Ingegneria del software → approccio sistematico allo sviluppo, all'operatività, alla manutenzione ed al ritiro del software. L'ingegneria del software è la disciplina tecnologica e manageriale che riguarda la produzione sistematica e la manutenzione dei prodotti software che vengono sviluppati e modificati entro tempi e costi preventivati.
Class diagram → diagramma che contiene classi e relazioni tra le classi.
Associazione → relazione tra oggetti/classi; ogni lato ha un nome, una cardinalità e può essere navigabile o no.
Sequence diagram → mette in evidenza la successione degli eventi nel tempo.
Communication diagram → mette in evidenza le relazioni tra gli oggetti.
Processo → definisce chi fa cosa, quando la fa e come raggiungere un certo risultato.
Processo di sviluppo software → insieme strutturato di attività richieste per sviluppare un sistema software. Il processo software è un insieme di attività che porta alla creazione di un prodotto software. E' un processo intellettuale e creativo.

Modello a cascata:

1) **Analisi e definizione dei requisiti** → vengono definiti, assieme agli utenti, i servizi che il sistema dovrà fornire, i vincoli da rispettare e gli obiettivi da raggiungere, in un modo comprensibile per utenti e sviluppatori. Fase suddivisa in:
    1. Studio di fattibilità.
    2. Analisi dei requisiti.
    3. Definizione dei requisiti.
    4. Specifica dei requisiti.
2) **Progettazione del sistema e del software** → vengono definiti uno o più modelli del sistema, che lo rappresentano a diversi livelli di dettaglio. Fase suddivisa in:
    1. Progettazione del sistema: identifica i componenti hardware e software del sistema.
    2. Progettazione del software: in modo che possa essere trasformato in uno o più programmi eseguibili.
3) **Programmazione e test dei moduli** → codifica e test dei vari moduli. Fase divisa in:
    1. Realizzazione dei componenti software, scrivendo il codice o copiandolo da altre fonti.
    2. Test dei singoli componenti del software.
4) **Integrazione e test del sistema** → integrazione dei moduli del programma e test del corretto funzionamento dell'intero sistema. Fase divisa in:
    1. Integrazione dei componenti.
    2. Test d'integrazione.
    3. Test dei requisiti del sistema.
    4. Consegna del sistema al cliente.
5) **Installazione e manutenzione** → messa in funzione del sistema e assistenza ai clienti. Fase divisa in:
    1. Installazione → il sistema è messo in funzione.
    2. Manutenzione → identificazione e correzione di problemi o errori.
    3. Evoluzione → il sistema evolve nel tempo a causa del cambiamento dei requisiti, dell'aggiunta di nuove funzioni o del cambiamento dell'ambiente tecnico.
    4. Uscita di scena → il sistema è rimosso dal servizio.

Sviluppo software Agile → struttura concettuale per l'ingegneria del software che promuove lo sviluppo di iterazioni nel ciclo di vita del progetto. Privilegia la comunicazione faccia a faccia a quella scritta e la collaborazione tra le persone dei vari team di sviluppo. Ogni 1-4 settimane un team del progetto finisce un modulo (iterazione) del progetto, sviluppato seguendo l'iter classico.
L'obiettivo di ogni iterazione è avere una release funzionante, anche se incompleta.
Sviluppo iterativo → ha come obiettivo ottenere qualcosa di funzionante nel minor tempo possibile. L'implementazione iniziale è perfezionata finché il sistema non è completo. Le tecniche sono:
✔Vaporware.
✔Throw-away software component.
✔Dummy modules.
✔Rapid prototyping.
✔Incremental refinement.
Extreme programming (programmazione estrema) → è forse il metodo più utilizzato: i vari scenari sono rappresentati da storie degli utenti implementate come serie di azioni (task). I programmatori sviluppano test per ogni task prima di scriverne il codice e tutti i test vengono eseguiti quando del nuovo codice è integrato nel sistema. E' basato su quattro attività:
1. Ascolto.
2. Progettazione.
3. Codifica.
4. Test.

Processo unificato → struttura estensibile che può essere personalizzata per organizzazioni o progetti specifici. Viene guidata dai casi d'uso (use cases) e si basa su una preventiva identificazione e gestione dei rischi.
Sviluppo guidato dal modello → si riferisce all'uso sistematico di modelli come artefatti ingegneristici primari per tutta la durata della progettazione. Si basa sulla trasformazione di un modello in un programma eseguibile.
Architettura guidata dal modello → approccio allo sviluppo del software che fornisce un insieme di linee guida per strutturare le specifiche espresse come modelli per la realizzazione del sistema per diverse piattaforme tecnologiche.

UML → Unified Modeling Language, linguaggio standard per la visualizzazione, la specifica, la costruzione e la documentazione degli artefatti di un sistema software complesso. Fornisce diagrammi multipli che consentono di visualizzare diversi tipi di dettaglio dell'architettura. E':

✔Semplice, perché richiede solo pochi concetti e simboli.
✔Espressivo, in quanto è applicabile a un largo spettro di sistemi e metodi.
✔Utile, perché si focalizza solo sugli elementi necessari all'ingegneria del software.
✔Consistente, perché lo stesso concetto/simbolo deve essere applicato allo stesso modo.
✔Estensibile, perché gli utenti e gli sviluppatori hanno una certa libertà di estendere la notazione.

Le specifiche dell'UML sono:

✔Infrastruttura → costrutti linguistici fondamentali.
✔Superstruttura → costrutti a livello utente.
✔Object Constraint Language (OCL) → linguaggio formale usato per descrivere le espressioni nei modelli UML.
✔Interscambio di diagrammi → viene permesso uno scambio di documenti.

Diagrammi UML → permettono di analizzare il sistema da diversi punti di vista:

1. **Vista dei casi d'uso (Use Case View)** → la più importante, descrive casi d'uso che hanno valore per gli utenti. É rappresentata da:
   ✔ ***Use case diagram*** → uno use case rappresenta un'unita d'interazione discreta tra un utente (umano o macchina) e il sistema. L'interazione è una singola unità di lavoro utile, che può includere una complessa interazione tra le parti. Il diagramma è composto dal sistema (rettangolo), attori, use case (racchiusi in ovali) e relazioni tra attori e use cases, con la loro molteplicità.
2. **Vista strutturale (Structural View)** → rappresentata dai diagrammi statici:
   ✔ ***Class diagram*** → mostra i blocchi (classi) che costruiscono un sistema orientato agli oggetti. É composto da classi (rettangoli) o interfacce (<<interface>> prima del nome), suddivise in compartimenti: nome della classe, attributi della classe (+ pubblici, -privati) nella forma "<+/-> <nome> : <tipo>", e dai collegamenti tra le classi, che possono essere:
   ▪ *Associazioni* → rappresentate da un connettore con nome/ruolo, cardinalità, direzioni e vincoli.
   ▪ *Generalizzazione* → usata per l'ereditarietà, ha una freccia bianca verso la classe padre.
   ▪ *Nidificazione* → per le classi interne, ha un simbolo di dalla parte dell'ospitante.
   ▪ *Dipendenza* → linea tratteggiata.
   ▪ *Realizzazione* → linea tratteggiata con una freccia bianca.
   ▪ *Aggregazione* → usata per rappresentare elementi formati da piccoli componenti, ha un simbolo verso la classe genitore.
   ▪ *Composizione* → è una forma forte di aggregazione usata quando i componenti possono essere inclusi in un massimo di una composizione alla volta, ha come simbolo verso la classe genitore.
   ✔ ***Object diagram*** → descrive la struttura statica di un sistema in un particolare momento e può essere considerato un caso particolare del class diagram. Gli oggetti sono rappresentati da rettangoli con intestazione "Object :Class".
   ✔ ***Composite structural diagram*** → mostra la struttura interna di un classificatore (elemento UML descritto da attributi e metodi: classe, interfaccia o componente), inclusi i suoi punti di interazione con altre parti del sistema. Simile a un class diagram, ma mostra le parti interne al posto della classe generale. É composto da classi (rettangoli), parti di una classe, ovvero ruoli giocati da un'istanza di una classe in fase di esecuzione (rettangoli), porte, ovvero punti di interazione per collegare parti tra loro e con l'ambiente esterno (quadratini), interfacce previste (linee con cerchio verso l'esterno) e interfacce richieste (linee con mezzaluna verso l'esterno).
   ✔ ***Package diagram*** → decompone il sistema in unità di lavoro logiche che descrivono le dipendenze tra loro, fornisce una vistadel sistema da diversi livelli di astrazione.
3. **Vista di comportamento (Behavioral View)** → rappresenta le interazioni dinamiche tra i componenti del sistema per implementare i requisiti. Mostra come sono distribuiti i compiti. É rappresentata dai diagrammi dinamici:
   ✔ ***Sequence diagram*** → descrive come viene portato avanti un processo con una sequenza di interazioni tra oggetti. É rappresentato da: classi/oggetti (rettangoli), linee di vita (tratteggiate, indicano l'esistenza di un oggetto), attivazioni (linee tra linee di vita con il nome dell'operazione svolta e un verso).
   ✔ ***Communication diagram*** → mostra informazioni simili al sequence diagram ma si focalizza sulle relazioni tra gli oggetti. É rappresentato da: classi/oggetti (rettangoli), relazioni (linee di collegamento con nome dell'azione svolta).
   ✔ ***Robustness diagram*** → communication diagram semplificato, ha come obiettivo dare un significato raffinando i casi d'uso. É rappresentato da cerchi: elementi di controllo, interfacce e entità.
   ✔ ***Activity diagram*** → mostra la sequenza delle attività da un punto iniziale a uno finale, dettagliando tutti i cammini decisionali esistenti nella progressione degli eventi contenuti nell'attività. É composto da attività (rettangoli con angoli arrotondati), azioni (singolo passo dell'attività, rappresentate dentro all'attività con un rettangolo con angoli arrotondati), vincoli (collegati con frecce tratteggiate alle azioni, racchiusi in "fogli"), frecce per collegare le azioni, nodo iniziale , nodo di fine attività , nodo di fine flusso , gestori delle eccezioni, collegati con una freccia a fulmine, decisioni .
   ✔ ***State diagram*** → modellizza il comportamento di un singolo oggetto, ovvero specifica la sequenza di stati che un oggetto attraversa durante il suo ciclo di vita in risposta a stimoli dell'ambiente. É composto da stati (rettangoli arrotondati), transizioni da uno stato all'altro (frecce), punto iniziale (pallino nero) e punto finale (cerchio con punto al centro), scelte (rombo), giunzioni (pallino nero a cui arrivano e partono tante frecce).
   ✔ ***Interaction overview diagram*** → forma di activity diagram in cui i nodi rappresentano interaction diagrams.
   ✔ ***Timing diagram*** → usato per visualizzare il cambiamento di stato o di valore di uno o più elementi nel tempo. Può anche mostrare l'interazione tra eventi programmati e i vincoli temporali e di durata che li governano.
4. **Vista di implementazione (Implementation View)** → descrive l'implementazione dei componenti del sistema definiti nella vista strutturale. É rappresentata dai diagrammi:
   ✔ ***Component diagram*** → illustra le parti di software che formano il sistema. Ha un livello di astrazione più alto di un class diagram.
   ✔ ***Composite structural diagram***
5. **Vista d'ambiente (Environment View)** → rappresenta la topologia hardware del sistema. Fornisce informazioni per l'installazione e la configurazione del sistema. É rappresentata da:
   ✔ ***Deployment diagram*** → modellizza l'architettura d'esecuzione di un sistema. Descrive la configurazione hardware in un sistema in termini di nodi e connessioni, la relazione fisica tra hardware e software e visualizza e mostra come gli artefatti sono stati installati e si muovono nel sistema.

OCL → Object Constraint Language, è un linguaggio che permette di descrivere le espressioni e i vincoli in un sistema orientato agli oggetti. É un linguaggio formale tipizzato con una precisa sintassi e semantica. Non è un linguaggio di programmazione. Permette di descrivere il sistema in modo approfondito con parole semplici, a differenza dell'UML che aumenta la complessità del linguaggio man mano che aumenta il livello di dettaglio.
Vincolo → restrizione su uno o più valori di un modello/sistema orientato agli oggetti.
Invariante → vincolo connesso a un elemento del modello. Deve essere vero per tutto il tempo in cui l'istanza è a riposo, ovvero quando nessuna operazione è eseguita su di essa.
Precondizione → vincolo che deve essere vero quando è invocata un'operazione.
Postcondizione → vincolo che deve essere vero dopo il completamento di un'operazione.
Guardia → vincolo che deve essere vero prima dell'avvenimento di una transizione.
Collezione → tipo astratto predefinito di OCL. Diviso in quattro sottotipi:

- ✔***Insieme (Set)*** → non contiene duplicati.
- ✔***Insieme ordinato (OrderedSet)*** → insieme con elementi ordinati.
- ✔***Borsa (Bag)*** → può contenere duplicati.
- ✔***Sequenza (Sequence)*** → borsa con elementi ordinati.

Requisito → espressione astratta di alto livello di un vincolo di un servizio/sistema o specifica matematica funzionale dettagliata. Rappresenta il comportamento del software che soddisfa le necessità del cliente.
Ingegneria dei requisiti → processo per stabilire i servizi che il cliente richiede al sistema, i vincoli sotto i quali il sistema deve operare ed essere sviluppato. Ha come obiettivo raccogliere e analizzare le descrizioni dei servizi e dei vincoli di un sistema.
Classificazione dei requisiti → i requisiti si possono dividere in varie categorie:

- ✔***Requisiti funzionali*** → rappresentano il comportamento atteso dal sistema. Descrivono come il sistema deve reagire a particolari dati di ingresso e come il sistema deve comportarsi in certe situazioni.
- ✔***Requisiti non funzionali*** → vincoli sui servizi o sulle funzioni offerte dal sistema. Specificano i criteri che possono essere usati per valutare l'operato del sistema, invece che specifici comportamenti. Vincolano la progettazione e l'implementazione del sistema, ma non descrivono un servizio che il sistema deve fornire. Sono suddivisi in tre sottotipi:
  - ✔*Requisiti produttivi* → vincoli sulla catena produttiva, su come il prodotto dovrà essere(velocità di esecuzione, affidabilità, efficienza, portabilità...).
  - ✔ *Requisiti organizzativi* → vincoli che sono una conseguenza delle politiche e delle procedure organizzative dell'azienda (requisiti di consegna, implementazione, standard).
  - ✔*Requisiti esterni* → requisiti che derivano da fattori esterni al sistema e al suo processo di sviluppo (requisiti etici, legislativi, di interoperabilità...).
- ✔ ***Requisiti di dominio*** → requisiti derivanti dal dominio di applicazione del sistema e che riflettono le caratteristiche di quel dominio.
- ✔***Requisiti durevoli*** → requisiti stabili derivati dall'attività principale dell'azienda del cliente.
- ✔ ***Requisiti volatili*** → requisiti che cambiano durante lo sviluppo o quando il sistema è in funzione. Possono essere divisi in quattro sottotipi:
  - ✔*Requisiti mutabili* → requisiti che cambiano a causa del cambiamento dell'ambiente in cui l'azienda opera.
  - ✔*Requisiti emergenti* → requisiti che emergono dallo sviluppo della comprensione del sistema da parte del cliente nel corso dello sviluppo del sistema.
  - ✔*Requisiti consequenziali* → requisiti derivanti dall'introduzione di un computer.
  - ✔*Requisiti di compatibilità* → requisiti che dipendono dal particolare sistema o processo aziendale dell'organizzazione.
- ✔***Requisiti dell'utente*** → descrizione dei servizi del sistema e dei suoi vincoli operazionali comprensibile dagli utenti del sistema che non hanno conoscenza tecnica. Sono definiti usando linguaggio naturale, tabelle e grafici.
- ✔***Requisiti del sistema*** → specifica dei requisiti degli utenti più dettagliata, rivolta al cliente e agli sviluppatori.

Documento dei requisiti → documento che specifica chiaramente i requisiti del sistema ottenuti durante il processo dei requisiti. I requisiti devono essere:

- ✔***Completi*** → devono includere la descrizione di tutte le funzioni richieste.
- ✔***Consistenti*** → non devono esserci conflitti o contraddizioni nella descrizione delle funzioni.
- ✔***Comprensibili*** → descritti in un linguaggio comprensibile ai più (spesso impossibile).
- ✔***Espliciti*** → non devono dare nulla per scontato.

Problemi del linguaggio naturale → mancanza di chiarezza, confusione nel descrivere i requisiti, ambiguo, troppo flessibile, inadatto alla struttura dei requisiti del sistema. Soluzione: invento un linguaggio standard, utilizzo parole chiave per vincoli, azioni dell'utente, ambiente, cambi di rischio.

Studio di fattibilità → decide se il sistema proposto è realizzabile o no. Controlla che il sistema contribuisca agli obiettivi aziendali, sia sviluppabile usando la tecnologia corrente e il budget assegnato e che possa essere integrato con altri sistemi utilizzati.

Processo di ingegnerizzazione dei requisiti → viene diviso in quattro parti:

- ✔ ***Elicitazione*** → identifica le sorgenti più rilevanti di requisiti, determina quali informazioni sono necessarie, le elabora e le sintetizza in una frase appropriata. Permette al cliente di capire meglio di quali requisiti ha bisogno e allo sviluppatore di affrontare il problema corretto. Le tecniche di elicitazione sono quattro:
  - ✔ *Campionamento* → raccoglie esempi di documentazione, registrazioni, moduli e li sceglie o a caso o con un criterio ben preciso (stratificazione).
  - ✔ *Osservazione dell'ambiente di lavoro* → osserva le persone al lavoro per capire come funziona il sistema. Occorre essere cauti e cercare di non farsi notare (le persone non è detto che lavorino allo stesso modo se osservate).
  - ✔ *Questionario* → documento che permette all'analista di raccogliere informazioni e opinioni dai partecipanti. Può essere di due tipi:
    - ✔ *Free format* → domande a risposta aperta
    - ✔ *Fixed format* → domande a risposta multipla
  - ✔ *Intervista* → tecnica in cui l'analista colleziona informazioni da un colloquio individuale con i partecipanti. Può essere di due tipi:
    - ✔ *Non strutturata* → poche domande, si lascia all'intervistato la possibilità di dirigere la conversazione.
    - ✔ *Strutturata* → insieme di domande precise da porre all'intervistato.

  Le domande possono essere di due tipi:
  - ✔ Domande aperte → permettono all'intervistato di rispondere nella maniera che gli sembra più appropriata.
  - ✔ Domande chiuse → necessitano di risposte specifiche, brevi e dirette.
- ✔ ***Analisi*** → raffina e struttura i requisiti per facilitarne la comprensione, il riuso e la manutenzione. Ha come obiettivo il ***modello di analisi*** ovvero un'astrazione che illustra: le funzioni da realizzare nel sistema (caso strutturale), i casi d'uso e la rappresentazione degli oggetti e dei concetti reali del dominio del sistema (caso orientato agli oggetti, formato da classi d'analisi, astratte, che gestiscono i requisiti funzionali). Le tecniche per individuare le classi di analisi sono cinque:
  - ✔ *Analisi nome-verbo* → individua nei nomi classi, attributi e istanze, e nei verbi operazioni, relazioni e vincoli.
  - ✔ *Guidato dagli use case* → centrato sui casi d'uso, analizza i diversi scenari descritti dal caso d'uso e individua le classi da questi scenari.
  - ✔ *Common class patterns* → deriva le classi dalla generica teoria di classificazione degli oggetti.
  - ✔ *Carte CRC* → basata su nome, responsabilità e collaboratori della classe e su una sessione di brainstorming animato.
  - ✔ *Approccio misto*.
- ✔ ***Validazione*** → processo per stabilire e giustificare la credenza che i requisiti documentati corrispondono alle volontà del cliente, degli utenti e di altre parti interessate. Si può fare con tre tecniche:
  - ✔ *Ricontrollo dei requisiti* → ricontrolli regolari dovrebbero essere svolti durante la definizione dei requisiti; coinvolge sia il cliente che lo sviluppatore.
  - ✔ *Prototipazione* → utilizza dei prototipi (modelli eseguibili) del sistema per verificare i requisiti.
  - ✔ *Generazione di test-case* → sviluppo di test per ogni requisito che ne controllano la verificabilità.
- ✔ ***Gestione*** → processo di gestione del cambiamento dei requisiti durante il processo di ingegnerizzazione dei requisiti e lo sviluppo del sistema. Questa fase è svolta in tre passi:
  - ✔ *Analisi del problema* → discute i problemi dei requisiti e propone cambiamenti.
  - ✔ *Cambiamento di analisi e costi* → valuta gli effetti dei cambiamenti sugli altri requisiti.
  - ✔ *Cambiamento dell'implementazione* → modifica il documento dei requisiti e altri documenti per applicare il cambiamento.

Use case → descrizione di sequenze di eventi che, presi assieme, contribuiscono a far fare qualcosa di utile al sistema. Descrive l'interazione tra un attore primario, l'iniziatore dell'interazione, e il sistema stesso, attraverso una sequenza di semplici passi. Dovrebbe descrivere tutte le possibili interazioni con il sistema.

Rischi dell'ingegneria dei requisiti:

- ✔ Incomprensione del dominio o del vero problema.
- ✔ Cambiamento rapido dei requisiti.
- ✔ Tentativo di fare troppo.
- ✔ Difficoltà di risolvere i conflitti tra requisiti.
- ✔ Difficoltà di identificare precisamente lo stato dei requisiti.

Design Patterns

Progettare per il riuso

Un sistema di classi è ben progettato se
- le classi esprimono concetti ben individuabili,
- gli stati descritti delle classi sono semplici,
- i metodi delle classi sono pochi e il loro scopo è ben preciso (ma generale),
- sono minimizzate associazioni e dipendenze,
- le strutture dati sono tenute separate dagli algoritmi,
- gli algoritmi non dipendono da come le strutture dati sono memorizzate,
- …

Per una migliore progettazione conviene sfruttare dei design pattern.
Nella progettazione si incontrano problemi ricorrenti.
È utile trovare soluzioni riusabili per trattare problemi ricorrenti
- design pattern.

Sono soluzioni utilizzate in progetti reali per risolvere problemi comuni.
Sono indipendenti dal linguaggio di programmazione.
Vengono raggruppati in pattern language
    - raccolte divise secondo categorie di problemi affrontati.

Variano da semplici soluzioni programmative a complesse architetture di sistema.
Se utilizzati durante la fase di sviluppo, vengono detti idiomi programmativi.

Caratteristiche

I design pattern consentono di
- definire un vocabolario comune,
- comunicare concetti complessi in modo semplice,
- documentare i progetti,
- catturare parti essenziali di un progetto in una forma compatta,
- descrivere astrazioni.

I design pattern non offrono
- soluzioni esatte e complete,
- soluzioni a tutti i problemi di progettazione.

Formato GoF di un Design Pattern

Nome e tipo
Intento
- cosa fa il pattern e quando la soluzione è applicabile e può portare benefici.

AKA (Also Known As)
- altri nomi in uso per il pattern.

Motivazione
- problemi che sono stati risolti mediante l'uso del pattern

Applicabilità
- situazioni dove il pattern può essere applicato e può portare beneficio

Struttura
- descrizione della soluzione,
- indica i partecipanti e le collaborazioni.

Conseguenze
- trade- off e questioni che nascono dall'impiego del pattern.

Implementazione
- suggerimenti e tecniche utili per implementare il pattern.

Codice d'esempio.
Usi noti
- sistemi reali in cui il pattern è stato utilizzato con successo.

Pattern correlati.

I pattern più noti sono i GoF
I pattern GoF sono classificati in
- creazionali, gestiscono la creazione dinamica degli oggetti all'interno di un sistema;
- strutturali, micro-architetture statiche di utilizzo generale;
- comportamentali, descrivono il comportamento di un'architettura di oggetti.

Pattern creazionali

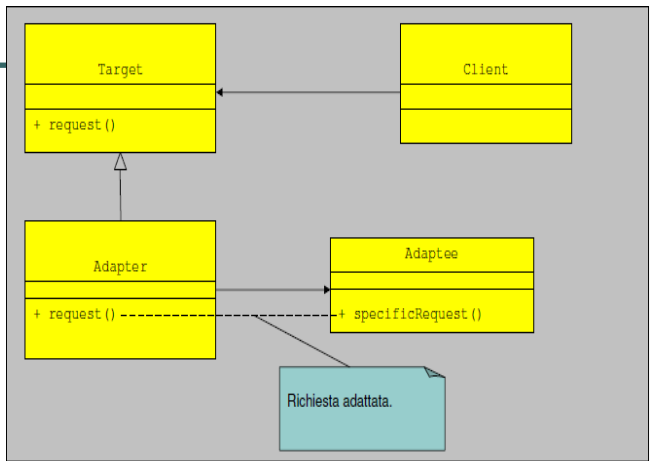
Abstract factory, utilizzata per creare oggetti senza conoscerne l'implementazione concreta.
Builder, usato per creare oggetti complessi indipendentemente dalla loro rappresentazione interna.
Prototype, usato per creare oggetti partendo da un'istanza prototipo.
Singleton, usato per garantire che una sola istanza di un oggetto venga creata nel sistema.

Pattern strutturali

Gestiscono il modo in cui classi e oggetti vengono composti per formare architetture complesse.
Adapter, usato per risolvere problemi di incompatibilità tra interfacce.
Bridge, usato per fornire più implementazioni ad un'interfaccia.
Composite, usato per trattare in modo uniforme oggetti singoli e gruppi di oggetti.
Façade, usato per fornire un'interfaccia unificata ad un gruppo di oggetti.
Proxy, usato per fornire funzionalità aggiuntive ad un oggetto.

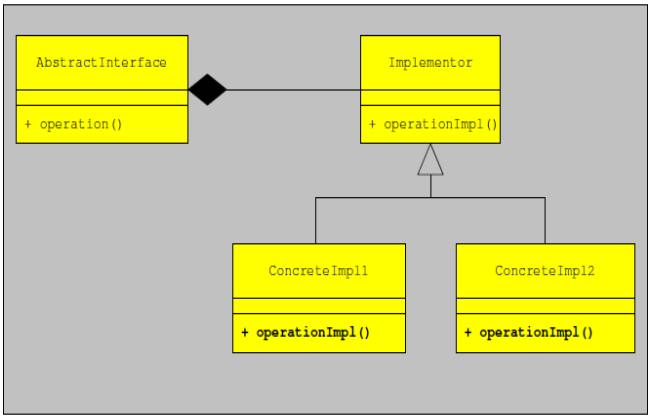
Adapter (1/2)

- Consente ad un oggetto di essere utilizzato mediante un'interfaccia che non implementa.
- L'adapter è uno dei pattern più usati
  - consente a oggetti diversi di essere utilizzati insieme anche se le rispettive interfacce non sono compatibili,
  - aumenta il riuso delle classi perché consente di utilizzarle in situazioni non previste inizialmente senza doverle modificare.
- Un adapter viene usato
  - quando si vuole utilizzare una classe che non offre un'interfaccia corretta, senza modificarla,
  - per creare una classe che utilizza i servizi di un'altra classe di cui non è ancora nota l'interfaccia.



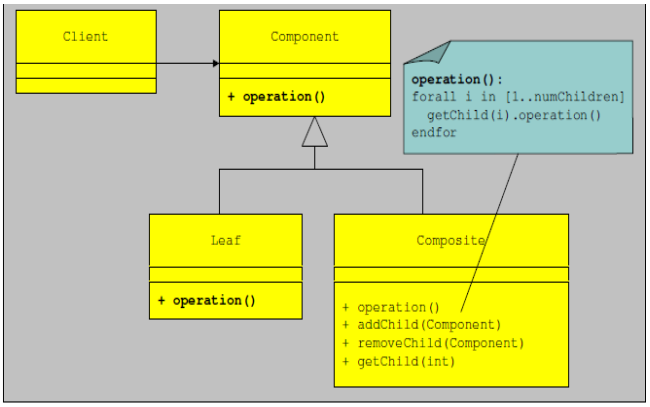
Bridge (1/2)

- Disaccoppia un'astrazione dalla sua implementazione
  - la classe di implementazione e la classe degli utilizzatori possono variare indipendentemente.
- Un bridge viene usato quando
  - si vuole evitare di legare un'astrazione ad una sola implementazione,
  - ogni cambiamento di un'implementazione non è influente sugli utilizzatori,
  - per nascondere dettagli implementativi.



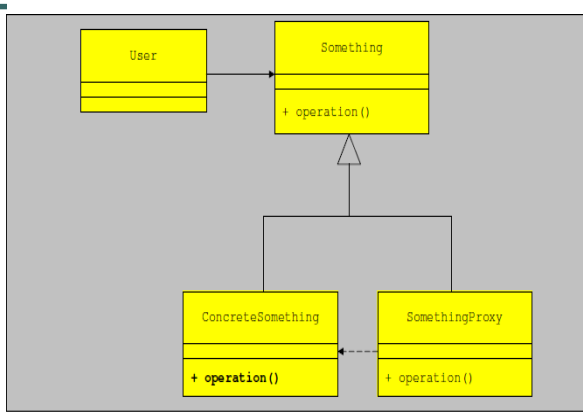
Composite (1/2)

- Consente ad oggetti utilizzatori di creare strutture ad albero in modo che sia possibile trattare in modo uniforme
  - oggetti foglia,
  - oggetti contenitore.
- Un composite viene usato quando
  - si vuole rappresentare gerarchie di oggetti,
  - si vuole consentire di ignorare le differenze tra oggetti foglia e oggetti contenitore.



Proxy (1/2)

- Un proxy è un delegato di un altro oggetto.
- I proxy hanno diversi usi
  - per controllare l'accesso alle risorse
    - per implementare politiche di sicurezza,
  - per implementare una gestione soffisticata di un oggetto
    - accesso sincronizzato,
    - persistenza,
  - per implementare oggetti remoti
    - il proxy e l'oggetto reale sono su due host diversi.

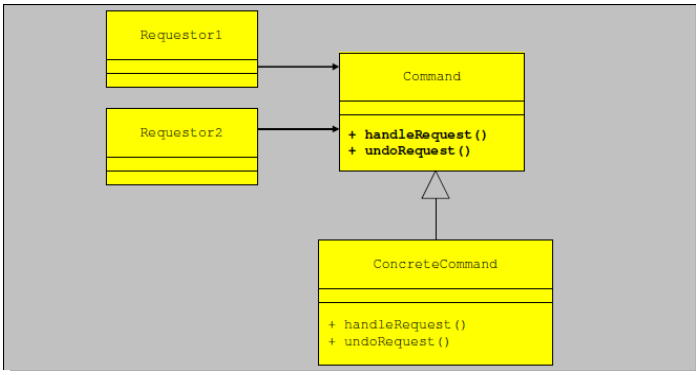


Pattern Comportamentali

- I pattern comportamentali definiscono collaborazioni tra oggetti
  - essenzialmente definiscono comunicazioni tra oggetti.
- Command, usato per incapsulare l'astrazione di richiesta.
- Iterator, usato per incapsulare la navigazione di un oggetto composto.

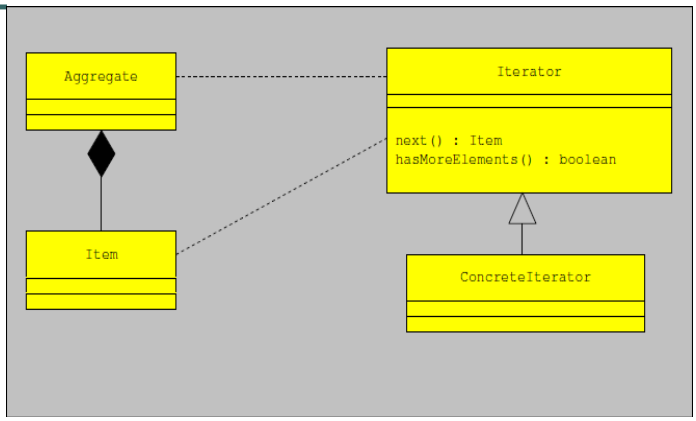
Command (1/2)

- Incapsula una richiesta in un oggetto
  - consente a molte sorgenti di generare richieste e le gestisce secondo una politica,
  - consente ad un oggetto di modificare il proprio comportamente in base alle richieste.
- Un command viene usato quando:
  - si vuole consentire di creare una richiesta in molti modi diversi,
  - si vuole consentire di fare undo delle richieste.



Iterator (1/2)

- Implementa un meccanismo di accesso ad oggetti aggregati in modo sequenziale
  - un iteratore naviga sequenzialmente un oggetto aggregato.
- Un iteratore viene utilizzato
  - per accedere al contenuto di un oggetto aggregato in modo indipendente da come i componenti sono memorizzati nell'oggetto aggregato,
  - quando si vuole offrire diverse politiche di attraversamento di un oggetto aggregato.



JAVA e PROGRAMMAZIONE AD OGGETTI
Byte code → risultato di un compilatore Java, che non produce codice eseguibile da una particolare piattaforma, che viene poi trasformato in codice nativo dalla Java Virtual Machine.
Garbage collector → servizio offerto dalla JVM per distruggere gli oggetti inutilizzati e liberare memoria.
Information hiding → principio secondo il quale occorre nascondere i dettagli implementativi di un oggetto.
Incapsulamento → l'oggetto è gestibile come una scatola nera in cui l'attenzione è ai servizi che esso offre e non a come essi vengono implementati.
Dipendenza tra classi → quando l'esistenza di una classe A con certe caratteristiche è necessaria per l'esistenza di una classe B, si dice che B dipende da A.
Polimorfismo → un metodo della classe base può essere ridefinito nelle classi derivate.
Classe astratta → dichiara dei metodi senza implementarli (non può essere istanziata).
Interfaccia → insieme dei metodi pubblici di una classe senza la loro implementazione.
Package → insieme di classi coese e interdipendenti che sono logicamente viste come un unico gruppo.
Classe interna → classe la cui dichiarazione si trova all'interno di un'altra classe detta classe ospitante.
Classe locale → classe definita all'interno di un metodo.
Libreria → insieme di package che implementano funzionalità comuni.
Eccezione → evento verificato durante l'esecuzione di un programma che interrompe il suo normale flusso di istruzioni.
Eccezioni controllate (checked) → eventi eccezionali che una applicazione affidabile dovrebbe catturare (blocco "try...catch...finally...", oppure "throws..." nel metodo) e gestire.
Errori → eventi eccezionali esterni all'applicazione, che un'applicazione spesso non può prevedere o risolvere. Di solito, se avvengono, il programma termina.
Eccezioni non controllate (unchecked) → eventi eccezionali interni che un'applicazione dovrebbe prevenire.
ANT → interprete di comandi che viene utilizzato principalmente per automatizzare le operazioni (compilazione, creazione file .jar, esecuzione...) in ambiente Java. Legge un file .xml per capire cosa fare.