Messaggi asincroni:** freccia con solo la punta
Il msg iniziale si chiama msg trovato
Modi di mostrare un risultato _____ d1 = getDate _____ oppure ------ > <- - - date - - - -
Self/this: si fa una freccia che va sulla linea di vita di se stessi
Creazione istanze: Si crea il rettangolo nel punto in cui si crea **Distruzione di istanze:** si mette una X sulla linea di vita
….. slide

Diagrammi delle classi (DCD)
Descrive le classi software + responsabilità di fare
Il diagramma delle classi di progetto è un diagramma delle classi utilizzato da un punto di vista software e di progetto
I DCD sono usati per la modellazione statica degli oggetti. Da un punto di vista concettuale servono a visualizzare un modello di dominio

DESGIN PATTERN GRASP - General Responsability Assignment Software Patterns

Responsabilità La progettazione OO è legata alle responsabilità

Principi e pattern Durante le attività di disegno e codifica devono essere applicati i vari principi di progettazione OO, PATTERN GRASP e GoF

Responsability-driven Development (RDD) L'approccio complessivo al fare la modellazione per la progettazione OO si baserà sulla metafora della **Progettazione guidata dalle responsabilità (RDD)** ovvero pensare a come assegnare le responsabilità a degli oggetti che collaborano.

Gli oggetti sono considerati dotati di responsabilità, ovvero un'astrazione di cio che fa l'oggetto

In UML la responsabilità è un contratto o un obbligo di un classificatore

Le responsabilità sono correlate agli obblighi o al comportamento di un oggetto in relazione al suo ruolo

La RDD porta a considerare un progetto OO come una comunità di oggetti con responsabilità che collaborano

Tipi di responsabilità **DI FARE** l'oggetto deve fare qualcosa, ad esempio creare un oggetto o calcolare qualcosa
Deve delegare ad altri oggetti di eseguire azioni
Deve controllare e coordinare le attività di altri oggetti
DI CONOSCERE Conoscere i propri dati privati incapsulati
Conoscere gli oggetti correlati
Conoscere cose che puo derivare o calcolar

Passi della RDD Identifica le responsabilità e considerale una alla volta
Chiediti a quale oggetto software assegnare la responsabilità
Chiediti come fa l'oggetto scelto a soddisfare questa responsabilità

Pattern GRASP: per l'assegnazione di responsabilità
Pattern GoF per idee di progettazione avanzate

GRASP

Questo procedimento va basato su opportuni criteri per l'assegnazione di responsabilità, come i pattern GRASP

I pattern GRASP per l'assegnamento di responsabilità rappresentano solo un aiuto per apprendere la struttura e dare un nome ai principi, ovvero sono un aiuto per l'apprendimento degli aspetti essenziali della progettazione.

Sono uno strumento per aiutare ad acquisire la padronanza delle basi dell'OOD e a comprendere l'assegnazione di responsabilità nella progettazione ad oggetti.

Pattern I principi e gli idiomi se codificati in un formato strutturato che descrive il problema e la soluzione a cui è assegnato un nome, possono essere chiamati patter
E' una coppia problema/soluzione ben conosciuta con nome che puo essere applicata a nuovi contesti
E' quindi una descrizione, con nome, di un problema di progettazione di una soluzione ben provata che si puo applicare a nuovi contesti

Low Rapresentational Gap (LRG) Nella progettazione vale sempre LRG (salto rappresentazionale basso) tra il modo in cui si pensa il dominio e una corrispondenza diretta con gli oggetti software

Progettazione modulare Comprensibilità, modificabilità, impatto nei cambiamenti basso, flessibilità, riuso, semplicità, sono sostenute dal principio della progettazione modulare, secondo cui il software deve essere decomposto in un insieme di moduli coesi e debolmente accoppiati

Obiettivi di GRASP Un sistema software ben progettato _e facile da comprendere, da mantenere e da estendere. Inoltre le scelte fatte consentono delle buone opportunita di riusare i suoi componenti software in applicazioni future.

Comprensione, manutenzione, estensione e riuso sono qualita fondamentali in un contesto di sviluppo iterativo, in cui il software viene continuamente modi_cato, estendendolo con nuove funzionalita (relative a nuove operazioni di sistema e a nuovi casi d'uso) oppure manutenendo le funzionalita implementate (per esempio, a fronte di cambiamenti nei requisiti). Comprensibilita e semplicita facilitano queste attivita evolutive.

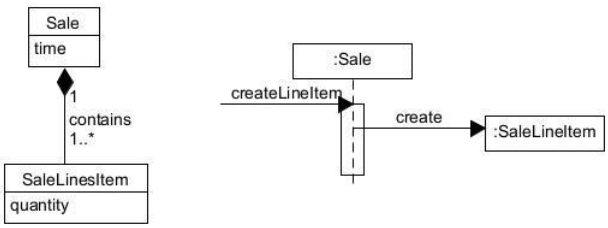
PATTERN CREATOR (grasp)

Problema Chi crea un oggetto A? Ovvero, chi deve essere responsabile della creazione di una nuova istanza di una classe?

Soluzione Assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (piu sono vere meglio è)
-B contiene o aggrega una composizione di oggetti di tipo A
-B registra A (salva il riferimento di una classe A)
-B utilizza strettamente A
-B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della creazione (B è un expert rispetto alla creazione di A)

E' importante trovare un creatore che abbia veramente bisogno di essere collegato all'oggetto creato. Si devono usare classi di supporto se la creazione puo essere in alternativa a "riciclo" o se una prorprietà esterna condiziona la scelta della classe creatrice tra un insieme di classi simili.

Creator e Low Coupling sono fortemente correlati perché Creator favorisce un accoppiamento basso

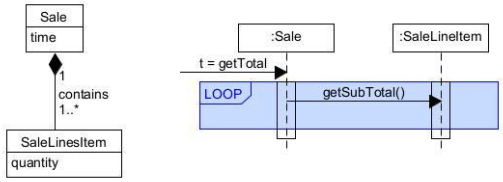


Sale ha la responsabilità di creare SaleLinesItem, quindi in Sale ci sarà un metodo per creare SaleLinesItem

PATTERN EXPERT (grasp)

Problema Qual è il principio base, generale, per l'assegnazione di responsabilità agli oggetti?

Soluzione Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla, all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità



Il miglior candidato per conoscere il totale complessivo di una vendita è Sale in quanto conosce tutte le istanze di SalesLineItem e quindi tutti i prezzi

Si devono individuare informazioni parziali di cui classi diverse sono esperte: queste classi collaborano insieme per raggiungere l'obiettivo

Sale salva i propri oggetti in DB? NO avrebbe problemi di coesione accoppiamento e duplicazione, DIVIDERE LE LOGICHE

PATTERN LOW COUPLING (grasp)

Problema Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?
Soluzione Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative

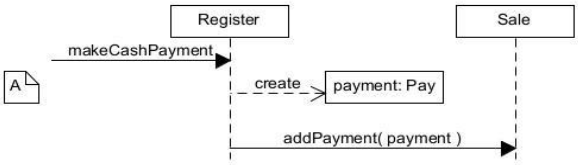
Il problema non è l'accoppiamento alto di per se, ma l'accoppiamento alto con elementi per certi aspetti instabili, per esempio nell'interfaccia, nell'implementazione o per loro pura e semplice presenza

Vantaggi Una classe o componente con basso accoppiamento non è influenzata dai cambiamenti nelle altre classi e componenti
E' semplice da capire separatamente dalle altre classi e componenti
E' conveniente da riusare

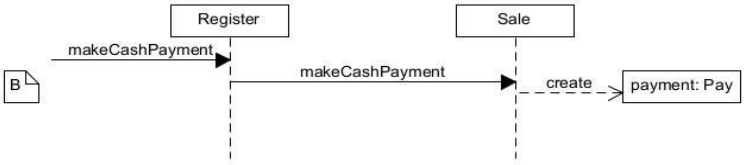
Una classe con accoppiamento alto dipende da molte altre classi. Tali classi fortemente accoppiate possono essere inopportune presentando i seguenti problemi:
-I cambiamenti delle classi correlate, da cui queste classi dipendono, obbligano a cambiamenti locali anche in queste classi
-Queste classi sono piu difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono
-Sono piu difficili da riusare, poiche il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono

Forme di accoppiamento
-La classe X ha un attributo (variabile di istanza o dato membro) di tipo Y o referenzia un'istanza di tipo Y o una collezione di oggetti Y
-Un oggetto di tipo X richiama operazioni o servizi di un oggetto di tipo Y
-Un oggetto di tipo X crea un oggetto di tipo Y
-Il tipo X ha un metodo che contiene un elemento di tipo Y o che referenzia un'istanza di tipo Y
-La classe X è sottoclasse, diretta o indiretta, della classe Y
-Y è un'interfaccia, e la classe X implementa questa interfaccia

•Le classi che sono per natura generiche e che hanno un'alta probabilita di riuso devono avere un accoppiamento particolarmente basso
• Un certo grado moderato di accoppiamento tra le classi _e normale, anzi _e necessario perla creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una collaborazione tra oggetti connessi
• Una sottoclasse _e fortemente accoppiata alla sua superclasse. Si consideri attentamente ogni decisione di estendere una superclasse, poiche e una forma di accoppiamento forte
• Porzioni di codice duplicato sono fortemente accoppiate tra di loro; infatti, la modifica di una copia spesso implica la necessita di modificare anche le altre copie



Accoppiamenti: 1) Register-CashPayment //durante la creazione
2) Register-Sale //per la add
3) Sale -Pay //per il passaggio di parametro



Accoppiamenti 1) Register-Sale
2) Sale-Payment

➔ B è meglio di A

PATTERN HIGH COESION (grasp)

Problema Come mantenere gli oggetti focalizzati, comprensibili e gestibili e come effetto collaterale sostenere il Low Coupling?
Soluzione Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative

La coesione è una misura di quanto fortemente siano correlate e concentrate le responsabilità di un elemento
Problemi di classi con coesione bassa Difficili da comprendere, mantenere, riusare, sono delicate e continuamente soggette a cambiamenti
Tipi di coesione
Coesione di dati Una classe implementa un tipo di dati (molto buona)
Coesione funzionale Gli elementi di una classe svolgono una singola funzione (buona o molto buona)
Coesione temporale Gli elementi sono raggruppati perché usati circa nello stesso tempo (es controller, a volte buona a volte no)
Coesione per pura coincidenza es una classe usata per raggruppare tutti i metodi il cui nome inizia per una certa lettera (molto cattiva)
Coesione
Coesione molto bassa: una classe è la sola responsabile di molte cose in aree funzionali molto diverse
Coesione bassa una classe ha da sola la responsabilità di un compito in una sola area funzionale
Coesione alta una classe ha responsabilità moderate in un area funzionale e collabora con altre classi per svolgere i suoi compiti
Coesione moderata Una classe ha da sola responsabilità in poche aree diverse logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra
Regola pratica Una classe con coesione alta ha un numero di metodi relativamente basso, con delle funzionalità altamente correlate e focalizzate e non fa troppo lavoro, collabora con altri oggetti se il compito è grande
Vantaggi si sostiene maggiore chiarezza e facilità di comprensione, spesso sostiene il low coupling, manutenzione semplificata, maggiore riuso di funzionalità

es. Un oggetto con coesione bassa fa molte operazioni diverse e non correlate.
Nell'esempio di prima B è meglio perché ogni classe fa solo i compiti distinti che gli spettano

PATTERN CONTROLLER (grasp)

Problema Qual è l'oggetto oltre lo strato UI che riceve e coordina un'operazione di sistema?
Soluzione Assegna la responsabilità ad un oggetto che rappresenta una delle seguenti scelte:
-Rappresenta il sistema complessivo, un oggetto radice, un dispositivo all'interno del quale viene eseguito il software, un punto di accesso al software (facade controller)
- Rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema (controller di caso d'uso o controller di sessione)
Nel caso di questo controller si usa la stessa classe controller per tutti gli eventi di sistema nello stesso UC, una sessione è un'istanza di una conversazione con l'attore

Controller come delega Controller è solo un pattern di delega. Gli oggetti dello strato UI devono delegare le richieste di lavoro agli oggetti di un altro strato
NB normalmente il controller deve delegare ad altri oggetti il lavoro da eseguire durante l'operazione di sistema, il controller coordina e controlla le attività, ma NON ESEGUE di per se molto lavoro

Usare la stessa classe controller per tutti gli eventi di sistema di un UC permette di conservare le informazioni sullo stato del caso d'uso
Controller MVC: Fa parte della UI e gestisce l'interazione con l'utente, la sua implementazione dipende in larga misura dalla tecnologia UI e dalla piattaforma che viene utilizzata
Controller GRASP: Fa parte dello strato di dominio e controlla o coordina la gestione delle richieste delle operazioni di sistema. Non dipende dalla tecnologia UI usata

Facade controller: Rappresenta il sistema complessivo, una facciata sopra gli altri strati dell'applicazione e fornisce un punto di accesso per le chiamate dei servizi
Controller di caso d'uso: Un controller diverso per ogni caso d'uso
Vantaggi Maggiore potenziale di riuso e interfaccie inseribili, Opportunita di ragionare sullo strato del caso d'uso

DESIGN PATTERN GOF

Ci sono tre tipi di pattern GoF classificati in base allo scopo
CREAZIONALI Abstarct Factory, Singleton
STRUTTURALI Adapter, Compisite, Decorator
COMPORTAMENTALI Observer, State, Strategy, Visitor

Il GoF predilige la COMPOSIZIONE rispetto all'EREDITARIETA' tra classi perché aiuta a mantenere le classi incapsulate e coese, la delegazione permette di rendere la composizione tanto potente quanto l'ereditarietà

Ereditarietà tra classi Definisce un oggetto in termini di un altro, riuso WHITE BOX la visibilità della sovraclasse è la visibilità della sottoclasse
Composizione di oggetti Le funzionalità sono ottenute assemblando o componendo oggetti per avere funzionalità piu complesse, Riuso BLACK BOX I dettagli interni non sono conosciuti

Il meccanismo di ereditarieta puo essere utilizzato in due modi diversi:

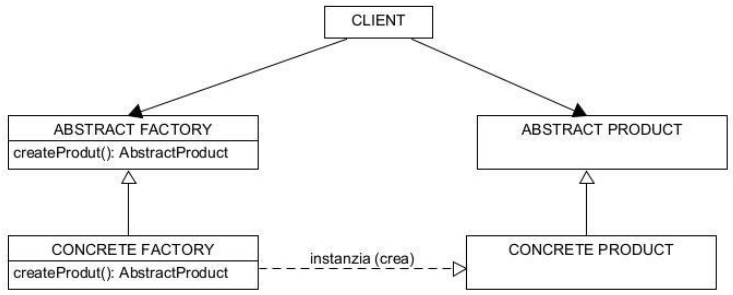
- Polimorfismo: le sottoclassi possono essere scambiate una per l'altra, possono essere `castate' in base al loro tipo, nascondendo il loro effettivo tipo alle classi cliente
- Specializzazione: le sottoclassi guadagnano elementi e proprieta rispetto la classe base, creando versioni specializzare rispetto alla classe base

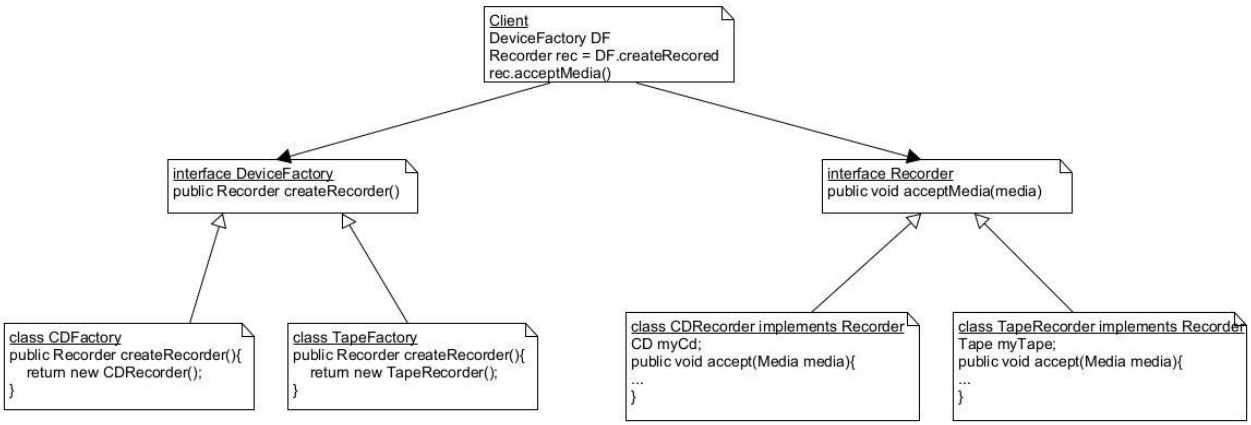
Il riuso è meglio ottenerlo attraverso il meccanismo di delega piuttosto che attraverso il meccanismo di ereditarietà, almeno per quanto riguarda l'utilizzo della specializzazione.

PATTERN CREAZIONALI

PATTERN ABSTRACT FACTORY (CREAZIONALI gof)

Problema Come creare famiglie di classi correlate che implementino un'interfaccia comune?
Soluzione Definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una cera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.





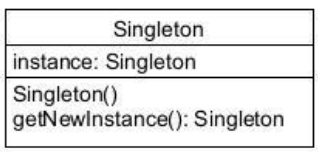
PATTERN SINGLETON (CREAZIONALI gof)

Problema E' consentita esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale al singleton
Soluzione Definisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton

Il singleton pattern definisce una classe della quale è consentita una sola instanziazione, tramite invocazione a un metodo della classe, incaricato della produzione degli oggetti. Le diverse richieste di instanziazione comportano la restituzione di un riferimento allo stesso oggetto

Tre diverse implementazioni in java **Singleton come classe statica** non è un vero e proprio singleton, si lavora con la classe statica, non un oggetto, questa classe ha metodi statici che offrono i servizi richiesti
Singleton creato da un metodo statico Una classe che ha un metodo statico che deve essere chiamato per restituire l'instanza del singleton. L'oggetto verrà istanziato solo la prima volta. Le successive sarà restituito un riferimento allo stesso oggetto (inizializzazione pigra)
Singleton multithread versione multi thread della soluzione precedente

Il singleton creato da un metodo statico è preferibile a quello creato come classe statica per tre motivi:
-I metodi di istanza consentono la ridefinizione nelle sottoclassi e il raffinamento della classe singleton in sottoclassi
-La maggior parte dei meccanismi di comunicazione remota orientati agli oggetti supporta l'accesso remoto solo a metodi di istanza
-Una classe non è sempre un singleton i tutti i contesti applicativi




```
class PrintSpooler
private static PrintSpooler instance;
private printSpooler(){}
public static PrintSpooler getInstance(){
   if(instance==null)
      instance = new PrintSpooler()
   return instance
}
```

PATTERN STRUTTURALI

PATTERN ADAPTER (STRUTTURALI gof)

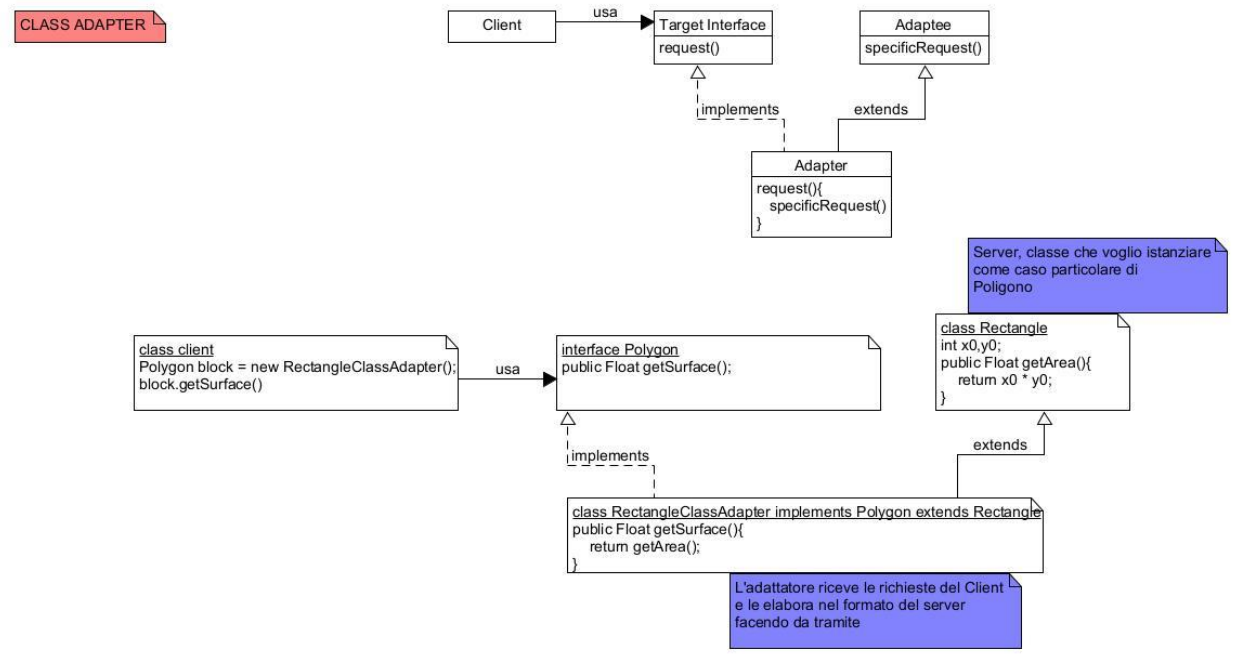
Problema Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?

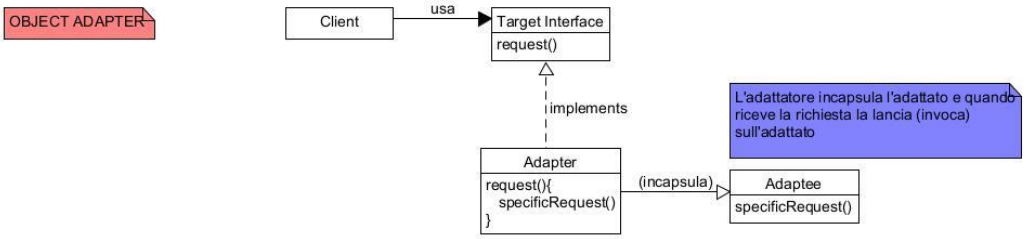
Soluzione Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio

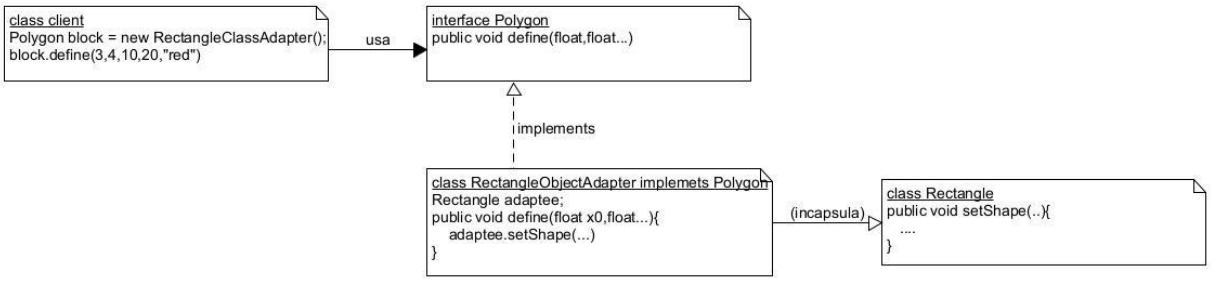
Ci sono 2 tipi di adapter: 1) Coppia di oggetti in relazione client-server. **Le interfacce sono incompatibili** quando l'oggetto server offre servizi di interesse per l'oggetto Client ma l'oggetto client vuole usufruirne in modo diverso.

2) Ci sono piu oggetti server che offrono servizi simili ma hanno intefacce diverse. Un ogg client vuole usufruire dei servizi da uno tra questi **componenti simili con interfacce diverse**

In generale un adattatore riceve richieste dai suoi client nel formato client dell'adattatore, poi adatta la richiesta al formato server e fa così da dispatcher







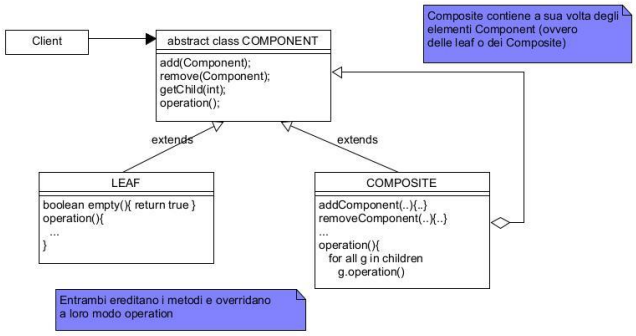
PATTERN COMPOSITE (Strutturali gof)

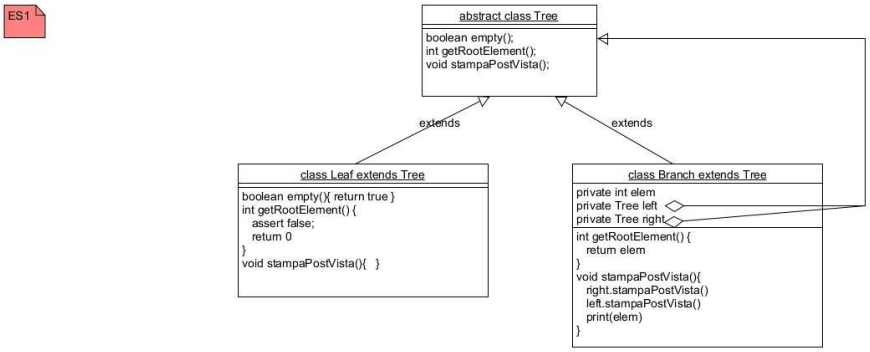
Problema Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto(atomico)
Soluzione Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia
Consente la costruzione di gerarchie di oggetti composti. Gli oggetti composti possono essere formati da oggetti singoli. Oppure da altri oggetti composti.
Questo pattern è utile nei casi in cui si vuole: rappresentare gerarchie di oggetti tutto-parte
Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti
E' nota anche come struttura ad albero, composizione ricorsiva, struttura induttiva: foglie e nodi hanno le stesse funzion
Implementa la stessa interfaccia per tutti gli elementi contenuti

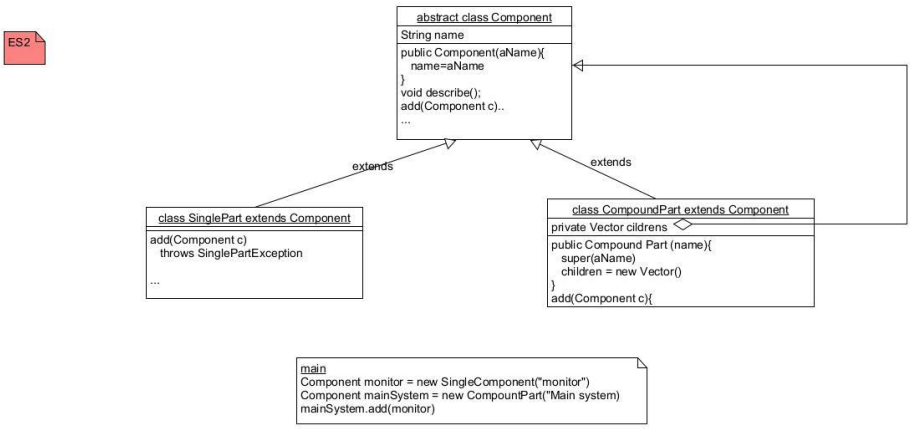
Il pattern definisce la classe astratta come componente che deve essere estesa in due sottoclassi:

- Una che rappresenta i singoli componenti (foglia)
- L'altra che rappresenta i componenti composti e che si implementa come contenitore di componenti

In breve il pattern composite permette di creare strutture ricorsive in modo che ad un client (una classe che usa la struttura) l'intera struttura sia vista come una singola entità. Quindi l'interfaccia alle entità atomiche (foglie) è esattamente la stessa dell'interfaccia delle entità composte. In essenza tutti gli elementi della struttura hanno la stessa interfaccia sia che siano composti o atomici.



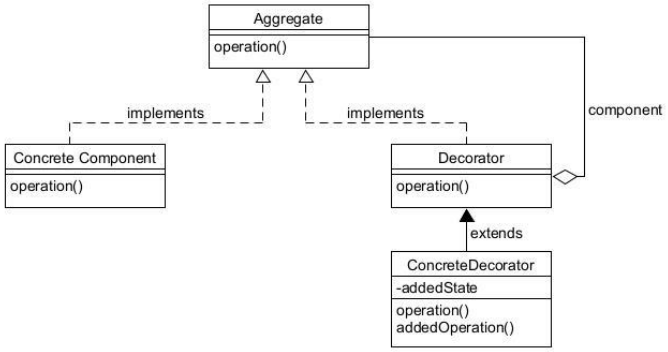


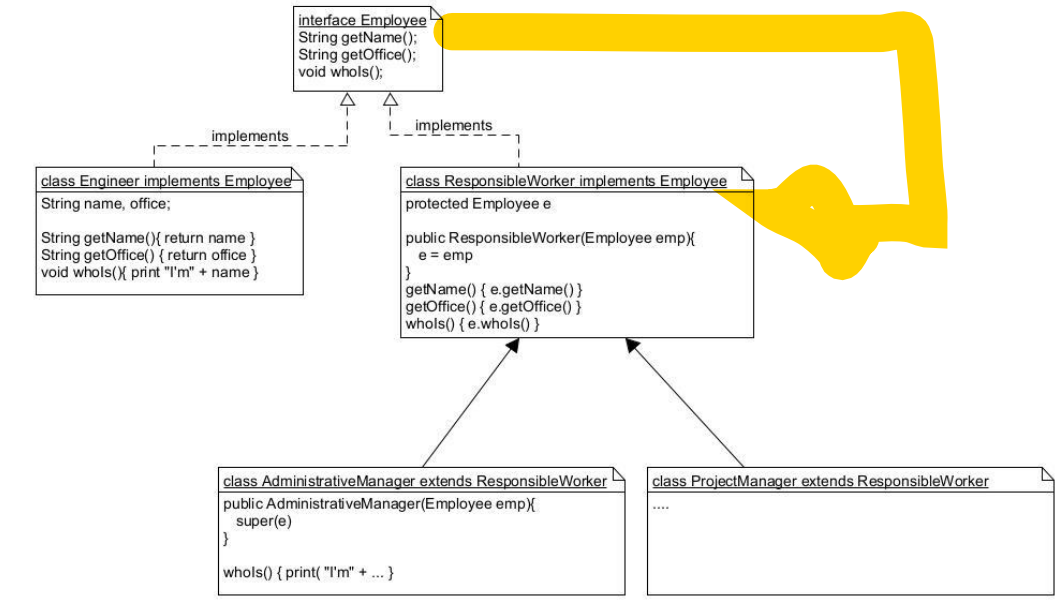


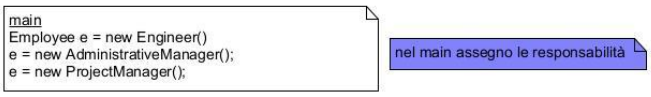
PATTERN DECORATOR (Strutturali GoF) //Noto anche come WRAPPER

Problema Come permettere di assegnare una o piu responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica?
Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

Soluzione Inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità.
Permette di aggiungere responsabilità ad oggetti individualmente, dinamicamente e in modo trasparente, ossia senza impatti su altri oggetti
Le responsabilità possono essere ritirate
Permette di evitare l'esplosione delle sottoclassi per supportare un ampio numero di estensioni e combinazioni di esse oppure quando le definizioni sono nascoste e non disponibili alle sottoclassi
- In sostanza il pattern decorator permette ad un'entità di contenere completamente un'altra entità cosi che l'utilizzo del decoratore sia identico all'entità contenuta. Questo consente al decoratore di modificare il comportamento e/o il contenuto di tutto cio che sta incapsulato senza cambiare l'aspetto esteriore dell'entità. Ad esempio e possibile usare un decoratore per aggiungere l'attività di logging dell'elemento contenuto senza cambiare il comportamento di questo.
- Composite vs decorator: composite fornisce un'interfaccia comune a elementi atomici (foglie) e composti, Decorator fornisce caratteristiche addizionali a elementi atomici mantenendo un'interfaccia comune







PATTERN COMPORTAMENTALI

PATTERN OBSERVER (Comportamentali GoF)

Problema Diversi tipi di oggetti subscriber sono interessati ai cambiamenti di stato o agli eventi di un publisher. Ciascun subscriber vuole reagire in un suo modo Proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

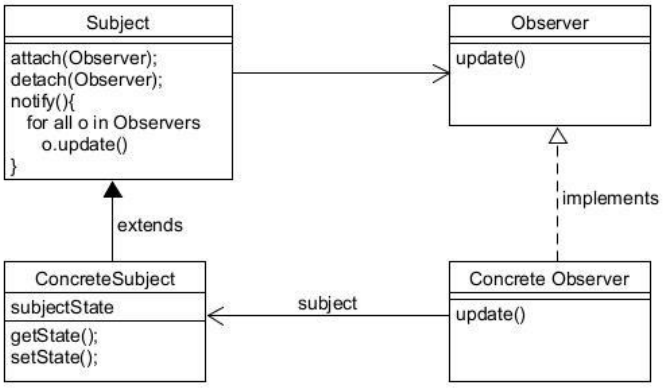
Soluzione Definisci un'interfaccia subscriber o LISTENER. Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subsriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

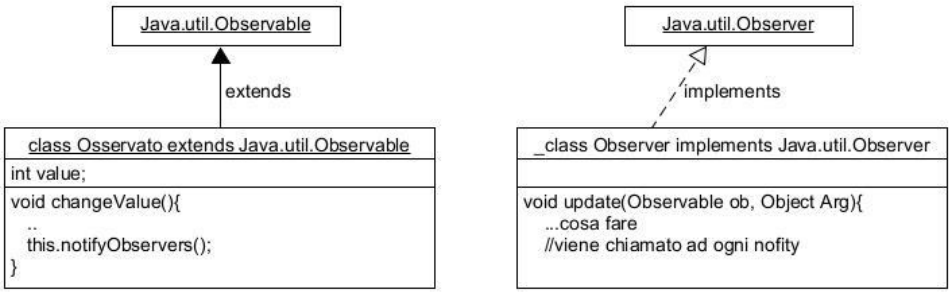
-Definisce una dipendenza tra oggetti di tipo uno-a-molti: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati.

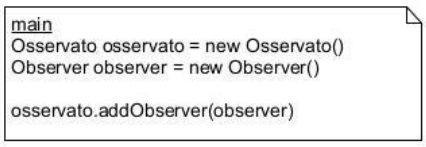
-L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti notificati: le due tipologie di oggetti sono disaccoppiati

-Il numero degli oggetti affetti dal cambiamento di stato di un oggetto non è noto a priori

-Fornisce un modo per accoppiare in maniera debole degli oggetti che devono comunicare. I publisher conoscono i subscriber solo attraverso un'interfaccia, e i subscriber possono registrarsi dinamicamente con il publisher







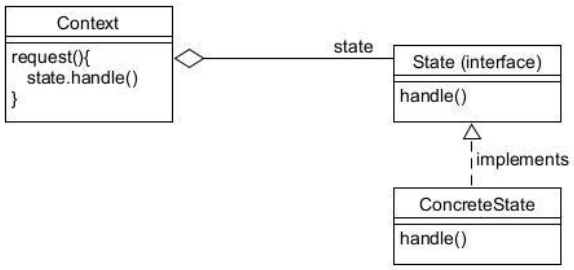
PATTERN STATE (Comportamentali GoF)

Problema Il comportamento di un oggetto dipende dal suo stato e i suoi metodi contengono logica condizionale per casi, che riflette le azioni condizionali che dipendono dallo stato. C'è una alternativa alla logica condizionale?

Soluzione Crea delle classi stato per ciascuno stato, che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato dall'oggetto conteso all'oggetto stato corrente corrispondente. Assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.

Permette ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno

Puo sembrare che l'oggetto modifichi la sua classe




```
class Clock
private ClockState clockState
private int ht,min

public Clock() {
    clockState = new NormalDisplayState( this );
}

 public void setState( ClockState cs ) {
    clockState = cs;
}
public void modeButton() {
    clockState.modeButton();
}
```

```
abstract class ClockState
protected Clock clock ;
public ClockState(Clock clock) {
    this.clock = clock;
}
public abstract void modeButton();
public abstract void changeButton();
```

implements

implements

```
class NormalDisplayState extends ClockState

public NormalDisplayState(Clock clock) {
    super(clock)
}
public abstract void modeButton(){
    clock.setState( new UpdateHrState(clock) )
```

```
class UpdateHrState extends ClockState

public UpdateHrState(Clock clock) {
    super(clock)
}
public abstract void modeButton(){
    clock.setState( ...new  )
```

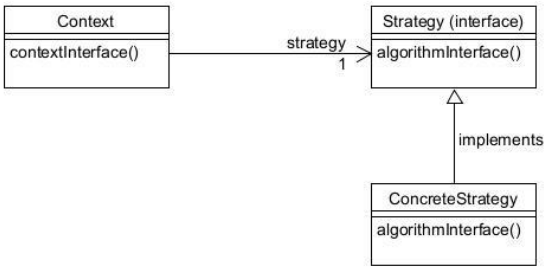
PATTERN STRATEGY (GoF comportamentali) //noto anche come policy

Problema Come progettare per gestire un insieme di algoritmi/politiche variabili ma correlati? Come progettare per permettere di modificare questi algoritmi/politiche?

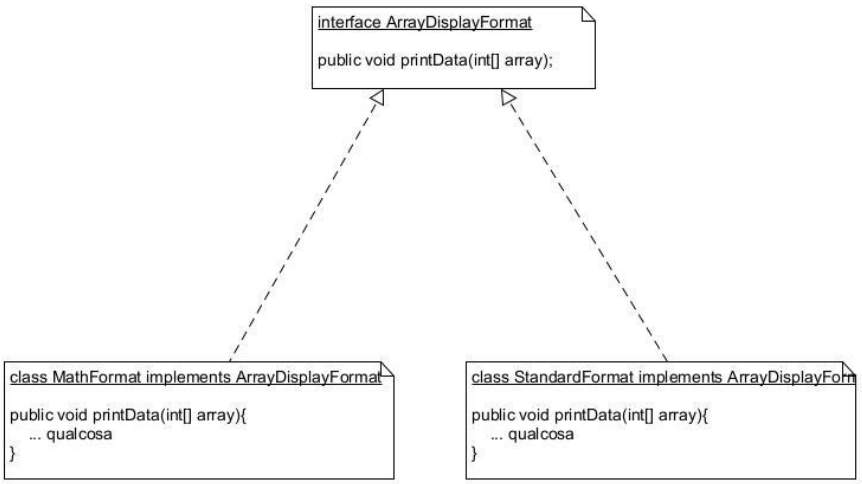
Soluzione Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune

-L'oggetto contesto e l'oggetto a cui va applicato l'algoritmo
-L'oggetto contesto è associato a un oggetto strategia, che è l'oggetto che implementa un algoritmo
-Consente la definizione di una famiglia di algoritmi, incapsula ognuno e li rende intercambiabili tra loro
-Permette di modificare gli algoritmi in modo indipendente dai client che ne fanno uso
-Disaccoppia gli algoritmi dai client che vogliono usarli dinamicamente
-Permette che un oggetto client possa usare indifferentemente uno o l'altro algoritmo
-E' utile dove è necessario modificare il comportamento a runtime di una classe
-Usa la composizione invece dell'ereditarietà: i comportamenti di una classe non dovrebbero essere ereditati ma incapsulati usando la dichiarazione di interfaccia

E' molto simile a state, State si occupa di che cosa (stato o tipo) un oggetto è (al suo interno) e incapsula un comportamento dipendente dallo stato. Mentre Strategy si occupa del modo in cui un oggetto esegue un determinato compito: incapsula un algoritmo




```
class MyArray

private int array[]
int size;
ArrayDisplayFormat format;

public MyArray( int size ) {
    array = new int[ size ];
}
...
public int getValue( int pos ) {
    return array[pos];
}
public void setDisplayFormat( ArrayDisplayFormat adf ) {
      format = adf;
}
```



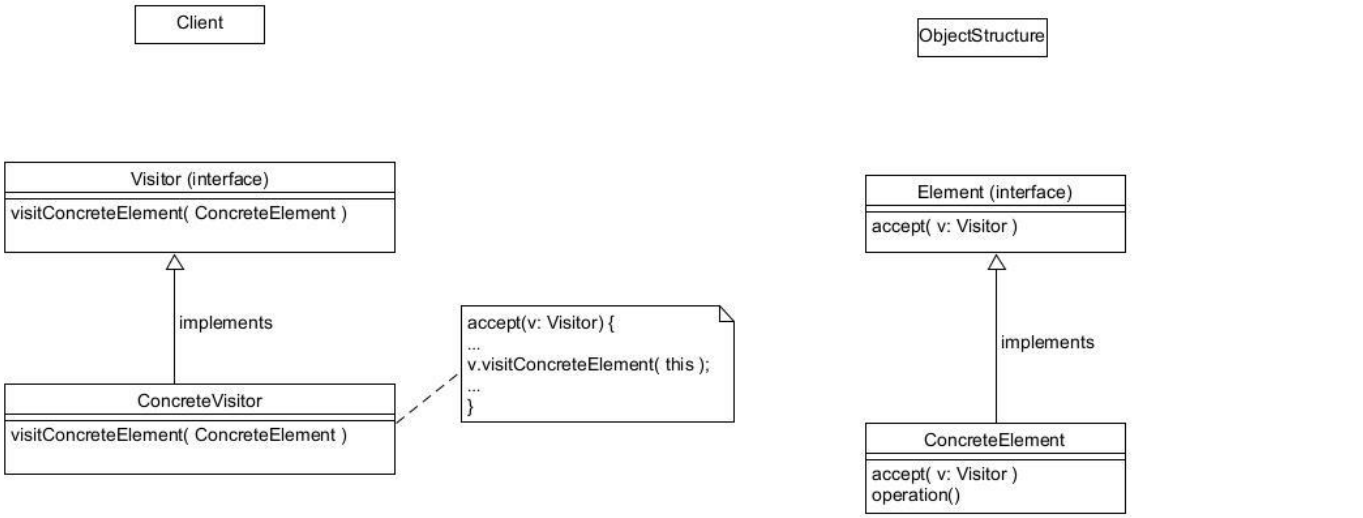

```
main
MyArray m = new MyArray(10);

m.setDisplayFormat( new MathFormat );
m.display
```

PATTERN VISITOR (GoF Comportamentali)

Problema Come separare l'operazione applicata su un contenitore complesso alla struttura dati a cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo di elementi?

Soluzione Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad una interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

-Flessibilità delle operazioni
-Organizzazione logica
-Visita di vari tipi di classe
-Mantenimento di uno stato aggiornabile ad ogni visita
-Le diverse modalità div visita della struttura possono essere definite come sottoclassi del Visitor




```
client
WhatcherVisitor browser = new WatcherVisitor()
browser.visit( aVectorWithFloatAndStrings )
```

```
interface Visitor
void visit( VisitableString s);
void visit (VisitableFloat f);
```

implements

```
class WatcherVisitor implements Visitor
public int stringhe = 0;
public int numFloat = 0;

public void visit (VisitableString vString) {
   stringhe++
   print( vString.getString() )
}

public void visit (VisitableFloat vFloat ){
   numFloat++
   print ( vFloat.getFloat().toString() )
}


public void visit(Collection collection) {
    Iterator iterator = collection.iterator();
    while (iterator.hasNext()) {
       Object o = iterator.next();
       if (o instanceof Visitable)
          ((Visitable)o).accept(this);
       else if (o instanceof Collection )
          visit( (Collection) o );
    }
}
```

```
interface Visitable
void accept( Visitor visitor);
```

implements implements

```
class VisitableString implements Visitable
private String value;

public String getString(){
   return value
}

public accept(Visitor visitor){
   visitor.visit(this)
}
```

```
class VisitableFloat implements Visitable
private Float value;

public String getFloat(){
   return value
}

public accept(Visitor visitor){
   visitor.visit(this)
}
```

Dal progetto al codice

Trasformare i progetti in codice

La sequenza dei messaggi in un diagramma di interazione (DSD) si traduce in una serie di istruzione nelle definizioni di metodi e costruttori
Le relazioni uno-a-molti sono implementate di solito con l'introduzione di un oggetto collezione
Le classi possono essere implementate in modo e in ordine diverso

Sviluppo guidato dai test e refactoring

Extreme Programming (XP) e test Extreme programming ha promosso la pratica dei test: scrivere i test per primi
Extreme Programming (XP) e refactoring Extreme programming ha promosso il refactoring continuo del codice per migliorare la qualità: meno duplicazioni, più chiarezza ecc.
Test driven Development (TDD) Una pratica promossa dal metodo iterativo e agile XP (applicabile a UP) è lo **sviluppo guidato dai test** noto anche come sviluppo Preceduto dai test. Il codice di test è scritto **prima** del codice da verificare, immaginando che il codice da testare sia scritto.

Vantaggi dello sviluppo guidato dai test

- I test unitari (ovvero i test relativi a singole classe e metodi) vengono effettivamente scritti
- La soddisfazione del programmatore porta a una scrittura più coerente dei test1
- Chiarimento dell'interfaccia e del comportamento dettagliati
- Verifica dimostrabile, ripetibile e automatica
- Fiducia nei cambiamenti

In generale il TDD prevede l'utilizzo di diversi tipi di test:

- **Test unitari**: hanno lo scopo di verificare il funzionamento delle piccole parti (unita) del sistema ma non di verificare il sistema nel suo complesso
- **Test di integrazione**: per verificare la comunicazione tra specifiche parti (elementi strutturali) del sistema
- **Test end-to-end**: per verificare il collegamento complessivo tra tutti gli elementi del sistema
- **Test di accettazione**: hanno lo scopo di verificare il funzionamento complessivo del sistema, considerato a scatola nera e dal punto di vista dell'utente, ovvero con riferimento a scenari di casi d'uso del sistema

Test unitari

Un metodo di test unitario e logicamente composto da quattro parti:

- Preparazione: crea l'oggetto (o il gruppo di oggetti) da verificare (chiamato anche la fixture) e prepara altri oggetti e/o risorse necessari per l'esecuzione del test
- Esecuzione: fa fare qualcosa alla fixture (per esempio, eseguire delle operazioni), viene richiesto lo specifico comportamento da verficare
- Verifica: valuta che i risultati ottenuti corrispondano a quelli previsti
- Rilascio: opzionalmente rilascia o ripulisce gli oggetti e/o le risorse utilizzate nel test (per evitare che altri test vengano corrotti)

Refactoring Il refactoring è un metodo strutturato e disciplinato per scrivere o ristrutturare del codice esistente senza però modificare il comportamento esterno applicando piccoli passi di trasformazione in combinazione con la ripetizione dei testa d ogni passo.
Refactoring e test Dopo ciascuna trasformazione i testi unitari vengono eseguiti nuovamente per dimostrare che il refactoring non abbia provocato una regressione (fallimento). C'è una relazione tra refactoring e il TDD: tutti i test unitari sostengono il processo di refactoring

Obiettivi del refactoring

Gli obiettivi del refactoring sono gli obiettivi e le attivita di una buona programmazione:

- Eliminare il codice duplicato
- Migliorare la chiarezza
- Abbreviare i metodi lunghi
- Eliminare l'uso dei letterali costanti hard-coded
- altro...

Regole per il TDD e Refactoring

Regole per il TDD e refactoring:

- Scrivi un test unitario che fallisce, per dimostrare la mancanza di una funzionalita o di codice
- Scrivi il codice piu semplice possibile per far passare il test
- Riscrivi o ristruttura (refactor) il codice, migliorandolo, oppure passa a scrivere il prossimo test unitario

Alcuni refactoring

Refactoring	Descrizione
Rename	Per cambiare il nome di una classe, un metodo o un campo, per rendere più evidente il suo scopo. Semplice ma estremamente utile.
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 da un'altra classe.
Extract Constant	Sostituisce un letterale costante con una variabile costante.
Move Method	Crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più.
Introduce Explaining Variable	Mette il risultato dell'espressione, o di una parte dell'espressione, in una variabile temporanea con un nome che ne spiega lo scopo.
Replace Constructor Call with Factory Method	In Java, per esempio, sostituisce l'uso dell'operazione *new* e la chiamata di un costruttore con l'invocazione di un metodo di supporto che crea l'oggetto (nascondendo i dettagli).

Extend rappresenta una dipendenza, il passaggio di parametro no.

scenario di sviluppo

Tabella 2.1 Scenario di Sviluppo di esempio (i – inizio; r – raffinamento).

Disciplina	**Pratica**	**Elaborato** Iterazione →	**Ideazione** I1	**Elaboraz.** E1..En	**Costr.** C1..Cn	**Transiz.** T1..T2
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio		i		
Requisiti	workshop requisiti esercizio sulla visione votazione a punti	Modello dei Casi d'Uso	i	r		
		Visione	i	r		
		Specifica Supplementare	i	r		
		Glossario	i	r		
Progettazione	modellazione agile sviluppo guidato dai test	Modello di Progetto		i	r	
		Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	
Implementazione	sviluppo guidato dai test programmazione a coppie integrazione continua standard di codifica	...				
Gestione del progetto	gestione del progetto agile riunioni Scrum	...				