

Ingegneria del Software

Introduzione

L'*astrazione* è il processo di formulare idee o concetti generalizzati, estraendo qualità comuni da esempi specifici.

Un *modello* è una rappresentazione del sistema in analisi che risponde come tale ad un dato insieme di quesiti. Tutti i modelli sono riflessioni semplificate della realtà, ma, nonostante la loro intrinseca falsità, sono tuttavia estremamente utili.

Software Process Model

Un processo è un insieme di attività coordinate che portano ad un obiettivo.

Le *attività* di un processo software sono ad esempio la specificazione, il design, l'implementazione, la validazione e l'evoluzione.

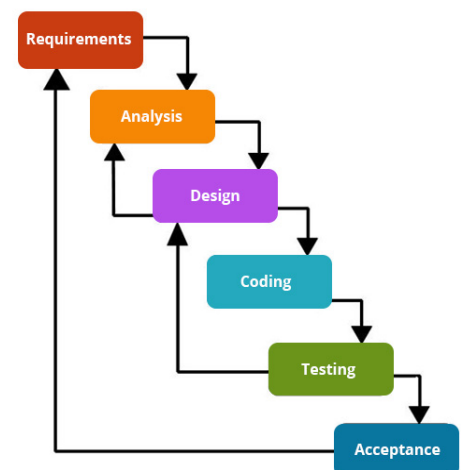
L'*obiettivo* è pianificare, organizzare ed eseguire un progetto software considerando certi vincoli, come costo, tempo e risorse (ottimizzando progresso, qualità e rischio).

Il modello a cascata

Questo modello applica concetti provenienti da altri campi dell'ingegneria, alla produzione di software, è essenzialmente composto da rigide attività sequenziali.

I pro sono che identifica risultati, milestones e artefatti (principalmente documentazione).

I contro sono che è irrealistico, un processo molto lungo e ha una gestione del rischio inefficace. È basato sul presupposto erroneo che le specifiche siano prevedibili e stabili e che vi sia un basso tasso nei cambiamenti. Ha infatti una percentuale elevata di fallimenti.



Il modello a spirale

Questo modello ha un approccio basato sul rischio (ha l'obiettivo infatti di ridurre i rischi presenti). Ogni ciclo inizia con obiettivi, alternative e vincoli, il

passo successivo è definito in base ai rischi rimanenti, e ogni ciclo termina con una revisione da parte degli stakeholders.

I pro sono la valutazione del rischio e la natura iterativa dello sviluppo del software.

I contro sono che l'analisi del rischio non è semplice e potrebbe ritardare la consegna.



Sviluppo iterativo ed evolutivo

Pratica fondamentale di molti processi software, organizzato in una serie di mini-progetti brevi (iterazioni). Il sistema cresce in modo incrementale nel tempo, iterazione dopo iterazione. Si basa su un atteggiamento di accettazione del cambiamento e sull'adattamento. In questo modo, attraverso il feedback iterativo e l'adattamento, il sistema sviluppato evolve e converge verso i requisiti corretti.

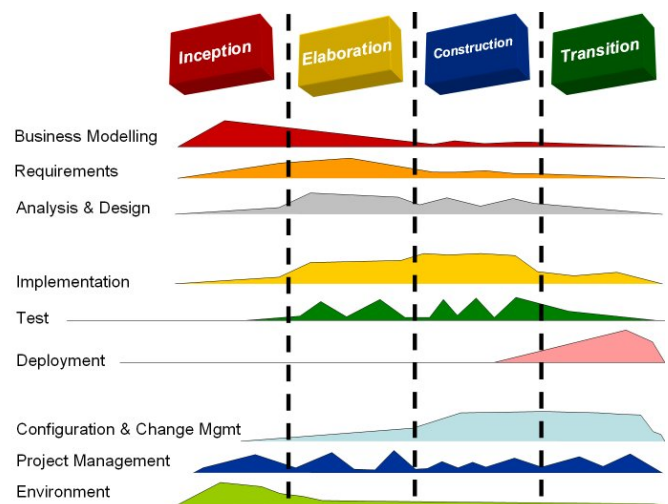
I pro sono la minore probabilità di fallimento, migliore produttività, riduzione precoce dei rischi maggiori e progresso visibile fin dall'inizio.

Unified Process (UP)

Questo è un framework iterativo e incrementale per il processo di sviluppo software.

Unified Process combina cicli di vita iterativi e sviluppo basato sul rischio, in una descrizione coesa e ben documentata.

Questo processo divide il progetto in quattro fasi: inizio, elaborazione, costruzione e transizione.



> Inception: gli obiettivi sono definire i rischi, i casi d'uso e le architetture candidate. Termina con il Lifecycle Objective Milestone.

> Elaborazione: gli obiettivi sono affrontare i rischi e convalidare l'architettura (implementando una linea di base dell'architettura eseguibile). Termina con un piano per la fase di costruzione (con costi e tempistiche).

> Construction: l'obiettivo è implementare le caratteristiche del sistema, utilizzando iterazioni time-boxed e producendo una release (raffinamento incrementale).

> Transition: gli obiettivi sono distribuire il sistema, coinvolgere gli utenti e raccogliere feedback.

Analysis Model

Gli obiettivi del modello di analisi sono la comprensione di cosa è richiesto nello specifico che faccia il software, comunicare queste direttive al team di sviluppo e agli stakeholders e definire un insieme di requisiti che saranno validati una volta che il software sarà costruito.

Gli artefatti del modello di analisi sono ad esempio il documento di specifica, i casi d'uso, il modello di dominio, il glossario, le business rules, ecc.

Una *storia utente* (user story) è una breve descrizione di una funzionalità o di un requisito di interesse per gli utenti del sistema.

Un *requisito* è una capacità o una condizione a cui il sistema deve essere conforme. Deriva da richieste degli utenti.

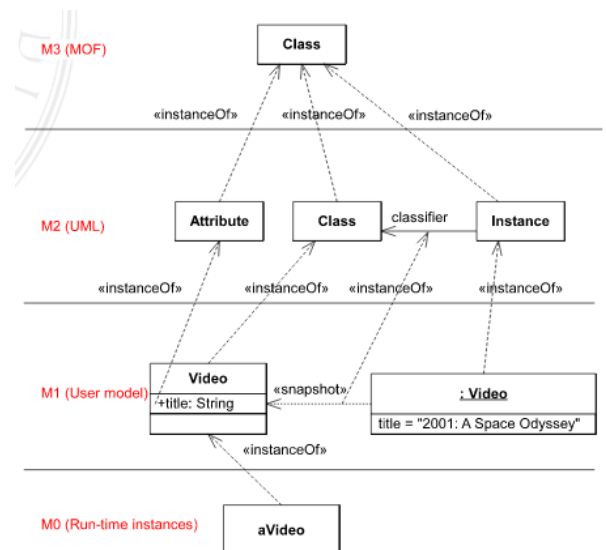
I requisiti possono essere di tipo *funzionale*, ovvero descrivono le interazioni tra il sistema e il suo ambiente, indipendentemente dall'implementazione (esprimono cosa dovrebbe fare il sistema); o *non-funzionale*, ovvero proprietà del sistema non direttamente collegate ad aspetti funzionali (descrivono come il sistema dovrebbe essere, o come dovrebbe fare ciò che fa).

Introduzione a UML

UML (Unified Modeling Language) è un linguaggio di modellazione object oriented per sistemi software intensivi.

O-O è un paradigma che sposta l'attenzione su analisi e progettazione da algoritmi e dati agli oggetti, intesi come entità autonome con uno stato e un comportamento. I principi cardine sono: astrazione, incapsulamento, ereditarietà e polimorfismo.

UML è basato su una modellazione standard OMG (Object Manager Group) chiamata MOF (Meta-Object Facility). Quest'ultima è strutturata in 4 livelli: M0, M1, M2 ed M3; i linguaggi basati su MOF possono essere serializzati come definito dallo standard XML (XML Metadata Interchange).



Un modello UML include *classificatori* (insiemi di cose), *eventi* (insiemi di occorrenze) e *comportamenti* (insiemi di esecuzioni).

Una *classe* è un classificatore che raggruppa oggetti con gli stessi attributi, operazioni, relazioni e semantiche.

Un *caso d'uso* è un classificatore che raggruppa interazioni con un risultato osservabile.

Una *collaborazione* è una collezione di ruoli che offrono un comportamento cooperativo che può realizzare un'operazione o un caso d'uso.

Un'*interfaccia* è una collezione di operazioni che può essere offerta o richiesta da altri elementi.

Un'*interazione* è un'unità di comportamento basata su uno scambio osservabile di messaggi.

Una *state machine* descrive il comportamento di un elemento di un modello come transizioni discrete tra un insieme finito di stati.

Un'*annotazione* migliora la leggibilità di un diagramma, non ha effetto sul modello.

Le relazioni in UML correlano due o più elementi in un modello, sono rappresentate come linee e possono avere nomi. I quattro tipi base sono: associazione, generalizzazione, dipendenza e realizzazione.

Un'*associazione* è una relazione strutturale tra due elementi di un modello che mostra che gli oggetti di un classificatore si connettono e possono attraversare gli oggetti di un altro classificatore [linea].

Una *generalizzazione* indica che un elemento specializzato del modello (figlio) è basato su un altro elemento del modello (genitore), esiste una relazione "è-un" tra i due elementi [linea con freccia chiusa da figlio a genitore].

Una *dipendenza* indica che i cambiamenti ad un elemento del modello (il fornitore o un elemento indipendente) possono causare cambiamenti in un altro elemento del modello [linea tratteggiata con freccia aperta dall'elemento dipendente all'elemento a cui dipende].

Una *realizzazione* è una relazione esistente tra due elementi del modello, in cui uno deve realizzare o implementare il comportamento che l'altro specifica [linea tratteggiata con freccia chiusa dall'elemento che deve realizzare a quello che realizza].

Ad uno stereotipo possono essere associati tagged value specifici (una coppia [Tag, Valore] che, una volta associata ad un'entità di modello, determina una nuova proprietà, comune a tutti gli oggetti di tale tipo) e vincoli specifici. Si intende che qualsiasi elemento di modello venga "stereotipato" usando quel

particolare stereotipo acquisirà i tagged value corrispondenti e sarà soggetto ai vincoli definiti per lo stereotipo. (Es. <<call>>).

OCL (Object Constraint Language) è una specifica OGM e un linguaggio dichiarativo per descrivere le regole che si applicano ad UML.

Casi d'uso

I casi d'uso sono storie scritte, utilizzate per scoprire e registrare i requisiti, che indicano che cosa deve fare il sistema.

I diagrammi dei casi d'uso sono diagrammi di comportamento utilizzati per descrivere una serie di azioni (casi d'uso) che un qualche sistema (soggetto) dovrebbe o può eseguire in collaborazione con uno più utenti del sistema (attori).

I diagrammi dei casi d'uso sono utilizzati per specificare:

- I requisiti di un soggetto, gli utilizzi richiesti di un sistema (cosa dovrebbe fare un sistema in costruzione).
- La funzionalità offerta da un soggetto (cosa un sistema può fare).
- I requisiti che il soggetto specificato pone sul proprio ambiente (definendo come l'ambiente dovrebbe interagire con il soggetto in modo che sia in grado di eseguire i suoi servizi).

In UML un *attore* è un classificatore comportamentale che specifica un ruolo svolto da un'entità esterna che interagisce con il soggetto (es. scambiando segnali e dati), un utente umano del sistema progettato o qualche altro sistema/hardware che utilizza i servizi del soggetto [stick-man].

Uno *scenario* è una sequenza specifica di azioni e interazioni tra il sistema e alcuni attori. Descrive una particolare storia nell'uso del sistema, ovvero un percorso attraverso il caso d'uso.

Un *soggetto* è il sistema in analisi o progettazione al quale sono applicati un insieme di casi d'uso. Il soggetto potrebbe essere un business, un sistema software, un sistema fisico o un subsistema più piccolo che ha un qualche comportamento [rettangolo].

Un *caso d'uso* è un classificatore comportamentale che specifica il comportamento di un soggetto descrivendo un insieme di sequenze di azioni eseguite dal sistema al fine di ottenere un risultato osservabile di qualche valore per uno (o più) attori o stakeholders del sistema. In altre parole, ogni caso d'uso

descrive un'unità di funzionalità completa e utile, che il soggetto fornisce ai propri utenti [ellisse].

Tra gli *attori* può esserci una relazione di *generalizzazione*, ad esempio Admin e Customer possono essere specializzazioni di Web Client [linea con freccia, da figlio a padre].

Tra *attori* e *casi d'uso* può esserci solo una relazione binaria, ovvero un attore è collegato a un caso d'uso (quest'ultimo può essere collegato ad altri attori).

Tra i *casi d'uso* possono esserci vari tipi di relazione:

- Generalizzazione [linea con freccia da figlio a padre]

- Estende: specifica un comportamento

supplementare. È una relazione diretta che specifica

come e quando il comportamento definito nel caso

d'uso che estende può essere inserito nel

comportamento definito nel caso d'uso che viene

esteso. L'estensione può avvenire in uno o più extension points definite del caso

d'uso che viene esteso. Un extension point è una feature di un caso d'uso che

identifica un punto nel comportamento del caso d'uso in cui tale

comportamento può essere esteso da qualche altro caso d'uso [linea

tratteggiata con freccia aperta dal caso d'uso che estende verso quello esteso

con keyword <<extend>>].

- Include: specifica un comportamento

obbligatorio. È una relazione diretta tra due casi

d'uso utilizzata quando nessun comportamento

opzionale del caso d'uso incluso è inserito nel

comportamento del caso d'uso che include. Viene

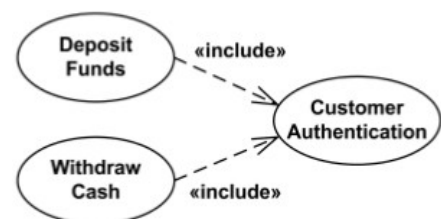
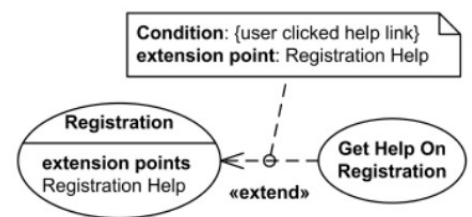
usata quando ci sono parti comuni del

comportamento di due o più casi d'uso, oppure per semplificare casi d'uso molto

grandi dividendoli in molti casi d'uso più piccoli [linea tratteggiata con freccia

aperta dal caso d'uso che include verso quello incluso con keyword

<<include>>].



Use case name
ID
Actors
Pre-conditions
Main sequence
Alternative sequences
Post-conditions

TIPS: quando un caso d'uso coinvolge più attori con obiettivi diversi, si dovrebbe modellare attraverso più casi d'uso. Il tempo può essere modellato come un attore. Uno scenario è un'istanza di un caso d'uso. A lato il template di un caso d'uso.

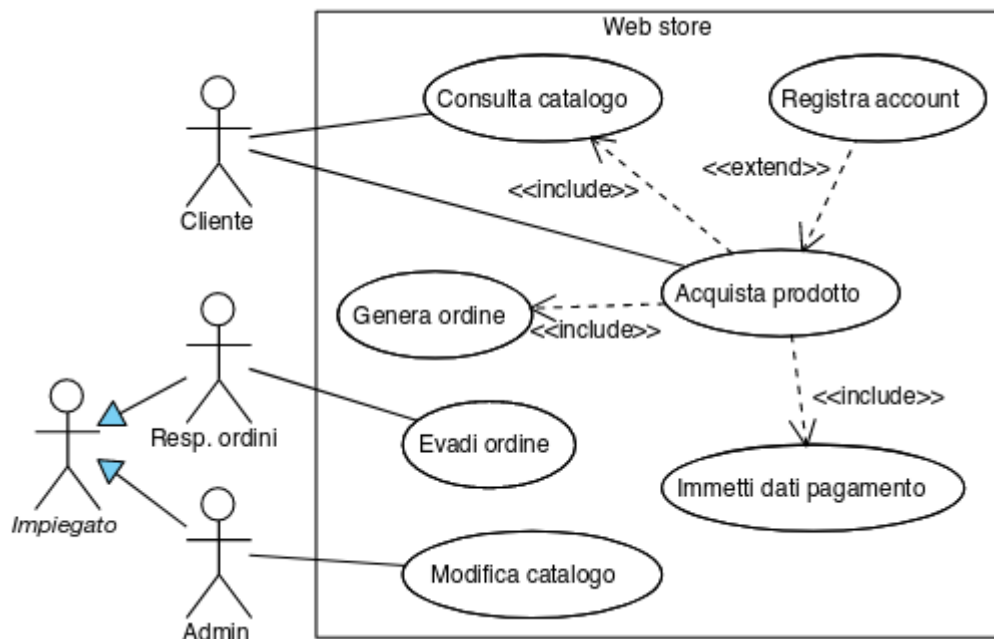


Diagramma delle classi

Una classe è un tipo di rappresentazione di oggetti che condividono caratteristiche comuni [rettangolo]. Nella rappresentazione UML può avere (oltre al nome) dei compartimenti, solitamente quello delle operazioni e quello degli attributi.

Attributi: [+,-,#,~][</>]<name>[:<type>][<mult>]

Operazioni: [+,-,#,~]<name>[(<params>)][:<type>]

in cui: + (public), - (private), # (protected), ~ (package)

L'istanza di una classe (od oggetto) deve contenere valori per ogni attributo membro di quella classe, seguendo le caratteristiche di tale attributo, come tipo e molteplicità.

Generalizzazione/specializzazione: ogni generalizzazione mette in relazione una classe specifica con una più generica, assumendo un meccanismo di ereditarietà. Se due o più classi sono diverse ma mostrano anche molte somiglianze, queste si possono raccogliere in una superclasse [linea con freccia chiusa da figlio a padre].

Dipendenza: una dipendenza mostra una relazione fornitore-client tra due elementi del modello, dove la modifica della classe destinazione (fornitore) può causare un cambiamento della classe sorgente (client). Indica che le semantiche dei clienti non sono complete senza i fornitori [linea tratteggiata con freccia aperta e keyword <<use>>].

Realizzazione (o implementazione): è una relazione tra una classe che implementa un'interfaccia, e l'interfaccia stessa [linea tratteggiata con freccia chiusa].

Associazione: un'associazione indica una relazione statica tra le istanze di due classi. Ha una molteplicità (numero di oggetti che prendono parte a tale relazione) e una navigabilità (il verso di percorribilità del collegamento) [linea con freccia aperta].

Aggregazione: un'aggregazione indica che parte dell'istanza è dipendente dal composto, è cioè presente una generica relazione "HAS-A". Ad esempio Scaffale aggrega Libro [linea con rombo vuoto dal lato della classe che aggrega].

Composizione: una composizione è un'associazione forte in cui l'oggetto componente può appartenere a un solo oggetto composto, e quindi la cancellazione dell'oggetto composto si propaga a tutti gli oggetti componenti. Ad esempio Cartella compone File [linea con rombo pieno dal lato della classe che compone].

Classi astratte: una classe astratta con ha istanze dirette, le sue istanze sono istanze di una delle sue specializzazioni. Le classi astratte sono composte da operazioni astratte, ovvero operazioni prive di implementazione.

Interfacce: le interfacce dichiarano servizi coerenti che sono implementati da classi che le implementano, non hanno cioè implementazioni al loro interno.

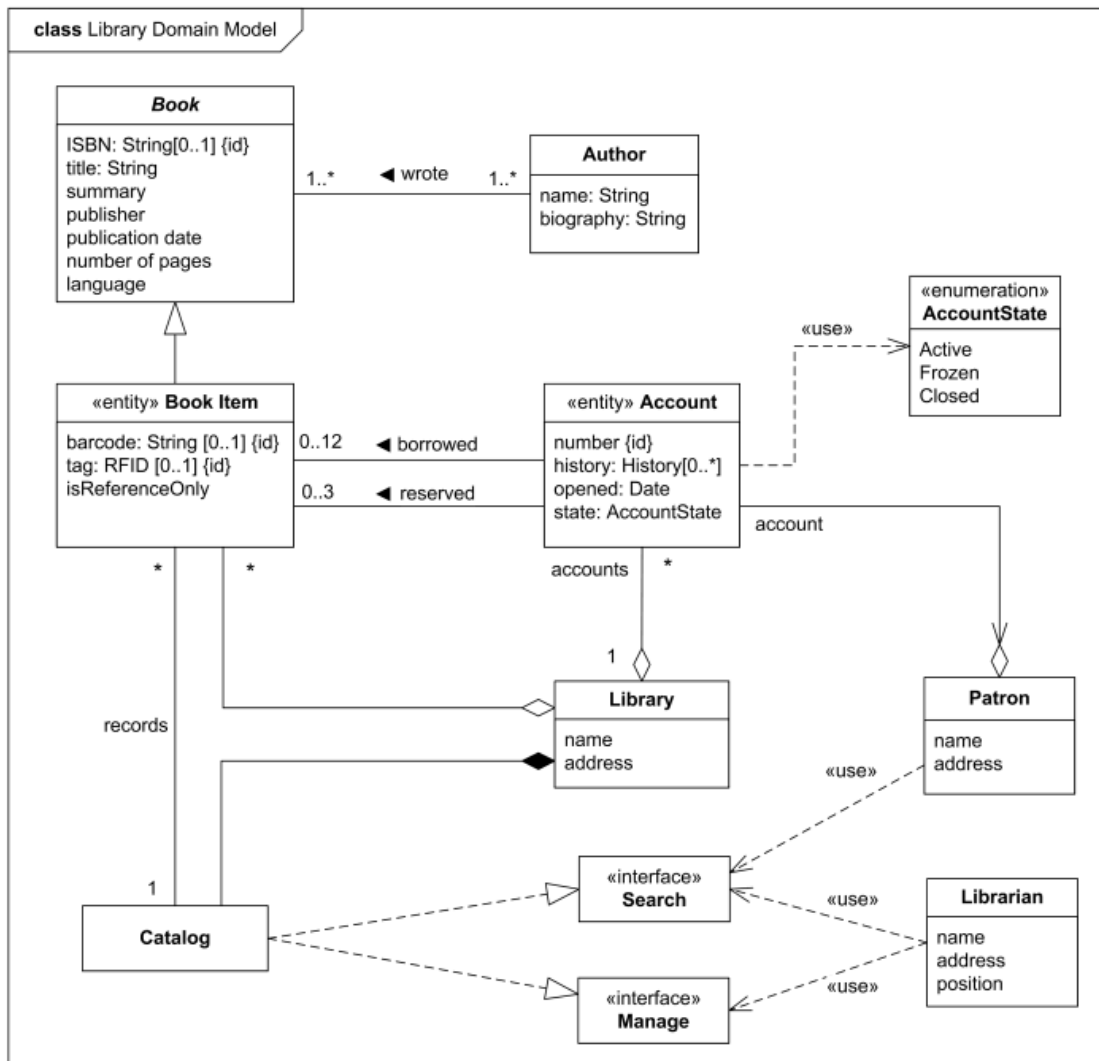


Diagramma delle attività

Il diagramma delle attività è un diagramma comportamentale UML che mostra il flusso di controllo o il flusso di un oggetto con enfasi sulla sequenza e le condizioni di tale flusso. Le azioni coordinate da modelli di attività possono essere iniziate in seguito al termine dell'esecuzione di altre attività, quando oggetti e dati diventano disponibili, o perché avvengono alcuni eventi esterni al flusso.

I diagrammi delle attività possono essere utilizzati per rappresentare il comportamento di elementi come: classi, casi d'uso, componenti o operazioni di una classe. Le entità sono: attività, nodi di attività (azione, oggetto, controllo) e archi di attività.

Attività: è un comportamento parametrico rappresentato come flusso coordinato di azioni. Il flusso di esecuzione è modellato tramite nodi di attività

connessi da archi di attività [rettangolo con bordi arrotondati al cui interno vi è il nome e i vari nodi di attività, connessi dagli archi di attività].

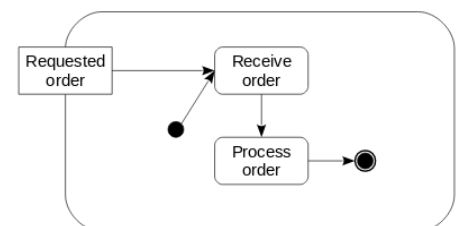
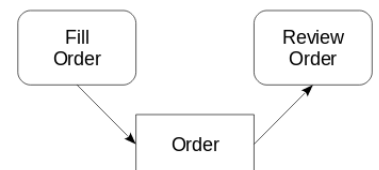
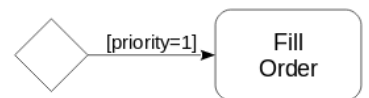
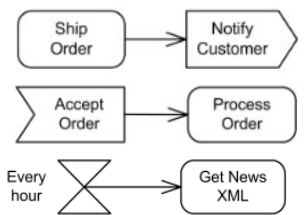
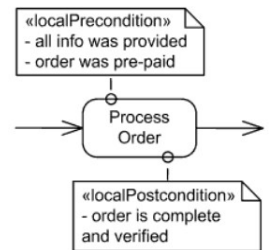
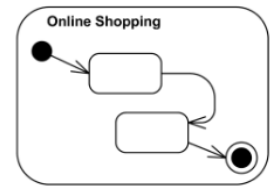
Azioni: è un elemento con un nome che rappresenta un singolo passo atomico all'interno di un'attività [rettangolo con i bordi arrotondati con all'interno il nome, può avere archi in entrata e in uscita]. Le azioni possono avere anche precondizioni e postcondizioni che per procedere vanno soddisfatte rispettivamente prima e dopo che l'azione sia eseguita. Sono inoltre presenti degli eventi per le azioni come invio di un segnale, l'accettazione di un segnale e un tempo ripetuto. Una chiamata al comportamento di un'azione è rappresentata tramite una forma a rastrello all'interno di un'azione.

Archi: è una connessione diretta tra due nodi di attività, da quello di origine, a quello di destinazione. Un arco può avere una guardia, ovvero una specificazione valutata a runtime che determina se l'arco può essere attraversato o meno.

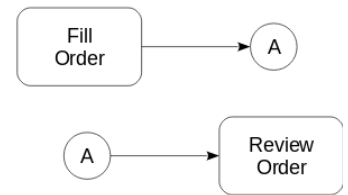
Oggetto: un nodo oggetto è un nodo di attività che è parte della definizione del flusso in un'attività. Indica che un'istanza di un certo classificatore, in un certo stato, potrebbe essere disponibile in un certo punto dell'attività.

Parametro di attività: è un oggetto posizionato all'inizio e alla fine dei flussi, fornisce i mezzi per accettare input a un'attività e fornire output dall'attività. È rappresentato come un pallino nero come inizio, e un pallino nero cerchiato come fine.

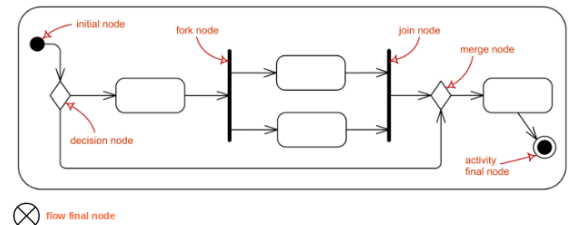
Input/Output: i pin di input sono nodi oggetto che ricevono valori da altre azioni attraverso flussi oggetto. I pin di output sono oggetti che consegnano valori ad altre azioni attraverso flussi oggetto.



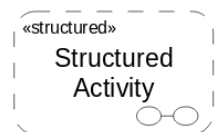
Connettori: un arco di attività può essere rappresentato utilizzando un connettore, è un cerchio con un nome all'interno. Questi sono utilizzati per evitare archi troppo lunghi.



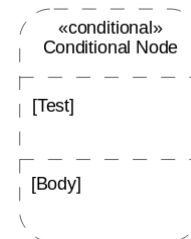
Controllo: il nodo di controllo è un nodo di attività utilizzato per coordinare il flusso tra altri nodi, include un nodo iniziale, un nodo di fine flusso, un nodo di fine attività, un nodo di decisione, un nodo di merge, un nodo di fork e un nodo di join.



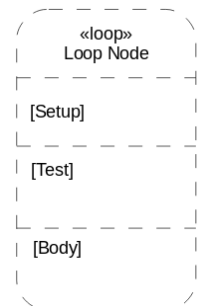
Attività strutturata: questi nodi sono nodi che contengono altri nodi. Un nodo non può essere contenuto da più di un nodo strutturato, tuttavia nodi strutturati possono contenere altri nodi strutturati.



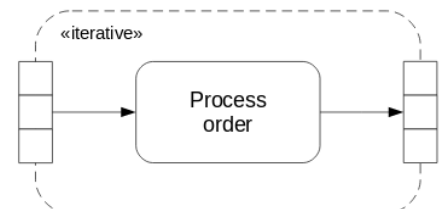
Nodo condizionale: è un'attività strutturata che rappresenta una scelta esclusiva tra un certo numero di alternative.



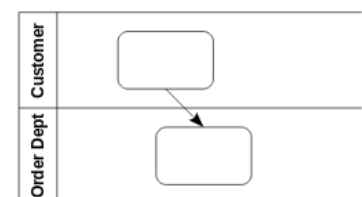
Nodo loop: è un'attività strutturata che rappresenta un loop con sezioni setup, test e corpo.

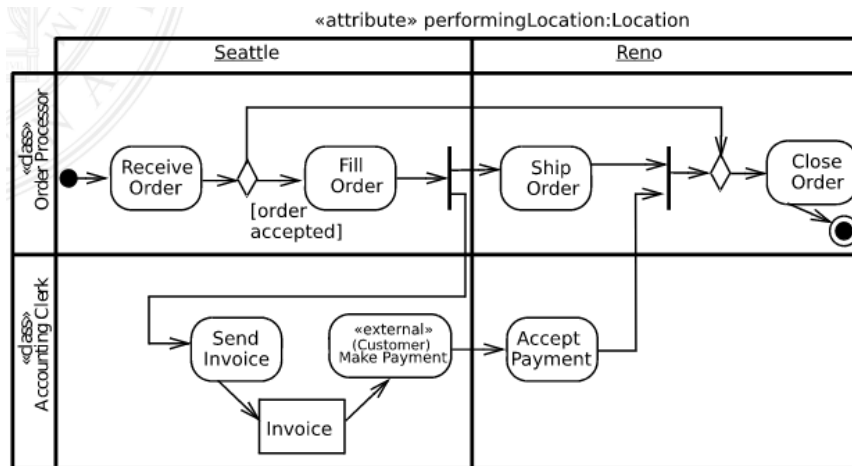


Regione di espansione: è un nodo strutturato che prende in input collezioni, opera su ogni elemento delle collezioni in modo individuale e produce elementi per le collezioni di output.



Partizione dell'attività: è un gruppo di attività per azioni che hanno alcune caratteristiche comuni. Le partizioni forniscono una visione vincolata sui comportamenti invocati nelle attività e spesso corrispondono a unità organizzative o attori di business in un modello di business.





Regioni interrompibili e archi di interruzione: una regione interrompibile è un tipo di gruppo di attività che fornisce un meccanismo per distruggere tutti i token e terminare tutti i comportamenti nella sezione dell'attività racchiusa tra i bordi della regione. Quando un token è accettato da un particolare tipo di arco chiamato arco di interruzione, che è rappresentato tramite un fulmine, abbandona la regione e tutti gli altri token sono distrutti e i comportamenti all'interno della regione sono terminati.

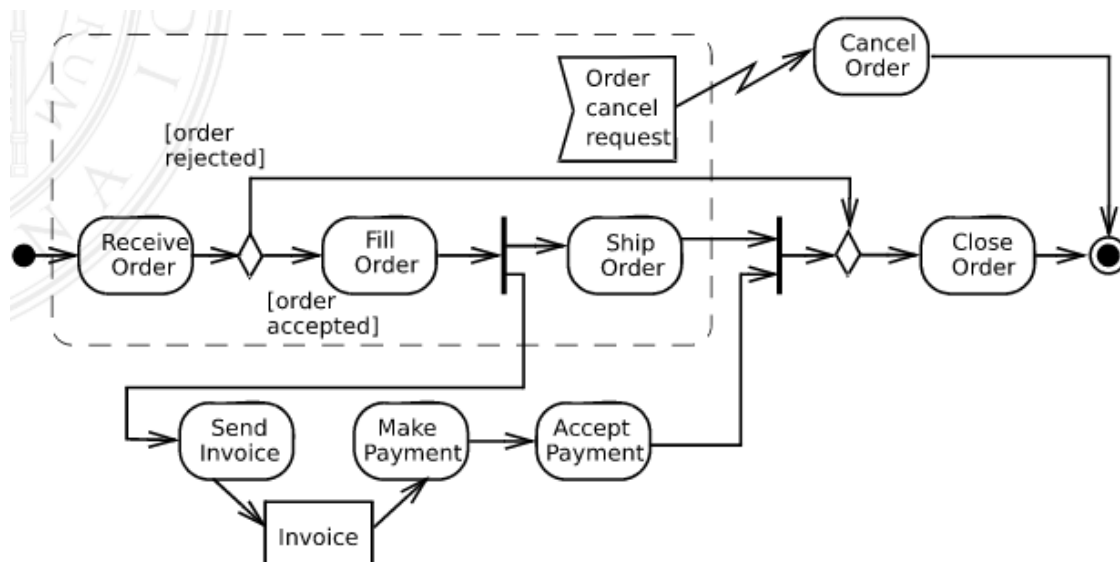


Diagramma di interazione

Sono diagrammi che modellano il *comportamento*: visualizzano le interazioni tra varie entità di un sistema mostrando lo scambio di messaggi tra entità nel tempo.

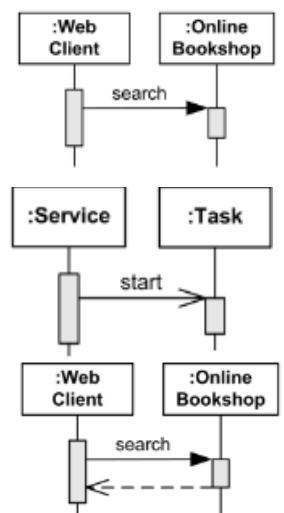
Il loro scopo è mostrare come un certo comportamento viene realizzato dalla collaborazione delle entità in gioco (fino ad ora ci siamo occupati solo di cosa fa il sistema). Per procedere è necessario un comportamento da realizzare tratto da un classificatore (come un caso d'uso, o un'operazione di una classe) e una serie di elementi che realizzano il comportamento (come attori o istanze di classe).

Il *diagramma di sequenza* è adatto a mostrare la sequenza temporale degli avvenimenti per ogni entità nel diagramma. È il tipo più comune di diagramma di interazione, e si focalizza sullo scambio di messaggi tra un certo numero di lifelines.

Una *lifeline* è un elemento che rappresenta un partecipante individuale nell'interazione. Mentre le parti e le caratteristiche strutturali potrebbero avere molteplicità superiore ad 1, le lifelines rappresentano solo una delle entità che interagiscono.

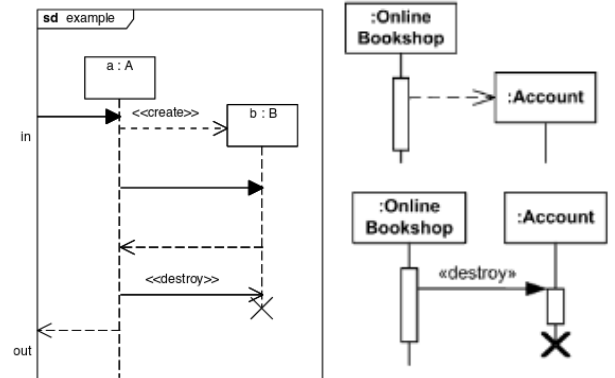
Un *messaggio* è un segnale che si scambiano le istanze di classificatori, è costituito da un elemento con un nome che definisce uno specifico tipo di comunicazione tra le lifelines di un'interazione. Il messaggio specifica non solo il tipo di comunicazione, ma anche il mittente e il destinatario (endpoints). Un messaggio riflette sia una chiamata di un'operazione e l'inizio dell'esecuzione sia l'invio e la ricezione del messaggio. Il messaggio può essere: chiamata sincrona/asincrona, segnale asincrono, risposta, create o delete.

- *Chiamata sincrona*: una chiamata sincrona solitamente rappresenta una chiamata operativa (invio del messaggio e sospensione dell'esecuzione mentre si attende la risposta).
- *Chiamata asincrona*: una chiamata asincrona prevede l'invio del messaggio e procede immediatamente senza attendere il valore di ritorno.
- *Risposta*: un messaggio di risposta ad una chiamata operativa è la risposta a tale operazione.
- *Create*: è un messaggio inviato alla lifeline per crearla, di solito si invia questo messaggio a un oggetto non ancora esistente per crearlo.



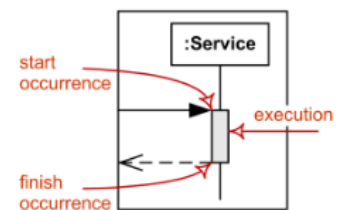
- *Delete*: è un messaggio invitato per terminare una lifeline (di solito la lifeline termina con una X in fondo).

Un *gate* è un qualunque punto sulla cornice esterna del diagramma, che può essere la fonte o la destinazione di una freccia (a destra i due gates sono in e out).

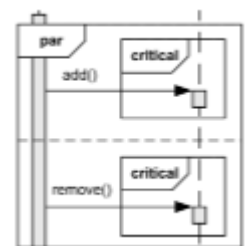


Un *frammento di interazione* è un elemento con un nome, che rappresenta l'unità di interazione più generale. Ogni frammento di interazione è concettualmente come un'interazione con se stesso. Non c'è una notazione generale per questo elemento (alcuni esempi sono: occorrenza, esecuzione, stato invariante, frammento combinato, interazione d'uso).

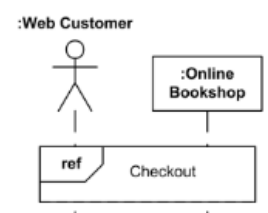
Occorrenza: è un frammento di interazione che rappresenta un momento nel tempo (evento) all'inizio o alla fine di un messaggio o di un'esecuzione.



Esecuzione: (in modo informale attivazione) è un frammento di interazione che rappresenta un periodo nella lifeline del partecipante in cui: esegue un'unità di comportamento o azione all'interno della lifeline, invia un segnale ad un altro partecipante oppure attende un messaggio di risposta da un altro partecipante.



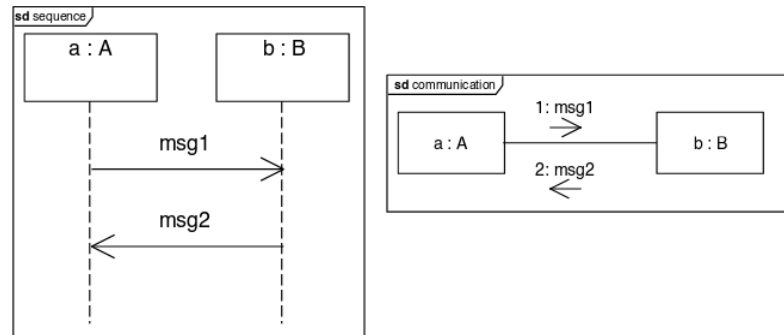
Frammento combinato: è una sottoarea di un diagramma di sequenza che racchiude una parte dell'interazione e le modalità della sua esecuzione. È definito da un operatore di interazione e i corrispondenti operandi di interazione (e può avere guardie). Attraverso l'uso di frammenti combinati l'utente sarà in grado di descrivere un numero di tracce in modo compatto e coinciso.



Interazione d'uso: è un frammento di interazione che permette di utilizzare (o chiamare) un'altra interazione. Sequenze di diagrammi grandi e complesse potrebbero essere semplificate tramite interazioni d'uso. È anche comune riutilizzarle come interazioni tra molte altre interazioni.

Diagrammi di comunicazione: mostrano le interazioni tra lifelines utilizzando una disposizione senza forma, tali diagrammi possono essere convertiti in diagrammi

di sequenza. Si tratta comunque di diagrammi utili a visualizzare i canali di comunicazione tra le entità. Sono simili ai diagrammi degli oggetti, ma i link tra gli oggetti trasportano messaggi come nei diagrammi di sequenza. Nei diagrammi di comunicazione si assume che i messaggi siano ricevuti nello stesso ordine in cui sono generati. (a sinistra diagramma di sequenza a destra di comunicazione)



Espressioni di sequenza dei messaggi:

Sequence-expression ::= sequence-term '...' message-name

Sequence-term ::= [integer[name]][recurrence]

I sequence-term sono usati per rappresentare l'annidamento dei messaggi all'interno di un interazione. Esempi

1.2.1 [!error]: msg(param1,param2)

il messaggio è inviato solo se la guardia è vera

1.2.1 * [for i=1 to n]: : sendMessageTo(i)

esempio con l'operatore di iterazione *

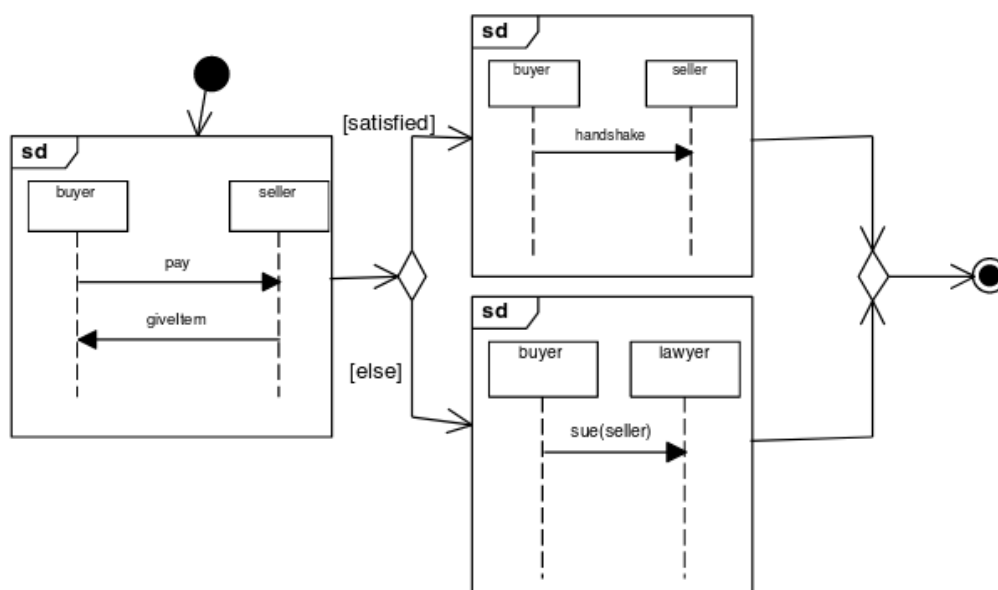


Diagramma di macchina di stato

Una *macchina di stato comportamentale* descrive un comportamento discreto basato su eventi di un sistema (o una parte di un sistema), come attraversamento dei vertici di un grafo (solitamente stati) connessi da transizioni.

Un *protocollo state machine* descrive il ciclo di vita o le sequenze di interazioni valide (protocolli) per parti di un sistema (un classificatore).

Diagramma di attività vs comportamento

Il diagramma di attività modella un comportamento (che riguarda una o più entità) come un insieme di azioni organizzate secondo un flusso.

Il diagramma di stato modella il comportamento (generalmente di una sola entità) come variazioni del suo stato interno.

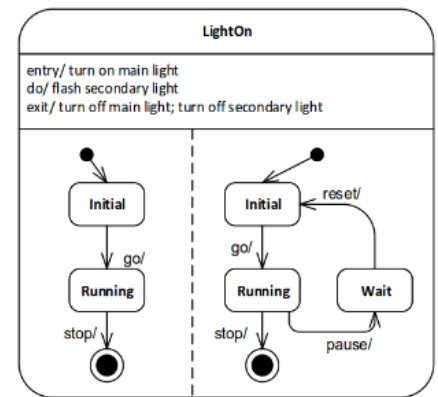
Uno *stato* è una situazione nella quale è valida una qualche condizione invariante (stessi stimoli, stessa risposta). Gli stati possono essere semplici (nessun vertice interno o transizioni), composti (contiene una o più regioni, gli stati in queste regioni sono chiamati sub-stati) o submachine. All'interno di uno stato si possono usare alcune azioni speciali: *enter* (eseguita quando l'oggetto entra nello stato), *exit* (eseguita quando l'oggetto esce dallo stato), *doActivity* (eseguita mentre l'oggetto è nello stato) [rettangolo arrotondato].

Una *transizione* descrive il passaggio (atomico) da uno stato ad un altro. Le transizioni sono innescate da eventi e possono essere: raggiunte, attraversate o completate. Durante l'attraversamento la state machine può eseguire alcune attività [arco diretto].

Un *evento* è un'occorrenza osservabile (nell'ambiente del soggetto). Prende luogo in un certo punto del tempo e non ha durata. Può avere parametri e i tipi base sono: *MessageEvent* (*CallEvent*, *SignalEvent*), *ChangeEvent*, *TimeEvent*.

Lo *stato finale* è uno speciale tipo di stato che specifica che la regione che lo racchiude è completata. Ci sono inoltre molti *pseudo-stati* utilizzati per arricchire la semantica della state machine (initial, join, choice, fork, junction, terminate, entryPoint, exitPoint).

Stati e transizioni possono essere organizzati in *regioni* (possibilmente in modo gerarchico). Regioni ortogonali descrivono un comportamento concorrente (destra).



Pseudo-stati:

Initial: rappresenta il punto di partenza di una regione. È l'origine di al massimo una transizione non associata a un trigger o a una guardia.

Join: è il punto di raccordo tra due o più transizioni originate da vertici in diverse regioni ortogonali; hanno una funzione di sincronizzazione, dove tutte le transizioni in entrata devono essere completate prima che l'esecuzione possa continuare attraverso la transizione d'uscita.

Fork: divide una transizione in entrata in una o più transizioni che termineranno in vertici contenuti in diverse regioni ortogonali di uno stato composto.

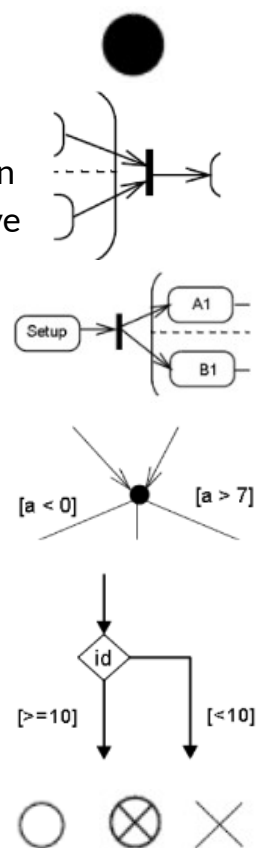
Junction: unisce o divide transizioni.

Choice: è un tipo di junction utilizzato per realizzare un branching condizionale dinamico. Una guardia di tipo 'else' può essere utilizzata in una transizione predefinita che è selezionata quando tutte le altre guardie vengono valutate come false.

EntryPoint: è un punto di entrata per uno stato composto o uno stato sub-machine.

ExitPoint: è un punto di uscita per uno stato composto o uno stato sub-machine.

Terminate: in questo modo l'esecuzione della state machine viene terminata immediatamente.



La *state history* è utilizzata per tener traccia della configurazione dello stato di una regione. La regione può essere reimpostata ad una configurazione di stato tramite una transizione (locale) che si connette ad una history pseudostate.

Al momento della creazione, una state machine esegue la propria inizializzazione eseguendo la transizione composta iniziale, poi entra in un wait-

point. Quando gli eventi vengono inviati, i trigger vengono valutati e se almeno una transizione può essere innescata, viene eseguita una nuova transizione composta (uno step) e viene raggiunto un nuovo wait-point. Questo ciclo si ripete fino a quando la state-machine completa il proprio comportamento, o quando viene terminata in modo asincrono da qualche agente esterno. Questo modello di esecuzione è conosciuto come *Run-To-Completion* (RTC). Tale modello semplifica la gestione della concorrenza nelle macchine di stato: quando la macchina non è in uno stato ben definito, non è reattiva (l'invio non è effettuato).

Qualità del Software e Principi Object Oriented

Il software può avere qualità esterne, come correttezza, usabilità, efficienza, affidabilità, integrità, adattabilità, accuratezza e robustezza. O qualità interne, come manutenibilità, flessibilità, portabilità, ri-usabilità, leggibilità, ecc.

Rigidità: è la tendenza del software ad essere difficile da modificare, anche in modi semplici. Più moduli devono essere cambiati e più rigido è il design.

Fragilità: è la tendenza di un programma a “rompersi” in vari punti quando una singola modifica viene fatta.

Immobilità: un design è immobile quando contiene parti che potrebbero essere utili in altri sistemi, ma lo sforzo e il rischio necessari a separare queste parti dal sistema originale sono troppo elevati.

Viscosità: quando alcune opzioni per effettuare modifiche al sistema preservano il design, mentre altre no. Un ambiente è viscoso quando lo sviluppo di tale ambiente è lento e inefficiente.

Complessità inutile: quando contiene elementi che non sono al momento utili (succede quando si vogliono anticipare possibili modifiche future).

Ripetitività inutile: quando lo stesso codice appare spesso in forme leggermente diverse (gli sviluppatori non hanno utilizzato bene l'astrazione).

Opacità: è la tendenza di un modulo ad essere difficile da capire.

SOLID

Single Responsibility Principle (SRP): ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso. Una classe dovrebbe avere una sola ragione per cambiare, sarebbe cattiva progettazione accoppiare due cose che cambiano, per motivi diversi in momenti diversi (pena: fragilità).

Open-Closed Principle (OCP): le entità (classi, moduli, funzioni, ecc.) software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche; in maniera

tale che un'entità possa permettere che il suo comportamento sia modificato senza alterare il suo codice sorgente. Quindi una volta completata l'implementazione di un'entità, questa non dovrebbe essere più modificata, eccetto che per eliminare errori di programmazione. Non rispettare questo principio fa sì che un minimo cambiamento porti a modifiche a cascata. (pena: rigidità)

Liskov Substitution Principle (LSP): la relazione tra classi in una gerarchia dovrebbe essere sottotipata. Se S è un sottotipo di T , allora oggetti di tipo T dichiarati in un programma possono essere sostituiti con oggetti di tipo S senza alterare la correttezza dei risultati del programma. Di conseguenza se ad esempio un metodo f , che accetta come argomento una referenza a B , si comporta in modo errato quando è passata una referenza a D , sottoclasse di B , allora D è fragile in presenza di f .

Interface Segregation Principle (ISP): La dipendenza di una classe ad un'altra dovrebbe dipendere dall'interfaccia più piccola possibile. Quindi un client non dovrebbe dipendere da metodi che non usa, e pertanto è preferibile che le interfacce siano molte, specifiche e piccole (composte da pochi metodi) piuttosto che poche, generali e grandi. (pena: complessità inutile)

Dependency Inversion Principle (DIP): I moduli di alto livello non devono dipendere da quelli di basso livello; entrambi devono dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli; sono i dettagli che dipendono dalle astrazioni.

GRASP

UML descrive una responsabilità come “un contratto o un obbligo di un classificatore”. Il *Responsibility-Driven Design* è un metodo di design che si focalizza sugli oggetti come astrazioni comportamentali caratterizzate da responsabilità, e migliora l'incapsulazione utilizzando un modello client-server. Per creare le astrazioni comportamentali si utilizzano comunemente le CRC-card.

Applicare le responsabilità di un oggetto include: fare qualcosa da solo, come creare un oggetto o effettuare calcoli; inizializzare un'azione in altri oggetti; controllare e coordinare attività in altri oggetti.

GRASP (General Responsibility Assignment Software Patterns) è una collezione di pattern, usata nella progettazione object-oriented, che fornisce delle linee guida per l'assegnazione di responsabilità alle classi e agli oggetti.

Ogni *pattern* illustra un problema che può verificarsi continuamente nel nostro ambiente, e descrive il cuore della soluzione al problema, in modo tale che possa essere usata sempre. I pattern sono: Creator, Information Expert, Low Coupling, Controller, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations.

Creator: risolve il problema di chi debba essere responsabile per la creazione di una nuova istanza di una classe. Date due classi A e B, B potrebbe essere responsabile per la creazione di A nel caso in cui B contenga (o nel complesso aggregi, registri, usi strettamente e contenga) le informazioni iniziali per A.

Information Expert: permette di assegnare correttamente le responsabilità agli oggetti. Stabilisce che la responsabilità deve essere assegnata all'Information Expert (la classe che possiede tutte le informazioni essenziali). Migliora l'incapsulamento e riduce così l'accoppiamento di altre classi per l'implementazione della classe Information Expert.

Low Coupling: (accoppiamento: grado con cui ciascuna componente di un software dipende dalle altre componenti) è un pattern valutativo, che stabilisce come assegnare le responsabilità per supportare: bassa dipendenza tra classi; basso impatto in una classe dei cambiamenti in altre classi; e alto potenziale di riuso.

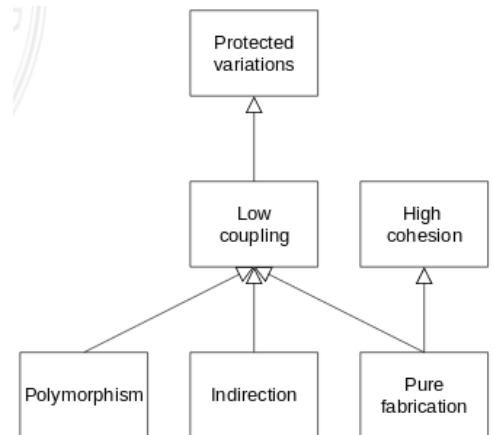
Controller: assegna la responsabilità di eventi di chiamata a sistema ad una classe (che non sia ad interfaccia utente) che rappresenta l'intero sistema in uno scenario di caso d'uso. Un controllore di casi d'uso potrebbe essere impiegato per comunicare con tutti gli eventi di sistema di uno o più casi d'uso (ad esempio, per i casi d'uso *Crea Utente* ed *Elimina Utente*, si può usare un solo controllore *ControlloreUtente*, anziché due separati per ciascun caso d'uso).

High Cohesion: (coesione: misura di quanto strettamente correlate siano le varie funzionalità messe a disposizione da un singolo modulo) è un pattern valutativo che cerca di mantenere gli oggetti focalizzati, gestibili e intelligibili in maniera appropriata. Un'alta coesione significa che le responsabilità di un dato elemento sono fortemente correlate ed altamente focalizzate.

Polymorphism: la responsabilità di definizione delle variazioni dei comportamenti basati sul tipo viene assegnata ai tipi per i quali avviene tale variazione. Ciò viene ottenuto usando le operazioni di polimorfismo.

Pure Fabrication: è una classe che non rappresenta un concetto nel dominio del problema. Ma viene creata per ottenere basso accoppiamento, alta coesione e potenziale di riuso da esso derivante (quando non vi è una soluzione presentata dal pattern Information Expert).

Indirection: supporta il basso accoppiamento (ed il potenziale di riuso) tra due elementi attraverso l'assegnazione di responsabilità di mediazione tra di loro ad un oggetto intermedio. Un esempio è l'introduzione di un componente controllore per la mediazione tra i dati (modello) e la loro rappresentazione (vista) nel pattern Model-View-Controller.



Protected Variations: protegge gli elementi dalle variazioni compiute in altri elementi (oggetti, sistemi, sottosistemi) mascherandoli con un'interfaccia ed usando il polimorfismo per creare diverse implementazioni di questa interfaccia.

Design Patterns

La Gang of Four (GoF) indica che i patterns possono avere 3 scopi:

Creazionali: astraggono il processo di istanziazione - Abstract Factory, Builder, Factory Method, Prototype, Singleton.

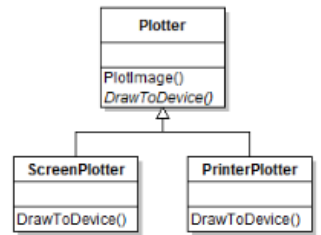
Strutturali: sono legati a come le classi e gli oggetti sono composti per formare strutture più grandi - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

Comportamentali: sono legati agli algoritmi e l'assegnamento di responsabilità tra oggetti - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Considerazione: composizione invece di ereditarietà

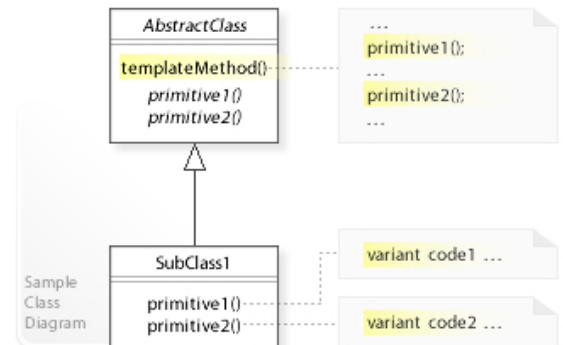
Le due tecniche più comuni per riutilizzare funzionalità nei sistemi object-oriented sono l'ereditarietà tra classi e la composizione di oggetti (combinare oggetti semplici o tipi di dato in oggetti più complessi). Favorire la composizione di oggetti all'ereditarietà tra classi aiuta a mantenere ogni classe incapsulata e concentrata su un compito. La delegazione è un modo di rendere la composizione tanto potente per il riuso quanto l'ereditarietà.

Il problema dell'ereditarietà è che non viene affrontato in modo corretto né l'Open/Closed Principle (OCP) né Protected Variations (PV). Tuttavia si deve sapere che le superclassi non sono l'unica astrazione possibile, ed è possibile utilizzare la composizione, che è più flessibile e può cambiare a runtime.



Template Method [Pattern Comportamentale]

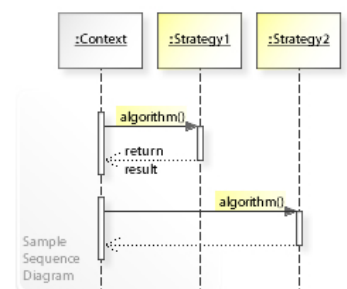
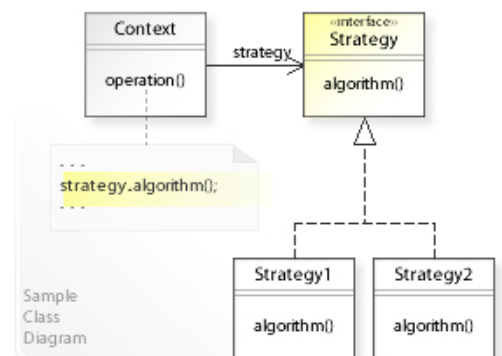
Questo pattern permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono. In questo modo si può ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune. Normalmente sono le sottoclassi a chiamare i metodi delle classi genitrici; in questo *pattern* è il metodo template, appartenente alla classe genitrice, a chiamare i metodi specifici ridefiniti nelle sottoclassi. Tale pattern aiuta ad adempiere all'OCP.



Strategy [Pattern Comportamentale]

L'obiettivo di questa architettura è isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione.

Si pensi ad esempio alle possibili visite in una struttura ad albero (visita anticipata, simmetrica, posticipata); mediante il pattern strategy è possibile selezionare a tempo di esecuzione una tra le visite ed eseguirla sull'albero per ottenere il risultato voluto. Tale pattern aiuta ad implementare OCP e obbedisce alle PV, inoltre favorisce la composizione rispetto all'ereditarietà (has-a invece di is-a).

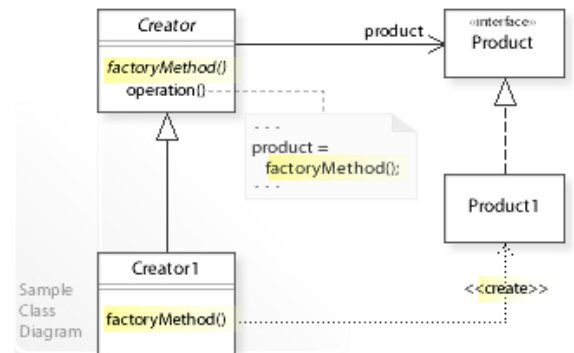


Considerazione: il New è considerato dannoso

Il New alloca memoria ogni volta che un client ha bisogno di una referenza, non c'è controllo sul ciclo di vita delle istanze e crea una dipendenza tra l'utente e la classe concreta implementata dalla referenza (viola il Dependency Inversion Principle - DIP). Inoltre le modifiche nel costruttore della classe concreta creano problemi nei client (viola l'OCP).

Factory Method [Pattern Creazionale]

Questo pattern 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.

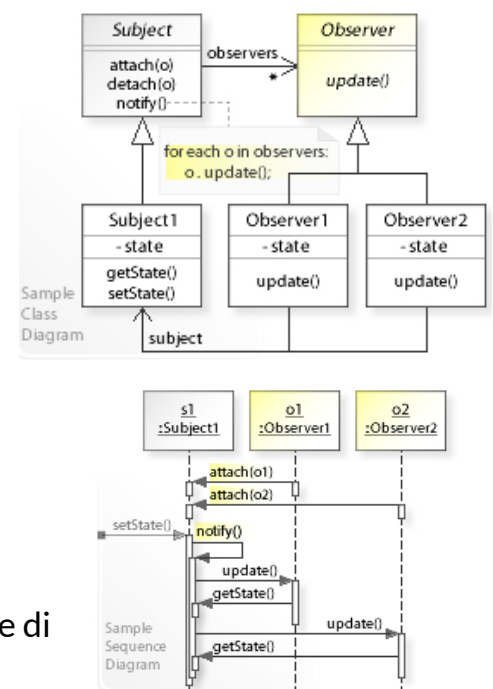


Considerazione: il problema delle notifiche

Nuovi osservatori (e nuovi tipi di osservatori) possono apparire in un tempo futuro. Quando una classe "notifica" un'altra è esposta alla sua interfaccia (quindi dipende da quell'interfaccia).

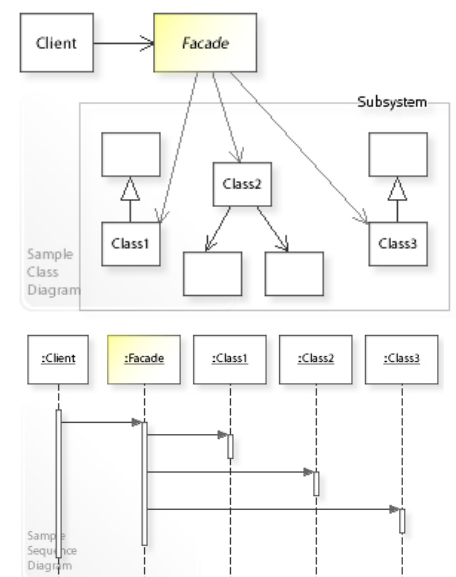
Observer [Pattern Comportamentale]

Sostanzialmente il pattern si basa su uno o più oggetti, chiamati osservatori o observer, che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto "osservato", che può essere chiamato soggetto. Oltre all'observer esiste il concrete Observer 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, a cui spesso vengono passati dei parametri in fase di generazione dell'evento.



Façade [Pattern Strutturale]

Significa "facciata", ed infatti 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. Fornisce un'interfaccia unificata ad un insieme di interfacce di un sottosistema: definisce un'interfaccia a livello più alto che rende il sottosistema più facile da utilizzare. Nelle librerie standard Java (Java 2 Platform, Standard Edition) questo pattern viene spesso usato; si considerino ad esempio tutte le classi disponibili per fare il rendering del testo o delle forme geometriche,

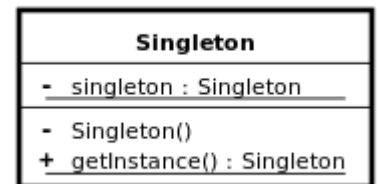


un programmatore può ignorare tutte queste classi e utilizzare unicamente le classi *façade* (Font e Graphics) che offrono un'interfaccia più semplice e omogenea.

Singleton [Pattern Creazionale]

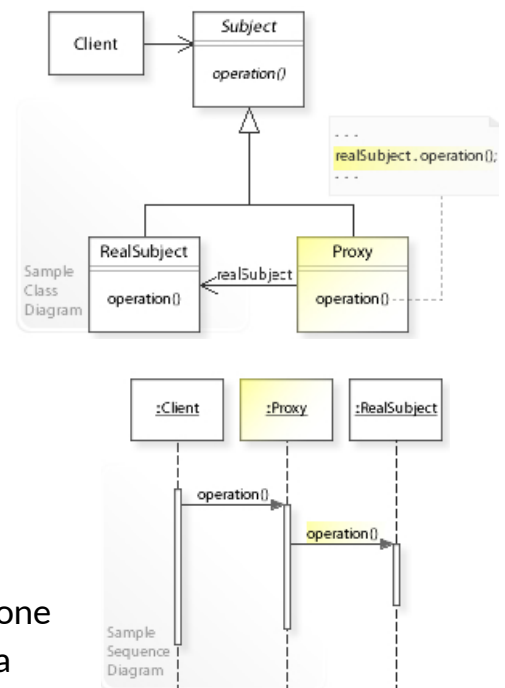
È un pattern che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziatura diretta della classe (fornisce anche un metodo "getter" statico che restituisce l'istanza della classe).



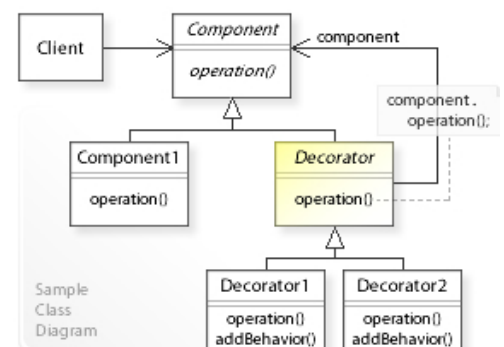
Proxy [Pattern Strutturale]

Nella sua forma più generale, un proxy è una classe che funziona come interfaccia per qualcos'altro. L'altro potrebbe essere qualunque cosa: una connessione di rete, un grosso oggetto in memoria, un file e altre risorse che sono costose o impossibili da duplicare. Tipicamente viene creata un'istanza di oggetto complesso, e molteplici oggetti proxy, ognuno dei quali contiene un riferimento al singolo oggetto complesso. Ogni operazione svolta sui proxy viene trasmessa all'oggetto originale. Una volta che tutte le istanze del proxy sono distrutte, l'oggetto in memoria può essere deallocato. Un esempio ben conosciuto di proxy pattern è la tecnica reference counting dei puntatori (tecnica di memorizzazione del numero di riferimenti, puntatori o handle a una risorsa come un oggetto o un blocco di memoria, tipicamente usata come metodo per deallocare oggetti che non sono più usati in modo automatico e sicuro).



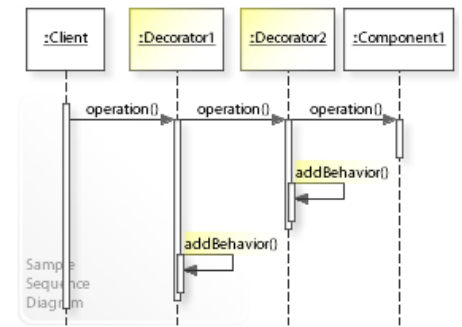
Decorator [Pattern Strutturale]

Questo pattern consente di aggiungere nuove funzionalità ad oggetti già esistenti. Ciò 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



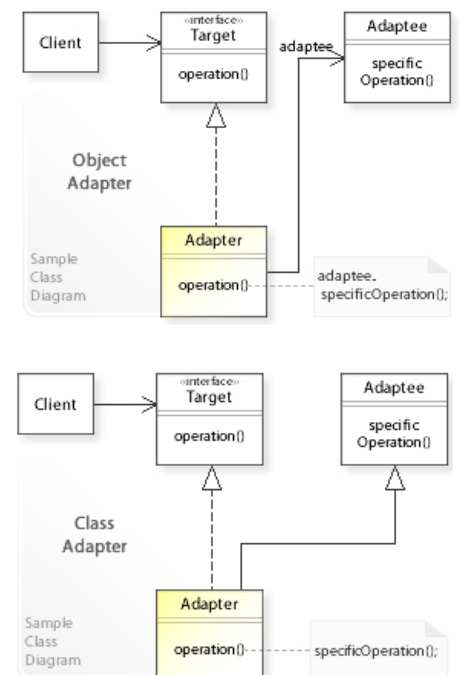
decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall'ultimo anello della catena).

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.



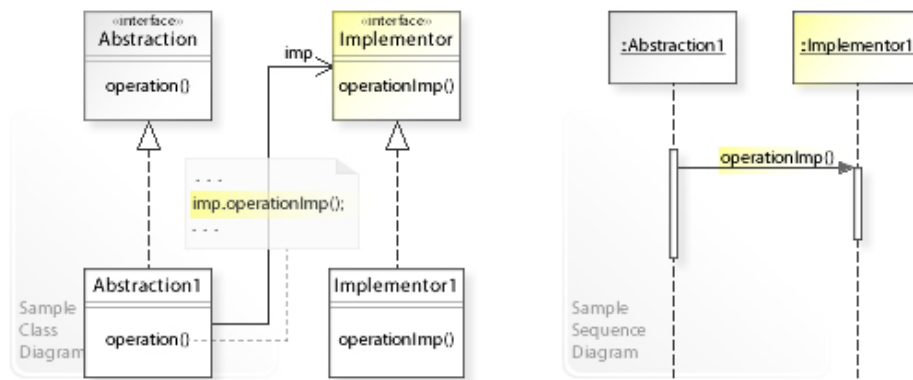
Adapter [Pattern Strutturale]

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 librerie) 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.



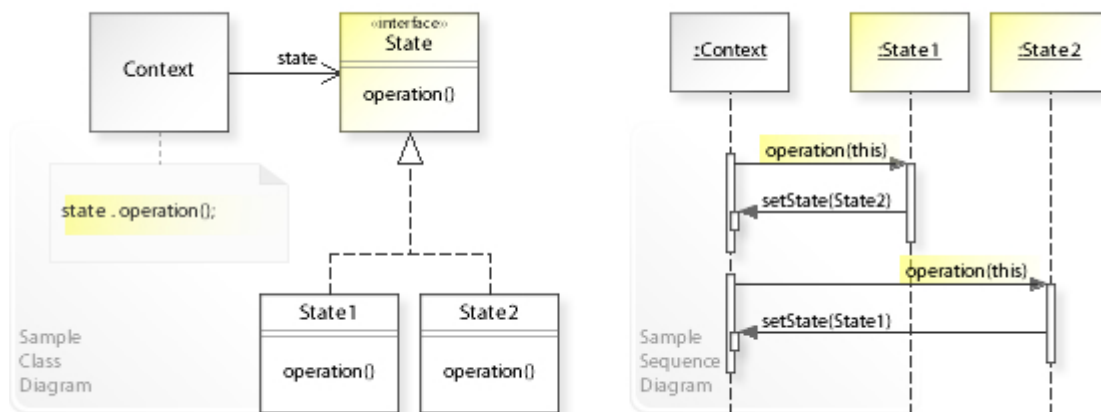
Bridge [Pattern Strutturale]

Disaccoppia un'astrazione dalla sua implementazione in modo che le due possano variare in modo indipendente. Il pattern Bridge permette di separare l'interfaccia di una classe (che cosa si può fare con la classe) dalla sua implementazione (come lo fa). In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato.



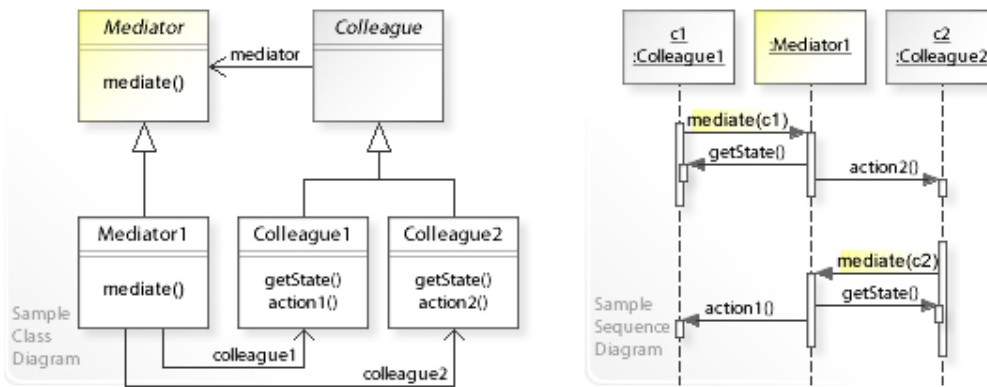
State [Pattern Comportamentale]

Permette ad un oggetto di alterare il proprio comportamento a run-time in funzione dello stato in cui si trova. Sembrerà che l'oggetto cambi la propria classe.



Mediator [Pattern Comportamentale]

Questo pattern ha l'intento di disaccoppiare entità del sistema che devono comunicare fra loro. Infatti fa in modo che queste entità non si referenzino 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.

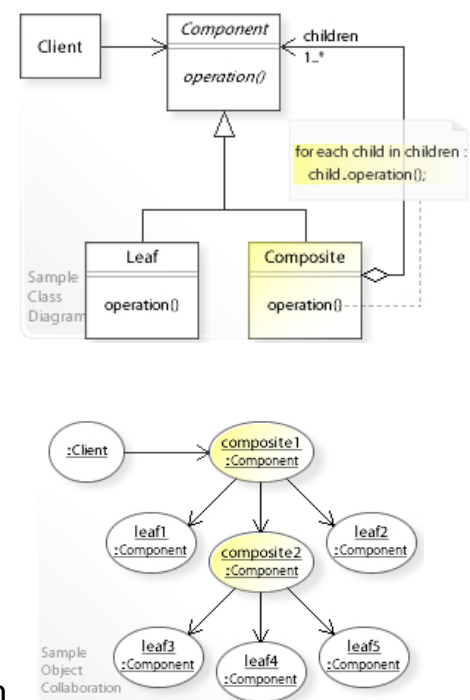


Composite [Pattern Strutturale]

Questo pattern permette di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. 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.

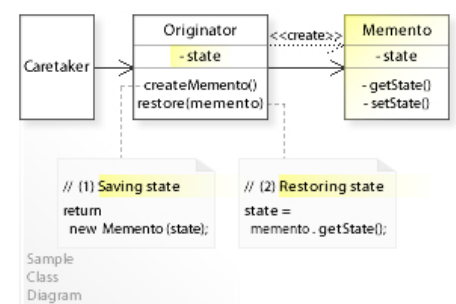
Nella programmazione orientata agli oggetti un Composite è un oggetto (per esempio una figura geometrica) progettato per composizione di uno o più oggetti simili (una figura geometrica come un trapezio può essere vista a sua volta come formata da due triangoli rettangoli e un rettangolo) che offrono tutte le medesime funzionalità. Questa caratteristica è conosciuta anche come relazione "has-a" tra gli oggetti.

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.



Memento [Pattern Comportamentale]

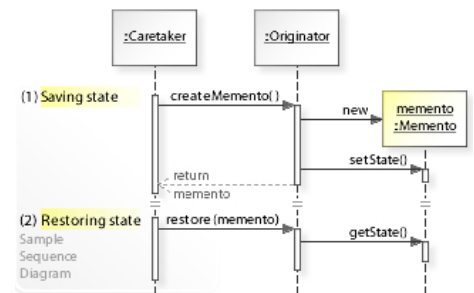
l'operazione di estrarre lo stato interno di un oggetto, senza violarne l'incapsulamento, e memorizzarlo, per poterlo poi ripristinare in un momento successivo. Tipico esempio è l'operazione di Undo, che consente di ripristinare lo stato di uno o più oggetti a come era/erano prima dell'esecuzione di una data operazione.



Il punto chiave di questo pattern è la definizione di un oggetto di tipo *memento* nel quale verrà immagazzinato lo stato di un oggetto, l'*originator*. Tale oggetto *memento* disporrà di una doppia interfaccia:

- quella verso l'*originator*, più ampia, che consentirà a questo di salvare il suo stato interno e di ripristinarlo.
- quella verso gli altri, che esporrà solamente l'eventuale distruttore.

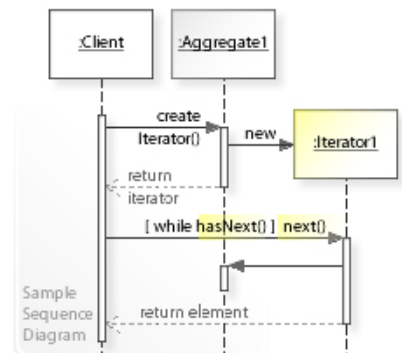
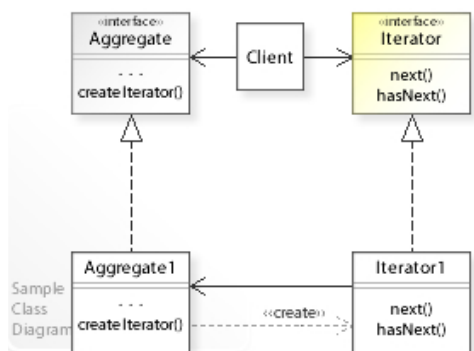
Solo l'*originator* conoscerà quindi la reale interfaccia del *memento*, e solo esso sarà in grado di istanziarlo.



Iterator [Pattern Comportamentale]

Risolve diversi problemi connessi all'accesso e alla navigazione attraverso gli elementi, in particolare, di una struttura dati contenitrice, senza esporre i dettagli dell'implementazione e della struttura interna del contenitore. L'oggetto principale su cui si basa questo *design pattern* è l'iteratore.

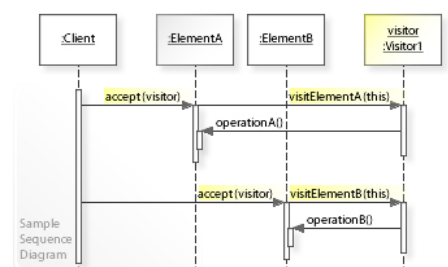
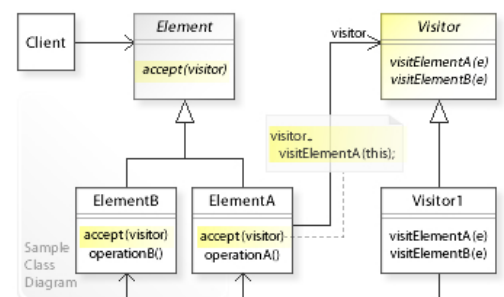
L'idea chiave di questo pattern consiste nel trasferire la responsabilità dell'accesso e della navigazione attraverso gli elementi a una classe separata dal contenitore: l'iteratore. La classe iteratore consente di visitare ad uno ad uno l'insieme degli elementi di un contenitore come se fossero ordinati in una sequenza finita.



Visitor [Pattern Comportamentale]

Permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa.

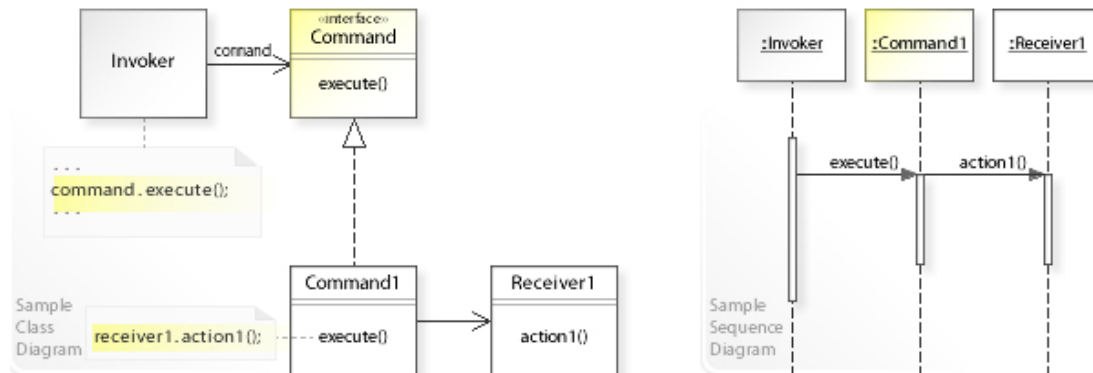
Un client che voglia utilizzare un Visitor deve creare un oggetto ConcreteVisitor e utilizzarlo per attraversare la struttura, chiamando il metodo *accept* di ogni oggetto. Ogni chiamata invoca nel ConcreteVisitor il metodo corrispondente alla classe dell'oggetto chiamante, che passa sé stesso come parametro per fornire al Visitor un punto d'accesso al proprio stato interno.



Command [Pattern Comportamentale]

Permette di isolare la porzione di codice che effettua un'azione (eventualmente molto complessa) dal codice che ne richiede l'esecuzione; l'azione è incapsulata nell'oggetto Command.

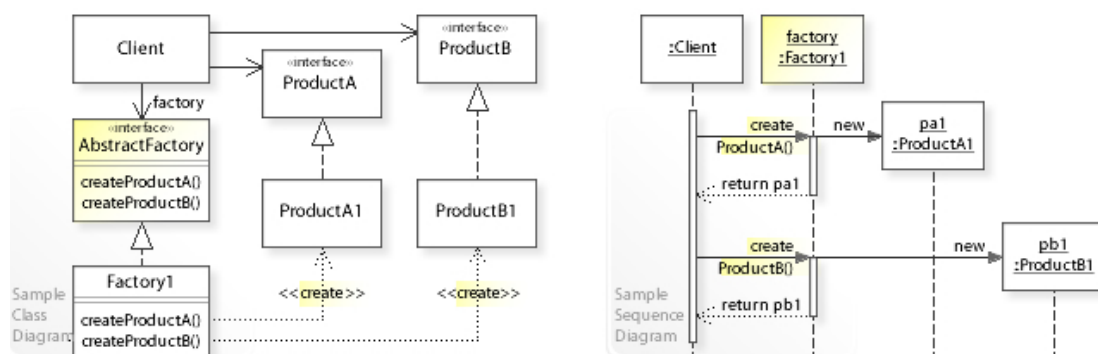
L'obiettivo è rendere variabile l'azione del client senza però conoscere i dettagli dell'operazione stessa. Altro aspetto importante è che il destinatario della richiesta può non essere deciso staticamente all'atto dell'istanziazione del command ma ricavato a tempo di esecuzione.



Abstract Factory [Pattern Creazionale]

Fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte dei client di specificare i nomi delle classi concrete all'interno del proprio codice.

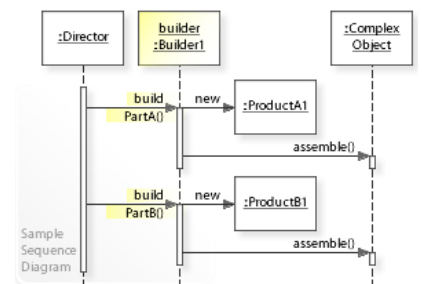
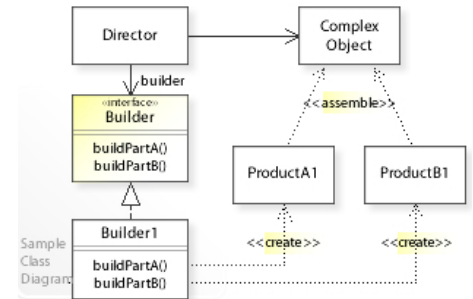
In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti.



Builder [Pattern Creazionale]

Questo pattern, nella programmazione ad oggetti, separa la costruzione di un oggetto complesso dalla sua rappresentazione cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. L'algoritmo per la creazione di un oggetto complesso è indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate.

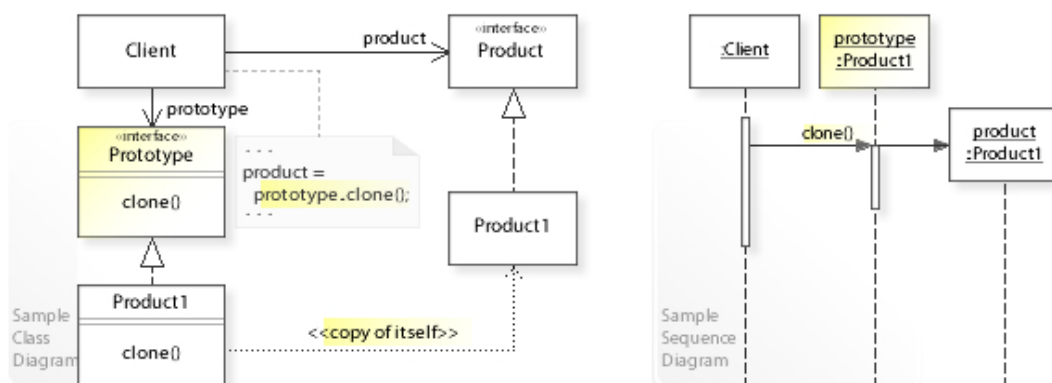
Ciò ha l'effetto immediato di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla corretta costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti. Questo è particolarmente utile quando volete assicurarvi che un oggetto sia valido prima di istanziarlo, e non volete che la logica di controllo appaia nei costruttori degli oggetti. Un builder permette anche di costruire un oggetto passo-passo, cosa che si può verificare quando si fa il parsing di un testo o si ottengono i parametri da un'interfaccia interattiva.



Prototype [Pattern Creazionale]

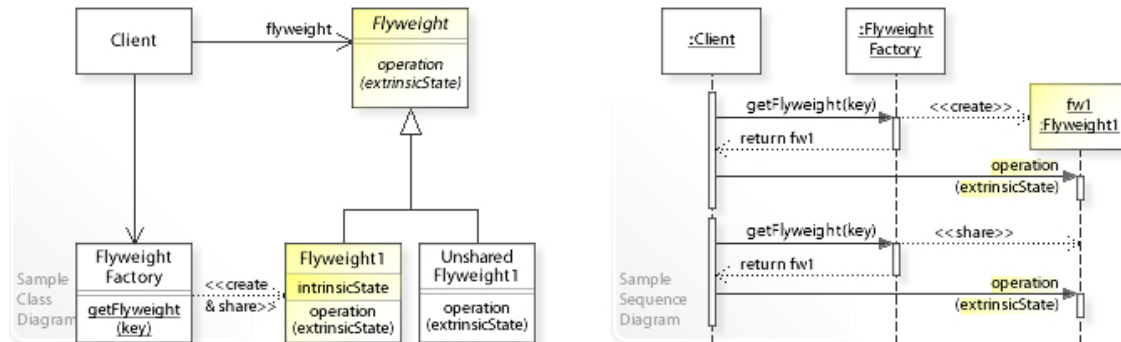
Questo pattern permette di creare nuovi oggetti clonando un oggetto iniziale, detto appunto prototipo. A differenza di altri *pattern* come Abstract factory o Factory method permette di specificare nuovi oggetti a tempo d'esecuzione (*run-time*), utilizzando un gestore di prototipi (*prototype manager*) per salvare e reperire dinamicamente le istanze degli oggetti desiderati.

Come altri *pattern* creazionali, ovvero che si occupano di istanziare oggetti, *prototype* mira a rendere indipendente un sistema dal modo in cui i suoi oggetti vengono creati.



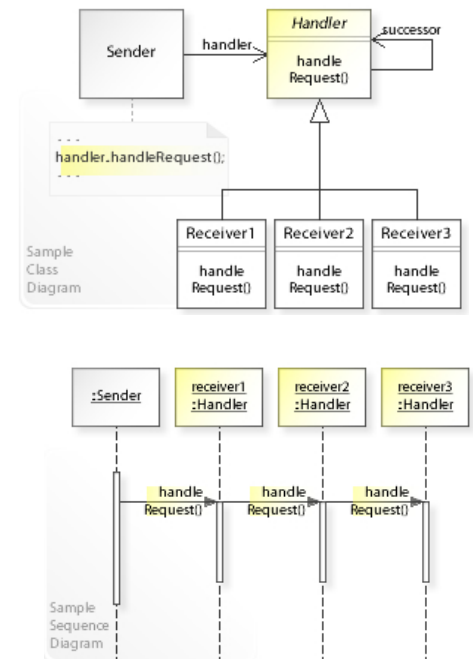
Flyweight [Pattern Strutturale]

È un pattern che permette di separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima fra differenti istanze. L'oggetto Flyweight deve essere un oggetto immutabile, per permettere la condivisione tra diversi client e thread.



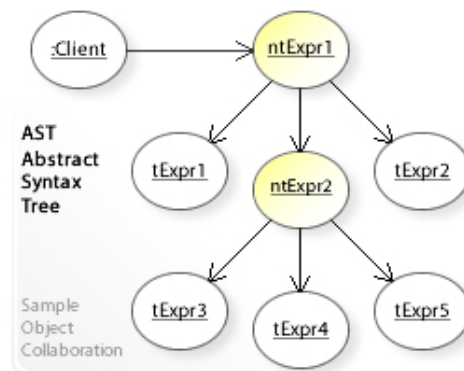
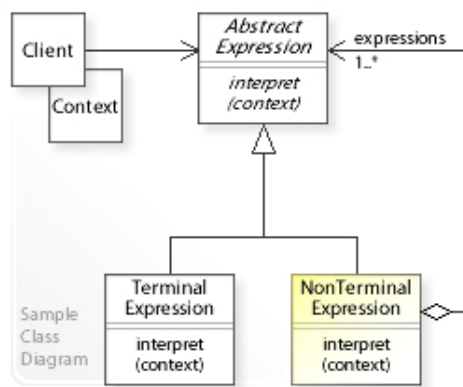
Chain of Responsibility [Pattern Comportamentale]

Il pattern 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. Evita di accoppiare il mittente di una richiesta al suo destinatario, fornendo a più di un oggetto la possibilità di gestire la richiesta. 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.



Interpreter [Pattern Comportamentale]

Questo pattern difatti precisa come valutare le frasi in una determinata lingua o linguaggio. L'idea di base è quella di avere una classe per ciascun simbolo (*terminale* o *non terminale*) in un linguaggio di programmazione specifico. L'albero sintattico di una frase nella lingua è quindi un esempio del modello sintattico composito e viene usato per valutare (interpretare) la frase.



Modern patterns

Null Object

Invece di utilizzare una referenza null per esprimere l'assenza di un oggetto, si utilizza un oggetto che implementa l'interfaccia attesa, ma in cui il corpo del metodo è vuoto, "non fa niente". Il vantaggio è che non si avranno `NullPointerException`s, ma al tempo stesso non sarà possibile controllare se un oggetto è null.

Type Object

Si delega la creazione di nuovi "tipi", è utile quando l'ereditarietà è troppo rigida o quando sarebbero richieste molte classi. È un pattern simile a Strategy e State, la differenza principale è che i Type Object possono contenere uno stato.

Extension Object

Anticipa che l'interfaccia di un oggetto necessiterà di essere estesa in futuro. Possono essere richieste agli oggetti le interfacce implementate, una classe base comune con un metodo di interrogazione può essere utilizzata da tutte le classi estensibili.

Dependency Injection

È un metodo per raggiungere l'*Inversion of Control* (IoC), ovvero una situazione in cui il comportamento di un oggetto (i metodi che chiama) è determinato da codice al di fuori dell'oggetto. Con il Dependency Injection, il metodo che chiama il costruttore della classe target gli passa anche degli oggetti, tali oggetti conterranno i metodi che saranno chiamati dall'oggetto della classe target.

Agile Software Development

I metodi agile si contrappongono al modello a cascata e altri processi software tradizionali, proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, software funzionante e di qualità. In generale i metodi agile tentano di ridurre il rischio di fallimento sviluppando il software in finestre di tempo limitate chiamate iterazioni che, in genere, durano qualche settimana. Ogni iterazione è un piccolo progetto a sé stante e deve contenere tutto ciò che è necessario per rilasciare un piccolo incremento nelle funzionalità del software: pianificazione, analisi dei requisiti, progettazione, implementazione, test e documentazione.

Manifesto dell'Agile Software Development

Stiamo scoprendo modi migliori per sviluppare software, costruendolo e aiutando gli altri a farlo. Attraverso questo lavoro siamo arrivati a valutare:

- Meglio gli individui e le interazioni rispetto ai processi e gli strumenti.
- Meglio software funzionante rispetto a documentazione comprensiva.
- Meglio collaborazione con il cliente rispetto alla negoziazione del contratto.
- Meglio rispondere tempestivamente al cambiamento rispetto al seguire un piano.

L'Agile software development non è un processo o un metodo di progettazione, piuttosto è una collezione di pratiche guidate da un insieme di principi.

Gli sviluppatori e il reparto business devono lavorare insieme al progetto ogni giorno, il software funzionante è la misura principale di progresso.

Test-Driven Design Practice

È uno stile di programmazione in cui tre attività sono strettamente intrecciate: produrre codice, testare e progettare. Si può descrivere come: scrivere una singola unità di test descrivendo gli aspetti del programma, esegui il test che dovrebbe fallire in quanto al programma manca quella feature. Scrivere abbastanza codice, il più semplice possibile, per passare il test, ristrutturare il codice (refactoring) finché non è conforme ai criteri più semplici, e infine ripetere accumulando unità di test nel tempo.

Code review

Solitamente si intende la pratica con cui codice nuovo/rivisitato deve passare una revisione prima che venga committato.

Pair Programming

consiste in due programmatori che condividono una singola workstation. Il programmatore alla tastiera è chiamato “driver”, l’altro, il “navigator”, svolge un ruolo di supervisione e di revisione simultanea del codice; Il “driver” ha l’obiettivo principale di portare a termine una soluzione funzionante del problema in considerazione, mentre al “navigator” è lasciato il compito di segnalare errori del conducente o proporre strategie alternative di soluzione. Ci si aspetta che i programmatori si scambino i ruoli ogni qualche minuto.

User stories

Sono frasi concise, scritte nel linguaggio di dominio, che catturano le aspettative dell’utente, le user stories non sono casi d’uso. Es. “Come impiegato voglio acquistare un pass per il parcheggio così posso andare al lavoro in macchina”.

Extreme Programming (XP)

Ci si basa sul fatto che essere in grado di adattarsi al cambiamento dei requisiti durante qualsiasi punto della vita del progetto è più realistico e un approccio migliore che pretendere di definire tutti i requisiti all’inizio di un progetto e poi fare sforzi per controllare i cambiamenti eventuali.

Vi sono cinque valori: comunicazione, semplicità, feedback, coraggio e rispetto; e quattro attività: coding, testing, listening e designing. Le best practices in XP sono divise in quattro gruppi: feedback a grana fine, processo continuo, comprensione condivisa e benessere del programmatore.



Scrum

Scrum è un framework di processo utilizzato per gestire lo sviluppo di prodotti complessi. Scrum non è un processo o una tecnica per costruire prodotti ma piuttosto è un framework all'interno del quale è possibile utilizzare vari processi e tecniche. Scrum rende chiara l'efficacia relativa del proprio product management e delle proprie pratiche di sviluppo così da poterle migliorare.

Sprint

Il progresso in un progetto Scrum è visto come una sequenza di sprint, uno sprint è un iterazione time-boxed (un compito da eseguire in un certo intervallo di tempo) che dura da 1 a 4 settimane. Include al suo interno cicli di design, coding e testing e termina con un incremento potenzialmente consegnabile.

Ruoli

I ruoli possono essere di tipo Core (pigs):

Product Owner - gli stakeholders del progetto, decide le priorità, le deadlines e le features, inoltre accetta o rifiuta il lavoro.

Scrum master - è una sorta di leader responsabile della corretta applicazione delle pratiche e dei principi di Scrum. In quanto facilitator è responsabile della rimozione degli ostacoli che limitano la capacità del team di raggiungere l'obiettivo dello Sprint.

Development team - è composto da 5 a 9 persone, è auto-organizzato e lavora a tempo pieno allo sviluppo del progetto.

Oppure di tipo Additional (chickens): clienti e management esecutivo.

Artefatti

Product Backlog: una lista, ordinata dal product owner, di requisiti relativi ad un prodotto. Contiene delle stime approssimative sia del valore di business che dello sforzo necessario a svilupparle (i valori sono dati secondo una successione di Fibonacci).

Sprint Backlog: una lista creata dal development team di compiti, derivata dagli elementi più importanti del product backlog, tale lista deve essere completata nello sprint corrente. I compiti sono generalmente suddivisi in attività (es. To do, In progress, Testing, ...)

Burn Down Chart: un grafico che mostra pubblicamente il lavoro da fare (backlog) su un progetto nel tempo.

Nota: la composizione del product backlog comprende: *stories* (caratteristiche descritte ad alto livello da raffinare dei progressi del progetto), *epics* (user stories molto grandi che necessitano di più di uno sprint per essere completate) e *themes* (una collezione di storie collegate tra loro).

Eventi

- *Sprint planning*: il product owner presenta gli elementi più importanti del product backlog e ognuno viene discusso in dettaglio e si stima lo sforzo richiesto. Il dev team popola lo sprint backlog e suddivide gli elementi in task.
- *Daily scrum*: un meeting di una quindicina di minuti in cui i membri del dev team rispondono a tre domande: cosa ho fatto ieri che ha aiutato il dev team a raggiungere lo sprint goal? Cosa farò oggi per aiutare il dev team a raggiungere lo sprint goal? C'è qualche impedimento che impedisce a me o al gruppo di raggiungere lo sprint goal?
- *Sprint review*: al termine di uno sprint viene fatta una revisione da parte degli stakeholders e del team, e successivamente lo scrum master e il dev team discutono dei successivi sprint.

Kanban

È un metodo di scheduling per controllare la catena di produzione per strutture just-in-time. Si utilizza un approccio di continua consegna (niente cicli), dividendo tutto il lavoro in unità e procedendo attraverso una pipeline composta di stage. Il tutto utilizzando una board per visualizzare progressi e limiti.

Le cinque proprietà Kanban sono:

- Visualizzare il flusso di lavoro
- Limitare il Work-in-Process
- Misurare e gestire il flusso
- Rendere le politiche di processo esplicite
- Utilizzare i modelli per riconoscere le opportunità di miglioramento

Testing

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.

Lo *unit testing* testa singole unità di codice (funzioni, metodi), viene usato per assicurarsi che il codice raggiunga le aspettative e che continui a raggiungerle nel tempo.

In Breve

Diagramma delle classi

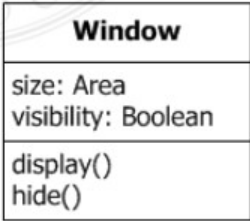
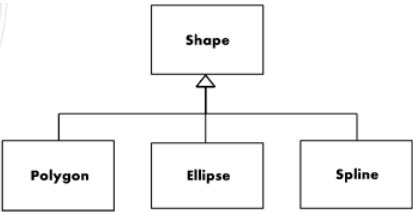

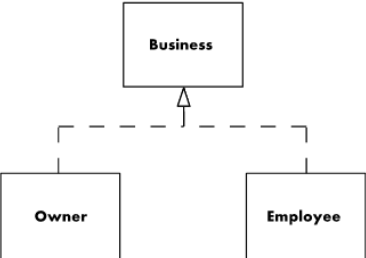
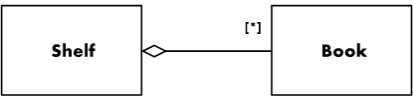
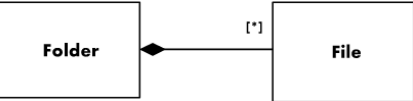
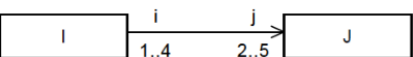
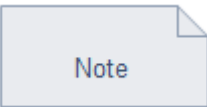
Simbolo	Nome	Note
	Classi e Interfacce	Mostrano l'architettura e le caratteristiche del sistema progettato, hanno un'area con il nome, una con gli attributi ed una con i metodi implementati.
	Generalizzazione / Specializzazione	Utilizzati quando vi è ereditarietà, ovvero la classe che punta la freccia eredita elementi dalla classe puntata.
	Dipendenza	Quando una classe ha all'interno di un metodo una referencia ad un'altra classe, dipende da quella.
	Realizzazione	Particolare tipo di dipendenza in cui la classe che punta implementa l'interfaccia, che viene puntata dalla freccia.
	Aggregazione	Usata quando una classe è composta da elementi di un'altra classe, ma quest'ultima può comunque esistere al di fuori di quell'insieme.
	Composizione	Usata quando una classe è composta da elementi di un'altra, ma quest'ultima da sola non ha senso di esistere.
	Associazione	Relazione tra due classi, può avere o meno una direzione, ed ha cardinalità.
	Nota	Usata per aggiungere note supplementari ad elementi del diagramma.

Diagramma dei casi d'uso





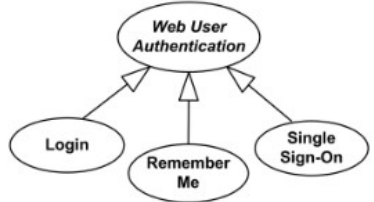
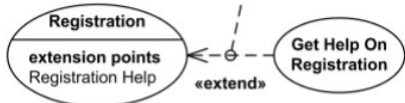
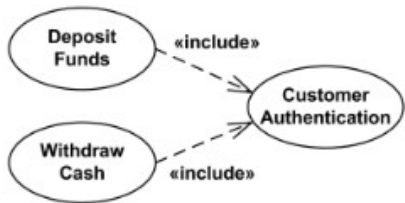
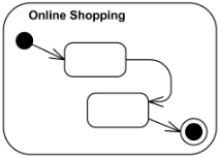
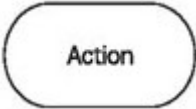
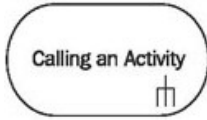
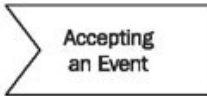



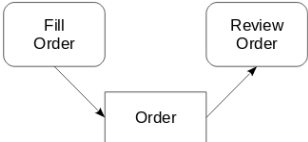
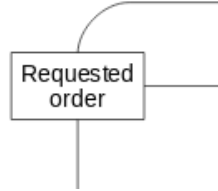
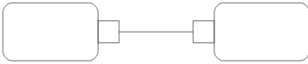
Simbolo	Nome	Note
	Attore	Un classificatore che interagisce con il sistema, può essere un utente, un ente, un'altro sistema.
	Soggetto	Il sistema in analisi, è rappresentato da un rettangolo contenente tutti i casi d'uso.
	Caso d'uso	Un comportamento che può tenere il soggetto in analisi, contiene all'interno il nome dell'azione.
	Associazione Attore - Caso d'uso	Rappresenta l'interazione che un attore può avere con un caso d'uso, ad esempio invocandone l'esecuzione.
	Generalizzazione	Indica che uno o più casi d'uso (figli) hanno elementi in comune con un'altro caso d'uso (padre) dal quale ereditano certe caratteristiche. Possono esserci generalizzazioni anche tra attori.
	Estensione	Si usa quando un caso d'uso <u>può</u> essere integrato con un altro, Sotto la voce "extension points" si indica dove può essere integrato il caso d'uso.
	Inclusione	Si usa quando un caso d'uso <u>integra</u> un altro caso d'uso al suo interno, per funzionare. Si può usare anche per evitare di ripetere il caso d'uso incluso, ogni volta.

Diagramma delle attività

Simbolo	Nome	Note
	Attività	Ha un nome che la identifica e al suo interno un insieme di azioni e altri elementi che definiscono il processo relativo all'attività.
	Azione	Un'azione è un passaggio fondamentale dell'attività, ovvero un elemento minimale che non può essere suddiviso ulteriormente. Possono avere pre e post condizioni.
	Chiamata di Attività	Un'azione che chiama un'attività dall'interno di un'altra attività.
	Evento in Arrivo	È un elemento che rimane in attesa di un certo evento, quando arriva, il flusso che parte da qui viene eseguito.
	Evento Temporale	Definisce un evento nel tempo che una volta verificatosi fa procedere il flusso da qui.
	Invio di un Segnale	Con questo elemento si vuole indicare l'invio di un segnale ad una certa attività ricevente.
	Arco	Connette gli elementi del diagramma e mostra il flusso che va seguito nell'attività.
	Oggetto	Indica che l'istanza di un classificatore è disponibile in un certo punto dell'attività. Si usa quando si vuole indicare che quell'istanza è completa e utilizzabile.
	Parametro di Attività	Elemento all'inizio o alla fine di un'attività che fornisce un modo per accettare <u>input</u> o <u>output</u> all'attività.
	Pin di Input / Output	Elementi che permettono l'invio di output o la ricezione di input tra azioni.

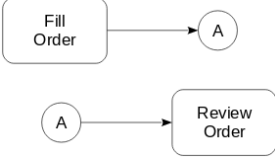
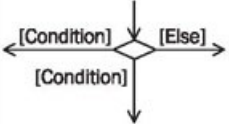
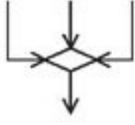
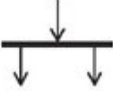
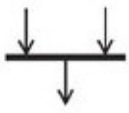




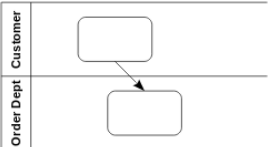
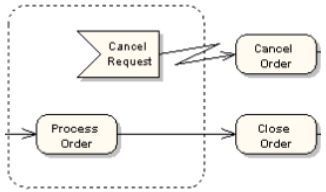
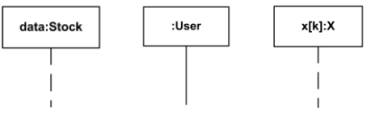
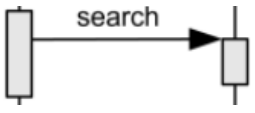
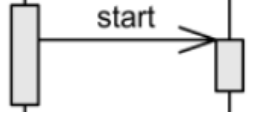
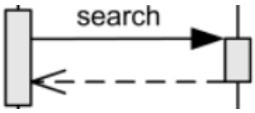


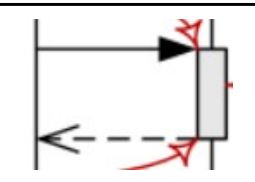
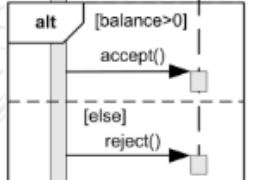
	Connettori	Una sorta di teletrasporto, utilizzati per evitare di usare archi molto lunghi che creerebbero confusione.
	Decisione	Tramite gli elementi tra quadre viene determinato tramite guardie, quale percorso prendere. Ha 1 input e 2 o più output.
	Merge	Unisce vari flussi provenienti da diversi elementi. Ha molti input e un unico output.
	Fork	Suddivide il flusso in due o più flussi <u>paralleli</u> utilizzando una barra di sincronizzazione.
	Join	Riunisce due o più flussi paralleli in uno unico, utilizzando una barra di sincronizzazione.
	Nodo Iniziale	Elemento da cui inizia il flusso all'interno dell'attività. È possibile non esista questo elemento ma vi sia solo l'attesa di un evento ad iniziare l'attività.
	Nodo Finale	Elemento che indica che un'attività è completata. Possono essere presenti più nodi finali in un'attività.
	Nodo Finale di Flusso	Questo elemento termina un flusso, non l'intera attività, infatti in caso di più flussi paralleli solo quello che raggiunge questo nodo terminerebbe.
	Attività Strutturata	Sono nodi all'interno di un'attività che contengono altri nodi. Un nodo strutturato può essere contenuto da altri nodi strutturati. (alcuni tipi sono nodi <i>condizionali</i> e nodi <i>loop</i>).
	Partizione	Gli elementi all'interno di un diagramma di attività possono essere divisi in aree individuali o "partizioni".
	Regione Interrompibile	Con questo elemento si ha la possibilità di terminare tutti i comportamenti all'interno della regione nel caso in cui il flusso esca dalla sezione attraverso il fulmine. Usato per applicare il concetto "tutto-o-niente".

Diagramma di sequenza

Simbolo	Nome	Note
	Lifeline	Rappresenta un partecipante nell'interazione, di solito si nomina con "nomeElem:TipoObj" può essere un rettangolo o un'icona.
	Chiamata Sincrona	Con questa chiamata si blocca il flusso del partecipante che la invia, e riprenderà solo dopo che avrà ricevuto una risposta.
	Chiamata Asincrona	Con questa chiamata il flusso del partecipante che la genera non viene bloccato, non è importante la ricezione di una risposta.
	Risposta	La linea tratteggiata di ritorno mostra la risposta alla precedente chiamata effettuata.
	Creazione	Con questa chiamata si crea un nuovo partecipante, che prima non aveva senso di esistere, di solito si aggiunge la voce <<create>>.
	Distruzione	Effettuata quando si vuole eliminare la presenza di un partecipante nell'interazione, tale chiamata ha la voce <<destroy>> e una X alla fine.
	Gates	Quando si riceve in input una chiamata, o si manda in output una risposta, all'esterno dell'interazione; ovvero non vi è apparentemente un partecipante a riceverla.
	Frammento	Un frammento può fare tante cose, ad esempio fungere da if-else, essere invocato come un operation, loop o distinguere in base ai parametri ricevuti cosa eseguire.

I diagrammi di comunicazione essenzialmente usano anche loro dei partecipanti (come rettangoli o icone) ma per mostrare l'andamento della comunicazione si usano piccole frecce lungo i collegamenti che indicano la direzione del messaggio, con un numero che indica l'ordine in cui tale messaggio è inviato.

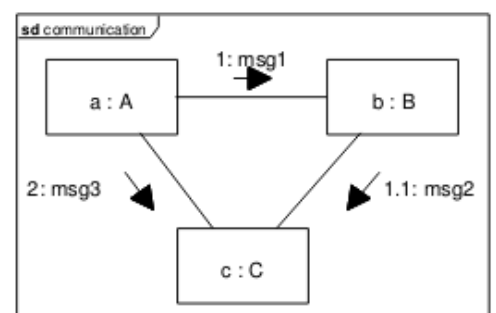

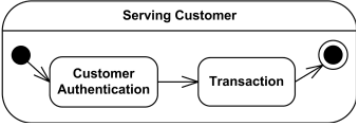

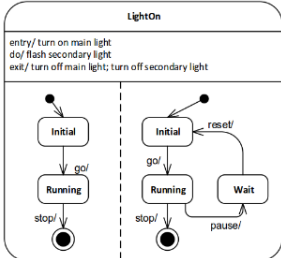

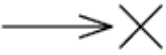
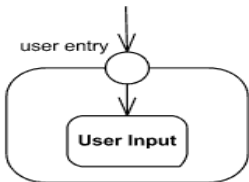
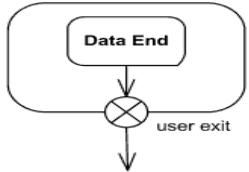
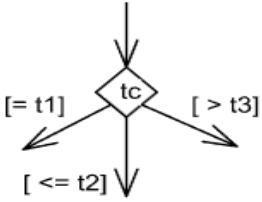
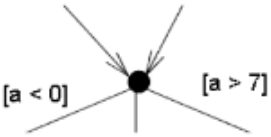
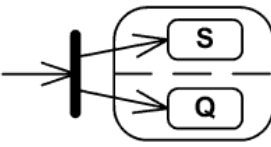
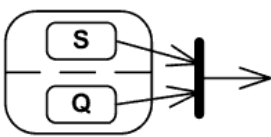



Diagramma di stato

Simbolo	Nome	Note
	Stato Semplice	È uno stato che non ha sotto-stati, non ha regioni né stati submachine. Può avere le voci “entry”, “do” ed “exit” che indicano i comportamenti eseguiti dallo stato nelle rispettive situazioni.
	Stato Composto	È uno stato che sotto-stati (annidati), talvolta le relazioni tra i sottostati vengono nascoste e si aggiunge semplicemente il simbolo:  .
	Regioni	Le regioni sono insiemi di stati e transizioni posti in modo concorrente (ortogonale) all'interno di uno stato composto.
	Pseudostato Iniziale	È un vertice che rappresenta una singola transazione allo stato di default.
	Pseudostato Terminale	Ha la funzione di terminare lo stato che lo invoca, non termina tutta la state machine.
	Pseudostato Punto di Entrata	È il punto di entrata di una state machine o di uno stato composto.
	Pseudostato Punto di Uscita	È il punto di uscita di una state machine o di uno stato composto, ha la funzione di terminare completate la state machine in questione.
	Pseudostato Scelta	Tramite le guardie tra parentesi quadre, il flusso viene incanalato in una certa direzione.

	Pseudostato Giunzione	Unisce o divide più transizioni in più transizioni in base a delle guardie.
	Pseudostato Fork	Permette di dividere una transizione in entrata in due o più transizioni terminanti in una regione (o stato composto) parallela.
	Pseudostato Fork	Unisce più transizioni provenienti da una regione ortogonale in un'unica transizione.
	Stato Finale	Particolare tipo di stato che indica che la regione in cui è contenuto è completata.

La forma in cui viene espressa una *transizione* è importante:

transition ::= [**triggers**] [**guard**] ['/' **behavior-expression**]

In cui:

triggers ::= trigger [' ' trigger]*

guard ::= '[' constraint ']'

behavior-expression ::= do-something

Ancora:

trigger ::= call-event | signal-event | any-receive-event | time-event | change-event

constraint ::= '{' [name ':'] boolean-expression '}'

Esempio completo:

[**triggers**] [**guard**] ['/' **behavior-expression**]

left-mouse-down(coordinates) [**coordinates** == '324;954'] / **select-link(coordinates).follow()**

Pattern

Nome	Tipo	In breve	Descrizione	Implementazione
Abstract Factory	Creaz	Crea un'istanza di diverse famiglie di classi	Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. Una gerarchia che incapsula: molte possibili "piattaforme" e la costruzione di una suite di "prodotti". L'operatore <i>new</i> è ritenuto dannoso.	Si istanzia una AbstractFactory e si crea ogni ConcreteFactory come Singleton in modo che ne esista solo un'istanza a run-time. Dato che AbstractFactory definisce solo l'interfaccia saranno le classi ConcreteFactory a creare i prodotti.
Builder	Creaz	Separa la costruzione dell'oggetto dalla sua rappresentazione	Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa creare rappresentazioni differenti. Analizza una rappresentazione complessa e crea una delle varie sagome.	Il Client crea un oggetto Director e lo configura con gli oggetti Builder desiderati. Il Director notifica al Builder se una parte del prodotto deve essere costruita, il Builder riceve le richieste dal Director e aggiunge le parti al prodotto. Il Client riceve il prodotto dal Builder.
Factory Method	Creaz	Crea un'istanza di varie classi derivate	Definisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale classe istanziare. Questo metodo consente a una classe di delegare l'istanziamento alle sottoclassi. Definisce un costruttore "virtuale". L'operatore <i>new</i> è ritenuto dannoso.	Si crea una classe A che nel costruttore utilizza un metodo <code>doSomething()</code> presente in A. La classe B ora, estendendo A, potrà effettuare l'override di <code>doSomething()</code> e far sì che il codice in A sia valido ma con le specifiche che desidera B.
Prototype	Creaz	Un'istanza completamente inizializzata da copiare o clonare	Specifica i tipi di oggetti da creare utilizzando un'istanza prototipo e crea nuovi oggetti copiando questo prototipo. Delega un'istanza di una classe da utilizzare come generatore di tutte le istanze future. L'operatore <i>new</i> è ritenuto dannoso.	Il linea generale il client crea un nuovo oggetto del tipo desiderato chiedendo a un prototipo di clonarsi, ovvero invocando il metodo <code>clone</code> definito dal <code>ConcretePrototype</code> di cui si vuole la copia.
Singleton	Creaz	Una classe di cui può esistere solo una singola istanza	Assicura che una classe abbia una sola istanza e fornisce un accesso globale ad essa.	Una classe singleton ha un unico costruttore privato, in modo da impedire l'istanziamento diretta della classe e un metodo "getter" che restituisce sempre se stessa.
Adapter	Strutt	Abbina interfacce di classi diverse	Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano. Adapter	Questo pattern può basarsi su classi, utilizzando l'ereditarietà multipla per adattare interfacce

			consente alle classi di funzionare insieme che non potrebbero altrimenti a causa di interfacce incompatibili. Impacchetta una classe esistente con una nuova interfaccia.	diverse tramite ereditarietà, oppure può basarsi sulla composizione di oggetti in cui all'interno si hanno metodi che si interfacciano con elementi con interfacce diverse.
Bridge	Strutt	Separa l'interfaccia di un oggetto dalla sua implementazione	Disaccoppia un'astrazione dalla sua implementazione in modo che i due possano variare in modo indipendente. Pubblica l'interfaccia in una gerarchia di ereditarietà e nasconde l'implementazione nella propria gerarchia di ereditarietà.	Un interruttore che controlla luci, ventilatori, ecc. è un esempio di Bridge. Si avrà una classe Switch che gestisce l'ON e OFF a livello elettrico e più SwitchImplementation che gestiscono l'ON e OFF dei vari dispositivi presenti.
Composite	Strutt	Una struttura ad albero di oggetti semplici e composti	Compone oggetti in strutture ad albero per rappresentare parti intere di gerarchie. Questo pattern consente ai client di trattare in modo uniforme singoli oggetti e composizioni di oggetti. "Le directory contengono voci, ognuna delle quali potrebbe essere una directory."	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.
Decorator	Strutt	Aggiunge responsabilità agli oggetti in modo dinamico	Allega dinamicamente ulteriori responsabilità a un oggetto. I decorator offrono un'alternativa flessibile alla sottoclasse per estendere le funzionalità. Avviene un abbellimento specifico di un oggetto core specificato dal cliente avvolgendolo in modo ricorsivo. "Avvolgere un regalo, metterlo in una scatola e avvolgere la scatola."	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.
Facade	Strutt	Una singola classe che rappresenta un intero sottosistema	Fornisce un'interfaccia unificata a un insieme di interfacce in un sottosistema. Facade definisce un'interfaccia di livello superiore che semplifica l'utilizzo del sottosistema. Impacchetta un sottosistema complicato con un'interfaccia più semplice.	Il consumatore chiama un numero e parla con un rappresentante del servizio clienti. Il rappresentante del servizio clienti funge da facciata, fornendo un'interfaccia al reparto di evasione ordini, al reparto fatturazione e al reparto spedizioni.
Flyweight	Strutt	Un'istanza a grana fine utilizzata per una condivisione efficiente	Usa la condivisione per supportare un numero elevato di oggetti a grana fine in modo efficiente. La strategia Motif GUI di sostituire i widget heavyweight con gadget leggeri.	Quando il browser carica una pagina Web, attraversa tutte le immagini su quella pagina, il browser carica le nuove immagini da Internet e le posiziona nella cache interna. Per le immagini già caricate, viene creato un oggetto flyweight, che ha alcuni dati

				univoci come la posizione all'interno della pagina, ma per tutto il resto viene fatto riferimento a quello memorizzato nella cache.
Proxy	Strutt	Un oggetto che rappresenta un altro oggetto	Fornisce un surrogato o segnaposto per un altro oggetto per controllarne l'accesso. Utilizza un livello aggiuntivo di riferimento indiretto per supportare l'accesso distribuito e controllato. Aggiunge un involucro per proteggere il componente reale da una complessità eccessiva.	Un assegno è un proxy per i fondi in un conto. Un assegno può essere utilizzato al posto del contante per effettuare acquisti e in definitiva controlla l'accesso al contante nel conto dell'emittente.
Chain of Responsibility	Comp	Un modo per passare una richiesta tra una catena di oggetti	Evita di accoppiare il mittente di una richiesta al suo ricevitore dando a più di un oggetto la possibilità di gestire la richiesta. Incatena gli oggetti riceventi e passa la richiesta lungo la catena finché un oggetto non la gestisce. Avvia e lascia le richieste con una singola pipeline di elaborazione che contiene molti possibili gestori.	Un esempio è nel caso di un processo di acquisto in cui si hanno differenti ruoli ognuno con un limite massimo per un acquisto e un successore. Ogni volta che una persona (che ha un determinato ruolo) riceve un ordine di acquisto che sorpassa il proprio limite, passa la richiesta al successore nella catena di comando.
Command	Comp	Incapsula una richiesta di comando come oggetto	Incapsula una richiesta come oggetto, consentendo in tal modo di parametrizzare i client con richieste diverse, accodare o registrare richieste e supportare operazioni annullabili. Promuove "l'invocazione di un metodo su un oggetto" allo stato dell'oggetto completo. Una callback object-oriented.	Il conto al ristorante è un esempio di pattern Command. Il cameriere o la cameriera prende un ordine o comando da un cliente e incapsula quell'ordine scrivendolo sull'assegno. L'ordine viene quindi accodato per negli ordini di cottura. Si noti che il foglietto del conto utilizzato da ciascun cameriere non dipende dal menu e quindi può supportare i comandi per cucinare molti articoli diversi.
Interpreter	Comp	Un modo per includere elementi linguistici in un programma	Dato un linguaggio, definisce una rappresentazione per la sua grammatica insieme a un interprete che usa la rappresentazione per interpretare le frasi nella lingua. Mappare un dominio in una lingua, la lingua in una grammatica e la grammatica in un progetto gerarchico orientato agli oggetti.	I musicisti sono esempi di interpreti. Il tono di un suono e la sua durata possono essere rappresentati in notazione musicale su un pentagramma. Questa notazione fornisce il linguaggio della musica. I musicisti che suonano la musica dalla partitura sono in grado di riprodurre l'intonazione e la durata originali di ciascun suono rappresentato.
Iterator	Comp	Accede in modo sequenziale agli	Fornire un modo per accedere sequenzialmente agli elementi di	I file sono oggetti aggregati. Negli uffici in cui l'accesso ai file avviene

		elementi di una raccolta	un oggetto aggregato senza esporre la sua rappresentazione sottostante.	tramite personale amministrativo o di segreteria, tale modello mostra come il segretario che agisca da Iterator.
Mediator	Comp	Definisce una comunicazione semplificata tra le classi	Definisce un oggetto che incapsula il modo in cui un insieme di oggetti interagisce. Il mediatore promuove l'accoppiamento libero impedendo esplicitamente agli oggetti di riferirsi l'un l'altro e consente di variare la loro interazione in modo indipendente. Progetta un intermediario per disaccoppiare molti soggetti "colleghi".	La torre di controllo di un aeroporto dimostra bene questo schema. I piloti degli aerei che si avvicinano o si allontanano dall'area terminale comunicano con la torre anziché comunicare esplicitamente l'uno con l'altro. I vincoli su chi può decollare o atterrare sono imposti dalla torre. La torre non controlla l'intero volo, esiste solo per far rispettare i vincoli nell'area terminale.
Memento	Comp	Cattura e ripristina lo stato interno di un oggetto	Senza violare l'incapsulamento, acquisisce ed esterna lo stato interno di un oggetto in modo che possa essere restituito a questo stato in un secondo momento. Un cookie magico che incapsula una capacità di "check point". Promuove l'annullamento o il rollback allo stato dell'oggetto completo.	Quando un ci si ripara da soli i freni a tamburo, si rimuovono da entrambi i lati i tamburi, mostrando sia il freno destro che quello sinistro. Si smonta solo un lato e l'altro funge da Memento di come le parti del freno devono essere. Solo dopo che il lavoro è stato completato da un lato, l'altro lato viene smontato. Ora sarà il primo lato a fungere da Memento.
Observer	Comp	Un modo per notificare un cambiamento ad un certo numero di classi	Definisce una dipendenza uno-a-molti tra gli oggetti in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente. Incapsula i componenti principali (o comuni) in un'astrazione del soggetto e i componenti variabili (o facoltativi, o dell'interfaccia utente) in una gerarchia di Observer. La parte "View" di Model-View-Controller.	Alcune aste dimostrano questo modello. Ogni offerente possiede una pagaia numerata che viene utilizzata per indicare un'offerta. Il banditore inizia l'offerta e "osserva" quando viene sollevata una pagaia per accettare l'offerta. L'accettazione dell'offerta modifica il prezzo dell'offerta che viene trasmesso a tutti gli offerenti sotto forma di nuova offerta.
State	Comp	Altera il comportamento di un oggetto quando il suo stato cambia	Permette ad un oggetto di alterare il proprio comportamento quando il suo stato interno cambia. L'oggetto sembrerà cambiare la sua classe. Una macchina a stati orientata agli oggetti.	Questo modello può essere osservato in un distributore automatico. I distributori automatici hanno stati basati sull'inventario, sull'ammontare della valuta depositata, sulla possibilità di apportare modifiche, sulla voce selezionata, ecc. Quando viene immesso il denaro e viene effettuata una selezione, o il distributore consegna il prodotto e non cambia, o consegna il prodotto e cambia, o non consegna prodotti

				a causa di insufficiente valuta in deposito o non consegna prodotti a causa di esaurimento delle scorte.
Strategy	Comp	Incapsula un algoritmo all'interno di una classe	Definisce una famiglia di algoritmi, incapsula ciascuno e li rende intercambiabili. La strategia consente all'algoritmo di variare in modo indipendente dai client che lo utilizzano. Cattura l'astrazione in un'interfaccia e nasconde i dettagli di implementazione nelle classi derivate.	Le modalità di trasporto verso un aeroporto sono un esempio di strategia. Le opzioni sono la propria auto, un taxi, una navetta, un autobus, ecc. Per alcuni aeroporti, ci sono anche metropolitane ed elicotteri. Ogni modalità avrà un viaggiatore, e possono essere intercambiabili. Il viaggiatore deve scegliere in base al trade-off tra costi e tempo.
Template Method	Comp	Rinvia i passaggi esatti di un algoritmo a una sottoclasse	Definisce lo scheletro di un algoritmo in un'operazione, rinviando alcuni passaggi alle sottoclassi del client. Questo pattern consente alle sottoclassi di ridefinire determinati passaggi di un algoritmo senza modificare la struttura dell'algoritmo. La classe base dichiara l'algoritmo "segnaposto" e le classi derivate implementano i segnaposto.	I costruttori di case usano questo metodo quando sviluppano una nuova suddivisione. Una tipica suddivisione consiste in un numero limitato di piante del pavimento con diverse varianti disponibili per ciascuna. All'interno di una pianta, la fondazione, l'incorniciatura, l'impianto idraulico e il cablaggio saranno identici per ogni casa. La variazione viene introdotta nelle fasi successive della costruzione per produrre una più ampia varietà di modelli.
Visitor	Comp	Definisce una nuova operazione in una classe senza modifiche	Rappresenta un'operazione da eseguire sugli elementi della struttura di un oggetto. Visitor consente di definire una nuova operazione senza modificare le classi degli elementi su cui opera. La tecnica classica per recuperare le informazioni perse sul tipo. "Fai la cosa giusta in base al tipo di due oggetti".	Questo pattern può essere osservato nel funzionamento di una compagnia di taxi. Quando una persona chiama una compagnia di taxi (accettando un visitatore), la compagnia invia un taxi al cliente. Entrando nel taxi il cliente, o il visitatore, non ha più il controllo del proprio trasporto, il taxi (autista) ce l'ha.