Processi di sviluppo del software:
Un processo software descrive le attività o task necessarie allora sviluppo di un prodotto software e come queste attività sono collegate tra loro.
Attività o fasi dello sviluppo:

- Analisi dei requisiti.
- Progettazione.
- Codifica o implementazione.
- Convalida o testing.
- Manutenzione.

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.

- Requisito: ciascuna delle caratteristiche il software deve avere.
- Specifica: descrizione rigorosa delle caratteristiche del software.
- Feature: un set di requisiti correlati tra loro.

Fasi per l'ingegneria dei requisiti:

1. Studio di fattibilità.
2. Analisi dei requisiti.
3. Specifica dei requisiti.
4. Convalida dei requisiti.

Requisiti:

- Funzionali: Cosa il sistema deve fare.
- Non-funzionali: Come il sistema lo fa.

Progettazione ed implementazione:
Fase di progettazione:
Il processo che stabilisce la struttura software che realizza le specifiche.
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, algoritmi.

Ognuna delle attività suddette produce un documento che descrive un modello.
Modello degli oggetti, di sequenza, di transizione stati, strutturale, data-flow.
La fase di implementazione produce un programma eseguibile a partire dalla struttura stabilita.
Progettazione ed implementazione sono attività correlate e spesso sono alternate.

Fase di implementazione:
Consiste nella programmazione ovvero nella traduzione dei modelli del progetto in un programma e della rimozione degli errori dal programma.
I programmatori effettuano alcuni test sul programma prodotto per scoprire bug e rimuoverli.
Per rimuovere i bug:

1. Localizzare l'errore nel codice.

2. Rimuovere l’errore nel modello e poi nel codice.
3. Effettuare nuovamente il test del programma.

Fase di convalida:
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 del cliente.
Viene condotta tramite processi di revisione e test del sistema software.
I test mirano ad eseguire il sistema software in condizioni dalla specifiche di dati reali che il sistema dovrà elaborare.

Fase di test:
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.

Evoluzione:
Il software è intrinsecamente flessibile e può cambiare.
Al cambiamento dei requisiti per cambiamenti dell’ambiente a cui è rivolto, il software deve evolvere se deve rimanere ad essere utile.

Modello a cascata (Waterfall):
Si comincia con la fase successiva solo se la fase precedente è completa.
Processo statico con tanta documentazione.
- Lungo tempo per ottenere il prodotto.
- Poche interazioni con il cliente.
- Difficoltà ad introdurre i cambiamenti richiesti dal cliente.
- Consistenza tra artefatti.
- Ampia documentazione
- Utile se i requisiti sono stabili e chiaramente definiti.
- Usato principalmente per sistemi grandi, complessi, critici, per gestire team numerosi.
- Alta qualità del codice prodotto.

Processo evolutivo:
Il processo evolutivo ha due varianti: esplorazione e Build and Fix.
Sviluppo per esplorazione:
- Gli sviluppatori lavorano con i clienti.

- Dalle specifiche iniziali si arriva per mezzo di trasformazioni successive fino al sistema software finale.
- Dovrebbe partire da requisiti ben chiari ed aggiungere nuove caratteristiche definite dal cliente.

Sviluppo Build and Fix:
- Documentazione inesistente o quasi.
- Comprensione limitata del sistema da produrre.

Problemi:
- Tempi lunghi.
- Sistemi difficilmente comprensibili e modificabili, probabilmente non corretti.
- Mancanza di visione d'insieme del progetto.

Applicabilità:
- Sistemi di piccole dimensioni.
- Singole parti di sistemi grandi.
- Sistemi con vita breve.

Modello a Spirale:
Focalizza su tanti prodotti parziali.
Ogni loop è una fase e consiste nei seguenti settori:
1. Identificazione obiettivi specifici per la fase corrente.
2. Valutazione rischi del progetto.
3. Produzione di una parte e convalida della parte.
4. Revisione del progetto e pianificazione fase successiva.

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.
Settore del processo a Spirale:
Stabilire obiettivi: gli obiettivi per la fase corrente sono identificati.
Valutare il rischio e ridurlo: i rischi sono valutati ed attività sono intraprese per ridurre quelli più importanti.
Sviluppo e convalida: secondo uno dei modelli precedenti.
Pianificazione: il progetto è revisionato e la prossima fase della spirale è pianificata.

Rational Unified Process (RUP):
Processo iterativo.
Fasi: avvio, elaborazione, costruzione, transizione (e produzione).
Fase di avvio (inception):
- Comunicazione con il cliente.
- Identificazione dei requisiti.
- Identificazione architettura base.
- Pianificazione.

Fase di elaborazione:
- Comunicazione con cliente.
- Raffinamento ed ampliamento dei casi d'uso.

- Espansione della rappresentazione dell'architettura.
- Sviluppo piano di progetto e rischi.

Fase di costruzione:

- Sviluppa o acquisisce i componenti software che serviranno agli utenti nei vari casi d'uso.
- Test sui componenti, integrazione dei componenti.
- Alla fine di queste fase il sistema dovrebbe esse funzionante.

Fase di transizione:

- Fornisce una versione del sistema su cui sono condotti i beta test.
- Sposta il sistema dall'ambiente di sviluppo all'ambiente reale.

Fase di produzione:

- Viene monitorato l'utilizzo del software.

Pratiche RUP:

Ciascuna iterazione è svolta entro un tempo prefissato:

Evitare di identificate molti requisiti all'inizio. Progettazione un'architettura che mette insieme le parti identificate e permette il riuso di componenti esistenti.

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 iterzione.

Gestione cambiamenti:

Disciplinare gestione configurazioni, controllo versioni, protocollo di richiesta di cambiamenti, release minime alla fine di ogni iterazione.

Confronto tra processi:

Processo	*Punti di forza*	*Punti di debolezza*
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
Spirale	Approccio disciplinato regolato da documenti; valutazione dei rischi	Può essere usato solo per sistemi grandi
RUP	Approccio iterativo; uso di componenti; modellazione grafica	Generico => Poco chiaro (?)
Build and fix	Approccio per piccoli sistemi che non necessitano manutenzione	Architettura incidentale; non soddisfacente per sistemi non banali
XP	Permette l'adattamento dei requisiti	Poca documentazione

Refactoring:
È il processo che cambia un sistema software in modo che il comportamento esterno del sistema non cambi e la struttura interna sia migliorato.
Chiamato anche: miglioramento del design dopo che è stato scritto il codice.
L'idea del refactoring è di riconoscere che è difficile fare fin dall'inizio un buon design (e codice) e, man mano che i requisiti cambiano, il design deve essere cambiato.
Vantaggi:

- Spesso la dimensione del codice si riduce dopo il refactoring.
- Le struttura complicate si trasformano in struttura più semplici, più facili da capire e manutenere.
- Si evita di introdurre un debito tecnico all'interno del design.

Metodi di refactoring:

1. Estrai metodo
   Un frammento di codice può essere raggruppato. Far diventare quel frammento un metodo il cui nome spiega lo scopo del frammento.
2. Sostituisci Temp con Query.
   Una variabile temporanea (temp) è usata per tenere il risultato di una espressione. Estrarre l'espressione ed inserirla in un metodo. Sostituire tutti i riferimenti a temp con l'espressione. Il nuovo metodo può essere usato anche altrove.
3. Dividi variabile Temp
   Una variabile temporanea è assegnata più di una volta, ma non è una variabile assegnata in un loop o usata per collezionare valori.

Sviluppo agile:
Sono più importanti auto-organizzazione, collaborazione, comunicazione tra membri del team e adattabilità del prodotto rispetto ad ordine e coerenza delle attività del progetto.
Privilegiare:

Individui, disponibilità di software funzionante, collaborazione con il cliente, pronta risposta ai cambiamenti.

Agilità:

- 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)
Approccio basato sullo sviluppo e la consegna di piccoli incrementi di funzionalità.

- Solo 2 settimane per lo sviluppo degli incrementi.
- Piccoli gruppi di sviluppatori.
- Costante miglioramento del codice.
- Poca documentazione: uso di Story Card e CRC.
- Enfasi su comunicazione diretta tra persone.

- Iterazioni corte e di durata costante.
- Coinvolgimento di sviluppatori, clienti e manager.

XP è adatto per progetti in cui i requisiti non sono stabili. I rischi sono grandi.

Principi di XP: avere feedback rapidamente, assumere la semplicità, cambiamenti incrementali, supportare i cambiamenti, produrre lavoro di qualità.

12 Pratiche di XP:

1. Gioco di pianificazione.
2. Piccole release.
3. Metafora.
4. Testing.
5. Refactoring.
6. Pair Programming.
7. Cliente in sede.
8. Design semplice.
9. Possesso del codice collettivo.
10. Integrazione continua.
11. Settimana di 40 ore.
12. Usare gli standard per il codice.

Story Card:
Storie utente (story card) = casi d'uso leggeri. Le dimensioni delle card sono circa 12x7cm.
Le story card 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.
Template per story card:

- Data, numero, priorità, tempo stimato, riferimento.
- Descrizione requisito.
- Lista di task per ciascun requisito, ovvero ciò che lo sviluppatore dovrà fare.
- Note.

1. Gioco di pianificazione:
Gli utenti scrivono le storie.
Gli sviluppatori stimano il tempo per lo sviluppo di ciascuna storia.
Gli utenti dividono, fondono e assegnano priorità alle storie.
Gli addetti al business prendono decisioni su date per le release, contesto, priorità dei task.
Pianificare l'intera release e la nuova iterazione.
Per l'attuale release, gli sviluppatori dividono ciascuna storia in task, stimano i task, ciascuno si impegna per realizzare un task dando priorità a quelli più rischiosi.

2. Piccole release:
Rendere ogni release il più piccola possibile.
Effettuare un design semplice e sufficiente per la release corrente.

Piccole release forniscono agli sviluppatori di rilasciare feedback rapidamente, rischio ridotto, la fiducia del clienti, possibilità di fare aggiustamenti per requisiti che cambiano.

3. Metafora:
Guidare il progetto con una singola metafora.

4. Testing:
Si testa tutto ciò che potenzialmente può andare male per tutto il tempo.
I test sono la specifica dei requisiti.
Test funzionali:

- Scritti dall'utente.
- Effettuati da: utenti, sviluppatori e team di testing.
- Automatizzati.
- Eseguiti almeno giornalmente.
- Sono una parte della specifica dei requisiti, quindi documentano requisiti.

Unit test:

- Scritti dagli sviluppatori.
- Scritti prima e dopo della codifica.
- Supportano design, codifica, refactoring e qualità.

5. Refactoring:
Refactoring significa migliorare la struttura del codice senza influenzarne il comportamento. Fatto in piccoli passi. Supportato da unit test, design semplice e pair programming. Punta a codice senza ripetizioni.

6. Pair Programming:
Programmatori esperti e motivati, scambio dei partner.
Pair programming aiuta la disciplina, sparge la conoscenza sul sistema.

7. Cliente in sede:
Scrive i test funzionali. Stabilisce priorità e fornisce il contesto per le decisioni dei programmatori. Risponde alle domande. Porta avanti il suo lavoro.

8. Design semplice:
Il giusto design per il software si ha quando: passa i test, non ha parti duplicate, esprime ciascuna intenzione importante per i programmatori, ha il numero più piccolo di classi e metodi.
Non preoccuparsi di dover apportare cambiamenti.
Usare le CRC card per documentare il design permettendo di ragionare meglio in termini di oggetti, inoltre contribuiscono a fornire una visione complessiva del sistema.

9. Possesso del codice collettivo:
Chiunque può aggiungere qualunque codice su qualunque parte del sistema. Unit test proteggono le funzionalità del sistema. Chiunque trova un problema lo risolve. Ciascuno è responsabile per l'intero sistema.

10. Integrazione continua:
Integrazione del codice testato ogni poche ore.
Tutti gli unit test devono essere superati.
Se un test fallisce la coppia che ha prodotto il codice deve ripararlo.
Se non può ripararlo, buttare il codice e ricominciare.

11. Settimana di 40 ore:
Se per te non è possibile fare il lavoro in 40 ore, allora hai troppo lavoro.
40 ore a settimana di lasciano "fresco" per risolvere i problemi.
Prevenire l'inserimento di errori difficili da trovare.
Pianificazioni frequenti evitano a ciascuno di avere troppo lavoro.
Ore extra di lavoro è sintomo un problema serio.

12. Standard di codifica:
Costruzioni complicati (per il design) non sono permesse.
Il codice appare uniforme e quindi più facile da leggere.
Usare tutti la stessa convenzione, così non si ha la necessità di riformattare il codice. Sapere come usare: spazi e tab per indentazione, posizione parentesi graffe, scelta di nomi classi, metodi, attributi, posizione commenti.

Gestione progetti:
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 organizzazione di sviluppo e di acquisto.
Attività di gestione dei progetti:

- Scrittura della proposta.
- Pianificazione e scheduling progetto.
- Individuazione costi di progetto.
- Monitoraggio e revisioni progetto.
- Selezione e valutazione personale.
- Scrittura report e presentazione.

Personale:
Non è sempre possibile avere le persone ideali per lavorare su un progetto. Il budget potrebbe non consentire di usare il personale molto costoso. Il personale con l'esperienza appropriata potrebbe non essere disponibile. Un organizzazione potrebbe voler formare del personale lavorando su un progetto.

Pianificazione del progetto:
Pianificazione: gestione delle risorse umane e dei costi, individuazione dello scheduling necessario per produrre i risultati sperati.
La pianificazione è probabilmente l'attività che prende più tempo.
Piani di tipo differenti possono essere sviluppati a supporto del piano principale che si concentra su scheduling e budget.

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.
Il processo a cascata permette direttamente la definizione di milestone.

Scheduling del progetto:
Dividere il progetto in task e stimare tempo e risorse necessarie a completare ciascun task.
Organizzare i task in modo concorrente per fare un uso ottimale della forza lavoro.
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.

Problemi della schedulazione:
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.

Diagrammi:
Uso di notazioni grafiche per illustrare lo scheduling del progetto.
I digrammi delle attività mostrano le dipendenze tra i task ed i percorsi critici.
I diagrammi a barre mostrano lo scheduling su un calendario.

Ingegneria dei requisiti:
I requisiti sono la descrizione dei servizi del sistema.
La gestione dei requisiti consiste in:

- Un approccio per l'estrazione, l'organizzazione e la documentazione dei requisiti di un sistema.
- Nel processo che stabilisce e mantiene l'accordo tra il cliente ed il team del progetto.

Definizione di requisito:

Una capacità del software, che l'utente necessità per risolvere un problema o per ottenere un risultato.
Una capacità che il software deve avere per soddisfare un contratto, uno standard, una specifica.

Tipi di requisti:

- Requisiti utente:
  Descrizioni, in linguaggio naturale e diagrammi, dei servizi che il sistema fornisce.
- Requisiti di sistema:
  Un documento strutturato che dettagli i servizi del sistema. Scritto come contratto tra cliente e fornitore.
- Specifiche del software (SRS)
  Descrizione dettagliata del software che serve come base per il design o l'implementazione. Scritto per gli sviluppatori.

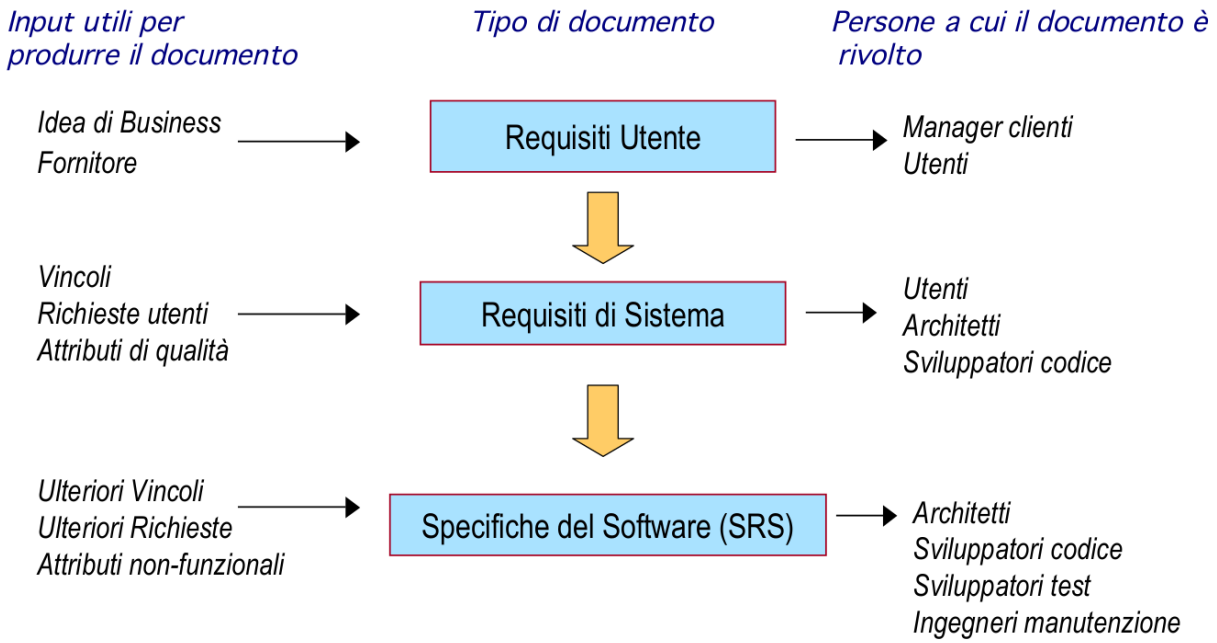
I requisiti si dividono in funzionali e non.
I requisiti devono essere completi e consistenti.
Completi: includere la descrizione di tutto ciò che è richiesto.
Consistenti: non ci devono essere contraddizioni nella loro descrizione.
I requisiti non funzionali possono essere più critici di quelli funzionali, essi si dividono in:

- Requisiti di prodotto: specificano il comportamento del prodotto.
- Requisiti organizzativi: quelli derivati da prassi organizzative.
- Requisiti esterni: derivati da fattori esterni.

Metriche per 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

Livelli di requisiti:



Requisiti utente:
Dovrebbero descrivere requisiti funzionali e non in modo comprensibile per chi non ha conoscenze tecniche dettagliate. Sono definiti usando il linguaggio naturale, tabelle e diagrammi.
Problemi con il linguaggio naturale: mancanza di chiarezza, confusione.

Requisiti di sistema:
Sono più dettagliati dei requisiti utente ma non dovrebbero imporre scelte di design e implementazione. Servono come base per il design. Possibili notazioni per la scrittura dei requisiti come ad esempio l'uso del linguaggio naturale, linguaggio naturale strutturato, program, notazioni grafiche, formulazione matematica.

Specifiche del software (SRS):
Descrive cosa è richiesto agli sviluppatori, non è un documento di design. Dice cosa il sistema dovrebbe fare e non come.
Struttura (template) suggerita da IEE standard 830 1984:

1. Introduzione: scopo dell'SRS, ambiente del prodotto, definizione acronimi, abbreviazioni, overview.
2. Descrizione generale: scopo del prodotto, funzioni del prodotto, caratteristiche, utenti, vincoli.
3. Requisiti specifici: usa un formato standard per descrivere ciascun requisito funzionali.
4. Requisiti di interfaccia: interfaccia utente, interfaccia hardware, comunicazione in rete, interfacce con altri software.
5. Requisiti di performance.
6. Vincoli di design: aderenza a standard, limiti hardware.
7. Attributi non-funzionali: sicurezza, affidabilità, manutenzione.
8. Scenari operativi: casi d'uso.
9. Piano di progetto preliminare.
10. Appendici: definizioni, riferimenti.

Studio di fattibilità:
Decide se costruire il sistema è di interesse sulla base di: contributi agli obiettivi dell'organizzazione, ingegnerizzare con la tecnologia corrente entro il budget, integrazione con altri sistemi usati. Raccolta informazioni.

Scoperta e analisi dei requisiti:
Richiede colloqui con i clienti per ricavare ciò che il sistema deve fornire ed i suoi vincoli operazionali.
Attività:

- Comprensione del dominio.
- Collezione requisiti.
- Classificazione in gruppi coerenti.
- Risoluzione conflitti tra requisiti.
- Stabilire priorità
- Controllare (revisionare) requisiti.

Analisi dei requisiti:
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.

Convalidare i requisiti:
Ha lo scopo di mostrare che i requisiti definiscono il sistema che il cliente vuole.
Gli errori nei requisiti costano e quindi la convalida è importante.

- 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 bugdet, le tecnologie e nel tempo a disposizione?
- Verificabilità: sarà possibile mostrare che il sistema soddisfa i requisiti?

Tecniche di convalida dei requisiti:

- Revisione requisiti: analisi sistematica dei requisiti per scoprire anomalie e omissioni.
- Prototipazione: uso di modello (prototipo) del sistema per verificare i requisiti.
- Generazione test: sviluppare test dei requisiti per controllarli.
- Analisi di consistenza automatica: controllo della consistenza di una descrizione dei requisiti strutturata o formale.

Revisione:
Il processo di revisione consiste di dialoghi tra clienti e fornitori.
Gli obiettivi sono Consistenza e Completezza, inoltre si possono verificare anche Comprensibilità, Tracciabilità e Adattabilità.

La gestione dei requisiti è il processo di gestione dei cambiamenti durante lo sviluppo del sistema.

Modelli dei requisiti:
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.

Tipi di modelli:

- Comportamentali: modello per l'elaborazione dei dati, modello di stimoli/risposte.

- Strutturali: modello architetturale, modello di composizione, modello di classificazione.

Progettazione archiettura:
L'architettura software di un sistema è un artefatto, frutto della attività di progettazione. Un architettura software è descritta dai suoi componenti e relazioni tra essi.

- Componenti: costituiscono i "building block" di un sistema.
- Relazioni: denotano una connessione tra componenti (aggregazione, eredità, interazione).

Un sottosistema è un sistema le cui operazioni sono indipendenti dai servizi di altri sottosistemi.
Un componente fornisce servizi ad altri componenti e non è considerato come un sistema a sé stante.

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

Stili archietturali:
L'architettura può essere conforme ad un modello o stile.
La conoscenza degli stili può semplificare la definizione dell'architettura per un sistema.
I sistemi grandi sono eterogenei e non seguono un singolo stile.
Caratteristiche importanti di un architettura dovrebbero esibire modularità e information hiding.

Modello repository:
Adatto a problemi per cui non sono note soluzioni deterministiche.
I sottosistemi necessitano di scambiare dati tra loro.
I dati sono in un database centrale accessibile dai sottosistemi.
Adatto per sistemi che condividono grandi quantità di dati.

Modello client-server:
Ambiente distribuito.
Set di server che forniscono specifici ad altri sottosistemi.
Set di client che chiamano i server.
La rete permette ai client di accedere ai server.

Modello a macchina a stratta (o a strati):
Organizza un sistema in un set di strati ognuno dei quali fornisce un gruppo di servizi ad un certo livello di astrazione.
Supporto lo sviluppo incrementale dei diversi livelli. Quando l'interfaccia di un livello cambia solo il livello adiacente è influenzato.
Le prestazioni possono degradare.

Event-driven:
Eventi esterni pilotano i sottosistemi di elaborazione.

Broadcast: un evento è inviato a tutti i sottosistemi, ogni sottosistema che è in grado di gestire l'evento lo gestisce, i sottosistemi decidono se l'evento è di interesse.
Interrupt-driven: usati in sistemi realtime.

Pipe and filter:
Ogni filtro trasforma i dati in input e passa i risultati in output.
I filtri non sanno a chi sono connessi.
I filtri possono essere implementati in parallelo e riusati.
Il comportamento del sistema è la composizione dei filtri.

Riflessione computazionale:
Un sistema riflessivo è un sistema a livelli in cui:

- Il livello sottostante non conosce che esistono i livelli superiori.
- Un livello superiore è in grado di ridefinire alcune operazioni dei livello sottostante e di alterare le attività ed il flusso di esecuzione del livello inferiore.

A runtime, il livello superiore intercetta le operazioni del livello inferiore e fa introspezione del livello inferiore.

UML – Unified Modeling Language:
È una famiglia di notazioni grafiche per modellazione visuale del software.
Modellazione: rappresentazione di elementi che corrispondo a parti del software.
L'UML può essere utilizzato in diversi modi.
Come abbozzo (sketch) per documentare alcuni aspetti:

- Prima che il sistema sia sviluppato (forward engineering)
- Partendo da un sistema esistente (reverse engineering)

Come progetto dettagliato, per guidare la realizzazione di un sistema. Lo scopo principale è fornire agli sviluppatori un modello dettagliato su cui basarsi. I diagrammi creati fanno parte della documentazione del sistema e vanno modificati per rispecchiare il sistema stesso.
Come linguaggio per programma infatti alcuni tool generano codice direttamente dall'UML.

I diagrammi UML si dividono in:
Diagrammi strutturali:

- Diagramma delle classi.
- Diagramma dei componenti.
- Diagramma di deployment.

Diagrammi comportamentali:

- Diagramma dei casi d'uso.
- Diagramma delle attività.
- Diagramma di macchina a stati.
- Diagramma di interazione.

Diagramma dei casi d'uso:

Un diagramma di caso d’uso descrive il comportamento del sistema come apparato dall’esterno. Individua le funzionalità del sistema significative per gli attori.
Attori: persone o sistemi fuori dal prodotto che si sta sviluppando e che interagiscono con esso.
Un diagramma di caso d’uso descrive una interazione come una sequenza di messaggi tra il sistema e uno o più attori.
Un caso d’uso è una unità che offre funzionalità del sistema visibili all’esterno.
In un caso d’uso un attore sta cercando 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.

Caratteristiche dei casi d’uso:
Descrivono l’interazione di un sistema con il suo ambiente.
Non rivelano la struttura interna del sistema ovvero non mostrano attività e componenti interni ad esso.
Sono espressi informa di testo e con diagramma.
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.

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

Modelli di comportamento:
Sono usati per descrivere il comportamento globale del sistema:
Data processing model:
- Mostrano i passi per l’elaborazione di dati che attraversano il sistema.
- Notazione intuitiva comprensibile ai clienti.
- Mostrano lo scambio di informazioni tra sistemi e sottosistemi.
- Simili al diagramma delle attività UML

State machine model:
- Modellano il comportamento in risposta ad eventi interni o esterni.
- Mostrano stati del sistema come nodi ed eventi come archi tra i nodi.
- Quando un evento si verifica, il sistema passa da uno stato ad un altro.
- Utili per sistemi real-time poiché spesso pilotati da eventi.

Diagramma delle attività:
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.

Mostra dipendenza tra attività.
Non indicano quali sono gli oggetti che svolgono le attività.
Sono il punto iniziale della progettazione.
Vanno ri-elaborati per arrivare ad assegnare un o più operazioni ad una classe che le implementa.
Posso essere usati come punto di partenza per ottenere i diagrammi UML di collaborazione fra oggetti.

Diagramma UML degli stati:
Descrive un intervalli di tempo durante la vita di un oggetto, esso è caratterizzato da valori di oggetti o intervallo di cui un oggetto aspetta certi eventi o intervallo in cui un oggetto fa certe azioni.

Transazione: permette di lasciare uno stato in risposta ad un certo evento, un oggetto gestisce un solo evento alla volta.

Stati composti:
Uno stato composto è uno stato che consiste di vari sottostati sequenziali o concorrenti. Solo uno dei sottostati sequenziali può essere attivo in un certo momento. Lo stato esterno rappresenta la condizione di essere in uno qualsiasi degli stati interni. Una transizione verso o da uno stato composto può provocare varie azioni di entry o exit.

Modelli ad oggetti:
Descrivono il sistema in termini di classi (OOP). Una classe ha attributi ed operazioni comuni ad un set di oggetti. Vari modelli ad oggetti possono essere prodotti.

Notazione UML per classi e interfacce:
Esistono varie notazioni per le classe e l'interfaccia.
Le notazioni indicano:

Nome classe; nome classe e attributi; nome classe, attributi e metodi.
Per la visibilità e metodi: + public, - private, #protected.
I nomi delle interfacce sono in corsivo.

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

Diagramma UML delle classi:
Il diagramma delle classi mostra le classi, le loro caratteristiche e le loro relazioni (ereditarietà, implementazione, associazione, uso).
Un associazione descrive una connessione tra istanza delle classi e specifica:

- Molteplicità.
- Nome ruolo.
- Navigabilità.
- Nome.

Diagramma UML di ereditarietà:
Organizza le classi in una gerarchia:

- Le classi in alto nella gerarchia (superclassi) mostrano le proprietà comuni delle classi in basso (sottoclassi).
- Le classi ereditano gli attributi e i servizi da uno o più super-classi.

Design pattern:
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 micro-architetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti.

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

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 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 comprensione dei principi del design.
- Analizzano le loro proprietà non-funzionali, ovvero come una funzione è portata a termine.

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

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

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: 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.

Progettazione dei design pattern:

Creare un oggetto specificandone la classe esplicitamente.

- Orienta ad un particolare implementazione invece che ad una interfaccia.
- Può complicare i cambiamenti futuri.
- È meglio creare oggetti indirettamente.
- Pattern utili: Abstract Factory, Factory Method, Protorype.

Le dipendenze da operazioni specifiche.

- Vincolano ad un modo di soddisfare una richiesta.
- Modificare a compile-time e run-time le richieste è più facile se si evita che le richieste siano inserite nel codice.
- Pattern: Chain of Responsability, Command.

Dipendenze da piattaforme hardware e software.

- API esterne al sistema cambiano.
- Progettare il sistema limitando le dipendenza dalla piattaforma.
- Pattern: Abastract Factory, Bridge.

Dipendenze da rappresentazioni o implementazioni di oggetti.

- Client che conoscono come un oggetto è rappresentato, conservato, o implementato posso necessitare di essere cambiati quando l'oggetto cambia.
- Nascondere informazioni ai client evita cambiamenti in cascata.
- Pattern: Abstact Factory, Bridge, Memento, Proxy.

Dipendenze da algoritmi.

- Gli algoritmi sono spesso estesi, ottimizzati e rimpiazzati durante lo sviluppo e il riuso.
- Si devono isolare gli algoritmi soggetti a cambiamenti.
- Pattern: Builder, Iterator, Strategy, Template Method, Visitor

Stretto accoppiamento.

- Porta a sistemi monolitici.
- Classi strettamente accoppiate sono difficili da riusare singolarmente.
- Pattern: Abstract Factory, Bridge, Chain of Responsability, Command, Façade, Mediator, Observer.

Estendere funzionalità tramite sottoclassi: white box reuse.

- Richiede profonda comprensione della superclasse.
- Ovveride di una operazione può chiedere ovveride di un'altra.
- Può condurre a un grande numero di classi.
- Quindi è meglio limitare il numero di livelli in una gerarchia e, quando possibile, è opportuno usare la delegazione anziché ereditare.
- Pattern: Bridge, Chain of Responsaibilty, Composite, Decorator, Observer, Strategy.

Impossibilità di modificare classi.

- Sorgenti non disponibili.
- La modifica può richiedere cambiamenti a tante altre classi.
- Pattern: Adapter, Decorator, Visitor.

Design pattern creazionali:
Permettono di astrarre il processo di creare oggetti rendendo un sistema indipendente da come i suoi oggetto sono creati, composti e rappresentati.
Sono importanti se i sistemi evolvono per dipendere più su composizioni di oggetti che su ereditarietà tra classi.
Incapsulano conoscenza sulle classi concrete un sistema usa.
Nascondono come le istanze delle classe sono create e composte.

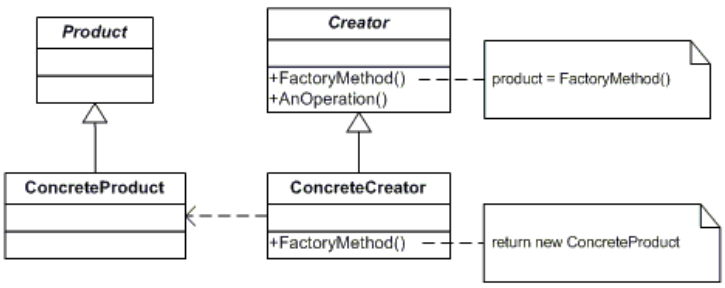
Factory Method:
Intento:
- Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare.
- Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi.

Motivazioni:
- Un framework usa classi astratte per definire e mantenere relazioni tra oggetti.
- Il framework deve creare oggetto ma conosce solo classi astratte che non può istanziare.
- Un metodo responsabile per l'istanziazione incapsula la conoscenza quale classe creare.

Soluzione:
- Product è l'interfaccia comune degli oggetti che il FactoryMethod() crea.
- ConcreteProduct è una implementazione dell'interfaccia Product.
- Creator è l'interfaccia che dichiara il FactoryMethod(). Tale metodo ritorna un oggetto di tipo Product.
- ConcreteCreator implementa il FactoryMethod() scegliendo quale ConcreteProduct istanziare e ritorna tale istanza.

Varianti:
- Il FactoryMethod() ha un parametro in ingresso per far indicare al client la classe da creare.
- Il FactoryMethod() usa Class.forName() e newIstance() per togliere dal codice la dipendenze da una specifica classe.

Conseguenze:
- Il codice delle classi dell'applicazione conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct.
- È necessario implementare una sottoclasse di Creator per istanziare ciascun ConcreteProduct.



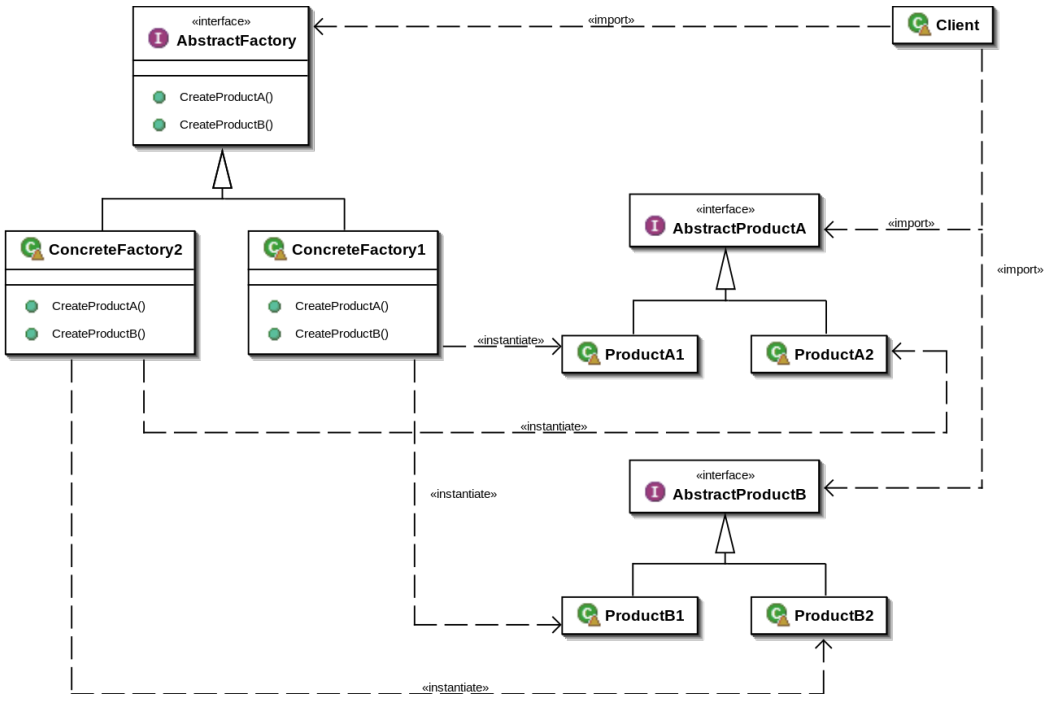
Abstract Factory:
Intento:
- Fornire una interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete.

Problema:
- 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.

Soluzione:
- Interfaccia astratta per le famiglie di classi.
- Classi per creare ciascuna famiglia di classi.
- Classi concrete.

- 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.
- Le famiglie di classi sono facilmente intercambiabili
- Non è facile supportare nuove classi Product poiché bisogna aggiugnere un metodo su AbstractFactory e su ogni ConcreteFactory.

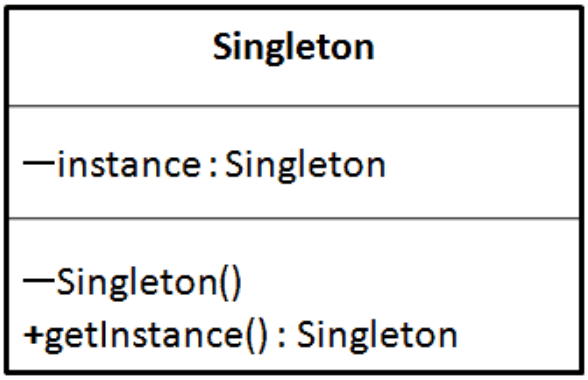


Singleton:
Intento:
- Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale all'istanza.

Motivazione:
- Alcune classi dovrebbero avere esattamente un istanza.
- 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.

Soluzione:
- Singleton definisce un operazione istance() per la classe 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.
- Permette di controllare il numero di istanza create in un programma, facilmente ed in un solo punto.
- La soluzione è più flessibile rispetto a quella di usare static per tutte le variabili poiché si può cambiare facilmente il numero di istanze consentite.



Design pattern strutturali:
Se agiscono sulle classi usano l'ereditarietà per comporre interfacce o classi.
Se agiscono sugli oggetti descrivono modi per comporre oggetti, composizione di un oggetto può variare a run-time.
Permettono di diminuire le dipendenze tra classi o tra algoritmi.

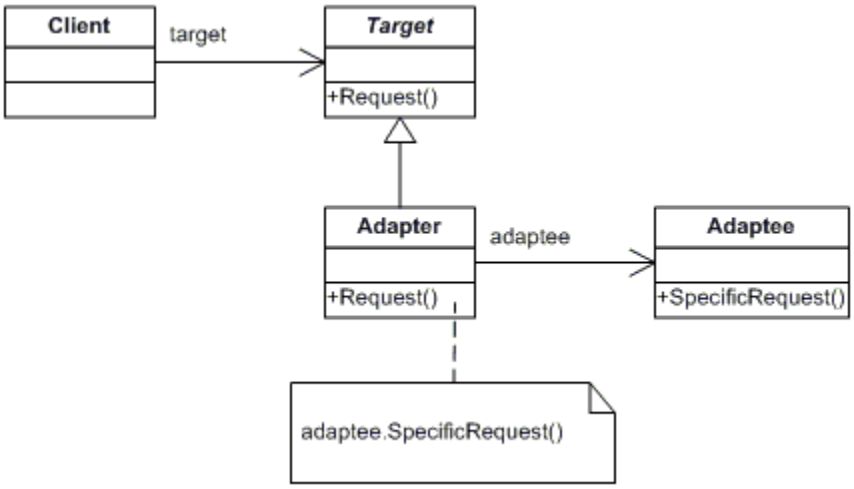
Adapter:
Intento:
- Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano.
- Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce compatibili.

Motivazione:
- Certe volte volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia richiesta dall'applicazione.
- Non è possibile cambiare l'interfaccia di quella libreria.
- Non è possibile cambiare l'applicazione.

Soluzione:
- 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'Adpter possiede il riferimento all'oggetto di libreria e sa come invocarlo.

Variante:
- La classe Adapter include l'interfaccia di Adaptee.
- La versione Class Adapter fornisce l'Adapter da due vie.

Conseguenze:
- Client e classe di libreria rimangono indipendenti.
- L'Adapter può cambiare comportamento dell'Adaptee.
- L'Adapter aggiunge un livello di indirettezza.



Facade:
Intento:
- 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.

Problema:
- 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 delle classi.
- Si vogliono ridurre le comunicazione e le dipendenze dirette tra i client ed il sottosistema.

Soluzione:

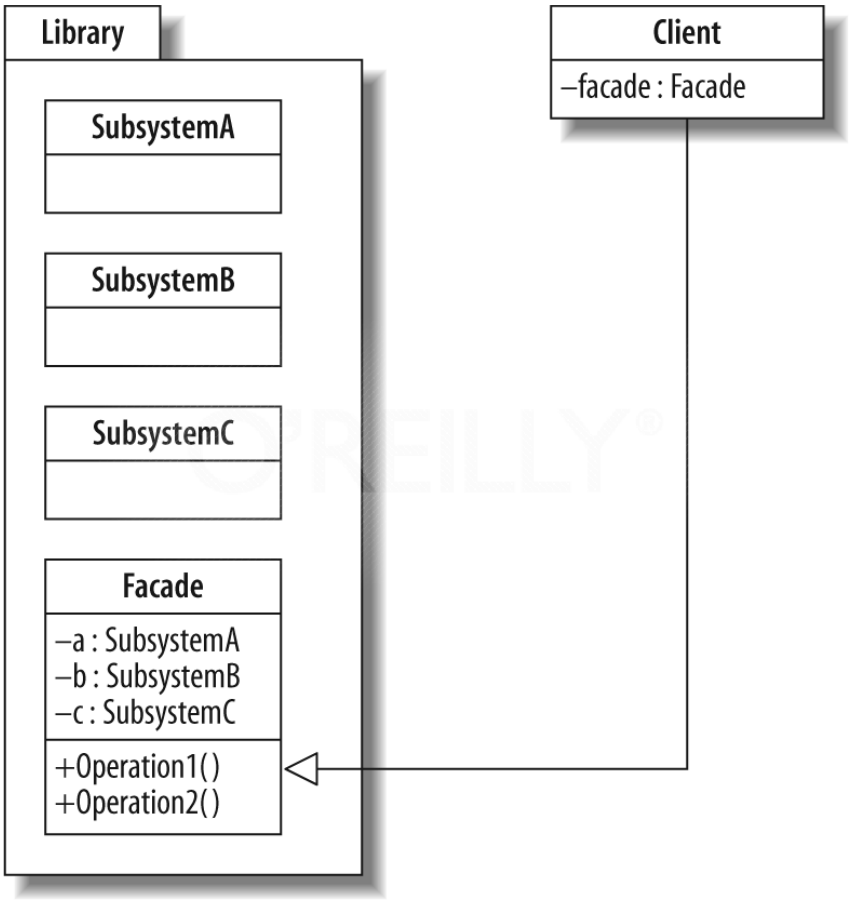
- Un modo per ridurre la complessità è introdurre un oggetto Facade che fornisce un'unica interfaccia semplificata e nasconde gli oggetti del sottosistema.
- Il client interagisce solo con l'oggetto Facade.
- Il Facade invoca i metodi degli oggetti che nasconde.

Conseguenze:

- Nasconde ai client l'implementazione del sistema.
- 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.

Implementazione:

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



Composite:

Intento:

- Comporre oggetti in strutture ad albero per rappresentare gerarchie di parti o del tutto. Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente.

Motivazione:

- È necessario raggruppare elementi semplici tra loro per formare elementi più grandi.
- 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.
- Composite permette di descrivere una composizione ricorsiva, in modo che i client non debbano fare distizioni tra tipi di elementi.

Soluzione:

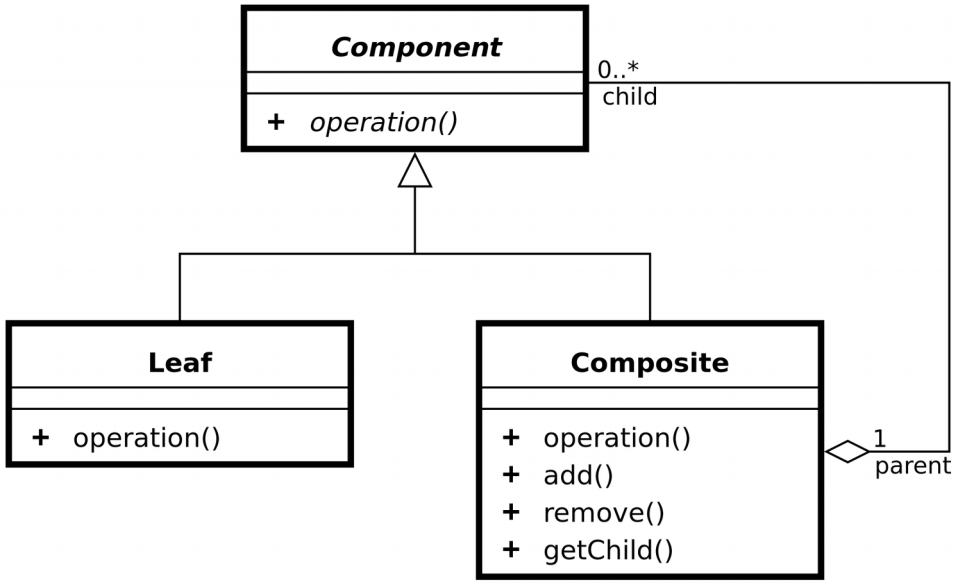
- Component: classe abstract, rappsenta elementi semplici e non:
  Dichiara le operazioni degli oggetti della composizione.
  Implementa le operazioni comuni alle sottoclassi, in modo appropriato.
  Dichiara le operazioni per l'accesso e la gestione di elementi semplici.
  Può definire un'operazione che permette ad un elemento di accedere all'oggetto padre nella struttura ricorsiva.
- Leaf: classe che rappresenta elementi semplici (detti child):
  È sottoclasse di Component.
  Implementa il comportamento degli oggetti semplici.
- Composite: classe che rappresenta elementi contenitori:
  Definisce il comportamento per l'aggregato di elementi child.
  Tiene il riferimento a ciascuno degli elementi child.
  Implementa operazioni per gestire elementi child.

Collaborazioni:

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

Conseguenze:

- Elementi semplici possono essere composti in elementi più complessi, questi possono esser composti, e così via.
- Un client che aspetta un elemento semplice può riceverne uno composto.
- I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite.
- Nuovi tipi di elementi (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 componenti Leaf (in base al tipo), invece dovranno essere fatti dei controlli a runtime.



Design Pattern Comportamentali.
Focalizzano sul controllo del flusso tra oggetti.
Descrivono le comunicazioni tra oggetti.
Aiutano a valutare le responsabilità assegnate agli oggetti.
Suggeriscono modi per incapsulare algoritmi dentro classi.

Mediator:
Intento:

- Definisce un oggetto che incapsula come un gruppo di oggetti interagisce.
- Promuove lasco accoppiamento fra oggetti, poiché permette ad essi di non interagire direttamente.

Motivazione:

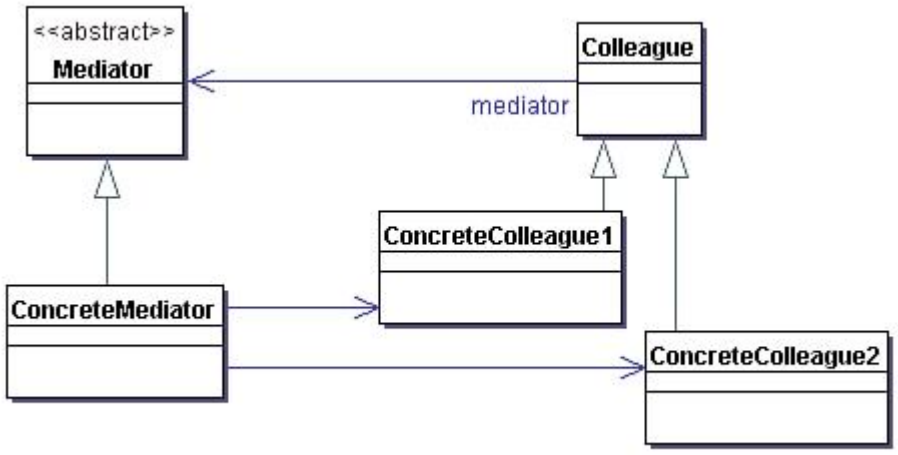
- La distribuzione di responsabilità fra vari oggetti può risultare in molte connessioni fra oggetti.
- Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico.
- Diminuire le dipendenze di una classe e renderla più generale

Soluzione:

- Isolare le comunicazioni (complesse) tra oggetti dipendenti creando una classe separata per esse.
- Mediator definisce una interfaccia tra oggetti che comunicano.
- ConcreteMediator implementa il comportamento cooperativo e coordina Colleague.
- Colleague definisce un'interfaccia per oggetti che devono comunicare.
- ConcreteColleague conosco il Mediator e comunicano con il Mediator quando avrebbero comunicato con altri ConcreteColleague.

Conseguenze:

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



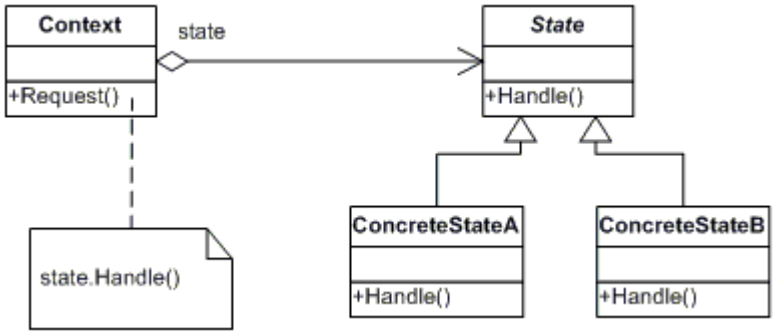
State:
Intento:
- Permette ad un oggetto di alterare il suo comportamento quando il suo stato cambia e sembra che l'oggetto abbia cambiato la sua classe.

Applicabilità:
- Il comportamento di un oggetto dipende dal suo stato e il comportamento cambia a run-time in dipendenza del suo stato.
- Le operazioni da svolgere hanno rami condizionali e frammenti di codice che dipendono dallo stato.
- Lo stato è spesso rappresentato dal valore di una variabile.
- La soluzione inserisce ciascun ramo condizionale in una classe separata.

Soluzione:
- Inserire ogni ramo condizionale in una classe separata.
- Context costituisce l'interfaccia per i client, e mantiene un'istanza di un ConcreteState che definisce lo stato corrente.
- State definisce un'interfaccia che incapsula il comportamento associato con un particolare stato.
- ConcreteState implementa il comportamento associato con uno stato.

Conseguenze:
- Il comportamento associato ad uno stato è in un a sola classe (ConcreteState).
- La logica che gestisce il cambiamento di stato è separata da vari comportamenti ed'è in una sola classe (Context), anziché sulal classe che implementa i comportamenti.

- La separazione suddetta aiuta ad evitare stati inconsistenti poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante.
- Il numero di classi totale è maggiore, le classi sono più semplici.



Decorator:

Intento:

- Aggiungere responsabilità ad un oggetto dinamicamente. Fornire un'alternativa flessibile all'implementazione di sottoclassi per estendere funzionalità.

Motivazione:

- Si vogliono aggiungere responsabilità ad un oggetto, non all'intera classe.
- L'aggiunta di responsabilità è trasparente ovvero i client non devono cambiare.
- Le responsabilità possono essere sottratte dinamicamente.
- I Decorator possono essere annidati ricorsivamente, per aggiungere più 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.

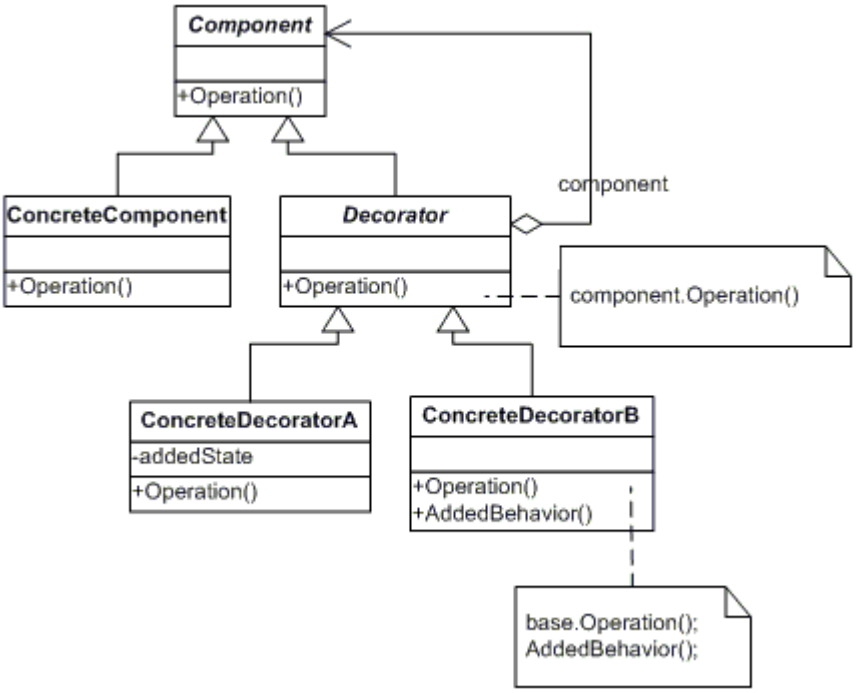
Soluzione:

- Component definisce l'interfaccia per gli oggetti che possono avere aggiunte le responsabilità dinamicamente.
- ConcreteComponent definisce un oggetto su cui poter aggiungere responsabilità.
- Decorator mantiene un riferimento all'oggetto Component e definisce un'interfaccia conforme a quella di Component.
  Decorator inoltra le richieste al suo oggetto Component e può fare altre operazioni prima e dopo l'inoltro della richiesta.
- ConcreteDecorator implementa la responsabilità aggiunta al ConcreteComponent.

Conseguenze:

- Più flessibilità rispetto all'ereditarietà poiché si possono aggiungere responsabilità dinamicamente.
- La stessa responsabilità può essere aggiunta più volte, semplicemente aggiungendo due istanze dello stesso ConcreteDecorator.
- Prevedere per le classi in alto nella gerarchia quali responsabilità servono significherebbe avere per esse troppe responsabilità. I ConcreteDecorator sono invece indipendenti e definibili successivamente.

- L'identità (tipo e riferimento) del ConcreteDecorator non è quella del ConcreteComponent, quindi non si dovrebbero confrontare i riferimenti.
- Si avranno tanti piccoli oggetti che differiscono nel modo in cui sono interconnessi.



Evoluzioni metriche:

Manutenzione:

Il processo di introduzione di modifiche ad un prodotto software dopo la sua consegna al cliente.

Evoluzione:

- Spesso usato con lo stesso significato di manutenzione.
- Singolo passo di u 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 uovo 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.

Dinamiche di evoluzione:

- Sono i processi di cambiamento di un sistema.
- Si basano su studi empirici.

Leggi di Lehman:

Cambiamento continuo:

I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti.

Aumento della complessità:

Quando un sistema evolve, la sua struttura aumenta di complessità, a meno che del lavoro viene fatto per preservare o semplificare la sua struttura.

Auto-regolazione:

Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti.

Stabilità organizzativa:

Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo.

Conservazione di familiarità:

In media, l'incremento di crescita di un sistema tende a rimanere costante o a diminuire.

Continua crescita:

Il contenuto di funzioni di un sistema deve continuamente essere incrementato per mantenere la soddisfazione dell'utente.

Diminuzione della qualità:

La qualità di un sistema diminuisce se non viene rigorosamente gestita ed adattata durante i cambiamenti.

Applicabilità delle 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, piccole organizzazioni.

Costo di manutenzione:

Fattori che contribuiscono al costo sono:

- Il costo è ridotto se il tema 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 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:

- Re-factoring o re-structuring
  Processo di cambiamento del software che non altera il comportamento del codice ma migliora la struttura interna.
- Reverse engineering
  Analizzare un sistema per estrarre informazioni sul suo comportamento o sulla sua struttura.
- Re-engineering
  Alterare un sistema per ricostruirlo in un'altra forma.

Metriche:

Una metrica definisce un set di misure per un sistema software.
Obiettivi dell'adozione di metriche

- 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.