

Appunti di

Ingegneria del Software

Rosario Terranova

Sommario

Introduzione.....	4
Cos'è l'ingegneria del software.....	4
Caratteristiche del software e differenze dagli altri prodotti.....	5
Categorie di software.....	6
Evoluzione del software.....	6
Processi e modelli di sviluppo del software.....	8
Attività, o fasi dello sviluppo del software.....	8
Modelli per i processi.....	10
Modelli a processo prescrittivo.....	10
Modelli a processo evolutivo.....	11
Altri modelli di processi.....	12
Modelli a processo di sviluppo agile.....	15
Pratica e gestione dei progetti software.....	19
Pratiche dello sviluppo software.....	19
Organizzazione attività.....	21
Pianificazione temporale.....	21
Gestione personale (stuff).....	22
Ingegneria dei Requisiti.....	25
Requisiti.....	25
Tipi di requisiti.....	26
Processo dell'ingegneria dei requisiti.....	29
Quality Function Deployment.....	30
Scoperta dei requisiti.....	31
Modellazione analitica attraverso UML.....	33
Suddivisione dei modelli UML.....	34
Diagramma dei casi d'uso.....	35
Diagramma delle attività.....	38
Diagramma degli stati.....	39
Diagramma di sequenza.....	41
Diagramma dei flussi (Data Flow Diagram, DFD).....	42
Diagramma delle classi.....	43
Diagramma di collaborazione.....	45
Progettazione delle architetture software.....	46
Progettazione.....	46
Progettare le architetture.....	46
Elementi dell'architettura software.....	47
Progettazione dei dati.....	47

Viste architetturali.....	47
Stili e modelli dell'architettura.....	48
Software ad architettura riflessiva.....	51
Paradigma Object-Oriented.....	55
Qualità del software nei sistemi ad oggetti.....	56
Ereditarietà nella OO.....	57
Polimorfismo.....	59
Classi astratte e interfacce.....	60
Thread.....	61
IDE Eclipse.....	66
Design Patterns.....	68
Abstract Factory.....	70
Factory method.....	71
Singleton.....	73
Adapter.....	74
Bridge.....	76
Composite.....	77
Decorator.....	80
Façade.....	81
Chain of Responsibility.....	83
Mediator.....	84
Observer.....	85
State.....	86
Model View Controller (MVC).....	87
Antipattern.....	88
Manutenzione e metriche.....	91
Dinamiche di evoluzione.....	91
Costi e modelli di manutenzione.....	91
Metriche.....	92
Test e collaudo del software.....	95
Processo di Verifica e Valutazione (V&V).....	95
Definizioni in ambito test.....	96
Principi per i test.....	96
Testing.....	97
Test strutturali.....	98
Test di integrazione.....	99
Test sotto stress.....	99
Testing manuale e automatico.....	99
Test regressivi.....	100

Trend di difetti scoperti.....100

Introduzione

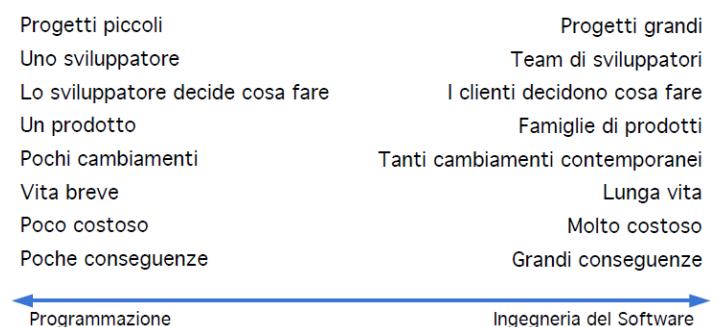
L'invenzione di una tecnologia può avere effetti profondi e talvolta inattesi su altre tecnologie a prima vista non correlate, su imprese commerciali, e talvolta perfino su intere culture. Questo fenomeno viene chiamato la **Legge delle conseguenze impreviste**. Oggi il software rappresenta la tecnologia più importante a livello mondiale ed è anche un chiaro esempio di questa legge. Nessuno avrebbe potuto prevedere che il software sarebbe stato incorporato in ogni genere di sistemi: trasporti, telecomunicazioni, apparecchi industriali, macchine d'ufficio, ecc., e che avrebbe consentito lo sviluppo di nuove tecnologie, come l'ingegneria genetica. Infine nessuno avrebbe potuto prevedere che milioni di programmi avrebbero dovuto essere corretti, adattati e migliorati nel corso del tempo, e che questa attività di manutenzione avrebbe richiesto più persone e risorse di quelle necessarie per creare il software.

Dunque data l'importanza del software, era necessario creare una scienza che analizzi passo per passo i vari metodi di sviluppo e manutenzione di esso; tale processo comprende un insieme di metodi e una serie di strumenti, la cosiddetta **ingegneria del software**.

Cos'è l'ingegneria del software

L'ingegneria del software si occupa della creazione di soluzioni economiche ed efficienti per problemi pratici, applicando conoscenze scientifiche ai fini di costruire prodotti software con benefici per i clienti ed anche per le società.

La differenza tra ingegneria del software e computer science è che la seconda si orienta sulla teoria ed i fondamenti, mentre la prima si orienta ai problemi pratici di sviluppo e consegna di prodotti software utili.



Ruolo del software

Oggi il software svolge un duplice ruolo: è un prodotto ma allo stesso tempo è un veicolo per distribuire un prodotto.

- In quanto prodotto, esso esprime il potenziale elaborativo dell'hardware in cui sta girando, o più in generale, di una rete di computer. Collocato in un telefono o in un mainframe, esso è un trasformatore di informazioni in grado di produrre, gestire, acquisire, modificare, presentare o trasmettere.
- In quanto veicolo di prodotti, il software agisce come base per il controllo di computer (sistema operativo), la comunicazione di informazioni (reti informatiche) e la produzione e controllo di altri programmi (strumenti e ambienti di sviluppo).

Storia del software

Il ruolo del software ha subito un enorme cambiamento nell'arco di poco più di 50 anni. Gli straordinari progressi delle prestazioni dell'hardware, le profonde innovazioni nell'architettura dei computer, l'incremento consistente delle capacità di memoria, di archiviazione e la vasta gamma di dispositivi di input e output hanno portato alla realizzazione di sistemi di elaborazione sempre più sofisticati. Negli anni '60 i sistemi software diventarono sempre più grandi (controllo aereo, prodotti commerciali). Nel '68 in un convegno NATO fu coniato il termine *crisi del software*, poiché vi erano produzioni in ritardo, costi maggiori di quelli preventivati, mancata affidabilità dei sistemi software prodotti.

Progettazione del software e costi

La **progettazione** del software è un processo iterativo (di ripetizione dei processi) attraverso il quale i requisiti sono tradotti un unico schema per la costruzione del software. All'inizio tale schema da una vista globale del software, rappresentando il progetto a un alto livello di astrazione che si può direttamente riportare a requisiti specifici per i dati, le funzioni e il comportamento. Procedendo con le iterazioni, i raffinamenti successivi conducono a rappresentazioni su livelli sempre meno astratti.

I **costi** orientativi per lo sviluppo di un sistema software dipendono dal modello adottato, ad esempio per i sistemi di lunga durata si ha 25% allo sviluppo mentre il 75% all'evoluzione.

Abilità degli ingegneri del software

Conoscenza di

- Algoritmi e strutture dati
- Linguaggi di programmazione

Capacità di modellare

- Operare a vari livelli di astrazione (es. macchine virtuali)
- Capire i requisiti, scrivere specifiche
- Costruire modelli e ragionare con essi

Capacità sociali

- Lavorare in team grandi
- Comunicare con le persone del team e con i clienti
- Gestire il tempo e le risorse

Caratteristiche del software e differenze dagli altri prodotti

Per afferrare compiutamente la natura del software è importante esaminare le caratteristiche che lo distinguono dagli altri prodotti del lavoro umano. Il software è un elemento *logico*, immateriale. Di conseguenza, le caratteristiche tipiche del software sono molto diverse da quelle dell'hardware.

1. Il software si sviluppa o si struttura, non si fabbrica nel senso tradizionale
2. Il software non si consuma nel tempo
3. Mentre l'industria si rivolge sempre più verso l'assemblaggio di vari componenti da diverse aziende (es. produzione di auto), la maggior parte del software continua a essere realizzato in modo specifico.

Caratteristiche proprie del software

Abbiamo diverse caratteristiche proprie del software che non sono di altri prodotti:

- *Complessità*: Il software è comunque un prodotto *complesso* poiché:
 - o Componenti tutte differenti
 - o Numeri di stati crescente in modo combinatorio
 - o Grandi dimensioni
 - o Astratto ed immateriale
 - o Non esistono leggi naturali che lo regolano
 - o Difficile da comprendere
- *Conformità*: Il software deve confermare (adeguare) all'ambiente esterno (molte interfacce hardware, vari utenti con profili differenti, processi lavorativi predefiniti); la conformità aggiunge complessità al software.
- *Modificabilità*: se un sistema software è di successo esiste sempre la necessità di cambiarlo per adattarlo ad una realtà che cambia. Le richieste di estensione aumentano all'aumentare del successo.
- *Invisibilità*: il software è invisibile e immateriale poiché non può essere catturato completamente da un'unica rappresentazione geometrica. Comunque esistono delle rappresentazioni che mirano ad evidenziare il flusso di controllo e dei dati, e le dipendenze dei componenti e variabili (UML).
- *Qualità*: le tecniche dell'ingegneria cercano di produrre sistemi software entro i *costi* e i *tempi* preventivati e con *qualità* accettabile. Per valutare la qualità di un software ci si basa sulla seguente definizione: **Totalità di caratteristiche di un prodotto che si basano sulla abilità a soddisfare i bisogni esplicativi ed impliciti**. Dunque i criteri per valutare la qualità sono:
 - o *Correttezza*: ovvero l'aderenza allo scopo e conformità alle specifiche; un sistema software è corretto se soddisfa le specifiche funzionali.
 - o *Efficienza*: l'uso di risorse da parte del software dovrebbe evitare sprechi di memoria o processore.
 - o *Manutenibilità*: la facilità di cambiare il software per soddisfare le esigenze che cambiano

- o *Dependability*: il software dovrebbe essere affidabile (probabilità che operi come atteso in un certo intervallo di tempo) e sicuro (security per proteggere i dati, safety per proteggere l'hardware).
- o *Usabilità*: il software dovrebbe essere usato facilmente dagli utenti per cui è stato progettato.

Gli attributi della qualità

Hewlett-Packard ha sviluppato un insieme di attributi di qualità del software cui è stata assegnata la sigla **FURPS** (funzionalità, usabilità, affidabilità, prestazioni e supportabilità). Tali attributi rappresentano un obiettivo per tutti i progetti software.

- *Funzionalità*. Valutata controllando le capacità del programma, la generalità delle funzioni derivate e la sicurezza del sistema globale.
- *Usabilità*. Valutata considerando i fattori umani, l'estetica generale, la coerenza e la documentazione.
- *Affidabilità*. Valutata misurando la frequenza e la gravità dei guasti, la precisione dei risultati di output, la capacità di recuperare una situazione di guasto e la prevedibilità del programma.
- *Prestazioni*. Misurate in termini di velocità di elaborazione, tempo di risposta, consumo di risorse, produttività ed efficienza.
- *Supportabilità*. Combina la capacità di estendere il programma, l'adattabilità, la facilità di servizio, la collaudabilità, la compatibilità, la configurabilità, la facilità d'installazione e la facilità di individuazione problemi.

Categorie di software

Oggi esistono sette grandi categorie di software che presentano continue sfide per gli ingegneri del software.

- **Software di sistema**: collezione di programmi al servizio di altri programmi. Alcuni tipi di software di sistema (ad es. compilatori, editor, gestori di file) elaborano strutture di dati complesse ma ben determinate; altri (S.O., driver, software di rete) elaborano dati ampiamente indeterminati. I tratti salienti sono: fitta interazione con l'hardware, utilizzo da più utenti, condivisione delle risorse, pianificazione dei tempi e gestione dei processi, strutture dati complesse e interfacce esterne multiple.
- **Software applicativo**: costituito da programmi indipendenti che risolvono specifici bisogni. Le applicazioni di questo tipo elaborano dati commerciali o tecnici in modo da facilitare le operazioni commerciali oppure le attività decisionali, gestionali o tecniche. Oltre alla tradizionale elaborazione dei dati, il software applicativo viene utilizzato per controllare funzioni commerciali in tempo reale es. elaborazione delle transizioni nei punti vendita).
- **Software scientifico e per l'ingegneria**: è stato sempre legato ad algoritmi di calcolo intensivo. Le applicazioni vanno dall'astronomia alla vulcanologia ai robot industriali. Tuttavia, nuove applicazioni in area scientifico-ingegneristica si staccano dagli algoritmi classici (es. CAD) per assumere sempre più le caratteristiche del software real-time e perfino di sistema.
- **Software embedded**: risiede in un prodotto o sistema che viene utilizzato per implementare e controllare caratteristiche e funzioni limitate per l'utente finale o per il sistema stesso. Svolge funzioni limitate e talvolta fuori dal comune (es. controllo dei comandi di un forno a microonde) oppure offre funzionalità di controllo rilevanti (es. controllo del carburante di un auto). Possono anche avere un sistema operativo proprietario molto limitato (es. TV) o avere solo un software che gira perennemente (es. orologio digitale).
- **Software a linea di prodotti**: progettato per fornire funzionalità specifiche utilizzabili da vari clienti, possono concentrarsi su un mercato specifico (es. controllo inventario) oppure rivolgersi al mercato dei consumatori di massa (es. programmi di videoscrittura, fogli elettronici, grafica computerizzata, intrattenimento, ecc.).
- **Applicazioni web**: insieme di file ipertestuali collegati che presentano informazioni utilizzando testo e limitate funzionalità grafiche. Tuttavia si sono evoluti fino a diventare sofisticati ambienti in grado di fornire all'utente finale funzionalità indipendenti.

- **Software per l'intelligenza artificiale:** si serve di algoritmi non numerici per risolvere problemi complessi. Le applicazioni in quest'area comprendono la robotica, i sistemi esperti, le reti neurali, ecc.

Software preesistente (legacy)

Vi sono centinaia di programmi software che rientrano in uno dei sette grandi dominio applicativi. Alcuni di questi sono software assolutamente nuovi e attuali, altri sono meno recenti, altri decisamente antichi. Questi programmi più vecchi sono denominati **software legacy**, ovvero sistemi software antichi ma di valore (indispensabili) che devono essere mantenuti ed aggiornati.

L'unico loro difetto è la *scarsa qualità*. Dato che i sistemi preesistenti talvolta presentano una struttura non estendibile, un codice complesso, una documentazione scarsa e inesistente, e tanti altri problemi, vi è la necessità di un loro cambiamento radicale. Se il software preesistente risponde ai bisogni degli utenti e si comporta in modo affidabile, non rappresenta un problema e non deve essere corretto. Tuttavia, con il passare del tempo essi devono evolversi per rispondere ai bisogni delle nuove tecnologie o essere interpolabili con sistemi più moderni. Quando si presentano queste forti motivazioni, il software legacy deve essere sottoposto a *reingegnerizzazione* in modo da mantenerlo utilizzabile per il futuro.

Evoluzione del software

Indipendentemente dalla sua applicazione, dimensione o complessità, il software si evolve nel corso del tempo. Le operazioni di modifica o manutenzione si verificano quando vengono corretti degli errori o il software viene adattato a un nuovo ambiente, quando il cliente richiede nuove funzionalità e quando l'applicazione deve essere riprogettata per fornire benefici legati a un contesto più moderno.

Esiste una **teoria unificata dell'evoluzione del software** che descrive un'analisi dettagliata sull'evoluzione del software, le leggi derivate da essa sono:

- o **Legge del cambiamento continuo (1974):** i sistemi elettronici devono essere costantemente adattati o diverranno sempre meno soddisfacenti.
- o **Legge della complessità crescente (1974):** mentre un sistema elettronico evolve, aumenta la complessità, a meno che non si lavori sulla riduzione di essa.
- o **Legge della conservazione della stabilità organizzativa (1980):** il livello di attività globale efficace in un sistema elettronico è invariante nel corso della vita del prodotto.
- o **Legge della conservazione della familiarità (1980):** a mano a mano che un sistema elettronico si evolve, gli sviluppatori, personale vendita e utenti devono mantenere la padronanza del suo contenuto e comportamento, in modo da garantire un'evoluzione soddisfacente.
- o **Legge della crescita continua (1980):** il contenuto funzionale del sistema elettronico deve essere costantemente incrementato in modo da mantenere la soddisfazione degli utenti nel corso della vita del sistema.
- o **Legge della riduzione della qualità (1996):** la qualità del sistema elettronico declina costantemente, a meno che non venga gestito rigorosamente e adattato ai cambiamenti dell'ambiente lavorativo.
- o **Legge dei feedback del sistema (1996):** i processi operativi di un sistema elettronico sostituiscono sistemi feedback multilivello, multiciclo, multiagente e devono essere trattati come tali per ottenere un miglioramento significativo.

Obiettivi e sfide dell'ingegneria del software

- Affrontare sistemi legacy, eterogenei e ridurre i tempi di consegna.
- Sviluppo di sistemi e software applicativi che consentano a tutti i tipi di dispositivi di comunicare attraverso grandi reti wireless.
- Studiare applicazioni semplici ma sofisticate che offrano benefici all'utente finale ovunque egli si trovi nel mondo.
- Realizzare codice autodescrittivo e sviluppare tecniche che consentano ai clienti e agli sviluppatori di sapere quali modifiche devono essere apportate e come tali modifiche si manifesteranno nel software.

- Mantenere viva la *New Economy* costruendo applicazioni che facilitino le comunicazioni di massa e la distribuzione di prodotti di massa utilizzando concetti che si stanno formando solo ora.

Responsabilità

Un ingegnere del software deve comportarsi in modo onesto ed eticamente responsabile: la confidenzialità di collaboratori e clienti deve essere rispettata, il livello di competenza non deve essere falsato, le leggi sulla proprietà intellettuale (*copyright*) devono essere conosciute e rispettate, le capacità tecniche non devono essere impiegate in modo non appropriato (es. per diffondere virus, danneggiare sistemi altrui, hacking, ecc.).

Processi e modelli di sviluppo del software

Quando si realizza un prodotto è importante svolgere una serie di passi prevedibili, una sorta di percorso che aiuti ad ottenere risultati di alta qualità nel tempo prefissato. Questo percorso è chiamato “*processo di sviluppo software*”. Esso descrive le attività (o **task**) necessarie allo sviluppo di un prodotto software e come queste attività sono collegate tra loro.

Gli ingegneri del software adattano il processo alle loro esigenze e poi lo seguono. È importante perché introduce stabilità, controllo e organizzazione in un’attività che, se lasciata libera, diventa piuttosto caotica.

- Un **processo software** è un insieme di attività ed i loro risultati che permettono di produrre in sistema software.
- Un **modello software** è una descrizione semplificata di un processo software sotto una particolare prospettiva.

Adattabilità del processo

A un livello dettagliato, il processo che si adotta dipende al software che si sta realizzando. Un determinato processo può essere appropriato per creare il software di un aereo, mentre per realizzare un sito web può essere necessario impiegare un processo completamente differente.

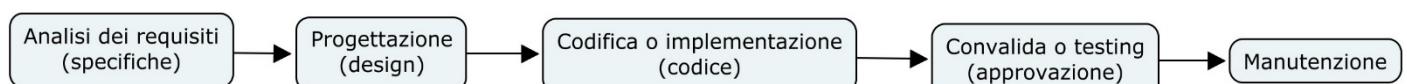
Prodotto e verifica

Il prodotto del lavoro sono programmi, documenti e dati sviluppati come conseguenza dell’attività di ingegneria del software definite dal processo. Per determinare la maturità di un processo software possono essere impiegati vari meccanismi di valutazione; tuttavia sono i migliori indicatori dell’efficacia del processo utilizzato:

- La qualità
- L’ottimizzazione dei tempi
- L’utilizzabilità a lungo termine del prodotto che si sta realizzando

Attività, o fasi dello sviluppo del software

La seguente struttura di un processo può essere applicata alla maggior parte dei progetti software:



Fase di analisi dei requisiti

L’analisi dei requisiti è il processo che porta a definire le specifiche, stabilisce i servizi richiesti ed i vincoli del software. Questa attività strutturale riguarda le comunicazioni e collaborazioni con il cliente e comprendere la raccolta dei requisiti e altre attività correlate.

- **Requisito**: ciascuna delle caratteristiche che il software deve avere (tendono ad essere granulari, cioè molti e piccoli)
 - **Requisito funzionale**: COSA il sistema deve fare (funzionalità)
 - **Requisito non-funzionale**: COME il sistema lo fa (affidabilità, efficienza, prestazioni, ecc.)
- **Specifiche**: descrizione rigorosa delle caratteristiche del software
- **Feature**: un set di requisiti correlati tra loro (una feature permette all’utente di soddisfare un obiettivo)

Es. Feature di Firefox 3.6: Browsing privatamente: navigazione del web senza lasciare tracce; Password manager: ricordare le password dei siti, senza usare pop-up; Awesome Bar: trovare i siti preferiti in pochi secondi; One-click bookmark: bookmark, cerca e organizza siti web velocemente e facilmente

Requisiti di Firefox 3.7: Eseguire i plug-in in un processo separato per migliorare la stabilità dell’applicazione e diminuire i tempi di risposta; Migliorare i tempi di startup; Ottimizzare caricamento delle pagine.

Prima di fare un'applicazione bisogna fare lo **studio di fattibilità**, che si occupa dei nostri requisiti per creare il software, come tempo da dedicare, fondi monetari, ecc.

Fase di progettazione

Il processo che stabilisce la *struttura* software che realizza le specifiche. Questa attività, quindi, stabilisce un piano per il successivo lavoro: descrive le operazioni che devono essere svolte, i rischi probabili, le risorse necessarie, i prodotti da realizzare e una pianificazione del lavoro. Attività della progettazione:

1. Suddivisione dei requisiti
2. Identificazione sottosistemi, ovvero progettazione architettura software
3. Specifica delle responsabilità dei sottosistemi
4. Progettazione di interfacce, componenti, strutture dati e algoritmi

Ognuna delle attività suddette produce un documento corrispondente che descrive un modello (design) di procedimento per il processo del software, il quale consente allo sviluppatore e al cliente di comprendere meglio i requisiti software e progettuali.

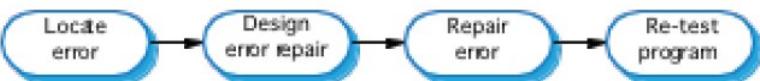
Esempio. Per gestire le password potrebbero servire due componenti, un contenitore e un altro che protegge questo contenitore. Creo l'UML per sapere cosa far corrispondere come codice. Il progettista deve anche sapere che tipo di output restituire anche non mettendo mano al codice.

Fase di implementazione

Produce un programma eseguibile a partire dalla struttura stabilita, dalla descrizione o dall'UML. Questa attività comprende la programmazione (o generazione del codice), ovvero nella traduzione dei modelli del progetto in un programma e le attività di collaudo necessarie per individuare errori ai fini della rimozione degli errori dal programma.

I programmati effettuano alcuni test sul programma prodotto per scoprire **bug** e rimuoverli. Per rimuovere i bug bisogna:

1. Localizzare l'errore nel codice
2. Rimuovere l'errore nel modello
3. Rimuovere l'errore nel codice
4. Effettuare nuovamente il test del programma



Finisce quando ho gestito tutti i casi possibili

Esempio. Nella ricerca se non trovo niente restituisco un messaggio d'errore.

Progettazione ed implementazione sono attività correlate e spesso sono alternate.

Fase di convalida (verifica & validazione)

La fase di convalida o Verifica e Validazione (V&V) del sistema software intende mostrare che il sistema software è conforme alle specifiche e che soddisfa le richieste (aspettative) del cliente.

- Viene condotta tramite processi di revisione e test del sistema software
- I test mirano ad eseguire il sistema software in condizioni derivate dalle specifiche di dati reali che il sistema software dovrà elaborare.

Il cliente valutando il prodotto potrebbe fornire indicazione che si basano sulla propria valutazione.

Fase di Test

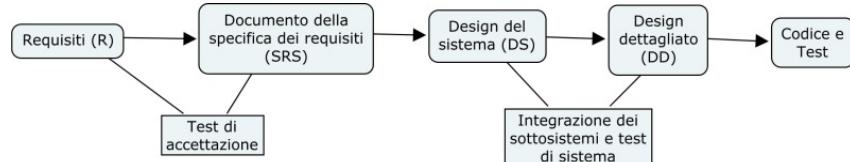
Vengono applicati diversi tipi di test del sistema software:

- *Test di componenti o unità (unit test):* I singoli componenti sono testati indipendentemente; componenti potranno essere funzioni, o oggetti, o loro raggruppamenti
- *Test di sistema:* L'intero sistema è testato, dando speciale importanza alle proprietà emergenti
- *Test di accettazione (alpha testing):* Test condotti dagli sviluppatori con dati del cliente per verificare che il sistema soddisfi le esigenze del cliente

- **Beta test:** Test condotti da alcuni clienti sul prodotto quasi completo
- Prodotti software corrispondenti alle varie fasi di test
- **versione alfa** (90% delle funzionalità pronte), con test a carico del cliente.
 - **versione beta** (100% delle funzionalità pronte), questa versione si fornisce a pochi utenti a fini di test nell'ambiente reale
 - **versione golden**, versione completa che verrà consegnata come prodotto finale.

Importanza delle figure professionali: Tester → Analista → Progettista → Programmatore

Quadro generale



- Dai requisiti (R) otteniamo il documento della specifica dei requisiti (SRS)
- Dall'SRS ricaviamo il design del sistema (DS)
- Dal DS ricaviamo il design dettagliato (DD)
- Da DD ricaviamo codice e test
- Da DS e da DD ricaviamo come integrare i sottosistemi e come fare i test di sistema
- Da R e SRS ricaviamo come fare i test di accettazione

Modelli per i processi

Il processo di sviluppo software può essere definito come una raccolta di modelli che definiscono un insieme di attività, azioni, compiti di lavoro, prodotti intermedi e/o comportamenti correlati necessari per sviluppare un software. In termini generali, un *modello software* fornisce una **struttura**, uno schema coerente per descrivere una caratteristica importante del processo di sviluppo software. Combinando tali modelli, un team di sviluppo software può realizzare un processo che risponda al meglio ai bisogni del progetto.

Obiettivi

I modelli possono essere definiti a qualsiasi livello di astrazione. In alcuni casi, un modello può essere utilizzato per descrivere un intero processo, o per descrivere un'attività strutturale, o un compito all'interno di un'attività strutturale.

Struttura di un modello

- **Nome del modello.** Al modello viene assegnato un nome che descrive la sua funzione principale.
- **Scopo.** Si definisce brevemente l'obiettivo del modello.
- **Tipo.** Si specifica il tipo del modello; tipicamente abbiamo 3 tipi:
 - o *Modello a compiti*. Definiscono il compito di lavoro che fa parte del processo
 - o *Modello a stadi*. Definiscono un'attività strutturale del processo
 - o *Modello a fasi*. Definisco la sequenza di attività strutturali che si verificano nel processo.
- **Contesto iniziale.** Le condizioni in base alle quali viene descritto il modello. Prima dell'attivazione del modello occorre chiedersi
 - o Quali attività sono già state svolte
 - o Qual è l'attuale stato del processo
 - o Quali informazioni sono disponibili in termini di ingegneria del software e di progetto
- **Problema.** Si descrive il problema che deve essere risolto dal modello.
- **Soluzione.** Viene descritta l'implementazione del modello.
- **Contesto risultate.** Le condizioni che si otterranno una volta che il modello è stato implementato con successo.

Modelli a processo prescrittivo

I modelli a processo prescrittivo definiscono un insieme distinto di attività, azioni, compiti, risultati e prodotti che sono necessari per ingegnerizzare un software di alta qualità. Questi modelli non sono perfetti, ma rappresentano un utile percorso per il lavoro dell'ingegnere del software.

Tali processi sono chiamati *prescrittivi* poiché prescrivono (o prevedono) un insieme di elementi del processo: attività strutturali, azioni di ingegneria del software, compiti, risultati, valutazione della qualità e meccanismi di controllo delle modifiche per ciascun progetto. Ogni modello a processo prescrive anche un *flusso di lavoro*, ovvero il modo in cui gli elementi del processo vengono correlati fra loro.

Anche se un processo è descrittivo, non è necessariamente statico: i modelli prescrittivi devono essere adattati alle persone, ai problemi e al progetto.

Modello a cascata (Waterfall)

Modello ideato da Royce negli anni 70, ispirato dalle catene di montaggio, è stato il primo modello di processo. Inizio la prima fase e mi dimentico delle successive, inizio la seconda e mi dimentico della prima e delle prossime, ecc., posso solo andare avanti; *si comincia la fase successiva solo se la fase precedente è completa* (prima specifica tutti i requisiti, poi produci tutto, poi testa tutto, ecc.).

Chiamato anche *ciclo di vita classico*, o *modello sequenziale lineare*, il modello a cascata suggerisce un approccio sistematico e sequenziale allo sviluppo di software, che inizia con la specifica dei requisiti e procede attraverso la pianificazione, la modellazione, la costruzione e il dispiegamento, culminando in un supporto continuo del software completo.



Pro	Contro
<ul style="list-style-type: none">- Consistenza tra artefatti- Ampia documentazione- Utile se i requisiti sono stabili e chiaramente definiti- Usato principalmente per sistemi grandi, complessi, critici e per gestire team numerosi.- Alta qualità del codice prodotto.	<ul style="list-style-type: none">- Lungo tempo per ottenere il prodotto- Poche interazioni con i clienti (solo nella fase iniziale)- Difficoltà ad introdurre i cambiamenti richiesti dal cliente- Poca conformità con tanti progetti reali

Oggi il lavoro sul software è più rapido e soggetto a un infinito flusso di cambiamenti. Il modello a cascata spesso è inappropriato per questo tipo di lavoro. Tuttavia è molto utile nelle situazioni in cui i requisiti sono fissi e il lavoro deve procedere fino al completamento in modo lineare.

Modelli a processo evolutivo

Il software, come tutti i sistemi complessi, è intrinsecamente flessibile e può cambiare nel corso del tempo. Al cambiare dei requisiti per cambiamenti dell'ambiente a cui è rivolto (business, hardware, etc.), il software deve evolvere se deve rimanere ad essere utile. Tutto ciò rende poco realistica la definizione di un percorso lineare per la realizzazione di un software. In queste situazioni, gli ingegneri utilizzano un modello a processo che sia stato realizzato esplicitamente per considerare un prodotto che evolve nel corso del tempo.

Il processo evolutivo ha due varianti:

- *Sviluppo per esplorazione*: spesso un cliente definisce un insieme di obiettivi generali per il software ma non identifica requisiti dettagliati in termini di input, elaborazione o output. In tal caso gli sviluppatori lavorano con i clienti, e dalle specifiche iniziali si

arriva per mezzo di trasformazioni successive (evoluzione) fino al sistema software finale; si deve partire da requisiti ben chiari ed aggiungere nuove caratteristiche definite dal cliente

- **Sviluppo Build and Fix:** lo sviluppatore può non essere sicuro dell'efficienza di un algoritmo, dell'adattabilità di un sistema operativo o della forma di iterazione; la documentazione inesistente o quasi; la comprensione limitata del sistema da produrre. In tal caso bisogna costruire la prima versione e modificarla fino a che il cliente è soddisfatto. Tuttavia avviene che la fase di design è pressoché inesistente e il codice prodotto è di bassa qualità.

Ne triamo che in queste situazioni viene prodotto un **prototipo** del software: esso assiste l'ingegnere e il cliente aiutandoli a comprendere meglio che cosa deve essere realizzato quando i requisiti sono troppo vaghi. Teoricamente, un prototipo serve per identificare i requisiti del software, ed avviene il seguente processo:

1. Vengono definiti i gli obiettivi globali del software
2. Modellazione sotto forma di *progettazione rapida*
3. Costruzione e implementazione del prototipo
4. Valutazione del cliente
5. Feedback del cliente utilizzato per affinare i requisiti del software

Si deve resistere ad ogni pressione di trasformare un prototipo in un prodotto funzionante. Il risultato sarà certamente di qualità scadente. Il prototipo può solo fungere da sistema preliminare solo per scoprire i requisiti, dopo di che deve essere scartato per realizzare il software vero e proprio, più orientato alla qualità.

Pro	Contro
<ul style="list-style-type: none"> - Necessario per scoprire i requisiti quando non sono chiari né al cliente né allo sviluppatore - Applicabile a sistemi di piccole dimensioni - Applicabile a singole parti di sistemi grandi (es. interfaccia utente) - Usato per sistemi con vita breve (es. prototipi) 	<ul style="list-style-type: none"> - Tempi lunghi - Sistemi difficilmente comprensibili e modificabili, probabilmente non corretti - Mancanza di visione d'insieme del progetto - Scarsa qualità del codice

Altri modelli di processi

Processo di sviluppo incrementale

Vi può essere un urgente bisogno di fornire all'utente un insieme limitato di funzionalità software con una certa rapidità per poi raffinare ed espandere queste funzionalità nelle release successive. In questo caso viene scelto un modello a processo progettato per produrre i software a incrementi.

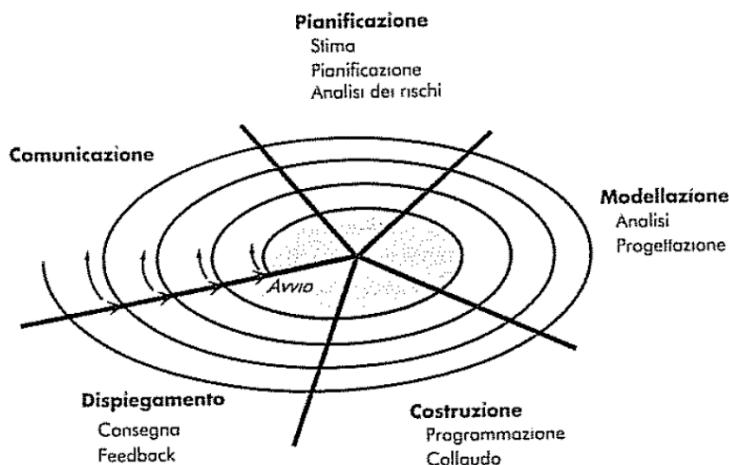
Esso combina alcuni aspetti del modello a cascata applicati iterativamente, e consiste nell'applicare più sequenze lineari, scalate nel tempo. Ogni sequenza lineare produce uno stadio operativo del software. Il primo stadio consiste spesso in un *prodotto base*, cioè un prodotto che soddisfi i requisiti fondamentali. Esso viene usato e provato dal cliente, dopo di che si stende un piano per lo stadio successivo dove si applicano modifiche al prodotto per soddisfare meglio le esigenze del cliente e si inseriscono nuove funzioni. Il processo si ripete fino a raggiungere il prodotto finale.

Processo CBSE

Il Component-based software engineering (CBSE) è un approccio per lo sviluppo software basato sul riuso. Si basa sul COTS (components of the shelf, componenti essitenti); esso prevede che ci siano delle parti di software già pronte che posso acquisire per creare un nuovo sistema software. Devo conoscere bene queste parti, analizzarle e posso modificare i requisiti proponendo al cliente di modificare le sue richieste per farle agganciare al software che già ho. Al cliente si fa pagare il lavoro anche dei componenti aggiunti. Abbiamo una forte componente di sviluppo e integrazione.

Modello a spirale

Proposto da Boehm, è un modello di processo di sviluppo software che abbina la natura iterativa della prototipazione e gli aspetti controllati e sistematici del modello a cascata, consentendo un rapido sviluppo di versioni via via più complete del software.



Esso si focalizza su tanti prodotti parziali. Il software viene sviluppato secondo una serie di release evolutive; durante le prime iterazioni, la release può essere un modello cartaceo o un prototipo, mentre durante le successive iterazioni, vengono prodotte versioni sempre più complete del sistema ingegnerizzato. Ogni *loop* (giro della spirale) è una fase, e consiste nei seguenti settori:

1. Comunicazione, per identificare gli obiettivi specifici per la fase corrente.
2. Pianificazione, dove sono valutati i rischi del progetto per ridurli (Rischio = qualcosa che può impedire il successo e che è sconosciuto. Successo = soddisfare tutti i requisiti).
3. Modellazione e Costruzione, ovvero sviluppo e convalida.
4. Dispiegamento, consegna, revisione del progetto e pianificazione fase successiva

Diversamente dagli altri modelli di processo che si arrestano alla messa in opera del prodotto, il modello a spirale può essere adattato in modo da estendersi all'intero arco di vita del prodotto. Pertanto, la prima rivoluzione della spirale può rappresentare un progetto di sviluppo del concetto che inizia dalla parte centrale del progetto e procede per più iterazioni finché lo sviluppo non è completo. Se il nuovo concetto deve essere realizzato in un prodotto vero e proprio, il progetto continua attraverso la spirale e inizia lo sviluppo di un nuovo prodotto. A ogni iterazione della spirale questo progetto evolve, e successivamente un altro giro attorno alla spirale può rappresentare un *ampliamento* del prodotto.

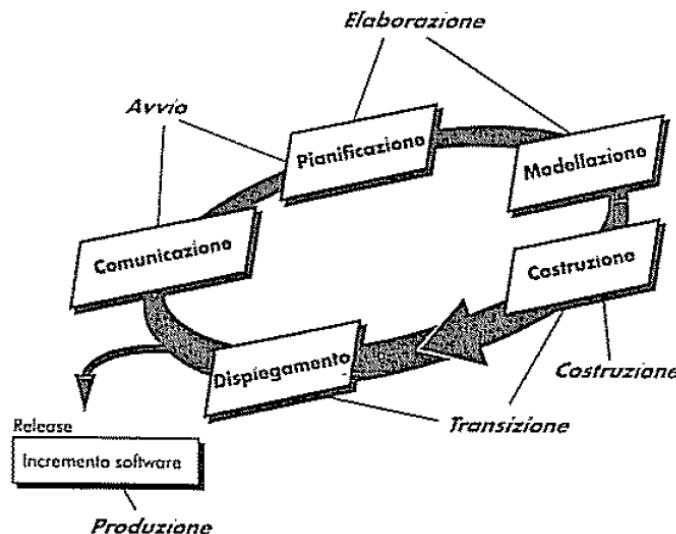
Es. Dobbiamo produrre una calcolatrice, al primo loop si progetta disegnando la GUI, si valuta il rischio, e si produce l'interfaccia grafica; al secondo loop si progettano gli algoritmi da utilizzare creando l'UML, si valutano i rischi e poi si implementano, ecc.

Pro	Contro
<ul style="list-style-type: none"> - Processo agile - Poco tempo per la prima versione del prodotto - Opportunità di interagire con il cliente - Ogni fase produce un codice testato ed integrato nel sistema complessivo - Facilmente modificabile e aggiornabile con un altro giro della spirale - Possibilità di applicare la prototipazione a ogni stadio dell'evoluzione - Rischi risolvibili a ogni loop. 	<ul style="list-style-type: none"> - Difficoltà nel convincere il cliente che la strategia evolutiva è sotto controllo. - Esigenza di competenze di alto livello per la stima dei rischi. - Problemi in caso di non risoluzione di un rischio rilevante.

Processo di sviluppo unificato razionale (Rational Unified Process, RUP)

RUP è un tentativo di sfruttare le migliori funzionalità e caratteristiche dei modelli convenzionali dei processi di sviluppo software, ma caratterizzandoli in modo da implementare molti dei migliori principi dello sviluppo agile. Esso riconosce l'importanza delle comunicazioni con cliente e dell'adozione di metodi lineari per descrivere il punto di vista del cliente nei confronti del sistema.

Le fasi (o attività) utilizzate sono: avvio, elaborazione, costruzione e transizione (produzione). Esse sono ripetute più volte, e ogni singola iterazione dura circa 2-6 settimane.

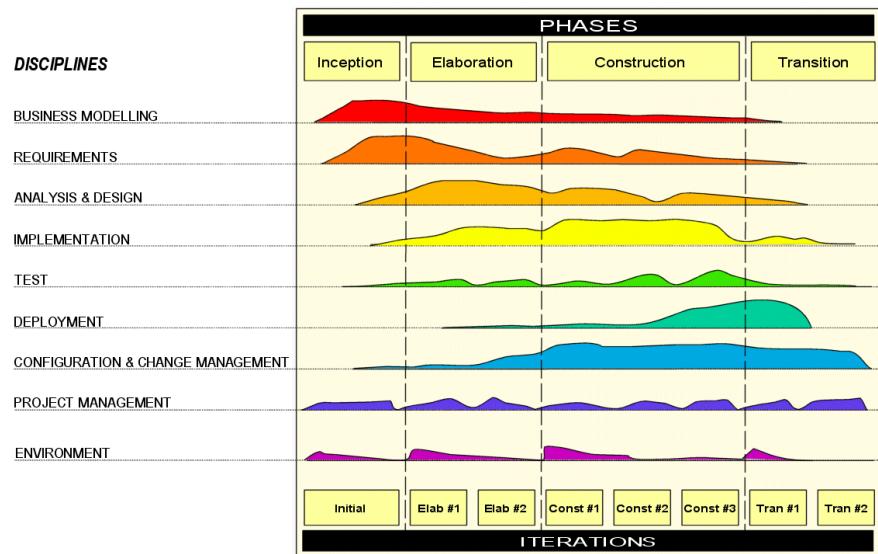


- **Fase di avvio:** comprende le attività di comunicazione con il cliente e di pianificazione. Collaborando con il cliente e gli utenti finali, è possibile identificare i requisiti del software, proporre un'architettura di base per il sistema e sviluppare un piano per la natura iterativa e incrementale del progetto come le risorse, i principali rischi, piano dei tempi e le fasi da seguire. I requisiti operativi vengono definiti tramite un insieme di casi d'uso preliminari che descrivono le caratteristiche e funzioni richieste da ogni grande classe d'utente. In generale, un caso d'uso descrive una sequenza di azioni che vengono svolte da un *attore* (persona, macchina, sistema) che interagisce con il software. Essi aiutano a identificare il campo d'azione del progetto e a definire le basi per la sua pianificazione.
- **Fase di elaborazione:** comprende le attività di elaborazione con il cliente e modellazione del modello generico del processo. L'elaborazione raffina e amplia i casi d'uso preliminari che sono stati sviluppati nell'ambito della fase di avvio ed espande la rappresentazione dell'architettura per includere cinque diverse viste del software: modello a casi d'uso, analitico, di progettazione, implementativo e di dispiegamento. Alla fine della fase, il piano viene attentamente riconsiderato per garantire che il raggio d'azione, i rischi e le date di consegna rimangano ragionevoli.
- **Fase di costruzione:** si sviluppa o acquisisce i componenti software che renderanno operativi per i singoli utenti i vari casi d'uso. Per fare ciò, i modelli analitico e di progettazione avviati durante la fase precedente vengono completati per riflettere la versione finale dell'incremento software. Tutte le nuove release vengono poi implementate nel codice sorgente. A mano a mano che vengono implementati i componenti, vengono anche eseguiti i test delle unità, e condotte delle attività di integrazione (assemblaggio dei componenti e collaudo). Alla fine di questa fase il sistema dovrebbe essere funzionante.
- **Fase di transizione:** il software viene fornito all'utente finale per effettuare i beta test e gli utenti rilevano i difetti e le necessarie richieste per le release. Alla conclusione della fase di transizione, l'incremento software diviene una release software utilizzabile.
- **Fase di produzione:** viene monitorato l'utilizzo del software, viene fornito un supporto per l'ambiente operativo e vengono ricevuti e valutati i rapporti sui difetti e sulle richieste di modifiche.

Ogni flusso di lavoro identifica i compiti necessari per ottenere un'importante azione id
ingegneria del software; come conseguenza del completamento con successo di questi
compiti vengono realizzati dei prodotti.

Le pratiche per un corretto funzionamento della RUP sono:

- ✓ Ciascuna iterazione è svolta entro un tempo prefissato
- ✓ Evitare di identificare molti requisiti all'inizio
- ✓ Progettare un'architettura che mette insieme le parti identificate e permette il riuso di componenti esistenti
- ✓ Per progetti grandi: requisiti e architettura base sviluppata da un piccolo team, dopo tale team si divide ed ognuno gestisce un sotto-progetto
- ✓ Gestione requisiti
- ✓ Trovare, organizzare e tracciare i requisiti iterativamente, tramite tool
- ✓ Modellazione grafica
- ✓ Prima della programmazione, effettuare il design grafico tramite UML
- ✓ Verifica continua della qualità
- ✓ Testare presto, spesso e integrando il software ad ogni iterazione
- ✓ Gestione cambiamenti
- ✓ Disciplinare gestione configurazioni, controllo versioni, protocollo di richiesta di cambiamenti, release minime alla fine di ogni iterazione



Modelli a processo di sviluppo agile

L'ingegneria del software agile combina una filosofia e un insieme di linee guida per lo sviluppo. La filosofia incoraggia i seguenti fattori:

- La soddisfazione del cliente e una consegna incrementale anticipata del software
- L'impiego di un team di progettazione compatto e molto motivato
- L'impiego di metodi formali
- Un livello minimo di prodotto di ingegneria del software
- Una generale semplicità di sviluppo

Si pone l'accento sulla consegna rispetto all'analisi e al progetto, e su una comunicazione attiva e continua fra gli sviluppatori e i clienti. Se ne occupano gli ingegneri software e gli altri **stakeholder** (dirigenti, clienti, utenti finali), i quali collaborano formando un team agile, un team che si auto-organizza e controlla il proprio destino. Un team agile favorisce la comunicazione e la collaborazione fra tutti i membri.

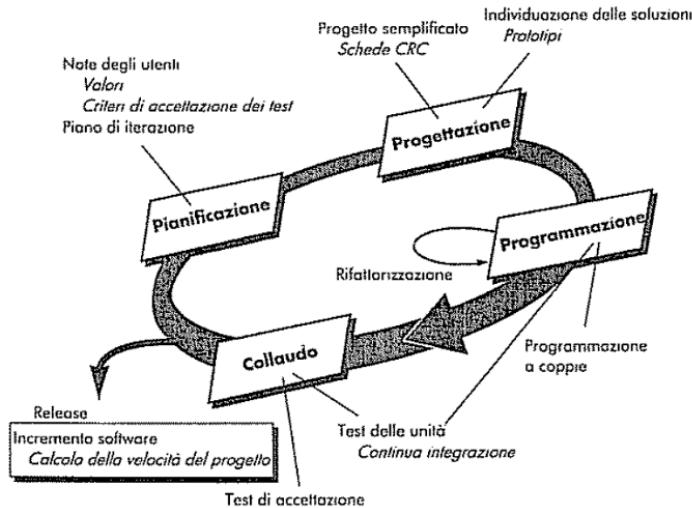
L'ingegneria del software agile rappresenta un'alternativa ragionevole all'ingegneria software convenzionale per determinare classi di software e determinati tipi di progetti software. È dimostrato che consente di fornire sistemi di successo con maggiore rapidità. Permangono tutte le attività strutturali principali (comunicazione con il cliente, pianificazione, modellazione, costruzione, consegna e valutazione), ma viene incentivato di più la costruzione e la consegna, sacrificando l'analisi e la progettazione della soluzione.

L'unico prodotto veramente importante è un **incremento software** che viene consegnato al cliente nella data specificata. Se il team agile concorda sul fatto che il processo funziona e il team produce incrementi software che soddisfano il cliente, si sta procedendo correttamente. La motivazione principale per l'agilità è la **pervasività del cambiamento**; gli ingegneri software devono essere rapidi per riuscire a considerare i rapidi cambiamenti di qualunque fattore (hardware, software, ecc.). Inoltre l'agilità incoraggia strutture e attitudini che agevolano le comunicazioni, pone l'accento su una rapida consegna del software funzionante, riduce l'importanza dei prodotti intermedi e adotta il cliente nel team di sviluppo.

Fattori privilegiati	Fattori di agilità
<ul style="list-style-type: none">- Individui rispetto a processi e strumenti- Disponibilità di software funzionante rispetto alla documentazione- Collaborazione con il cliente rispetto alla negoziazione dei contratti- Pronta risposta ai cambiamenti rispetto all'esecuzione di un piano	<ul style="list-style-type: none">- Considerare positivamente le richieste di cambiamento anche in fase avanzata di sviluppo- Fornire release del sistema software funzionante frequentemente- Costruire sistemi software con gruppi di persone motivate- Continua attenzione all'eccellenza tecnica

Extreme Programming (XP)

La programmazione XP è un modello di processo a sviluppo agile, e adotta un approccio orientato agli oggetti quale paradigma di sviluppo preferenziale. Essa comprende un insieme di regole e pratiche che determinano quattro attività strutturali: pianificazione, progettazione, programmazione e collaudo.



➤ Pianificazione

Si inizia con la descrizione delle caratteristiche e funzionalità che il software deve avere (**racconti**). Ogni racconto viene scritto dal cliente e viene inserito in una scheda indice; il cliente assegna a questo racconto un **valore** (una priorità) sulla base del valore globale di questa caratteristica o funzione; i membri del team XP valutano questi racconti e gli assegnano un **costo**. Se il racconto richiede più di tre settimane di sviluppo, al cliente viene chiesto di suddividerlo in tanti racconti più piccoli. Dopo aver raggiunti un accordo sulla data di consegna, il team XP ordina i racconti che verranno sviluppati nei seguenti modi:

1. Tutti i racconti verranno implementati immediatamente
2. Verranno implementati prima i racconti con il valore più elevato
3. Verranno implementati rimanendo i racconti più rischiosi

Dopo la prima release, il team XP calcola la **velocità del progetto**, ovvero il numero di racconti del cliente implementati nelle prime release. Essa viene utilizzata per stimare le date di consegna e le pianificazioni per le successive release, e per determinare se i racconti sono stati sottovalutati nell'ambito dell'intero progetto di sviluppo.

➤ Progettazione

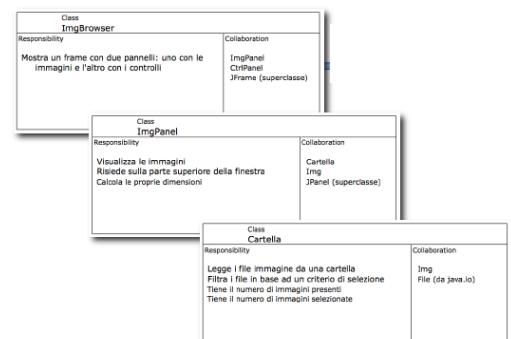
La progettazione XP segue i principi della semplicità, guidando l'implementazione di un racconto così come viene descritto (niente di più e niente di meno) creando una o più **storie**.

La programmazione XP incoraggia l'uso di **schede CRC** (d, figura a destra) quale meccanismo efficace per riflettere sul software in un contesto orientato agli oggetti.

Se nell'ambito della progettazione viene rilevato un problema progettuale difficoltoso, XP consiglia l'immediata creazione di un prototipo operativo per tale porzione del progetto; questo prototipo, chiamato **spike solution**, viene poi implementato e collaudato. Lo scopo è quello di ridurre i rischi all'avvio dell'implementazione e convalidare le stime originarie per la storia contenente il problema progettuale.

XP incoraggia anche la **rifattorizzazione** (refactoring), tecnica realizzativa e progettuale che consiste nel modificare un sistema software in modo da non alterare il comportamento esterno del codice ma da migliorare la sua struttura interna; è un modo disciplinato per "ripulire" il codice minimizzando la probabilità di introdurre bug. In pratica la rifattorizzazione migliora il progetto del codice già scritto.

Il giusto design per il software si ha quando passa i test, non ha parti duplicate, esprime ciascuna intenzione importante per i programmati e ha il numero più piccolo di classi e metodi. Bisogna non preoccuparsi di dover apportare cambiamenti dopo, ma fare la cosa più semplice che può funzionare



➤ Programmazione

Dopo che sono state sviluppate le storie ed è stata svolta tutta la progettazione preliminare, il team non deve procedere subito alla programmazione, ma piuttosto sviluppare una serie di **test** di unità che metteranno alla prova ognuna delle storie che devono essere incluse nella

release corrente, per tutto il tempo. Si eseguono i test più volte al giorno, non appena è stato prodotto del nuovo codice. Ci sono due tipi di test:

- ❖ **Test funzionali**: scritti dall'utente; effettuati dal cliente, sviluppatori e team testing; automatizzati; eseguiti almeno giornalmente; sono una parte della specifica dei requisiti, quindi documentano i requisiti.
- ❖ **Unit test**: scritti dagli sviluppatori (punto di vista del programmatore); scritti prima della codifica (TDD, Test Driven Development) ed anche dopo la codifica; supportano design, codifica, refactoring e qualità

Dopo la creazione dell'unità test, lo sviluppatore sarà maggiormente in grado di concentrarsi su ciò che deve essere implementato per passare tale test. XP incoraggia la tecnica della **programmazione a coppie** (pair programming), due persone collaborano alla stessa workstation per sviluppare il codice di realizzazione di una storia; questo fornisce un meccanismo di soluzione in tempo reale dei problemi (due teste ragionano meglio di una) e una garanzia di qualità. Essi sono programmati esperti e motivati dove il ruolo di uno dei partner è usare il mouse e la tastiera e pensare al miglior modo di implementare il metodo, mentre il ruolo dell'altro è chiedersi se l'approccio funzionerà, pensare ai test e pensare se il codice potrebbe essere fatto più semplicemente. Il pair programming aiuta la disciplina e sparge la conoscenza sul sistema

Quando la coppia di programmati completa il proprio lavoro, il codice da essi sviluppato viene **integrato** con il lavoro degli altri. Può esserci un team d'integrazione che si occupa di questo, o in altri casi sono i programmati a coppie stessi ad avere la responsabilità dell'integrazione. In ogni caso l'integrazione è continua, massimo un giorno. Tutti gli unit test devono essere superati, se un test fallisce la coppia che ha prodotto il codice deve ripararlo, se non può ripararlo, bisogna buttare il codice e ricominciare.

Ultimo fattore è che le costruzioni complicate (per il design) non sono permesse; bisogna mantenere le cose semplici in modo che il codice appaia uniforme e più facile da leggere. Bisogna usare tutta la stessa convenzione, così non si ha necessità di riformattare il codice, e avere la conoscenza di come usare

- o Spazi e Tab per indentazione
- o Posizione parentesi graffe
- o Scelta di nomi classi, metodi, attributi
- o Posizione commenti

➤ *Collaudo*

Le unità di test create all'inizio della fase di programmazione devono essere implementate utilizzando una struttura che consenta la loro automatizzazione. Questo incoraggia una strategia a test di regressione ogni volta che il codice viene modificato. I test di integrazione e convalida del sistema possono verificarsi anche quotidianamente, fornendo al team XP un'indicazione continua dei progressi e consentendo maggior facilità di individuazione degli aspetti problematici.

Infine abbiamo i **test di accettabilità**, chiamati anche "test del cliente", sono specificati dal cliente e si concentrano sulle funzioni e sulle caratteristiche globali del sistema che sono visibili e controllati dal cliente.

Riassumendo, XP si focalizza sul codice facendo solo le cose che sveltiscono la produzione del codice, codifica e test. XP si orienta sulla gente poiché la conoscenza del sistema è trasferita attraverso la comunicazione tra la gente. XP è leggero poiché rimuove i costi aggiuntivi, crea prodotti di qualità tramite test rigorosi.

Ogni settimana ho una nuova versione del codice, ho pochissima documentazione, si privilegia il cambiamento.

I test si fanno ogni una/due ore. Piccole release, i requisiti gli scrive il cliente, gli sviluppatori stimano la durata e si sforzano di capire quello da fare, se no la card viene strappata e il cliente deve riscriverla fino a quando non si raccolgono un numero di **storycard** (storie dell'utente).

Story Card

Una story card è un insieme di descrizioni delle caratteristiche che un software deve avere date dal cliente, la quale dà indicazioni sulle necessità del cliente nella propria mansione

lavorativa per applicare il software. Sono le storie dell'utente, e si applicano per casi d'uso leggeri. Le dimensioni della card sono: 5" x 3" circa 12x7 cm.

La descrizione delle storie comprende 2-3 frasi su una card che

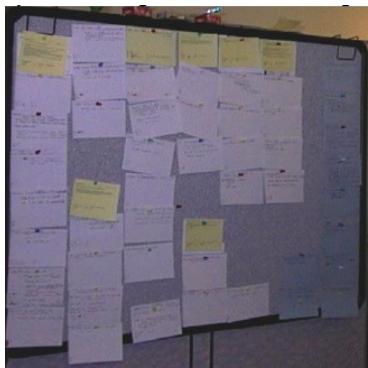
- Sono importanti per il cliente e sono scritte dal cliente
- Possono essere testate
- Permettono di ricavare una stima del loro tempo di sviluppo
- Possono essere associate a priorità

Il Template per story card (ovvero campi di una story card) sono:

- ✓ Data, Numero, Priorità, Tempo stimato, Riferimenti
- ✓ Descrizione requisito
- ✓ Lista di task per ciascun requisito, ovvero ciò che lo sviluppatore dovrà fare
- ✓ Note

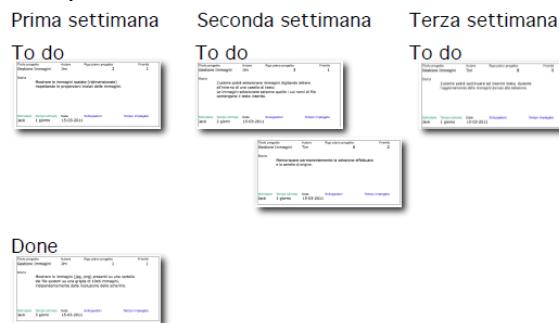
Customer Story and Task Card		B/W Development \ COLA
DATE: 3/19/98	TYPE OF ACTIVITY: NEW: <input checked="" type="checkbox"/> FIX: <input type="checkbox"/> ENHANCE: <input type="checkbox"/> FUNC. TEST: <input type="checkbox"/>	
STORY NUMBER: 1275	PRIORITY: USER: <input type="checkbox"/> TECH: <input type="checkbox"/>	
PRIOR REFERENCE: _____		
RISK: _____ TECH ESTIMATE: _____		
TASK DESCRIPTION: SPLIT COLA: When the COLA rate chgs. in the middle of the B/W Pay Period, user will want to pay the 1 st week of the pay period at the OLD COLA rate and the 2 nd week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design. For the OT, we will run a mframe program that will pay or calc the COLA in the 2 nd week of OT. The plant currently retransmits the hours data for the 2 nd week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA.		
NOTES: _____		
TASK TRACKING: Gross Pay Adjustment, Create RM Boundary and Place in DEEnt Express COLA		
Date	Status	To Do
		Comments: BIN

Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	1	1
Storia			
Mostrare le immagini (jpg, png) presenti su una cartella del file system su una griglia di 10x6 immagini, indipendentemente dalla risoluzione dello schermo.			
Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	2	1
Stimatore			
Jack	Tempo	3 giorni	
Storia			
Mostrare le immagini scalate (ridimensionate) rispettando le proporzioni iniziali delle immagini.			
Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	3	1
Stimatore			
Jack	Tempo st	1 giorni	
Storia			
L'utente potrà selezionare immagini digitando lettere all'interno di una casella di testo. Le immagini selezionate saranno quelle i cui nomi di file contengono il testo inserito.			
Stimatore	Tempo stimato	Data	Sviluppatori
Jack	2 giorni	15-03-2011	Tempo impiegato



Story Board

È una bacheca dove vengono appese tutte le story card del cliente, ognuna contiene una caratteristica del software con la sua priorità, rischio, importanza, descrizione, ecc.



Confronto tra processi

Processo	Punti di forza	Punti di debolezza	Durata
Cascata	Approccio regolato da documenti; appropriato per sistemi con requisiti stabili	Non soddisfacente per i clienti per il poco feedback permesso; non permette l'adattamento a requisiti variabili	Tipicamente un anno
Build and fix	Approccio per piccoli sistemi che non necessitano manutenzione	Architettura incidentale; non soddisfacente per sistemi non banali	Qualche mese
Spirale	Approccio disciplinato regolato da documenti; valutazione dei rischi	Può essere usato solo per sistemi grandi	1 anno a ciclo
RUP	Approccio iterativo; uso di componenti; modellazione grafica	Generico => Poco chiaro	
XP	Permette l'adattamento dei requisiti	Poca documentazione	Qualche mese

Dentro un processo di sviluppo possiamo inglobare un altro processo di sviluppo, ad esempio nella spirale al suo interno uso la cascata.

Pratica e gestione dei progetti software

La gestione di un progetto software (management) descrive le attività (**task**) necessarie affinché il prodotto software sia finito in tempo e in accordo ai requisiti delle organizzazioni di sviluppo e di acquisto.

Le attività da seguire per la gestione dei progetti sono:

- Scrrittura della proposta (proposal)
- Pianificazione e scheduling progetto
- Individuazione costi di progetto
- Monitoraggio e revisioni progetto
- Selezione e valutazione personale
- Scrrittura report e presentazioni

Tali attività sono descritte dettagliatamente dalla pratica dello sviluppo software.

Pratiche dello sviluppo software

Una pratica è un insieme generico di concetti, principi, metodi e strumenti che occorre considerare durante la pianificazione e lo sviluppo del software. Rappresenta i dettagli del processo di sviluppo software: gli elementi necessari per produrre software di alta qualità. Esso fornisce a tutte le persone coinvolte nella realizzazione del prodotto software un percorso per giungere con successo a destinazione. Dice dove si trovano i ponti, le interruzioni e i bivi. Essa è tutto ciò che di deve svolgere per trasformare un'idea in realtà.

L'essenza della pratica

George Polya ha definito l'essenza della soluzione dei problemi, e di conseguenza, l'essenza della pratica dell'ingegneria del software:

1. Comprendere il problema (comunicazione e analisi)
2. Pianificare una soluzione (modellazione e progettazione del software)
3. Esecuzione del piano (generazione del codice)
4. Esame della precisione dei risultati (test e valutazione qualità).

Principi base

Un principio è un'importante legge o presupposto necessario in un sistema di pensiero. Indipendentemente da loro livello di profondità, i principi aiutano a definire una struttura mentale per la pratica dell'ingegneria del software. David Hooker ha proposto sette principi di base dell'ingegneria del software.

1. Il senso dell'esistenza. Un software esiste per un motivo, ovvero ha un valore per i suoi utenti. Tutte le decisioni devono essere prese sulla base di questo concetto. Prima di specificare un requisito, trascrivere una funzionalità del sistema, e prima ancora di esaminare le piattaforme hardware o i processi di sviluppo, occorre porsi la domanda di che valore rappresenta questo per il sistema.
2. Il principio della semplicità KISS (Keep It Simple, Stupid!). Tutta la progettazione deve essere improntata alla massima semplicità, ma senza esagerare. Questo facilita la realizzazione di un sistema di facile comprensione e manutenzione. Ciò non vuol dire che determinate caratteristiche debbano essere eliminate a nome della semplicità. Ciononostante i progetti più eleganti sono normalmente anche quelli più semplici. Non è facile raggiungere la semplicità.
3. Concentrarsi sulla visione. Una visione chiara è fondamentale nel successo del progetto software. Senza una visione, quindi senza un'integrità concettuale, un sistema rischia di diventare un'accozzaglia di progetti incompatibili, tenuti insieme da viti sbagliate.
4. Ciò che si sta producendo verrà consumato da altri. Si deve sempre intraprendere l'attività di definizione delle specifiche, di progettazione e di implementazione considerando che qualcun altro (utenti, sviluppatori, manutentori) dovrà comprendere ciò che si sta facendo.
5. Essere aperti verso il futuro. Non si deve mai progettare chiudendosi in un angolo. La vera potenza commerciale di un sistema software è la progettazione fin dall'inizio tenendo in considerazione le necessità future.

6. Pianificare il risultato. Il riutilizzo consente di risparmiare tempo e fatica. Il riutilizzo del codice e dei progetti è stato dichiarato come uno dei principali vantaggi delle tecnologie ad oggetti. Tuttavia per sfruttare tali funzionalità di riutilizzo è necessaria un'attenta pianificazione, e delle ben note tecniche. È fondamentale comunicare agli altri le opportunità di riutilizzo. La pianificazione e il riutilizzo riducono i costi e aumentano il valore sia dei componenti riutilizzabili sia dei sistemi in cui essi verranno incorporati.
7. Riflettere. Una riflessione chiara e completa prima di ogni azione produce quasi sempre risultati migliori.

Se ogni ingegnere software e ogni team di sviluppo seguisse i sette principi di Hooker, molte difficoltà sperimentate nella realizzazione di sistemi complessi semplicemente non esisterebbe più.

Pratiche di comunicazione

Prima che i requisiti del cliente possano essere analizzati, modellati o specificati, occorre raccoglierli tramite un'attività di *comunicazione* o *descrizione requisiti*. Un'efficace comunicazione è fra le attività più problematiche che devono essere svolte dall'ingegnere del software. Per poter comunicare occorre aver compreso il punto di vista dell'altra parte, essere disposti a comprendere i suoi bisogni e poi ascoltare.

1. Ascoltare. Occorre cercare di concentrarsi sulle parole dell'altro invece di formulare le proprie risposte a queste parole. Chiarire ogni dubbio evitando di interrompere continuamente.
2. Prepararsi prima di comunicare. Dedicare del tempo alla comprensione del problema prima di partecipare a un incontro tramite ricerche e documentazioni. Se si conduce una riunione è opportuno preparare un'agenda.
3. Qualcuno dovrebbe facilitare l'attività. Ogni riunione di comunicazione deve avere un *leader*, ovvero un moderatore che aiuti a far progredire la conversazione in modo produttivo, che serve a mediare per ogni conflitto che può sorgere, e che serve a garantire che vengano seguiti altri principi.
4. La migliore comunicazione è sempre il colloquio diretto. Domande e risposte chiare eliminano la maggior parte delle complessità. Possono aiutare alla comprensione dei disegni o grafici.
5. Prendere appunti e documentare le decisioni. Una partecipante alla comunicazione dovrà occuparsi di trascrivere tutti i punti e le decisioni importanti.
6. Privilegiare la collaborazione. Ogni collaborazione contribuisce ad ampliare la fiducia fra i membri del team e a creare un obiettivo comune del team.
7. Rimanere concentrati e modularizzare la discussione. Più persone sono coinvolte nella comunicazione, più è probabile che la discussione salti improvvisamente da un argomento all'altro. Il moderatore deve cercare di mantenere modulare la conversazione, lasciando un argomento solo dopo che è stato risolto.
8. Se qualcosa è poco chiaro, tracciare un disegno. Dato che la comunicazione verbale presenta limiti, uno schizzo o un disegno aiuta a chiarire il discorso.
9. Quando ci si accorda su un argomento, procedere. La comunicazione richiede del tempo. Invece di soffermarsi troppo su un argomento, i partecipanti devono riconoscere che la discussione presenta molti argomenti, e talvolta procedere è il modo migliore per mantenere l'agilità della comunicazione. Se una caratteristica è poco chiara e non più essere chiarita al volo, procedere.
10. La negoziazione non è una gara o un gioco. Vi sono molti casi in cui l'ingegnere software e il cliente devono negoziare funzioni e caratteristiche, priorità e date di consegna. Se il team ha collaborato bene, tutte le parti avranno un obiettivo comune, pertanto la negoziazione richiede che entrambe le parti siano disposte a scendere a compromessi.

Pratiche di pianificazione

L'attività di pianificazione comprende un insieme di pratiche gestionali e tecniche che consentono al team di sviluppo di definire un percorso che consenta di raggiungere i suoi obiettivi strategici e tattici.

Essa comprende la gestione delle risorse umane e dei costi, e l'individuazione dello scheduling (lista) necessario per produrre i risultati sperati.

La pianificazione (compreso il monitoring) è probabilmente l'attività che prende più tempo. È continuativa dall'inizio alla consegna del prodotto software, e i piani devono essere revisionati (arricchiti di dettagli, corretti, aggiornati) quando nuove informazioni diventano disponibili

Indipendentemente dal rigore con cui viene condotta la fase di pianificazione, devono sempre essere applicati i seguenti principi.

1. Comprendere l'ampiezza del progetto. Capire qual è la destinazione.
2. Coinvolgere il cliente nella pianificazione. Tramite la negoziazione.
3. Riconoscere che la pianificazione è iterativa. Il piano dovrà essere predisposto in modo da considerare cambiamenti.
4. Effettuare stime sulla base di ciò che si conosce. Identificare sforzo, costo e durata del lavoro da svolgere.
5. Considerare i rischi durante la definizione del piano. Sviluppando un eventuale piano d'emergenza.
6. Essere realistici. Il team non può lavorare costantemente al 100%.
7. Adattare la granularità durante la definizione del piano. La **Granularità fine** presenta numerosi dettagli di lavoro pianificati sulla base di incrementi temporali relativamente brevi. La **Granularità ampia** fornisce i compiti generali più ampi pianificati su periodi temporali lunghi. In generale la granularità procede da quella più fine a quella più ampia a mano a mano che il progetto si allontana dalla data corrente.
8. Definire come si intende garantire la qualità. Se devono essere condotte delle revisioni tecniche formali occorre programmarle.
9. Descrivere come si intendono considerare le modifiche. Identificare il modo in cui possono essere considerate le modifiche mentre procede il lavoro.
10. Monitorare frequentemente il piano e apportare le modifiche necessarie. Ha senso verificare quotidianamente i progressi, ricercando le situazioni problematiche e le situazioni in cui il lavoro programmato non coincide con il lavoro condotto. Questo porta dei ritardi, e il piano deve essere regolato di conseguenza.

Organizzazione attività

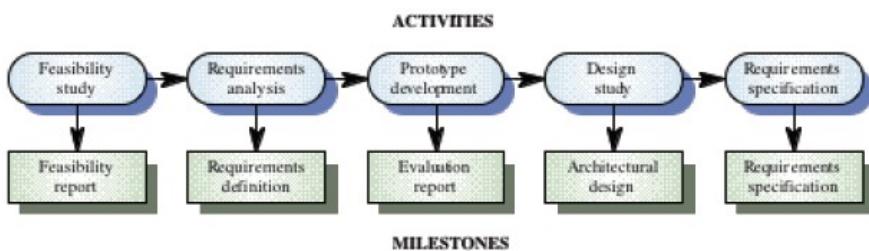
Le attività del progetto dovrebbero essere organizzate per produrre risultati tangibili che i manager possono esaminare per stabilire i progressi raggiunti

- **Milestone:** sono il punto finale di un'attività del processo software
- **Deliverable:** sono i risultati del progetto che sono consegnati ai clienti

Milestone (pietra miliare)

Indica importanti traguardi intermedi nello svolgimento del progetto. Molto spesso sono rappresentate da eventi, cioè da attività con durata zero o di un giorno, e vengono evidenziate in maniera diversa dalle altre attività nell'ambito dei documenti di progetto. Esempi di milestones sono: la fine dei collaudi di un impianto, la firma di un contratto, il varo di un nave, la fine delle opere di muratura di un edificio, eccetera.

Il termine milestone viene tipicamente utilizzato nella pianificazione e gestione di progetti complessi per indicare il raggiungimento di obiettivi stabiliti in fase di definizione del progetto stesso.



Il processo a cascata permette direttamente la definizione di milestone. A sinistra un esempio di attività e milestones nella fase di specifica dei requisiti.

Pianificazione temporale

Come per le altre aree dell'ingegneria del software, anche per la pianificazione temporale si possono descrivere alcuni principi fondamentali.

- ❖ Ripartizione. Un progetto deve essere ripartito in attività e compiti di dimensione ragionevoli. A tale scopo si scompongono sia il prodotto sia il progetto.
- ❖ Dipendenze. Occorre determinare le dipendenze reciproche fra le attività e i compiti in cui è stato ripartito il progetto. Alcuni compiti devono essere svolti in sequenza, altri in parallelo. Dunque alcune attività non possono cominciare finché non è disponibile un semilavorato di un'altra attività.
- ❖ Assegnazione del tempo. A ogni compito si deve assegnare un certo numero di "unità di lavoro" (per esempio giorni-uomo). A ogni compito si attribuiscono anche la data di inizio e data di fine, tempo pieno o parziale, con cui si è svolto.
- ❖ Convalida dell'assegnazione. A ogni progetto è assegnato un numero definito di membri nello staff. Al momento dell'assegnazione dei tempi, il capo progetto deve accertarsi di non attribuire più persone di quelle disponibili.
- ❖ Responsabilità definite. Ogni compito deve essere affidato a un membro del team
- ❖ Risultati definiti. Ogni compito deve produrre un risultato predefinito.
- ❖ Punti di controllo. A ogni compito si deve associare un punto di controllo del progetto.

Scheduling del progetto

È necessario dividere il progetto in **task** e stimare tempo e risorse necessarie a completare ciascun task, ovvero individuare i task con una granularità opportuna.

Ogni task dovrebbe durare almeno 1 settimana e non dovrebbe superare le 10 settimane

Es. T1: requisiti modulo 1; T5: Interfacciare moduli, etc.

Bisogna inoltre organizzare i task in modo concorrente per fare un uso ottimale della forza lavoro, utilizzare tutto il personale a disposizione, e minimizzare le dipendenze tra task per evitare ritardi a cascata dovuti ad un task che aspetta il completamento di un altro task in ritardo.

Lo scheduling dipende dall'intuizione e dall'esperienza dei manager del progetto.

Individuazione dei rischi

L'individuazione dei rischi è il tentativo sistematico di specificare le minacce al piano di un progetto. L'individuazione dei rischi noti e prevedibili è il primo passo verso la loro prevenzione quando possibile, e la loro gestione quando necessario. Possiamo distinguere fra:

- *Rischi generici*. Comuni a qualunque progetto software;
- *Rischi specifici*. Si possono individuare solo a patto di avere una visione chiara della tecnologia, delle persone e dell'ambiente specifici del progetto in esame.

Anche se i rischi generici sono importanti, normalmente i rischi specifici sono quelli che provocano più problemi.

Al fine di individuare i rischi è utile stilare un catalogo dei rischi, suddiviso in varie categorie di rischi noti e prevedibili:

- o Dimensione del prodotto
- o Effetti commerciali
- o Caratteristiche del cliente
- o Definizione del processo
- o Ambiente di sviluppo
- o Tecnologia
- o Dimensione ed esperienza dello staff

È difficile valutare la difficoltà dei problemi e quindi il costo per lo sviluppo di una soluzione. La produttività non è proporzionale al numero di persone che lavorano per un task, aggiungere persone ad un progetto in ritardo fa aumentare il ritardo, a causa della comunicazione necessaria. Qualcosa di inaspettato può capitare, bisogna prevedere un piano per le emergenze.

È possibile stimare la durata di un progetto:

- Ricorrendo all'esperienza (propria o altrui)
- Facendo affidamento che niente vada male
- Aggiungendo un 30% per i problemi che possono essere intravisti
- Aggiungendo un ulteriore 20% per tener conto di ciò che non è stato immaginato

La **stima dei rischi** ha lo scopo di quantificare due aspetti di ogni rischio: la probabilità che il rischio sia reale e le conseguenze dei problemi che sorgono qualora il rischio si realizzi. Il pianificatore svolge quattro operazioni relative alla proiezione dei rischi:

1. Stabilire una scala che riflette la probabilità presunta di un rischio
2. Delineare le conseguenze del rischi
3. Stimare gli effetti del rischio sul progetto e sul prodotto
4. Indicare l'accuratezza globale della proiezione, così che non sorgano malintesi

Lo scopo di questi passi è quello di considerare i rischi in modo da stabilirne la priorità.

Gestione personale (stuff)

Non sempre è possibile avere le persone ideali per lavorare su un progetto; il budget potrebbe non consentire di usare il personale molto costoso o il personale con l'esperienza appropriata potrebbe non essere disponibile ma impegnata per altri progetti. Comunque un'organizzazione potrebbe voler formare del personale lavorando su un progetto.

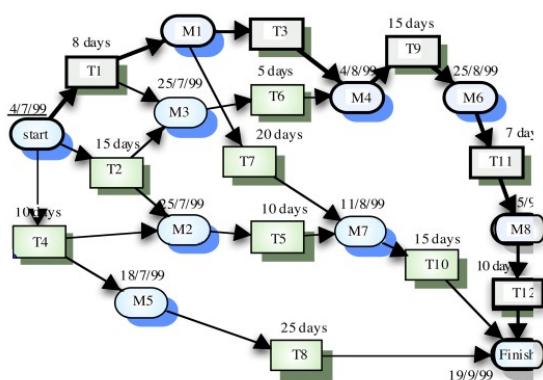
Rete di compiti o delle attività

Quando il progetto comprende più persone e ci sono dipendenze reciproche, occorre coordinare i compiti svolti simultaneamente affinché siano completati nel momento in cui i compiti successivi ne richiedono i risultati. Una *rete dei compiti o delle attività* è una rappresentazione grafica della successione dei compiti in un progetto. Essa prevede l'uso di notazioni grafiche per illustrare lo scheduling del progetto, mostrare le divisioni in task (i task non dovrebbero essere troppo piccoli, la loro durata dovrebbe essere di una o due settimane), mostrare le dipendenze tra i task ed i **percorsi critici** (cioè quei compiti che devono rispettare i tempi previsti affinché il progetto in generale possa rispettare la propria scadenza); i diagrammi a barre invece mostrano lo scheduling su un calendario.

Poiché i compiti svolti in parallelo non sono sincronizzati, il pianificatore deve determinare le relazioni di dipendenza per coordinare l'avanzamento del progetto.

La tabella a destra mostra la durata e dipendenza dei task di un progetto software.

Grafo delle attività PERT



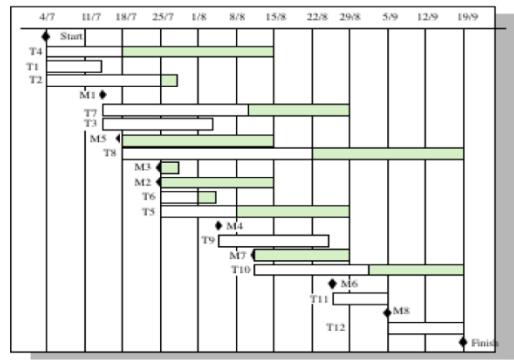
Il PERT (Program Evaluation and Review Technique, tecnica di valutazione e revisione dei piani) è un esempio di grafo delle attività dove i *nodi* sono le attività o milestone, mentre gli *archi* sono le dipendenze fra attività.

Il grafo fornisce strumenti quantitativi che consentono al pianificatore di determinare il **percorso critico**, che è il tempo minimo richiesto per completare il progetto, ed è dato dal più lungo percorso del grafo delle attività. Il grafo permette anche di stabilire le stime più probabili dei tempi necessari al completamento di

ogni singolo compito e di calcolare i limiti temporali che definiscono la finestra temporale entro cui ciascun compito deve essere iniziato e terminato. Ritardi nei task che non fanno parte del percorso critico non causano ritardi globali, a patto che il percorso critico non cambi.

Diagramma di GANTT

Un diagramma di Gantt permette la rappresentazione grafica di un **calendario di attività**, utile al fine di pianificare, coordinare e tracciare specifiche attività in un progetto dando una chiara illustrazione dello stato d'avanzamento del progetto rappresentato; di contro, uno degli aspetti non tenuti in considerazione in questo tipo di diagrammazione è le dipendenze delle attività, caratteristica invece della programmazione reticolare, cioè del diagramma PERT. Ad ogni attività possono essere in generale associati una serie di attributi: durata (o data di inizio e fine), predecessori, risorsa, costo.



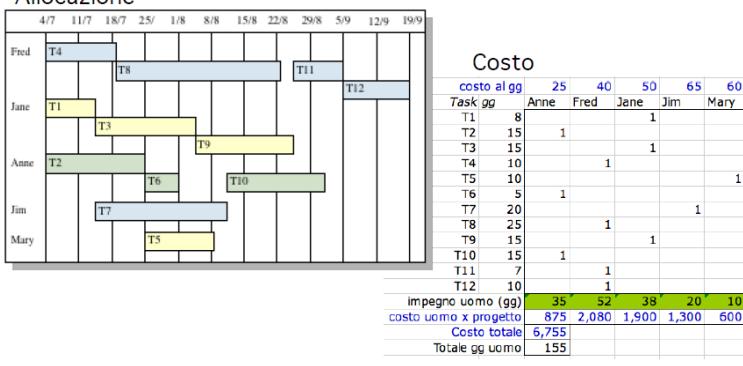
Ad ogni attività possono essere associate una o più risorse. Alcuni software disponibili sul mercato permettono di visualizzare il carico di lavoro di ogni risorsa e la sua saturazione, impostando una certa disponibilità per ogni risorsa. Contestualmente, può essere definito il calendario dei giorni lavorativi e festivi, e il numero di ore di lavoro giornaliero.

Ad ogni attività può poi essere associato un costo. Il costo può essere attribuito a una singola attività oppure si può assegnare un costo orario alle risorse, determinando il costo dell'attività in base al relativo impegno orario. Nel corso del progetto, ad ogni attività o alla risorsa può essere attribuito un costo effettivo.

Allocazione e costo del personale

Per l'allocazione del nostro personale a disposizione bisogna tener conto delle capacità individuali. Ovviamente più esperienza richiede il task da eseguire più il costo del dipendente aumenta.

Allocazione



Personale	Specialità	Task assegnati	Stipendio
Saro	Req., Progett.	T3	115€*4h*3d
Tim	Req.	T1, T4	30€*8h*7d
Claire	Req.	T2	15€*8h*6d
Mario	Test, Progett.	T4	Volontario
Dan	Progr.	T2	10€*8h*6d
Michele	Pulizie	T?	0,50€*24h*7d
Antonio	Porta caffè	T*	0,30*1h*7d

Ingegneria dei Requisiti

L'ingegneria dei requisiti aiuta gli ingegneri software a capire meglio il problema che stanno cercando di risolvere. Comprende un insieme di compiti che conducono alla comprensione dell'impatto sul cliente del software, di ciò che i clienti desiderano e del modo in cui gli utenti finali interagiscono col software.

Fasi

Essa inizia con la fase di **avvio**, la quale definisce l'ampiezza e la natura del problema da risolvere; procede con la fase di **deduzione**, un compito che aiuta il cliente a definire ciò di cui vi è bisogno; e poi con la fase di **elaborazione**, dove i requisiti di base vengono raffinati e modificati. A mano a mano che il cliente definisce il problema, si verifica una negoziazione su quali sono le priorità, cosa è fondamentale, quali sono i tempi, ed infine il problema viene specificato, riveduto e convalidato per garantire che la propria conoscenza del problema coincida con la comprensione del problema da parte del cliente.

Scopo

Lo scopo del processo di ingegneria dei requisiti è quello di fornire a tutte le parti coinvolte una **documentazione** scritta del problema. Ciò può essere ottenuto in vari modi, utilizzando scenari utente, funzioni ed elenchi di funzioni caratteristiche, modelli analitici o specifiche.

Un ponte verso progettazione e costruzione

L'ingegneria dei requisiti, come ogni altra attività dell'ingegneria del software, deve essere adattata ai bisogni del processo, del progetto e delle persone che svolgono il lavoro. Essa inizia durante l'attività di comunicazione e procede nell'attività di modellazione. È fondamentale che il team di sviluppo software faccia uno sforzo reale per comprendere i requisiti di un problema *prima* di tentare di risolverlo.

Essa rappresenta un ponte vero la progettazione e costruzione del software che inizia dove si trovano coloro che sono coinvolti nel progetto nel luogo in cui vengono definiti i bisogni, gli scenari utente, le funzioni, le caratteristiche, e i vincoli del progetto. L'attraversamento del ponte porta ad osservare il progetto dall'alto, consentendo al team di sviluppo di avere una visione generale del progetto e di esaminare il contesto del lavoro che deve essere realizzato, i bisogni che devono essere risolti dalla progettazione e dalla costruzione, l'ordine in cui deve essere completato il lavoro e infine le informazioni, funzioni e comportamenti che avranno un impatto profondo sul progetto finale.

Requisiti

Un requisito è una descrizione di un servizio del sistema. Sono le funzionalità astratte che il sistema deve fornire, le proprietà del sistema e le caratteristiche che non bisogna esibire.

Definizione e caratteristiche dei requisiti

Un requisito è una capacità del software che l'utente necessita per risolvere un problema o per ottenere un risultato. È anche una capacità che il software deve avere per soddisfare un contratto, uno standard, una specifica.

Un requisito va da una descrizione con alto livello di astrazione di un servizio o di un vincolo ad una specifica funzionale dettagliata matematicamente.

I requisiti possono servire:

1. Come base per una offerta per un contratto: devono essere sufficientemente astratti per non indicare una soluzione predefinita;
2. Come base per il contratto: devono essere sufficientemente dettagliati
3. Per descrivere ciò che è richiesto agli sviluppatori: devono essere molto dettagliati

I requisiti devono essere **completi** e **consistenti**

- *Completi*: includere la descrizione di tutto ciò che è richiesto
- *Consistenti*: non ci devono essere contraddizioni nella loro descrizione

Nella pratica è difficile ottenere un documento con entrambe le caratteristiche

Requisiti funzionali e non-funzionali

- **Requisiti Funzionali.** Descrivono le funzionalità ed i servizi principali forniti dal sistema.
- **Requisiti Non-Funzionali.** Descrivono funzioni che non sono collegate direttamente con le funzioni implementate dal sistema, ma piuttosto alle modalità operative, di gestione, ecc. Definiscono vincoli sullo sviluppo del sistema.

Es. Servizio bancomat, requisiti funzionali: Il servizio deve mettere a disposizione le funzioni di prelievo, saldo, estratto conto; Le operazioni di prelievo devono richiedere autenticazione tramite un codice segreto memorizzato sulla carta.

Servizio bancomat, requisiti non-funzionali: Il sistema deve essere disponibile a persone portatori di Handicap, deve garantire un tempo di risposta inferiore al minuto, e deve essere sviluppato su architettura X86 con sistema operativo compatibile con quello della Banca. Il sistema deve essere facilmente espandibile, e adattabile alle future esigenze bancarie.

I requisiti non-funzionali possono essere più critici di quelli funzionali, poiché se non sono soddisfatti, il sistema è inutile

Classificazione requisiti

- o **Requisiti di prodotto.** Specificano il comportamento del prodotto, es. velocità, affidabilità, etc.
- Es.* I dati comunicati dovranno essere rappresentati in ASCII
- o **Requisiti organizzativi.** Quelli derivanti da prassi organizzative, es. metodi di design o linguaggi usati, etc.
- Es.* I documenti deliverable dovranno essere conformi allo standard XYZ
- o **Requisiti esterni.** Derivanti da fattori esterni, es. interoperabilità con altri sistemi, aderenza alle leggi, fattori etici, etc.
- Es.* Il sistema non dovrà rivelare agli operatori informazioni personali dei clienti, eccetto il loro nome.

Metriche per i requisiti

Proprietà	Misure
Velocità	Transazioni elaborate al secondo Tempo di risposta Tempo di refresh dello schermo
Dimensione	KB Numero di chip di RAM
Facilità d'uso	Tempo per il training del personale Numero di finestre di aiuto
Affidabilità	Tempo medio di un guasto (MTTF) Probabilità di non disponibilità Rate di occorrenza dei guasti
Robustezza	Tempo necessario a riavviare dopo un guasto Percentuale di eventi che causano guasti Probabilità di corruzione dati a causa di un guasto
Portabilità	Percentuale di istruzioni dipendenti dalla piattaforma Numero di piattaforme supportate

Tipi di requisiti

Abbiamo fondamentalmente tre tipi di requisiti:

- **Requisiti utente:** Scritto per i clienti, è un insieme di descrizioni in linguaggio naturale e diagrammi dei servizi che il sistema fornisce.

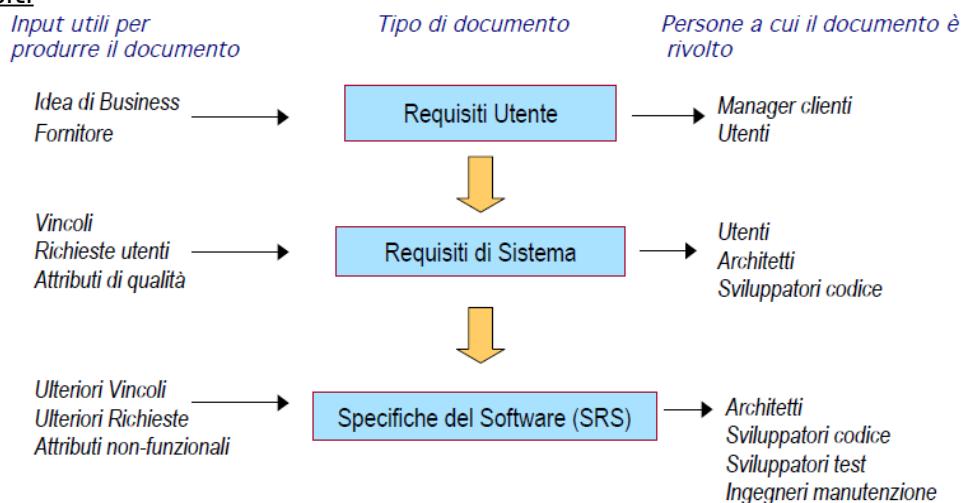
Esempio: Il software dovrà fornire il modo per rappresentare ed accedere a file esterni creati con altri tools.

- **Requisiti di sistema:** Scritto come contratto tra cliente e fornitore, è un documento strutturato che dettaglia i servizi del sistema.

Esempio: L'utente dovrebbe essere fornito di strumenti per definire il tipo di file esterni. Ogni tipo di file esterno può avere associato un tool che può essere applicato al file. Ogni tipo di file esterno può essere rappresentato da una icona specifica. La selezione dell'icona provoca l'applicazione del tool al file rappresentato.

- **Specifiche del software (SRS: Software Requirements Specification):** Scritto per gli sviluppatori, è un documento che descrive dettagliatamente il software, e serve come base per il design o l'implementazione.

Livelli di requisiti



Requisiti utente

Dovrebbero descrivere requisiti funzionali e non-funzionali in modo comprensibile per chi non ha conoscenze tecniche dettagliate. Sono definiti usando il **linguaggio naturale**, tabelle e diagrammi.

Vi sono comunque dei problemi con il linguaggio naturale:

- Mancanza di chiarezza, la precisione è difficile senza rendere il documento difficile da leggere;
- Confusione, requisiti funzionali e non-funzionali tendono a mischiarsi;
- Amalgamazione, differenti requisiti possono essere espressi insieme.

Esempio:

- Requisito per un database
 - Il database supporterà la generazione ed il controllo di oggetti di configurazione, ovvero gli oggetti sono essi stessi raggruppamenti di altri oggetti del database. Gli strumenti di configurazione permetteranno l'accesso agli oggetti in un gruppo per mezzo di un nome incompleto.
- Requisito per un tool per il design
 - Per assistere il posizionamento di entità in un diagramma, l'utente può attivare una griglia in centimetri o pollici, attraverso una opzione sul pannello di controllo. Inizialmente la griglia è disattivata. La griglia può essere attivata o disattivata in qualunque momento durante una sessione di editing e cambiata da centimetri a pollici in qualunque momento. Una opzione per la griglia sarà fornita sulla vista "adatta" ma il numero di linee della griglia mostrate sarà ridotto per evitare il riempimento del diagramma con tali linee.

Dettaglio per SRS

Mischia requisiti funzionali
Con dettagli non funzionali

Linee guida per scrivere requisiti

Bisogna scegliere un formato *standard* e usarlo per tutti i requisiti tra due tipi di linguaggio:

- o La notazione tramite **Linguaggio Naturale (NL)** usa il linguaggio in modo consistente, evidenzia le parti importanti dei requisiti (corsivo, grassetto, etc.), e per i requisiti utente evita il gergo informatico.

Esempio. **Dovrà** (per il necessario) o **dovrebbe** (per ciò che è preferibile o gradito avere, una feature sacrificabile)

- o La notazione tramite **Linguaggio Naturale Strutturato (SNL)**: descrive ciascun requisito seguendo un formato standard con voci fisse, come:
 - Nome funzione
 - Descrizione
 - Input, output
 - Pre-condizioni (indicano cosa deve essere verificato per poter eseguire la funzione)
 - Post-condizioni (indicano cosa è verificato se tutto è andato bene)

Requisiti di sistema

Sono più dettagliati dei requisiti utente, ma non dovrebbero imporre scelte di design e implementazione. Servono come base per il design e hanno possibili notazioni per la scrittura dei requisiti:

- **Linguaggio naturale (NL)**: valgono le linee guida precedenti
- **Linguaggio naturale strutturato (SNL)**: definisce un formato standard o template per esprimere specifiche
- **Program (o Design) Description Language (PDL)**: usa un linguaggio simile ad un linguaggio di programmazione ma più astratto per definire le specifiche e le modalità operative del prodotto (pseudocodice)
- **Notazioni grafiche**: casi d'uso UML
- **Formulazione matematica**: linguaggi formali, macchine a stati finiti

Specifiche del software (SRS)

Describe cosa è richiesto agli sviluppatori. Non è un documento di design, ma dice **cosa** il sistema dovrebbe fare e non *come* lo dovrebbe fare. La struttura (o template) è stata suggerita dall' IEEE standard 830 1984, ed è:

1. Introduzione

Scopo dell'SRS, ambiente del prodotto (utenti, clienti, sviluppatori), definizioni acronimi abbreviazioni, overview.

2. Descrizione generale

Scopi del prodotto, funzioni del prodotto, caratteristiche (problemi, obiettivi) utenti, vincoli (protocolli, hardware, etc.).

3. Requisiti specifici

Usa un formato standard per descrivere ciascun requisito funzionale; Nome, Descrizione, Criticalità (vedi Quality Function Deployment); Elementi tecnici utili per soddisfare il requisito, Costo, Rischi, Dipendenze.

4. Requisiti di interfaccia

Interfaccia utente (GUI, CLI, API), interfacce hardware, comunicazione in rete, interfacce con altri software

5. Requisiti di performance

6. Vincoli di design

Aderenza a standard, limiti hardware, etc.

7. Attributi non-funzionali

Sicurezza, affidabilità, manutenzione, etc.

8. Scenari Operativi

Casi d'uso

9. Piano di progetto preliminare

10. Appendici

Definizioni, riferimenti

Dipendenze tra requisiti

Le dipendenze tra i requisiti possono essere descritte tramite le tabelle di tracciabilità. Una tabella di tracciabilità mette in relazione i requisiti identificati con uno o più aspetti del sistema o del suo ambiente. Fra le tante possibili tabelle di tracciabilità vi sono le seguenti:

- **Tabella di tracciabilità delle funzionalità**. Mostra le relazioni fra requisiti e funzionalità del sistema più importanti osservabili dal cliente.
- **Tabella di tracciabilità dell'origine**. Identifica l'origine di ciascun requisito.
- **Tabella di tracciabilità delle dipendenze**. Indica le relazioni esistenti fra i requisiti.
- **Tabella di tracciabilità dei sottosistemi**. Suddivide i requisiti per categorie, in base ai sottosistemi che li governano.
- **Tabella di tracciabilità dell'interfaccia**. Mostra le relazioni fra i requisiti e le interfacce interne ed esterne del sistema.

In molti casi queste tabelle di tracciabilità vengono gestite all'interno di un database dei

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1	U	R						
1.2	U			R				U
1.3	R			R				
2.1		R		U				U
2.2								U
2.3	R		U					
3.1							R	
3.2							R	

U = usato, es. 1.1 usa 1.2

R = relazione

requisiti, in modo da agevolarne la ricerca per comprendere come una modifica in un requisito possa influenzare i vari aspetti del sistema che deve essere costruito.

Processo dell'ingegneria dei requisiti

Tutte le funzioni principali dell'ingegneria del software (comprendere, analisi, valutazione e negoziazione dei bisogni del cliente) si verificano in parallelo e vengono adattate ai bisogni del progetto. Tutte mirano a definire ciò che il cliente desidera e tutte servono per costruire solide basi per la progettazione e realizzazione di ciò che otterrà il cliente. Il processo di ingegneria dei requisiti viene ottenuto tramite l'esecuzione di sette diverse funzioni: *avvio, deduzione, elaborazione, negoziazione, specifica, convalida e gestione*.

Avvio

La maggior parte dei progetti inizia quando viene identificato un bisogno o quando viene scoperto un nuovo mercato potenziale o servizio. Tutte le figure coinvolte definiscono un caso per l'idea, tentano di identificare l'ampiezza e la profondità del mercato, svolgono un'analisi approssimativa della fattibilità e identificano una descrizione dell'ampiezza del progetto.

Deduzione

Si delineano gli obiettivi del sistema, cosa deve essere ottenuto e in quale modo il sistema o il prodotto risponde ai bisogni stabiliti. Eppure sorgono numerose difficoltà nella fase di deduzione quali:

- *Problemi di ampiezza.* Limiti del sistema mal definiti o il cliente specifica dettagli tecnici non necessari che possono confondere invece di chiarire gli obiettivi generali del sistema.
- *Problemi di comprensione.* Il cliente non è completamente sicuro di ciò di cui ha bisogno, ha una vaga idea dei limiti dell'ambiente di calcolo, ha problemi a comunicare i bisogni, omette informazioni ovvie e specificano requisiti ambigui.
- *Problemi di volatilità.* I requisiti cambiano nel corso del tempo.

Per risolvere questi problemi, gli ingegneri dei requisiti devono avvicinarsi all'attività di raccolta dei requisiti in modo organizzato.

Elaborazione

Attività che si concentra sullo sviluppo di un modello tecnico raffinato delle funzioni, delle caratteristiche e dei vincoli del software. Essa è un'azione di modellazione dell'analisi. È guidata dalla creazione e dal raffinamento degli scenari utente che descrivono il modo in cui l'utente finale interagisce con il sistema.

Ogni scenario viene analizzato per estrarre le classi dell'analisi e ne vengono definiti gli attributi per ciascuna classe di analisi, e poi vengono identificati i servizi richiesti da ciascuna classe. Vengono identificate le relazioni e le collaborazioni fra classi e vengono prodotti vari diagrammi UML. Il risultato finale dell'elaborazione è un **modello analitico** che definisce il dominio informativo, funzionale e comportamentale del problema.

Negoziazione

Molto spesso i clienti richiedono più di quanto può essere ottenuto in base alle risorse disponibili o propongono requisiti in conflitto; l'ingegnere dei requisiti deve riconciliare questi conflitti tramite un processo di negoziazione.

I clienti e le altre figure coinvolte devono valutare i requisiti, discuterne le priorità, identificare e analizzare i rischi associati a ciascun requisito ed effettuare delle stime dell'attività di sviluppo. I requisiti vengono così eliminati, combinati e/o modificati in modo che ognuna delle parti venga il più possibile soddisfatta.

Specifiche

Un documento scritto che combina descrizioni in linguaggio naturale e modelli grafici. Le specifiche rappresentano il risultato finale del prodotto dell'ingegneria dei requisiti. Sono gli elementi di base per le successive attività dell'ingegneria del software. Esse descrivono la funzione e le prestazioni di un sistema computerizzato, e i vincoli che governano il suo sviluppo.

Convalida

Viene valutata la qualità del prodotto finale realizzato come conseguenza dell'ingegneria dei requisiti. Essa esamina le specifiche per garantirsi che tutti i requisiti software siano stati

affermati in modo non ambiguo; che siano state rilevate e corrette tutte le incoerenze, le omissioni e gli errori e che i prodotti finali siano conformi agli standard definiti per il processo, progetto e il prodotto.

Gestione dei requisiti

La gestione dei requisiti (*requirement management*) è il processo di gestione dei cambiamenti durante lo sviluppo del sistema. Esso è l'insieme di attività che aiuta il team del progetto a identificare, controllare e tracciare i requisiti e i cambiamenti a mano a mano che procede lo sviluppo del progetto.

La gestione requisiti consiste in un approccio per l'estrazione, l'organizzazione e la documentazione dei requisiti di un sistema, e nel processo che stabilisce e mantiene l'accordo tra il cliente ed il team del progetto.

Inevitabilmente i requisiti sono incompleti poiché nuovi requisiti emergono mano a mano che aumenta la comprensione del sistema; la priorità dei requisiti cambia durante lo sviluppo; l'ambiente del sistema (affari, leggi, tecnologie) cambia durante lo sviluppo.

Piano di gestione dei requisiti

- Identificare requisiti (con un id)
- Prevedere un processo di cambiamento dei requisiti a seguito di un'analisi e della rilevazione di un cambiamento
- Politiche di tracciabilità che indicano le informazioni mantenute sulle relazioni tra requisiti
- Tool di supporto (CASE: IBM Requisite Pro, etc.) che permettono registrazione dei requisiti, gestione delle informazioni in caso di cambiamenti e tracciamento automatico dei requisiti.
- Uso di **Modelli** per lo sviluppo del sistema, tali modelli danno una descrizione astratta del sistema i cui requisiti sono analizzati. La modellazione aiuta l'analista a capire le funzionalità del sistema. Differenti modelli presentano il sistema da prospettive diverse
 - Prospettive esterne mostrano il contesto del sistema, tracciano i confini del sistema;
 - Prospettive comportamentali mostrano il comportamento del sistema
 - Prospettive strutturali mostrano il sistema o l'architettura dei dati

I tipi di modelli realizzabili tramite **UML** sono:

- Comportamentali
 - Modello per l'elaborazione dei dati (data-flow diagram): mostra come i dati vengono trasformati in diversi passi
 - Modello di stimoli/risposte: mostra la reazione del sistema agli eventi interni ed esterni
- Strutturali
 - Modello architettonico: mostra i principali sottosistemi
 - Modello di composizione (entità-relazione): mostra come le entità sono composizioni di altre entità
 - Modello di classificazione (ereditarietà): mostra in che modo le entità hanno caratteristiche comuni.

Quality Function Deployment

La tecnica QFD traduce i bisogni del cliente in requisiti tecnici per il software. Questa tecnica si concentra sulla massimizzazione della soddisfazione del cliente nei confronti del processo di ingegneria del software. Per ottenere ciò, la tecnica QFD pone l'accento sulla comprensione di ciò che è prezioso per il cliente e poi implementa tali valori tramite il processo di ingegneria. Tale tecnica identifica 3 requisiti:

Requisiti normali

Riflettono gli obiettivi affermati per il prodotto durante la riunione con il cliente. Se questi requisiti sono presenti, il cliente è soddisfatto.

Es. Visualizzazioni grafiche, specifiche funzioni e specifici livelli prestazionali.

Requisiti attesi

Requisiti impliciti nel prodotto talmente fondamentali che il cliente potrebbe perfino evitare di affermare. La loro assenza può causare insoddisfazioni.

Es. Iterazioni uomo/macchina, corretta e affidabilità generale, facilità di installazione del software.

Requisiti interessanti

Requisiti che riflettono funzionalità che vanno oltre le attese del cliente ma possono essere di notevole soddisfazione se presenti.

Es. Modalità di impaginazione particolari di un programma di videoscrittura.

Dispiegamento

Per stabilire il valore di ciascuna funzione necessaria per il sistema, nelle riunioni con il cliente viene utilizzato un *dispiegamento* delle funzioni.

- o *Dispiegamento delle informazioni*: identifica i dati e gli eventi che il sistema deve consumare e produrre.
- o *Dispiegamento dei compiti*: esamina il comportamento del sistema o del prodotto nel contesto del proprio ambiente.

Viene poi condotta un'analisi del valore per determinare la priorità relativa dei requisiti determinanti durante ognuna di queste fasi.

Tabella della voce del cliente

La tecnica QFD impiega le interviste, osservazioni, indagini e l'esame dei dati storici del cliente come materia prima per l'attività di raccolta dei requisiti. Questi dati vengono poi tradotti in una tabella dei requisiti, chiamata *tabella della voce del cliente*, che viene rielaborata insieme al cliente stesso.

Gli scenari utente

A mano a mano che vengono raccolti i requisiti, inizia a formarsi una visione globale delle funzioni e delle caratteristiche del sistema. Tuttavia è difficile procedere nelle attività più tecniche dell'ingegneria del software fino a quando il team di sviluppo non comprende il modo in cui queste funzioni caratteristiche verranno utilizzate dalle varie classi di utenti finali. Per ottenere ciò, gli sviluppatori e gli utenti possono creare un insieme di scenari che identifica uno schema d'uso del sistema che deve essere costruito. Gli scenari (chiamati anche *casi d'uso*) forniscono una descrizione del modo in cui il sistema verrà utilizzato.

Scoperta dei requisiti

I prodotti da realizzare come conseguenza della **fase di deduzione dei requisiti** variano a seconda delle dimensioni del sistema software da realizzare. Questi prodotti includono i seguenti elementi:

- Affermazione del bisogno e della fattibilità
- Affermazione dell'ampiezza del sistema o del prodotto
- Elenco di clienti, utenti e altre figure che hanno partecipato alla scelta dei requisiti
- Descrizione dell'ambiente tecnico del sistema
- Elenco di requisiti organizzati per funzioni e dei vincoli di dominio applicabili ad ognuno di essi
- Insieme di scenari d'uso che forniscono i dettagli d'uso del sistema o del prodotto nelle varie condizioni operative
- Ogni prototipo sviluppato per definire meglio i requisiti

Ognuno di questi prodotti viene riveduto da tutti coloro che hanno partecipato alla fase d'individuazione dei requisiti.

Studio di fattibilità

Decide se costruire il sistema è di interesse sulla base dei contributi agli obiettivi dell'organizzazione, dell'ingegnerizzare con la tecnologia corrente entro il budget e dell'integrazione con altri sistemi usati. La raccolta delle informazioni si pone le seguenti domande:

- o Cosa succede se il sistema non venisse implementato?
- o Come aiuterà il sistema proposto?

- o Quali tecnologie/abilità richiederà la sua costruzione?

Analisi dei requisiti

Richiede colloqui con i clienti per ricavare ciò che il sistema deve fornire ed i suoi vincoli operazionali. Le attività sono:

- Comprensione del dominio (come funziona un aereo?)
- Collezione dei requisiti
- Classificazione in gruppi coerenti
- Risoluzione conflitti tra requisiti
- Stabilire priorità
- Controllare (revisionare) requisiti per determinare completezza e consistenza con ciò che utenti, clienti, etc. (stakeholder) vogliono.

Requisiti di negoziazione

Il cliente e lo sviluppatore entrano in un processo di negoziazione dove al cliente può essere chiesto di valutare dei compromessi fra funzionalità, prestazioni e altre caratteristiche del prodotto o del sistema da un lato e costi o tempi di uscita sul mercato dall'altro. Lo scopo di questa negoziazione è quello di sviluppare un piano per il progetto che corrisponda ai bisogni del cliente riflettendo allo stesso tempo i vincoli del mondo reale (tempo, persone, budget) che sono stati imposti al team di sviluppo software.

Le migliori negoziazioni cercano di ottenere vantaggi per tutte le parti; il cliente ha un vantaggio ottenendo un sistema che soddisfa la maggior parte dei suoi bisogni, e il team di sviluppo ha un vantaggio potendo lavorare su un budget e scadenze realistiche e ottenibili.

Fattori sociali

I sistemi software sono usati in un contesto sociale ed in una organizzazione. I **fattori sociali** possono influenzare o persino dominare i requisiti. L'analista deve saper percepire tali fattori, ma al momento non ci sono metodi sistematici a supporto della loro analisi.

Es. Consideriamo un sistema che permette ai senior manager di accedere informazioni senza chiedere ai manager di livello medio.

- o *Status.* I senior manager possono credere di essere troppo importanti per usare la tastiera (Questo può limitare il tipo di interfaccia usato)
- o *Responsabilità.* I senior manager potrebbero non aver il tempo di imparare come usare il sistema
- o *Resistenza.* I manager di livello medio che diventano ridondanti, possono fornire informazioni incomplete o non esatte per far fallire il sistema.

Requisiti di convalida

La convalida dei requisiti ha lo scopo di mostrare che i requisiti definiscono il sistema che il cliente vuole.

A mano a mano che viene creato ogni elemento del modello analitico, occorre esaminare la sua coerenza, eventuali omissioni e l'esistenza di ambiguità. I requisiti rappresentati dal modello vengono classificati per priorità dal cliente e vengono raggruppati nei pacchetti di requisiti che verranno implementati mano a mano che al cliente verranno forniti gli incrementi del software.

Gli *errori* nei requisiti costano e quindi la convalida è importante. Aggiustare un errore nei requisiti dopo la consegna del sistema costa fino a 100 volte il costo di aggiustare errore nell'implementazione.

I fattori che incidono nella convalida dei requisiti sono:

- *Validità.* Il sistema fornisce le funzioni che meglio supportano le necessità delle varie classi di utenti?
- *Consistenza.* Ci sono conflitti tra i requisiti, o descrizioni differenti della stessa funzione?
- *Completezza.* Sono incluse tutte le funzioni richieste dagli utenti?
- *Realismo.* I requisiti possono essere implementati con il budget, le tecnologie e nel tempo a disposizione?

- **Verificabilità.** Sarà possibile mostrare che il sistema soddisfa i requisiti?

Tecniche dei convalida requisiti

- o *Revisione requisiti.* Analisi sistematica dei requisiti per scoprire anomalie e omissioni.
- o *Prototipazione.* Uso di un modello (prototipo) del sistema per verificare i requisiti (isolando alcuni aspetti). Può servire per dimostrazioni con utenti e clienti.
- o *Generazione test.* Sviluppare test dei requisiti per controllarli
- o *Analisi di consistenza automatica.* Controllo della consistenza di una descrizione dei requisiti strutturata o formale.

Il processo di **revisione** consiste di dialoghi tra clienti e fornitori. Gli obiettivi sono

Consistenza e Completezza, ma si possono verificare anche

- *Comprensibilità:* Il requisito è stato compreso esattamente?
- *Tracciabilità:* L'origine del requisito è espressa chiaramente? (è importante ai fini della valutazione dell'impatto dei cambiamenti)
- *Adattabilità:* è possibile cambiare il requisito senza un grande impatto sugli altri requisiti?

Modellazione analitica attraverso UML

La modellazione analitica (di analisi) usa una combinazione di forme grafiche e di testo per rappresentare i requisiti dei dati, delle funzionalità e dei componenti in un modo facile da comprendere, che agevola la verifica della correttezza, della completezza e dell'uniformità. Se ne occupa un ingegnere del software chiamato **analista**, che realizza il modello utilizzando requisiti individuali tramite richieste al cliente.

Tale modellazione concettuale rappresenta i requisiti in più dimensioni, aumentando così la probabilità di individuare errori, di rilevare inconsistenze e di scoprire omissioni. I requisiti vengono modellati utilizzando vari formati grafici:

- o Modellazione basata su scenari, rappresenta il sistema dal punto di vista dell'utente.
- o Modellazione orientata al flusso, fornisce un'indicazione del modo in cui gli oggetti/dati vengono trasformati dalle funzioni di elaborazione.
- o Modellazione basata su classi, definisce gli oggetti, gli attributi e le relazioni.
- o Modellazione comportamentale, rappresenta gli stati del sistema e delle classi, e l'impatto degli eventi su questi stati.

Per il modello analitico può essere scelta un'ampia varietà di diagrammi. Ognuna di queste rappresentazioni fornisce una rappresentazione di uno o più elementi del modello.

UML

L'UML (**Unified Modeling Language**, "linguaggio di modellazione unificato") è un linguaggio di modellazione e specifica basato sul paradigma object-oriented.

- *Linguaggio di modellazione*: linguaggio formale che può essere utilizzato per descrivere (modellare) un sistema di qualche natura.
- *Linguaggio di specifica*: linguaggio formale usato per descrivere un sistema software a un livello di astrazione superiore a quello dei linguaggi di programmazione

Esso contiene una famiglia di notazioni grafiche per la modellazione visuale del software, cioè la rappresentazione degli elementi che corrispondono a parti del software. Il nucleo del linguaggio fu definito nel 1996 da Grady Booch, Jim Rumbaugh e Ivar Jacobson (detti "*i tre amigos*") sotto l'egida dello **Object Management Group (OMG)**, consorzio che tuttora gestisce lo standard UML. Il linguaggio nacque con l'intento di unificare approcci precedenti (dovuti ai tre padri di UML e altri), raccogliendo le migliori prassi nel settore e definendo così uno standard industriale unificato.

L'ultima versione del linguaggio, la 2.0, è stata consolidata nel 2004 e ufficializzata da OMG nel 2005.

Funzione di lingua franca

L'UML svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Una lingua franca è una lingua che viene usata come strumento di comunicazione internazionale o comunque fra persone di differente lingua madre e per le quali è straniera. Gran parte della letteratura di settore usa UML per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico.

Applicazioni

Di per sé, UML è solo un linguaggio di modellazione, e non definisce alcuna specifica metodologia per la creazione di modelli (o alcun processo software). UML può quindi essere utilizzato nel contesto di diversi approcci metodologici.

UML consente di costruire modelli object-oriented per rappresentare domini di diverso genere. Nel contesto dell'ingegneria del software, viene usato soprattutto per descrivere il **dominio applicativo** di un sistema software e/o il **comportamento** e la **struttura** del sistema stesso. Il modello è strutturato secondo un insieme di viste che rappresentano diversi aspetti della cosa modellata (funzionamento, struttura, comportamento, e così via), sia a scopo di **analisi** che di **progetto**, mantenendo la tracciabilità dei concetti impiegati nelle diverse viste.

Oltre che per la modellazione di sistemi software, UML viene non di rado impiegato per descrivere domini di altri tipi, come sistemi hardware, strutture organizzative aziendali, processi di business.

Caratteristiche generali

La notazione UML è semi-grafica e semi-formale; un modello UML è costituito da una collezione organizzata di diagrammi correlati, costruiti componendo elementi grafici (con significato formalmente definito), elementi testuali formali, ed elementi di testo libero. Ha una semantica molto precisa e un grande potere descrittivo.

Il linguaggio è stato progettato con l'obiettivo esplicito di facilitare il supporto software alla costruzione di modelli e l'integrazione di questo supporto con gli ambienti integrati di sviluppo. UML è un linguaggio di modellazione *general purpose*, che fornisce concetti e strumenti applicabili in tutti i contesti.

Modalità d'uso

- o *UML come abbozzo*: l'UML è utilizzato come abbozzo (sketch) per documentare alcuni aspetti del software sia prima che il sistema sia sviluppato (forward engineering) sia partendo da un sistema esistente (reverse engineering). Tale modalità d'uso favorisce la comunicazione nelle discussioni, e i criteri fondamentali sono:
 - *Selettività*: solo alcuni aspetti sono modellati graficamente, qualsiasi informazione può essere soppressa, ma non significa che non esiste.
 - *Espressività*: diagrammi creati rapidamente ed in collaborazione.
- o *UML come progetto dettagliato*: usato per guidare la realizzazione o la manutenzione di un sistema; lo scopo principale è fornire agli sviluppatori un modello dettagliato su cui basarsi. Criteri fondamentali:
 - Completezza
 - Non ambiguitàI diagrammi creati fanno parte della documentazione del sistema e vanno modificati per rispecchiare il sistema stesso
- o *UML come linguaggio per programmi*: alcuni tool per UML generano codice direttamente dai diagrammi, il codice viene completato dagli sviluppatori. Altri tool permettono agli sviluppatori di programmare in modo visuale, indipendente dalla piattaforma; gli sviluppatori creano un modello che è trasformato in modo quasi automatico in codice.

Suddivisione dei modelli UML

UML consente di descrivere un sistema secondo due aspetti principali, per ciascuno dei quali si utilizzano un insieme di tipi di diagrammi specifici (che possono poi essere messi in relazione fra loro):

- Diagrammi comportamentali (*behaviour diagram*)

Utilizzano il **modello funzionale** (functional model), il quale rappresenta il sistema dal punto di vista dell'utente, ovvero ne descrive il suo comportamento così come esso è percepito all'esterno, prescindendo dal suo funzionamento interno. Questo tipo di modellazione corrisponde, in ingegneria del software, all'analisi dei requisiti. La modellazione funzionale utilizza gli Use Case Diagram (**Diagrammi dei casi d'uso**).

Utilizzano anche il **modello dinamico** (dynamic model), il quale rappresenta il comportamento degli oggetti del sistema, ovvero la loro evoluzione nel tempo e le dinamiche delle loro interazioni. È strettamente legato al modello a oggetti e viene impiegato negli stessi casi. Utilizza i sequence diagram (**Diagrammi di sequenza**), gli activity diagram (**Diagrammi delle attività**) e gli statechart diagram (**Diagrammi degli stati**). Comprendono infine anche l'insieme del diagramma di interazione (di sequenza, comunicazione, interazione generale, temporizzazione).

- Diagrammi strutturali (*structure diagram*)

Utilizzano il **modello a oggetti** (object model), il quale rappresenta la struttura e sottostruttura del sistema utilizzando i concetti object-oriented di classe, oggetto, le relazioni fra classi e fra oggetti. In ingegneria del software, questo tipo di modellazione può essere utilizzata sia nella fase di analisi del dominio che nelle varie fasi di progetto a diversi livelli di

dettaglio. Utilizza class diagram (**Diagrammi delle classi**), object diagram (**Diagrammi degli oggetti**), e deployment diagram (**Diagrammi di dislocamento**).

Struttura di un modello UML

Un modello UML è costituito da:

1. *Viste*: mostrano i diversi aspetti del sistema per mezzo di un insieme di diagrammi.
2. *Diagrammi*: permettono di descrivere graficamente le viste logiche.
3. *Elementi del modello*: concetti che permettono di realizzare vari diagrammi (es. attori, classi, packages, oggetti, e così via).

Viste

Una vista è una porzione di un modello orientata alla rappresentazione di un particolare aspetto di un sistema software. Lo strato più esterno dell'UML è costituito dalle seguenti viste:

1. *Vista dei casi d'uso* (Use Case View) utilizzata per analizzare i requisiti utente. Obiettivo di questo livello di analisi è studiare il sistema considerandolo come una scatola nera. È necessario concentrarsi su cosa il sistema deve fare astraendosi il più possibile dal come: è necessario individuare tutti gli attori, i casi d'uso e le relative associazioni. Importante è dettagliare i requisiti del cliente, capirne i desideri più o meno consapevoli, cercare di prevedere i possibili sviluppi futuri, ecc.
2. *Vista di progettazione* (Design View) descrive come le funzionalità del sistema devono essere realizzate; in altre parole analizza il sistema dall'interno (scatola trasparente).
3. *Vista di implementazione* (Implementation View) descrive i packages, le classi e le reciproche dipendenze.
4. *Vista dei processi* (Process View) individua i processi e le entità che li eseguono sia per un utilizzo efficace delle risorse, sia per poter stabilire l'esecuzione parallela degli oggetti.
5. *Vista di sviluppo* (Deployment View) mostra l'architettura fisica del sistema e definisce la posizione delle componenti software nella struttura stessa.

Diagramma dei casi d'uso

I diagrammi dei casi d'uso (**UCD**, Use Case Diagram) sono diagrammi dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori (utenti) che interagiscono col sistema stesso. Sono impiegati soprattutto nel contesto della **Use Case View** (vista dei casi d'uso) di un modello, e in tal caso si possono considerare come uno strumento di rappresentazione dei requisiti funzionali di un sistema.

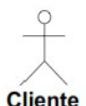
Tuttavia, non è impossibile ipotizzare l'uso degli UCD in altri contesti; durante la progettazione, per esempio, potrebbero essere usati per modellare i servizi offerti da un determinato modulo o sottosistema ad altri moduli o sottosistemi.

In molti modelli di processo software basati su UML, la Use Case View e gli Use Case Diagram che essa contiene rappresentano la vista più importante, attorno a cui si sviluppano tutte le altre attività del ciclo di vita del software.

Elementi

Un diagramma di caso d'uso descrive il comportamento del sistema come appare dall'esterno. Esso individua le funzionalità del sistema significative per gli attori; un diagramma di caso d'uso descrive una interazione come una sequenza di messaggi tra il sistema e uno o più attori.

- **Sistema**: Il sistema nel suo complesso è rappresentato come un rettangolo vuoto. Questo simbolo viene messo in relazione con gli altri nel senso che i modelli elementi che rappresentano caratteristiche del sistema verranno posizionati all'interno del rettangolo, mentre quelli che rappresentano entità esterne (appartenenti al dominio o al contesto del sistema) sono posizionati all'esterno.
In molti diagrammi UML il simbolo per il sistema viene omesso in quanto la distinzione fra concetti relativi al sistema e concetti relativi al suo contesto si può in genere considerare implicita.
- **Attori**: Gli attori sono persone o sistemi fuori dal prodotto software che si sta sviluppando, e che interagiscono con esso. Essi sono rappresentati graficamente nel diagramma da un'icona che rappresenta un uomo stilizzato (stickman) con il nome sotto. Formalmente, un attore è una classe.
Praticamente, un attore rappresenta un ruolo coperto da un certo insieme di entità



Cliente

interagenti col sistema (inclusi utenti umani, altri sistemi software, dispositivi hardware e così via).

- **Caso d'uso:** Un caso d'uso è una unità che offre funzionalità del sistema visibili all'esterno. Esso è rappresentato graficamente come un'ellisse contenente il nome del caso d'uso. Formalmente lo si potrebbe intendere come una classe di comportamenti (o metodi) correlati, che rappresenta una funzione o servizio offerto dal sistema a uno o più attori. La funzione deve essere completa e significativa dal punto di vista degli attori che vi partecipano. Un caso d'uso comunque non rivela struttura o comportamento interni del sistema.

Caratteristiche generali dei casi d'uso

Descrivono l'interazione di un sistema con il suo ambiente e rivolgono l'attenzione alle interazioni tra attori e sistema. Non rivelano la struttura interna del sistema, dunque non mostrano le attività interne e non ne mostrano componenti interni.

Sono espressi in forma di testo e con *diagrammi*, ogni caso d'uso può soddisfare più requisiti, oppure un requisito può dare origini a più casi d'uso. Ad ogni caso d'uso partecipa almeno un attore.

Obiettivo dell'attore

In un caso d'uso, un attore sta cercando di ottenere un certo **obiettivo**. Il caso d'uso consiste di tutte le interazioni tra un attore ed il sistema che sono necessarie per raggiungere quell'obiettivo. Ci possono essere vari scenari in un caso d'uso, ognuno mostra un percorso alternativo in base al successo o meno di certi passi intermedi.

- Es.
- Un amministratore vuole inserire un nuovo utente
 - Un utente vuole registrare un avvenuto pagamento
 - Un cliente vuole stampare un report con i suoi dati personali

Processo

1. Un caso d'uso *inizia* con un messaggio inviato al sistema da un attore;
2. Il sistema risponde con una serie di azioni ed inviando messaggi all'attore che ha iniziato il caso d'uso o ad altri attori.
3. Gli attori possono rispondere con altri messaggi;
4. Il caso d'uso *termina* quando sono state fornite tutte le risposte e l'obiettivo è stato soddisfatto.

Tipi di flussi:

- Di base: quelli che prevedono lo svolgimento di successo ("se tutto va bene")
- Alternativi: possono essere di successo o fallimento

Differenti livelli di astrazione dei casi d'uso

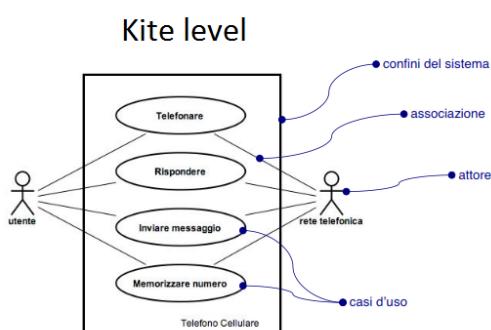
I casi d'uso sono divisi in diversi livelli di granularità, conosciuti come:

- **Cloud level (livello nuvola).** Livello ad altissima astrazione, troppo in alto per essere di interesse per progetti di sviluppo software, e di gran lunga troppo elevato per generare il codice. Ad esempio i libri che si vendono online.
- **Kite level (livello aquilone).** Livello d'astrazione che fornisce pochi dettagli e funzionalità del sistema.
- **Sea level (livello del mare).** Livello dell'obiettivo utente dove un singolo caso d'uso a questo livello descrive un singolo processo elementare, realizzato da un singolo utente (attore). Un esempio potrebbe essere l'ordine del posto.
- **Fish level (livello dei pesci).** Livello di subfunzione dove i casi d'uso sono spesso utilizzati e riutilizzati come parte di un caso d'uso sea level e descrivono dei processi come ad esempio la selezione di un cliente, o la convalida di carte di credito con una società di carte di credito. Questo livello è indicato anche come livello di sub-funzione, e comprende i casi d'uso come seleziona prodotto e ordine di pagamento. Questi casi d'uso, compreso il sea level, sono un buon inizio per la generazione di codice che gestisce il comportamento del software.
- **Clam level (livello mollusco).** Livello dove si tocca il fondo; i casi a questo livello non dovrebbero essere definiti come casi d'uso, ma piuttosto appaiono come passi di un altro caso d'uso, molto probabilmente a fish level. Un chiaro esempio è l'inserimento di nuovi clienti nel database.

Relazioni tra attori e casi d'uso

Descriviamo le relazioni tra gli attori e i casi d'uso partendo dall'esempio della costruzione di un UML de casi d'uso di un software di gestione del telefono.

Casi d'uso:



- Telefonare: L'utente vuole effettuare una telefonata (per far questo l'utente interagisce tramite un flusso di operazioni, tutte queste sono rappresentate dal caso d'uso telefonare)

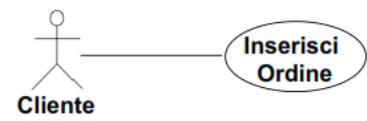
- Rispondere
- Inviare SMS
- Memorizzare numero

Funzionalità:

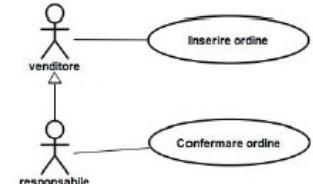
- Trasmissione e ricezione dati
- Gestione I/O (display, tasti, microfono)
- Gestione rubrica



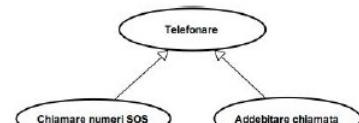
➤ **Associazione.** È una comunicazione tra gli attori e i casi d'uso a cui essi partecipano. Un attore può essere associato a un qualsiasi numero di casi d'uso, e viceversa; essa infatti implica uno scambio messaggi fra attori e use case associati. Viene indicata con una linea che congiunge un attore e un caso d'uso.



➤ **Generalizzazione tra attori.** Relazione tra un attore più generale ed uno più specifico che eredita da quello più generale ed aggiunge capacità. In altre parole, il sottoattore (o attore specializzato) eredita la possibilità di partecipare a tutti i casi d'uso del superattore (attore generico), ed eventualmente può partecipare a casi d'uso aggiuntivi. Il sottoattore viene indicato con una freccia che parte da esso e finisce al superattore.

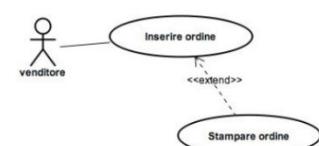
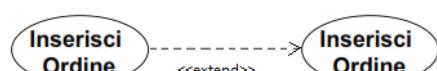


➤ **Generalizzazione tra casi d'uso.** Relazione tra casi d'uso, uno più specifico eredita da uno più generale ed aggiunge caratteristiche a quello più generale. In altre parole, un sottocaso d'uso deve fornire lo stesso servizio generale del supercaso d'uso, eventualmente producendo valore aggiuntivo, o fornendolo a qualche tipologia di attore aggiuntiva; o seguendo un procedimento parzialmente diverso per ottenere il risultato, e così via. Un caso d'uso figlio può essere usato al posto del padre. Il sottocaso viene indicato con una freccia che parte da esso e finisce al supercaso.



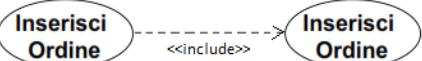
➤ **Estensione tra casi d'uso.** Definisce un caso d'uso come incremento opzionale di un caso d'uso base. Rappresentata da una linea tratteggiata con indicazione dello stereotipo «extend», indica che la funzione rappresentata dallo use case "estendente" (alla base della freccia) può essere impiegata nel contesto della funzione "estesa" (lo use case alla punta), ovvero ne rappresenta una sorta di arricchimento. Il caso d'uso di base definisce il punto di estensione e la condizione che attiva il caso d'uso d'estensione.

Es. Il "Venditore" usa prima "Inserire l'ordine", e poi da quel punto avvia opzionalmente "Stampa l'ordine". L'avvio di "Stampa ordine" è subordinato all'avvio di "Inserire ordine"; "Stampa ordine" è un'attività con cui "Venditore" agisce.

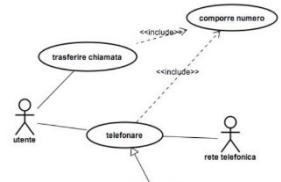


Questa relazione viene spesso utilizzata per isolare in uno use case a parte la specifica di attività opzionali o eccezionali che potrebbero aver luogo durante l'espletamento della funzione principale.

- **Inclusione tra casi d'uso.** Un caso d'uso incorpora comportamenti di altri casi d'uso come parti del proprio comportamento. Rappresentata da una linea tratteggiata con indicazione dello stereotipo «include», indica che la funzione rappresentata da uno dei due use case (quello alla base della freccia) include completamente la funzione rappresentata dall'altro (quello alla punta). In altre parole, una sequenza di passi che l'attore compie può essere rappresentata come un caso d'uso incluso in altri casi d'uso. Il caso d'uso incluso descrive un *obiettivo secondario* rispetto al caso d'uso base.



Es. “Comporre numero” è incluso in “Telefonare” e “Trasferire chiamata”; “Utente” avvia “Trasferire chiamata” e dopo usa “Comporre numero”.



Si noti che le relazioni di estensione e inclusione rappresentano concetti piuttosto vicini, ma che l'orientamento delle frecce nei due casi si può descrivere come "opposto". La sottile differenza fra i due stereotipi è la seguente:

- o «extend» indica un frammento che *può* essere eseguito in determinate circostanze del caso d'uso alla punta della freccia;
 - o «include» indica un frammento che viene *sempre* eseguito durante l'esecuzione del caso d'uso alla base della freccia.

Identificazione casi d'uso

1. Identificare gli attori principali del sistema
 2. Per ogni attore individuare le azioni che può fare sul sistema
 3. Per ogni caso d'uso
 - o Chiarire come inizia l'attività
 - o Le risposte che l'attore si aspetta dal sistema
 - o La sequenza di passi che l'attore usa per svolgere l'attività
 - o Eventuali altri attori coinvolti
 - 4.I casi d'uso (descrizione del sistema) forniscono
 - o Un documento per analisti, progettisti e test
 - o Indicazione sulla dimensione del sistema
 - o La guida per l'utente

Esempio di descrizione caso d'uso (apri file)

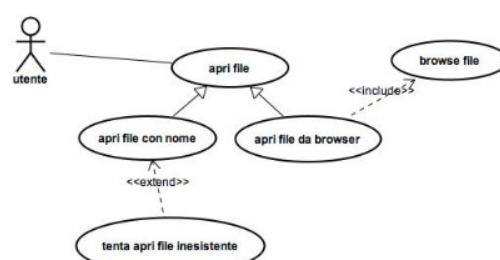
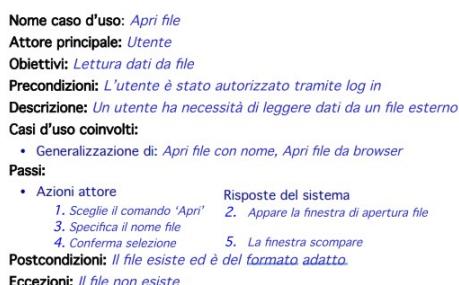
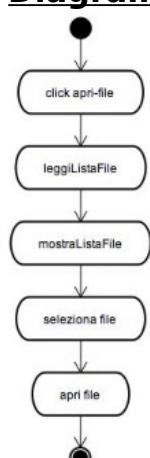


Diagramma delle attività



Il diagramma delle attività è un diagramma definito all'interno dello UML che definisce le attività da svolgere per realizzare una data funzionalità. Può essere utilizzato durante la progettazione del software per dettagliare un determinato algoritmo. Più in dettaglio, un activity diagram definisce una serie di **attività o flusso**, anche in termini di relazioni tra le attività, i responsabili per le singole attività e i punti di decisione. L'activity diagram è spesso usato come modello complementare al *diagramma dei casi d'uso*, per descrivere le dinamiche con cui si sviluppano i diversi use case.

È utilizzato per la modellazione di processi e workflow; mostra dettagli di funzionamento interno del sistema software da realizzare; è una vista sull'esecuzione delle attività e mostra anche le dipendenze tra attività (o passi).

Es. A sinistra, Attività: Apri file da browser

Elementi

Un diagramma delle attività utilizza *rettangoli arrotondati* per specificare una determinata funzione del sistema, *frecce* per rappresentare il flusso attraverso il sistema, *rombi decisionali* per rappresentare una decisione che prevede una ramificazione e *linee continue* orizzontali per indicare le attività parallele che si stanno verificando.

Struttura

Avrò tanti diagrammi delle attività quanto obiettivi diversi ha il sistema software. I flussi sono paralleli e contemporanei, e vi è la possibilità di crearsi dei **fork**, ovvero quando si stanno avviando due cose da una base. Entra una freccia e ne escono 2, da un'attività ne verranno eseguite due attività parallele. Esse finiranno nella *barra di sincronizzazione*, dove si aspetterà che le due attività vengano completate per riunirle, eseguirle insieme e dare un'altra unica attività e finire il processo.

Flusso

Il flusso è rappresentato tramite delle frecce orientate, che indicano la sequenza temporale con cui devono essere effettuate le diverse attività. È previsto un simbolo per indicare l'inizio del flusso ed un altro per indicarne il termine. Le attività possono essere anche rese in parallelo, in questo caso il punto di divisione (fork) è rappresentato da frecce divergenti rispetto al segmento.

Nel caso le attività siano alternative, cioè svolte o meno rispetto ad una scelta, il punto di decisione è rappresentato da dei rombi da cui partono i flussi alternativi.

Il punto di ricongiungimento (join) è reso tramite un segmento su cui le frecce si ricongiungono.

Diagramma delle attività con corsie o Swimlane

I diagrammi swimlane rappresentano un'utile variante del diagramma di attività che consente al moderatore di rappresentare il flusso delle attività descritte dal caso d'uso e contemporaneamente indicare quale attore o classe ha la responsabilità per l'azione descritta dal rettangolo.

Rielaboriamo i requisiti, cerchiamo di capire il sistema software e lo speziamo in passi elementari, formando il diagramma delle attività; questi dati potrebbero diventare delle classi.

Esempi

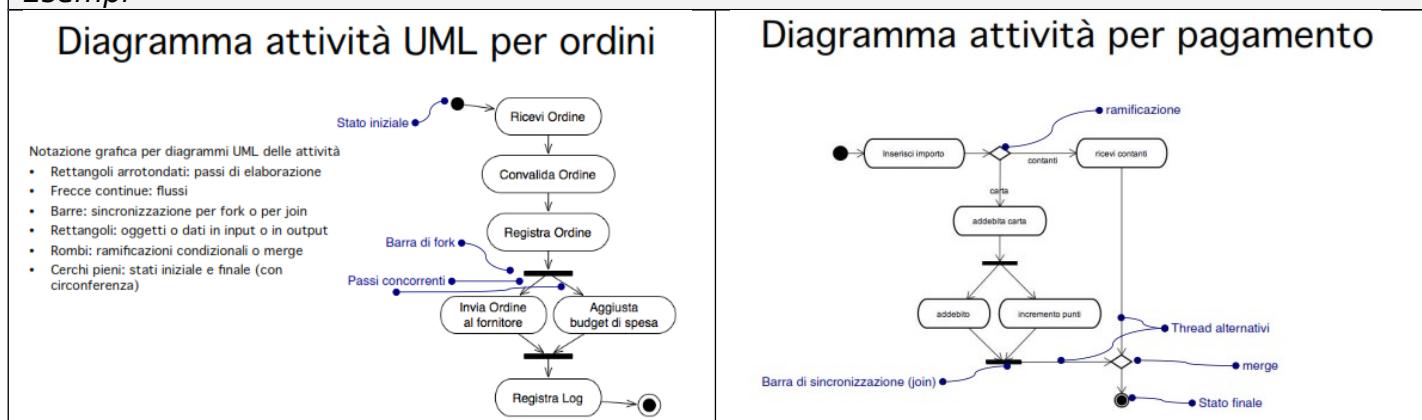
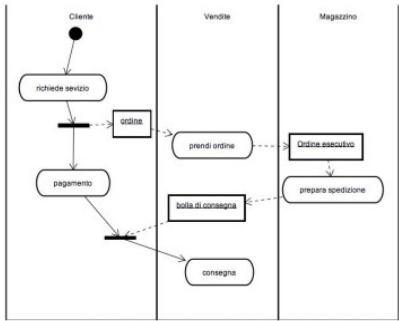


Diagramma attività con Swimlane

- **Swimlane:** partizione (corsia) che indica il responsabile (unità, attore, etc.) dell'attività
- **Rettangoli:** oggetti o dati
- **Frecce tratteggiate:** input o output di un oggetto per/da una attività



I diagrammi delle attività non dicono quali sono gli oggetti che svolgono le attività (lo fanno con gli swimlane), sono il punto iniziale del design, vanno ri-elaborati per arrivare ad assegnare una o più operazioni ad una classe che le implementa e possono essere usati come punto di partenza per ottenere i collaboration diagram.

Diagramma degli stati

Rappresentazione degli stati

Nel contesto della modellazione comportamentale, occorre considerare due diverse caratterizzazioni degli stati:

1. Lo stato di ciascuna classe mentre il sistema svolge la propria funzione
2. Lo stato del sistema così come viene osservato dall'esterno mentre il sistema svolge la propria funzione

Lo stato di una classe può avere caratteristiche attive e passive.

- Uno **stato passivo** è semplicemente lo stato corrente di tutti gli attributi, per esempio lo stato passivo della classe *Videogiocatore* comprenderà gli attributi posizione, orientamento e disponibilità incantesimi.
- Lo **stato attivo** di un oggetto indica lo stato corrente dell'oggetto mentre questo si sottopone a una continua trasformazione o elaborazione. Per esempio la classe *Videogiocatore* può avere i seguenti stati: in movimento, a riposo, colpito, curato, intrappolato, perso, ecc.

Può anche verificarsi un evento (chiamato anche *trigger*) per forzare un oggetto a fare una transizione da uno stato attivo a un altro.

Esistono due diverse rappresentazioni del comportamento degli stati. I diagrammi di stato indicano il modo un cui una singola classe cambia stato sulla base di eventi esterni; mentre i diagrammi di sequenza mostrano il comportamento del software in funzione del tempo.

Diagramma degli stati

I diagrammi descritti fino ad ora rappresentano gli elementi statici del modello analitico.

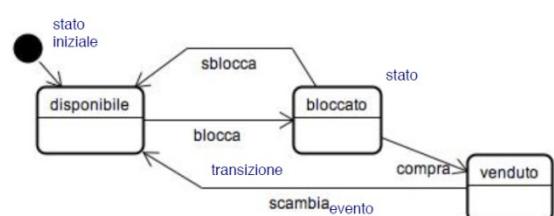
Esistono anche dei diagrammi che descrivono il comportamento dinamico del sistema software. Per ottenere ciò occorre rappresentare il comportamento del sistema in funzione degli eventi e del tempo. Il **modello comportamentale** indica il modo in cui il software risponde agli eventi esterni.

Lo *State Chart Diagram* è un diagramma previsto dall'UML per descrivere il comportamento di entità o di classi in termini di stato (macchina a stati).

Il diagramma mostra gli stati che sono assunti dall'entità o dalla classe in risposta ad eventi esterni.

Es. Diagramma UML degli stati che mostra la vita di un biglietto per uno spettacolo.

I rettangoli con angoli arrotondati sono gli **stati**, le frecce le **transizioni** tra stati, il **cerchio piano** è lo stato iniziale e il **cerchio con circonferenza** è lo stato finale.



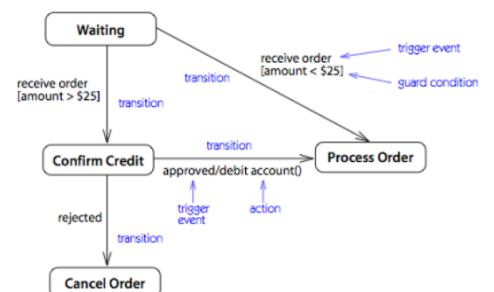
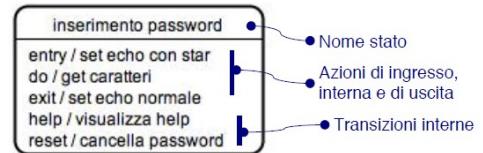
Il biglietto da uno stato iniziale *disponibile* diventa *bloccato* a seguito di *prenotazioni* e poi passa allo stato di *venduto*. L'operazione blocca biglietto è data solo quando lo stato del biglietto è disponibile.

Il concetto di stato è spesso posto in relazione a ciclo di vita; l'insieme completo di stati che un'entità o una classe può assumere, dallo stato iniziale a quello finale, ne rappresenta il ciclo di vita.

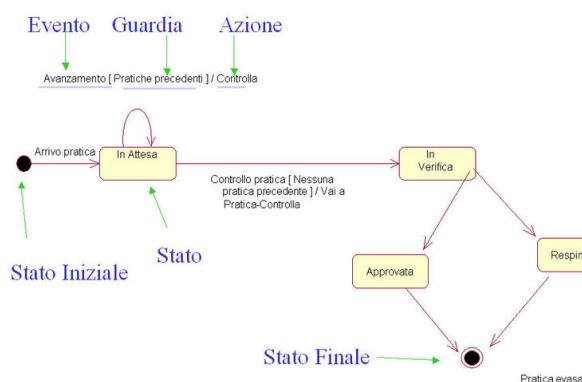
Elementi

Gli elementi rappresentati da uno State Chart Diagram sono lo stato (distinguendo tra iniziale, intermedio e finale), l'evento, l'azione e la guardia.

- **Stato.** Descrive una qualità dell'entità o classe che si sta rappresentando (pratica aperta, in lavorazione, sospesa, chiusa) durante un intervallo di tempo della vita di un oggetto. È caratterizzato dai valori degli oggetti, o dall'intervallo in cui un oggetto aspetta certi eventi, o dall'intervallo in cui un oggetto fa certe azioni.
- **Evento.** Descrizione dell'azione che comporta il cambiamento di stato. In generale, esso si verifica ogni volta che il sistema e un attore si scambiano informazioni.
- **Transizione.** Permette di lasciare uno stato in risposta ad un certo evento; un oggetto gestisce un solo evento alla volta. La transizione è caratterizzata da: **event-trigger[guard]/action**
- **Trigger.** Evento di inizio.
- **Azione.** L'evento che ne consegue.
- **Guardia.** Eventuale condizione che si deve verificare perché si possa compiere l'azione (espressione booleana). Essa è valutata quando l'evento avviene, e fa avvenire la transizione solo quando è valutata true.
- **Oggetti.** Possono essere responsabili per la generazione degli eventi o per il riconoscimento di eventi che si sono verificati altrove.

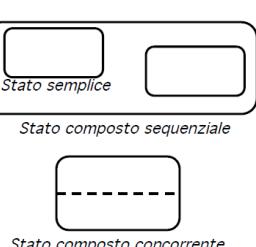


State Chart Diagram



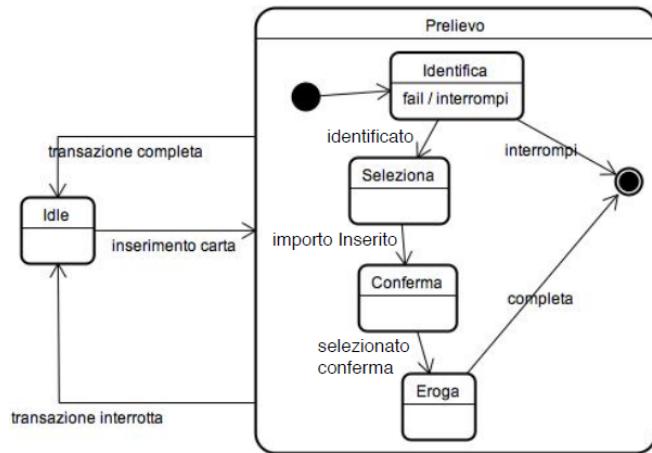
Stati composti

Uno stato composto è uno stato che consiste di vari **sottostati sequenziali** o **concorrenti**. Lo stato esterno rappresenta la condizione di essere in uno qualsiasi degli stati interni; solo uno dei sottostati sequenziali può essere attivo in un certo momento.

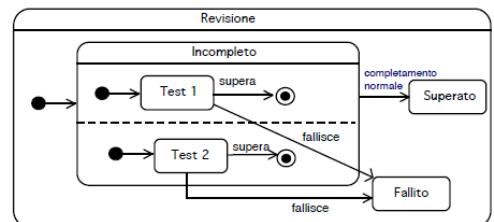


Una transizione verso, o da uno stato composto può invocare varie azioni di entry (dalla più esterna) o di exit (dalla più interna).

Esempio: Diagramma UML degli stati per un bancomat. Il diagramma comprende uno stato composto con all'interno sottostati sequenziali.



Es. Diagramma degli stati per Revisioni. *Revisione* è uno stato composto con sottospazi sequenziali *Incompleto*, *Superato* e *Fallito*. *Incompleto* è uno sottostato composto con sottostanti concorrenti *Test 1* e *Test 2*.



Es. Diagramma degli stati con Fork e Join.

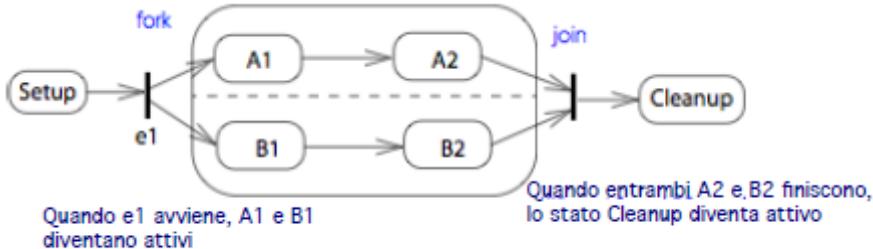


Diagramma di sequenza

Un *Sequence Diagram* è un diagramma previsto dall'UML utilizzato per descrivere uno **scenario**, ovvero una determinata sequenza di azioni e interazioni tra oggetti nel tempo di vita del software, in cui tutte le scelte sono state già effettuate; in pratica nel diagramma non compaiono scelte, né flussi alternativi. Esso illustra cosa avviene al runtime durante l'esecuzione del programma e mostra l'evoluzione nel tempo quando si esegue il programma.

Normalmente da ogni Activity Diagram sono derivati uno o più Sequence Diagram; se per esempio l'Activity Diagram descrive due flussi di azioni alternativi, se ne potrebbero ricavare due scenari, e quindi due Sequence Diagram alternativi.

Il Sequence Diagram descrive le relazioni che intercorrono, in termini di messaggi, tra Attori, Oggetti di business, Oggetti od Entità del sistema che si sta rappresentando.

Sequence Diagram

Messaggi dei diagrammi di sequenza

Un messaggio è un'informazione che viene scambiata tra due entità. Solitamente chi invia il messaggio, la parte attiva, è l'attore. Il messaggio è **sincrono** se l'emittente rimane in attesa di una risposta, o **asincrono**, nel caso l'emittente non aspetti la risposta e questa può arrivare in un secondo momento.

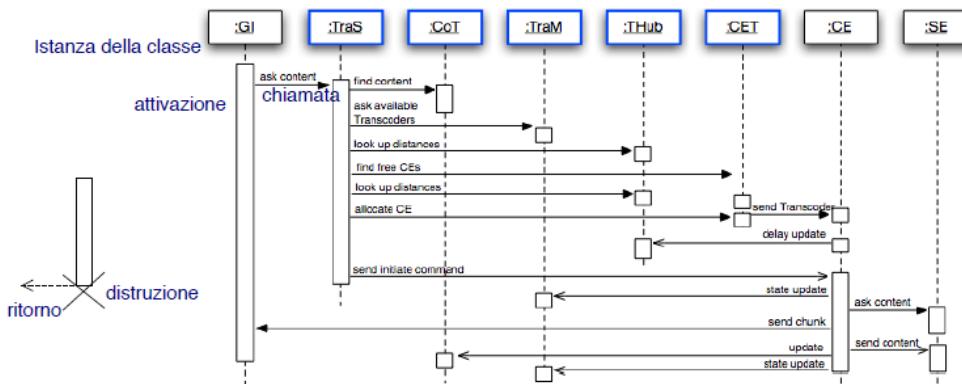
Il messaggio che viene generato in risposta ad un precedente messaggio, al quale si riferisce anche come contenuto informativo, è detto messaggio di risposta.



Un messaggio, in cui il ricevente è nello stesso tempo l'emittente, è detto ricorsivo.

Rappresentazione grafica del diagramma di sequenza

L'asse temporale è inteso in verticale, in orizzontale invece ci sono i vari oggetti che ne prendono parte. In ciascuna colonna verticale, se l'oggetto che partecipa esiste è indicato con una linea tratteggiata, se è attivo con un rettangolo (di attivazione). Un messaggio è una freccia dal rettangolo di attivazione di un oggetto ad un altro; frecce piene indicano comunicazione sincrona, viceversa vuote asincrona.



Gli elementi che stanno in basso sono avvenuti dopo di quelli che stanno più in alto. Il controllo ce l'ha sempre il metodo a sinistra che poi chiama quelli a destra e quando finiscono il controllo torna a sinistra.

Diagramma dei flussi (Data Flow Diagram, DFD)

Diagrammi che mostrano i passi per l'elaborazione dei dati che attraversano il sistema. Utilizzano una notazione intuitiva comprensibile ai clienti e mostrano lo scambio di informazioni tra sistemi differenti e sottosistemi. Sono simili al diagramma delle attività UML. Attraverso i Data Flow Diagram si definiscono soprattutto come fluiscono (e vengono elaborate) le informazioni all'interno del sistema, quindi l'oggetto principale è il **flusso delle informazioni** o, per meglio dire, dei dati. Motivo per il quale diventa fondamentale capire dove sono immagazzinati i dati, da che fonte provengono, su quale fonte arrivano, quali componenti del sistema li elaborano.

Le componenti di questo tipo di diagramma sono:

- **Funzioni.** Rappresentate da bolle, rappresentano unità di elaborazione dei dati e trasformano i dati in ingresso in quelli in uscita.
- **Flussi di dati.** Rappresentati da frecce che collegano i diversi componenti di un diagramma tra loro, rappresentano i dati gestiti dal sistema. Gli archivi e gli agenti esterni NON possono essere collegati tra loro.
- **Archivi di dati.** Rappresentati da scatole aperte, sono dei depositi permanenti di informazione. Possono essere basati su qualunque tecnologia. I dati che entrano in un archivio vengono scritti e i dati che escono dall'archivio sono letti (ma non cancellati).
- **Agenti esterni o Input/Output di dati.** Rappresentati da scatole chiuse, sono delle entità esterne al sistema. Non sono soggette ad ulteriore modellazione e sono le sorgenti e le destinazioni dei dati del sistema.

Un generico sistema è sempre rappresentabile ne-

Es.

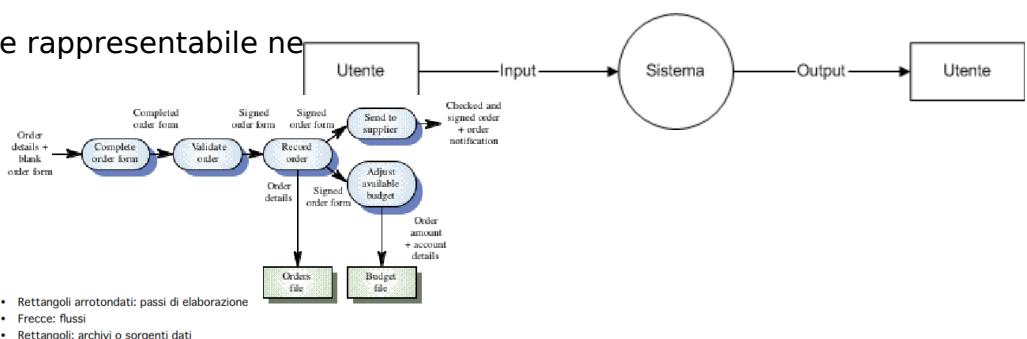


Diagramma delle classi

Modelli di dati ad oggetti

I **modelli di dati** sono usati per descrivere la struttura logica dei dati elaborati. Il modello entità-relazione-attributo definisce entità e relazioni tra queste ed i loro attributi. Sono usati nella progettazione di database e facilmente implementabili con db relazionali. I dizionari di dati contengono tutti i nomi usati e descrivono entità, relazioni ed attributi. Sono simili ai diagrammi di classe UML.

I **modelli di dati ad oggetti** descrivono il sistema in termini di classi (OOP). Una classe ha attributi ed operazioni comuni ad un set di oggetti; esistono vari modelli (e diagrammi) ad oggetti con associazioni di ereditarietà, aggregazione, interazione.

Modelli ad oggetti	
Pro	Contro
<ul style="list-style-type: none"> - Mappa naturalmente entità del mondo reale - Classi che rappresentano entità del dominio sono riusabili 	<ul style="list-style-type: none"> - Entità astratte sono più difficilmente modellabili - L'identificazione di classi è un processo difficile che richiede una comprensione profonda del dominio applicativo

Diagramma delle classi

Tipi di diagrammi che consentono di descrivere tipi di **entità (istanze)**, con le loro caratteristiche e le eventuali relazioni fra questi tipi. Gli strumenti concettuali utilizzati sono il concetto di classe del paradigma **object-oriented** e altri correlati (per esempio la *generalizzazione*, che è una relazione concettuale assimilabile al meccanismo object-oriented dell'*ereditarietà*).

Uno degli assunti fondamentali del paradigma a oggetti è che il concetto di **classe** si presta a rappresentare in modo diretto e intuitivo la realtà, in qualsiasi ambito. I diagrammi delle classi UML sono basati su versioni astratte di tali concetti, e possono essere utilizzati per descrivere sostanzialmente qualsiasi contesto a qualsiasi livello di astrazione. Di conseguenza, UML prevede un loro impiego a livello di analisi e in particolare analisi del dominio (ovvero la descrizione del contesto in cui un sistema software deve operare), ma anche a livello di progettazione (nella descrizione della struttura interna del sistema, dei suoi componenti e delle loro relazioni).

Identificazione delle classi

Dall'elenco dei requisiti bisogna avviare una vera e propria analisi grammaticale del testo, identificando:

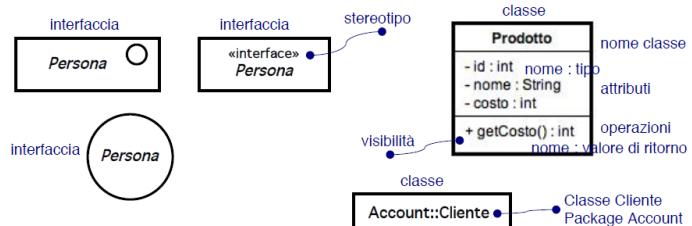
- Nomi → classi o attributi
- Verbi → operazioni

Ad ogni oggetto fisico corrisponde una classe, ma anche raggruppando in modo coeso operazioni e dati tra loro ci suggeriranno delle classi.

Es. Gestione degli ordini

Elementi

- **Classe**. L'elemento di modello principale dei diagrammi delle classi è la classe. Una classe rappresenta una categoria di entità (istanze), nel caso particolare dette **oggetti**; il nome della classe indica la categoria di entità descritta dalla classe. Ogni classe è corredata da un insieme di **attributi** (che descrivono le caratteristiche o lo stato degli oggetti della classe) e **operazioni** (che descrivono il comportamento della classe). Il simbolo grafico che rappresenta le classi UML è un rettangolo suddiviso in tre scomparti, rispettivamente dedicati al *nome della classe*, *nome attributi* e *nome metodi (operazioni)*.



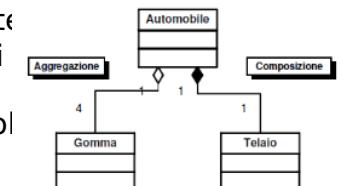
- La struttura degli attributi è: **visibilità nome (parametri): tipoRestituito = default {proprietà}**
- Per la visibilità di attributi e metodi si usa: + **public**, # **protected**, - **private**
- I nomi delle *interfacce* sono in corsivo
- I metodi statici sono sottolineati
- I metodi astratti sono tra le virgolette seguenti <<abstract>>

Esempio:

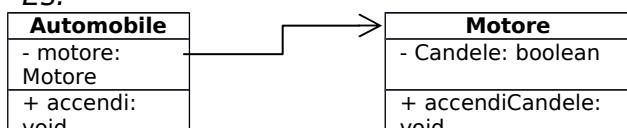
```
+ setNome(nome: String): void - attributoPrivato: int + <<attributoPublicoAstratto: int>> #metodoProtected(): void
```

Uno **Stereotipo** indica una variazione di un elemento UML esistente, che ha tutte le proprietà dell'elemento di partenza.

- **Associazioni.** Due classi possono essere legate da associazioni che rappresentano i legami (link) che possono sussistere fra gli oggetti delle classi associate. Ogni categoria di associazione (*aggregazione*, *composizione*, *dipendenza*, *generalizzazione*, ecc.) viene rappresentata mediante una particolare freccia che connette le due classi coinvolte. Tali associazioni possono essere corredate da un insieme di informazioni aggiuntive, per esempio relative alla molteplicità (il numero di oggetti delle due classi associate che possono essere coinvolti in un link).
- Aggregazione.** Consiste nella creazione di una classe a parte che rappresenta un'associazione uno a molti, in cui un oggetto di una classe è parte di un'oggetto di altre.
 - Composizione.** La classe a parte creata appartiene ad un solo oggetto e muore insieme all'oggetto che la contiene.



Esempio:



```
public class Automobile{
    private Motore motore;
    public void accendi(){
        motore.accendiCandeles();
    }
}

public class Motore{
    private boolean Candeles = false;

    public void accendiCandeles(){
        Candeles = true;
    }
}
```

Le associazioni, inoltre possono descrivere delle informazioni scritte accanto le frecce di collegamento. Esse possono comprendere:

- Molteplicità:** quante istanze di una classe possono essere in relazione con una istanza di un'altra classe (* indica illimitate)
- Nome ruolo:** usato per attraversare l'associazione (una sorta di attributo per la classe all'altro estremo).
- Navigabilità:** verso di attraversamento
- Nome:** spiega l'associazione e il verso

- **Dipendenze.** Due classi possono essere legate da una relazione di dipendenza, che indica che la definizione di una delle due fa riferimento alla definizione dell'altra.

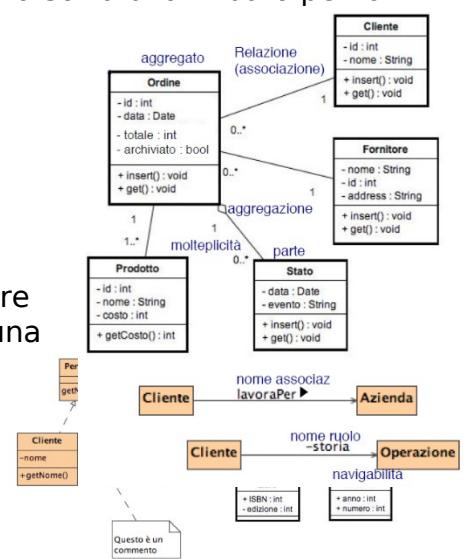
- **Generalizzazione (ereditarietà).** Due classi possono essere legate da una relazione di generalizzazione, che indica che una delle due classi (detta superclasse) si può considerare una generalizzazione dell'altra (detta sottoclasse). Le classi in alto nella gerarchia (superclassi) mostrano le proprietà comuni delle classi in basso (sottoclassi), le quali ereditano gli attributi e i servizi da una o più superclassi.

Graficamente viene indicata con una freccia come questa:

→ il cui verso della freccia è da sottoclasse a superclasse.

Costrutti di estendibilità

Sono presenti dell'UML delle classi dei speciali costrutti che hanno le seguenti proprietà:



- ❖ **Vincoli (Constraints)**. Si usano per indicare condizioni o restrizioni e sono rappresentati da espressioni entro parentesi graffe.
Es. Accanto ad un attributo: {il valore è multiplo di 10}
- ❖ **Stereotipi**. Si usano per definire nuovi elementi o per specificare tipi di relazioni o dipendenze più generiche tra le associazioni tra classi, e sono rappresentati da testo entro « ». Quelli predefiniti sono: «use», «call», «instantiate», «destroy». «use» indica che un elemento è richiesto per il corretto funzionamento di un altro. «implements» invece è usato per la generalizzazione tra classi.
- ❖ **Tagged Value**. Coppia di stringhe che indica un dato ed il suo valore dentro le { }
Es. dentro una classe: {nome=John}

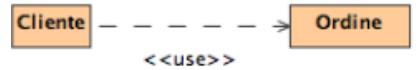
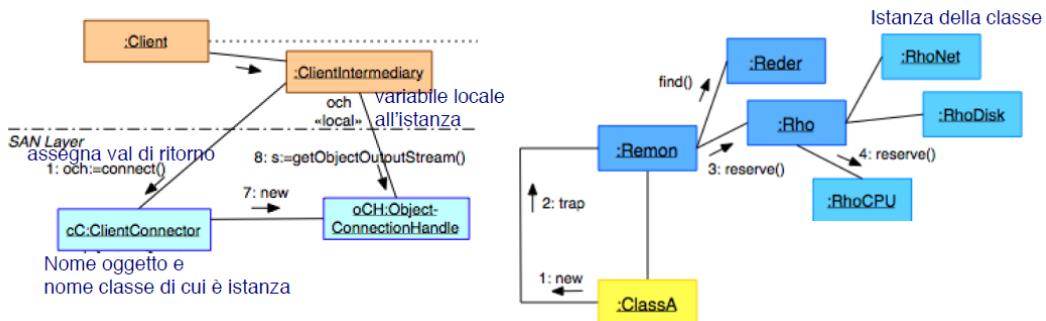


Diagramma di collaborazione

Rappresenta le istanze delle classi sparsi nel foglio mostrando le interazioni tra esse. Il flusso dei messaggi è indicato da frecce accanto le associazioni tra istanze che partecipano all'interazione.

I messaggi sono mostrati da etichette sulle frecce ed hanno un numero sequenziale che indica l'ordine temporale con cui avvengono, il metodo chiamato e un valore di ritorno (opzionale).



Progettazione delle architetture software

Progettazione

La progettazione è il luogo in cui domina la creatività, dove i requisiti del cliente, gli effettivi bisogni e le considerazioni tecniche determinano la formulazione di un prodotto o di un sistema. La progettazione crea una rappresentazione o un modello del software, ma a differenza del modello analitico che si concentra sulla descrizione di dati, funzioni e componenti richiesti, il **modello progettuale** fornisce dettagli relativi alle strutture dati, all'architettura, alle interfacce e ai componenti del software che sono necessari per implementare il sistema.

Scopo

La progettazione consente ad un ingegnere software di modellare il sistema che deve essere realizzato, il quale può essere valutato in termini di qualità e migliorato prima che venga generato il codice, prima che vengano condotti i test e prima che vengano coinvolti gli utenti finali. La progettazione è il luogo in cui viene valutata la **qualità** del software.

Rappresentazione del software

La progettazione rappresenta il software in vari modi:

1. Innanzitutto occorre rappresentare l'architettura del sistema o prodotto;
2. Poi occorre modellare le interfacce che connettono il software agli utenti finali, agli altri dispositivi e ai componenti costruttivi;
3. Infine devono essere progettati i componenti software utilizzati per costruire il sistema.

Ognuna di queste viste rappresenta un aspetto differente della progettazione.

Progettare le architetture

La progettazione dell'architettura rappresenta la struttura dei dati e i componenti del programma che sono necessari per realizzare un sistema software. Essa considera lo stile dell'architettura che verrà assunto dal sistema, le strutture dati che si utilizzeranno, le proprietà dei componenti, e le relazioni che si stabiliscono fra tutti i componenti dell'architettura di un sistema.

Lo scopo della progettazione delle architetture è creare il progetto d'insieme del software e assicurare che sia facilmente comprensibile.

Architetto di sistemi

Se ne occupa l'architetto di sistemi o il progettista di database o strutture, una figura specializzata che si dedica alla realizzazione di sistemi molto grossi o complessi. Egli crea l'architettura dei dati per il sistema selezionando uno stile di architettura appropriato per i requisiti determinati durante la fase di ingegneria del sistema e di analisi dei requisiti software.

Passi necessari

La progettazione dell'architettura inizia con la progettazione dei dati e poi procede con la determinazione di una o più rappresentazioni della struttura di architettura del sistema. Per determinare la struttura più adatta per i requisiti del cliente e per gli attributi di qualità vengono analizzati più stili e modelli di architettura. Dopo aver selezionato una delle alternative, l'architettura viene elaborata utilizzando un determinato metodo di progettazione.

Prodotto

Vengono prodotti i dati dell'architettura e la struttura del programma; inoltre vengono descritte le proprietà e le relazioni fra i componenti.

Verifica

In ogni fase si controlla la chiarezza, correttezza, completezza e consistenza del progetto software rispetto ai requisiti.

Suddivisione in livelli

L'architettura software è divisa in due livelli:

- *Progettazione dei dati*, rappresentazione dei dati che compongono l'architettura;
- *Progettazione dell'architettura*, rappresentazione della struttura dei componenti software e delle loro proprietà e interazioni.

Elementi dell'architettura software

L'architettura software di un programma è costituita dalle strutture del sistema, che comprendono le componenti software, le loro proprietà visibili, e le relazioni fra di essi. Essa è un artefatto, frutto della attività di progettazione, ed è descritta dai suoi componenti (e sottosistemi) e dalle relazioni tra essi

Componenti

Costituiscono i *building block* di un sistema, i componenti software possono essere delle cose semplici, come ad esempio moduli, classi, funzioni; ma possono anche essere elementi complessi come database o elementi che consentono la configurazione di una rete di client e server.

Un componente fornisce servizi ad altri componenti e non è considerato come un sistema a sé stante.

Relazioni

Denotano una connessione tra componenti: aggregazione, eredità, interazione

Sottosistema

È un sistema le cui operazioni sono indipendenti dai servizi di altri sottosistemi.

Progettazione dei dati

La progettazione dei dati traduce gli oggetti/dati definiti dalla modellazione analitica in strutture di dati a livello dei componenti software, o, se necessario in un'architettura di database a livello dell'applicazione.

Progettazione dei dati a livello dell'architettura

Ogni piccola o grande impresa ha gigabyte di dati contenuti in grandi database, e la loro più grande sfida è aumentare sempre di più la ricerca dei dati. Oggi si è arrivati ad una soluzione chiamata *magazzino dati* (**warehouse**), che aggiunge un ulteriore livello all'architettura dei dati. Una warehouse è un ambiente distinto che non è direttamente integrato con le applicazioni di uso quotidiano ma comprende tutti i dati utilizzati dall'azienda; esso è come un grosso database indipendente che comprende alcuni ma non tutti i dati contenuti nei database che servono l'insieme delle applicazioni utilizzate dall'azienda.

Progettazione dei dati a livello dei componenti

Essa si concentra sulla rappresentazione delle strutture dati il cui accesso avviene direttamente da parte di uno o più componenti software.

Viste architetturali

Una vista su un'architettura software è una proiezione dell'architettura secondo un criterio stabilito. Secondo questa definizione, una vista considera solo alcuni sottosistemi, per esempio considera solo la strutturazione del sistema in componenti, o solo alcune relazioni tra sottosistemi.

Vari punti di vista possono produrre varie architetture che rappresentano prospettive diverse del sistema, come mostrare i componenti del sistema o definire le interfacce tra sottosistemi.

Classifica delle viste

Un tipo di vista caratterizza un insieme di vista. Esse sono così classificate:

- **Viste di tipo strutturale**: descrivono la struttura del software in termini di unità di realizzazione. Un'unità di realizzazione può essere una classe, un package Java, un livello, etc.
- **Viste di tipo comportamentale (componenti e connettori)**: descrivono l'architettura in termini di unità di esecuzione, con comportamenti e interazioni. Un componente può essere un oggetto, un processo, una collezione di oggetti, etc.

- **Viste di tipo logistico:** descrivono le relazioni con altre strutture, tipo hardware o organigramma aziendale. Per esempio, l'allocazione dei componenti su nodi hardware.

Stili e modelli dell'architettura

Uno stile architettonico è una particolare proprietà di un'architettura tale che descriva con poche parole un particolare modello di costruzione del sistema software. Ogni stile descrive una categoria di sistemi che comprendono:

1. Un insieme di componenti (database, moduli computazionali) che svolgono una funzionalità richiesta dal sistema;
2. In insieme di connettori che consentono la comunicazione e il coordinamento fra i componenti;
3. I vincoli che definiscono il modo in cui i componenti possono essere integrati per costruire il sistema;
4. I modelli semantici che consentono a un progettista di comprendere le proprietà globali di un sistema analizzando le proprietà note delle parti che lo compongono.

Uno stile di architettura è una trasformazione che viene imposta al progetto di un intero sistema. Lo scopo è quello di stabilire una struttura per tutti i componenti del sistema. La conoscenza degli stili può semplificare la definizione dell'architettura per un sistema. I sistemi più grandi sono eterogenei e non seguono un singolo stile.

Tipi di stili architettonici

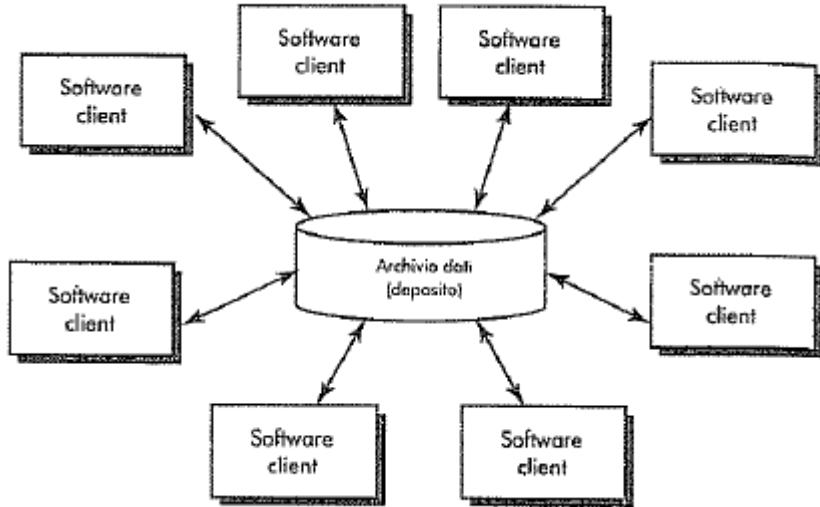
Gli stili architettonici noti sono proprietà che valgono su grandi insiemi di architetture. Essi si dividono in:

➤ **Architetture basate sui dati**

Il nucleo di questa architettura è un archivio di dati (un file o un database) consultato frequentemente da altri componenti che aggiornano, aggiungono, cancellano o modificano in qualche modo i dati in esso contenuto. Le architetture basate sui dati promuovono l'**integrabilità**, ovvero i componenti esistenti possono essere cambiati e all'architettura è possibile aggiungere nuovi componenti e client senza problemi per altri client. Inoltre i dati possono essere passati fra i client utilizzando un meccanismo a *blackboard* che coordina il trasferimento delle informazioni fra i client. I componenti client eseguono i processi in modo indipendente.

Modello Repository (Blackboard): Si considerino sottosistemi che necessitano di comunicare pesantemente. Due possibili soluzioni sono: utilizzare uno spazio condiviso, o inviare i dati mantenuti localmente.

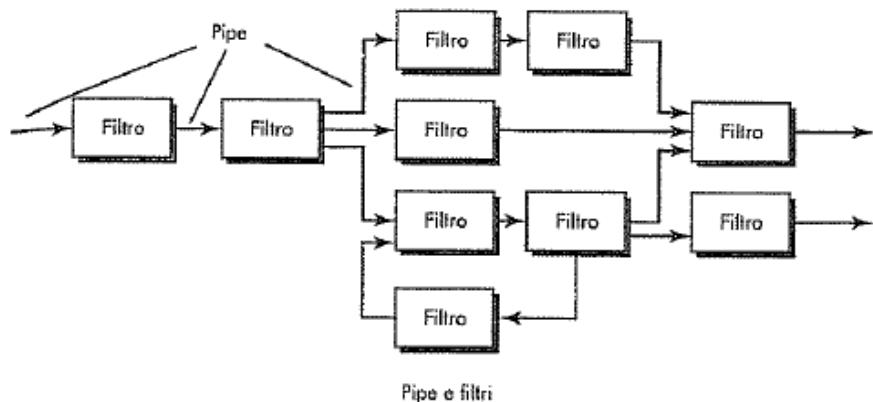
Tipicamente la soluzione è quella di strutturare il sistema attorno ad un *repository* comune tra tutti i componenti del sistema (ogni componente poi può essere basato su un repository interno), dove tutti i dati sono contenuti nel repository o database centrale che è accessibile da tutti i sottosistemi. L'interazione tra i componenti avviene accedendo e modificando i dati nel repository. Questa è la tipica soluzione in strumenti CAD and CASE.



Pro	Contro
<ul style="list-style-type: none"> - Metodo semplice per scambiare grosse moli di dati tra i componenti senza doverli duplicare in ciascun sottosistema - Un sottosistema non si interessa di come gli altri sistemi utilizzeranno i dati (disaccoppiamento) - Backup, controllo degli accessi, recovery sono centralizzati - Attività di gestione dei dati semplificata - Facile integrare nuovi componenti se questi sono "a conoscenza" del formato dato che la struttura del repository è pubblica 	<ul style="list-style-type: none"> - Ci deve essere un accordo tra i sottosistemi sul formato dei dati e struttura del repository - Difficile riuso - Possibili problemi di performance - Modifiche al formato dei dati possono diventare complesse - La centralizzazione dei dati può creare problemi di affidabilità - Attività di gestione dei dati poco flessibile e difficile evoluzione - La distribuzione dei dati per migliorare la performance complica la gestione

➤ Architettura a flusso di dati

Questa architettura viene applicata quando i dati di input devono essere trasformati tramite



una serie di componenti (calcolo o manipolazione) in dati di output. Uno schema a pipe e filtri ha un insieme di componenti chiamati **filtri**, connesse da **pipe** che trasmettono i dati da un componente al successivo. Ogni filtro lavora in modo indipendente dagli altri seguendo la direzione del flusso ed è progettato per attendersi dati di input con una certa forma e produrre i dati di output (per il filtro successivo) nella forma specificata. Ogni filtro non richiede alcuna conoscenza del

funzionamento dei filtri che lo procedono o seguono.

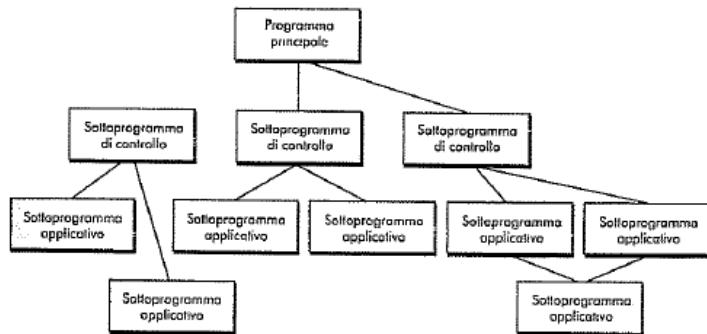
I filtri, inoltre, possono essere implementati in parallelo e riusati. Se il flusso di dati degenera in un'unica linea di trasformazioni, si dice che è un filtro batch sequenziale, schema che accetta un blocco di dati e poi applica una serie di componenti (filtri) per trasformarli. Esempi di pipe and filter sono: shell comandi Unix, Compilatori.

➤ Architettura call and return

Questo stile consente ad un architetto di sistema di ottenere una struttura di programma relativamente semplice da modificare e da trasformare in scala. Esistono due sottostili che rientrano in questa categoria:

- o **Architettura a programma principale/sottoprogramma**. Questa struttura dei programmi decomponete le funzioni in una gerarchia di controllo dove un

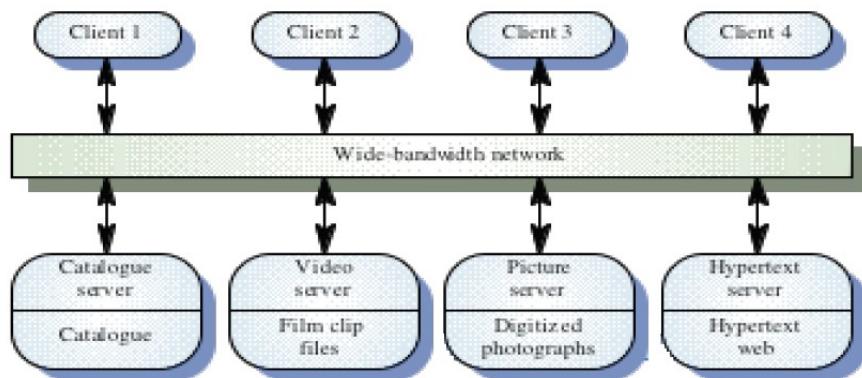
programma principale richiama una serie di programmi componenti che a loro volta possono richiamare altri componenti.



- o **Architettura a chiamata di procedure remote.** I componenti di un'architettura a programma principale/sottoprogramma vengono distribuiti sui computer di una rete.

Modello Client-Server: Rientra nello stile d'architettura a chiamata di procedura remota. Tale modello racchiude funzionalità del sistema distribuite *logicamente* all'interno di un insieme di componenti **server**, i quali forniscono servizi specifici ad altri sottosistemi (stampa, gestione dati, ecc.); i **client** invece sono quei componenti che hanno necessità di accedere ai servizi.

L'ambiente è distribuito, e la rete permette ai client di accedere ai server. Non necessariamente client e serventi si trovano su macchine differenti.



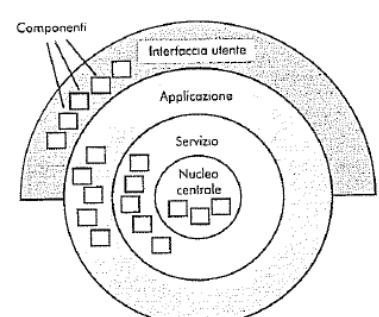
Pro	Contro
<ul style="list-style-type: none"> - Paradigma particolarmente adatto in ambito distribuito (ovvero tanti elaboratori, ciascuno dedicato ad un servizio) - Disaccoppia fortemente parti del sistema (serventi da clienti) rendendo facile aggiungere componenti client e server senza coinvolgere altri sottosistemi - Può richiedere hardware meno costoso rispetto ad un sistema centralizzato 	<ul style="list-style-type: none"> - Modifiche ad un servente possono portare a revisioni importanti dei clienti particolarmente difficoltose in caso di distribuzione - Ogni client gestisce i propri dati e li trasferisce ai server per le elaborazioni, tale scambio di dati può essere inefficiente - Management dati (per backup, sicurezza) ridondante su ciascun server - Non esiste un registro di nomi e servizi, può essere difficile trovare i servizi disponibili

➤ Architetture orientate agli oggetti

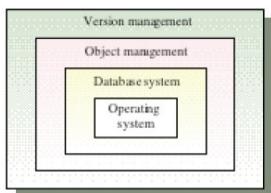
I componenti di sistema encapsulano i dati e le operazioni che devono essere applicate per manipolarli. La comunicazione e il coordinamento fra i componenti viene ottenuto tramite il passaggio di messaggi.

➤ Architetture a livelli

Vengono definiti vari livelli, ognuno dei quali svolge operazioni che diventano via via più vicine al set di istruzioni della macchina. Al livello più esterno i componenti servono le operazioni dell'interfaccia



utente. Al livello più interno i componenti svolgono l'interfacciamento il sistema operativo. I livelli interni forniscono le funzionalità di servizio e del software applicativo.



Modello stratificato: Sistema strutturato a set di strati (o macchine astratte), ogni livello fornisce un servizio più “astratto” ai livelli superiori. La stratificazione favorisce sviluppo incrementale ed evoluzionario, quando l'interfaccia di un livello cambia, solo il livello adiacente è influenzato. Non è sempre semplice definire struttura a strati per un sistema.

Questi tipi di architettura sono solo un sottoinsieme di quelli disponibili per un progettista software. Dopo che l'ingegneria dei requisiti avrà mostrato le caratteristiche e i vincoli del sistema da realizzare, potrà essere scelto lo stile dell'architettura o la combinazione di stili che sia più adatta alle caratteristiche ed ai vincoli. In molti casi possono essere adatti più schemi o occorre progettare e valutare più stili di architettura.

Stili di controllo

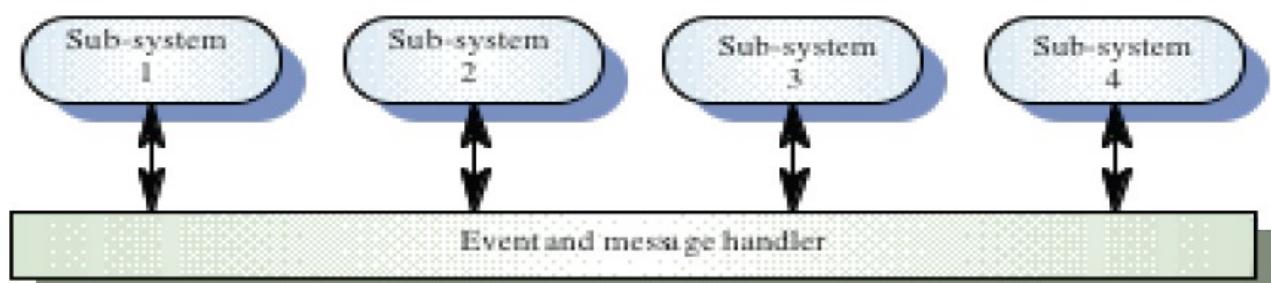
Il controllo si riferisce a come il flusso del controllo si propaga tra i vari sottosistemi di un sistema software. Principalmente si può optare tra due stili principali:

- **Controllo centralizzato:** esiste un sottosistema che controlla gli altri ed ha anche il compito di far partire ed arrestare il sistema.
Es. *Modelli call-return:* Il sistema parte con una routine principale (main) che passa il controllo alle altre routine le quali potranno fare lo stesso. Ogni routine si aspetta comunque di riavere il controllo “indietro” e dovrà dunque restituirlo. (No concorrenza).
Modelli con gestore: Esiste un sottosistema (gestore) che può far partire concorrentemente più attività che possono procedere in parallelo.
- **Controllo basato su eventi:** ogni sottosistema è capace di rispondere indipendentemente ad eventi esterni.

Modelli event-driveri: L'andamento del sistema e del processo è guidato da eventi generati esternamente (Sistemi guidati da eventi). Sono particolarmente adatti nel caso di sistemi interattivi, dove gli eventi esterni “scatenano” l'esecuzione di particolari procedure. Tipicamente diversi modelli possono essere pensati per gestiti gli eventi:

- *Modello Broadcast o Publish/Subscribe:* Un evento è inviato a tutti i sottosistemi, ogni sottosistema che è in grado di gestire l'evento lo gestisce e i sottosistemi decidono se l'evento è di interesse.
- *Modello ad interrupt:* Usati in sistemi realtime.

Esempi: fogli elettronici, sistemi di produzione.



Pro	Contro
- Nuovi sottosistemi possono facilmente essere aggiunti	- Un sottosistema non sa chi gestisce gli eventi che genera

Software ad architettura riflessiva

Nell'ambito della progettazione delle architetture software, ci soffermiamo sui software che utilizzano come nucleo lo stile di riflessione computazionale. Essa è la capacità di un sistema

(detto riflessivo) di contenere strutture che rappresentano aspetti di sé stesso (metadata), e che gli permettono di supportare azioni su sé stesso. In altre parole, è la capacità di un programma di controllare se stesso e, se necessario, di adattare il proprio comportamento durante l'esecuzione.

Un metadata o dato su un altro, è un'informazione che descrive un insieme di dati. Un esempio tipico di metadati è costituito dalla scheda del catalogo di una biblioteca, la quale contiene informazioni circa il contenuto e la posizione di un libro, cioè dati riguardanti i dati che si riferiscono al libro.

Struttura e funzioni di un sistema riflessivo

Un sistema riflessivo è rappresentabile come un sistema a livelli in cui il livello sottostante, il **livello base** (il libro nel nostro esempio), non conosce che esistono livelli superiori, i **livelli meta** (il catalogo nel nostro esempio).

Il livello superiore è in grado di **ridefinire** alcune operazioni del livello sottostante e di alterare le attività ed il flusso di esecuzione del livello inferiore, ai fini di cambiare la natura delle operazioni eseguite e per inserire nuove funzionalità; è anche capace di **intercettare** le operazioni del livello inferiore ed **ispezionare** il livello inferiore. Ispezionare serve a conoscere la struttura del livello inferiore (es. classi, attributi, metodi) ed il valore di variabili a runtime.

La funzione principale di un sistema di metadati è quella di consentire il raggiungimento dei seguenti obiettivi:

- **Ricerca**, che consiste nell'individuare l'esistenza di un documento;
- **Localizzazione**, ovvero rintracciare una particolare occorrenza del documento;
- **Selezione**, realizzabile analizzando, valutando e filtrando una serie di documenti;
- **Interoperabilità semantica**, che consiste nel permettere la ricerca in ambiti disciplinari diversi grazie a una serie di equivalenze fra descrittori;
- **Gestione risorse**, ossia gestire le raccolte di documenti grazie all'intermediazione di banche dati e cataloghi;
- **Disponibilità**, ovvero ottenere informazioni sull'effettiva disponibilità del documento

I campi di una collezione di metadati sono costituiti da informazioni che descrivono le risorse informative a cui si applicano, con lo scopo di migliorarne la visibilità e facilitarne l'accesso. I metadati infatti permettono il recupero di documenti primari indicizzati attraverso le stringhe descrittive contenute in **record**: schede in cui vengono rappresentate le caratteristiche più significative o le proprietà peculiari dei dati in questione, così che la loro essenza possa essere catturata da un'unica concisa descrizione, che, in modo sintetico e standardizzato, fornisce a sua volta una via di recupero dei dati stessi.

Tipi di metadato

Attualmente, in letteratura il termine "metadato" è utilizzato quasi esclusivamente in riferimento al contesto dell'informazione elettronica in rete: i metadati sono comunemente intesi come un'amplificazione delle tradizionali pratiche di catalogazione bibliografica in un ambiente elettronico.

Nel contesto dei documenti digitali, il termine metadato adotta il significato che gli è stato affidato dall'informatica e dalla filosofia. Dall'informatica deriva la restrizione del concetto di "dato", limitando il suo dominio semantico ai soli dati digitali e discreti che vengono gestiti da un computer.

I metadati possono essere distinti in vari modi; uno dei più diffusi li raggruppa in tre macrocategorie:

- **Metadati descrittivi**: servono per l'identificazione ed il recupero degli oggetti digitali; sono costituiti da descrizioni dei documenti fonte, o dei documenti nati in formato digitale, risiedono generalmente nelle basi dati dei sistemi di Information Retrieval all'esterno dell'archivio digitale, e sono collegati a quest'ultimo tramite appositi collegamenti.
- **Metadati amministrativi e gestionali**: evidenziano le modalità di archiviazione e manutenzione degli oggetti digitali nel sistema di gestione dell'archivio digitale, e sono

necessari per una corretta esecuzione delle relative attività. Nel mondo digitale, data la labilità dell'informazione elettronica, questi tipi di metadata assumono un'importanza preponderante ai fini della conservazione permanente degli oggetti digitali: essi possono documentare i processi tecnici associati alla conservazione permanente, fornire informazioni sulle condizioni e i diritti di accesso agli oggetti digitali, certificare l'autenticità e l'integrità del contenuto, documentare la catena di custodia degli oggetti, identificarli in maniera univoca.

- **Metadati strutturali:** collegano le varie componenti delle risorse per un'adeguata e completa fruizione, che spesso avviene attraverso la mappatura di schemi di metadata diversi. Questi metadata inoltre forniscono dati di identificazione e localizzazione del documento, come il codice identificativo, l'indirizzo del file sul server, l'archivio digitale di appartenenza e il suo indirizzo Internet.

Riflessione in JAVA

La riflessione permette di costruire programmi che

- ❖ Introspezionano strutture e dati di programmi, cioè analizzano la propria interiorità di codice;
- ❖ Prendono decisioni in base ai risultati dell'introspezione;
- ❖ Cambiano il comportamento, la struttura o i dati del programma in base alle decisioni prese.

Java fornisce un supporto parziale per la riflessione (solo per l'introspezione) attraverso le librerie `java.lang` e `java.lang.reflect`. Tramite tali librerie di Java possiamo:

- Scoprire campi, metodi e costruttori di una classe (non noti a compile time)
- Istanziare una classe (il cui nome è noto a compile time)
- Invocare metodi
- Ispezionare e cambiare il contenuto dei campi (conosciuti solo a run-time)
- Scoprire la superclasse e le interfacce di una classe

Innanzitutto, ricordiamo quali sono le tre fasi nel ciclo di un'applicazione:

- **Design Time:** la fase di progettazione del codice come, ad esempio, la creazione di un form o di una classe e la scrittura del codice;
- **Compile Time:** fase di compilazione del progetto;
- **Run Time:** fase di esecuzione del software.

Per l'**invocazione riflessiva** di tutte le informazioni di una determinata classe Object utilizziamo il metodo `getClass()`, il quale ritorna una istanza c di Class. Tale istanza c rappresenta la classe dell'oggetto sui cui è evocato `getClass()`. Tramite l'istanza c di Class possiamo avere tutte le informazioni, attraverso introspezione, della classe rappresentata.

Se invece vogliamo attuare un'**invocazione dinamica**, tramite `invoke()` riusciamo a chiamare un metodo di un oggetto a runtime senza specificare a design time di quale metodo si tratti, ovvero, a design time non conosciamo il nome metodo né il nome classe.

`getMethods()` con la variante `getDeclaredMethods()` restituisce tutti i metodi pubblici di una classe, e il valore di ritorno è di tipo `java.lang.reflect.Method`. `getMethod()` restituisce il metodo il cui nome è passato come argomento

- Es. Vogliamo invocare un metodo `show()` su una istanza di cui non conosciamo la classe (perché sviluppata da terze parti). Senza riflessione dovremmo usare il metodo `invoke()` per ogni specifica classe. Le conseguenze sono: codice di invocazione dipendete dalla classe e codice da modificare ogni volta che è prodotta una classe. Con la riflessione, invece, l'invocazione è resa indipendente dalla classe a cui il metodo `show()` appartiene.

```
//cls rappresenta la classe dell'istanza o
Class cls = o.getClass();
//m rappresenta il metodo show
Method m = cls.getMethod("show", new Class[]
{String.class});
//invocazione dinamica del metodo
```

```
m.invoke(o, new Object[] {ms});
```

Esempio. Vogliamo invocare i metodi `*get()` su una istanza di cui non conosciamo la classe e scrivere (su disco) i valori di ritorno del metodo chiamato

```
Class cls = o.getClass();
Method mt[] =
cls.getMethods();
for(int i = 0; i < mt.length; i++){
}

if(mt[i].getName().startsWith("get"))
try {
    Object v =
mt[i].invoke(o,null);
```

Per quanto riguarda il **caricamento dinamico**, invece, tramite `forName(..)` riusciamo a caricare una classe a runtime senza specificare a design time di quale classe si tratti. Tramite `newInstance(...)` riusciamo a creare un oggetto della classe che conosciamo solo a runtime.

Esempio. Vogliamo caricare una classe di cui conosciamo il nome solo a runtime (non a design time) e creare una sua istanza, al fine di invocare su tale istanza un metodo

```
//carica una classe e ritorna la corrispondente
rappresentazione
Class cls = Class.forName(nomeClasse);
//crea una istanza della classe cls
Object o = cls.newInstance();
//invoca un metodo dell'istanza creata
m.invoke(o, null);
```

Pro	Contro
<ul style="list-style-type: none">- Fornisce un modo per collegare ad un programma nuove classi, non conosciute a compile time- Permette di manipolare oggetti di una qualsiasi classe senza inserire nel codice la classe, quindi rinviando il binding (collegamento fra un entità software e il suo corrispettivo valore) fino a runtime.	<ul style="list-style-type: none">- Invocare metodi o accedere a campi con i meccanismi riflessivi è molto più lento che col codice diretto

Metaoggetto VerboseMO

La classe VerboseMO è in grado di catturare le operazioni di altri oggetti, e viene invocato quando l'oggetto associato deve essere creato. La classe VerboseMO deve essere una sottoclasse di `javassist.reflect.Metaobject` ed implementare il costruttore ed i metodi `trapFieldRead()`, `trapFieldWrite()` e `trapMethodcall()`. I metodi di VerboseMO ricevono il controllo dall'applicazione (ovvero catturano l'esecuzione) quando viene fatta una lettura di un campo, una scrittura o un'invocazione di un metodo, rispettivamente.

Alcuni vantaggi offerti dalla riflessione Javassist:

- Fornisce il modo di comporre vari aspetti di un'applicazione (funzionalità, sincronizzazione, distribuzione, ecc.), mantenendo i codici sorgenti separati;
- La separazione tra vari aspetti di un'applicazione rende più semplice lo sviluppo, il riuso e l'evoluzione dei singoli aspetti.

```

import javassist.tools.reflect.*;

public class VerboseMO extends Metaobject {
    // cattura l'istanziazione
    public VerboseMO(Object self, Object[] args) {
        super(self, args); // costruisce l'oggetto del baselevel
        System.out.println("* Costruito: " + self.getName());
    }

    // cattura la lettura di un campo
    public Object trapFieldRead(String name) {
        System.out.println("* Catturata lettura campo " + name);
        return super.trapFieldRead(name);
    }

    // cattura la scrittura di un campo
    public void trapFieldWrite(String name, Object value) {
        System.out.println("* Catturata scrittura " + name+ " valore: "+value);
        super.trapFieldWrite(name, value);
    }
}

...
// cattura l'invocazione di un metodo
public Object trapMethodcall(int identifier, Object[] args)
throws Throwa
System.out.println("* Catturato metodo: " + getMethodName(identifier)
"O della classe baselevel "+getClassMetaobject().getNam
// chiama il metodo del baselevel
Object ob = super.trapMethodcall(identifier, args);
// il metodo a livello base è terminato
System.out.println("* Il baselevel ha terminato l'esecuzione");
// restituisce il parametro di ritorno
return ob;
}

```

Paradigma Object-Oriented

In ingegneria del software, l'espressione paradigma orientato agli oggetti o paradigm object-oriented (OO) si riferisce a un insieme di concetti introdotti dai linguaggi di programmazione orientati agli oggetti e in seguito estesi a numerosi altri contesti della information technology. Le caratteristiche fondamentali della programmazione ad oggetti sono:

- Le classi encapsulano dati ed operazioni che manipolano i dati;
- Gli oggetti possiedono uno stato e reagiscono eseguendo del codice quando ricevono una invocazione;
- La comunicazione tra componenti (classi e oggetti) avviene tramite passaggio di messaggi (ovvero invocazioni di operazioni).

Sistemi orientati agli oggetti nel processo di sviluppo software

Lo sviluppo di sistemi orientati agli oggetti mira a descrivere un software nelle sue diverse fasi (analisi, progettazione, codifica) in termini di **classi**. Tali sistemi sono documentati focalizzando sulle classi ed usando diagrammi di classi. La classe è spesso pensata come l'entità riusabile più piccola (ovvero la **granularità** del riuso è a livello di classe). I svantaggi della programmazione OO sono:

- o Le conseguenze dell'uso delle classi possono non essere chiare.
- o La progettazione può essere non corretta ai fini del riuso o di altre proprietà.
- o Anche se ben progettate, le classi sono sì entità riusabili ma esse sono troppo piccole per ottenere grandi vantaggi dal loro riuso.

Quello che si spera (e si tenta) di riusare è un insieme di classi, ovvero la soluzione ad un certo problema, che molto spesso è costituita da più di una classe.

Entità con granularità più grande, **architetture di sistemi ad oggetti**, permettono di ridurre i costi di sviluppo, poiché cercano di **riusare** insiemi di classi e di produrre sistemi più facili da evolvere.

Elementi OO

I concetti fondamentali del paradigma object-oriented includono:

- o **Classe**: Nella programmazione orientata agli oggetti una classe è un costrutto di un linguaggio di programmazione usato come modello per creare oggetti. Il modello comprende **attributi** e **metodi** che saranno condivisi da tutti gli oggetti creati ((**istanze**), ovvero oggetti istanziati).
Una classe può rappresentare una persona, un luogo, oppure una cosa, ed è quindi l'astrazione di un concetto, implementata in un programma per computer. Fondamentalmente, essa definisce al proprio interno lo stato e il comportamento dell'entità di cui è rappresentazione. I dati che ne descrivono lo stato sono memorizzati nelle variabili membro, mentre il comportamento è descritto da blocchi di codice riutilizzabile chiamati metodi.
- o **Oggetto**: Con oggetto, in informatica ed in particolar modo nell'ambito della programmazione, si intende nella maniera più generica una regione di memoria allocata.
Poiché i linguaggi di programmazione usano variabili per accedere agli oggetti, i termini oggetto e **variabile** sono spesso usati in alternativa tra loro. In ogni caso finché un'area di memoria non è allocata nessun oggetto può esistere.
- o **Incapsulamento**: In informatica si definisce encapsulamento (o encapsulation) la tecnica di nascondere il funzionamento interno - deciso in fase di progetto - di una parte di un programma, in modo da proteggere le altre parti del programma dai cambiamenti che si produrrebbero in esse nel caso che questo funzionamento fosse difettoso, oppure si decidesse di implementarlo in modo diverso. Per avere una protezione completa è necessario disporre di una robusta **interfaccia** che protegga il resto del programma dalla modifica delle funzionalità soggette a più frequenti cambiamenti.
- o **Ereditarietà**: l'uso dell'ereditarietà dà luogo a una gerarchia di classi. Il suo funzionamento verrà spiegato in seguito.

- o **Polimorfismo:** metodi con lo stesso nome che si comportano diversamente rispetto al tipo di dato ricevuto in input.

Linguaggi di programmazione a oggetti (OOP)

I concetti fondamentali del paradigma object-oriented furono inizialmente introdotti nei linguaggi di programmazione. La programmazione orientata agli oggetti rappresenta tuttora il paradigma di programmazione dominante nell'industria del software, e molte nuove tendenze stanno emergendo come sviluppo o estensione di questo paradigma.

La programmazione orientata agli oggetti (OOP, Object Oriented Programming) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. È particolarmente adatta nei contesti in cui si possono definire delle relazioni di interdipendenza tra i concetti da modellare (contenimento, uso, specializzazione). Un ambito che più di altri riesce a sfruttare i vantaggi della programmazione ad oggetti è quello delle interfacce grafiche.

Tra gli altri vantaggi della programmazione orientata agli oggetti:

- Fornisce un supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre;
- Permette una più facile gestione e manutenzione di progetti di grandi dimensioni;
- L'organizzazione del codice sotto forma di classi favorisce la modularità e il riuso di codice.

Linguaggi di modellazione a oggetti

Un linguaggio di modellazione (modeling language in inglese) è un linguaggio formale che può essere utilizzato per descrivere (modellare) un sistema di qualche natura.

I linguaggi di modellazione consentono di costruire modelli di sistemi software come strumenti di analisi e progetto. La diffusione della programmazione a oggetti ha portato all'emergere di numerosi approcci alla modellazione che fanno uso dei concetti fondamentali di classe, oggetto, ereditarietà e così via. Il linguaggio di modellazione object-oriented dominante è UML.

Database a oggetti

Il successo del paradigma object-oriented nella programmazione ha portato a numerosi tentativi di applicare lo stesso paradigma nel contesto dei Database Management System. Sebbene sia opinione diffusa che lo standard dominante nel settore continuerà a essere quello relazionale, non mancano proposte di "OODBMS" (Object-Oriented Database Management System).

Qualità del software nei sistemi ad oggetti

La definizione della qualità del software si è centrata sulla *conformità ai requisiti, funzionali e prestazionali, enunciati esplicitamente, agli standard di sviluppo esplicitamente documentati e alle caratteristiche implicite che è lecito aspettarsi da un prodotto professionale*. La definizione ha il merito di sottolineare tre punti importanti:

1. I requisiti sono il fondamento su cui misurare la qualità. La mancanza di conformità ai requisiti è una mancanza di qualità.
2. Gli standard specificati definiscono i criteri a cui attenersi nello sviluppo del software. Se non si seguono tali criteri, è quasi certo che ne deriva una qualità insufficiente.
3. Esiste una famiglia di *requisiti impliciti* spesso taciti (ad esempio la facilità di manutenzione). Se il software rispetta i requisiti espliciti ma non quelli impliciti, la qualità del prodotto è dubbia.

La qualità del software è un complesso insieme di fattori che vanno secondo il tipo di applicazione e il committente.

Caratteristiche qualitative della Object-Oriented

Un sistema orientato agli oggetti deve rispondere alle seguenti caratteristiche per essere un software di alta qualità:

- **Semantica:** Risponde alla domanda: *Cosa risolve una classe?* Bisogna dunque seguire questi fattori:
 - Scegliere il nome della classe in modo che esso sia significativo
 - Descrivere il contesto per cui la classe è implementata
 - Il lavoro di progettazione, modifica o riuso è facilitato se vi è corrispondenza tra ciò che la classe implementa e quello che il nome suggerisce
- **Correttezza:** Risponde alla domanda: *Quando una classe smette di comportarsi bene?* Bisogna dunque seguire questi fattori:
 - Una classe funziona correttamente solo sotto certe condizioni
 - Documentare tali condizioni, esplicitando dipendenze da variabili, (range di) valori dei parametri in ingresso e uscita, flusso di controllo
 - Quando si riusa la classe bisogna verificare se le ipotesi su cui l'implementazione si basa sono soddisfatte
- **Responsabilità:** Non sviluppare "God classes". Bisogna dunque seguire questi fattori:
 - La granularità per il riuso è la classe, classi piccole sono più riusabili
 - Separare compiti (o responsabilità) diversi in classi distinte permette di ottenere classi aventi alta coesione
 - Alta coesione logica tra sotto-compiti implica maggiore facilità e probabilità di comprensione, riuso, evoluzione
- **Information hiding:** Risponde alla domanda: *Quanto si conosce di una classe?* Bisogna dunque seguire questi fattori:
 - Ciò che non contribuisce alle caratteristiche essenziali di una classe deve essere nascosto (gli attributi ed alcuni metodi sono private). Sono i metodi che forniscono ciò che si vuol far conoscere all'esterno
 - Avere caratteristiche nascoste limita la propagazione dei cambiamenti all'esterno
- **Coesione:** Risponde alla domanda: *Quanti compiti ha una classe?* Bisogna dunque seguire questi fattori:
 - Indica la connessione degli elementi all'interno di un modulo
 - Una classe è coesa se implementa un solo piccolo compito, quindi tutti i suoi elementi (metodi) contribuiscono a realizzare il singolo compito (lo stesso vale per la coesione di un metodo)
 - Classi poco coese (usate ad es. per visualizzare, trasformare e registrare dati) sono difficili da comprendere e da riusare

Classi in relazione

La **legge di Demetra** dice: *Ogni unità di programma dovrebbe conoscere solo poche altre unità di programma strettamente correlate; ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente.*

Questi due principi possono essere riassunti col motto "**non parlate con gli sconosciuti**".

Una classe dovrebbe avere una conoscenza limitata delle altre classi: solo di quelle strettamente legate ad essa; l'accoppiamento indipendente tra classi facilita la loro comprensione, sostituzione, riuso ed evoluzione.

Qui l'accoppiamento è rivelato dal numero di classi usate.

Riusare o evolvere una classe risulta difficile se essa implementa molte invocazioni a metodi di altre classi; il codice diventa più difficile da comprendere e modificare.

Qui l'accoppiamento è rivelato:

- o dal numero di classi usate;
- o dal numero di invocazioni a metodi diversi di altre classi;
- o dal numero di invocazioni a metodi (anche allo stesso metodo).

Eliminare una classe o evolvere la sua interfaccia implica modificare tutte le classi che la usano: tanto lavoro quando i suoi metodi sono invocati da molte classi.

Rimedi per le classi

Se una classe è poco coesa → Dividerla in più classi;

Se una classe invoca molte altre classi → Probabilmente fa molte cose, dividerla in più classi;
Se l'accoppiamento tra due classi è molto alto → Probabilmente hanno lo stesso obiettivo, fondere le due classi.

Ereditarietà nella OO

In informatica l'ereditarietà è uno dei concetti fondamentali nel paradigma di programmazione a oggetti. Essa consiste in una relazione che il linguaggio di programmazione, o il programmatore stesso, stabilisce tra due classi.

In generale, l'uso dell'ereditarietà dà luogo a una gerarchia di classi; nei linguaggi con ereditarietà singola, si ha un albero se esiste una superclasse "radice" unica di cui tutte le altre sono direttamente o indirettamente sottoclassi (come la classe Object nel caso di Java) o a una foresta altrimenti; l'ereditarietà multipla definisce invece una gerarchia a grafo aciclico diretto.

Riuso di classi

Spesso si ha bisogno di classi simili, e quindi riusare classi esistenti per gestire attributi e metodi leggermente diversi. Copiare la classe originaria e modificarne attributi o metodi non è pratico poiché avremo una proliferazione di classi e il programmatore deve fare tante attività. Il riuso di classi esistenti deve avvenire senza dover modificare codice esistente (e funzionante) e in modo semplice per il programmatore. Con questi bisogni entra in gioco l'ereditarietà.

Scopi dell'ereditarietà

Attraverso l'ereditarietà è possibile definire una nuova classe indicando solo cosa ha in più rispetto ad una classe esistente; è possibile aggiungere attributo e metodi nuovi o modificare quelli esistenti.

Es. La classe "Persona" ha come attributi "Nome" e "Cognome", più vari metodi. La classe "Studente" dovrebbe avere tutto ciò che "Persona" fornisce (attributi e metodi) ed inoltre deve avere nuovi attributi e metodi come "Esami", "Voti", ecc. La classe "Studente" eredita da "Persona" con
`public class Studente extends Persona {...}`, così "Studente" è sottoclasse di "Persona", e "Persona" è superclasse di "Studente".

Elementi dell'ereditarietà

Se la classe B eredita dalla classe A, si dice che B è una **sottoclasse** di A e che A è una **superclasse** di B. Denominazioni alternative equivalenti, sono *classe madre* o *classe base* per A e *classe figlia* o *classe derivata* per B. La sottoclasse eredita tutti i metodi e gli attributi della superclasse e può usarli come se fossero definiti localmente. Nella sottoclasse si possono aggiungere altri metodi, ridefinire i metodi della superclasse ma non si possono eliminare i metodi o gli attributi definiti dalla superclasse.

Es. La classe "Studente" può usare tutti i metodi della classe "Persona", ad esempio `setName()`, e può aggiungere nuovi metodi, come ad esempio `getMediaVoti()`.

Visibilità

Ciò che è dichiarato come **private** è visibile solo alla classe, non alla sottoclasse; ciò che è dichiarato **public** è visibile a tutti, anche alla sottoclasse. Se voglio far vedere qualche metodo o attributo alle sottoclassi, ma non a tutti, utilizzo **protected**.

Ereditarietà semplice e multipla

A seconda del linguaggio di programmazione, l'ereditarietà può essere **ereditarietà singola** o **semplice** (ogni classe può avere massimo una superclasse diretta) o **multipla** (ogni classe può avere più superclassi dirette).

Compatibilità tra classi

L'ereditarietà permette una classificazione di tipi: una sottoclasse è un **sottotipo** compatibile con la superclasse, ovvero una sottoclasse è anche ciò che è la superclasse.

Es. Il tipo "Studente" è compatibile con il tipo "Persona". La classe "Studente" fa tutto ciò che fa la classe "Persona", inoltre la classe "Studente" fa altre cose oltre quelle che fa "Persona".

Una sottoclasse può prendere il posto della superclasse, ad esempio posso usare un'istanza di "Studente" al posto di una di "Persona". Ad esempio se prima utilizzavo `p.setName()` con p di tipo "Persona" posso sostituire `s.setName()` con s di tipo "Studente". Attenzione, non vale il contrario, non posso usare la superclasse dove usavo la sottoclasse.

Considerazioni sul codice d'esempio

La classe "Studente" eredita tutto ciò che fornisce "Persona", ridefinisce il metodo

`printAll()` (ovvero fa **override**), quindi modifica il comportamento del metodo `printAll()` ereditato. Super permette di accedere ai metodi della superclasse, infatti `super.printAll()` viene usato per invocare `printAll()` di "Persona" da "Studente".

```
public class Persona {
    protected String nome, cognome;
    public void setName(String nom,
                        String cog) {
        nome = nom;
        cognome = cog;
    }
    public void printAll() {
        System.out.println("Nome: " +
                           nome + " " + cognome);
    }
}

public class Studente extends Persona {
    private int numEsami = 0;
    private String[] esami = new String[10];
    private int[] voti = new int[10];
    public void nuovoEsame(String e, int v) {
        esami[numEsami] = e;
        voti[numEsami] = v;
        numEsami++;
    }
    public float media() {
        if (numEsami == 0) return 0;
        float sum = 0;
        for (int i=0; i<numEsami; i++)
            sum = sum + voti[i];
        return sum/numEsami;
    }
    public void printAll() {
        super.printAll();
        for (int i=0; i<numEsami; i++)
            System.out.println(esami[i] + " " +
                               voti[i]);
        System.out.println("media = " + media());
    }
}

public class Test {
    static void main(String[] args) {
        Studente s = new Studente();
        s.setName("Jeff", "Riddle");
        s.nuovoEsame("Italiano", 8);
        s.nuovoEsame("Fisica", 7);
        s.printAll();
        Persona p = s;
        p.printAll();
    }
}
```

Per la classe "Test" la variabile p è di tipo "Persona", ma punta una istanza di "Studente"; posso invocare su p il metodo `printAll()`, e verrà invocato il `printAll()` di "Studente". Non posso invocare su p il metodo `nuovoEsame()`, il tipo di p a compile-time ("Persona") non ha il metodo `nuovoEsame()`, quindi il compilatore non può far invocare `nuovoEsame()` su p (nonostante p punterà a runtime ad una istanza di "Studente").

Binding

La JVM opera con la tecnica del "**Binding Dinamico**": il codice di un metodo viene legato agli oggetti nel momento della esecuzione dalla JVM. Questo consente di chiamare per "Persona" il metodo della sua classe, e stessa cosa per "Studente".

```
public class Test {
    public static void main(String[] args) {
        Persona p = new Persona();
        Studente s = new Studente();
        Persona px;
        ...
        if (...) px = p;
        else px = s;
        px.printAll();
    }
}
```

Nell'esempio di codice a destra, `printAll()` invocato su px può assumere il comportamento definito in "Persona" o quello definito in "Studente". Il compilatore riconosce che `printAll()` è definito per px (qualunque sia l'istanza puntata), ed a runtime si decide quale `printAll()` eseguire, ovvero si ha late binding (il comportamento di `printAll()` è polimorfo).

Polimorfismo

In informatica, il termine polimorfismo (dal greco "avere molte forme") viene usato in senso generico per riferirsi a espressioni che possono rappresentare valori di diversi tipi (dette espressioni polimorfiche). In un linguaggio non tipizzato, tutte le espressioni sono intrinsecamente polimorfiche. Il termine viene associato a due significati specifici:

- nel contesto della programmazione orientata agli oggetti, si riferisce al fatto che una espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione);
- nel contesto della programmazione generica, si riferisce al fatto che il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori (polimorfismo parametrico).

Polimorfismo in OO

Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse). Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero si ha un comportamento polimorfo.

Esempio:

```
public class TestShape {
    public static void main(String[] args) {
        Shape[] s = new Shape[3];
        s[0] = new Box(2,3);
        s[1] = new Circle(4);
        s[2] = new TriangleRect(3, 4);
        for (int i=0; i<3; i++) s[i].show();
    }
}
```

Ciascun elemento di s può indicare una qualsiasi istanza la cui classe ha come super classe "Shape".

Il polimorfismo è una caratteristica fondamentale dei sistemi ad oggetti, e il late binding è tipico dei sistemi in cui esiste il polimorfismo. Senza polimorfismo dovremmo inserire uno switch sul chiamante per valutare la classe di ciascuna istanza e chiamare il metodo corrispondente a tale classe.

Dispatch (smaltimento dei metodi)

```
public class Account {
    public void setBalance(float amount) {
        if (check(amount)) balance = amount;
    }
    public boolean check(float amount) {
        return (balance+amount) >= 0
    }
}

public class SavingAccount extends Account {
    public boolean check(float amount) {
        return (balance+amount) >= 1000
    }
}

Account acc = new SavingAccount();
acc.setBalance(1234);
```

Vogliamo capire quale versione di `check()` è chiamata da `setBalance()`. Quando un metodo è chiamato su un oggetto, viene controllato il suo tipo (a runtime): `SavingAccount`. Viene cercato il metodo; se non trovato si cerca la superclasse: `Account`. Se trovato si esegue: `Account.setBalance()`. Questo chiama `check()`, come prima. Viene cercato prima su `SavingAccount`, e viene trovato. Si esegue `SavingAccount.check()`.

Classi astratte e interfacce

Classe astratta

Una classe astratta è una classe parzialmente implementata; alcuni metodi sono implementati, altri no poiché sono dichiarati **abstract**. L'utilità di avere un metodo abstract (senza implementazione) è che i client si aspettano di poterlo invocare, e forza le sottoclassi (concrete) ad implementare il metodo dichiarato prima abstract. La classe abstract non può essere istanziata, e le sottoclassi ereditano implementazioni ed attributi.

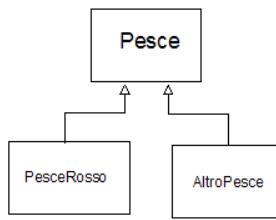
<pre>abstract class poligono{ int lati; public poligono (int lati) { this.lati = lati; } abstract public int perimetro(); }</pre>	<pre>class quadrato extends poligono{ int lati; public quadrat(int lati){ super (lati) } public int perimetro(){ int x = lato*lato; return x } }</pre>
---	--

Interfacce

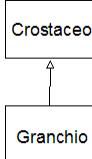
In java un'interfaccia riprende il concetto di interfaccia di sistemi orientati agli oggetti. Permettono di stabilire lo **scheletro** di una classe, e servono a dichiarare una classe e i suoi metodi senza definirli del tutto.

In altre parole, servono a fornire una forma, ma non un'implementazione (simili alle classi astratte).

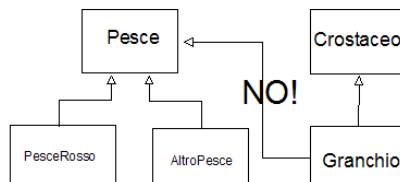
Esempio: Supponiamo di avere la superclasse "Pesce" e le sottoclassi "PesceRosso" e "AltroPesce". Definiamo una classe "Acquario, alla quale è possibile aggiungere dei pesci con il metodo `public void aggiungi (Pesce p)` che prende un parametro di tipo "Pesce" per evitare di inserire oggetti di tipo diverso.



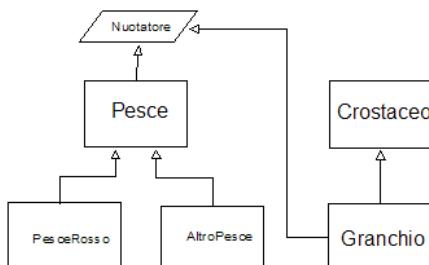
Ipotizziamo adesso di avere una classe "Granchio" che deriva dalla superclasse "Crostaceo" e lo vogliamo inserire dentro l'acquario.



Purtroppo l'acquario aggiunge solo pesci, ma noi vogliamo esplicitamente aggiungere il granchio all'acquario. La soluzione ovvia sarebbe quella di estendere acquario da pesce. Purtroppo questo non è possibile, perché Java ammette solo ereditarietà singola.



Per fortuna in java c'è una soluzione, usare le interfacce. Usando un'interfaccia è possibile implementarla sia per il pesce che per il granchio, consentendoci di mantenere le proprietà di ereditarietà.



Il metodo aggiungi() dovrà essere modificato per accettare non più il pesce, ma un generico "Nuotatore" che implementa l'intefaccia, diventando **public void aggiungi(Nuotatore n)**.

Un'interfaccia non fornisce un'implementazione per i metodi, ma permette di definire un tipo, elenca le signature dei metodi public (senza corpo dei metodi, ovvero posso solo dichiararli) e non ha attributi non inizializzati o costruttori, ma saranno dichiarati nelle classi a cui si va ad applicare. Possono contenere solo dichiarazioni di metodi, ma anche variabili unicamente final o static. I metodi dichiarati all'interno dell'interfaccia sono senza corpo, ma devono averlo nelle classi che implementano l'interfaccia.

```

class Pesce implements Nuotatore {
    public void nuota() { ... }
}
class Granchio extends Crostaceo
    implements Nuotatore {
    public void nuota() { ... }
}

```

Una classe deve implementare un'interfaccia, ovvero deve fornire un'implementazione dei metodi definiti dall'interfaccia. Sono ammesse implementazioni multiple.

Non è possibile istanziare un'interfaccia; però posso dichiarare variabili e posso assegnarli tipi compatibili, cioè tipi che implementano l'interfaccia. Nel nostro esempio, se "Pesce" implementa la variabile "Nuotatore", io posso assegnare ad una variabile di tipo nuotatore un pesce **Nuotatore p = new Pesce();** e aggiungere all'acquario un oggetto di tipo nuotatore. Così, posso invocare sull'istanza p anche un metodo della classe "Pesce" anche se p è di tipo "Nuotatore".

Tramite l'interfaccia i client sanno cosa invocare, e posso usare (per istanziazioni di oggetto e invocazioni di metodo) una qualsiasi delle implementazioni disponibili per l'interfaccia. Un client che usa un'interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia.

È possibile combinare più interfacce (situazioni dove l'oggetto x è sia di tipo a, b, c non risolvibile con l'ereditarietà dato che Java non ammette quella multipla).

Differenze tra classi abstract e interfacce

La differenza tra interfacce e classi abstract è che quest'ultime risultano più versatili perché oltre a dichiarare variabili astratte, nel suo interno troviamo anche delle definizioni reali; le interfacce però possono essere ereditate da più oggetti.

Tipi dichiarati e tipi a runtime

A compile time, un tipo dichiarato determina su quale specifica chi lo usa può contare; può essere una classe o un'interfaccia; un oggetto può essere di uno o più tipi differenti.

A run time, un oggetto ha una sola implementazione che è sempre una classe, non un'interfaccia; il tipo a runtime di un oggetto non cambia mai.

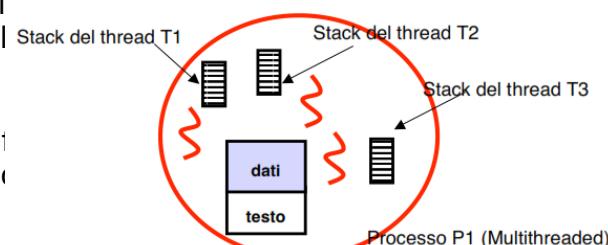
Thread

Un thread (o processo leggero) è una attività, descritta da una sequenza di istruzioni, che esegue all'interno del contesto di esecuzione di un programma. La particolarità di un thread è che procede nella sua esecuzione per portare a termine un certo obiettivo **parallelamente** agli altri thread presenti.

I thread sfruttano i core del processore della macchina che gli eseguisce; un **core** è una parte del sistema autonoma che condivide memoria con gli altri core, ognuno può eseguire un'operazione alla volta. Se non eseguiamo parallelismo sul nostro software, utilizziamo un solo core.

I thread **interagiscono** tra loro poiché possono scambiarsi dei messaggi e poiché condividono risorse di sistema (come i file) e gli oggetti dell'ambiente di esecuzione competono nell'uso della CPU e di altre risorse condivise (file, memoria).

I thread di uno stesso processo **condividono** dati e codice esecuzione ed il program counter (registro della CPU la cui valuta l'indirizzo di memoria della prossima istruzione in linguaggio privati).



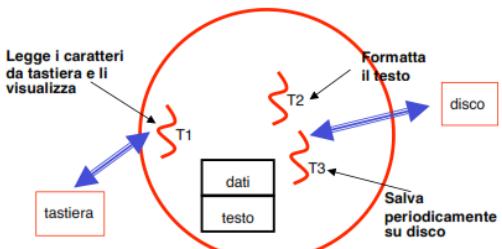
Pro	Contro
<ul style="list-style-type: none"> - Context-switch fra thread, e quindi l'attivazione di un thread, è meno oneroso rispetto all'equivalente per i processi. - Lo scambio di dati fra thread di un processo è molto semplice, poiché condividono lo stesso spazio di indirizzamento. 	<ul style="list-style-type: none"> - I problemi di sincronizzazione sono più frequenti e rilevanti

Modello a Thread

Per un processo con più thread di controllo, lo stato di avanzamento della computazione di ogni thread è dato da:

- Valore del PC (prossima istruzione da eseguire)
- Valore di SP/PSW e dei registri generali
- Contenuto dello Stack privato di quel thread (ovvero le variabili locali ad un metodo)
- Stato del thread: *pronto*, *in esecuzione*, *bloccato*

Esempio: un word processor multi-thread



Sono invece comuni a tutti i thread:

- stato dell'area testo e dati globali (gli attributi di una classe)
- stato dei file aperti e delle strutture di IPC utilizzate

Ciclo di vita dei Thread

- Un thread viene creato, riceve un identificatore (tid), ed entra nello stato **created**;
- Dopo l'avvio (tramite `start()`) il suo stato è **ready**;

- Il thread sarà schedulato per l'esecuzione dal SO (o dal sistema di supporto run-time), quando arriva il suo turno, inizia l'esecuzione passando allo stato di **running**;
- Il thread può trovarsi in stato di **blocked** quando:
 - o Ha invocato *sleep(m)*
 - o Ha invocato *wait()*
 - o È in attesa che un'operazione di I/O si completi
 - o Ha chiamato un metodo *synchronized* su un oggetto il cui lock non è disponibile
- Il thread termina la sua esecuzione e passa nello stato di **stopped**.

I thread possono essere realizzati da librerie che eseguono in modalità user. Il SO e lo **scheduler** (programma sotto forma di un algoritmo che, dato un insieme di richieste di accesso ad una risorsa, stabilisce un ordinamento temporale per l'esecuzione di tali richieste, privilegiando quelle che rispettano determinati parametri, in modo da ottimizzare l'accesso a tale risorsa e consentire così l'espletamento del servizio/istruzione o processo desiderato) non conoscono l'esistenza dei thread e gestiscono solo il processo. Lo scheduling viene effettuato dal supporto a run-time della libreria, oppure, i thread possono essere realizzati all'interno del kernel.

Thread in Java

Ogni esecuzione di una JVM dà origine ad un processo. Ogni programma in Java consiste di almeno un thread, quello che esegue il metodo *main()* della classe fornita alla JVM in fase di start up.

La JVM ha libertà sulla mappatura dei thread Java su quelli del SO. Può sfruttare il supporto multi-threading del SO sottostante (Windows) o prendersi carico della gestione dei thread interamente. In questo ultimo caso il SO vede la JVM come un processo con un unico thread (Unix).

In Java i thread sono nativi, cioè supportati a livello di linguaggio. I thread si possono implementare in due modi:

- Creando una sottoclasse della classe *Thread*
- Creando una classe che implementa l'interfaccia *Runnable*

Il programmatore non deve rendere conto alla gestione dei thread del sistema operativo, dato che in java la gestione è data dalla JVM (java virtual machine). Lo scheduler interno della JVM gestisce i thread.

Classe Thread

La **classe Thread** è una classe non astratta che fornisce vari metodi (es. *start()*, *isAlive()*, *interrupt()*) che consentono di controllare l'esecuzione del thread.

Il Procedimento per creare un thread tramite sottoclasse di Thread è:

1. La sottoclasse di Thread deve implementare il metodo *run()*
2. Bisogna creare una istanza della sottoclasse tramite new
3. Si esegue il thread chiamando il metodo *start()* che a sua volta richiama il metodo *run()*

Esercizio. Creiamo una classe che eredita dalla classe Thread e che implementa il metodo run()

```
class MioThread extends Thread {  
    int id;  
    MioThread(int n) {  
        System.out.println("Creato miothread");  
        id = n;  
    }  
    public void run() {  
        // esegue istruzioni  
        System.out.println("MioThread running");  
        for (int i = 0; i < 1000; i++)  
            if ((i%30)== 0) System.out.print(id);  
    }  
}
```

Creazione di istanze della classe MioThread ed esecuzione:
// creazione primo thread
MioThread t1 = new MioThread(1);
// creazione secondo thread
MioThread t2 = new MioThread(2);
// esecuzione parallela del primo thread
t1.start();
// esecuzione parallela del secondo thread
t2.start();

Interfaccia Runnable

Un'interfaccia Runnable è uno dei due modi disponibili in Java per creare un thread. Ciò avviene tramite la seguente procedura:

1. Implementare in una classe R l'interfaccia Runnable (che implementa il metodo *run()*)
2. Creare una istanza della classe R
3. Creare una istanza t della classe Thread passando come parametro al costruttore l'istanza della classe R
4. Invocare il metodo *start()* sul thread t, questo eseguirà il metodo *run()* della classe R

```
class MioRun implements Runnable {  
    MioRun() {  
        System.out.println("Creato oggetto");  
    }  
    public void run() {  
        // esegue istruzioni  
    }  
}
```

Per creare il thread ed eseguirlo:
Thread t = new Thread(new MioRun());
t.start();

Quest'ultima è una modalità di creazione dei thread leggermente più complessa, rispetto a quella che eredita da Thread, ma libera dal vincolo di avere una superclasse fissata. Si rivela utile, non disponendo dell'ereditarietà multipla in Java, quando vogliamo realizzare un thread per una classe che ha bisogno di ereditare da una classe dell'applicazione.

Metodi della classe Thread

- ***start()*:** chiama il metodo *run()* sull'oggetto/thread t della classe *MioThread* che abbiamo implementato. Il thread t termina quando il metodo *run()* ha finito l'esecuzione. Un thread **non** può essere fatto ri-partire, cioè il metodo *start()* deve essere chiamato solo una volta, altrimenti viene generata una *InvalidThreadStateException*.
- ***boolean isAlive()*:** permette di testare se il thread è stato avviato e non è ancora finito.
- ***setPriority(int p)*:** permette di cambiare la priorità di esecuzione del thread p, che può variare tra Thread.MIN_PRIORITY e Thread.MAX_PRIORITY (1 e 10). La priorità di default di un thread è pari alla priorità del thread che lo ha creato. Per default, il main() ha priorità 5.
- ***int getPriority()*:** ritorna la priorità del thread.
- ***yield()*:** metodo statico della classe Thread che ferma momentaneamente l'esecuzione del thread corrente per permettere ad un altro thread di eseguire.
- ***join()*:** permette di bloccare il chiamante fino a che il thread sul quale si chiama *join()* non termina la propria esecuzione. Il metodo *join(long millis)* fa aspettare il chiamante per la terminazione del thread al massimo millis milli-secondi. Lancia *InterruptedException* se *interrupt()* è chiamato sul thread.
- ***sleep(long millis)*:** metodo statico, fa aspettare il thread chiamante per millis (millisecondi). Nessun lock viene rilasciato. Lancia *InterruptedException* se *interrupt()* è chiamato sul thread.

- **stop()**: forza la terminazione del thread sul quale si invoca. Tutte le risorse usate dal thread vengono liberate (inclusi lock). È deprecato, poiché non garantisce la consistenza dell'oggetto.
- **suspend()**: blocca l'esecuzione del thread sul quale si invoca. Il thread rimane in attesa di una operazione resume(). Non libera le risorse impegnate. È deprecato, poiché può determinare situazioni di blocco critico (deadlock).
- **interrupt()**: permette di interrompere l'esecuzione del thread sul quale si invoca, solo quando lo stato dell'oggetto lo consente, cioè quando non è in esecuzione, ma in attesa di un evento. Ciò consente (a differenza del metodo stop()) di mantenere lo stato dell'oggetto consistente.
- **Thread currentThread()**: è un metodo statico della classe Thread che restituisce un identificativo del thread che sta correntemente eseguendo.
- **toString()**: restituisce una rappresentazione del thread, che include nome, priorità e gruppo.

L'uso di `stop()` e `suspend()` è sconsigliata poiché bloccano bruscamente l'esecuzione di un thread. Possono quindi generare problemi allo stato dell'oggetto, poiché una azione atomica (indivisibile) viene interrotta. Inoltre, con `suspend()`, tutte le risorse acquisite non sono rilasciate quando il thread è bloccato e possono rimanere inutilizzabili indefinitamente.

Sincronizzazione

Differenti thread della stessa applicazione condividono lo stesso spazio di memoria. È quindi possibile che più thread accedano alla stessa sezione di codice o allo stesso dato. La durata e l'ordine di esecuzione dei thread non è predicibile. Non possiamo stabilire quando lo scheduler del SO interromperà l'esecuzione di un thread per eseguirne un altro.

Esempio:

thread_1	thread_2	num
num=0;		0
genera();		0
num++;		1
	consumo();	1
	num--;	0
if (num > 0) notifica();		0

Quando più di una attività esegue, l'esecuzione è necessariamente non-deterministica e la comprensione del programma non è data dalla semplice lettura sequenziale del codice. Per esempio, una variabile che è assegnata con un valore in una istruzione di programma, può avere un differente valore nel momento in cui la linea successiva è eseguita (a causa dell'esecuzione di attività concorrenti).

Costrutto synchronized

Tale "interferenza" è eliminata con una progettazione che usa meccanismi di sincronizzazione, tipo semafori. Il **lock** su un semaforo permette di evitare l'ingresso di più thread in una **regione critica** (parte di un programma che accede a memoria o file condivisi o svolge azioni che possono portare a corse critiche) e di ottenere mutua esclusione. Usando i costrutti primitivi di Java (parola chiave synchronized) possiamo realizzare un lock su una sezione di codice o possiamo realizzare un semaforo.

`synchronized` può delimitare un frammento di codice o agire da modificatore di un metodo di una classe.

```

1 synchronized(this) {
2     num = 0;
3     genera();
4     num++;
5     if (num > 0) notifica();
6 }
```

Usato su un frammento di codice, per consentire l'esecuzione del codice ad un solo thread alla volta, necessita di una variabile su cui sarà acquisito il lock (per es. sull'oggetto `this`).

Il lock su `this` è acquisito automaticamente all'ingresso del codice (linea 1), e rilasciato automaticamente alla sua uscita (linea 6).

Quando `synchronized` è usato come modificatore di un metodo, la sua esecuzione è subordinata all'acquisizione di un lock sull'oggetto su cui si invoca tale metodo. In una classe dove tutti i metodi sono dichiarati `synchronized` un solo thread può eseguire al suo interno in un determinato momento. Si ha quindi l'associazione automatica di un lock ad un oggetto di questo tipo e l'accesso esclusivo al codice della classe.

Il costrutto `synchronized` permette di aggiornare una variabile in **modo atomico** e di creare una classe che fa attendere i thread la cui richiesta non può essere soddisfatta.

La seguente classe può essere utile per abilitare 10 esecuzioni esclusive su un oggetto:

```
public class EsecSingola {
    private int value;
    public EsecSingola() {
        value = 10;
    }
    synchronized public void reset() {
        if (value == 0) value = 10;
    }
    synchronized public void elabora() {
        if (getValue() > 0) {
            --value;
            // fai qualcosa di utile
        }
    }
    synchronized public int getValue() { return value; }
}
```

Quando un thread tenta di accedere ad un oggetto istanza di questa classe, acquisisce implicitamente il lock (se nessun thread sta eseguendo all'interno dello stesso oggetto). Il thread che detiene il lock per un oggetto di questo tipo può eseguire liberamente (senza alcun test) tutti i metodi dell'oggetto. I thread che dopo tentano di accedere allo stesso oggetto verranno sospesi, e risvegliati quando il thread che è all'interno finisce l'esecuzione del metodo. In pratica, il thread che era all'interno rilascia automaticamente il lock. Un metodo `synchronized` non è interrotto, cioè viene eseguito in modo atomico (il thread che lo esegue può essere interrotto). Se sono presenti dei metodi non `synchronized` all'interno della classe, su questi non viene acquisito il lock all'ingresso.

L'uso di `synchronized` introduce un overhead: il tempo necessario per cominciare ad eseguire il metodo è maggiore di quello di un metodo non `synchronized` (per alcune implementazioni costa 4 volte in più). Ogni volta che usiamo metodi `synchronized` riduciamo il parallelismo possibile all'interno del programma e potenzialmente costringiamo alcuni thread ad attendere.

L'uso di `synchronized` ci protegge da eventuali "interferenze" durante l'esecuzione del codice ed è quindi utile per garantire la correttezza, ma richiede una certa attenzione per prevenire ritardi e deadlock.

Metodi di sincronizzazione

In Java ogni oggetto è potenzialmente un monitor. La classe `Object` mette quindi a disposizione i metodi di sincronizzazione: `wait()`, `notify()` e `notifyAll()`. Esse possono essere invocate solo dopo che è stato acquisito un lock, cioè all'interno di un blocco o metodo `synchronized`.

- **wait()**: blocca l'esecuzione del thread invocante fino a che un altro thread invoca una `notify()` sull'oggetto. Si fa sempre dopo aver testato una condizione (ed in un ciclo, per essere sicuri che al risveglio la condizione è verificata).

```
while (! condition) // se non può procedere
    this.wait();      // aspetta una notifica
```

Il thread invocante viene bloccato, il lock sull'oggetto è rilasciato automaticamente. I lock su altri oggetti sono mantenuti (bisogna quindi fare attenzione a possibili deadlock). Un oggetto con metodi `synchronized` gestisce di per sé 2 code:

- o Una coda di lock, per i thread a cui l'accesso è escluso
- o Una coda di wait per le condizioni di attesa.

Un thread può essere in una sola delle due code. La variante `wait(long timeout)` blocca il thread per al massimo timeout millisecondi (se `timeout > 0`).

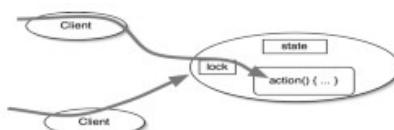
- **`notify()`**: risveglia un solo thread tra quelli che aspettano sull'oggetto in questione. Se più thread sono in attesa, la scelta di quale svegliare viene fatta dalla JVM. Una volta risvegliato, il thread compete con ogni altro (non in wait) che vuole accedere ad una risorsa protetta.
- **`notifyAll()`**: risveglia tutti i thread che aspettano sull'oggetto in questione. In pratica i thread nella coda di wait vengono trasferiti nella coda di lock ed aspettano il loro turno per entrare. `notifyAll()` è più sicura, poiché il thread scelto da `notify()` potrebbe non essere in grado di procedere e venire sospeso immediatamente, bloccando l'intero programma.

Esempio con classi “Produttore” e “Consumatore”. Una classe Produttore produce un item e lo inserisce in una classe Contenitore. Il Consumatore estrae l'item presente nel Contenitore, se esiste. Produttore e Consumatore sono 2 thread. Contenitore è la risorsa condivisa. Produttore non deve sovrascrivere l'item già presente su Contenitore, ma deve aspettare che qualcuno lo rimuova. Consumatore, una volta rimosso l'item di Contenitore, deve notificare i thread in attesa della risorsa.

Progettazione dei sistemi paralleli

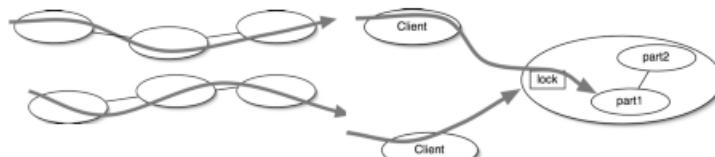
Per proteggere l'accesso a risorse (oggetti) condivise possiamo usare:

- Oggetti completamente sincronizzati: tutti i metodi sono dichiarati `synchronized` e le variabili sono private.



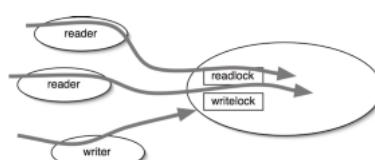
- Contenimento: uso di tecniche di encapsulamento per garantire che al massimo una attività avrà accesso ad un oggetto. Sono simili alle misure per garantire la sicurezza.

Evito che il riferimento ad alcuni oggetti sia conosciuto al di fuori di un certo numero di oggetti/thread, imponendo un unico percorso per accedere a certe risorse. Posso realizzare il contenimento tramite encapsulamento di oggetti all'interno di altri.



Per aumentare il parallelismo (eliminando qualche collo di bottiglia) possiamo realizzare:

- Divisione dei lock: anziché associare un lock ad un insieme di funzionalità di un oggetto, dispongo di diversi lock, ciascuno per una distinta funzionalità. In questo caso posso sempre avere un singola classe con tutte le funzionalità, ma più lock che regolano l'accesso dei thread. L'implementazione può essere ottenuta, per esempio, con l'uso di `synchronized` su un blocco di codice e non sui metodi della classe.
- Coppie di lock, per lettura e per scrittura: posso identificare quali operazioni sono in lettura (non modificano lo stato) e quali in scrittura e consentire più lettori contemporaneamente, ma un solo scrittore alla volta.



Gruppi di thread

Si possono raccogliere tanti thread all'interno di un gruppo e così facilitare le operazioni di sospensione o di ripartenza dell'insieme di thread con una sola invocazione.

La JVM associa un thread ad un gruppo al momento della creazione del thread. Tale associazione non può essere modificata a run-time.

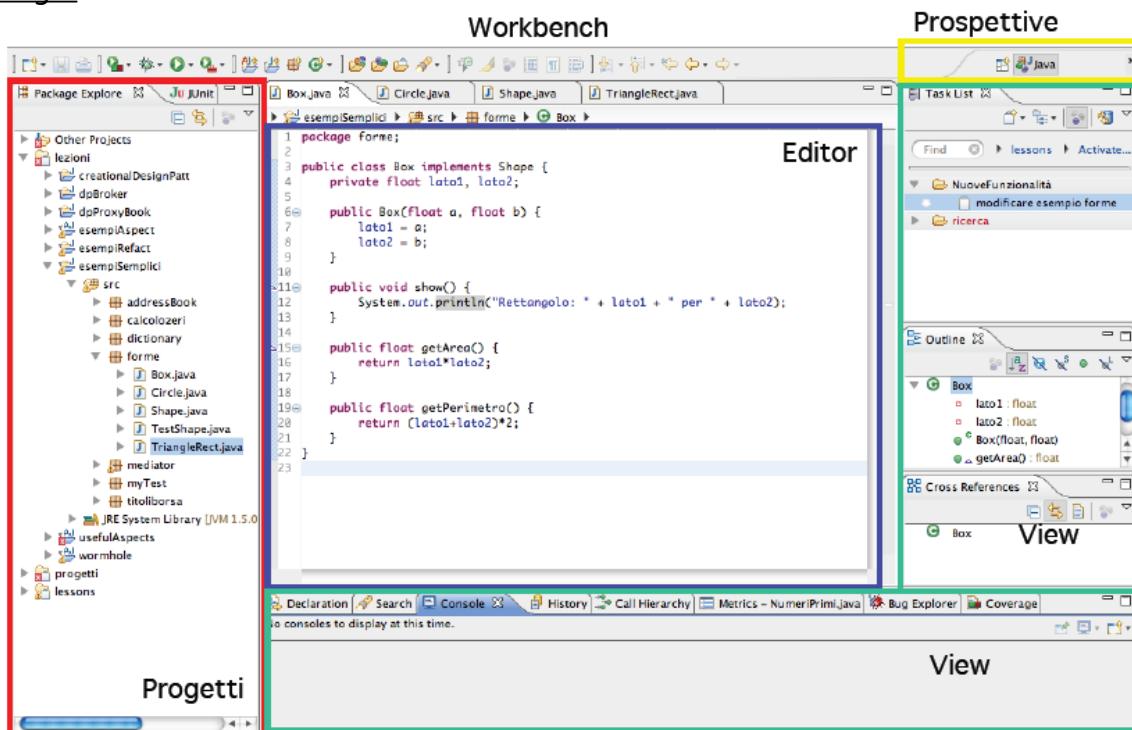
```
ThreadGroup mytg = new ThreadGroup("mio gruppo");
Thread myt = new Thread(mytg, "mio t");
```

IDE Eclipse

Eclipse è un ambiente di sviluppo per Java, C++,... Le sue caratteristiche sono:

- o Editor con viste
- o Formattazione del codice
- o Compilazione incrementale
- o Evidenziazione errori, quick fix
- o Regacotring
- o Debugging
- o Unit test
- o Tanti plug-in

Terminologia



Per cominciare ad usare Eclipse, bisogna creare un **workspace**, ovvero una directory dove sono memorizzati i file. La finestra principale di Eclipse è chiamata **workbench**, il quale gestisce vari tipi di risorse, tra cui: *progetti*, *cartelle* e *file*; i progetti sono l'unità più grande che contiene cartelle e file.

Ogni workbench Contiene più **prospettive**, la quale definiscono il set iniziale ed il layout delle **view**. Le prospettive contengono viste ed editor e controllano cosa appare nel menu e nelle toolbar. Una view è una finestra che ha una specifica funzione come progetti, gestione file, ecc.

Progetti

Un progetto Eclipse è una directory che contiene i file sorgenti ed alcuni file nascosti di informazioni per Eclipse. Per importare in Eclipse dei file Java già esistenti:

- File -> New -> Java Project
- inserire un nome progetto
- selezionare create project from existing source
- selezionare la directory

Per creare in Eclipse un nuovo progetto:

- File -> New -> Java Project
- inserire un nome progetto

Per eseguire:

- selezionare una classe che contiene il main()
- right-click e run (ctrl-alt-X J) (mac alt-cmd-X J)

Debug:

- inserire un breakpoint tramite doppio click a sinistra di una linea di codice
- eseguire tramite selezione di Debug
- l'esecuzione si blocca al breakpoint, step in, step over per andare avanti

Alcune funzionalità

- **Autocompletamento:** inserire "foo." si attiverà l'autocompletamento, selezionare un metodo e premere invio; ctrl-spazio attiva la finestra con le voci disponibili.
- **Suggerimenti:** spostando il cursore su un testo, viene visualizzato il javadoc.
- **Indentazione:** per la singola linea usare Source -> Correct Indentation; per l'intero codice Source -> Format.
- **Navigazione:** su una parola del codice Navigate -> Open Declaration (F3) porta alla definizione del metodo o classe; vai sull'ultimo codice editato tramite la freccia gialla.

Design Patterns

Nella progettazione del software, riferirsi a componenti con granularità più grande della classe, permette di ottenere un migliore riuso ed una maggiore astrazione. I design pattern sono strutture software (ovvero microarchitetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti. Tali micro-architetture specificano le diverse classi ed oggetti coinvolti e le loro interazioni.

Un design pattern (traducibile in lingua italiana come schema progettuale, schema di progettazione, schema architetturale), nell'ambito dell'ingegneria del software, è un concetto che può essere definito "*una soluzione progettuale generale a un problema ricorrente*". Esso non è una libreria o un componente di software riusabile, quanto piuttosto una descrizione o un modello da applicare per risolvere un problema che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software. Si possono paragonare ad elementi architettonici che possono essere usati nello sviluppo di software diversi, ma accomunati da problematiche simili; non a caso alcuni, come la facciata richiamano elementi architettonici tradizionali.

La nascita del "movimento" dei pattern in informatica si deve al celebre libro *Design Patterns: Elementi per il riuso di software ad oggetti* di *Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides* (1995). Grazie al successo di quest'opera, i suoi quattro autori divennero nomi talmente citati che la comunità scientifica iniziò, per brevità, a identificarli collettivamente con un nomignolo: la "banda dei quattro" (**Gang of Four** o Gof).

Esistono tanti cataloghi di design pattern, per vari contesti (sistemi centralizzati, concorrenti, distribuiti, real-time, etc.)

Definizione

I design pattern descrivono un **problema di design** ricorrente che si incontra in **specifici contesti** di progettazione e presentano una **soluzione collaudata generica** ma specializzabile. In altre parole, sono soluzioni riusabili per una certa classe di problemi ricorrenti.

Design pattern nella OO

I design pattern orientati agli oggetti tipicamente mostrano relazioni ed interazioni tra classi o oggetti, senza specificare le classi applicative finali coinvolte. Tali pattern risiedono quindi nel dominio dei moduli e delle interconnessioni. Ad un livello più alto sono invece i Pattern architettonici che hanno un ambito ben più ampio, descrivendo un pattern complessivo adottato dall'intero sistema.

Usi

- ✓ **Documentano** soluzioni già applicate che si sono rivelate di successo per certi problemi e che si sono evolute nel tempo.
- ✓ **Aiutano i principianti** ad agire come se fossero esperti.
- ✓ **Supportano gli esperti** nella progettazione di software su grande scala.
- ✓ **Evitano** di re-inventare concetti e soluzioni, riducendo il costo del software.
- ✓ Forniscono un **vocabolario** comune e permettono una **comprendione dei principi del design**.
- ✓ Analizzano le loro proprietà **non-funzionali**: ovvero, come una funzione è portata a termine, es. affidabilità, cambiabilità, sicurezza, testabilità, riuso.

Struttura

Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione:

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

La struttura di un design pattern include 5 parti fondamentali:

- **Nome:** permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern;
- **Intento:** descrive brevemente le funzionalità e lo scopo;
- **Problema (Motivazione+Applicabilità):** descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo;
- **Soluzione:** descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni;
- **Conseguenze:** indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern. Sono fondamentali in quanto possono essere l'ago della bilancia nella scelta dei pattern: le conseguenze comprendono considerazioni di tempo e di spazio, possono descrivere implicazioni del pattern con alcuni linguaggi di programmazione e l'impatto con il resto del progetto.

Nella sezione 'Problema' della descrizione di un design pattern si parla di **forze** (come in fisica), in pratica **obiettivi** e **vincoli**, spesso contrastanti, che si incontrano nel contesto di quel design pattern.

Altre parti possono essere presenti nella descrizione della struttura:

- **Esempi di utilizzo:** illustrano dove il design pattern è stato usato
- **Codice:** fornisce porzioni di codice che lo implementano

La differenza tra un algoritmo e un design pattern è che il primo risolve problemi computazionali, mentre il secondo è legato agli aspetti progettuali del software

Classificazione dei design pattern

I design pattern possono essere classificati con diversi criteri, i più comuni dei quali sono quelli che evidenziano il tipo di problema che si cerca di risolvere. Il tipo di problema può essere legato ad uno specifico dominio progettuale (telecomunicazioni, reti, software.) oppure, più comunemente, al problema progettuale in senso più ampio (nell'ingegneria del software, ad esempio, si può parlare di creazione, comportamento, navigazione di oggetti o strutture dati).

- **Pattern creazionali.** Nascondono i costruttori delle classi e mettono dei metodi al loro posto creando un'interfaccia. In questo modo si possono utilizzare oggetti senza sapere come sono implementati.

Permettono di astrarre il processo di creazione oggetti rendendo un sistema indipendente da come i suoi oggetti sono creati, composti e rappresentati. Sono importanti se i sistemi evolvono per dipendere più su composizioni di oggetti che su ereditarietà tra classi; l'enfasi va nel codificare un set fissato di comportamenti verso un più piccolo set di comportamenti fondamentali componibili. Incapsulano conoscenza sulle classi concrete che un sistema usa e nascondono come le istanze delle classi sono create e composte.

- L'*Abstract factory* (letteralmente, "fabbrica astratta") fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte degli utilizzatori di specificare i nomi delle classi concrete all'interno del proprio codice.
- Il *Factory method* ("metodo fabbrica") fornisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.
- Il *Singleton* ("singololetto") ha lo scopo di assicurare che di una classe possa essere creata una sola istanza.

- **Pattern strutturali.** Consentono di riutilizzare degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze. Se agiscono sulle classi usano l'ereditarietà per comporre interfacce o classi; se agiscono sugli oggetti descrivono modi per comporre oggetti, e tale composizione può variare a run-time. Permettono di diminuire le dipendenze tra classi o tra algoritmi.

- L'*Adapter* ("adattatore") converte l'interfaccia di una classe in una interfaccia diversa.
- *Bridge* ("ponte") permette di separare l'astrazione di una classe dalla sua implementazione, per permettere loro di variare indipendentemente.

- o Il *Composite* ("composto"), utilizzato per dare la possibilità all'utilizzatore di manipolare gli oggetti in modo uniforme, organizza gli oggetti in una struttura ad albero.
 - o Il *Decorator* ("decoratore") consente di aggiungere metodi a classi esistenti durante il run-time (cioè durante lo svolgimento del programma), permettendo una maggior flessibilità nell'aggiungere delle funzionalità agli oggetti.
 - o Il *Façade* ("facciata") permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e diverse tra loro.
- **Pattern comportamentali.** Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti. Si focalizzano sul controllo del flusso tra oggetti, descrivono le comunicazioni tra oggetti, aiutano a valutare le responsabilità assegnate agli oggetti e suggeriscono modi per incapsulare algoritmi dentro classi.
- o *Chain of Responsibility* ("catena di responsabilità") diminuisce l'accoppiamento fra l'oggetto che effettua una richiesta e quello che la soddisfa, dando a più oggetti la possibilità di soddisfarla
 - o Il *Mediator* ("mediatore") si interpone nelle comunicazioni tra oggetti, allo scopo di aggiornare lo stato del sistema quando uno qualunque di essi comunica un cambiamento del proprio stato.
 - o L'*Observer* ("osservatore") definisce una dipendenza uno a molti fra oggetti diversi, in maniera tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti vengono notificati del cambiamento avvenuto e possono aggiornarsi.
 - o *State* ("stato") permette ad un oggetto di cambiare il suo comportamento al cambiare di un suo stato interno.

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor	Descreve modi per assemblare oggetti Descreve algoritmi per il controllo del flusso

Rimanda la creazione di un oggetto ad un'altra classe

Rimanda la creazione di un oggetto ad un altro oggetto

Descreve modi per assemblare oggetti

Descreve algoritmi per il controllo del flusso

- **Pattern architetturali.** I pattern architetturali operano ad un livello diverso (e più ampio) rispetto ai design pattern, ed esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software. In questi schemi si descrivono sottosistemi predefiniti insieme con i ruoli che essi assumono e le relazioni reciproche.
- o *Model-View-Controller* (abbreviato spesso in *MVC*), che consiste nel separare i componenti software che implementano il modello delle funzionalità di business (model), dai componenti che implementano la logica di presentazione (view) e da quelli di controllo che tali funzionalità utilizzano (controller).

Abstract Factory

L'abstract factory, in italiano fabbrica astratta, è uno dei fondamentali design pattern creazionali della programmazione orientata agli oggetti.

Questo pattern è utile quando

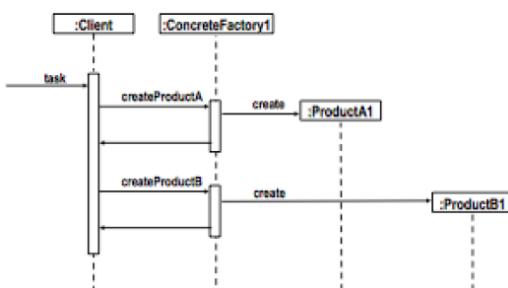
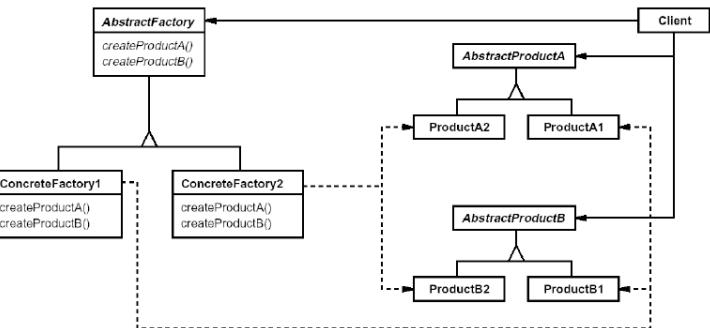
- o si vuole un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati
- o si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di prodotti
- o si vuole che i prodotti che sono organizzati in famiglie siano vincolati ad essere utilizzati con prodotti della stessa famiglia

- o si vuole fornire una libreria di classi mostrando solo le interfacce e nascondendo le implementazioni.

Scopo	Problema
<p>- Fornire una interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete. In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti.</p>	<p>- Il sistema complessivo dovrebbe essere indipendente dalle classi usate, così da essere configurabile con una di varie famiglie di classi. Le classi di una famiglia dovrebbero essere usate insieme (in modo consistente).</p> <p><i>Es.</i> Es. lo strato di interfaccia utente permette vari tipi di look-and-feel (Motif, Presentation Manager), così differenti comportamenti per accessori (widget) dell'interfaccia utente sono possibili.</p>

Struttura

- Interfaccia astratta per le famiglie di classi
- Classi per creare ciascuna famiglia di classi
- Classi concrete (per specifici look-and-feel)
- AbstractFactory è l'interfaccia per la creazione di famiglie di oggetti specifici
- ConcreteFactory implementa operazioni per creare famiglie di oggetti specifici
- AbstractProduct è l'interfaccia per una famiglia di oggetti
- Product definisce un oggetto, creato da un ConcreteFactory e che implementa l'interfaccia AbstractProduct
- Il client usa solo interfacce dichiarate da AbstractFactory e AbstractProduct



Conseguenze

Permette di usare classi consistentemente (per famiglie), le famiglie di classi sono intercambiabili facilmente e non è facile supportare nuove classi Product poiché bisogna aggiungere un metodo su AbstractFactory e su ogni ConcreteFactory.

Codice

```

interface Icon { // AbstractProductA
    void draw();
    void fill();
}
interface Text { // AbstractProductB
    public void tell();
    public void shout();
}
interface Creator { // AbstractFactory
    public Icon getIcon(); // create method
    public Text getText();
}
// ConcreteFactory
class Creator1 implements Creator {
    public Icon getIcon() {
        return new Circle();
    }
    public Text getText() {
        return new Japanese();
    }
}

class Creator2 implements Creator { // ConcretFact
    public Icon getIcon() {
        return new Box();
    }
    public Text getText() {
        return new English();
    }
}
class Circle implements Icon { // ProductA1
    public void draw() {
        System.out.print("C ");
    }
    public void fill() {
        System.out.print("O ");
    }
}
class Box implements Icon { // ProductA2
    public void draw() {
        System.out.print("[ ] ");
    }
    public void fill() {
        System.out.print("X ");
    }
}

```

Factory method

Nella programmazione ad oggetti, il Factory Method è uno dei design pattern fondamentali per l'implementazione del concetto di factories. Come altri pattern creazionali, esso indirizza il problema della creazione di oggetti senza specificarne l'esatta classe. Questo pattern raggiunge il suo scopo fornendo un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.

La creazione di un oggetto può, spesso, richiedere processi complessi la cui collocazione all'interno della classe di composizione potrebbe non essere appropriata. Esso può, inoltre, comportare la duplicazione di codice, richiedere informazioni non accessibili alla classe di composizione, o non provvedere un sufficiente livello di astrazione. Il factory method indirizza questi problemi definendo un metodo separato per la creazione degli oggetti; tale metodo può essere ridefinito dalle sottoclassi per definire il tipo derivato di prodotto che verrà effettivamente creato.

Il pattern factory può essere utilizzato quando:

- La creazione di un oggetto preclude il suo riuso senza una significativa duplicazione di codice.
- La creazione di un oggetto richiede l'accesso ad informazioni o risorse che non dovrebbero essere contenute nella classe di composizione.
- La gestione del ciclo di vita degli oggetti gestiti deve essere centralizzata in modo da assicurare un comportamento consistente all'interno dell'applicazione.

I metodi di factory sono spesso utilizzati in **toolkit** e **framework**, dove il codice delle librerie necessita di poter creare oggetti il cui tipo è implementato nelle sottoclassi delle applicazioni che utilizzano il framework.

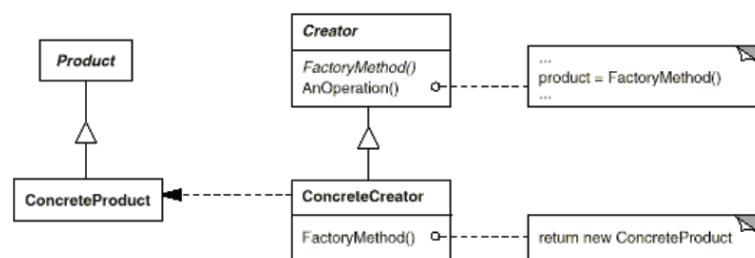
Scopo	Motivazione
<ul style="list-style-type: none"> - Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. - Il Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi. 	<ul style="list-style-type: none"> - Un framework usa classi astratte per definire e mantenere relazioni tra oggetti - Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare - Un metodo responsabile per l'istanziazione (detto Factory) incapsula la conoscenza su quale classe creare.

Struttura

“Product” è l’interfaccia comune degli oggetti che il `FactoryMethod()` crea.

“ConcreteProduct” è una implementazione dell’interfaccia “Product”. “Creator” è l’interfaccia che dichiara `FactoryMethod()`; tale metodo ritorna un oggetto di tipo “Product”.

“Concrete Creator” implementa il `FactoryMethod()` scegliendo quale “ConcreteProduct” istanziare e ritorna tale istanza.



Il FactoryMethod() ha un parametro in ingresso per far scegliere al client la classe da creare; usa Class.forName() e newInstance() per togliere dal codice la dipendenza da una specifica classe.

Conseguenze

Il codice delle classi dell'applicazione conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct. È necessario creare una sottoclasse di Creator solo per creare un ConcreteProduct (proliferazione di classi).

Codice

```
interface Shape { //Product
    void draw();
    void fill();
}
interface ShapeCreator { //Creator
    public Shape getShape(); //Factory Method
}
//ConcreteCreator
class CreatorA implements ShapeCreator {
    public Shape getShape() { //override
        //solo qui indica la classe da istanziare
        return new Circle();
    }
}
class Circle implements Shape { //ConcretProd
    public void draw() {
        System.out.println("A circle ( )");
    }
    public void fill() {
        System.out.println("Filled circle (o)");
    }
}

class Square implements Shape { //ConcretProd
    public void draw() {
        System.out.println("A Square [ ]");
    }
    public void fill() {
        System.out.println("Filled Square [X]");
    }
}
public class ShapeCreatorTest {
    public static void main(String args[]) {
        //istanzio il Concrete Creator
        ShapeCreator sc = new CreatorA();
        //ottengo una istanza di una
        //sottoclasse di Shape, non
        //decido qui quale sottoclasse
        Shape s = sc.getShape();

        s.draw();
        s.fill();
    }
}
```

Object Pool

Qualche volta è utile poter riusare le istanze già create, anziché crearne nuove, ovvero una nuova per ciascuna richiesta. Poiché in alcuni casi, istanziare ed inizializzare sono operazioni lente, il design pattern Factory Method può implementare un **Object pool**, ovvero un repository che gestisce le istanze già create; una istanza sarà estratta dal pool quando un client ne fa richiesta.

Il pool può crescere o può avere dimensioni fisse, ovvero se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi. Il client restituisce l'istanza usata al pool quando non più utile.

Il Factory Method è ottimo per gestire un pool di oggetti poiché i client fanno richieste, come visto prima per il Factory Method, e dovranno dire quando l'istanza non è più in uso, quindi riusabile; inoltre lo stato dell'istanza da riusare potrebbe dover essere ri-scritto. L'object pool dovrebbe essere unico, allora uso un Singleton.

```
import java.util.LinkedList;
// CreatorPool is a ConcreteCreator and implements an object pool
public class CreatorPool extends ShapeCreator {
    private LinkedList<Shape> pool = new LinkedList<Shape>();
    public Shape getShape() {
        Shape s;
        if (pool.size() > 0) s = pool.remove();
        else s = new Circle();
        return s;
    }
    public void releaseShape(Shape s) {
        pool.add(s);
    }
}
```

Dependency Injection

Il design pattern Factory Method può essere usato per inserire le dipendenze necessarie ad altri oggetti. Una classe che usa un servizio *dipende* dal servizio; come faccio conoscere alla classe il servizio da cui dipende? Uso la classe **Dependency injection**.

Esempio. Una classe TextEditor usa un servizio SpellingCheck, quindi TextEditor dipende dal servizio SpellingCheck. Ci sono tante classi che implementano il servizio SpellingCheck, in base alla lingua usata: SpellingCheckEnglish, SpellingCheckItalian, etc. TextEditor deve poter essere collegato ad una delle classi SpellingCheck.

```
public class TextEditor {
    private SpellingCheck speller;
    public TextEditor(SpellingCheck sp) {
        speller = sp;
    }
    public void put(String s) {
        // receive input
        if (speller.check(s)) ...
        else ...
    }
}

public class CreatorText {
    public static TextEditor getEnglishEditor() {
        return new TextEditor(new SpellingCheckEnglish());
    }
    public static TextEditor getItalianEditor() {
        return new TextEditor(new SpellingCheckItalian());
    }
}
```

, attraverso la chiamata al costruttore, con
SpellingCheck da usare ed inserisce (inject) la dipendenza
a l'uso di una classe SpellingCheck
nente.

Singleton

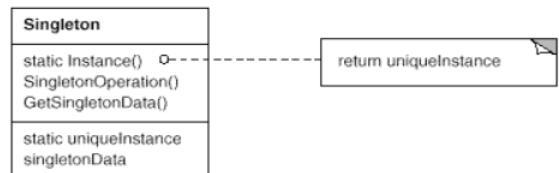
Il Singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

Scopo	Motivazione
<ul style="list-style-type: none"> - Assicurare che una classe abbia una sola istanza - Fornire un punto d'accesso globale all'istanza 	<ul style="list-style-type: none"> - Alcune classi dovrebbero avere esattamente una istanza (uno spooler di stampa, un file system, un window manager); - Una variabile globale rende un oggetto accessibile ma non proibisce di istanziare più oggetti; - La classe dovrebbe essere responsabile di tener traccia del suo unico punto di accesso.

Struttura

L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo "getter" statico che restituisce una istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.

Singleton definisce una operazione `instance()` per la classe (in Java è un metodo static) che ritorna l'unica istanza creata. Singleton è responsabile per la creazione dell'istanza, il suo costruttore è privato, quindi la creazione con `new` è inaccessibile ad altre classi.



Conseguenze

La classe ha pieno controllo di come e quando i client accedono, evita che esistano variabili globali che tengono la sola istanza condivisa e permette di controllare il numero di istanze create in un programma, facilmente ed in un solo punto.

La soluzione è più flessibile rispetto a quella di usare static per tutte le operazioni e le variabili poiché si può cambiare facilmente il numero di istanze consentite.

L'unico frammento di codice da variare quando si vuol variare il numero di istanze create è quello della classe Singleton, mentre usando static si dovrebbero variare tutte le invocazioni

Codice

<pre>// Singleton design pattern che contiene // un valore intero class Singleton { // l'unica istanza è riferita da s private static Singleton s = new Singleton(47); private int i; private Singleton(int x) { i = x; } public static Singleton Instance() { return s; } public int getValue() { return i; } public void setValue(int x) { i = x; } }</pre>	<pre>// Uso della classe Singleton public class testSing { public static void main(String[] args) { // richiede una istanza Singleton s = Singleton.Instance(); System.out.println("s contiene: " + s.getValue()); // richiede una nuova istanza Singleton s2 = Singleton.Instance(); s2.setValue(9); int v = s.getValue(); // quanto vale v? System.out.println("s contiene: "+v); // Si ha un errore a compile-time // Singleton s3 = (Singleton) s2.clone(); // Singleton s4 = new Singleton(33); } }</pre>
--	--

Adapter

Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il problema si presenta ogni qual volta nel progetto di un software si debbano utilizzare sistemi di supporto (come per esempio *library*) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti. Invece di dover riscrivere parte del sistema, compito oneroso e non sempre possibile se non si ha a disposizione il codice sorgente, può essere comodo scrivere un adapter che faccia da tramite.

L'uso del pattern Adapter risulta utile quando interfacce di classi differenti devono comunque poter comunicare tra loro. Alcuni casi possono includere

- l'utilizzo di una classe esistente che presenti un'interfaccia diversa da quella desiderata;
- la scrittura di una determinata classe senza poter conoscere a priori le altre classi con cui dovrà operare, in particolare senza poter conoscere quale specifica interfaccia sia necessario che la classe debba presentare alle altre.

Un altro contesto è quello in cui si desidera che l'invocazione di un metodo di un oggetto da parte dei client avvenga solo in maniera indiretta: il metodo "target" viene encapsulato all'interno dell'oggetto, mentre uno o più metodi "pubblici" fanno da tramite con l'esterno. Questo consente alla classe di subire modifiche future mantenendo la retrocompatibilità, oppure di implementare in un unico punto una funzionalità alla quale i client accedono tramite metodi più "comodi" da usare e con signatures differenti.

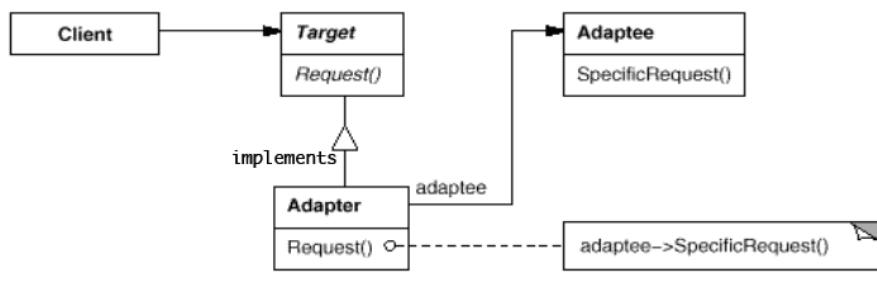
Scopo	Motivazione
<ul style="list-style-type: none"> - Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano - L'adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili. 	<ul style="list-style-type: none"> - Certe volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia richiesta dall'applicazione (nome metodo, parametri, tipo parametri non corrispondenti). - Non è possibile cambiare l'interfaccia della libreria poiché non si ha il sorgente (comunque non conviene cambiarla). - Non è possibile cambiare l'applicazione, si può voler cambiare quale metodo invocare, senza renderlo noto al client.

Struttura

Bisogna creare una classe Adapter che converte, ovvero adatta, l'interfaccia che il client si aspetta (Target) all'interfaccia della classe di libreria. Il client usa l'Adapter come se fosse

l'oggetto di libreria. L'Adapter possiede il riferimento all'oggetto di libreria (detto Adaptee) e sa come invocarlo.

Il pattern Adapter può essere basato su classi, utilizzando l'ereditarietà multipla per adattare interfacce diverse con il meccanismo dell'ereditarietà, oppure sulla composizione di oggetti.



I partecipanti sono:

Adaptee: definisce l'interfaccia che ha bisogno di essere adattata.
Target: definisce l'interfaccia che usa il Client.
Client: collabora con gli oggetti in conformità con l'interfaccia Target.
Adapter: adatta l'interfaccia Adaptee all'interfaccia Target.

Codice

```

public interface ILabel { // Target
    public String getNextLabel();
}

public class LabelServer { // Adaptee
    private int nextLabelNum = 1;
    private String labelPrefix;
    public LabelServer(String prefix) {
        labelPrefix = prefix;
    }
    public String serveNextLabel() {
        return labelPrefix+nextLabelNum++;
    }
}
    
```

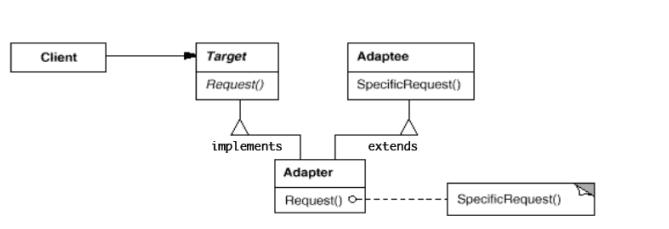
```

// Adapter
public class Label implements ILabel {
    private LabelServer theService;
    public Label(String prefix) {
        theService = new LabelServer(prefix);
    }
    public String getNextLabel() {
        return theService.serveNextLabel();
    }
}

public class Client {
    static public void main(String args[]) {
        ILabel s = new Label("LAB");
        String l = s.getNextLabel();
        if (l.equals("LAB1"))
            System.out.println("Test 1: Passed");
        else System.out.println("Test 1: Failed");
    }
}
    
```

Struttura alternativa: Class Adapter

Un'altra soluzione è la Class Adapter, la quale fa uso dell'ereditarietà multipla.



```

// Adapter
public class Label extends LabelServer implements ILabel {
    public Label(String prefix) {
        super(prefix);
    }
    public String getNextLabel() {
        return serveNextLabel();
    }
}
  
```

La classe Adapter include l'interfaccia di Adaptee, mentre la versione Class Adapter fornisce l'Adapter a due vie. Le conseguenze sono che il client e la classe di libreria rimangono indipendenti. L'Adapter può cambiare il comportamento dell'Adaptee aggiungendo codice, per es. test di precondizioni e postcondizioni, e permette di implementare la tecnica di Lazy Initialization

L'Adapter aggiunge un livello di indirezione dove ogni invocazione del client ne scatena un'altra fatta dall'Adapter, con un possibile overhead e codice più difficile da comprendere.

Bridge

Il bridge pattern permette di separare l'interfaccia di una classe (che cosa si può fare con la classe) dalla sua implementazione (come si fa). In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato.

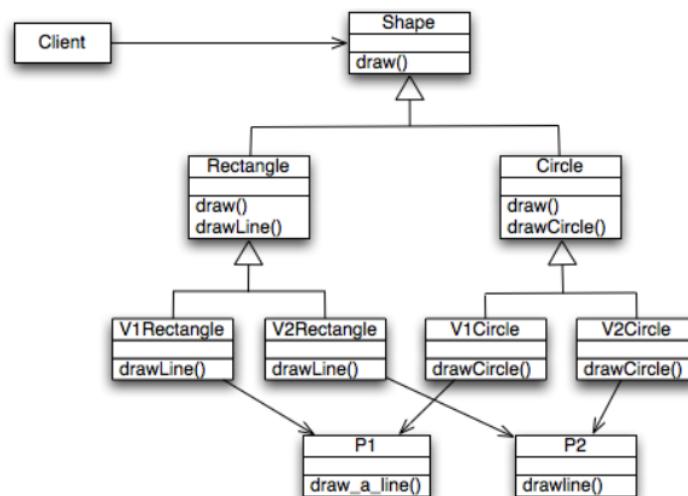
Scopo	Motivazione
- Disaccoppiare una <i>astrazione</i> dalla sua <i>implementazione</i> così che le due possano variare indipendentemente	Quando una astrazione può avere varie implementazioni, di solito si usa l'ereditarietà - Una classe astratta definisce l'interfaccia dell'astrazione, le sottoclassi concrete la implementano in modi diversi - Questo approccio non è flessibile poiché collega l'astrazione all'implementazione permanentemente

Problema

Consideriamo il seguente esempio: In un sistema, occorre avere Rettangoli e Cerchi (astrazioni). Occorre che Rettangoli e Cerchi siano disponibili per due tipi di piattaforme P1 e P2 (implementazioni)

- Per P1 devo usare draw_a_line() e draw_a_circle()
- Per P2 devo usare drawline() e drawcircle()

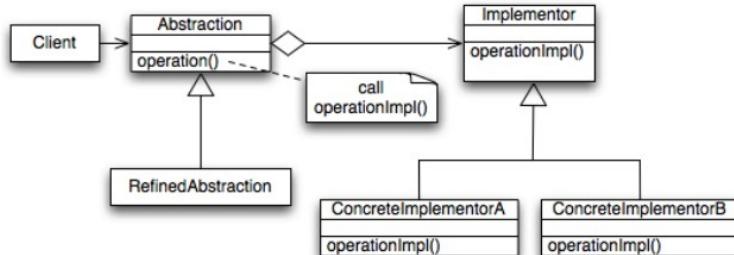
La struttura delle classi iniziali prima di usare bridge è questa



La soluzione appena vista porterebbe ad una proliferazione di classi; se introducessi un'altra piattaforma (implementazione) avrei bisogno di 6 classi concrete (2 shape x 3 piattaforme), e se introducessi un'altra shape (astrazione), avrei bisogno di 9 classi concrete (3 x 3).

Per ogni variazione da introdurre vorrei invece un incremento lineare del numero di classi. Inoltre, la classe V1Rectangle è legata alla piattaforma P1 in modo permanente, una istanza di V1Rectangle non può usare una piattaforma diversa da P1.

Soluzione tramite Bridge



Abstraction definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto di tipo **Implementor**.

RefinedAbstraction estende l'interfaccia definita da **Abstraction**.

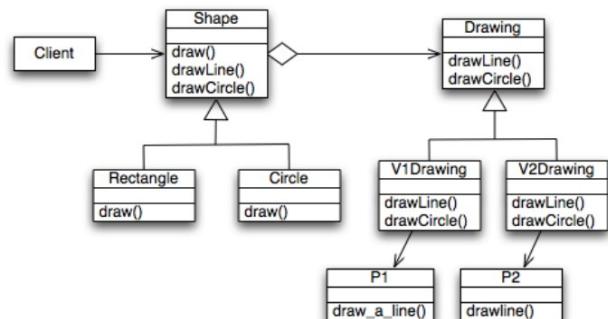
Implementor definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere ad

Abstraction, di solito **Implementor** fornisce operazioni primitive, mentre **Abstraction** definisce operazioni di più alto livello.

ConcreteImplementor implementa l'interfaccia di **Implementor** e fornisce le operazioni concrete.

Secondo la soluzione indicata dal design pattern Bridge, per il suddetto esempio si avrà il diagramma delle classi disegnato a destra.

`Rectangle.draw()` invoca `drawLine()` della superclasse **Shape**, quest'ultima invoca `drawLine()` dell'istanza di una sottoclasse di **Drawing**, analogamente per **Circle**. Una nuova classe, **Rombo**, sarà implementata come sottoclasse di **Shape**. La classe **Rombo** invocherà `drawLine()` presente in **Shape**.



Conseguenze

- ✓ Bridge permette ad una implementazione di non essere connessa permanentemente ad una interfaccia, l'implementazione può essere configurata ed anche cambiata a runtime.
- ✓ Il disaccoppiamento permette di cambiare l'implementazione senza dover ricompilare **Abstraction** ed i **Client**
- ✓ Solo certi strati del software devono conoscere **Abstraction** e **Implementor**
- ✓ I **Client** non devono conoscere **Implementor**
- ✓ Le gerarchie di **Abstraction** e **Implementor** possono evolvere in modo indipendente

Composite

Questo pattern permette di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. Il design pattern Composite organizza gli oggetti in una struttura ad albero, nella quale i nodi sono delle composite e le foglie sono oggetti semplici.

È utilizzato per dare la possibilità ai client di manipolare oggetti singoli e composizioni in modo uniforme.

Esempio: Le operazioni su disco permettono la gestione di file (immagini, testo, etc.) e la gestione di cartelle. I client possono voler rappresentare file e cartelle, senza distinguere tra questi. Una cartella deve, quindi, poter contenere al suo interno sia file che altre cartelle, queste a sua volta contengono altri elementi, e così via

Scopo	Motivazione
- Comporre oggetti in strutture ad albero per	- In molti casi è necessario raggruppare elementi semplici tra loro per formare elementi più grandi.

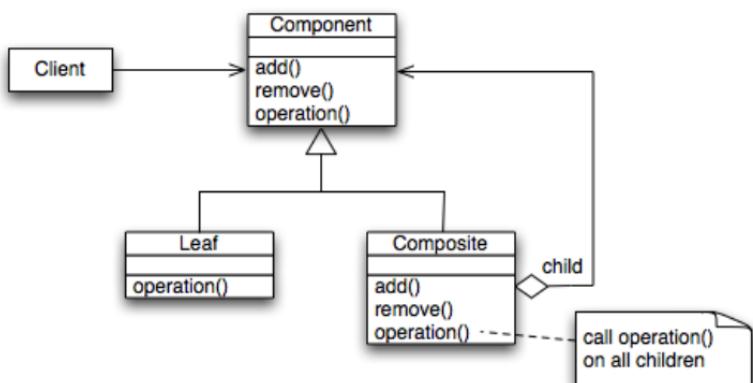
<p>rappresentare gerarchie di parti o del tutto. Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente.</p>	<ul style="list-style-type: none"> - Se nell'implementazione c'è distinzione tra classi per elementi semplici e classi per contenitori di questi elementi semplici, il codice che usa queste classi deve trattarli in modo differente; questa distinzione rende il codice più complicato. - Il design pattern Composite permette di descrivere una composizione ricorsiva, in modo che i client non debbano fare distinzione tra tipi di elementi. I client tratteranno tutti gli oggetti della struttura uniformemente.
--	--

Struttura

L'elemento chiave del design pattern Composite è la classe abstract Component che rappresenta elementi semplici e contenitori. La classe Component dichiara le operazioni degli oggetti da comporre, implementa le operazioni comuni alle sottoclassi, in modo appropriato, per gli elementi contenitori dichiara le operazioni per l'accesso e la gestione degli elementi semplici e può definire una operazione per permettere agli elementi di accedere all'oggetto che è il loro padre nella struttura ricorsiva.

La classe **Leaf** rappresenta elementi semplici (child), è sottoclasse di Component ed implementa il comportamento degli oggetti semplici. La classe Composite rappresenta elementi contenitori, definisce il comportamento per l'aggregato di componenti child, immagazzina il riferimento ai componenti child ed implementa operazioni per gestire i componenti child.

Client: manipola gli oggetti attraverso l'interfaccia Component.



oggetti primitivi, cioè delle foglie.

Scopo

Quando i programmatore trattano dati strutturati ad albero devono spesso discriminare se stanno visitando un nodo o una foglia. Questa differenza è una possibile fonte di complessità per il codice e, se non trattata a dovere, rende il programma facilmente soggetto a errori. La soluzione è adottare un'interfaccia che permetta di trattare oggetti complessi e primitivi in modo uniforme. Nella programmazione orientata agli oggetti un Composite è un oggetto (per esempio una figura geometrica) progettato per composizione di uno o più oggetti simili (sempre per rimanere sullo stesso esempio: una figura geometrica come un trapezio può essere vista a sua volta come formata da due triangoli rettangoli e un rettangolo) che offrono tutti le medesime funzionalità. Questa caratteristica è conosciuta anche come relazione "has-a" tra gli oggetti. Il concetto fondamentale è che il programmatore manipola ogni oggetto dell'insieme dato nello stesso modo: sia esso un "raggruppamento" o un oggetto singolo. Le operazioni che possono essere effettuate sul Composite hanno spesso un denominatore comune, per esempio: se il programmatore deve visualizzare a schermo un insieme di figure potrebbe essergli utile definirne il ridimensionamento in modo tale da avere lo stesso effetto (in senso lato) del ridimensionamento di una singola figura.

Quando usarlo

Il Composite può essere usato quando i client dovrebbero ignorare la differenza tra oggetti composti e oggetti singoli. Se durante lo sviluppo i programmatori scoprono che stanno usando più oggetti nello stesso modo, e spesso il codice per gestirli è molto simile, il Composite rappresenta una buona scelta di rifactorizzazione: in questa situazione, è meno complesso trattare oggetti primitivi e composti in modo omogeneo.

Component: dichiara l'interfaccia per gli oggetti nella composizione, per l'accesso e la manipolazione di questi, imposta un comportamento di default per l'interfaccia comune a tutte le classi e può definire un'interfaccia per l'accesso al padre del componente e la implementa se è appropriata.

Composite: definisce il comportamento per i componenti aventi figli, salva i figli e implementa le operazioni ad essi connesse nell'interfaccia Component.

Leaf: definisce il comportamento degli

Funzionamento

Attraverso l'interfaccia Component, il Client interagisce con gli oggetti della composite. Se l'oggetto desiderato è una Leaf, la richiesta è processata direttamente; altrimenti, se è una Composite, viene rimandata ai figli cercando di svolgere le operazioni prima e dopo del rimando.

In questo modo, si semplifica il Client, si creano delle gerarchie di classi, si semplifica l'aggiunta di nuovi componenti, anche se il design diventa troppo generale.

Collaborazioni

I client usano l'interfaccia di Component per interagire con gli oggetti della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo rimanda la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo.

Conseguenze

- Oggetti elementari possono essere composti in oggetti più complessi, questi possono essere composti, e così via;
- Un client che si aspetta un oggetto elementare può prendere anche un oggetto composto;
- I clienti sono semplici, possono trattare strutture composte ed oggetti semplici uniformemente. I client non sanno se trattano con un oggetto Leaf o un Composite;
- Nuovi tipi di componenti (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti;
- Non è possibile a design time vincolare il Composite solo su certi componentiLeaf, dovranno essere invece fatti controlli a runtime.

Dettagli

Per facilitare la navigazione della struttura composta i componenti child possono mantenere il riferimento all'oggetto che li contiene. Il riferimento può essere inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo.

Per avere client che non distinguono se stiano trattando Leaf o Composite (è uno degli obiettivi), la classe Component dovrebbe definire quante più operazioni in comune possibile. Le classi Leaf e Composite faranno override delle operazioni.

Dove dichiarare le operazioni di gestione dei child, add() e remove()?

- o Se dichiarate in Component si ha trasparenza, ma per le classi Leaf tali operazioni non hanno significato. La sicurezza nell'uso è quindi compromessa, poiché i client potrebbero avviare su Leaf. Se i client avviano una operazione add() su Leaf e si ignora la richiesta, questo potrebbe indicare un bug
- o Se dichiarate in Composite si ha sicurezza, poiché a compile time si verifica che tali operazioni non possono essere chiamate su Leaf, ma si perde la trasparenza.

Si può inserire una operazione getComposite() in Component che ritorna null e che è ridefinita in Composite per ritornare il riferimento a se stesso. I client dovrebbero comunque distinguere il tipo di risultato e fare operazioni differenti, niente trasparenza.

La lista che contiene i componenti child dovrebbe essere definita in Composite, altrimenti se definita in Component si spreca spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai.

L'ordinamento dei child per un composite potrebbe essere importante e va tenuto in considerazione su certe implementazioni.

Il Composite potrebbe implementare una cache, per ottimizzare le prestazioni, quando gli si richiede un valore che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad una operazione che permette di invalidare la cache del Composite.

Codice

```
// Resource plays role Component
public abstract class Resource {
    public abstract void show();
    //add, remove, not to be called on Leaf
    public void add(Resource c) { }
    public void remove(Resource c) { }
}
```

```
// Image plays role Leaf
public class Image extends Resource {
    private int x, y;
    private String name;
    public Image(String id, int a, int b) {
        name = id; x = a; y = b;
    }
    @Override public void show() {
        // some code here
    }
}
```

```
// Creator provides some Factory methods
public class Creator {
    public static Resource getFolder() {
        return new Folder("myBooks");
    }
    public static Resource getImage() {
        return new Image("car.jpg", 80, 60);
    }
}
```

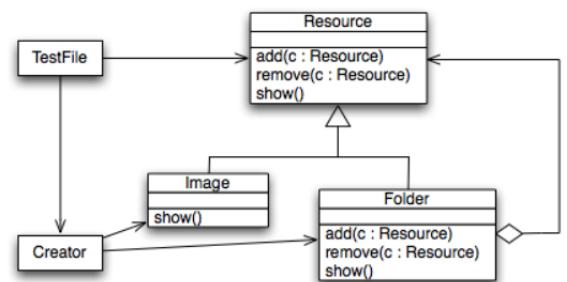
```
// Folder plays role Composite
import java.util.LinkedList;
public class Folder extends Resource {
    private String name;
    private LinkedList<Resource> r =
        new LinkedList<Resource>();
    public Folder(String f) {
        name = f;
    }
    @Override public void show() {
        for (int i = 0; i <r.size(); i++)
            r.get(i).show();
    }
    @Override public void add(Resource c) {
        r.add(c);
    }
}
```

```
// TestFile uses Composite design pattern
public class TestFile {
    public static void main(String[] args) {
        // create resources, hold Components
        Resource fold = Creator.getFolder();
        Resource img = Creator.getImage();

        // invoke show on a Leaf
        img.show();

        // insert img into Composite fold
        fold.add(img);

        // invoke show on a Composite
        fold.show();
    }
}
```



Decorator

Il design pattern decorator consente di aggiungere durante il run-time nuove funzionalità ad oggetti già esistenti. Questo viene realizzato costruendo una nuova classe decoratore che **"avvolge"** l'oggetto originale. Al costruttore del decoratore si passa come parametro l'oggetto originale. È altresì possibile passarvi un differente decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall'ultimo anello della catena).

La concatenazione dei decoratori può avvenire secondo una composizione arbitraria: il numero di comportamenti possibili dell'oggetto composto varia dunque con legge combinatoriale rispetto al numero dei decoratori disponibili.

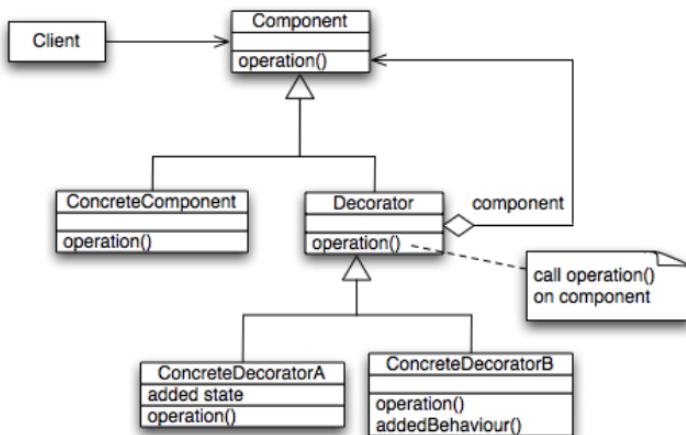
Questo pattern si pone come valida alternativa all'uso dell'ereditarietà singola o multipla. Con l'ereditarietà, infatti, l'aggiunta di funzionalità avviene staticamente secondo i legami definiti nella gerarchia di classi e non è possibile ottenere al run-time una combinazione arbitraria delle funzionalità, né la loro aggiunta/rimozione.

Scopo	Motivazione
<ul style="list-style-type: none"> - Attaccare responsabilità addizionali ad un oggetto dinamicamente. - Fornire un'alternativa flessibile alla creazione di sottoclassi per estendere funzionalità 	<ul style="list-style-type: none"> - A volte si vuole aggiungere una responsabilità ad un singolo oggetto, non all'intera classe <ul style="list-style-type: none"> - Le responsabilità possono essere sottratte dinamicamente Es. Per un componente grafico si vorrebbe poter aggiungere la proprietà bordo, se si eredita da una classe Bordo, tutti gli oggetti della sottoclasse avranno questa proprietà, quindi niente flessibilità Alternativa flessibile, inserire il componente grafico dentro un oggetto che aggiunge il bordo. L'oggetto che racchiude, chiamato Decorator, manda la richiesta al componente e aggiunge altre attività prima o dopo l'invio della richiesta - I Decorator possono essere annidati ricorsivamente, per aggiungere più di una responsabilità - A volte la creazione di sottoclassi non è praticabile. Un numero grande di estensioni produrrebbe un numero enorme di sottoclassi per gestire tutte le combinazioni.

Struttura

Il Decorator manda le richieste al Component che può svolgere le operazioni precedenti e successive alla spedizione della richiesta. In questo modo si ottiene una maggior flessibilità,

tanti piccoli oggetti al posto di uno molto complicato, andando a modificare il contorno e non la sostanza di una classe.



Component: definisce l'interfaccia dell'oggetto a cui verranno aggiunte nuove funzionalità.

ConcreteComponent: definisce l'oggetto concreto al quale aggiungere le funzionalità.

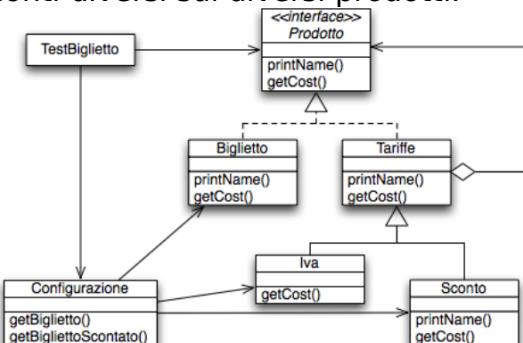
Decorator: mantiene un riferimento all'oggetto Component e definisce un'interfaccia conforme all'interfaccia Component. Oltre ciò, inoltre le richieste al suo oggetto Component e può effettuare altre operazioni prima e dopo l'inoltro della richiesta.

ConcreteDecorator: aggiunge le

funzionalità al component.

Esempio con codice

Ho alcuni prodotti come ad es. Biglietto, ognuno con un nome ed un costo; ho diversi modi di calcolare il costo di questi prodotti, in base a come applico sconti e percentuali IVA. Voglio poter combinare a runtime sconti diversi sui diversi prodotti.



```

// Biglietto e' un ConcreteComponent
public class Biglietto implements Prodotto {
    public void printName() {
        System.out.print("Biglietto n. 1231231");
    }
    public double getCost() {
        return 100.0;
    }
}

// Tariffe e' un Decorator,
// contiene l'istanza dell'oggetto annidato ed
// invoca operazioni sull'oggetto annidato
public abstract class Tariffe implements Prodotto {
    private Prodotto myComp;
    public Tariffe(Prodotto myC) {
        myComp = myC;
    }
    public void printName() {
        myComp.printName();
    }
    public double getCost() {
        return myComp.getCost();
    }
}

```

```

// Prodotto e' un Component
public interface Prodotto {
    public void printName();
    public double getCost();
}

// Sconto e' un ConcreteDecorator,
// aggiunge funzionalita' ad un prodotto
public class Sconto extends Tariffe {
    public Sconto(Prodotto myC) {
        super(myC);
    }
    public void printName() {
        //non chiama l'oggetto annidato
        System.out.print("Biglietto ridotto");
    }
    public double getCost() {
        //chiama getCost sull'oggetto annidato
        return super.getCost()*0.95;
    }
}

```

```

// Configurazione definisce alcuni metodi factory
public class Configurazione {
    public Prodotto getBiglietto() {
        // restituisce un tipo Prodotto
        return new Biglietto();
    }
    public static Prodotto getBigliettoScontato() {
        // qui due annidamenti
        return new Sconto(new Biglietto());
    }
}

// TestBiglietto e' un client
public class TestBiglietto {
    public static void main(String[] args) {
        Prodotto prod;
        prod = Configurazione.getBigliettoScontato();
        prod.printName();
        System.out.println(" costo: " + prod.getCost());
        prod = Configurazione.getBiglietto();
        prod.printName();
        System.out.println(" costo: " + prod.getCost());
    }
}

```

```

// Iva e' un ConcreteDecorator
public class Iva extends Tariffe {
    public Iva(Prodotto myC) {
        super(myC);
    }
    public double getCost() {
        return super.getCost()*1.2;
    }
}

```

Façade

Letteralmente façade significa "facciata", ed infatti nella programmazione ad oggetti indica un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi

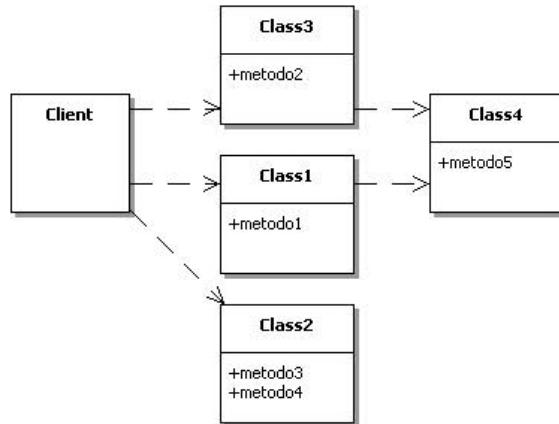
Scopo	Problema
<ul style="list-style-type: none"> - Fornire una interfaccia unificata ad un set di interfacce in un sottosistema (consistente di un gruppo di classi) - Definire una interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare 	<ul style="list-style-type: none"> - Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complessa. - Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi - Si vogliono ridurre le comunicazioni e le

dipendenze dirette tra i client ed il sottosistema.

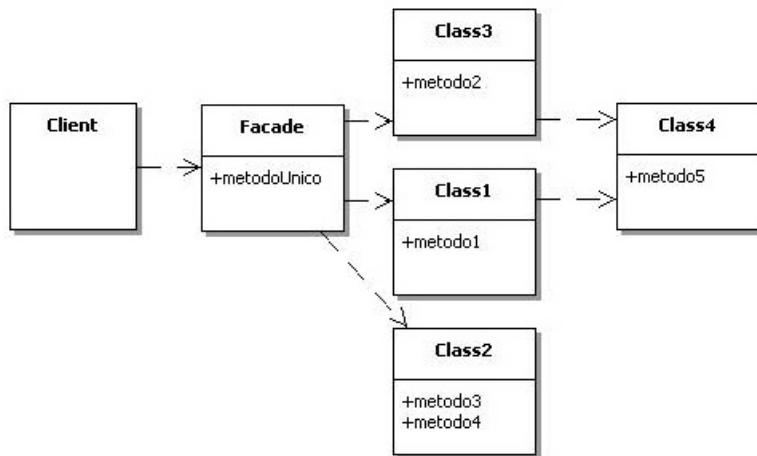
Struttura

Un modo per ridurre la complessità è introdurre un oggetto Facade che fornisce un'unica interfaccia semplificata e nasconde gli oggetti del sottosistema. Il client interagisce solo con l'oggetto Facade, il quale invoca i metodi degli oggetti che nasconde.

Consideriamo, ad esempio, la seguente situazione in cui una classe Client, per realizzare una singola operazione deve accedere ad alcune classi molto differenti tra loro.



L'utilizzo del pattern façade (qui realizzato attraverso la classe Facade) permette di nascondere la complessità dell'operazione, poiché in questo caso la classe Client chiama soltanto il metodo `metodoUnico` per realizzare la stessa operazione.



Il vantaggio è ancora più evidente se questo pattern viene utilizzato in una libreria software, poiché rende indipendente l'implementazione della classe Client dall'implementazione dei vari oggetti Class1, Class2, etc.

Codice

Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe Facade.

```

public class English {
    ...
    private String text = " ";
    private String[] EngDict = {"Alright",
        "Hello", "Understood", "Yes"};
    public int getPos(String s) { ... }
    public boolean test(String s) { ... }
    public void add(String s) {
        text = text + " " + s;
    }
    public String getText() {return text;}
    public void printText() {
        System.out.println(text);
    }
}

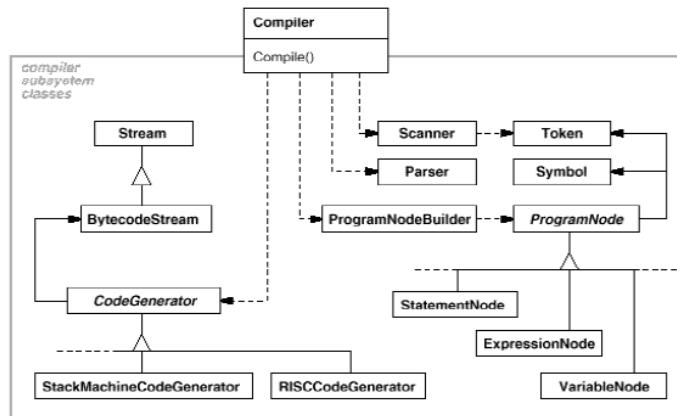
public class TestFacade {
    static public void main(String args[]) {
        Translator t = new Translator();
        t.addEnglish("Hello");
        t.multiPrinting();
    }
}

```

```

// Facade
public class Translator {
    private English eng = English.getInstance();
    private Italian it = Italian.getInstance();
    public void addEnglish(String s) {
        String s2 = null;
        if (eng.test(s)) {
            eng.add(s);
            s2 = it.intoItalian(s);
            it.add(s2);
        }
    }
    public void multiPrinting() {
        System.out.print("Italiano: ");
        it.printText();
        System.out.print("English: ");
        eng.printText();
    }
}

```



Conseguenze

- ✓ Nasconde ai client l'implementazione del sottosistema
- ✓ Promuove accoppiamento debole tra sottosistema e client
- ✓ Riduce dipendenze di compilazione in sistemi grandi
- ✓ Non previene l'uso di client più complessi, quando occorre, che accedono oggetti del sottosistema

Chain of Responsibility

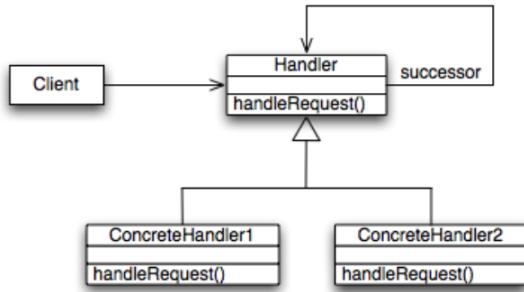
Il Chain-of-responsability è un design pattern utilizzato nella programmazione ad oggetti che permette di separare gli oggetti che invocano richieste, dagli oggetti che le gestiscono dando ad ognuno la possibilità di gestire queste richieste.

Viene utilizzato il termine catena perché di fatto la richiesta viene inviata e "segue la catena" di oggetti, passando da uno all'altro, finché non trova quello che la gestisce.

Il pattern è comodo quando non conosciamo a priori quale oggetto è in grado di gestire una determinata richiesta, sia perché effettivamente è sconosciuto staticamente o sia perché l'insieme degli oggetti in grado di gestire richieste cambia dinamicamente a runtime.

Scopo	Motivazione
- Evitare di accoppiare il mandante di una richiesta con il ricevente, dando così la possibilità a più di un oggetto di gestire la richiesta. Concatena gli oggetti riceventi e passa la richiesta tra questi fino a che un oggetto la gestisce.	<ul style="list-style-type: none"> - Consideriamo una interfaccia utente dove l'utente può richiedere aiuto su parti dell'interfaccia. Ad es. un bottone può fornire informazioni di aiuto. Se non esiste una informazione specifica allora il sistema dovrebbe fornire il messaggio d'aiuto del contesto più vicino (ad es. la finestra) - Il problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto (es. Button) che inizia la richiesta - Disaccoppiare mandante e ricevente

Struttura



Handler definisce l'interfaccia per gestire le richieste, rappresenta l'interfaccia che offre il metodo HandleRequest che sarà il metodo utilizzato dalle componenti per inoltrare richieste all'oggetto contenuto;

ConcreteHandler gestisce la richiesta per cui è responsabile, può accedere al suo successore, inoltra la richiesta se non può gestirla, e rappresenta l'effettiva implementazione della gestione degli eventi per un oggetto;

Client inizia la richiesta ad un oggetto ConcreteHandler.

Ogni oggetto facente parte della catena deve implementare il metodo HandleRequest che gestirà il tipo di richiesta ricevuta (se è lui che se ne deve occupare), altrimenti chiamerà lo stesso metodo sull'oggetto contenuto all'interno (ed è per questo che si viene a formare una catena, allo stesso modo del pattern decorator). La richiesta si propaga lungo la catena finché un oggetto ConcreteHandler la può gestire

Conseguenze

- ✓ Riduce l'accoppiamento: chi fa una richiesta non conosce il ricevente e viceversa
- ✓ Un oggetto della catena non deve conoscere la struttura della catena
- ✓ Gli oggetti handler anziché mantenere i riferimenti a tutti i candidati riceventi mantengono solo il riferimento al successore
- ✓ Si aggiunge flessibilità nella distribuzione di responsabilità agli oggetti. Si può cambiare o aggiungere responsabilità nella gestione di una richiesta cambiando la catena a runtime
- ✓ Non c'è garanzia che una richiesta venga gestita, poiché non c'è un ricevente esplicito, la richiesta potrebbe arrivare alla fine della catena senza essere gestita

Implementazione

Handler o ConcreteHandler possono definire i link per avere il successore della catena. Una gerarchia già esistente può essere usata, anziché ridefinire i link, se la gerarchia riflette la catena, altrimenti i link vanno definiti

Ciascun tipo di richiesta può corrispondere ad una operazione. In questo caso il set di richieste è definito dall'Handler; in alternativa, a ciascun tipo di richiesta corrisponde un codice e solo una operazione è definita. Il set di richieste non è fissato, richiedente e riceventi prendono accordi sulla codifica delle richieste. Abbiamo più controlli a runtime.

Mediator

Il mediator pattern è un design pattern utilizzato in informatica nella programmazione orientata agli oggetti che incapsula le modalità con cui oggetti diversi interagiscono fra loro.

Si tratta di un pattern comportamentale, ossia operante nel contesto delle interazioni tra oggetti, che ha l'intento di disaccoppiare entità del sistema che devono comunicare fra loro. Il pattern infatti fa in modo che queste entità non si riferiscano reciprocamente, agendo da "mediatore" fra le parti.

Il beneficio principale è il permettere di modificare agilmente le politiche di interazione, poiché le entità coinvolte devono fare riferimento al loro interno solamente al mediatore.

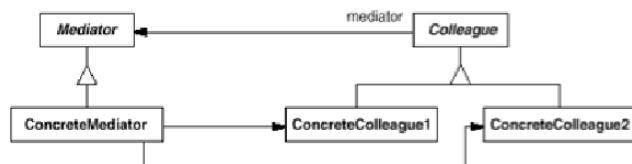
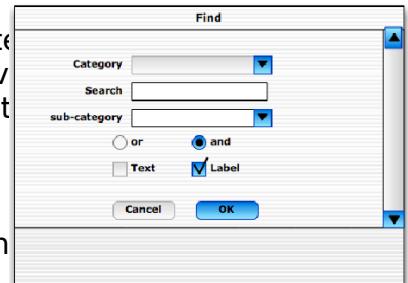
Scopo	Motivazione
<ul style="list-style-type: none"> - Definisce un oggetto che incapsula come un gruppo di oggetti che interagiscono - Promuove l'accoppiamento tra oggetti poiché essi non interagiscono direttamente 	<ul style="list-style-type: none"> - La distribuzione delle responsabilità tra gli oggetti può risultare in molte connessioni tra oggetti - Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si

comporta come se fosse monolitico
 - Diminuire le dipendenze di una classe e renderla più generale

- Esempio: Si consideri la finestra a destra. Ogni elemento visualizzato (text, button, listbox) è controllato da una corrispondente classe; ciascuna classe deve invocare i metodi delle altre per far aggiornare la visualizzazione.
- Senza Mediator: bisogna invocare i metodi di tutte le altre classi.

Struttura

Si isolano le comunicazioni (complesse) tra oggetti dipendenti creando un'interfaccia comune per esse.



Mediator: funge da intermediario fra i vari colleagues, oggetti dei quali mantiene una lista interna. Definisce un'interfaccia tra oggetti che comunicano, riceve gli eventi da essi e modifica lo stato del sistema di conseguenza.

ConcreteMediator: implementa il

comportamento cooperativo e coordina i colleagues.

Colleague: ognuno conosce il Mediator e comunica inviando notifiche al mediator quando vorrebbe comunicare con gli altri colleagues, e dunque il suo stato cambia.

Conseguenze

- o La maggior parte della complessità che risulta nella gestione di dipendenze è spostata dagli oggetti cooperanti al Mediator. Questo rende gli oggetti più facili da implementare e mantenere;
- o Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con codice che gestisce le dipendenze;
- o Il codice del Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione.

Codice

```

// classe che implementa un ConcreteMediator
public class WindowFind implements Mediator {
    // istanze di vari Colleague
    private TextButton fine = new TextButton("OK");
    private TextButton conc = new TextButton("Cancel");
    private TextBox searcher = new TextBox();
    private ListBox categ = new ListBox(categs, 2);
    // metodo che attiva i controlli
    public void selectCateg(String s) {
        if (s.compareTo(categs[0]) == 0) {
            subcateg.activate();
            fine.deactivate();
            categ.show();
        }
        ...
    }
    public void init() {
        fine.deactivate();
        conc.activate();
        searcher.show();
        categ.deactivate();
    }
}

// classe che implementa un ConcreteColleague
public interface Mediator {
    // metodo che i Colleague possono invocare
    public void selectCateg(String s);
}

public class TextBox extends Colleague {
    private String content;
    private boolean active = false;
    public void activate() {
        active = true;
        show();
    }
    public void deactivate() {
        active = false;
        show();
    }
    public void show() {
        if (active) console.bold();
        else console.normal();
        console.print(x, y, nome);
    }
    public String takeInput() {
        mediator.selectCateg(
            console.takeInput());
        // TextBox avvisa ConcreteMediator
    }
}

public class Colleague {
    protected Mediator mediator;
    public Mediator getMediator() {
        return mediator;
    }
    public void setMediator(Mediator m) {
        mediator = m;
    }
}
  
```

Observer

L'Observer pattern è un design pattern utilizzato per tenere sotto controllo lo stato di diversi oggetti.

È un pattern intuitivamente utilizzato come base architetturale di molti sistemi di gestione di eventi. Molti paradigmi di programmazione legati agli eventi, utilizzati anche quando ancora non era diffusa la programmazione ad oggetti, sono riconducibili a questo pattern. È possibile individuarlo in maniera rudimentale nella programmazione di sistema Windows, o in altri framework di sviluppo che richiedono la gestione di eventi provenienti da diversi oggetti.

Sostanzialmente il pattern si basa su uno o più oggetti, chiamati osservatori o **listener**, che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto "osservato".

Oltre all'observer esiste il **ConcreteObserver** che si differenzia dal primo perché implementa direttamente le azioni da compiere in risposta ad un messaggio; riepilogando il primo è una classe astratta, il secondo no.

Uno degli aspetti fondamentali è che tutto il funzionamento dell'observer si basa su meccanismi di *callback*, implementabili in diversi modi, o tramite funzioni virtuali o tramite puntatori a funzioni passati quali argomenti nel momento della registrazione dell'observer, e spesso a questa funzione vengono passati dei parametri in fase di generazione dell'evento.

Scopo	Problema
<ul style="list-style-type: none"> - Definire una dipendenza tra un particolare oggetto (detto subject) ed altri oggetti detti osservatori, cosicché quando il subject cambia stato, gli osservatori sono notificati ed aggiornati. <p>Es. Un oggetto contiene le coordinate di un punto e la sua variazione deve essere notificata ad un oggetto che effettua una rappresentazione grafica del punto.</p>	<ul style="list-style-type: none"> - Il subject deve essere indipendente dal numero e dal tipo degli osservatori. In altre parole, il subject non fa assunzioni sugli oggetti che dipendono da esso. Oggetti accoppiati sono più facili da riusare. - Deve essere possibile aggiungere nuovi osservatori durante l'esecuzione dell'applicazione.

Struttura

Il subject rende disponibili delle operazioni che consentono ad un osservatore di dichiarare il proprio interesse per un cambiamento di stato.

Subject: classe che fornisce interfacce per registrare o rimuovere gli observer e che implementa le seguenti funzioni: *Attach()*, *Detach()* e *Notify()*.

Observer: classe che riceve le notifiche dal soggetto e fornisce una interfaccia comune a

tutti gli oggetti che necessitano di tali notifiche. Implementa la funzione *Update()*, metodo astratto che deve essere implementato dagli observer.

ConcreteSubject: classe che fornisce lo stato dell'oggetto agli observer e che si occupa di effettuare le notifiche chiamando la funzione *notify* nella classe padre (Soggetto). Eredita da Subject, è la classe il cui stato deve essere notificato, conosce solo

la classe Observer. Contiene la funzione *GetState()* che restituisce lo stato del soggetto.

ConcreteObserver: classe che mantiene un riferimento al Subject (Concreto), per ricevere lo stato quando avviene una notifica.

Conseguenze

Il Subject conosce solo la classe Observer e non ha bisogno di conoscere le classi ConcreteObserver. ConcreteSubject e ConcreteObserver non sono accoppiati quindi più facili da riusare e modificare.

Es. X Window (ma non ad oggetti), ogni finestra registra il suo interesse ad un particolare evento e tutte le informazioni sono inviate indietro (callback) quando l'evento accade.

Observer in Java

Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è parte della libreria `java.util`. In tale libreria ci sono due classi Java che permettono di implementare l'Observer: `Observable` ed `Observer`.

- La classe **Observable** tiene traccia di tutti gli oggetti che vogliono essere informati quando accade un cambiamento. Essa notifica il cambiamento di stato e permette agli osservatori di aggiornare il proprio stato, attraverso il metodo `notifyObservers()`. Ha una variabile (`flag`) che indica se lo stato è cambiato. Questo è settato dal metodo `setChanged()`. La chiamata a `setChanged()` è da implementare nella classe che la eredita, secondo la logica del programma.
- **Observer** è una interfaccia che ha solo il metodo `update()`. Questo metodo è chiamato dall'oggetto osservato. Il metodo `update()` può avere un argomento che indica quale oggetto ha causato l'aggiornamento.

Codice

```
// AddrBook e' un ConcreteSubject
public class AddrBook extends Observable {
    private static AddrBook instance = new AddrBook();
    private Vector<Person> nomi = new Vector<Person>();

    public static AddrBook getInstance() {
        return instance;
    }

    public void clear() {
        nomi.clear();
        setChanged();
        notifyObservers(nomi);
    }

    public boolean insert(Person p) {
        if (nomi.contains(p)) return false;
        nomi.addElement(p);
        setChanged();
        notifyObservers(nomi);
        return true;
    }
    ...
}

public class Dialog {
    public static AddrBook book;

    public static void main(String args[]) {
        Store st = new Store();
        book = AddrBook.getInstance();
        book.addObserver(st);
        // aggiungo st come observer di book
        ...
    }
}

// Store e' un ConcreteObserver
public class Store implements Observer {
    private FileWriter f;

    public void update(Observable s, Object c) {
        System.out.println("Writing AddrBook");
        Vector<Person> v = (Vector<Person>) c;
        ...
    }
}
```

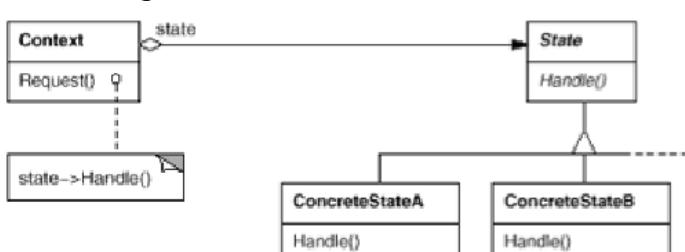
State

Nella programmazione orientata agli oggetti, lo State è un design pattern comportamentale. Esso consente ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello stato in cui si trova.

Scopo	Applicabilità
- Permette ad un oggetto di alterare il suo comportamento quando il suo stato cambia. L'oggetto appare come aver cambiato la sua classe.	- Il comportamento di un oggetto dipende dal suo stato e il comportamento cambia a run-time in dipendenza del suo stato - Le operazioni hanno condizioni che dipendono dallo stato

Struttura

Inserire ogni ramo condizionale in una classe separate.



Context: Definisce l'interfaccia del client e mantiene un'istanza di un `ConcreteState` che definisce lo stato corrente.

State: Definisce l'interfaccia, implementata dai `ConcreteState`, che incapsula la logica del comportamento associato ad un determinato stato.

ConcreteState: Implementa il comportamento associato ad un particolare stato.

Conseguenze

Tra i benefici dell'adozione di questo design pattern vi sono:

- Il comportamento associato ad uno stato dipende solo da una classe (ConcreteState)
- La logica che implementa il cambiamento di stato viene implementata in una sola classe (Context) piuttosto che con istruzioni condizionali (if o switch) nella classe che implementa il comportamento.
- Evita stati inconsistenti.

Tra le conseguenze:

- Incrementa il numero delle classi.
- Inserisce il comportamento associato ad uno stato in una sola classe (ConcreteState).
- Permette di incorporare la logica che gestisce il cambiamento di stato separatamente ed in una sola classe (Context), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti.
- Aiuta ad evitare stati inconsistenti poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante.

Codice

```
// TestOrologio e' un Client
public class TestOrologio {
    public static void main(String[] args) {
        Orario o = new Orario();
        o.intoDigital();
        o.oraAttuale();
    }
}

// Orologio definisce l'interfaccia State
public interface Orologio {
    public void displayTime();
}

// Orario ha il ruolo di Context
public class Orario {
    private Orologio o;
    public void oraAttuale() {
        o.displayTime();
    }
    public void intoDigital() {
        o = new Digitale();
    }
    public void intoAnalog() {
        o = new Analogico();
    }
}

// Digitale ha il ruolo di un ConcreteState
public class Digitale implements Orologio {
    public void displayTime() {
        Date d = Calendar.getInstance().getTime();
        System.out.print(
            DateFormat.getDateInstance().format(d));
    }
}

// Analogico ha il ruolo di un ConcreteState
public class Analogico implements Orologio {
    public void displayTime() {
        // implementa un comportamento
    }
}
```

Model View Controller (MVC)

Pattern architettonale molto diffuso nello sviluppo di sistemi software, in particolare nell'ambito della programmazione orientata agli oggetti; individua tre componenti per la gestione di applicazioni interattive:

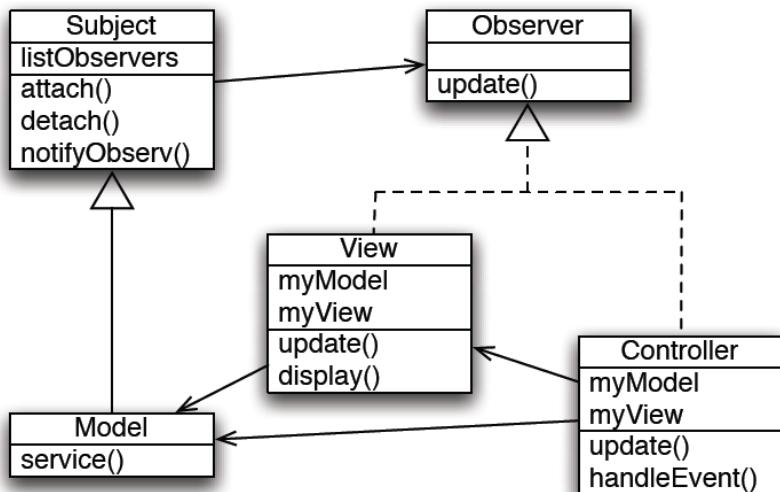
- o **Model**, per funzionalità principali e dati, fornisce i metodi per accedere ai dati utili all'applicazione;
- o **View**, per mostrare dati all'utente, visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- o **Controller**, per gestire gli input dell'utente, riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.

Motivazione	Soluzione
<ul style="list-style-type: none">- Le interfacce utente sono soggette a cambiare, poiché le funzionalità, i dispositivi o le piattaforme cambiano- Le stesse informazioni sono presentate in finestre differenti (per es. sotto forma di grafici diversi)	<ul style="list-style-type: none">- Model incapsula le funzionalità principali e i dati, è indipendente dalla rappresentazione degli output e dal comportamento degli input- View mostra i dati all'utente. Generalmente ci sono tante View. Ogni View è associata a un Controller

<ul style="list-style-type: none"> - Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati - I cambiamenti all'interfaccia utente dovrebbero essere facili - Il supporto a diverse modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali 	<ul style="list-style-type: none"> - Controller riceve gli input dell'utente (da mouse e tastiera) e traduce gli eventi in richieste di servizio per Model o per View
---	--

Struttura

La tipica interazione tra componenti del pattern MVC è:



Antipattern

Gli antipattern sono dei design pattern usati durante il processo di sviluppo del software, che pur essendo lecitamente utilizzabili, si rivelano successivamente inadatti o contro produttivi nella pratica.

Il termine fu coniato nel 1995 da *Andrew Koenig*, ispirato dal libro *Design Patterns: Elementi per il riuso di software ad oggetti* scritto dalla Gang of Four (la banda dei quattro), i quali svilupperanno il concetto di pattern nel campo del software. Secondo l'autore, devono presentarsi almeno due elementi chiave per poter distinguere un anti-pattern da un semplice errore logico o cattiva pratica:

- Qualche schema ricorrente di azioni, processi o strutture che inizialmente appaiono essere di beneficio, ma successivamente producono più problemi che benefici.
- Esiste una soluzione alternativa che è chiaramente documentata, collaudata nella pratica e ripetibile.

Scopo

Molti anti-pattern sono poco più che errori, problemi irrisolvibili o cattive pratiche da evitare quando possibile. A volte chiamati "pitfalls" (tranelli) o dark pattern (pattern oscuri), si riferiscono a classi di soluzioni di problemi reinventate in modo sbagliato.

Servono per descrivere in modo formale errori che tendono a ripetersi, individuare la forza con la quale questi si ripetono, e imparare come altre persone hanno rimediato a questi cattivi pattern.

Antipattern più comuni

Sono descritti da: nome dell'antipattern (che identifica il tipo di problema), forma (situazione che ha conseguenze negative), sintomi (come riconoscere l'antipattern), cause (cosa porta a produrre l'antipattern) e soluzione (come eliminare l'antipattern).

God class (aka Blob)

Forma	Una classe di grandi dimensioni, la God class o Controller, usa tante classi di
-------	---

	piccole dimensioni, le quali spesso contengono solo dati. Controller ha tante responsabilità, è difficile da comprendere, invoca molti metodi di altre classi e non contiene i dati su cui lavora. L'architettura del sistema che usa la God class è di stile procedurale.
Sintomi	Una classe con tanti attributi e metodi (60 o più) che classe manca di coesione (poiché ha tante responsabilità) ed è troppo complessa per essere riusata o testata.
Cause	Mancanza di una architettura ad oggetti, i progettisti non hanno adeguata comprensione dei principi OO; sono state aggiunte funzionalità al sistema senza revisionare l'allocazione di responsabilità.
Soluzione	Refactoring: distribuire le responsabilità tra varie classi; una classe dovrebbe contenere i dati che permettono ad essa di prendere delle decisioni; identificare set di dati ed operazioni coesi tra loro e creare una nuova classe per ogni set identificato; rimuovere interazioni tra classi che hanno relazioni "logiche" indirette.

Lava Flow	
Forma	In un sistema software, nato da una ricerca, ci sono parti di codice irremovibili (lava solidificata) che nessuno conosce, ed esistono frammenti di codice non chiaramente relazionati con il resto del sistema. Tali parti sono risalenti ad uno sviluppo precedente, quando gli sviluppatori, in fase di ricerca, hanno provato varie soluzioni per risolvere un problema Il codice Lava Flow è difficile da analizzare, verificare, testare, e spreca risorse di memoria e di processore.
Sintomi	Parti di codice e variabili non spiegabili, non documentate, non relazionate al resto, commentate senza spiegazioni; o codice non usato, poiché mai richiamato da altre parti (dead code).
Cause	Il codice di ricerca è diventato codice di produzione; è stato distribuito codice non completo; mancanza di architettura o architettura cambiata in fase di implementazione.
Soluzione	Assicurarsi che una architettura software appropriata guidi lo sviluppo. Per definire l'architettura è necessario scoprire quali sono le attività del sistema esistente e localizzare i componenti realmente usati e utili. Scoprire le attività del sistema permette di identificare linee di codice inutile, la quale eliminazione provoca l'inserimento di difetti che possono essere corretti solo comprendendo bene la causa dell'errore. Per evitare Lava Flow bisogna identificare e documentare interfacce software stabili

Functional Decomposition (aka No OO)	
Forma	Risultato del lavoro di sviluppatori con esperienza non OO che progettano ed implementano un'applicazione in un linguaggio OO, creando codice che assomiglia ad un linguaggio strutturale (Pascal, C, Fortran).
Sintomi	Classi con nomi di funzione (es. CalcolaInteresse); classi con una singola operazione (o poche operazioni); nessun uso di ereditarietà e polimorfismo; le classi presenti sono difficili da documentare, poiché senza senso ed il sistema difficile da testare.
Cause	Sviluppatori che non hanno compreso la tecnologia OO e mancanza di una architettura ben definita
Soluzione	Analizzare il sistema software e determinare quali sono i requisiti principali, documentarli e produrre un design che incorpora le parti principali del sistema. Bisogna combinare insieme piccole classi che insieme hanno un unico obiettivo, e se una classe non contiene uno stato, riscriverla come una funzione.

Spaghetti Code	
Forma	Le progressive estensioni hanno compromesso la struttura a tal punto che non è più comprensibile, la struttura sembra inesistente, e il sistema è difficile da mantenere ed estendere.

Sintomi	I metodi sono lunghi, senza parametri e usano variabili globali; Il flusso di esecuzione è obbligato dall'implementazione degli oggetti, non dai client degli oggetti; le interazioni tra oggetti sono minime; nomi di classi e metodi suggeriscono la programmazione procedurale; ereditarietà e polimorfismo non sono usati, riuso impossibile e gli oggetti non mantengono lo stato tra varie invocazioni di metodo.
Cause	Inesperienza con la tecnologia OO o nessuna progettazione prima dell'implementazione.
Soluzione	Convertire frammenti di codice in metodi; rimuovere codice che possono diventare inaccessibili; rinominare classi, metodi e dati per rendere il codice più facile da leggere; condurre una progettazione orientata agli oggetti e identificare oggetti piccoli, comprensibili agli sviluppatori.

Cut-and-paste Programming	
Forma	Frammenti di codice simili sparsi nel sistema, duplicazione di codice. Programmatori con poca esperienza che prendono esempio da programmati con esperienza, con modifiche a breve termine che soddisfano i requisiti.
Sintomi	Lo stesso bug su ripete malgrado molte correzioni sono state effettuate; Linee di codice vengono aggiunte senza migliorare la produttività; Difficile localizzare e sistemare tutte le istanze di un certo errore; Alti costi di manutenzione, difetti replicati; Tante correzioni ma nessun modo di correggere le tante versioni; Tipo di riuso che falsamente incrementa il numero di linee di codice.
Cause	Creare codice riusabile è faticoso e si preferisce ritorno d'investimento immediato anziché a lungo termine; La velocità di sviluppo del codice è più importante dei fattori di qualità; Gli sviluppatori non lavorano con le astrazioni, non hanno compreso ereditarietà e composizione; I componenti una volta creati non sono documentati sufficientemente e resi disponibili prontamente agli altri sviluppatori; Gli sviluppatori non pensano a quello che verrà dopo.
Soluzione	Ridurre o eliminare cloning; Fare refactoring per eliminare le tante versioni tramite parametrizzazione, e creare un set di componenti riusabili; Riusare tali componenti in modalità black-box, ovvero tramite composizione.

Excessive Dynamic Allocation	
Problema	Creazione e distruzione frequente di oggetti di una stessa classe che porta al decadimento delle prestazioni se avviene su un grande numero di oggetti. Per la creazione: allocazione di memoria, inizializ. codice, inizializ. Oggetto. Per la distruzione: esecuzione del garbage collector.
Soluzione	Cambiare il codice per eliminare molte esecuzioni di new (ad es. se new è dentro un ciclo, potrebbe essere portato fuori); Riciclare oggetti anziché crearne di nuovi (gestione con object pool); Eliminare la necessità di avere nuovi oggetti (usando ad es. il pattern Flyweight) Individuando e condividendo lo stato intrinseco immutabile (oggetto Flyweight) e separarlo dallo stato estrinseco (dipendente dal contesto).

Manutenzione e metriche

Manutenzione

Dicasi manutenzione il processo di introduzione di modifiche ad un prodotto software dopo la sua consegna al cliente.

Durante il ciclo di vita e l'uso di un qualsiasi software le aspettative e le condizioni in cui esso era stato sviluppato possono mutare, determinando la necessità di un adeguamento. Ad esempio si può verificare la necessità di ottemperare a nuove normative fiscali (per prodotti commerciali), oppure la segnalazione da parte di utenti di difetti nel software (**bug**) che necessitano correzioni più o meno urgenti.

Evoluzione

Spesso usato con lo stesso significato di manutenzione, dicasi evoluzione un singolo passo di un processo di manutenzione che prevede: evoluzione, rilascio di patch, rimozione sistema.

Dati statistici

I costi di manutenzione rappresentano il 67-80% dei costi del software. I cambiamenti si possono raggruppare in 4 categorie:

- Correttivi - rimozione errori (17%)
- Adattativi - aggiustamenti per un nuovo ambiente (18%)
- Perfettivi - miglioramento e aggiunta di funzionalità (60%)
- Preventivi - modifiche interne per prevenire problemi (5%)

Incorporare nuove funzionalità è la porzione maggiore di modifiche. È buona pratica anticipare i cambiamenti a design time /tramite parametrizzazione, incapsulamento, ecc.).

Dinamiche di evoluzione

Le dinamiche di evoluzione sono i processi di cambiamento di un sistema; essi si basano su dati empirici effettuati da Lehman e Belady (dal 1968 e confermati da studi recenti) su sistemi software di grandi dimensioni sviluppati da grandi aziende.

Leggi di Lehman

- o **Legge del cambiamento continuo [1974]**: I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti
- o **Legge dell'aumento della complessità [1974]**: Quando un sistema evolve, la sua struttura aumenta di complessità, a meno che del lavoro viene fatto per preservare o semplificare la sua struttura
- o **Legge dell'auto-regolazione [1974]**: Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti
- o **Legge della stabilità organizzativa [1978]**: Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo
- o **Legge della conservazione di familiarità [1978]**: In media, l'incremento di crescita di un sistema tende a rimanere costante o a diminuire
- o **Legge della continua crescita [1978]**: Il contenuto di funzioni di un sistema deve continuamente essere incrementato per mantenere la soddisfazione dell'utente
- o **Legge della Diminuzione della qualità [1994]**: La qualità di un sistema diminuisce se non viene rigorosamente gestita ed adattata durante i cambiamenti

Applicabilità delle leggi

Tali leggi sono in generale applicabili a grandi sistemi sviluppati da grandi organizzazioni. Non è chiaro come si adattano a piccoli prodotti, prodotti che incorporano un certo numero di COTS e a piccole organizzazioni.

Costi e modelli di manutenzione

Nel caso di mutate condizioni, la manutenzione del software può risultare molto difficoltosa poiché le modifiche necessarie spesso sono causa di profonde ristrutturazioni del prodotto. In questo caso si parla di "nuova versione" (o più comunemente si usa il termine inglese **release**) che può integrare o sostituire quella precedente.

Nel caso di semplici malfunzionamenti (detti *bachi* o, più comunemente, **bug**) vengono invece rilasciati dei file in grado di correggere l'errore nel software: in questo caso si parla di "pezza" (più comunemente si usa il termine inglese **patch**).

In entrambi i casi la difficoltà è maggiore se il codice è stato scritto in maniera **poco modulare**, senza dividerlo cioè in tante piccole parti in modo che ognuna si occupi di un compito preciso. È buona norma, quando si scrive un programma, evitare al massimo la ripetizione di codice seguendo uno standard (alla stregua del processo di normalizzazione in uso nei database). Se la stessa combinazione di istruzioni viene eseguita in varie parti dell'applicazione è conveniente racchiuderla in una routine riutilizzabile. Qualora si rendesse necessario ritoccare la procedura lo si potrà fare agendo in un punto solo con evidenti vantaggi in termini di tempo ed affidabilità.

Fattori che contribuiscono al costo:

- Il costo è ridotto se il team di sviluppo è coinvolto nella manutenzione
- Gli sviluppatori potrebbero non avere responsabilità contrattuali per la manutenzione, quindi non hanno incentivi a fare un design che può essere cambiato in futuro
- La struttura del programma si degrada mano a mano che si introducono cambiamenti

Modelli di manutenzione

- **Quick-fix**: cambiamenti a livello di codice
- **Miglioramento iterativo**: cambiamenti fatti in base ad un'analisi del sistema esistente; controllo della complessità e mantenimento del design
- **Riuso**: stabilire i requisiti per il nuovo sistema, riusando il più possibile

Tipi di modifiche

- o **Re-factoring o re-structuring**: processo di cambiamento del software che non altera il comportamento del codice ma migliora la struttura interna, ovvero bisogna prendere un sistema fatto male e modificarlo per ottenere una struttura ben fatta.
- o **Reverse engineering**: analizzare un sistema per estrarre informazioni sul suo comportamento o sulla sua struttura.
- o **Re-engineering**: Alterare un sistema per ricostituirlo in un'altra forma.

Metriche

Una metrica è uno standard per la misura di alcune proprietà del software o delle sue specifiche. Da tali metriche si basano le **valutazioni metriche** che permettono di dare un giudizio oggettivo all'operato svolto per il software. Esse si suddividono in due tipi di metriche:

- **Metriche del prodotto**: si concentrano sugli specifici attributi dei prodotti dell'ingegneria del software e vengono raccolte mentre vengono svolti i compiti tecnici (analisi, progettazione, programmazione e collaudo);
- **Metriche per il processo e il progetto**: considerano il processo o progetto nella sua interezza.

Valutazioni metriche

Le valutazioni metriche rappresentano gli elementi in base ai quali le attività di analisi, progettazione, codifica e collaudo possono essere condotte più obiettivamente e stabilite più quantitativamente. Esse consentono di conoscere l'efficacia dei processi e dei progetti software, e consentono anche di raccogliere dati relativi alla qualità e alla produttività, i quali verranno poi analizzati e confrontati con le medie precedenti per determinare se si sono verificati miglioramenti nella produttività.

Le valutazioni metriche vengono utilizzate anche per evidenziare aree problematiche in modo da sviluppare soluzioni e migliorare il processo del sviluppo software.

Obiettivi

Le metriche aiutano gli ingegneri software a valutare meglio la progettazione e la costruzione dei prodotti. Gli obiettivi dell'adozione di metriche sono:

- Monitorare il prodotto mentre si costruisce
- Identificare i livelli per ciascuna metrica

- Rimediare in caso i livelli non sono soddisfacenti

Solo gli attributi interni possono essere misurati direttamente (es. dimensione), quelli esterni sono ricavati indirettamente.

Le metriche software possono essere utilizzate per:

- stimare il budget per il progetto e la codifica
- stimare la produttività individuale e la qualità
- stimare la produttività del progetto e la qualità
- stimare la qualità del software

Le valutazioni metriche del software sono analizzate e definite dal **manager software**, mentre le misure vengono spesso raccolte dagli ingegneri software.

Metriche tradizionali

La metrica software tradizionale comunemente adottata sono:

➤ Complessità ciclomatica (CC)

Utilizzata per misurare la complessità di un programma, misura direttamente il numero di cammini linearmente indipendenti attraverso il grafo di controllo di flusso del programma.

I nodi del grafo corrispondono a gruppi indivisibili di istruzioni, mentre gli archi connettono due nodi se il secondo gruppo di istruzioni può essere eseguito immediatamente dopo il primo gruppo. La complessità ciclomatica può, inoltre, essere applicata a singole funzioni, moduli, metodi o classi di un programma.

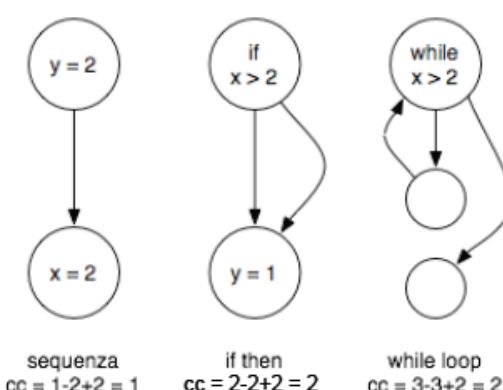
La complessità ciclomatica di una sezione di codice è il numero di **cammini linearmente indipendenti** attraverso il codice sorgente. Per esempio, se il codice sorgente non contiene punti decisionali come IF o cicli FOR, allora la complessità sarà 1, poiché esiste un solo cammino nel sorgente (e quindi nel grafo). Se il codice ha un singolo IF contenente una singola condizione, allora ci saranno due cammini possibili: il primo se l'IF viene valutato a TRUE e un secondo se l'IF viene valutato a FALSE. La definizione di "cammini linearmente indipendenti" risiede nella definizione di lineare indipendenza a livello algebrico, semplicemente estesa ai cammini su grafi orientati. Una complessità ciclomatica di 5, significa che 5 è il numero massimo di cammini possibili tra loro indipendenti e ogni altro cammino possibile sul grafo si può costruire a partire da uno di quei 5 (vale a dire una combinazione lineare di uno di essi). Trovare i 5 cammini indipendenti significa trovare una base per l'insieme di tutti i possibili cammini di un dato grafo G.

Matematicamente, la complessità ciclomatica di un programma strutturato è definita in riferimento ad un grafo diretto contenente i blocchi base di un programma con un arco tra due blocchi se il controllo può passare dal primo al secondo (il "grafo di controllo di flusso"). La complessità è quindi definita come:

$$v(G) = e - n + 2p$$

Dove $v(G)$ è la complessità ciclomatica del grafo G, e è il numero di archi del grafo, n è il numero di nodi del grafo e p è il numero di componenti connesse.

Es.



➤ Lines of code (LOC)

Misura le dimensioni di un software basandosi sul numero di linee di codice sorgente. Questo metodo di misura viene utilizzato per stabilire la complessità di un software e per stimare le risorse necessarie per la produzione e il mantenimento del software.

Va detto che la misura delle LOC è utile per definire un ordine di grandezza del codice ma non per fornire una misura esatta di un progetto software. A tal riguardo ha senso usare le linee di codice per confrontare un progetto di 10.000 linee con uno da 100.000, mentre ne ha poco paragonarne uno da 20.000 con uno da 21.000.

Ci sono due tipi di misure di LOC:

- *Physical LOC (PLOC)*: si contano tutte le righe di testo del codice sorgente includendo anche i commenti e le linee bianche se la loro percentuale non supera il 25% delle linee.
- *Logical LOC*: si contano gli "statements", ovvero le effettive istruzioni (per esempio in Java si considera LOC ogni istruzione terminante con ;)

Es. Consideriamo questo frammento di codice: `for (i=0; i<100; ++i) print("hello");`

In questo codice abbiamo: 1 Physical Lines of Code, e 2 Logical Lines of Code (un for e una print).

Se invece lo stesso codice è scritto con uno stile diverso:

```
for (i=0;  
i<100; ++i)  
{  
    print("hello");  
}
```

Le LOC saranno: 4 Physical Lines of Code, e 2 Logical Lines of Code.

con

Inoltre, nell'ambito LOC, possiamo gestire il conteggio dei commenti le seguenti misure:

- **Effective lines of code (ELOC)**: Questa metrica conta tutte le linee effettive del codice.
- **Non comment lines of code (NLOC)**: Questa metrica conta tutte le linee effettive del codice eccetto i commenti.
- **Non comment non blank (NCNB)**: Questa metrica conta tutte le linee effettive del codice eccetto i commenti e le linee vuote.
- **Comment lines of code (CLOC)**: Questa metrica conta solo le linee di commento del codice.

Definiamo la metrica **Comment percentage (CP)** come la percentuale di linee di commento rispetto a tutte le linee di codice (escluse le linee vuote), calcolabile con la formula:

$$CP = \frac{CLOC}{NLOC + CLOC}$$

Questa metrica è usata per valutare gli attributi di comprensibilità, riuso e manutenzione. Il CP ideale si attesta attorno al 30%.

Metriche object-oriented

Suite di Chidamber & Kemerer o metriche CK

- o **WMC (Weighted Methods per Class)**. La somma delle complessità dei metodi per una classe; se i metodi hanno pari complessità, WMC è il numero di metodi della classe. Un alto valore di WMC indica grande lavoro di manutenzione, grande specificità della classe e quindi poca possibilità di riuso.
- o **DIT (Depth of Inheritance Tree)**. Massimo numero di livelli dalla classe alla radice della gerarchia (radice=0). Più alto è DIT per una classe, più difficile è capirne il comportamento, data la grande quantità di metodi ereditati. Alto DIT indica maggiore complessità dell'intero sistema, ma anche maggiore riuso.
- o **NOC (Number of Children of a Class)**. Numero di sottoclassi di una classe della gerarchia. Indica l'influenza che una classe ha sul sistema; maggiore è NOC, maggiore è il riuso.
- o **CBO (Coupling Between Object Classes)**. Numero di classi con cui una classe interagisce. Maggiore è CBO maggiore è la dipendenza di una classe da altre, quindi minore possibilità di riuso, maggiore difficoltà a comprendere, modificare e correggere la classe.

- o **RFC (Response for a Class)**. Numero di metodi eseguiti al ricevimento di un messaggio, cioè numero di metodi di una classe + numero di metodi invocati da ciascuno di essi. Alto RFC indica grande complessità, quindi difficoltà di comprensione e di testing.
- o **LCOM (Lack of Cohesion of Methods)**. Per ogni campo della classe, si calcola la percentuale di metodi. Alta coesione (basso LCOM) indica semplicità, elevata possibilità di riuso.

Test e collaudo del software

In informatica, il collaudo del software (detto anche **testing** o software testing secondo le denominazioni inglesi) è un procedimento, che fa parte del ciclo di vita del software, utilizzato per individuare le carenze di correttezza, completezza e affidabilità delle componenti software in corso di sviluppo. Consiste nell'eseguire il software da collaudare, da solo o in combinazione ad altro software di servizio, e nel valutare se il comportamento del software rispetta i requisiti.

Malfunzionamenti e difetti

In generale, occorre distinguere i "malfunzionamenti" del software (in inglese, "*failure*"), dai "difetti" (o "errori" o "bachi") del software (in inglese, "*fault*" o "*defect*" o "*bug*").

- Un **malfunzionamento** è un comportamento del software difforme dai requisiti esplicativi o impliciti. In pratica, si verifica quando, in assenza di malfunzionamenti della piattaforma (hardware + software di base), il sistema non fa quello che l'utente si aspetta.
- Un **difetto** è una sequenza di istruzioni, sorgenti o eseguibili, che, quando eseguita con particolari dati in input, genera un malfunzionamento. In pratica, si ha un malfunzionamento solo quando viene eseguito il codice che contiene il difetto, e solo se i dati di input sono tali da evidenziare l'errore. Per esempio, se invece di scrivere " $a \geq 0$ ", il programmatore ha erroneamente scritto " $a > 0$ " in una istruzione, si può avere un malfunzionamento solo quando viene eseguita quell'istruzione mentre la variabile "a" vale zero. I difetti possono essere raggruppati in classi, in accordo alle fasi di sviluppo del software:
 - *Difetti di specifiche*: la descrizione di ciò che il prodotto fa è ambigua, contraddittoria o imprecisa
 - *Difetti di design*: le componenti o le interazioni tra queste sono progettati in modo non corretto per la progettazioni di algoritmi (es. divisione per zero), strutture dati (es. campo mancante, tipo sbagliato), interfaccia moduli (parametri di tipo inconsistente), etc.
 - *Difetti di codice*: errori derivanti dall'implementazione dovuti a poca comprensione del design o dei costrutti del linguaggio di programmazione (es. overflow, conversione tipo, priorità delle operazioni aritmetiche, variabili non inizializzate, non usate tra due assegnazioni, etc.). A volte è difficile classificare se un difetto è di design o di codice
 - *Difetti di test*: i casi di test, i piani per i test, etc. possono avere difetti

Per rilevare il maggior numero possibile di difetti, nel collaudo si sollecita il software in modo che sia eseguita la maggior quantità possibile di codice con svariati dati di input.

Può anche darsi che un errore nel codice sorgente generi un malfunzionamento solo se si utilizza un particolare compilatore o interprete, o solo se eseguito su una particolare piattaforma. Pertanto può essere necessario collaudare il software con vari ambienti di sviluppo e con varie piattaforme di utilizzo.

Scopo

Lo scopo del collaudo è di rilevare i difetti tramite i malfunzionamenti, al fine di minimizzare la probabilità che il software rilasciato abbia dei malfunzionamenti nella normale operatività.

Nessun collaudo può ridurre a zero tale probabilità, in quanto le possibili combinazioni di valori di input validi sono enormi, e non possono essere riprodotte in un tempo ragionevole; tuttavia un buon collaudo può rendere la probabilità di malfunzionamenti abbastanza bassa da essere accettabile dall'utente.

Probabilità di malfunzionamento

L'accettabilità di una data probabilità di malfunzionamento dipende dal tipo di applicazione. Il software per cui è richiesta la massima qualità, è quello cosiddetto "*life-critical*", cioè in cui un malfunzionamento può mettere a rischio la vita umana, come quello per apparecchiature

medicali o aeronautiche. Per tale software è accettabile solo una probabilità di malfunzionamento molto bassa, e pertanto il collaudo è molto approfondito e rigoroso.

Processo di Verifica e Valutazione (V&V)

La fase di verifica e di validazione serve ad accertare che il software rispecchi i requisiti e che li rispetti nella maniera dovuta. Precisamente:

- o La **verifica** serve a stabilire che il software rispetti i requisiti e le specifiche, quindi ad esempio che non ci siano requisiti mancanti; il sistema dovrebbe essere conforme alle sue specifiche (stiamo costruendo il prodotto nel modo giusto?)
- o La **validazione** serve ad accertare che i requisiti e le specifiche siano anche rispettati nella maniera giusta; il sistema dovrebbe fare ciò che l'utente ha realmente chiesto (stiamo costruendo il giusto prodotto?)

Questa fase è molto delicata in quanto, dopo tutto il processo si può ottenere un software perfettamente funzionante, senza errori, ma del tutto inutile in quanto non rispecchia quanto era stato chiesto all'inizio.

Secondo il modello applicato, questa fase si applica su stadi intermedi o su tutto il software.

Scopo

Il processo di V & V ha due obiettivi principali: scoprire i difetti del sistema e valutare se il sistema è usabile in una situazione operativa

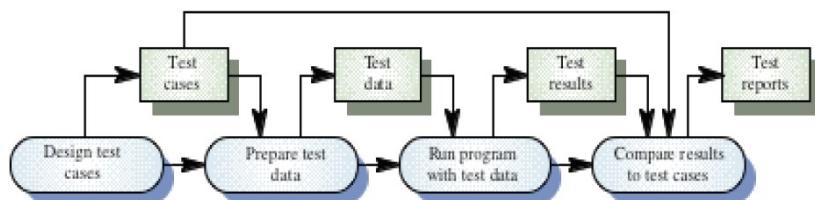
La verifica può essere *statica*, se effettuata su carta, cioè sul progetto, o *dinamica*, se effettuata attraverso il collaudo dello stesso software con dati di test.

Definizioni in ambito test

Il test del software può rivelare la presenza di errori, non la loro assenza; un test ha successo se scopre uno o più errori, e dovrebbero essere condotti insieme alle verifiche sul codice statico.

La fase di test ha come obiettivo rivelare l'esistenza di difetti in un programma. Un particolare tipo di test è il **debugging**. Il debugging si riferisce alla localizzazione ed alla correzione degli errori. Esso formula ipotesi sul comportamento del programma e verifica tali ipotesi trovando gli errori.

- **Test data (dati di test):** dati di input che sono stati scelti per testare il sistema
- **Test case (caso di test):** dati di input per il sistema e output stimati per tali input nel caso in cui il sistema operi secondo le sue specifiche. Gli input sono non solo parametri da inviare ad una funzione, ma anche eventuali file, eccezioni, e stato del sistema, ovvero le condizioni di esecuzione richieste per poter eseguire il test
- **Test suite:** insiemi di test case



Principi per i test

Fare test è il processo di esercitare i componenti usando un set selezionato di casi di test con l'intento di

- (i) Rivelare difetti
- (ii) Valutare la qualità

Quando l'obiettivo è trovare difetti allora un buon test case è uno che ha buona probabilità di rivelare difetti non noti, e i risultati dei test dovrebbero essere letti meticolosamente. Un test case deve contenere i risultati aspettati e dovrebbe essere sviluppato sia per condizioni di input valide che non valide.

La probabilità di esistenza di difetti addizionali per un componente software è proporzionale al numero di difetti del componente già individuati; i difetti spesso accadono in gruppi, e spesso un codice che ha grande complessità ha un cattivo design.

I test dovrebbero essere effettuati da un gruppo indipendente dal gruppo di sviluppatori, poiché gli sviluppatori sono orgogliosi del codice prodotto, inoltre possono avere difficoltà a capire dove trovare difetti, poiché il loro modello mentale oscura il codice reale.

I test devono essere ripetibili e riusabili (test di regressione), dovrebbero essere pianificati e i piani di test dovrebbero specificare gli obiettivi, allocare tempo e risorse umane e monitorare i risultati. Le attività di test dovrebbero essere integrate nel ciclo di sviluppo del software.

Difficoltà per chi fa i test (tester)

- Deve avere una conoscenza vasta delle discipline di ingegneria del software
- Deve avere conoscenza ed esperienza su come un software è descritto (specifiche), progettato e sviluppato
- Deve essere in grado di gestire molti dettagli
- Deve conoscere quali tipi di fault possono generare i costrutti del codice
- Deve ragionare come uno scienziato per proporre ipotesi che spiegano la presenza di tipi di difetti
- Deve avere una buona comprensione del dominio del software
- Deve creare e documentare casi di test, quindi selezionare gli input che con maggiore probabilità possono rivelare difetti
- Necessita di lavorare e cooperare con chi si occupa di requisiti, design, sviluppo codice e spesso con clienti ed utenti

Testing

L'obiettivo del testing è di stabilire la presenza di difetti nei sistemi. Un test ha successo se il test fa sì che il programma si comporti in modo anomalo. Esistono essenzialmente due tipi di test:

- **Test dei componenti (detti anche unit test):** test dei singoli frammenti (metodi, classi, etc.) effettuato dallo sviluppatore del componente, progettato in base a tecniche note ed all'esperienza dello sviluppatore
- **Test di integrazione:** test di gruppi di componenti già integrati (interagenti) che formano un sistema o un sottosistema. La responsabilità è di un team di test, e i test sono basati sulle specifiche

Test esaustivo e priorità

Solo un test esaustivo può mostrare se un programma è privo di difetti. I test esaustivi sono impraticabili.

Es. Una funzione che prende in ingresso 2 int, per essere testata esaustivamente dovrebbe essere eseguita $2^{32} \times 2^{32}$ volte, ovvero circa 1.8×10^{19} volte. Se la funzione esegue in $1\text{ ns} = 10^{-9}\text{s}$ occorrono $1.8 \times 10^{10}\text{s}$ ovvero, essendo $1\text{ Y} = 3 \times 10^7$, 600 anni!

Le priorità dei test, dunque, devono essere:

1. I test dovrebbero mostrare le capacità del software più che eseguire i singoli componenti
2. Il test delle vecchie funzionalità è più importante del test delle nuove
3. Testare situazioni tipiche è più importante rispetto a testare situazioni limite

Strategie di test

Un approccio in cui i test vengono effettuati senza avere conoscenza di come il sistema è fatto (ovvero della sua struttura interna) si dice **test black-box**, ovvero considera il sistema una scatola nera, dove i test case sono progettati sulla base della descrizione del sistema, ovvero partendo dal documento di specifiche del sistema

È possibile studiare (e predisporre) i test nelle fasi iniziali dello sviluppo del software. Dall'insieme dei dati di input possibili si individua il sottoinsieme che può rivelare la presenza di difetti nel sistema in modo da progettare casi di test efficaci.

Un altro approccio è quello **white-box** che focalizza sulla struttura interna del software da testare, bisogna avere a disposizione il codice sorgente (o un opportuna rappresentazione tramite pseudo-codice).

Entrambi gli approcci sono usati per rendere la fase di test più efficiente.

Partizionamento in categorie equivalenti

Nel caso di test black-box, un buon modo per selezionare gli input per il test al sistema è ricorrere a partizioni in categorie (classi) equivalenti. Dati di input e risultati si possono spesso raggruppare in **categorie** in cui tutti i membri di una categoria sono relazionati. Ognuna delle categorie è una **partizione equivalente**, ovvero mi aspetto che il programma effettui elaborazioni simili (equivalenti) per ciascun membro della stessa categoria.

Testare uno dei valori membri di una categoria equivale a testare ciascun altro valore della stessa categoria; viene meno la necessità di test esaustivi e permette di coprire un grande dominio con un piccolo set di valori.

I test case dovrebbero essere scelti da ciascuna partizione

Es. Una funzione può prendere in input solo numeri da 4 a 20.

Partizioni: numeri <4; numeri tra 4 e 20; numeri >20

Dati di test da scegliere: 3, 4, 12, 20, 21

Chi fa il test deve considerare sia categorie di equivalenza valide che categorie di equivalenza non valide. Una categoria di equivalenza non valida rappresenta input inaspettati o errati.

Categorie di equivalenza possono essere selezionate anche per le condizioni di output.

Non ci sono regole forti per individuare le categorie di equivalenza => il partizionamento è un processo euristico, tester diversi potrebbero individuare categorie diverse. Può essere difficile identificare categorie di equivalenza. Una lista di condizioni per il partizionamento potrebbe essere:

1. Se una condizione per l'input è specificata come un range di valori ammessi (o un numero di valori contigui), selezionare una categoria valida costituita dal range e due classi invalide, ciascuna ad un estremo del range
2. Se una condizione per l'input è data da un numero di valori, selezionare una categoria valida che include i valori consentiti e due invalide per i valori fuori da ciascun estremo del set
3. Se una condizione per l'input è data da un set di valori, selezionare una categoria valida costituita dai valori e una invalida per i valori fuori dal set
4. Se una condizione per l'input è descritta come "deve essere" bisogna considerare due categorie, una valida che rappresenta la condizione "deve essere" e una invalida che non include la condizione "deve essere"
Es. Il testo deve iniziare con una vocale
5. Se si crede che un elemento di una categoria di equivalenza non verrà trattato in modo identico agli altri elementi della categoria allora la classe deve essere ulteriormente partizionata in classi di equivalenza più piccole.

Es. Specifica: un identificatore deve avere da 3 a 15 caratteri alfanumerici ed i primi due caratteri devono essere lettere (notare il "deve essere"). Partizioni:

- Nome alfanumerico, classe valida
- Nome non alfanumerico, classe invalida
- Nome tra 3 e 15 caratteri, classe valida
- Nome con meno di 3 caratteri, classe invalida
- Nome con più di 15 caratteri, classe invalida
- Primi due caratteri lettere, classe valida
- Primi due caratteri non lettere, classe invalida

Test strutturali

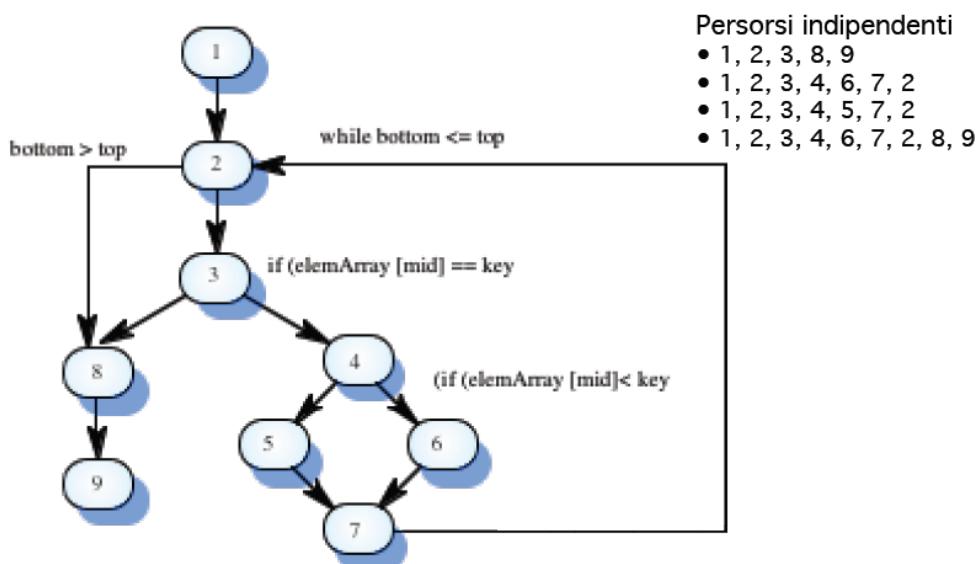
Chiamati anche test **white-box**, **glass-box**, o **clear-box**, sono test (addizionali a quelli black-box) derivati dalla conoscenza della struttura del programma. La finalità è di eseguire tutti i costrutti del programma (non tutte le combinazioni dei percorsi).

Test del percorso

La finalità è assicurare che i test case siano tali che ogni percorso all'interno del programma sia eseguito almeno una volta. È utile rappresentare il programma tramite un grafo di flusso dove i nodi rappresentano condizioni del programma e gli archi il flusso di controllo.

Per testare tutti le condizioni, il numero di test da effettuare è data dalla complessità ciclomatica

($cc = \text{numero archi} - \text{numero nodi} + 2$). Tutti i percorsi sono eseguiti, ma non tutte le combinazioni dei percorsi.



I test case dovrebbero essere scelti in modo che tutti i percorsi siano eseguiti. Un tool può essere usato a runtime per controllare che i percorsi siano stati eseguiti.

Test di integrazione

Sono test eseguiti su sistemi completi o su sottosistemi, dovrebbero essere black-box e derivati dalle specifiche. La principale difficoltà è di localizzare gli errori, ma effettuare i test di integrazione in maniera incrementale riduce tale problema.

Per i test di integrazione incrementali, sull'insieme dei componenti A, B si eseguono le suite di test T1, T2, T3, successivamente, mentre sull'insieme di componenti A, B, C si eseguono le suite di test T1, T2, T3, T4, etc.

Approcci per i test di integrazione

- **Top down:** integra i sotto-sistemi (componenti) di più alto livello e successivamente quelli dei livelli un po' più bassi. Sostituisce i componenti con stub quando appropriato. Permette di scoprire errori nell'architettura del sistema e di mettere a punto versioni demo nelle fasi iniziali.
- **Bottom-up:** integrare singoli componenti di basso livello e successivamente tali integrazioni con componenti di livello più alto. Rende la scrittura dei test più semplice. In pratica, ciò che avviene è una combinazione dei due precedenti approcci.

Test sotto stress

Questi test eseguono il sistema oltre il massimo carico previsto, consentendo di rendere evidenti i difetti presenti. Il sistema eseguito oltre i limiti consentiti non dovrebbe fallire in modo catastrofico. I Test di stress indagano su perdite, di servizio o dati, ritenute inaccettabili. Essi sono particolarmente rilevanti per i sistemi distribuiti che possono subire degradazioni in dipendenza delle condizioni della rete.

Stress

Stressare un sistema significa agire su:

- o *Prestazioni*: inserire i dati con frequenza molto alta, o molto bassa
- o *Strutture dati*: funziona per qualsiasi dimensione dell'array?
- o *Risorse*: test con poca memoria RAM, numero basso di file che possono essere aperti, connessioni di rete, etc.

Testing manuale e automatico

Testing manuale

I test case manuali sono liste di istruzioni per una persona che collaudano una singola azione:

- 1) Click su "login"
- 2) Inserisci username e password
- 3) Click su "ok"
- 4) Inserisci il dato ...

L'uso di test manuali è comune quando non è sostituibile grazie al test di usabilità, non è pratico da automatizzare o troppo costoso, e le persone che fanno i test non sanno gestire automatismi complessi.

Testing automatico

Non sono altro che dei test manuali registrati e rieseguirsi automaticamente con macro, script o programmi appositi (es. AutoHotkey). Spesso è poco robusto poiché smette di funzionare se cambia qualcosa dell'ambiente (es. posizione campi, nome campi, etc.).

Permettono di sviluppare programmi che eseguono il test sul codice, e chiamano funzioni, confrontano risultati, etc.

Test regressivi

Test i quali scoperto un difetto costruiscono un test che permette di rilevare il difetto ed eseguono lo stesso test tutte le volte che il codice viene cambiato. Il difetto non riapparirà più.

I test regressivi assicurano di non ritornare a versioni che presentano difetti già corretti. In pratica, eseguo spesso i test già scritti, sempre che la loro esecuzione non ha durata proibitiva, dunque ciascun test dovrebbe durare il meno possibile.

Copertura del codice

Per stabilire fino a quando dovremmo continuare a fare test, utilizziamo la metrica **Copertura del codice** (Code coverage). Bisogna dividere il programma in unità (es. costrutti, condizioni, comandi) e definire la copertura che dovrebbe avere la suite di test (es. 60%). La copertura codice sarà data dalla formula:

$$\text{copertura codice} = \frac{\text{numero di unità già eseguite}}{\text{numero di unità del programma}}$$

Si smette di eseguire test quando si è raggiunta la copertura desiderata. Avere una copertura del 100% non significa non avere difetti, basti pensare ad esempio ai diversi dati di input scelti. Parti critiche del sistema possono avere copertura maggiore di altre parti. La misura di copertura permette di capire se alla suite di test manca qualcosa.

Trend di difetti scoperti

Al momento un trend in voga nel testing di un prodotto software è il **bug trend**, il quale misura la frequenza con cui i difetti sono trovati. Quando la frequenza tende a zero non ci sono più difetti, ma vi è un drammatico aumento dei costi di ricerca dei difetti

Esistono anche delle pratiche standard molto in voga negli odierni software come:

- Eseguire i test spesso e lavorare con nuove versioni giornaliere o settimanali (**nightly build**)
- Fare progressi in avanti (**regression test**)
- Condizioni di stop (**coverage, bug trend**)