ANALISI PROGETTAZIONE DEL SOFTWARE SECONDO SEMESTRE 2023

PROF. RIGANELLO [T1]

Andrea Falbo - Quack

CAPITOLO 1: INTRODUZIONE AL CORSO E A PROCESSI DI SVILUPPO

INTRODUZIONE

Software

Il *software* è un programma e la relativa documentazione: specifica requisiti, modelli, progetto, manuale utente.

Un software può essere un prodotto:

- *generico*, ovvero per un ampio insieme di clienti
- personalizzato, per un singolo cliente

Le caratteristiche di un buon software sono:

- *mantenibilità*: deve poter essere capito da un futuro collaboratore
- *fidatezza*: deve essere affidabile, sicuro e protetto.
- *efficienza*, non bisogna avere spreco di risorse, buona reattività agli input e non deve usare eccessiva CPU o memoria RAM
- *accettabilità*: deve essere compatibile con altri sistemi.

L'*Ingegneria del Software* è la disciplina che si occupa di *tutti* gli aspetti della produzione di un software di buona qualità.

La *Disciplina Ingegneristica* è utilizzare *metodologie*, *strumenti* e *tecniche* appropriate per risolvere i problemi, tenendo conto dei vincoli organizzativi e finanziari.

Le attività di processo necessarie a tutti i processi software sono:

- *Analisi dei requisiti*: definire i servizi richiesti
- *Progettazione*: organizzazione sistema software
- *Implementazione*: si passa da progetto a codice
- *Evoluzione*: cambiare in base alle esigenze
- *Validazione*: verificare che il sistema fa ciò che il cliente chiede

N.B.: In questo corso ci occuperemo principalmente delle prime due fasi: *fare la cosa giusta (analisi) e fare la cosa bene (progettazione)*

Una descrizione dei processi ci viene fornita da:

- *Artefatti*: sono le attività da produrre
- *Ruoli*: come partecipano le persone
- *Pre e Post Condizioni*: regole da rispettare per poter eseguire un'attività

Le 4 Fasi dell' Analisi e Progettazione

Analisi: Enfatizza un'investigazione di un problema e dei suoi requisiti anziché di una soluzione.
Analisi ad Oggetti: enfatizza sull'identificazione degli oggetti nel dominio di un problema.

Progettazione: enfatizza una soluzione concettuale che soddisfi i requisiti.
Progettazione ad Oggetti: enfatizza sulla definizione di oggetti software che collaborano per poter soddisfare i requisiti

Ci sono 4 possibili fasi nell'analisi e progettazione del software:

- *casi d'uso*: storie su come il sistema è utilizzato.
- *modello di dominio*: mostra i concetti o oggetti del dominio con relative associazioni
- *definizione diagrammi di interazione*: vista dinamica sulle collaborazioni tra oggetti software

- definizione dei diagrammi delle classi: vista statica delle classi software, con attributi e metodi

Introduzione a UML

L'*UML* (Unified Model Language) è un linguaggio visuale di modellazione di sistemi software. Offre sia strutture statiche che dinamiche. Si può applicare in 3 modalità:

- *Abbozzo*: diagrammi informali ed incompleti
- Progetto: diagrammi dettagliati e completi
- Linguaggio: specifica completamente eseguibile di un sistema software

N.B. : Useremo UML applicato come abbozzo.

UML ha due punti di vista:

- *Concettuale*: come nei modelli di dominio, visualizziamo concetti del mondo reale
- Software: come nelle classi di progetto, visualizziamo concetti software

PROCESSI SOFTWARE

Processo di Sviluppo

Un *processo di sviluppo* di un software definisce l'approccio per la costruzione: chi fa che cosa, come e quando:

- cosa sono le attività
- chi sono i ruoli
- come sono le metodologie
- quando riguarda l'organizzazione temporale

Un *processo a cascata* è un sistema di processo di sviluppo. Si cerca di suddividere il processo in più parti, alle quali si può accedere soltanto una volta terminata la sezione prima. La suddivisione inflessibile del progetto in fasi distinte rende difficile rispondere alle mutevoli esigenze dei clienti. Pertanto, questo modello è appropriato solo quando i requisiti sono ben compresi e le modifiche saranno relativamente limitate durante il processo di progettazione

Sviluppo Iterativo, Incrementale ed Evolutivo

Nello Sviluppo *Iterativo, Incrementale ed Evolutivo* abbiamo per ciascuna iterazione lo sviluppo di un piccolo sottoinsieme dei requisiti. Il lavoro procede in costruzione-feedback-adattamento. C'è minore probabilità di fallimento, migliore produttività e meno difetti. I problemi a rischio maggiore vengono risolti nelle prime fasi anziché nelle ultime. Inoltre, c'è maggior coinvolgimento del cliente.

Il *Timeboxing* è la durata di ogni iterazione, che può variare dalle 2 alle 6 settimane. Una volta stabilito non si può più modificare, piuttosto si riduce il carico di lavoro.

Il progetto deve essere *flessibile* in quanto l'impatto dei cambiamenti sia basso. A tal fine il codice sorgente deve essere facilmente modificabile e quindi comprensibile.
La mancanza di queste qualità rende infatti difficile l'applicazione dello sviluppo iterativo, in cui il codice va esteso in modo incrementale

La *Pianificazione* è per l'appunto iterativa, ovvero non tentiamo di pianificare l'intero progetto dall'inizio, bensì cerchiamo di applicare:

- guida da *rischio*: identificare ed attenuare i rischi maggiori
- guida da *cliente*: costruire e rendere visibile le caratteristiche a cui il cliente tiene

UP (Unified Process) è un esempio di processo iterativo per sviluppo software ad oggetti. Ogni iterazione è un mini progetto che include:

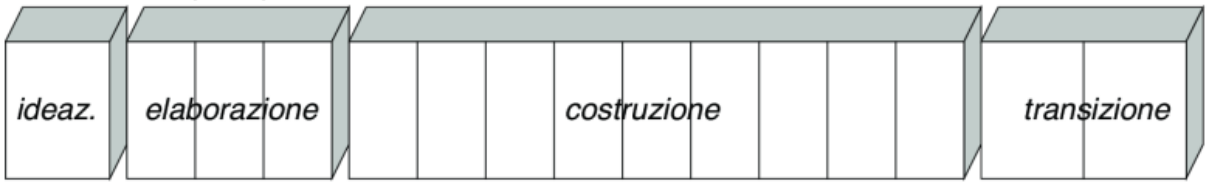
- Pianificazione
- Analisi e Progettazione
- Costruzione
- Test
- Rilascio

Una Release è un insieme di manufatti, previsti e approvati. Un incremento è la differenza tra una release e la successiva.

Le 4 Fasi Del Processo IIE

Il Processo si può dividere in 4 Fasi temporali:

- *Ideazione*: visione approssimativa, stime su costi e tempi
- *Elaborazione*: realizzazione del nucleo dell'architettura, risoluzione rischi maggiori
- *Costruzione*: risoluzione elementi a rischio minore
- *Transizione*: completamento, beta tester e rilascio



Una *Milestone* è la fine di un'interazione in cui si verifica una decisione significativa.

Sviluppo Agile

Lo *Sviluppo Agile* è una forma di sviluppo iterativo che incoraggia una risposta rapida e flessibile a cambiamenti con consegne incrementali. Utilizza:

- *Valori Agili*: semplicità, leggeri, comunicazione
- *Pratiche Agili*: programmazione a coppie, sviluppo guidato dai test (TDD), modificare porzioni di codice senza modificare comportamento esterno (refactoring)

L'*Agile Modeling* è agevolare la comprensione e la comunicazione sulle parti più difficili. Spesso si esegue in 2-3. I modelli creati non sono completi.

UP Agile: UP nasce pesante ma si può applicare un metodo agile.

Scrum è un metodo agile che consente di sviluppare e rilasciare prodotti software con il più alto valore per i clienti nel minor tempo possibile. Terminologia:

- *Scrum*: incontro breve faccia a faccia per definire le priorità del lavoro del giorno
- *Scrummaster*: coach, si assicura che si utilizzi scrum.
- *Sprint*: un'interazione di sviluppo, dura 2-4 settimane.
- *Velocity*: stima di quanto lavoro rimane che un team può fare in uno sprint.

- *Team di Sviluppo*: gruppo da massimo 7 persone responsabili dello sviluppo software.
- *Incremento potenzialmente rilasciabile*: ciò che dovrebbe essere in uno stato finito
- *Product baking*: elenco di elementi da fare
- *Product owner*: individuo il cui compito è rivedere ed aggiornare il product baking.

CAPITOLO 2: ANALISI DEI REQUISITI E CASI D'USO

ANALISI DEI REQUISITI

Prima Fase IIE: Ideazione

L'*Ideazione* è la prima fase temporale in cui dobbiamo chiederci, dopo una veloce analisi, se ha senso *proseguire o fermarsi*. Le parti interessate hanno un accordo di base sulla visione del progetto, e vale la pena di investire un'indagine seria?

Requisito

Un *Requisito* è una capacità o condizione a cui il sistema deve essere conforme. Il 34% delle cause di fallimenti di progetti riguardano attività di requisiti. Possono essere:

- *Funzionali*: descrivono il comportamento del sistema in termini di funzionalità ed informazioni
- *Non Funzionali*: sono relativi al sistema: sicurezza, prestazioni ecc. Possono essere molto difficili da dimostrare con precisione e quindi è utile verificarli attraverso unità di misura.

Esempio: Il sistema POS NextGen ha come requisiti funzionali:

- Deve consentire di verificare la disponibilità di un prodotto in tutti i negozi appartenenti alla stessa catena.
- Deve aggiornare ad ogni vendita le quantità dei prodotti disponibili in negozio
- Deve associare ad ogni vendita un identificativo univoco

Mentre ha come requisiti non funzionali:

- Requisiti del prodotto: Dovrebbe essere a disposizione di tutti i negozi durante il normale orario di lavoro (Lun-Ven, 08.30-19.30). L'orario di inattività nelle normali ore di lavoro non deve superare i cinque secondi in un giorno.
- Requisiti organizzativi: Dovrebbe autenticare gli utenti attraverso la loro tessera identificativa.
- Requisiti esterni: Dovrebbe applicare le disposizioni in materia di tutela della privacy dei clienti stabilite nella normativa italiana emanata con decreto legislativo 28 febbraio 2021 n. XYZ

C'è una linea guida su come scrivere i requisiti composta dalle seguenti regole:

- Ideare un *formato* standard ed utilizzarlo per tutti i requisiti
- Utilizzo del *linguaggio* in modo consistente. Utilizzare DEVE per requisiti obbligatori, DOVREBBE per requisiti desiderabili.
- Utilizzo dell'evidenziazione delle porzioni di testo per *identificare* le parti più importanti dei requisiti.
- *Evitare il gergo informatico*
- Includere una *spiegazione* (razionale) del motivo per cui è necessaria una disposizione.

Un *attore* è qualcosa o qualcuno dotato di comportamento.
Uno *scenario* è una sequenza specifica di azioni tra il sistema e alcuni attori.
Un *caso d'uso* è una correlazione di scenari correlati.
Il *Modello dei Casi d'Uso* è l'insieme di tutti i casi d'uso scritti.

Ci sono diversi tipi di attore:

- Attore *primario*: raggiunge degli obiettivi usando il Sistema in Discussione (SuD)
- Attore *finale*: vuole che il SuD sia utilizzato affinché vengono raggiunti dei suoi obiettivi
- Attore di *supporto*: offre un servizio al SuD
- Attore *fuori scena*: ha un interesse nel comportamento del caso d'uso SuD

Prima Fase AeP: Casi d'uso

I *casi d'uso* sono requisiti funzionali: sono storie scritte, ampiamente utilizzati per scoprire e registrare i requisiti. Non sono elaborati orientati agli oggetti, però influenzano molti aspetti di un progetto.

L'*analista di sistema* è colui che si occupa di trovare i casi d'uso e gli attori. Per farlo utilizza:

1. *Modello di* business o di dominio
2. *Modello dei requisiti*
3. *Elenco* delle *features*

E produce in output:

1. *Modello* dei *casi d'uso*
2. *Glossario* di progetto

I casi d'uso possono essere scritti usando diversi formati e livelli di formalità:

- Formato *breve*: riepilogo conciso di un solo paragrafo, normalmente relativo al solo scenario principale di successo
- Formato *informale*: più paragrafi scritti in modo informale, relativi a vari scenari
- Formato *dettagliato*: tutti i passi e le variazioni sono scritti nel dettaglio; ci sono anche delle sezioni di supporto, come le pre-condizioni e le garanzie di successo.

Scrivere i casi di uso in uno stile essenziale e concreto, ignorando l'interfaccia utente, concentrarsi sullo scopo dell'attore. Importante scriverli a scatola nera, ovvero specificando cosa deve fare il sistema, senza spiegare il come.

Per trovare i casi d'uso sono necessarie 4 fasi:

- Scegliere il confine di sistema: una volta identificati gli attori esterni, i confini diventano più chiari.
- *Identificare gli attori primari*: porsi domande sul sistema.
- *Identificare gli obiettivi* per ogni attore primario conseguenza del punto 2.

- *Definire i casi d'uso* che soddisfano questi obiettivi

È opportuno assegnare al caso d'uso un nome simile all'obiettivo dell'utente.

Esempio: Obiettivo: elaborare una vendita —> caso d'uso: Elabora Vendita.

ll nome dei casi d'uso deve iniziare con un verbo, tranne gli obiettivi CRUD (Create, Retrieve, Update, Delete) che si chiamano Gestisci X.

Per verificare la validità dei casi d'uso abbiamo 3 test:

- *Test del capo*: se il nostro capo sa che stiamo facendo x tutto il giorno sarà felice?
- *Test EBP*: processo di business elementare: non deve essere un singolo passo, deve essere eseguita in una sessione di lavoro e deve portare incremento di valore.
- *Test Dimensione*: Un buon caso d'uso non dovrebbe essere troppo breve.

Livelli per i casi d'uso: Abbiamo diversi livelli dei casi d'uso:

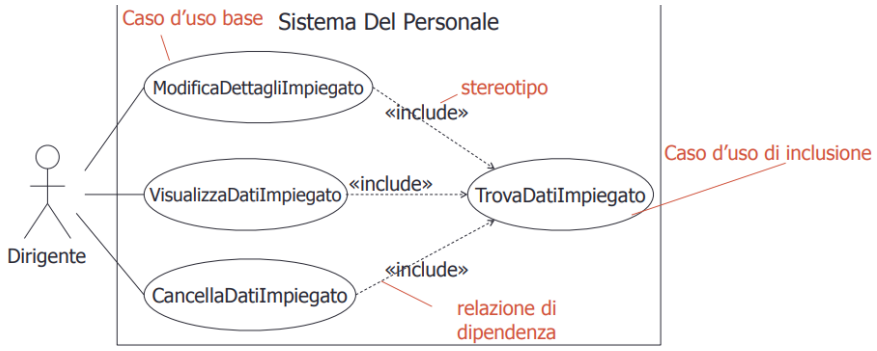
- Livello di *obiettivo utente*: è il più interessante nell'analisi dei requisiti
- Livello di *sotto-funzione*: utili per mettere a fattor comune parti di casi d'uso e/o per descrivere interazioni di dettaglio
- Livello di *sommario*: un caso d'uso a questo livello comprende più casi d'uso a livello di obiettivo utente

Notazione per descrivere i casi d'uso:

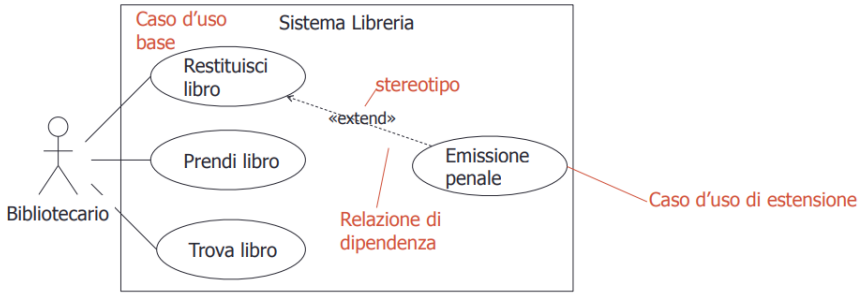
- Attori primari a sinistra, attori di supporto a destra
- Notazioni diverse per utenti umani ed informatici
- Considerare solo casi d'uso livello utente
- Associazione rappresentata da una singola linea direzionata

Ci sono 3 relazioni tra casi d'uso:

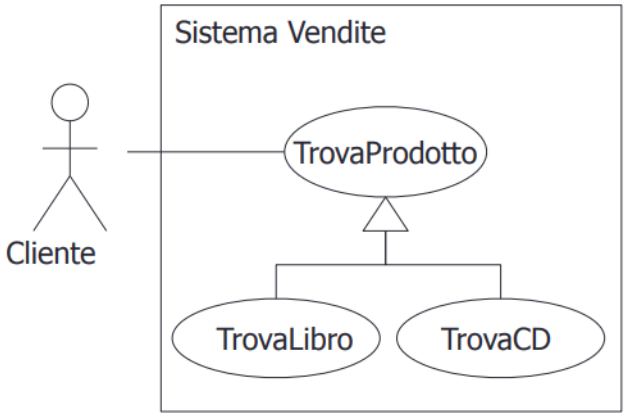
- *Include*:



- *Estende*: Aggiunge varianti ad un caso d'uso base.



- *Generalizzazione*: possono ereditare, aggiungere e sovrascrivere le funzioni del loro genitore. (può avvenire anche tra attori e non solo casi d'uso)



CAPITOLO 3: ANALISI ORIENTATA AD OGGETTI

MODELLI DI DOMINIO

Seconda fase IIE: Elaborazione

L'*elaborazione* è costruire il nucleo dell'architettura, risolvere gli elementi ad alto rischio, definire la maggior parte dei requisiti e stimare il piano di lavoro e le risorse complessive.

I requisiti e le iterazioni vanno organizzate 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): si riferisce alle funzioni che il cliente considera di elevato valore di business

Seconda Fase AeP: Modello di Dominio

Un *modello di dominio* è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale, nonché delle relazioni tra di essi, in un dominio di interesse. Si può definire anche come un "dizionario visuale". Serve per:

- Visualizzare il dominio del problema
- Ci aiuta a definire il sistema software

Per creare un modello di dominio bisogna:

- Trovare le classi concettuali attraverso 3 strategie: Riusare o modificare dei modelli esistenti, utilizzare un elenco di categorie comuni, identificare nomi e locuzioni nominali
- Disegnarle come classi in un diagramma delle classi UML
- Aggiungere associazioni e attributi

Un *modello dei dati* mostra i dati che devono essere memorizzati in modo persistente.

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

- *simbolo*: una parola o un'immagine usata per rappresentare la classe concettuale
- *intenzione*: la definizione (linguaggio naturale) della classe concettuale
- *estensione*: l'insieme degli oggetti descritti dalla classe concettuale

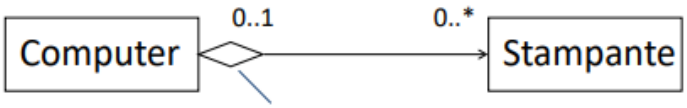
Modelli di Dominio in UML

In UML:

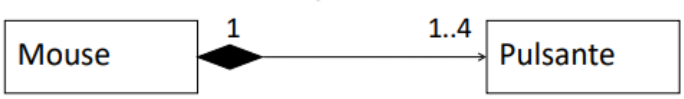
- Una *classe* è il descrittore per un insieme di oggetti che possiedono le stesse caratteristiche.
- Un'*associazione* è la relazione tra due o più classificatori che comporta connessioni tra le rispettive istanze. Un'associazione può anche essere multipla o riflessiva.
- Un *attributo* è la descrizione di una proprietà in una classe.
- Ciascuna estremità di una associazione è chiamata *ruolo.*
- La *molteplicità* di un ruolo indica quante istanze di una classe possono essere associate a una istanza dell'altra classe.

Come tipi di relazioni, oltre la classica, abbiamo:

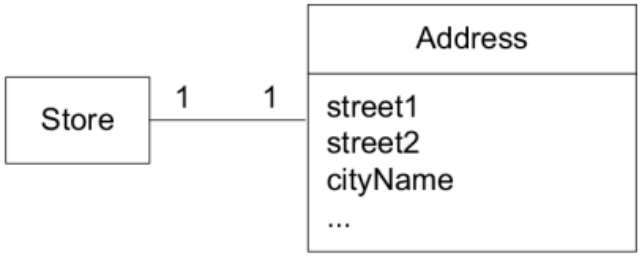
- Aggregazione: una associazione che rappresenta una relazione intero - parte. Le due entità si dicono anche debolmente collegate.



- Composizione: una forma forte di aggregazione in cui una parte appartiene a un composto alla volta, ciascuna parte appartiene sempre a un composto e questo è responsabile della creazione e cancellazione delle sue parti. Le due entità si dicono anche fortemente collegate.



Il tipo degli attributi deve essere un tipo di dato. In UML, un *tipo di dato* è un insieme di valori in cui l'identità univoca non è significativa.



Non è opportuno utilizzare gli attributi come chiavi esterne.

DIAGRAMMA DI SEQUENZA DI SISTEMA (SSD) E CONTRATTI

Diagramma di Sequenza di Sistema

Un *diagramma di sequenza di sistema* (SSD) mostra gli eventi di input e output dei sistemi in discussione. Gli SSD sono parte del Modello dei casi d'uso.

Possiamo anche dire che è un diagramma che mostra, per un particolare scenario di un caso d'uso, gli eventi generati dagli attori esterni al sistema, il loro ordine e gli eventi inter-sistema. Per comunicare si scambiano i contratti (operazioni di sistema).

Gli elementi mostri negli SSD sono concisi. Il Glossario può descrivere operazioni, parametri, valori restituiti in maggior dettaglio.

Eventi e Operazioni di Sistema

Un attore genera degli *eventi di sistema* per richiedere l'esecuzione di alcune operazioni sistema. Un sistema software reagisce a tre cose:

- eventi esterni da parte di attori (umani o sistemi informatici)

- eventi temporali
- Guasti o eccezioni (spesso provenienti da sorgenti esterne)

Una operazione di sistema è una operazione pubblica del sistema

- l'esecuzione è richiesta da un evento di sistema
- l'esecuzione di una operazione di sistema cambia lo stato del sistema:rappresentato dallo stato degli oggetti nel modello del domino
- interfaccia del sistema: l'insieme di tutte operazioni di sistema

Eventi di sistema e operazioni corrispondenti devono essere espressi a un livello astratto, intenzione, anziché azione. Devono iniziare il nome con un verbo.

Contratti

I *contratti* delle operazioni di sistema permettono di descrivere il comportamento del sistema in modo più dettagliato: usano pre-condizioni e post-condizioni per descrivere i cambiamenti agli oggetti in un modello di dominio.

Un contratto:

- è relativo a un'operazione di sistema
- descrive il cambiamento dello stato di oggetti del Modello del dominio causato dall'esecuzione dell'operazione di sistema

Post-condizioni: descrivono cambiamenti nello stato degli oggetti del Modello di dominio come:

- creazione o cancellazione di oggetto (istanza di una classe)
- cambiamento di valore di attributo
- formazione o rottura di collegamento (istanza di un'associazione)

Le post-condizioni vanno espresse con un verbo al passato. Possono essere chiamate anche Garanzia di Successo.

Il sistema e i suoi oggetti sono mostrati sul palcoscenico di un teatro:

- Prima dell'operazione, si fa una foto al palcoscenico

- Si chiude il sipario sul palcoscenico e si applica l'operazione di sistema (rumore sul palco)
- Si apre il sipario e si fa una seconda foto
- Si confrontano le foto prima e dopo: le post-condizioni vanno espresse come cambiamenti di stato sul palcoscenico

Le *precondizioni* descrivono ipotesi significative sullo stato del sistema prima dell'esecuzione all'operazione (descrizione sintetica dello stato di avanzamento del caso d'uso)

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.

Come creare e scrivere i contratti:

- Identificare le operazioni di sistema dagli SSD
- Creare i contratti per le operazioni più complesse, o che non sono chiare dai caso d'uso
- Per descrivere le post-condizioni utilizzare le seguenti categorie

Operazione, Metodo e Firma e Vincoli in UML

- un' *operazione* è la specifica di una trasformazione o interrogazione che un oggetto può essere chiamato ad eseguire
- un metodo è l'implementazione di una operazione
- una operazione ha una firma (signature) ed è associata a un insieme di vincoli (Constraint) classificati come pre e post condizioni che specificano la semantica dell'operazione.

CAPITOLO 4: PROGETTAZIONE ORIENTATA AGLI OGGETTI

Dai requisiti alla progettazione e architettura logica

L'*architettura logica* di un sistema software è l'organizzazione su larga scala delle classi software in package. Uno stile comune per l'architettura logica è l'*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. Ci sono due tipi di architettura:

- In un'architettura strati *stretta*, uno strato può solo richiamare i servizi dello strato immediatamente sottostante
- In un'architettura a strati *rilassata* uno strato più alto può richiamare i servizi di strati più bassi di diversi livelli

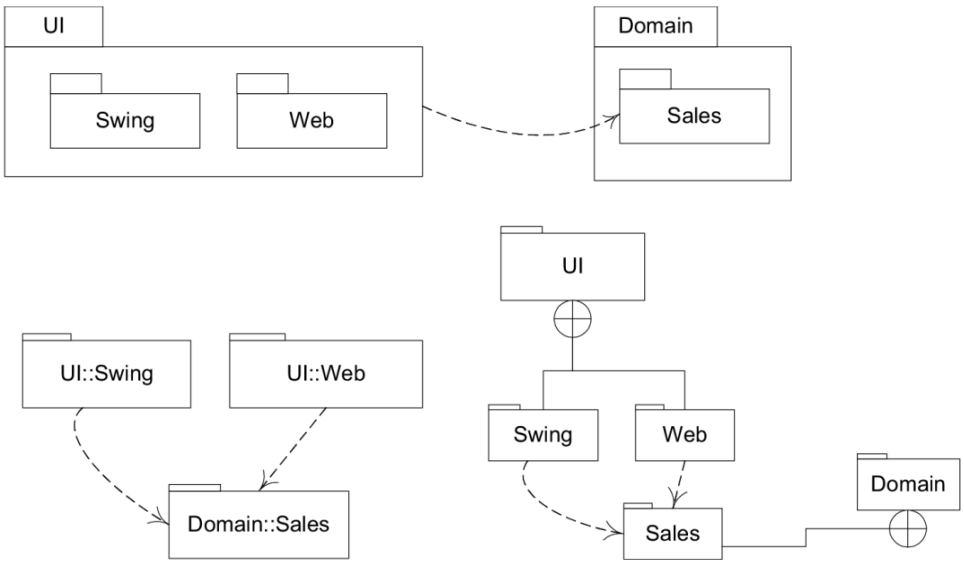
Osservazione: Il corso si concentra sullo strato della logica applicativa o strato del dominio.

L'*architettura software* è l'insieme delle strutture necessarie per ragionare su un sistema software e la disciplina di creazione di tali strutture e sistemi. L'architettura software ha a che fare con la larga scala.

L'architettura logica può essere illustrata mediante un diagramma dei package di UML:

- Uno strato può essere modellato come un *package* UML
- È molto comune l'*annidamento* di package
- Per mostrare la dipendenza tra i package viene utilizzata una *dipendenza* UML
- Un package UML rappresenta un *namespace* cosicché è possibile definire due classi con lo stesso nome su package diversi

Abbiamo 3 modalità per visualizzare i package:



Gli strati inferiori sono servizi generali e di basso livello, gli strati superiori sono più specifici per l'applicazione. La scelta comune di strati è:

- Presentazione: finestre GUI, interfaccia vocale, JS
- Applicazione: workflow, stato sessione, transizioni tra pagine
- Dominio: implementazione regole di dominio
- Infrastrutture business: servizi aziendali generali
- Servizi Tecnici: persistenza, sicurezza
- Fondamenta: strutture dati, thread, I/O di rete

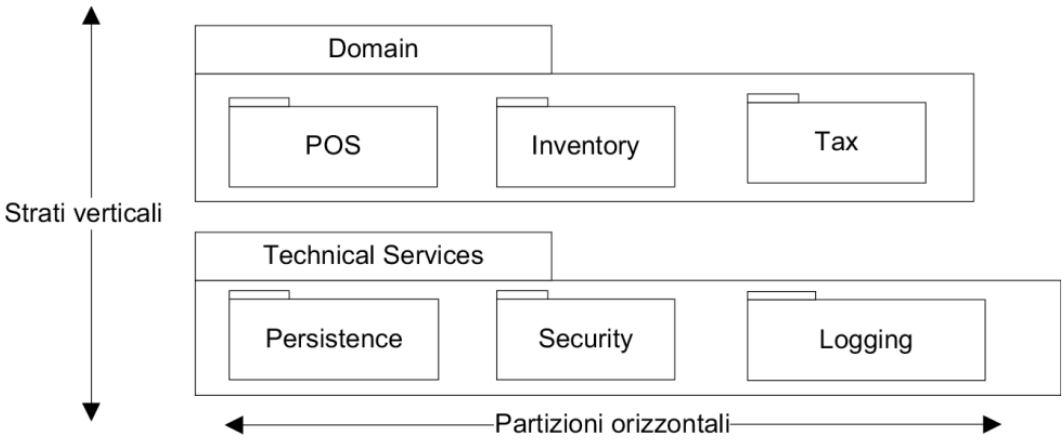
I vantaggi dell'uso a strati sono:

- C'è una *separazione* degli interessi
- Alcuni strati possono essere *sostituiti* da nuove implementazioni
- Lo *sviluppo* da parte dei team è favorito dalla segmentazione logica

Dobbiamo cercare di creare oggetti software con nomi e informazioni simili al dominio del mondo reale e assegnare a essi responsabilità della logica applicativa: questi sono chiamati *oggetto di dominio*.

Progettando gli oggetti in questo modo si arriva a uno strato della logica applicativa che può essere chiamato, in modo più preciso, strato del *dominio dell'architettura*.

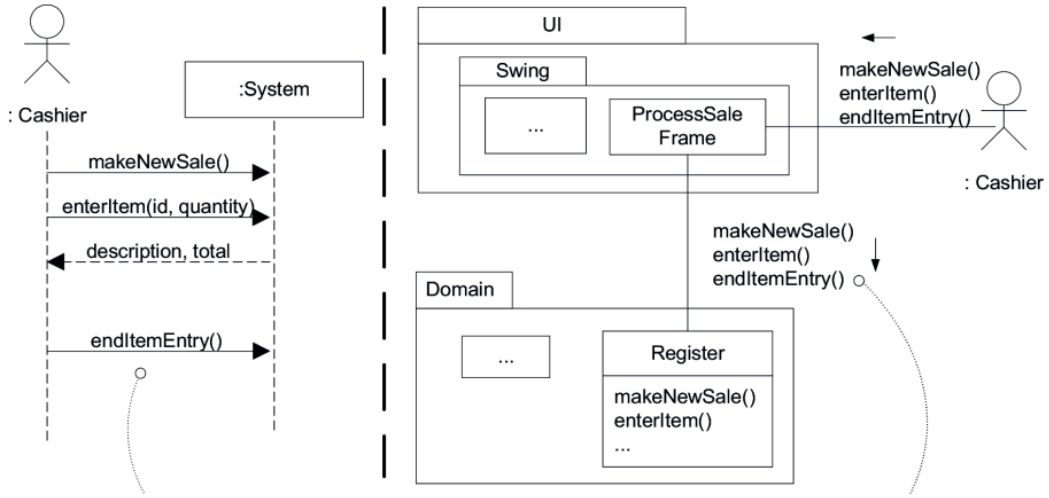
- *Livello (tier)*: solitamente indica un nodo fisico di elaborazione
- *Strato (layer)*: una sezione verticale dell'architettura
- *Partizione (partition)*: una divisione orizzontale di sottosistema di uno strato



Pattern Observer: Gli oggetti del dominio inviano messaggi a oggetti della UI, visti però solo indirettamente:

- Non conosce la sua classe UI concreta
- Sa solo che l'oggetto implementa l'interfaccia (non UI) Property Listener

I messaggio inviati dallo strato UI allo strato del dominio sono i messaggio mostrati negli SSD, come enterIte



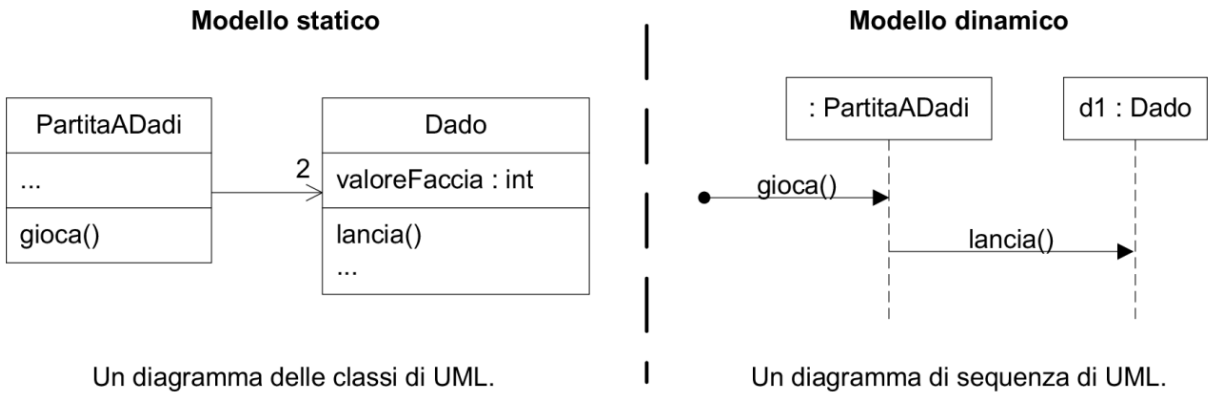
Progettazione a oggetti:

1. *Codifica*. Progettare mentre si codifica (in Java, C#, ..), possibilmente insieme a TDD e refactoring
2. *Disegno, poi codifica*. Disegnare alcuni diagrammi UML su una lavagna o con uno strumento CASE per UML per poi passare al punto 1 usando un IDE "testuale".
3. *Solo disegno*. In qualche modo, lo strumento genera ogni cosa dai diagrammi.

Per un'iterazione di 3 settimane:

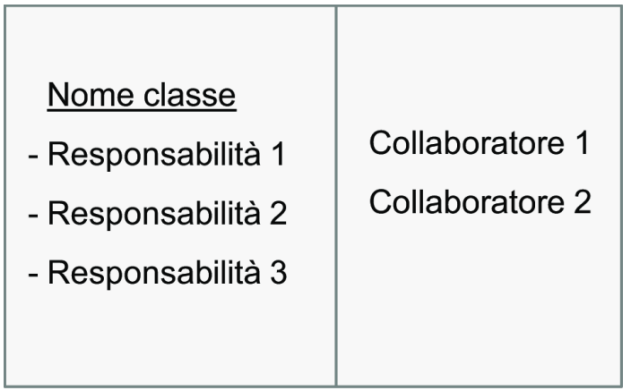
- Si dedicano alcune ore o al massimo un giorno al disegno di UML
- Si passi alla codifica per il resto dell'iterazione
- Nel corso dell'iterazione possono esserci altre sessioni di disegno più brevi
- Prima di ogni sessione di modellazione successiva, si esegua il reverse engineering e si faccia riferimento a questi diagrammi nel corso della sessione successiva

Due tipi di diagrammi UMl da portare avanti in parallelo:



Un diagramma delle classi di UML. Un diagramma di sequenza di UML.

Schede CRC: Class, Responsibility, Collaboration.



Terza Fase AeP: Diagrammi di Interazione

Un *oggetto* è un pacchetto coeso di dati e funzioni incapsulate in una unità riusabile. Oggetti incapsulano i dati. 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 dato istante
- *Comportamento*: l'insieme delle operazioni che l'oggetto può eseguire

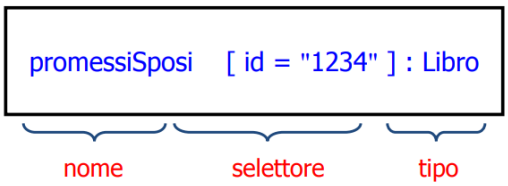
Incapsulamento o data hiding: I dati sono nascosti all'interno dell'oggetto, l'unico modo per accedere ai dati è attraverso le operazioni.

Un'*interazione* è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto

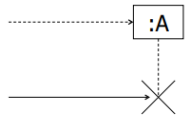
In UML i *diagrammi di interazione* illustrano il modo in cui gli oggetti interagiscono attraverso lo scambio di messaggi. Sono utilizzati per la modellazione dinamici degli oggetti. Catturano un interazione come *linee di vita* o *messaggi*.

Una linea di vita rappresenta un singolo partecipante a un'interazione. Hanno:

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



I messaggi sono la comunicazione tra due linee vita e possono essere:

Sintassi	Nome	Semantica
	Messaggio sincrono	Il mittente aspetta che il destinatario ritorni dall'esecuzione del messaggio
	Messaggio asincrono	Il mittente invia il messaggio e continua l'esecuzione: non aspetta un Ritorno dal destinatario
	Messaggio di ritorno	Il destinatario di unmessaggio precedente restituisce il focus di Controllo al mittente di quel messaggio
	Creazione dell'oggetto Distruzione dell'oggetto	Il mittente crea un istanza del classificatore specificato dal destinatario Il mittente distrugge il destinatario. Se la sua linea di vita ha una coda, questa termina con una X
	Messaggio trovato	il messaggio viene inviato dal di fuori dell'ambito dell'interazione. Si vuole mostrare la ricezione del messaggio ma non la provenienza.
	Messaggio prerso	Il messaggio non raggiunge mai la sua destinazione. Si può usare per indicare condizioni di errore in cui i messaggi vanno persi.

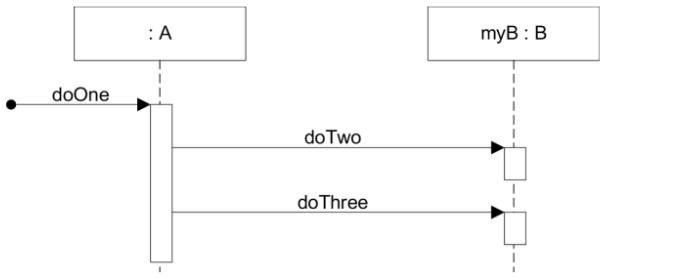
Diagrammi di interazione: abbiamo diversi tipi:

- Diagrammi di *sequenza*: enfatizzano la sequenza temporale degli scambi di messaggi, mostrano interazioni ordinate in una sequenza temporale, non mostrano le relazioni degli oggetti
- Diagrammi di *comunicazione*: enfatizzano le relazioni strutturali tra gli oggetti
- Diagrammi di *interazione generale*: illustrano come il comportamento complesso è realizzato da una serie di interazioni più semplici
- Diagrammi di *temporizzazione*: enfatizzano gli aspetti in tempo reale di una interazione

Tipo	Punti di forza	Punti di debolezza
Diagramma di sequenza	Mostra chiaramente la sequenza o l'ordinamentotemporale dei messaggi. Opzioni di notazione numerose e dettagliate.	Costringe a estendersi verso destra quando si aggiungono nuovi oggetti; consuma lo spazio orizzontale.
Diagramma di comunicazione	Economizza lo spazio; flessibilità nell'aggiunta di nuovi oggetti nelle due dimensioni.	Più difficile vedere la sequenza dei messaggi. Meno opzioni di notazione.

Una classe interfaccia mostra solo i metodi senza implementazioni. Una classe astratta può contenere dei metodi anche implementati ma non può essere istanziata. Servono classi concrete che estendono la classe astratta con metodi che estendono i metodi della classe astratta.

Esempio di codice associato a un diagramma:



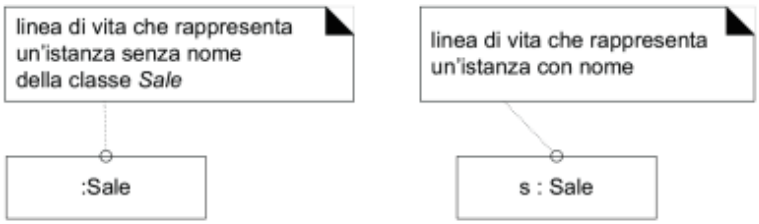

```
public class A {
private B myB = new B();
public void doOne() {
myB.doTwo();
myB.doThree();
}
...
}
```

```
public class B {
public void doTwo() {
...
}
public void doThree() {
...
}
...
}
```

Diagrammi di Sequenza

Notazione comune dei diagrammi di interazione:

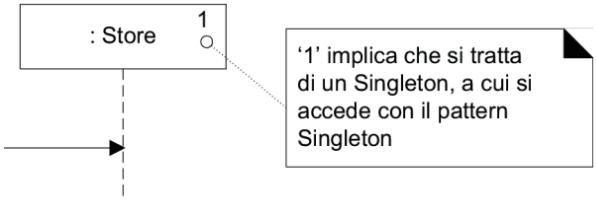
- *Linee di vita*:



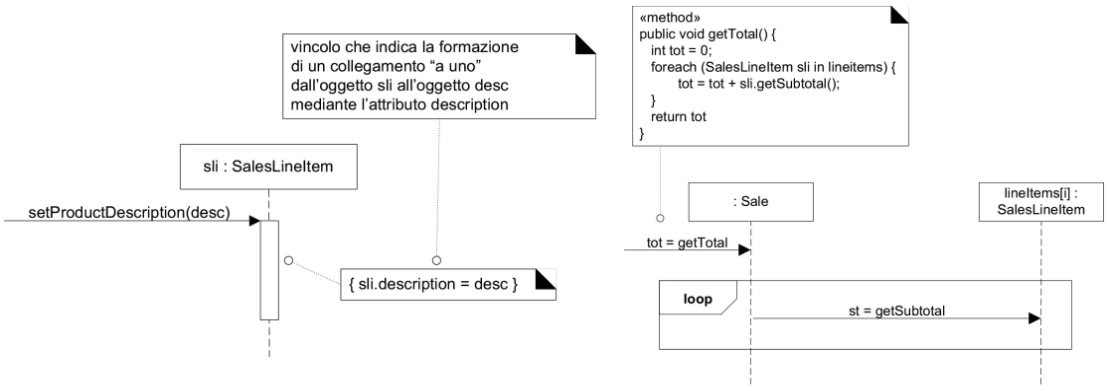
- Espressioni *messaggio*: UML ha una sintassi standard per queste operazioni:

```
return = message(parameter : parameterType) : returnType
```

- Oggetto *Singleton*: Oggetto che può essere istanziato solo una volta.

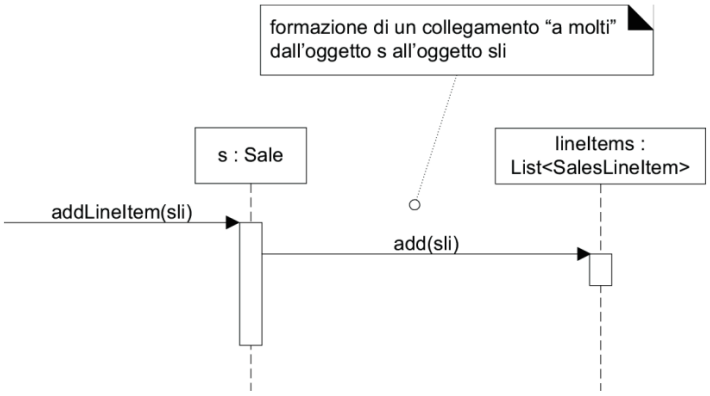


- *Note*: Possiamo scrivere delle note su un vincolo o sul corpo di un metodo:

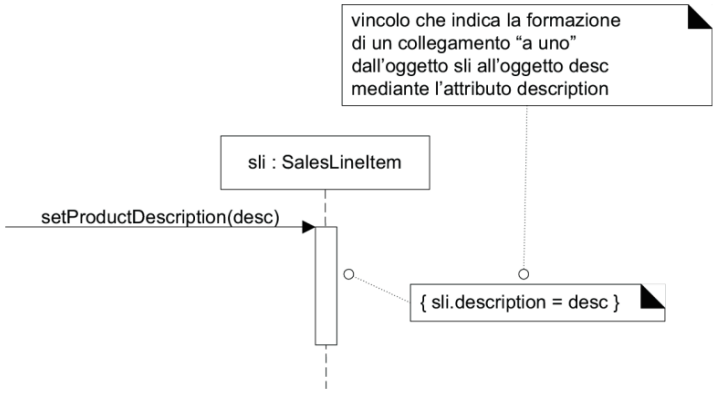


Vincolo **Corpo di un metodo**

- *Associazione "a molti":*

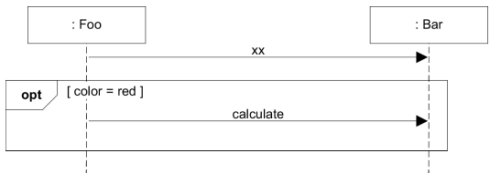
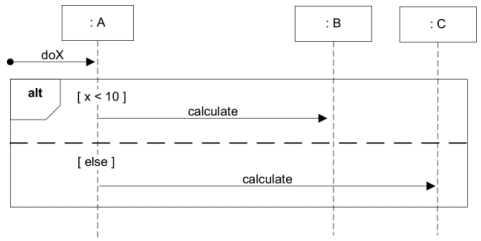
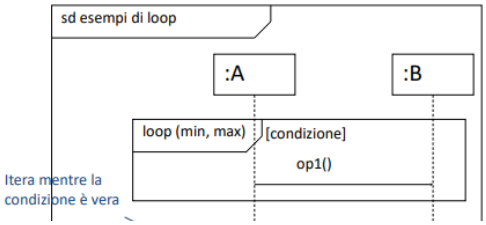
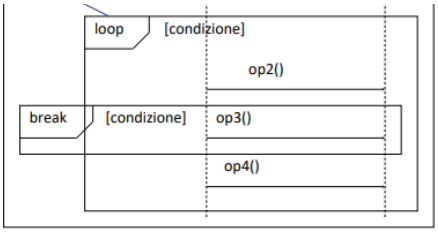


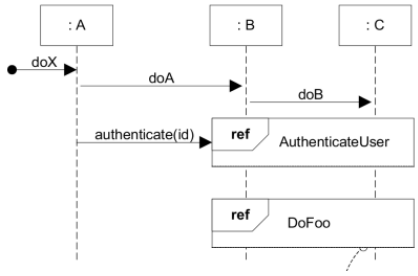
- *Associazione a uno*:



Nei diagrammi di sequenza, un *operatore* di interazione definisce la semantica di un frammento combinato e determina come utilizzare gli operandi di interazione in tale frammento. Ce ne sono di diversi tipi:

Operatore	Nome Completo	Significato	Pseudocodice	Esempio

opt	Option	un operando viene eseguito se la condizione è vera	If … then	
alt	Alternatives	viene eseguito l'operando la cui condizione è vera	Select … case	
loop	Loop	Esegue il loop un minimo di volte, poi mentre la condizione è vera, esegue altre iterazioni fino al massimo specificato.	do … while	
break	Break	Se la condizione è vera, si esegue solo l'operando e non il resto dell'interazione in cui è incluso	break	

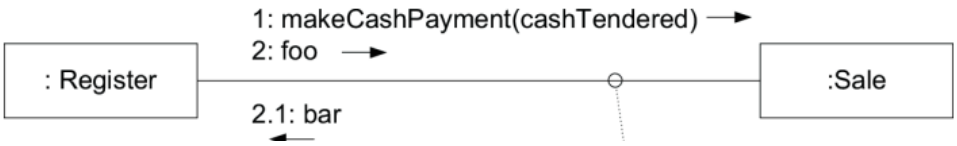
ref	Reference	Il frammento combinato fa riferimento ad un'altra iterazione		

Idiomi più comuni:

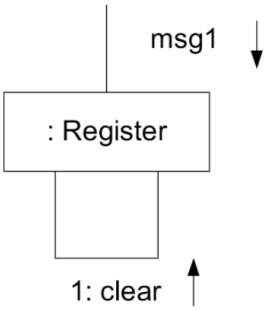
- loop * → continuare il ciclo sempre
- loop n → ripetere n volte
- loop [espressioneBooleana] → ripetere fino a che espressioneBooleana è vera
- loop 1, * espressioneBooleana → eseguire 1 volta e poi fino a che espressione N è vera
- loop[collezione] → foreach di collezione
- loop[classe] → foreach di classe

Diagramma di comunicazione

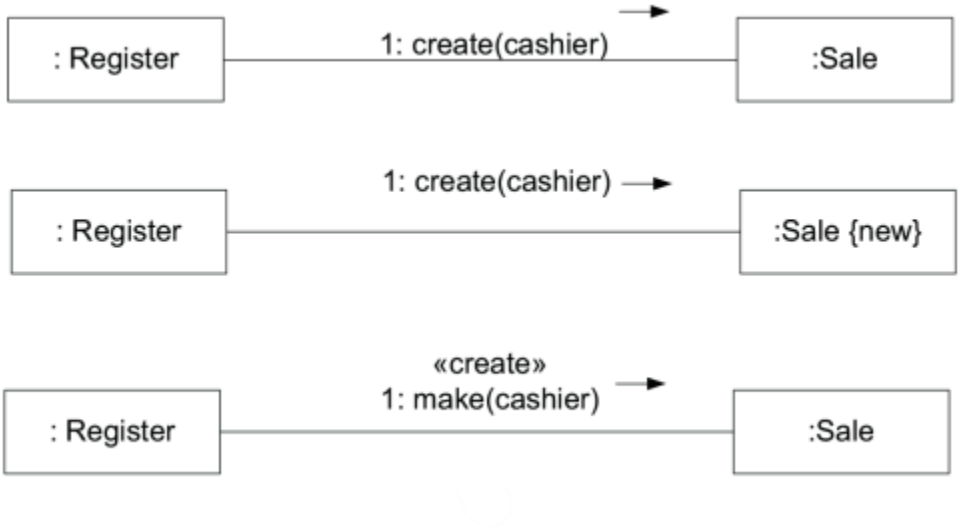
Un *collegamento* (link) è un percorso di connessione tra due oggetti. Non vanno numerati i messaggi iniziali in modo che la numerazione complessiva sia più comprensibile.



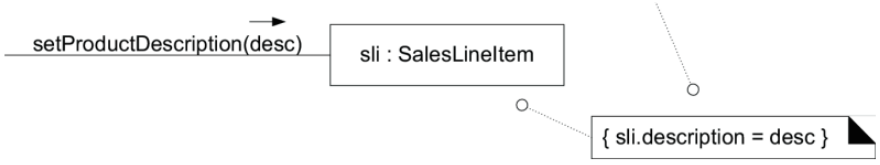
L'auto delega è un messaggio con collegamento a se stesso.



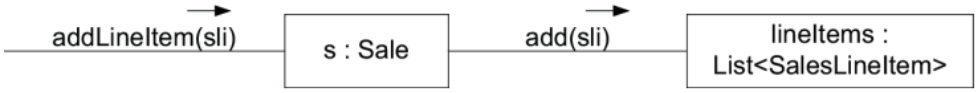
Creazione di istanze: Ci sono 3 modi per creare delle istanze:



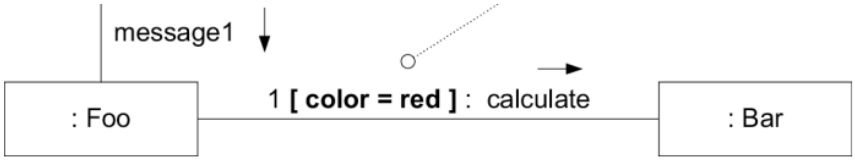
Formazione di un collegamento di un'associazione "a uno"

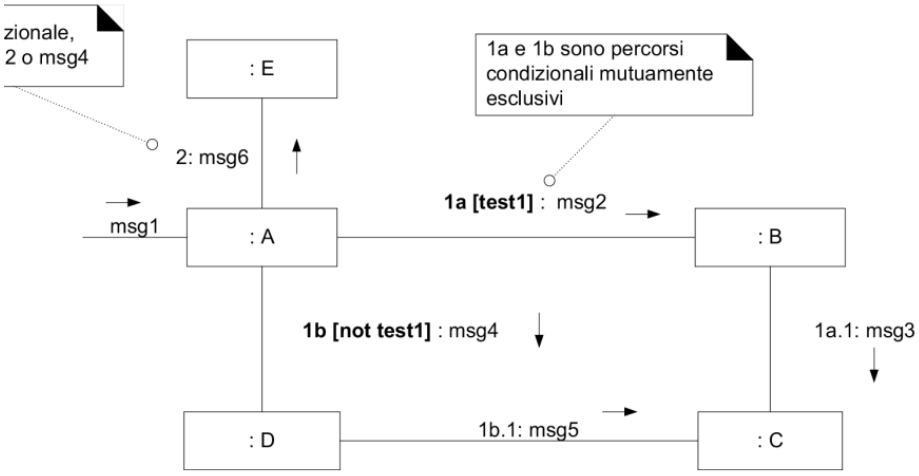


Formazione di un collegamento di un'associazione "a molti"



Messaggi condizionali (opt) e mutualmente esclusivi (alt)





L'iterazione è mostrata dall'utilizzo dell'indicatore di iterazione (*) e un'espressione di iterazione

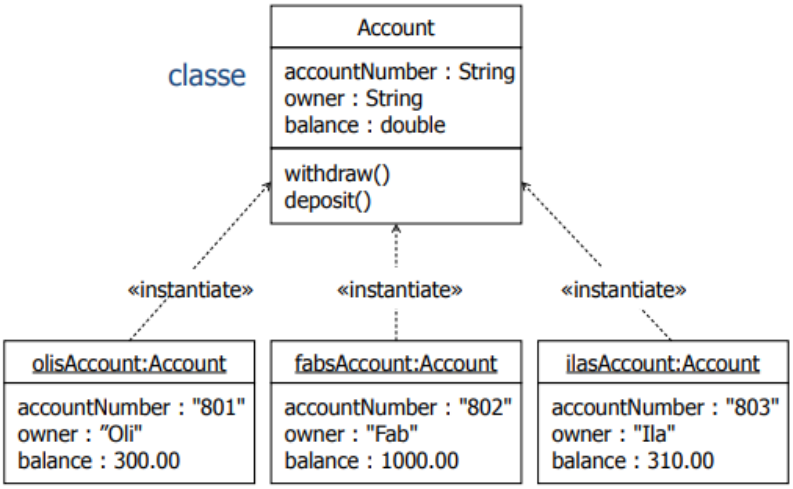
Quarta Fase AeP: Diagrammi delle Classi

Un *oggetto* è un pacchetto coeso di dati e funzioni incapsulate in una unità riusabile: hanno valori degli attributi e operazioni. Hanno inoltre:

- Identità: ogni oggetto ha il suo identificativo univoco

- Stato: viene stabilito dai valori effettivi dei dati memorizzati in un oggetto ad un dato istante
- Comportamento: l'insieme delle operazioni che l'oggetto può eseguire

Una *classe* modella un insieme di oggetti omogenei (istanze) ai quali sono associate proprietà statiche (attributi) e dinamiche (operazioni). Tra un oggetto che è istanza di una classe C e la classe C si traccia un arco *Instantiate*



Un classificatore è un "elemento di modello che descrive caratteristiche comportamentali e strutturali": possono essere classi ed interfacce. Le proprietà strutturali di un classificatore comprendono:

- attributi
- estremità di associazioni

Un *attributo* modella una proprietà locale della classe ed è caratterizzato da un nome e dal tipo dei valori associati. Ha una visibilità, un nome, un tipo, una molteplicità e un valore iniziale. Solo il nome è obbligatorio. Possiamo avere, oltre gli attributi di classi, degli attributi di associazioni e collegamenti

Visibilità: Essendo in un ambiente con incapsulamento, di solito hanno visibilità privata, ovvero solo le operazioni della classe possono accedere ai membri. Se non si specifica, sarà automaticamente privata.

Molteplicità: Se le molteplicità di un attributo non vengono indicate, si associa valore uno. come detto prima). Al contrario, se un attributo B di tipo T di una classe C ha molteplicità x..y, allora B associa ad ogni istanza di C al minimo x e al massimo y valori di tipo T.

Associazione: Un'associazione tra una classe C1 ed una classe C2 modella una relazione matematica tra l'insieme delle istanze di C1 e l'insieme delle istanze di C2:

- Alcune volte è interessante specificare un verso per il nome dell'associazione
- È anche possibile assegnare due nomi con i relativi versi alla stessa associazione.
- È possibile aggiungere alla associazione una informazione che specifica il ruolo che una classe gioca nella associazione
- Può avvenire tra più classi

La *navigabilità* indica che è possibile spostarsi da un qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione

La *molteplicità* limita il numero di oggetti di una classe che possono partecipare in una relazione in un dato istante

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 precondizioni e post condizioni:

visibility name(parameter-list) : return-type {property-string}

In UML, un *metodo* è l'implementazione di un'operazione:

- L'operazione *create* corrisponde a un costruttore
- Get e Set non vengono spessi mostrati per il rapporto disturbo-valore che generano

Una parola *chiave* in UML è un decoratore testuale per classificare un elemento di un modello:

- actor
- interface
- {abstract}
- {ordered}

I profili si usano per personalizzare UML per un uso specifico. Insieme di stereotipi, tag e vincoli:

- *Vincoli*: Estendono la semantica di un elemento consentendo di aggiungere nuove regole
- *Stereotipi*: Definiscono un nuovo elemento di modellazione UML basandosi su un esistente
- *Tag*: Permettono di estendere la definizione di un elemento tramite l'aggiunta di nuove informazioni specifiche

Una *proprietà* è un valore con un nome, che denota una caratteristica di un elemento

Generalizzazione : la relazione is-a. ogni istanza di ciascuna sottoclasse è anche istanza della superclasse

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

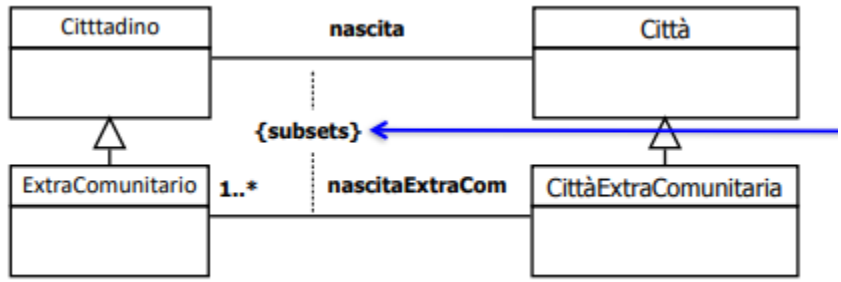
La *superclasse* però può anche generalizzare diverse sottoclassi rispetto ad un unico criterio. In questo caso la freccia è unita

i *classificatori* specifici non hanno istanze comuni: {*disjoint*}. Altrimenti si dice {*overlapping*}

l'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse: {*complete*} altrimenti {*incomplete*}

Specializzazione *attributo*: In una generalizzazione la sottoclasse può anche specializzare le proprietà ereditate dalla superclasse.

Specializzazione *associazione*: si indica con {subsets}



Specializzazione *operazioni*: Un'operazione si specializza specializzando i parametri e/o il tipo di ritorno.

Una classe con una o più operazioni *astratte* non può essere istanziata. Si scrive la classe in corsivo.

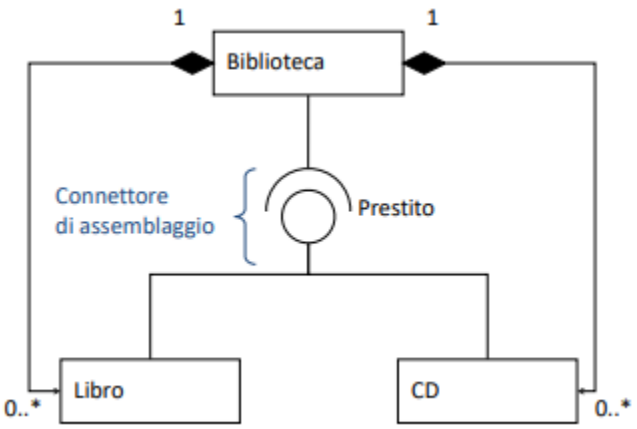
Un'operazione *polimorfica* ha molte implementazioni.

Una *dipendenza* è una relazione di dipendenza:

- un elemento cliente è a conoscenza di un elemento fornitore
- Un elemento è accoppiato con un altro elemento o dipende da un altro elemento

Un'interfaccia è un insieme di funzionalità pubbliche identificate da un nome. Separa le specifiche di una funzionalità dall'implementazione

Si possono connettere interfacce richieste e fornite utilizzando il *connettore di assemblaggio*:



Una aggregazione è una associazione che rappresenta una relazione intero-parte

Una composizione 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

L'*ereditarietà* è la forma di interdipendenza più forte tra classe, non è possibile cambiare la classe di un oggetto a tempo di esecuzione

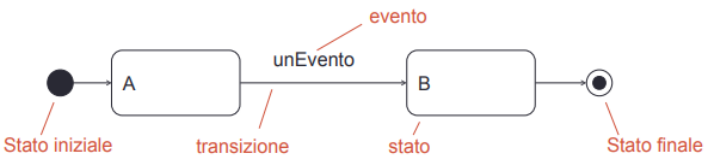
L'ereditarietà definisce interfaccia ed implementazione, l'interfaccia solo l'interfaccia. Utilizzare l'ereditarietà quando si vuole ereditare l'implementazione, utilizzare la realizzazione di Interfaccia quando si vuole definire un contratto

---

Capitolo 5: Diagrammi di Attività e Diagrammi di Macchina a Stati

Macchine 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.



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. Ogni stato è composto da:

1. Nome dello stato
2. Azioni di ingresso e di uscita: istantanee e non interrompibili

3. Transizioni interne: non causano la transizione in un nuovo stato
4. Attività interna: richiedono un intervallo di tempo finito e sono interrompibili

Come transazioni possiamo avere:

- lo pseudo-stato di *giunzione* che può essere semplice oppure con fusione e ramificazione
- lo pseudo-stato di *selezione* dalla quale si diramano più scelte.

Eventi: attivano le transizioni nelle macchine a stati. Sono di diversi tipi:

- *chiamata*: evento dovrebbe avere la stessa firma di un'operazione della classe di contesto
- *segnale*: pacchetto di informazioni inviato in modo asincrono tra oggetto.
- *variazione*: espressione booleana, passaggio da falso a vero.
- *temporali*: occorrono quando un'espressione di tempo diventa vera. Keywords: dopo, quando.

Gli *stati compositi* hanno una o più regioni ognuna delle quali contiene una sotto-macchina annidata. Possono essere semplici, con una regione, o ortogonali quando ne hanno due o più. Quando si entra nello stato composito, tutte le sotto-macchine iniziano la loro esecuzione in modo concorrente

Se vogliamo fare riferimento ad una macchina a stati in altre macchine a stati, senza ingombrare i diagrammi, dobbiamo usare gli *stati della sotto-macchina*, semanticamente equivalenti agli stati compositi.

La *comunicazione asincrona* è ottenuta da una sotto-macchina configurando un flag per un'altra sottomacchina

Lo stato di *sync* è uno stato speciale il cui compito è quello di tenere traccia di ogni singola attivazione della sua unica transizione di input. Funziona come una coda: si aggiunge un elemento alla coda ogni volta che viene attivata la transizione di input.

1. numero limitato di elementi specificando all'interno dello stato di sync un numero intero positivo
2. Asterisco per un numero illimitato di elementi

Lo *pseudo-stato con memoria semplice* ricorda in quale sottostato si era quando si è lasciato il super-stato. Si indica con H

Uno *stato con memoria multilivello* ricorda l'ultimo sottostato attivo di questo sottostato attivo. Si indica con H*

Diagramma dell'attività

Consentono di modellare un processo come un'attività costituita da un insieme di nodi connessi da archi.

Le attività sono reti di nodi connessi ad archi. Esistono tre categorie di nodi

- Nodi *azione*: rappresentano unità discrete di lavoro atomiche 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. Esistono due categorie di archi:

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

Le attività spesso iniziano con un singolo nodo iniziale. Le attività possono avere pre e post condizioni. Quando un nodo azione finisce, esso emette un *token* che potrebbe attraversare un arco per dare inizio alla prossima azione.

L'esecuzione dei nodi azione:

- Esiste un token simultaneamente su ciascun arco entrante
- I token in ingresso soddisfano tutte le pre condizioni locali del nodo azione
- Eseguono un AND logico sui loro token di entrata
- Eseguono una fork implicita su tutti i suoi archi uscenti quando l'esecuzione è terminata

Ci sono diversi tipi di nodi azione:

- Nodo azione di *chiamata*: invoca un'attività, un comportamento o un operazione. Il più comune. Si indica con rettangolo.
- *Invia segnale di azione*: invia un segnale in modo asincrono. Pentagono convesso.
- *Accetta un evento*: aspetta gli eventi individuati dal suo oggetto proprietario ed emette l'evento sull'arco in uscita. Pentagono concavo.
- *Accetta un evento temporale*: risponde al tempo. Doppio triangolo.

Nodi *controllo*: controllano il flusso attraverso l'attività. Oltre al nodo iniziale, al nodo finale di attività e nodo finale del flusso abbiamo:

- Nodo *decisione* è un nodo controllo che ha un arco entrante e due o più archi alternativi uscenti. Ogni arco uscente è protetto da una condizione di guardia mutuamente esclusiva.
- Nodo *fusione* accetta uno dei diversi flussi alternativi: ha due o più archi in ingresso e esattamente un arco in uscita.
- Nodo *biforcazione* modellano flussi concorrenti: I token che arrivano all'arco in ingresso vengono duplicati e offerti simultaneamente su tutti gli archi in uscita
- Nodo *ricongiunzione* sincronizzano due o più flussi concorrenti: hanno più archi in ingresso e un solo arco in uscita. Offrono un token sul loro arco in uscita quando c'è un token su tutti i loro archi in ingresso

I nodi oggetti indicano che sono disponibili istanze di un particolare classificatore:

- Ogni nodo oggetto può tenere un numero infinito di token oggetto
- Token sono offerti agli archi in uscita secondo FIFO o LIFO
- Ha un criterio di selezione ed uno stato

I nodi oggetto possono essere parametri di input e output per le attività.

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

CAPITOLO 6: GRASP: PROGETTAZIONE DI OGGETTI CON RESPONSABILITÀ

GRASP

Progettazione guidata dalle responsabilità (RDD): Pensare in termini di responsabilità, ruoli e collaborazioni.

UML definisce una responsabilità come “un contratto o un obbligo di un classificatore”

Per *responsabilità* si intende un’astrazione di ciò che fa o rappresenta un oggetto. Sono di due tipi:

- Le responsabilità di *fare* di un oggetto comprendono:
    - Fare qualcosa esso stesso, come per esempio eseguire un calcolo
    - Chiedere ad altri oggetti di eseguire azioni
    - Controllare e coordinare le attività di altri oggetti
- Le *responsabilità* di *conoscere* di un oggetto comprendono:
    - Conoscere i propri dati privati incapsulati
    - Conoscere gli oggetti correlati
    - Conoscere cose che può derivare o calcolare

Le responsabilità sono implementate per mezzo di oggetti e metodi che agiscono da soli oppure che *collaborano* con altri oggetti e metodi

Flusso di lavoro RDD:

1. Identifica le responsabilità, e considerale una alla volta
2. Chiediti a quale oggetto software assegnare questa responsabilità (potrebbe essere un oggetto già identificato o nuovo)
3. Chiediti come fa l'oggetto scelto a soddisfare questa responsabilità (da solo o collaborando)

Un pattern è una coppia problema/soluzione ben conosciuta e con un nome. Codificano soluzioni, idiomi e principi già applicati e che si sono dimostrati corrette in varie situazioni.

I pattern GRASP danno un nome e descrivono principi di base per la progettazione di oggetti e l’assegnazione di responsabilità: General, Responsibility, Assignment, Software, Patterns. Il Pattern GRASP di base è:

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

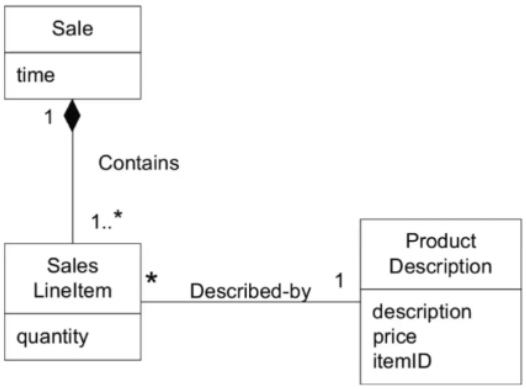
Creator

Problema: Chi crea un oggetto A?

Soluzione: Assegna alla classe B la responsabilità se:

- B contiene oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati di A

Esempio: Nel Sistema POS il Creator è Sale in quanto contiene molti oggetti SalesLineItem



Controindicazioni: se la creazione dell'oggetto è complessa, usatr altre soluzioni ad es., Abstract Factory [GoF]

Vantaggi: favorisce un accoppiamento basso, poiché gli oggetti creati sono probabilmente già visibili all'oggetto creatore

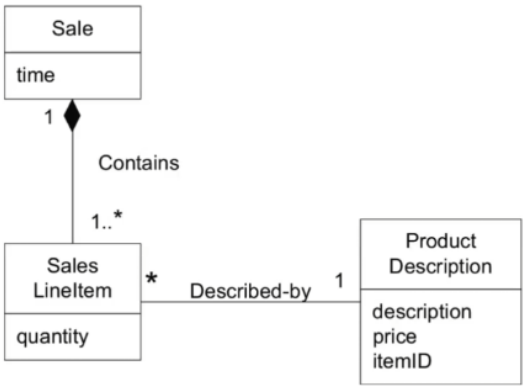
Information Expert

Problema: Qual è un principio di base per assegnare responsabilità agli oggetti?

Soluzione: Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla

Esempio: Nel Sistema POS l'Information Expert è Sale in quanto ha le conoscenze per calcolare il totale di una vendita. Dunque:

1. Aggiungo una classe sw Sale al modello di progetto
2. assegno alla classe la responsabilità di calcolare il suo totale



Controindicazioni:

- in alcuni casi problemi di accoppiamento e di coesione
- progetta per una separazione dei principali interessi del sistema

Vantaggi:

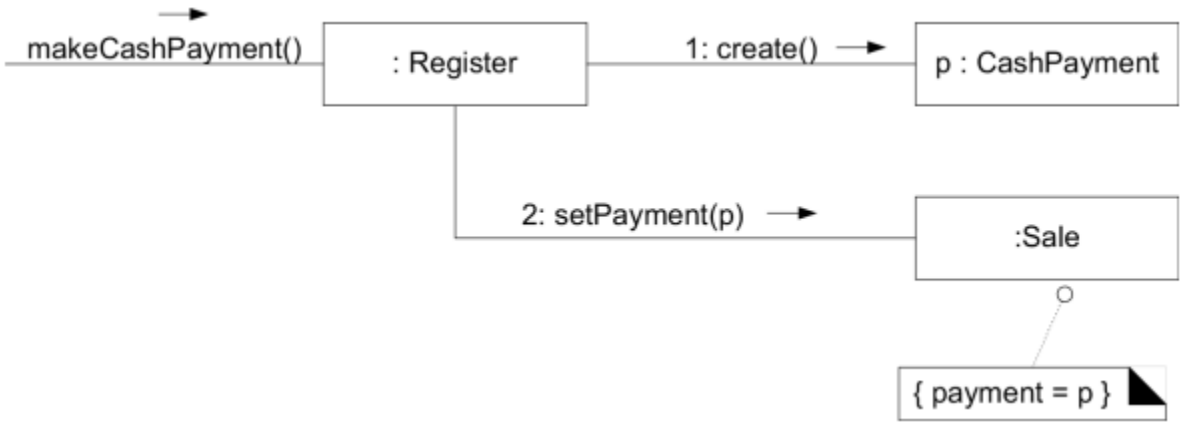
- l'incapsulamento dell'informazione viene mantenuto (low coupling)
- il comportamento è distribuito tra tutte le classi che possiedono le informazioni richieste, incoraggiando la definizione di classi più coese e leggere (high cohesion)

Low Coupling

Problema: Come ridurre l'impatto dei cambiamenti?

Soluzione: Assegna le responsabilità in modo tale che l'accoppiamento sia basso, ovvero quanto un elemento influenza altri.

Esempio: Supponiamo di dover creare un'istanza di CashPayment e di associarla alla Sale. Il responsabile sarà Register che invia un messaggio setPayment a Sale, passando il nuovo CashPayment come parametro



Controindicazioni: accoppiamento alto con elementi instabili

Vantaggi:

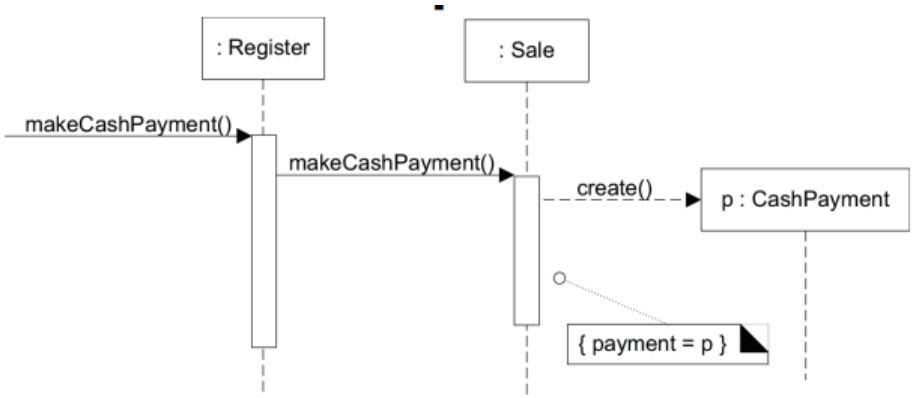
- il cambiamento di altre classi ha un impatto basso
- consente di definire classi semplici da comprendere in isolamento e convenienti da riusare

High Cohesion

Problema: Come mantenere gli oggetti comprensibili e sostenere Low Coupling?

Soluzione: Assegna le responsabilità in modo tale che la coesione rimanga alta, ovvero mantieni correlate le operazioni di un elemento sw dal punto di vista funzionale

Esempio: Register assume non solo la responsabilità di ricevere operazione sistema makeCashPayment ma anche parte della responsabilità di soddisfarla



Controindicazioni: casi in cui la coesione bassa è giustificabile

Vantaggi:

- chiarezza e facilità di comprensione
- manutenzione ed evoluzione semplificata
- spesso favorisce un accoppiamento basso
- funzionalità altamente correlate a grana fine favoriscono il riuso, perché una classe coesa può essere usata per uno scopo specifico

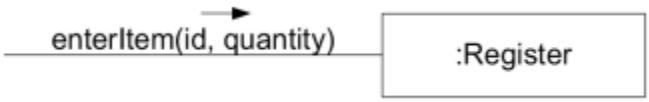
Controller

Problema: Qual è il primo oggetto oltre lo stato UI a ricevere e coordinare un'operazione di sistema?

Soluzione: Assegna le responsabilità ad un oggetto che:

- Rappresenta il sistema complessivo, una radice, un punto d'accesso.
- Rappresenta un caso d'uso all'interno del quale si verifica l'operazione di sistema

Esempio: Nel Sistema POS un buon Controller sarà Register:



Controindicazioni:

- un controller gonfio è un controller con coesione bassa che gestisce responsabilità in troppi contesti

Vantaggi:

- aumenta la possibilità di riuso dello strato del dominio e consente di definire interfacce utente "inseribili"
- Opportunità di ragionare sullo stato dei casi d'uso

Altri Pattern GRASP

Pure Fabrication:

- Problema: Quale oggetto deve avere la responsabilità quando non si vogliono violare High Cohesion e Low Coupling ma le soluzioni suggerite da Expert non sono appropriate?
- Soluzione: Soluzione: assegnare un insieme di responsabilità altamente coeso a una classe artificiale o di convenienza, che non rappresenta un concetto del dominio del problema, ma piuttosto è una classe inventata
- Vantaggi: high coesion, riuso.
- Svantaggi: si potrebbe avere un eccesso di oggetti comportamentali che non possiedono le informazioni richieste per soddisfarli

Polymorphism:

- Problema:
    - Come gestire alternative basate sul tipo: Ovvero, molti programmi usano if-then-else o switch case. Nel caso in cui si presenti una nuova variazione sarà necessario modificare la logica condizionale, spesso in molti punti.
    - Come creare componenti software inseribili: com'è possibile sostituire un componente server con un altro, senza ripercussioni sui client?
- Soluzione: utilizzando operazioni polimorfe, ovvero diamo lo stesso nome ai servizi di oggetti diversi. Ad esempio nel Monopoly un servizio "finisce sulla casella", che poi avrà significati diversi in base al tipo di casella.
- Vantaggi: Sono facili da aggiungere, permettono il riuso.

- Svantaggi: Prima di investire in una maggiore flessibilità, verificare che si possa effettivamente utilizzare più volte e ne valga la pena.

Indirection:

- Problema: dove assegnare una responsabilità, per evitare l'accoppiamento diretto tra più elementi? come disaccoppiare gli oggetti in modo da sostenere un alto riuso e un accoppiamento basso?
- Soluzione: si assegna la responsabilità ad un oggetto intermediario che media tra altri componenti o servizi, in modo che non ci sia un accoppiamento diretto tra di essi.
- Vantaggi:
    - le estensioni richieste per nuove variazione sono facili da aggiungere
    - è possibile introdurre nuove implementazione senza influire sui client
    - favorisce Low Coupling
    - è possibile ridurre l'impatto o il costo dei cambiamenti
- Soluzione: si assegna la responsabilità ad un oggetto intermediario che media tra altri componenti o servizi, in modo che non ci sia un accoppiamento diretto tra di essi.

Protected Variations:

- Problema: come progettare oggetti, sottosistemi e sistemi in modo tale che le variazioni o l'instabilità di questi elementi non abbian 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

CAPITOLO 7: DESIGN PATTERN

Ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente.

I design pattern GoF sono classificati in base al loro scopo, che può essere creazionale, strutturale o comportamentale.

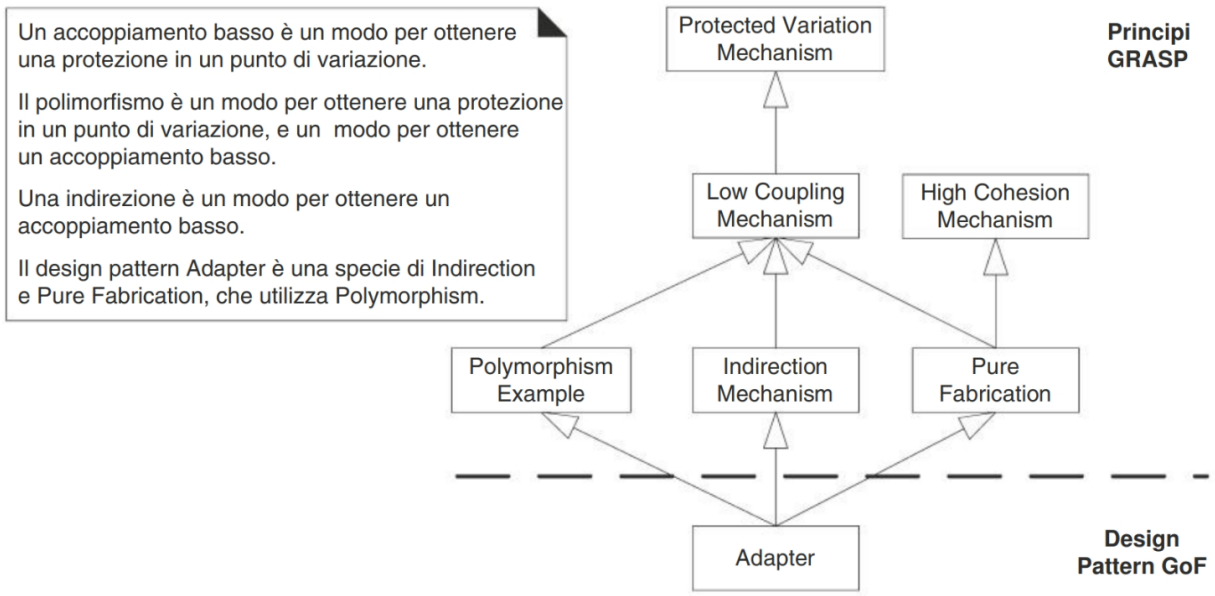
- un pattern *creazionale* riguarda la creazione di uno o più oggetti
- un pattern *strutturale* riguarda la rappresentazione di informazioni in termini della composizione di classi e oggetti
- un pattern *comportamentale* si occupa dell'interazione tra classi e oggetti per ottenere un certo comportamento

Nome: *Adapter*

Problema: Come gestire interfacce incompatibili (l'oggetto server offre servizi di interesse per l'oggetto client, ma l'oggetto client vuole fruire di questi servizi in una modalità diversa) o fornire un'interfaccia stabile a componenti simili ma con interfacce diverse?

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

La maggior parte dei design pattern può essere vista come una specializzazione di alcuni principi GRASP di base:



La scelta di un oggetto di dominio per creare gli adattatori non sostiene l'obiettivo della separazione degli interessi, e riduce la sua coesione. Un'alternativa comune in questo caso

consiste nell'applicare il design pattern creazionale Factory, in cui viene definito un oggetto "factory" Pure Fabrication per creare gli oggetti.

Nome: *Factory*

Problema: Chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica di creazione complessa, quando si desidera separare le responsabilità di creazione per una coesione migliore, e così via?

Soluzione: Crea un oggetto Pure Fabrication chiamato una Factory che gestisce la creazione.

Chi crea la factory stessa, e come vi si accede?

Nome: *Singleton*

Problema: È consentita esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

Soluzione: Definisci un metodo statico della classe che restituisce l'oggetto singleton

- Bisogna assicurare che una classe abbia una sola istanza e fornire un punto d'accesso globale a tale istanza
- Singleton definisce oggetti con una visibilità globale
- può essere non efficace nella realizzazione di applicazioni distribuite

Nome: *Strategy*

Problema: Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?

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

Gli algoritmi di cui si parla nel pattern strategy sono correlati nel senso che risolvono tutti uno stesso problema

C'è un modo per modificare il progetto in modo tale che l'oggetto Sale non sappia se si trova di fronte a una sola o più strategie di determinazione del prezzo e offrire anche un progetto per la risoluzione dei conflitti? Si, composite

Nome: *Composite*

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.

Nome: *Facade*

Problema: È richiesta un'interfaccia comune e unificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema. Può verificarsi un accoppiamento indesiderato a molti oggetti nel sottosistema, oppure l'implementazione del sottosistema può cambiare. Che cosa fare?

Soluzione: Definisci un punto di contatto singolo con il sottosistema, ovvero un oggetto facade (facciata) che copre il sottosistema. Questo oggetto facade presenta un'interfaccia singola e unificata ed è responsabile della collaborazione con i componenti del sottosistema.

Un Facade è un oggetto front end che rappresenta il *punto di entrata* singolo ai servizi di un sottosistema

CAPITOLO 8: SVILUPPO GUIDATO DAI TEST E REFACTORING

Progettare la Visibilità

Visibilità: la capacità di un oggetto di "vedere" o di avere un riferimento ad un altro oggetto. Affinché un oggetto mittente invii un messaggio a un oggetto destinatario, il destinatario deve essere visibile al mittente.

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 permanente persiste finché A e B esistono.
- Visibilità per parametro: B è un parametro di un metodo di A, visibilità relativamente temporanea, perché persiste solo nell'ambito del metodo
- Visibilità locale: B è un oggetto locale (non parametro) di un metodo di A, a visibilità relativamente temporanea perché persiste solo nell'ambito del metodo.
- Visibilità globale: B è in qualche modo visibile globalmente, visibilità relativamente temporanea perché persiste solo nell'ambito del metodo

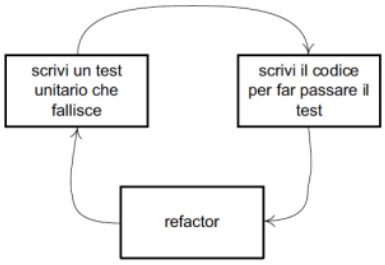
Sviluppo guidato dai test

Idea: il test è scritto prima del codice da testare, immaginando che il codice da testare sia stato già scritto. Poi si scrive il codice per far passare il test. Consideriamo solo il test unitario (ovvero i test relativi a singole classi e metodi

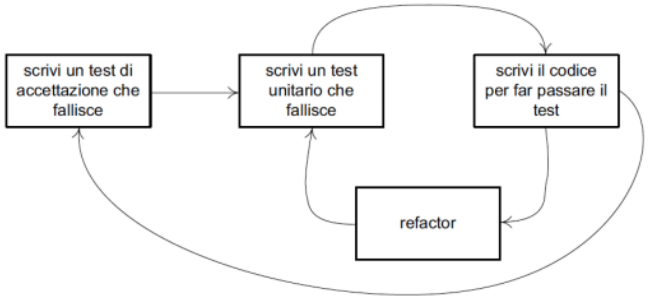
Schema dei metodi di test di unità:

- Preparazione: crea l'oggetto da verificare (detto anche fixture)
- Esecuzione: 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

Cicli:



Il ciclo di base del TDD per i test unitari.



Il doppio ciclo del TDD.

Linee guida:

- non scrivere codice di produzione fino a quando non hai scritto un test unitario che fallisce
- non scrivere più di un test unitario di quanto è sufficiente a far fallire la compilazione o l'esecuzione del test
- non scrivere più codice di produzione di quanto è sufficiente a passare il test che fallisce

Refactoring

Il *Refactoring* è un metodo strutturato e disciplinato per riscrivere o ristrutturare del codice esistente senza modificare il suo comportamento esterno applicando piccoli passi di trasformazione in combinazione con la ripetizione dei test dopo ciascun passo.

Si ha miglioramento del codice continuo e preparazione al cambiamento. Si effettua quando si aggiunge una funzionalità, si corregge un bug o durante la revisione di codice.

Il codice su cui è stato eseguito un buon refactoring è breve, conciso, chiaro e senza duplicazioni; sembra il lavoro di un maestro programmatore. Si dice che un codice che non ha queste qualità abbia un "cattivo odore", *code smell*.

Code smell:

- *Bloater*: codice troppo grande
- *Change Preventer*: rendono difficile il cambiamento

- *Object Orientation Abuser*: codice che non ha i principi OOA/D (es. Switch)
- *Coupler*: aumentano l'accoppiamento, problemi di responsabilità
- Dispensable: commenti grandi, codice duplicato, classi pigre.

Vediamo adesso alcuni esempi.

Long Method

- Sintomo: Un metodo contiene troppe righe di codice (più lungo di dieci righe)
- Problema: i metodi lunghi sono difficili da capire, modificare, riutilizzare
- Soluzione: utilizzare il Extract Method: si crea un nuovo metodo da una sezione di codice che esegue un'attività specifica e si sostituisce il codice originale con una chiamata al nuovo metodo

Duplicated Code

- Sintomo: Lo stesso codice o molto simile appare in molti luoghi
- Problema: Un fix in un codice duplicato clone non può essere propagata a tutti. Rende il codice più grande di quello che deve essere
- Soluzione: Se lo stesso codice si trova in due o più metodi nella stessa classe: usare il Extract Method e posizionare le chiamate per il nuovo metodo in entrambi i posti.

Feature Envy

- Sintomo: Un metodo accede ai dati di un altro oggetto più che ai propri dati.
- Problema: Errore di progettazione, metodo erroneamente inserito nella classe sbagliata
- Soluzione: Se un metodo deve essere chiaramente spostato in un altro luogo, utilizzare il Move Method.

Large Class

- Sintomo: Una classe contiene molti campi/metodi/linee di codice.
- Problema: Indica un errore di astrazione, è probabile che ci sia più di un interesse incorporata nel codice o, alcuni metodi appartengono ad altre classi
- Soluzione: Utilizza Move Method o Extract Class o Subclass

Data Class

- Sintomo: Una classe che ha solo variabili di classe, metodi/proprietà getter/setter e nient'altro. Sta solo agendo come contenitore di dati.
- Problema: Tipicamente, altre classi hanno metodi con Feature Envy. Indica che il progetto è veramente procedurale
- Soluzione: Esaminare i metodi che utilizzano i dati nella data class e usare Move Method o Extract Method per migrare questa funzionalità nella data class

Long Parameter List

- Sintomo: Molti parametri passati in un metodo.
- Problema: È difficile comprendere tali elenchi, che diventano contraddittori e difficili da usare man mano che si allungano.
- Soluzione: usare Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object.

Shotgun Surgery

- Sintomo: Per apportare eventuali modifiche è necessario apportare molte piccole modifiche a molte classi diverse.
- Problema: Un'unica responsabilità è stata suddivisa in un gran numero di classi.
- Soluzione: Usare il metodo Move Method e Move Field per spostare i comportamenti di classe esistenti in una singola classe

Comment

- Sintomo: Un metodo è ricco di commenti esplicativi.
- Problema: Il codice dovrebbe essere auto-esplicativo, quindi se avete bisogno di lunghi commenti, il vostro codice non è chiaro
- Soluzione: si può rinominare un metodo con Rename Method, oppure Extract Variable o Extract Method.

Refused Bequest

- Sintomo: una sottoclasse utilizza solo alcuni dei metodi e delle proprietà ereditati dai suoi genitori, la gerarchia non è corretta.
- Problema: I metodi non necessari possono semplicemente rimanere inutilizzati o essere ridefiniti e dare luogo a eccezioni.
- Soluzione: Replace Inheritance with Delegation o Extract Superclass.