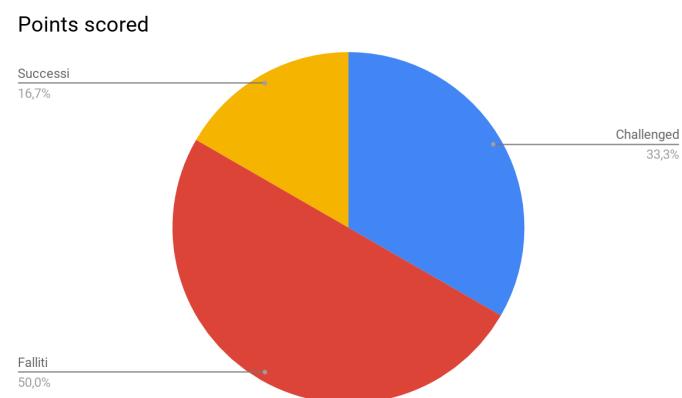


Ingegneria software - definizioni

Il problema fondamentale dell'ingegneria software è che i progetti software hanno come Artefatto ultimo un sistema software che molto spesso risulta fallimentare. E il fallimento si verifica in percentuali più alte rispetto ad altri progetti il cui prodotto è diverso. La causa è dovuta alla complessità della progettazione stessa infatti ci sono parti di prodotti software dinamici e in evoluzione.

Tanto che alla fine degli anni sessanta si verificò una vera e propria crisi del software: le risoluzioni non erano semplici i sistemi software sempre più complessi.

È possibile calcolare anche un tasso di fallimento noto come: standish Chaos Report. esso si occupa Di raccogliere l'andamento dei progetti software. Nella maggior parte dei casi i progetti software sono fallimentari è solo poco più di un quarto è venduto ed è efficiente (i progetti



Challenged sono invece quelli effettivamente realizzati ma con alti tempi di sviluppo oppure alti costi). In generale progetti molto più grandi sono stati successful solo nel 2% dei casi, contro invece il 62% dei successi i piccoli progetti (Maggiore ne è la dimensione minore è la probabilità di successo).

Sì parla per la prima volta dell'ingegneria software come una disciplina nel 1969 da parte della NATO (Conference on software Engineering). in questa discussione si cercò di stabilire una disciplina capace di abbassare la percentuale di fallimento dei progetti E che soprattutto facesse in modo da farli realizzare on Time e on budget.

Nei primi anni 70 era chiaro che giusto modo di sviluppare software dovesse necessariamente richiedere molto più che i principi di scienza del computer ma serviva anche rapporto delle tecniche analitiche e descrittive il rigore della disciplina dell'ingegneria.

Le definizioni di ingegneria software sono in realtà tantissime:

- IEEE 90: L'applicazione di un approccio sistematico disciplinato e quantificabile allo sviluppo, alla costruzione e al mantenimento di un software.
- Parnas 78: è la costruzione multipersonale di un software multi versionale

Da queste due definizioni anche se approssimative emerge che: i sistemi evolvono nel tempo di conseguenza il processo della creazione deve seguire Questa stessa evoluzione; non si lavora da soli, ma serve un team ben coordinato.

- Bauer 72: L'applicazione dei metodi o principi dell'ingegneria per realizzare un software che sia corretto nel minor tempo possibile e col Minor utilizzo di denaro (perfetto, economico e veloce da realizzare).
- CMU 90: Una forma di ingegneria che applica i principi dell'informatica e della matematica per raggiungere soluzioni di alto livello a problemi software.

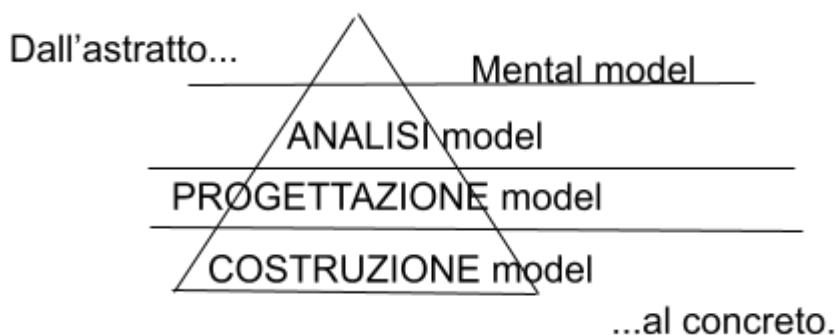
Queste definizioni chiaramente evidenziano che ingegneria software deve creare software di alta qualità in maniera sistematica, controllata ed efficiente. Il software inoltre è intangibile sì, è un Artefatto che manca di continuità (ha un'infinità di Stati e non passa linearmente tra questi).

La complessità necessita di semplificazioni

Dato che bisogna necessariamente affrontare una complessità Superiore è più facile utilizzare alcuni strumenti per semplificare il problema di partenza.

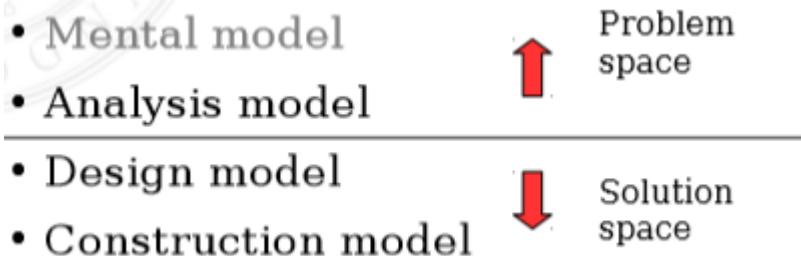
1. Astrazioni = Processo di formulazione generalizzata di idee o concetti estraendo qualità comuni da specifici esempi
2. Modelli = Il modello rappresenta la versione semplificata del sistema che risponde a domande date, così come risponderebbe il sistema analizzato (il modello è un falso creato ad arte).

Possiamo vedere il processo di astrazione come una piramide che varia per stadi di astrazione:



Questo è un processo di raffinamento che aggiunge mano a mano sempre più informazioni. Infatti possiamo vedere il processo software come un insieme di modelli a partire da quello mentale a quello di costruzione.

Si raccolgono in due parti questi modelli: modelli che operano nello spazio del problema i modelli che operano nello spazio della soluzione.



Modello di analisi

Il primo modello serve a rispondere alla domanda di che cosa faccia o debba fare precisamente il software, comunicarlo al team di sviluppo e a chi ha commissionato il software stesso.

si dice "elicitare" dal modello mentale Quali sono i compiti del sistema software.

una volta che il software è stato costruito Bisogna definire un insieme di requisiti per validare effettivamente il prodotto finale.

Il problema legato all'elicitazione è il linguaggio naturale, Esso infatti è ambiguo, Per questo si costruiscono a sostegno dell'interpretazione degli artefatti specifici (documentazione, casi d'uso, entità che compongono il progetto - report licenze, glossario).

I requisiti di un sistema possono anche essere standardizzati (come quelli forniti dallo IEEE), e sono così importanti da essere essi stessi una branca dell'ingegneria (Requirements engineering): passaggi fondamentali

- elicitazione
- analisi
- specifiche
- validazione

I requisiti Si possono dividere in due grandi famiglie:

1. i requisiti funzionali - sono quei requisiti che descrivono le interazioni tra il sistema e il suo ambiente, indipendentemente dall' implementazione. Per esempio premo sul bottone = e questo mi dà un risultato. sono requisiti indirizzati al cosa dobbiamo fare come sistema software.
2. i requisiti non funzionali - sono le proprietà del sistema che non sono direttamente collegate ai suoi aspetti funzionali. per

esempio Se premo su un qualsiasi bottone il risultato deve uscire in 3 secondi. Sono indirizzati al come il sistema dovrebbe farlo.

26/09/2019

Avevamo detto la scorsa volta che un modello è una rappresentazione astratta della realtà. i modelli sono usati per trovare delle risposte attraverso punti di vista diversi. Per sopperire al linguaggio formale che potrebbe nascondere difficoltà nella comprensione, si utilizzano dei linguaggi (=insieme di segni per rappresentare le entità nei modelli e le loro relazioni) visuali standardizzati come per esempio l'['O-O modeling](#) e di conseguenza [UML](#).

O-O principi:

L'Object Oriented Model è un paradigma che sposta l'attenzione dall'analisi e dalla progettazione degli algoritmi e dati, sugli oggetti. Gli oggetti sono qui intesi come entità autonome con un loro stato e un proprio comportamento. I principi del modello O-O sono:

1. astrazione
2. encapsulamento
3. ereditarietà
4. polimorfismo

1. Con l'astrazione mi focalizzo sulle caratteristiche essenziali: sulle variabili, sui metodi, ma non su cosa c'è al loro interno e come operano.
2. L'encapsulamento serve per esprimere l'astrazione: ovvero nascondere i dettagli sullo stato e sui comportamenti dell'oggetto, per esempio le variabili di istanza.

3. I comportamenti degli stati possono essere specializzati in particolare posso definire nuove istanze, ma questo modificando stato e comportamento. In sostanza una classe eredita stati e comportamenti e può aggiungerne di nuovi.

In particolare: ereditarietà significa ri-usare un oggetto esistente o una classe esistente specializzandone i comportamenti (override).

4. Il comportamento dipende da chi tu sia, è un concetto infatti strettamente collegato all'ereditarietà.

In particolare: quando un comportamento viene invocato su un oggetto, il risultato dipende dal tipo dell'oggetto, non dal tipo della referenza.

```
Public class Animale      Public abstract Class Cane extends Animale  
public string toString()...      public string toString()....  
Animale Jeff = new Cane(..);  
Jeff.toString(); -> questo stampa il toString di Cane non di Animale.
```

Intorno a questi quattro principi si costruiscono i modelli object-oriented, noi utilizziamo il linguaggio visuale **UML**.

Model language UML

È un linguaggio per i sistemi software, ed è un linguaggio grafico, semi formale (ha delle regole sintattiche -per creare diagrammi corretti- e semantiche - per creare diagrammi sensati-) usato per specificare, visualizzare, costruire e documentare artefatti (=qualunque cosa venga prodotto da un processo software). Esso assume un approccio O-O, inglobando tutte i quattro concetti precedenti ed essi vengono utilizzati sia per l'analisi sia per la progettazione della soluzione.

UML conta di 13 tipi di diagrammi appartenenti a due famiglie diverse:

- strutturali (modelli che rispondono alle domande sulla struttura)
- comportamentale (modelli che rispondono alle domande sul comportamento del sistema) -> **UML: USE CASE***

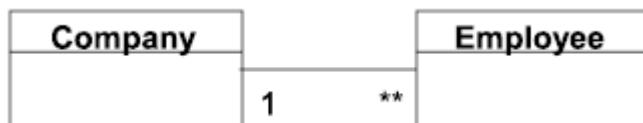
UML include primitive che si differenziano per i tipi che ammettono al loro interno in particolare:

- classifiers (set di oggetti) = classi
- events (set di occorrenze)
- behaviors (set di esecuzioni)

Gli elementi ammessi nelle primitive distinguono i diversi tipi di diagramma.

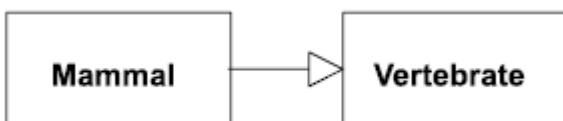
Le relazioni correlano due o più elementi in un modello: sono rappresentate come delle linee e possono avere dei nomi. I quattro principali tipi di relazione sono:

- 1) Associazione: È una relazione strutturale tra elementi che mostrano come un oggetto di una classe (classifier) sia connesso e possa navigare verso un oggetto di un'altra classe



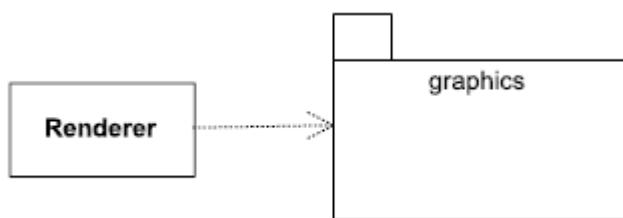
Nell'esempio sopra ci sono più impiegati **(in particolare due) che possono essere associati ad 1 sola compagnia.

- 2) Generalizzazione: Indica l'ereditarietà, in particolare che un elemento figlio, è basato su un elemento genitore, si può leggere la relazione esistente tra questi due elementi come "è un".



Nell'esempio sopra un mammifero è un vertebrato.

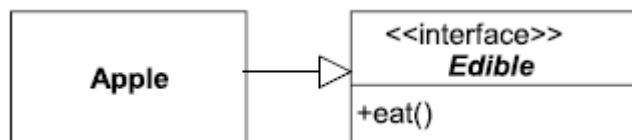
- 3) Dipendenza: Indica che i cambiamenti ad un elemento possono causare a loro volta dei cambiamenti in un altro elemento di un altro modello.



Nell'esempio sopra riportato eventuali cambiamenti all'interno del grafico portano a cambiamenti all'interno del render finale:
attenzione alla dipendenza!

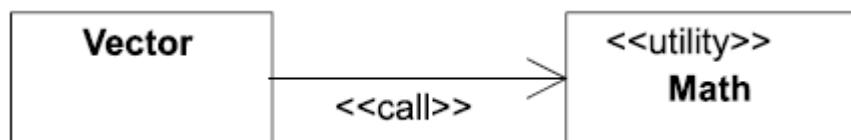
Il quadrato sopra il grafico indica il package.

- 4) Realizzazione: questa relazione esiste tra 2 elementi quando uno di questi realizza, o implementa, il comportamento che l'altro specifica.



Nell'esempio la mela si impegna a realizzare il metodo dell'interfaccia.

È possibile estendere il vocabolario di UML per creare nuovi elementi, derivandoli da quelli esistenti, ma con specifiche proprietà: questo è possibile grazie agli stereotipi. Lo stereotipo e' un nome racchiuso da guillemets <>>



In questo esempio sto arricchendo la dipendenza di chiamata.

Non tutto è modellabile con UML, esso viene arricchito con OCL.

01/10/2019

Software process model

Per arrivare allo sviluppo di un prodotto finale è necessario creare degli incrementi: gestire quindi le risorse del progetto (persone ecc). Come si può organizzare il lavoro, tra i membri del proprio Team, muovendosi tra tutte le attività che hanno tra loro dipendenze causali (un elemento dipende dai precedenti).

Processo = è un insieme di attività che vengono coordinate per raggiungere un obiettivo, nel nostro caso la produzione, evoluzione,

installazione e mantenimento di un software, in un contesto con dei vincoli rappresentati da costi, tempi e risorse. L'obiettivo è di controllare il processo cercando di affrontarne i rischi.

La natura delle attività da coordinare vengono divise in cinque gruppi chiamate ciclo di vita del software:

- 1) Attività di analisi(raccolta dei requisiti, Ovvero le funzionalità richieste)
- 2) Progettazione
- 3) Implementazione
- 4) Validazione (Ovvero se la nostra implementazione è corretta)
- 5) Evoluzione (mantenere aggiornate le funzionalità)

Tutte queste attività producono diversi tipi di output:

- codice
- modelli di analisi (documenti)
- modelli di progetto (diagrammi)
- prototipi, Report..ecc

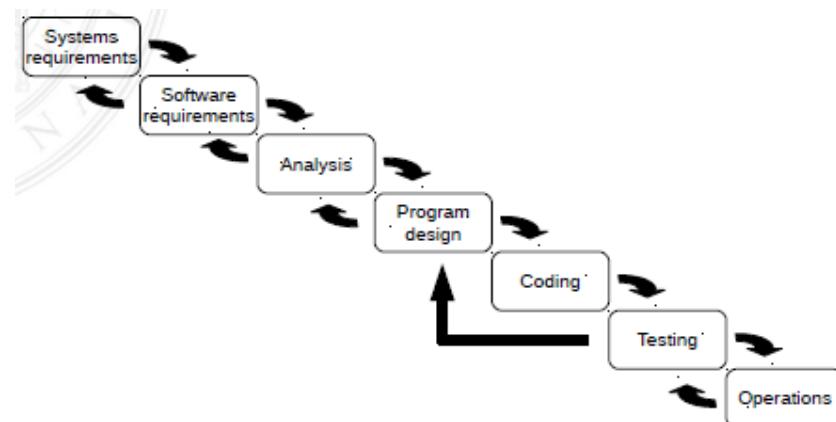
Quindi, c'è un unico modo per gestire tutte queste attività e i loro artefatti?

No, però, posso correlare attività diverse, unendole per skills.

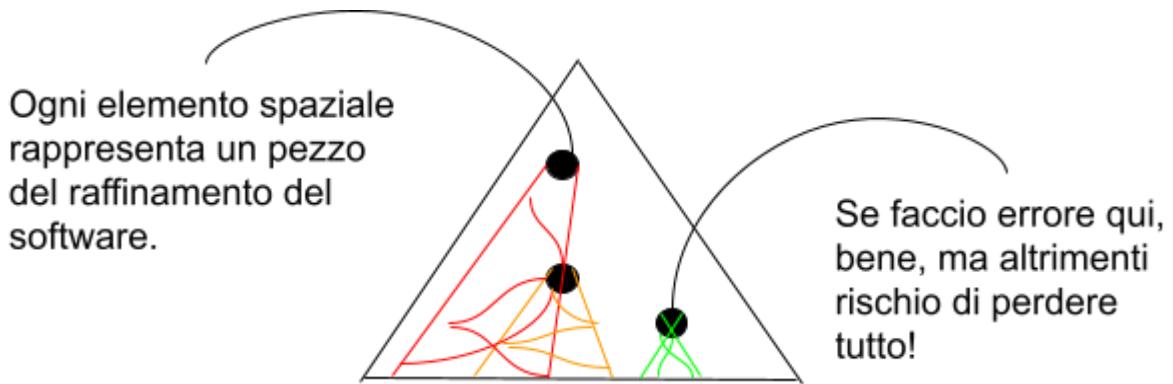
Vediamo alcuni modelli che cercano di organizzare il ciclo vita software:

Modello a cascata

Il modello più elementare è quello a cascata, ovvero una struttura a gruppi di attività ed esse vengono svolte rigidamente in sequenza.



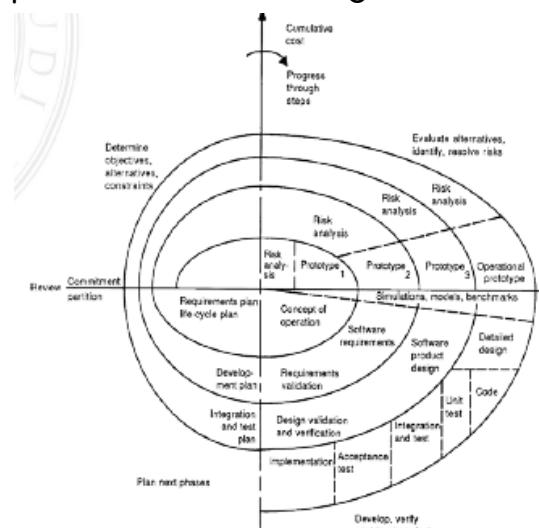
Il problema di questo modello e' che esso è il irrealo, funzionerebbe solo se non facessimo mai errori. Il motivo è che se all'ultimo step della Scala mi accorgo di un errore in fase di codice o peggio di requisiti, devo tornare a ritroso, percorrendo tutta la scala. Oltre al fatto che fare errori più in alto rischia di far perdere tutto il lavoro:



Di certamente positivo questo modello è che è facile da capire e da implementare, non solo anche da coordinare. Identifico facilmente i milestones e gli artefatti. Di negativo invece è l'irrealità per cascata solo in avanti. Non genera prodotti intermedi per controllare il lavoro durante la sua creazione. Ha una gestione dei rischi fallimentare in quanto se sbagliamo c'è il rischio reale di perdere tutto, oltre il fatto che attuare dei cambiamenti in corso è impossibile.

Modello a spirale

Il modello a spirale nasce nel 1986 e affronta i processi in modo iterativo. Il modello a spirale a un approccio basato sui rischi che prevede le ripetizioni delle attività più volte, cercando di organizzare per affrontare al meglio i rischi.



Si alternano attività di tipo simile, dove nella prima parte della spirale si identifica il rischio*. Il software è prodotto a pezzi (pezzi di requisiti).

In questo caso se sbaglio anche sono una fase, non perdo tutto il lavoro, ma solo una parte e non devo ripartire.



Siccome prendo un insieme di requisiti, Come faccio a sceglierli? Bisogna fare prima le cose più complesse. Per ogni ciclo lo stakeholder controlla il lavoro: coinvolgo il committente per ogni fase. Le cose positive di questo modello sono il fatto di essere iterativo e incrementale, un approccio decisamente più ragionevole e realizzabile rispetto il precedente modello a scala. Mentre di negativo è l'applicazione del modello matematico alla base della spirale. Inoltre il fatto di essere guidato solo dai rischi fa sì che all'inizio vengano prodotti solo dei prototipi ma nessuna funzionalità, il che significa che vengono prodotti moltissimi documenti con spesso la conseguenza di essere in ritardo di consegna.

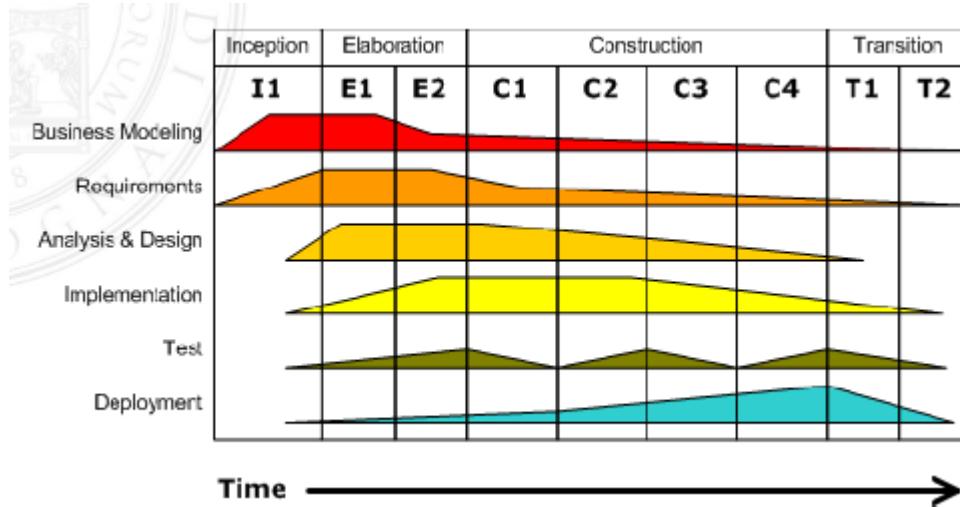
Dopo aver analizzato questi due principali modelli, prendiamo dal modello a spirale la sua iteratività: infatti la ricerca dimostra che ai metodi iterativi sono associati valori superiori di tassi di successo e di produttività, oltre che inferiori livelli di difetto. Il modello più utilizzato è UP.

Unified Process UP

È un metodo iterativo e incrementale per lo sviluppo di software. Combina pratiche in un ciclo iterativo con coesione ed è ben documentato. Questo modello combina le migliori e comuni pratiche, come un ciclo di vita iterativo e uno sviluppo risk-driven. In sintesi UP:

- Iterativo e incrementale
- casi d'uso
- architettura centralizzata
- focalizzato sui rischi

Tipicamente il progetto è diviso in quattro fasi, divise a loro volta in iterazioni:



- 1) Inception: L'obiettivo di questa fase è creare un business case, definire i casi d'uso, candidare delle possibili architetture da implementare successivamente e identificare i rischi.
- 2) elaboration: Se l'architettura definita nella fase 1 è giusta, viene validata. Si costituisce un piano per la costruzione che include costi e tempi da rispettare.
- 3) Construction: implementazione del sistema, Ma questo viene fatto raffinando le fasi precedenti, in particolare le iterazioni della costruzione sono time-boxed, ovvero decido delle deadline, per ogni box. Se il tempo per lo sviluppo di una box scade, posso decidere di buttare l'intera iterazione, oppure tengo l'errore di volutazione e decido come gestire nuovamente il tempo di conseguenza.
- 4) transition: Nell'ultima fase viene presentato il sistema, si raccolgono i feedback su di esso e ovviamente viene effettuato un training presso il cliente per istruirlo sull'utilizzo del software.

Nel grafico le iterazioni sono e possono essere diverse in numero per ogni fase (per esempio Construction ha quattro iterazioni). Alla fine di ognuna delle quattro fasi viene prodotto e rilasciato un prototipo. Questo modello in realtà non definisce delle modalità di svolgimento applicabili concretamente, ma ne esistono delle istanziazioni:

- RUP
- AUP
- OpenUP
- Oracle Unified Method

02/10/2019

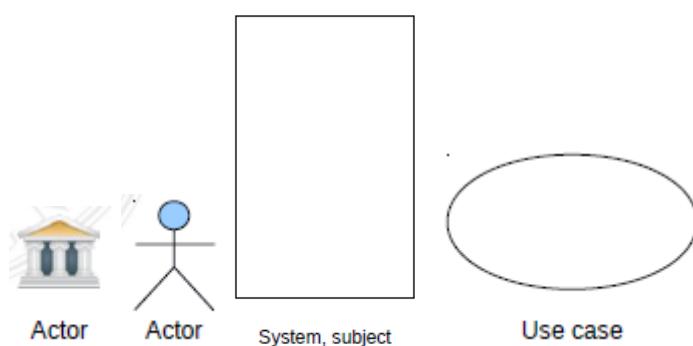
*UML: USE CASE

Analizziamo i modelli sotto forma di diagrammi in particolare vediamo: UC. Esso si usa solo nel dominio dell'analisi e serve a comprendere il problema:

Questo è un diagramma comportamentale usato per descrivere delle azioni (casi d'uso) che un sistema (soggetto) deve o può fare in collaborazione con uno o più soggetti esterni (attori) al sistema. Ogni caso d'uso produce dei risultati osservabili e valutabili dallo stakeholder. Il diagramma dei casi d'uso è usato per specificare:

- I requisiti di un soggetto, requisiti di un sistema, per catturare che cosa un sistema sotto costruzione dovrebbe fare
- Le funzionalità offerte dal soggetto, ovvero che cosa il sistema fa
- I requisiti che il soggetto pone al suo ambiente, ovvero come l'ambiente dovrebbe interagire con il soggetto in modo tale che esso sia in grado di performare i suoi servizi

Elementi di UC:



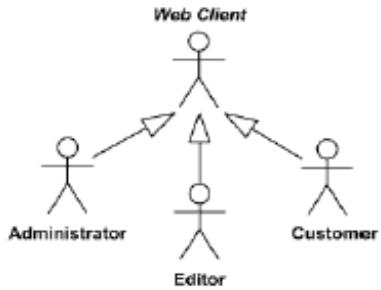
Actor: Dal punto di vista tecnico è un particolare classifier che specifica un ruolo giocato da un'entità esterna che interagisce con il soggetto, quindi con il sistema, esso potrebbe essere un essere umano oppure un servizio o un'azienda.

Soggetto: Il soggetto è il sistema sotto analisi o sotto progettazione con una serie di casi d'uso al suo interno.

Casi d'uso: I casi d'uso specificano il comportamento di un soggetto descrivendo una sequenza di azioni performate dal sistema per raggiungere un risultato osservabile, un obiettivo. In altre parole, ogni caso d'uso descrive un'unità di funzionalità completa e utile che il soggetto fornisce al suo utilizzatore.

Relazioni tra i vari elementi:

- Relazione tra attori: l'unica relazione possibile tra un attore ed un altro è la generalizzazione. Essa esprime il concetto di "e' un":



In questo esempio
l'amministratore è un web Client. La generalizzazione è disegnata



come: una freccia.

- Relazione tra attori e casi d'uso: in questo caso l'unica relazione possibile è l'associazione. Pesta esprime il concetto di coinvolgimento: l'attore "è coinvolto in" quel caso d'uso.



In questo esempio il cliente è coinvolto nel caso d'uso "gestione dell'account".

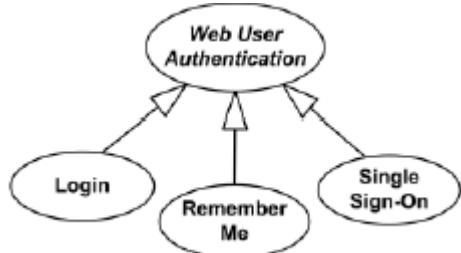
L'associazione è binaria 1 a 1.

Se ci sono più attori coinvolti, ed essi sono associati ad un caso, essi svolgono ruoli con importanza differente: In particolare ogni caso d'uso è iniziato da un solo attore (che viene chiamato iniziatore).

è buon uso porre gli iniziatori verso sinistra.

- Relazione tra casi d'uso:

- 1) Generalizzazione: la generalizzazione è una relazione che va dal caso d'uso più generale al caso d'uso più specifico, in particolare:



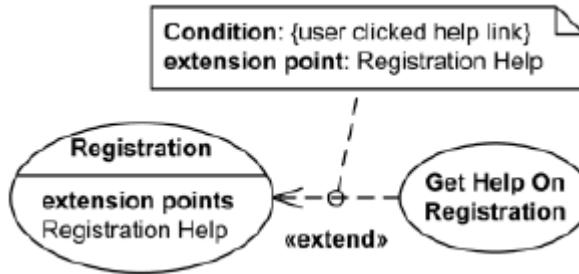
L'azione di Login è una specializzazione dell'autenticazione dell'utente. L'obiettivo di tutti questi casi d'uso è lo stesso, il motivo per cui vengono separati è che essi variano per qualche azione, ovvero interagiscono in modo diverso con l'esterno.

- 2) Estensione:

In questo caso i casi d'uso prendono due ruoli diversi:

- estende
- estendente

Il caso d'uso è una sequenza di interazioni, ho un caso che estende le interazioni dell'altro. Il primo caso deve rendersi disponibile ad essere esteso altrimenti non è possibile farlo, questo viene fatto attraverso degli extends points, dichiarati nel caso d'uso di partenza.

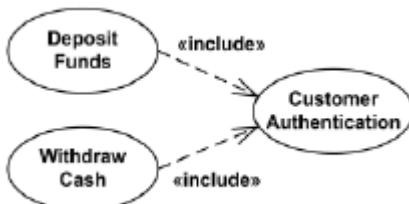


Sto svolgendo in

questo caso, una registrazione, se l'utente ha bisogno di aiuto, interviene il caso d'uso d'aiuto per la registrazione. Non sempre l'utente vuole aiuto, Difatti l'estensione non è obbligatoria nello svolgimento del sistema, essa è opzionale.

3) Inclusione: l'inclusione è una relazione diretta 1 a 1, non opzionale. Nel realizzare un certo caso d'uso io seguo delle interazioni che sono in un altro caso, questo perché:

- una certa sequenza potrebbe essere uguale in tutti i casi: allora prendo questa parte e la metto in un caso a parte da utilizzare tutte le volte in cui ne ho bisogno
- se ho casi d'uso molto lunghi e pesanti li devo splittare



Nell'esempio di fianco, tutte le volte che un cliente vuole depositare o prelevare del denaro deve per inclusione, quindi obbligatoriamente, autenticarsi.

4) Associazione - da non usare quasi mai

Business Use Case vs System Use Case

In un sistema che utilizza i casi d'uso il soggetto è il sistema stesso, invece in un business use case il soggetto è un'organizzazione, per esempio vogliamo spiegare come funziona una banca, quindi

l'interazione non è il sistema software ma le relazioni tra gli attori, Ecco perché gli attori in questo caso possono essere interni.

Tornando al nostro modello dei casi d'uso la difficoltà maggiore è nel inquadrare i vari casi d'uso del sistema. Bisogna ricordare che il caso d'uso è focalizzato, non descrive processi grandi, Magis su minimi elementi e questi dovrebbero esaurirsi nel tempo di una sessione, avere quindi una durata limitata. Quando ho più attori devo fare attenzione poiché è raro che più persone siano coinvolte nello stesso caso d'uso. Bisogna anche non decomporre troppo, quindi fare l'esatto contrario di ciò che abbiamo detto prima.

Il tempo può essere un attore per cose di cui non identifichiamo l'attore



e quindi non è possibile farne una stilizzazione: clessidra.

Dopo aver modellato il caso devo scrivere un documento per ordinare le sequenze (viene chiamato scenario), in realtà però non c'è una notazione standard l'importante è esplicitare alcune sequenze per confrontarci con il committente o per verificare che il sistema software faccia ciò che era stato richiesto .

Esercizio blog con posts e utenti.

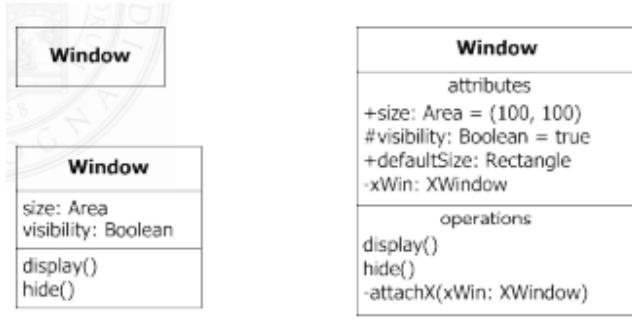
A blog is a web application presenting a collection of date-tagged messages (posts) on miscellaneous topics. Messages are posted by the blog owner who puts them online. The author can associate messages to one or more categories (expressed using keywords). Blog's visitors can comment messages; the comments, if approved by a moderator (usually the blog's owner), appear in a specific section under the original message.

03/10/2019

UML: CLASS DIAGRAM

Il diagramma delle classi riguarda e descrive idee generali, raramente sfrutta tutte le operazioni per il nostro uso. In UML la classe è un tipo, per natura un'astrazione, che rappresenta oggetti che condividono comportamenti e specifiche comuni.

In questo tipo di diagrammi la classe è dipinta come un rettangolo solido con il nome della classe in alto è suddivisa in compartimenti separati da linee orizzontali:



Comportamenti usuali sono

quello delle operazioni (metodi) e quello degli attributi(variabili e il loro tipo).

Fare caso al simbolo di fianco alle variabili:

+ = public

- = private

= protected

~ = package

I rapporti tra le classi possono essere di diverse molteplicità: (uno a molti, più a uno ecc.): in particolare per ogni rapporto-classe abbiamo un

- lower bound (se non specificato lower = upper)
- upper bound
- 1
- 0...1
- 1...*
- *

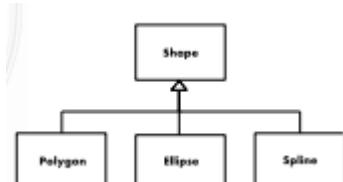
Le istanze sono oggetti con variabili di istanza della classe e sulle quali posso richiamare i metodi della classe a cui appartengono. Esse hanno uno stato (ovvero un insieme di proprietà dell'istanza, quindi le informazioni dentro l'oggetto) ma hanno anche un comportamento (ovvero quando richiamo metodo su di essa, il comportamento denota come si comporterà).

La notazione è la seguente: nomeDell'Istanza: Classe

Vediamo le relazioni tra le classi:

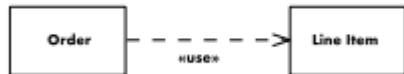
1) Generalizzazione:

Generalizzazione relaziona una specifica classe con una classe più generale (si assume un meccanismo di ereditarietà).

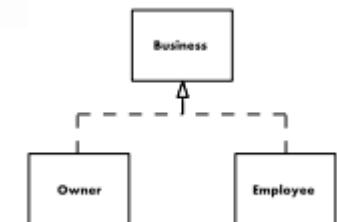


2) dipendenza

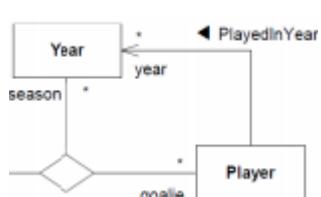
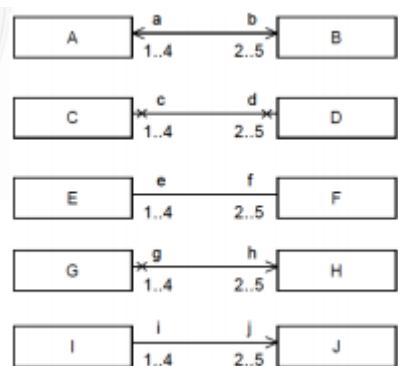
La dipendenza denota una classe fornitrice e una classe utilizzatrice, eventuali modifiche sul fornitrice possono avere impatto sull'utilizzatore. È possibile segnare una parola chiave o un nome opzionale sul collegamento:



- 3) realizzazione: è una specie di dipendenza, che in realtà corrisponde al caso in cui alcune classi realizzano un'interfaccia.

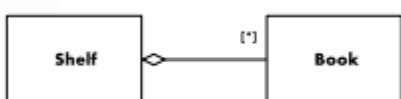


- 4) associazione: Denota la presenza di un legame tra gli elementi, non è un collegamento binario, posso associare più di un elemento (grazie al sistema di scrittura visto prima - molteplicità)



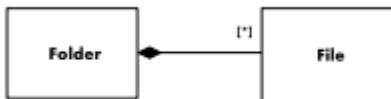
Dato il giocatore posso in maniera immediata sapere l'anno.

Se volessi specificare meglio il tipo di associazione e quindi trovare dei casi specifici posso usare questi ultimi due tipi di associazione:



- 5) aggregazione: In questo caso stiamo dicendo che il libro è collezionato dalla mensola, ma libro può esistere anche in altre forme(esiste SENZA mensola). L'istanza è indipendente dalla composizione.

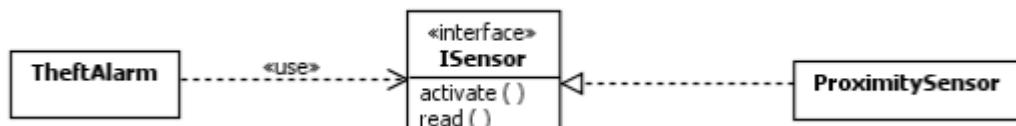
- 6) composizione: L'oggetto compositore ha una responsabilità per l'esistenza e la conservazione dell'oggetto composto:



Se non esistesse Folder non esisterebbe File.

Posso dichiarare anche **classi astratte**, che ricordiamo non avere stanze dirette, il nome di queste viene specificato usando l'annotazione testuale {abstract}.

Posso dichiarare allo stesso modo **interfacce**, che specificano cosa in comune:



08/10/2019

Avevamo parlato dell'O-O model come paradigma che spostava l'attenzione dall'analisi agli oggetti. Oggi vediamo invece l'**O-O Analysis** (OOA) model, che serve a rappresentare i concetti nel dominio del problema, le loro caratteristiche e come essi sono correlati tra di loro. Come un dizionario visuale per il dominio del problema.

Esistono diversi approcci per rappresentare il dominio delle classi: noi vedremo quello più semplice e basico.

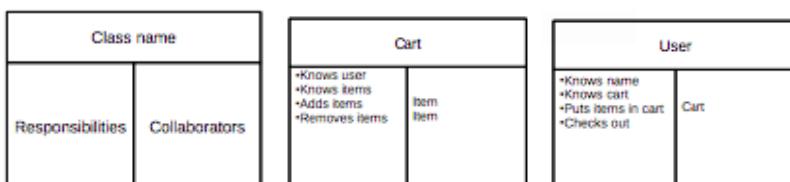
- 1) si parte da un enunciazione del problema, quindi una documentazione
- 2) si cercano i sostantivi e i concetti salienti, ma anche i verbi
- 3) poi si cercano le relazioni tra queste entità per identificare responsabilità e collaborazioni

Questi step sono iterati per rifinire il modello stesso.

Oltre al diagramma delle classi che abbiamo già visto, si usa anche un altro strumento:

CRC Cards

Class Responsibility Collaborator cards [Beck, Cunningham 89] are tools that can be used to derive a domain model. Se erano vere e proprie carte di carta e cercavano di creare i modelli suddividendo le classi in: responsabilità e collaboratori.



Quindi le CRC cards erano il risultato di un processo iterativo dove si identificavano le classi, si identificavano le responsabilità e i collaboratori.

Ma queste carte sono necessarie per forza?

In realtà no, però sono utili per chi non ha dimestichezza con i modelli, non solo tra i programmatori ma anche per coinvolgere clienti o stakeholder.

Esercizio Città di Paperopoli

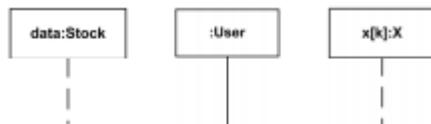
The City of Duckburg activates an initiative that allows dogs hosted in the kennels of the district to enjoy a walk in the Sunday afternoons. Citizens interested in taking custody of the animals register their availability specifying the Sunday and the area, among the many that make up the Municipality, in which they are willing to collect a dog (there are several kennels located in various areas). Given this availability the kennels assign dogs to citizens creating appointments which are then communicated to the volunteers.

15/10/2019

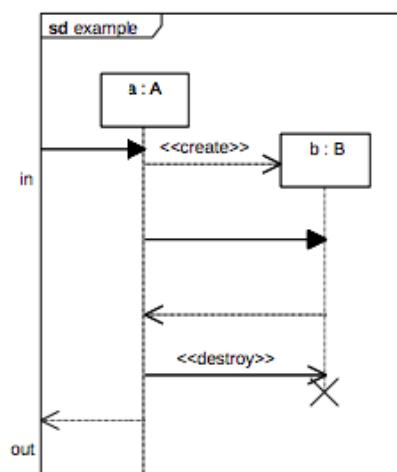
UML: INTERACTION DIAGRAM

(SEQUENCE DIAGRAM & COMMUNICATION DIAGRAM)

Il diagramma delle sequenze è il più comune tra i due. Si focalizza sul messaggio interscambiato tra un numero variabile di lifelines. Come determinati elementi interagiscono tra di loro al passare del tempo. Le lifeline sono elementi che rappresentano un partecipante individuale nell'interazione :



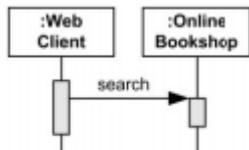
Il rettangolo rappresenta il partecipante, la linea tratteggiata conterrà le diverse occorrenze in cui il partecipante comparirà, nel tempo (che scorre verso il basso). Tutto il sistema può essere racchiuso in un rettangolo. Ovviamente se non ci sono gate dall'esterno posso anche non disegnarlo. I messaggi possono provenire anche dall'esterno, quindi da un punto non ben definito (il punto di ingresso è il gate):



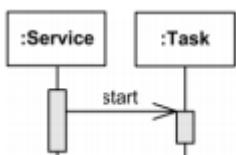
Ogni volta che un messaggio viene generato non ne sappiamo l'occorrenza temporale. Ma a seconda del tipo di azione che è stata usata per generare il messaggio, il messaggio stesso può essere:

- chiamata sincrona, chiamata asincrona

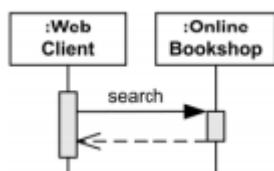
Una il contrario dell'altra: Sincrona - rappresenta l'invio di un messaggio tale per cui colui che manda il messaggio sospende l'esecuzione mentre aspetta per la risposta:



Asincrona - Presenta l'invio di un messaggio tale per cui colui che manda il messaggio procede immediatamente senza aspettare un valore di ritorno:



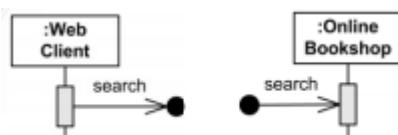
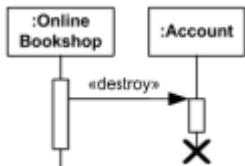
- segnale asincrono
- risposta: messaggio di risposta a una operazione di chiamata:



- creazione: sto creando un altro partecipante:



- cancellazione: viene usata per terminare un'altra lifeline
la x denota la distruzione dell'occorrenza:

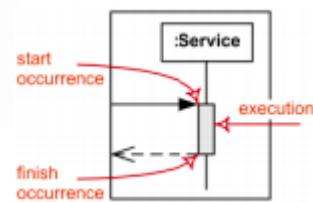


Modellazione verso o da ignoti:

Le interazioni possono essere raggruppate per formare le Interaction Fragment ovvero delle unità.

Non c'è una notazione generale, ma ne esistono di vario tipo:

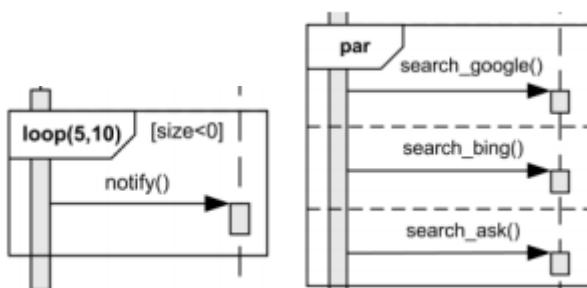
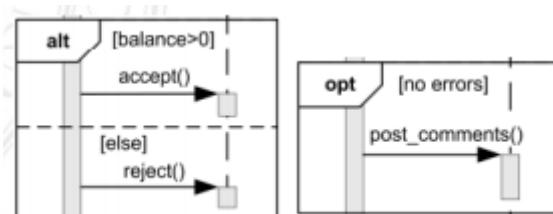
- occorrenza: è un frammento di interazione che rappresenta un momento nel tempo all'inizio alla fine di un messaggio o all'inizio o alla fine di un'esecuzione
- esecuzione: Frammento di interazione che rappresenta il periodo in cui la lifeline dei partecipanti...:
- Sta eseguendo un'unità di comportamento
- Sta mandando un segnale a un altro partecipante
- Sta aspettando per un messaggio di risposta da un altro partecipante



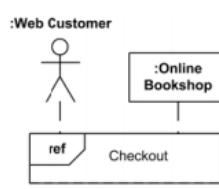
- frammenti combinati: l'utente è in grado di descrivere un numero di possibili traces in una maniera compatta e concisa:

Interaction operator could be one of:

- alt - alternatives
- opt - option
- loop - iteration
- break - break
- par - parallel
- strict - strict sequencing
- seq - weak sequencing
- critical - critical region
- ignore - ignore
- consider - consider
- assert - assertion
- neg - negative



- interaction use: Utilizzato per usare o chiamare un'altra interazione, serve inoltre a semplificare larghe e complesse sequenze:



Per cosa si usa questo modello?

- 1) Per il dominio del problema: ovvero rappresentare scenari o sequenze all'interno dei casi d'uso (così da non dover scrivere tutto a mano).

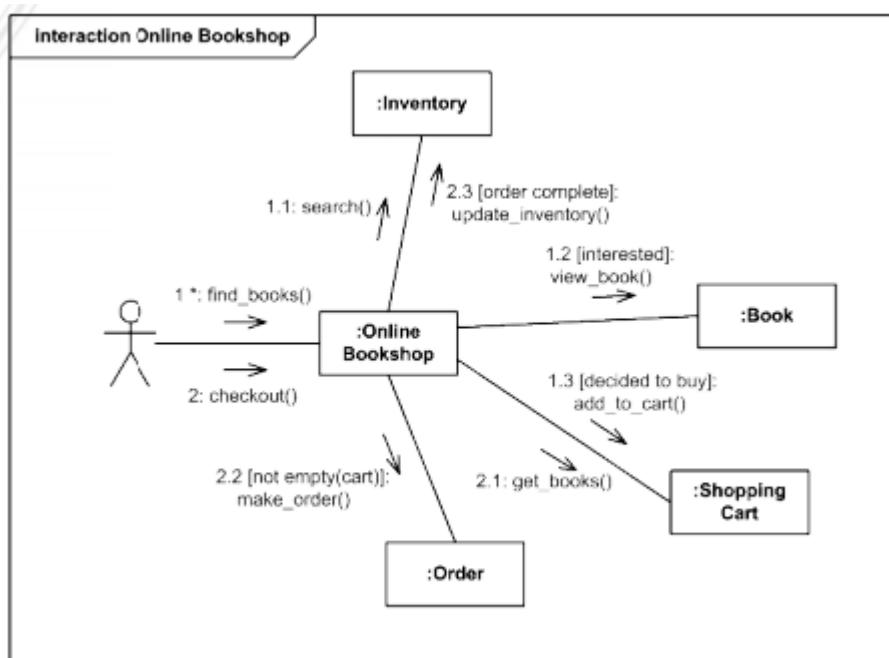
Se sto utilizzando questo modello, per questo tipo di dominio, ovvero per l'analisi, non è necessario tra asincrono o sincrono, quindi andare nello specifico. In questo caso voglio catturare tutte le interazioni tra il sistema e l'esterno, non è d'interesse il funzionamento del sistema internamente.

Bisogna fare attenzione a non sovraccaricare il modello con tante lifeline!

- 2) Per il dominio della soluzione: voglio esprimere delle dipendenze temporali (se non fai "questo" prima di "quello" ecc)

Non bisogna fare molta nidificazione di frammenti, al massimo tre Uno dentro l'altro.

Diagramma della comunicazione mostra invece, le interazioni tra lifeline usando una disposizione libera dalla forma: non dipende dalla temporizzazione, mi interessa solo l'ordine in cui avvengono!



Le lifeline

vengono sparse, ci sono i messaggi ma non sono caratterizzabili, ci sono meno informazioni rispetto a quelli di sequenza. Sappiamo chi parla con chi è l'ordine viene espresso con dei numeri.

Non vi è corrispondenza tra tempo e spazio perché lo spazio è arbitrario il tempo viene indicato soltanto da una numerazione.

Esercizio con macchinetta distributrice di cibo e bevande.

ACTIVITY DIAGRAM

Questo diagramma si intendono le attività come comportamenti del sistema. Gli elementi sono connessi per formare dei flussi, ovvero la combinazione delle attività e la sua coordinazione.

Si creano delle reti di elementi a rappresentazione di una rete casuale.

Posso rappresentare:

- singoli elementi
- business Process
- algoritmi

La semantica è semi formalizzata, basata sulla semantica matematica di Petri (= i flussi hanno dei token che attivano le varie azioni).

Portamenti sono realizzati attraverso più attività in concorrenza tra di loro.

Gli elementi fondamentali di questo diagramma sono:

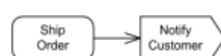
- Attività: le attività sono dei comportamenti rappresentati da un flusso di azioni, quest'ultimo è modellato come un nodo di attività connesso da frecce.
Viene disegnato come un rettangolo arrotondato con il nome in alto a sinistra e dentro tutto il flusso:



- Actions: L'azione è un elemento con nome che rappresenta un singolo atomico step di un'attività. Ce ne sono di vari tipi, possono essere chiamate di operazioni, comunicazioni, manipolazione di oggetti, invocazioni di comportamento.
Sono rappresentate esattamente come le attività ma possono essere collegate ad altre. Inoltre possono esserci delle condizioni di input o output ad essa collegate (pre-conditions e post-conditions).

Le azioni possono creare o ricevere degli eventi:

- **Send signal**



- **Accept signal**



- **Repetitive time**



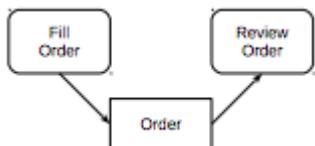
Le azioni di chiamate di comportamenti sono indicate Mettendo uno simbolo speciale nel rettangolo:



- Archi, ovvero le Activity Edge: è una connessione diretta tra due nodi di attività, dove il token scompare da una parte e compare dall'altra. Le Activity Age possono avere delle guardie che determinano quando un ponte può essere attraversato.



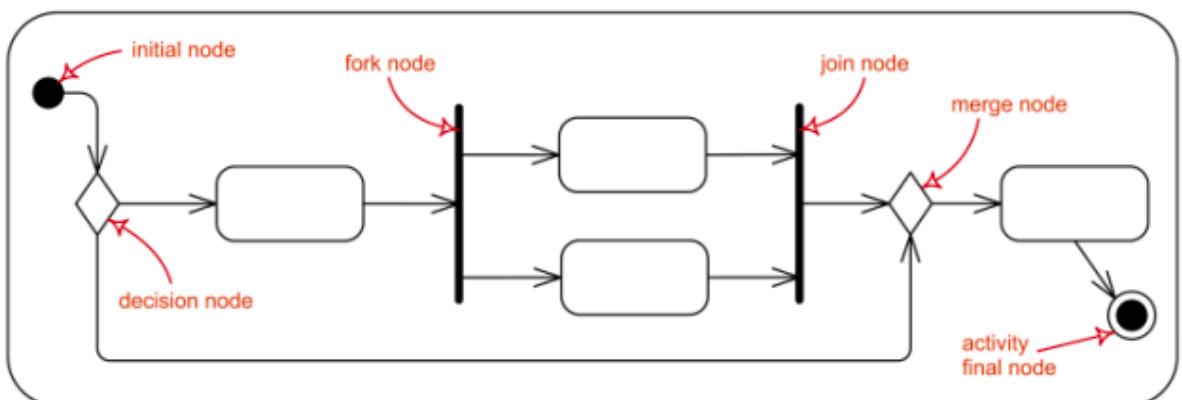
- Object node: indica che un'istanza di un particolare Classifier, può essere disponibile ad particolare punto di una attività:



Altri elementi visuali:

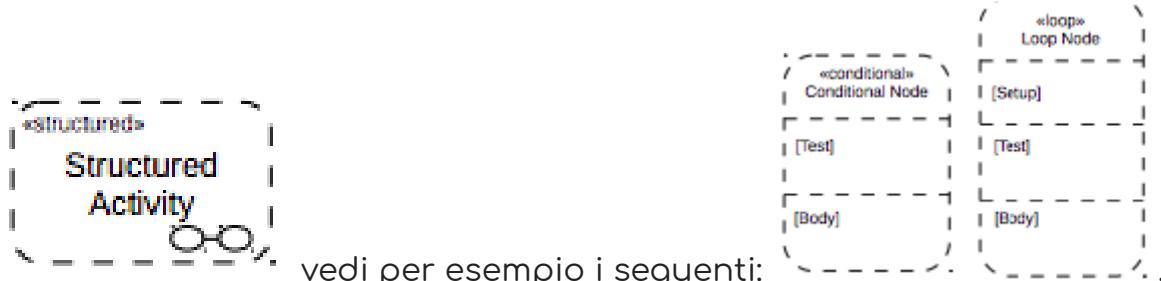


- connettori (per collegare pezzi di diagramma che non ci starebbero) per l'ordine.
- Nodi di controllo:



Nel fork node i token si sdoppiano e percorrono contemporaneamente le due strade.

Possiamo raggruppare attività, specificando anche frammenti o partizioni di tipo particolare:

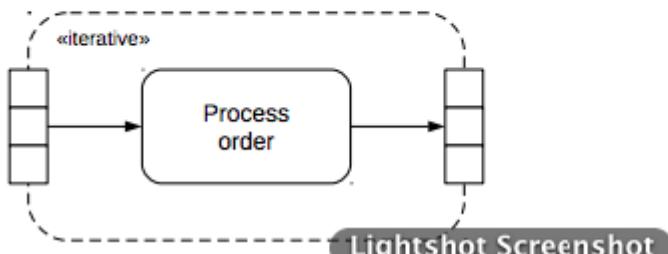


Particolarmente importanti sono le regioni di espansione:

1) Expansion region:

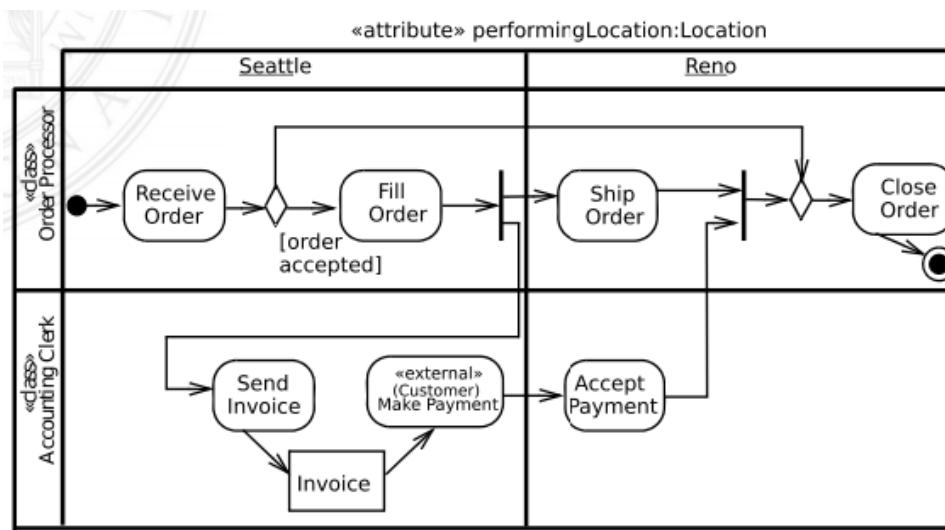
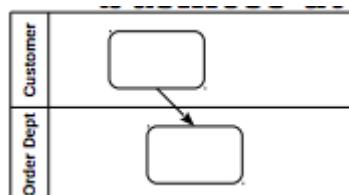
Strutturata come una regione che prende come input una collezione e agisce su ogni elemento della collezione individualmente e produce elementi in output della collezione stessa.

Elements processing can take place sequentially («iterative»), concurrently («parallel») or in a streamline («stream»).



Lightshot Screenshot

2) È buona pratica far capire Quale soggetto compie le azioni: Ecco perché si introducono le Activity Partition:

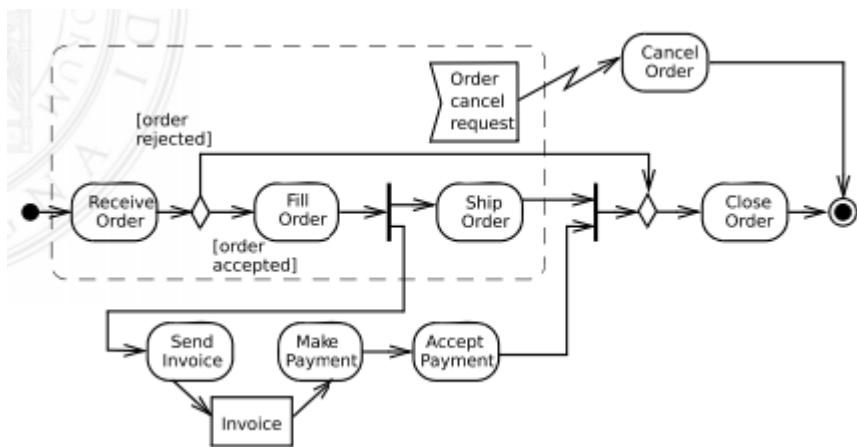


a fianco.

Esempio qui

3) Ricordiamo anche le regioni interrompibili: **interruptible regions**, che 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 all'interno del confine della regione.

Quando un token è accettato da un tipo speciale di bordo chiamato bordo di interruzione, che è disegnato da un fulmine, lascia la regione e tutti gli altri token sono distrutti e gli altri comportamenti all'interno della regione sono terminati:



Esercizio sulla macchinetta distributrice.

17/10/2019

STATE MACHINE DIAGRAM

Diagramma comportamentale che identifica le macchine da una serie di passaggi di stato di esse. Le principali due famiglie di stato delle macchine sono:

- 1) protocol state machine - descrive le interazioni tra le parti della macchina che sono funzioni di uno stato* di avanzamento (in sostanza ho diversi stati di avanzamento che descrivono le funzioni della macchina)

*stato = è una situazione in cui valgono degli invarianti. A fronte di diversi input produce comunque gli stessi output.

Gli stati possono essere:

- semplici - non scomponibili
- complessi - composti, possono contenere più di una regione (gli stati in queste regioni vengono detti substates)

- 2) behavioral state machine - come evolve nel tempo il sistema in funzione di una serie di eventi che hanno luogo.

Agli stati si associano dei comportamenti:

- enter

- exit
- doActivity

la cui notazione è uguale al diagramma studiato precedentemente (rettangoli quadrati).

Però:

gli stati possono essere suddivisi in compartimenti multipli separati da una linea orizzontale - compartimento del nome, compartimento del comportamento, compartimento delle transizioni e compartimento per gli stati composti.



Pseudostati e stati finali:

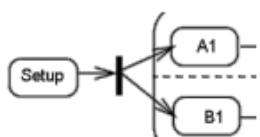
Lo stato finale è un speciale tipo di stato che significa che la regione di cui fa parte è stata completata. Il contrario è rappresentato dal punto di initial della macchina (ovvero il suo punto di partenza) dove però non rimarrà mai, al contrario del punto finale.

Gli pseudostati invece sono usati per arricchire la semantica degli stati di macchina: possono essere

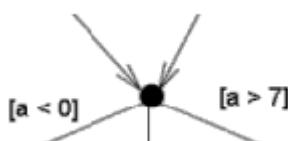
- initial - il punto di inizio della regione
- join - è il punto di unione per due o più transizioni, originate in diverse regioni. Attraverso il join si performa una sincronizzazione che ha termine solo se hanno termine tutte le transizioni di origine.



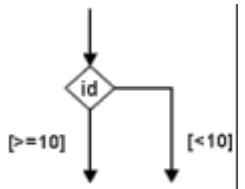
- fork - punto di arrivo o contrario del join



- junction - unisce e splitta transizioni differenziate per condizioni



- choice - è un tipo di giunzione usata per realizzare diverse strade possibili: una condizione else permette di scegliere



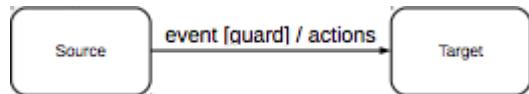
attenzione: usare solo quando l'evento di arrivo è strettamente collegato ai due successivi (usare meno possibile)

- entryPoint - punto di accesso ad una submachine
- exitPoint - punto di uscita
- terminate - terminazione totale

Come si collegano gli stati fra di loro? O meglio quando si passa da uno stato all'altro?

Attraverso le **transitions**.

Le transizioni sono atomiche e vengono attivate da eventi o condizioni. Durante il tempo di transizione si può definire un comportamento con un tempo di realizzazione (abbiamo quindi delle condizioni o guardie).

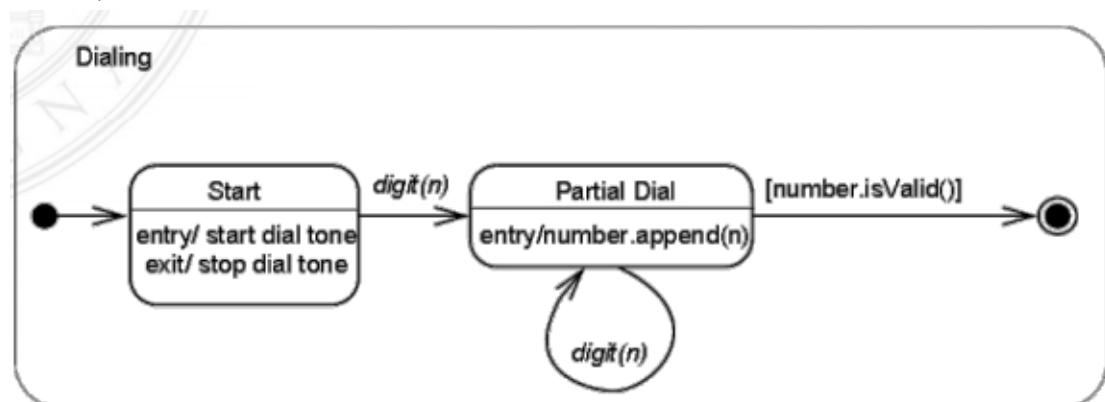


Es. lampadina

Consideriamo invece degli stati degli **eventi**:

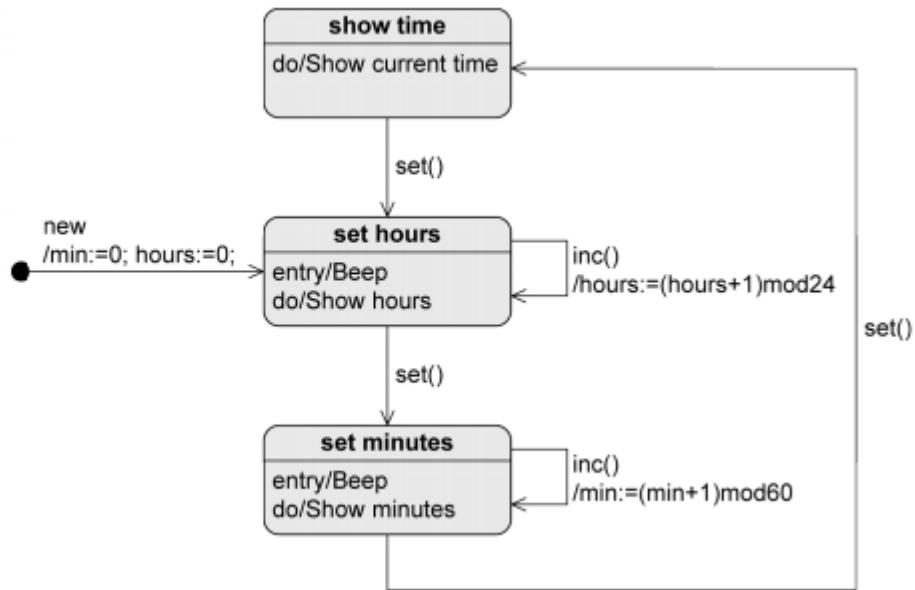
gli eventi sono occorrenze osservabili nell'ambiente del soggetto (mentre gli stati sono invisibili per l'utente, gli eventi no). Accadono ad un certo punto (sono associati ad un momento temporale), non hanno una durata. Possono avere dei parametri.

Esempi:

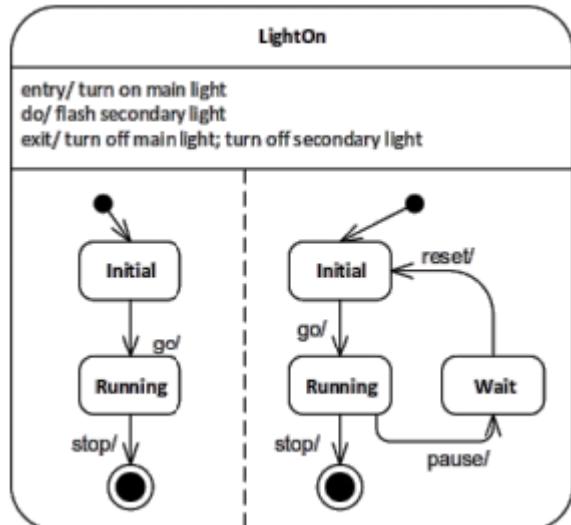




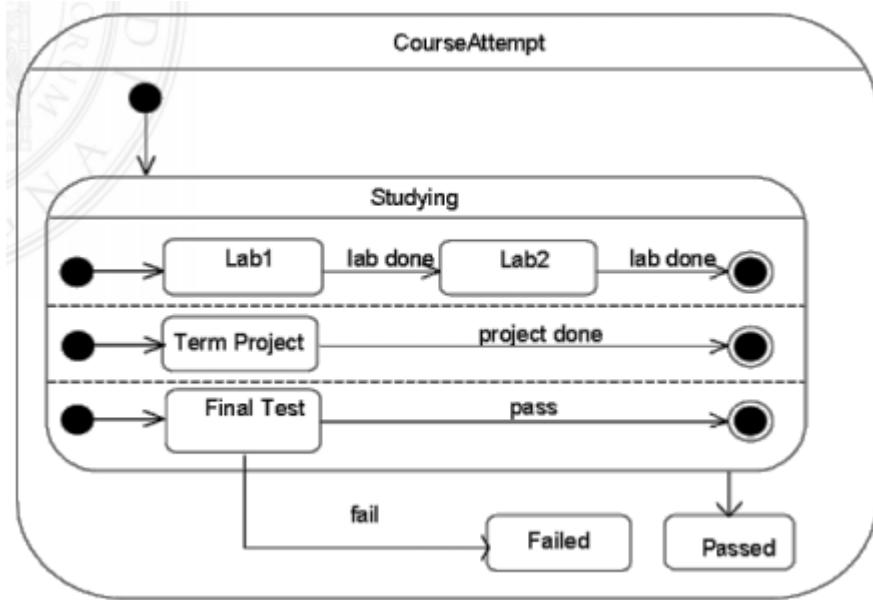
DigitalClock
- min: int
- hours: int
+ set(): void
+ inc(): void



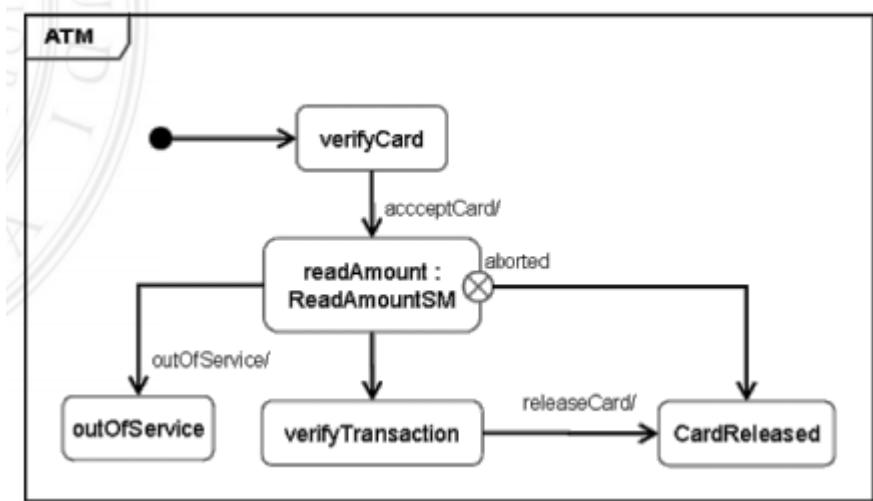
Ovviamente gli stati possono essere raggruppati in **regioni**: ogni regione sarà caratterizzata da un comportamento concorrente: per passare da una possibile regione all'altra entrambi i comportamenti devono essere terminati!



Però come nel caso successivo posso sempre attuare degli stratagemmi:

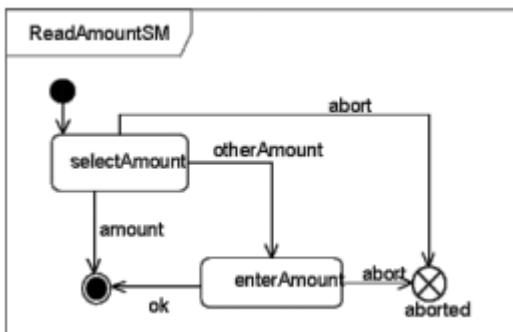


oppure



dove

readAmountSM è una submachine!



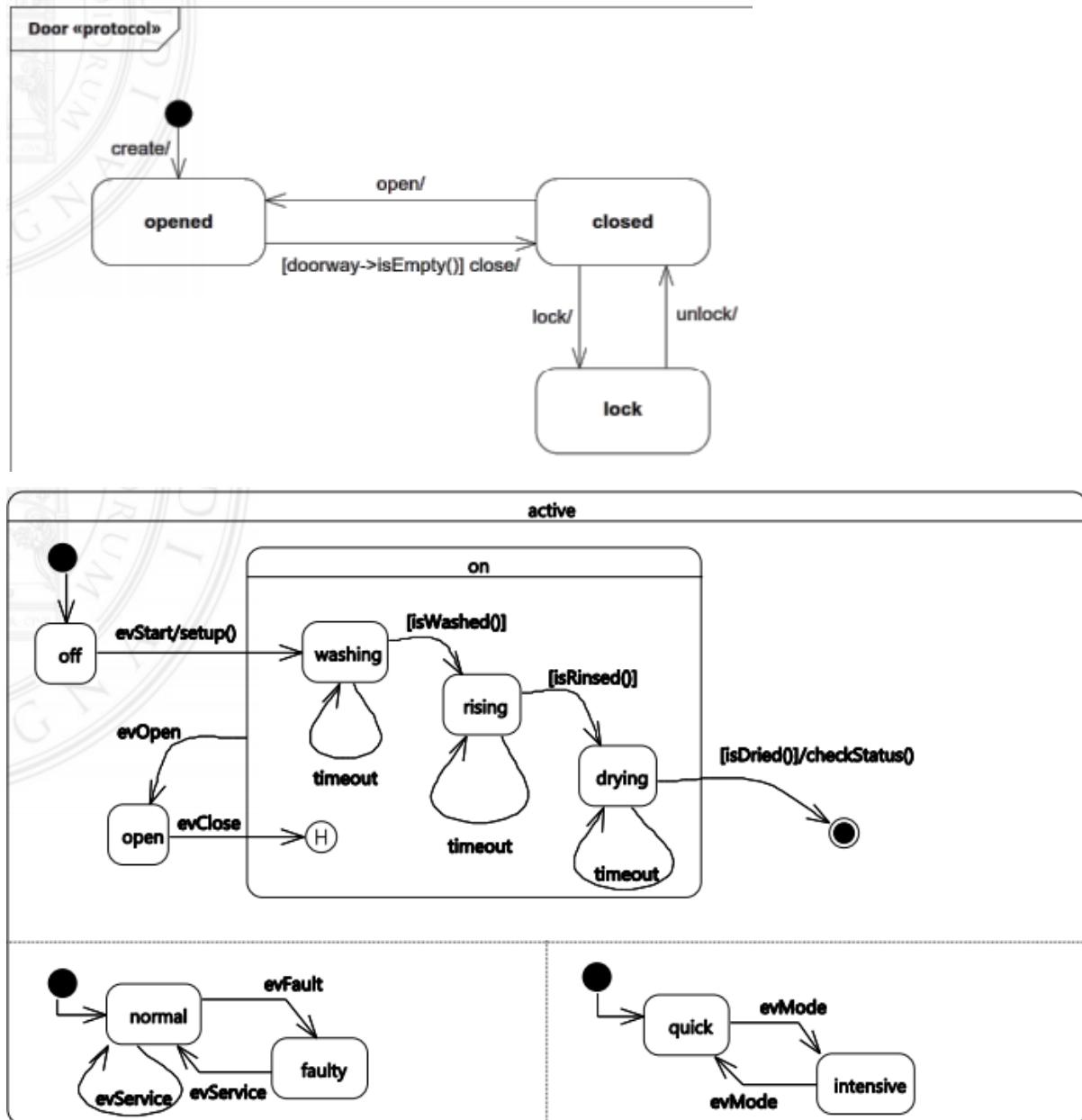
Ma uscire da una sottomacchina mi fa perdere tutti i dati oppure li mantengo?

Li mantengo grazie allo **state history** appunto usato per tenere traccia dello stato di configurazione di una regione. Può essere deep o shallow a seconda se le sottomacchine sono di più livelli o di primo livello (semplicità).

Tra lo stato iniziale e finale si attraversano tanti percorsi (tanti stati) il token che viaggia lungo le transition si accende e spegne da una parte e poi dall'altra. Il viaggio del token rappresenta il progresso, attraverso un run to completion (se la macchina si ferma allora si trova in un wait point, se il percorso è ben strutturato allora comunque la semantica deve poter dire dove e cosa la macchina deve fare).

Quando un evento non può essere processati finisce in una queue.

Di seguito altri esempi: porta di ascensore e lavastoviglie



Bisogna ricordare che se siamo in fase di progettazione il diagramma deve essere più specifico (metodi ecc.), altrimenti in fase di analisi importano soprattutto stati ed eventi, ma non specificando.

22/10/2019

Fino ad oggi abbiamo analizzato tutte le attività che hanno a che fare con la comprensione del problema, oggi invece dobbiamo cercare di capire cosa dobbiamo fare per realizzare il sistema software. Questo passaggio non è banale e non esistono delle pratiche universali: dobbiamo passare da una prospettiva descrittiva alla risoluzione del problema.

Lo faremo attraverso una versione super semplificata di UP (Unified Process):

Se io facessi questo processo per ogni caso d'uso, per ogni System sequence diagram del caso d'uso dovrei:

- dividere il sistema nel livello dell'interfaccia e logica aziendale / livello di dominio
- Trovare / creare elementi nel business livello logico / di dominio responsabile per supportare tutte le interazioni al livello di interfaccia
- Aggiungere in modo incrementale nuovi elementi al diagramma della classe di progettazione / operazioni di aggiunta

Certo, ma questo se non mi importasse della **qualità** della mia soluzione!

23/10/2019

Torniamo alla qualità:

Software quality and Object Oriented Principles

Le qualità possono essere:

- Esterne
 - Ovvero qualità che l'utente puo' percepire:
 - funzionali
 - non funzionali
- Interne
 - Ovvero qualità nascoste all'utente:
 - come il software è organizzato

Nello specifico vediamo entrambe:

- 1) Nel caso delle qualità esterne:
 - Correttezza: La correttezza del codice nel rispettare ciò che viene richiesto
 - usabilità: metrica di sforzo che viene chiesta all'utente per l'utilizzo del software
 - efficienza: l'uso che il software fa delle risorse
 - Affidabilità: garantisce un certo livello di sicurezza
 - integrità: livello di protezione contro i danni
 - adattabilità: si adatta all'uso e agli aggiornamenti
 - accuratezza: riguarda la precisione matematica dei calcoli

- robustezza: cosa succede al software nel caso di incidenti accidentali come esso reagisce

2) Nel caso delle qualità interne:

- mantenibilità: in termini di denaro quanto spendo per la manutenzione
- flessibilità: Per aggiungere nuove funzionalità o toglierle
- portabilità: quanto pesa a livello software per farlo girare su varie piattaforme
- riuso: il riciclo di pezzi di codice è possibile? classi, librerie ecc
- leggibilità: indentazioni, termini e nomi a livello di codice sono semplici?
- testabilità: posso testare il software da solo o mi servono altre classi o programmi?
- comprensibilità: quanto si comprende la struttura del software?

:) Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

Va tenuto presente che il codice viene letto e modificato moltissime volte: il costo associato alla sua evoluzione si stima sia tra il 50-90% del totale del costo di produzione dello stesso.

Esistono dei valori di valutazione per la qualità software? Sì, per esempio ISO 9126, che mette a disposizione 3 modelli di qualità:

- data quality model (dati e gestione)
- product quality model (processo di produzione)
- quality in use model (qualità percepita dagli utenti)

Di fatto a noi interessa certamente un metro di misura della qualità.. ma solo a volte: a noi interessa che il software sia riusabile e progettato per il cambiamento! **Designed for change**.

Faremo in particolare riferimento agli **OO Principles**: dobbiamo correttamente identificare i nostri oggetti e trovare il giusto mix di encapsulamento, ereditarietà e polimorfismo per ottenere un software di alta qualità. I principi possono essere usati per garantire la massimizzazione di queste ultime.

Encapsulamento: nascondere i dettagli, mostrando invece delle interfacce

Ereditarietà: condividere stati e comportamenti

Polimorfismo: riferirsi a tipi correlati come uno solo

Cosa è indizio di cattiva qualità? Design smells:

- Rigidità: risulta difficile cambiarlo, anche in maniera semplice. Una progettazione è rigida se anche un singolo cambiamento causa una cascata di cambiamenti in moduli dipendenti.
- Fragilità: è la tendenza di un programma a rompersi in diversi punti quando un singolo cambiamento viene fatto. Spesso il nuovo problema si trova in aree che non hanno una relazione concettuale con l'area che era stata cambiata.
- Immobilità: una progettazione è immobile quando contiene parti che potrebbero essere utili in altri sistemi, ma il rischio coinvolto nel separare queste parti dal sistema originale è troppo grande
- Viscosità: l'ambiente di sviluppo è lento inefficiente
- Inutile complessità: il sistema contiene elementi che non sono utili, questo succede soprattutto quando gli sviluppatori anticipano cambiamenti.
- Inutile ripetizione: quando lo stesso codice appare tante volte in diverse ma di poco forme, gli sviluppatori stanno commettendo un disastroso errore (il famoso copy and paste)
- Opacità: accade nella tendenza di un modulo di essere particolarmente difficile da capire, soprattutto per come il codice è scritto; a volte questo succede con l'aumentare del tempo, un costante sforzo nel mantenere il codice pulito ed espressivo deve essere richiesto per mantenere l'opacità al minimo.

Le soluzioni si specificano per ogni caso, ma soprattutto esistono delle soluzioni ad alto livello chiamate **SOLID**. Sono dei principi generali ed è probabile che se essi sono rispettati, la qualità abbia probabilità di aumentare.

24/10/2019

L'acronimo SOLID è dovuto ai principi che costituiscono questa soluzione:

- 1) Single responsibility principle
- 2) Open-closed principle
- 3) Liskov substitution principle
- 4) Interface segregation principle
- 5) Dependency inversion principle

Vediamoli nel dettaglio:

- 1) SRP

Se una classe ha più di una responsabilità, le responsabilità

si accoppiano. Il cambiamento di una responsabilità può compromettere o inibire la capacità della classe di incontrare le altre. Questo tipo di accoppiamento porta a design fragili.

2) OCP

Una classe dovrebbe essere aperta all'estensione, ma chiusa per la modifica.

Il mancato rispetto di OCP porta a un cambiamento con conseguente cascata di cambiamenti (rigidità). OCP ci consiglia di fare refactoring della nostra progettazione per evitarlo.

Refactoring = è una tecnica disciplinata per ristrutturare un sistema software così che la sua struttura interna sia modificata senza cambiare il suo comportamento interno.

Vedi croccantino gatti e cani... prendo le cose in comuni e creo la classe croccantino.

3) LSP

Quando creo una gerarchia di classi, la relazione tra esse dovrebbe essere la sotto-tipazione.

Se un metodo f, accettando come argomento un riferimento a B, si comporta male quando gli si passa, invece, un riferimento a un'istanza di D, sottoclasse di B, quindi D è fragile in presenza di f.

Vedi l'esempio del rettangolo e quadrato

Il principio di sostituzione di Liskov è uno dei i principali fattori abilitanti di OCP.

29/10/2019

4) ISP

Principio legato alla dipendenza: la dipendenza di una classe da un'altra dovrebbe dipendere sulla più piccola interfaccia possibile oppure la dipendenza non dovrebbe essere forzata per quanto riguarda i metodi che una classe non usa.

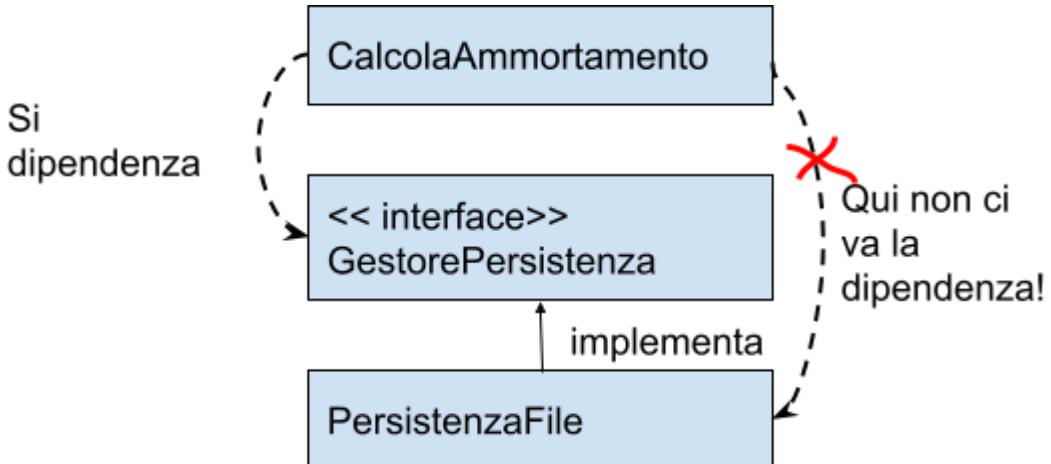
L'istanza più frequente della dipendenza è quella della dipendenza d'uso: Bisogna minimizzare la percentuale di contatto tra le classi, se come dicevamo prima la dipendenza riguarda dei metodi, ma soltanto una parte di essi, è meglio porli in un'interfaccia (in questo modo la classe A dipende da quella piccola interfaccia di B).

5) DIP

Depend upon abstraction. I metodi di alto livello (classi) non dovrebbero dipendere dalle classi di basso livello, ma entrambi dipendere invece dalle astrazioni.

Le classi non sono tutte uguali per importanza, peso o compiti, ma devo sforzarmi di fare lo sforzo di fare l'opposto:

se ho una classe di alto livello, questa non si deve occupare di scrivere su file (lo dovrebbe fare una classe di più basso livello), ma allo stesso tempo me ne devo rendere indipendente: come fare? Grazie alle astrazioni (interfacce) comandate dalla classe ad alto livello:



In questo caso, per esempio, `PersistenzaFile` è più volatile e sostituirla sarà più semplice.
`GestorePersistenza`, invece, non cambierà mai a meno di specifica necessità della classe ad alto livello, quindi raramente.
L'interfaccia è diventata un elemento per esprimere le necessità di alto livello ed è la classe utilizzatrice a definirla.

Attenzione:

Tutte le architetture object-oriented sono strutturate su livelli chiaramente definiti, con ogni strato che fornisce alcuni coerenti insiemi di servizi attraverso una ben definita e controllata interfaccia. "[G. Booch]

Un'ingenua applicazione dello stile a livelli può però facilmente portare a una violazione del DIP.

Una soluzione comune è quella di creare strati inferiori dipendenti da un'interfaccia di servizio dichiarata negli strati superiori.

Veniamo ora al punto: come uso OO principles per disegnare sistemi software?

Intanto: chi fa che cosa? Chi ha la responsabilità su chi?

RDD - RESPONSIBILITY DRIVEN DESIGN

RDD è un metodo per progettare sistemi software sulla base delle responsabilità.

Responsabilità del fare:

- creare oggetti o compiere delle operazioni
- inizializzare azioni in altri oggetti

- controllare e coordinare le attività di altri oggetti

Responsabilità del conoscere:

- Conoscere i campi privati
- conoscere gli oggetti collegati
- conoscere le cose che possono essere derivate o calcolate

Ma come applico RDD?

GRASP-GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS

Questo pattern* viene utilizzato per performare RDD, ma non solo, garantendo anche il rispetto dei principi SOLID durante la costruzione. Aiuta a comprendere la progettazione essenziale degli oggetti e applicare ragionamenti in un modo metodico, razionale e facilmente esplicabile.

*= Pattern = è una descrizione di un problema ricorrente in un dominio dello stesso (ambiente), accompagnata dalla sua soluzione, che si potrà riutilizzare milioni di volte, ma senza fare la stessa cosa più volte.

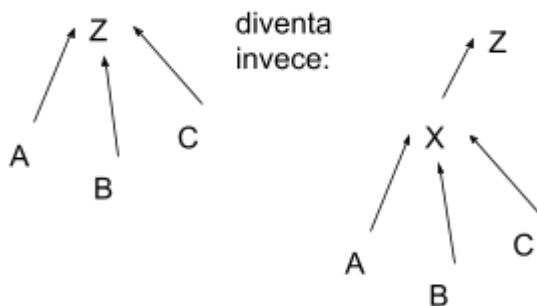
Pattern GRASP:

- 1) **Creator** - Chi deve creare una nuova istanza di una classe (oggetti)? Assegnare alla classe B la responsabilità di creare una istanza della classe A se una delle seguenti affermazioni è vera (se si deve scegliere tra più classi, vince quella che soddisfa più affermazioni):
 - B contiene o aggrega A
 - B monitora A (records - registra)
 - B usa molto frequentemente A (richiama metodi ecc)
 - B è un “esperto” di A (conosce già tutte le informazioni necessarie per creare l'inizializzazione) -> è un *Information Expert*
- 2) **Information Expert** - vedi discorso precedente per l'appunto
- 3) **Low Coupling** (pattern valutativo, è più un principio) - Come supportare basse dipendenze, low change impact e aumentare la possibilità di riuso? Scegliere le classi assegnando le responsabilità a chi ha un livello di accoppiamento basso (meno dipendenze possibile)
- 4) **Controller** - Quale oggetto coordina le operazioni di sistema? Una classe che fa interagire l'attore con il sistema: ovvero il sistema dovrebbe “sbattere” contro una classe creata apposta che serve per intercettare interazioni con l'utente.
- 5) **High Cohesion**(pattern valutativo, è più un principio)- Come mantengo gli oggetti comprensibili, amministrabili supportando il

Low Coupling? Assegnando le responsabilità facendo in modo che la coesione rimanga alta (ovvero lo do a chi ha responsabilità per lo più dello stesso tipo).

30/10/2019

- 6) **Polymorphism** - le variazioni condizionali (switch, if else...) per il controllo del flusso producono codice che è difficile da estendere. Bisogna evitare di fare il controllo del flusso con queste, perché altrimenti violo OCP (il codice funziona ma se un domani devo modificare if(colore), per esempio, dovrò aggiungere un if nuovo per ogni modifica in aggiunta di colore).
L'alternativa a questo è l'utilizzo del polimorfismo: vedi le reference in java: a seconda del tipo quando io chiamo un metodo su un oggetto, esso farà qualcosa di diverso a seconda di chi esso sia (tipo).
- 7) **Pure Fabrication** - Cosa fare quando non si desidera violare High Cohesion e il Low Coupling, o altri obiettivi, ma le soluzioni offerte dagli Expert (per esempio) non sono adeguate? Bisogna assegnare alta coesione a un insieme di responsabilità che vengono date a una classe artificiale o conveniente all'uso, che non fa parte del dominio del problema, ma è stata concepita come una classe per il supporto di questi principi.
- 8) **Indirection** - Questo principio è in realtà una diretta applicazione di quello precedente: ovvero per esempio io voglio assegnare la responsabilità ad un intermediario così da non essere direttamente dipendente dall'altra classe e quindi evitare l'accoppiamento diretto tra due o più classi.



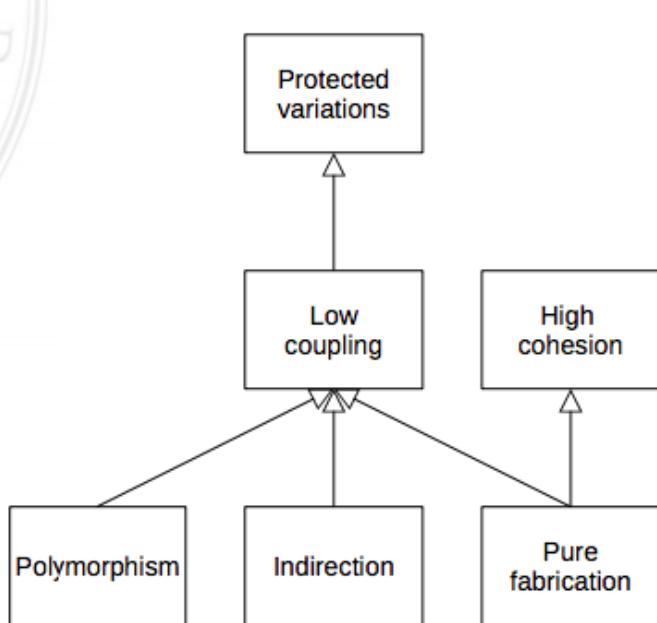
Quando ci guadago?

Quando Z è piuttosto volatile e X ha una interfaccia stabile per A, B e C, anche a fronte di eventuali cambiamenti in Z.

- 9) **Protected Variations** - I cambiamenti devono essere localizzati, poiché devo cercare entro un certo limite di proteggermi, spostando quindi le dipendenze necessarie verso elementi più stabili (Siamo certamente legati al principio di OCP). Questo concetto è anche legato alla Legge di Demetra: "non parlare con

gli stranieri": in programmazione a oggetti, in particolare, questo implica che un oggetto non dovrebbe interagire direttamente con (usare operazioni di) oggetti a cui accede solo indirettamente (attraverso operazioni o attributi dei suoi conoscenti diretti). Questo garantisce che un oggetto sia indipendente dalla struttura interna e dalle proprietà degli altri oggetti, compresi i loro eventuali componenti interni o le loro relazioni.

Riassunto schematico:



Esempio di applicazione principi (registratore di cassa a versioni)

Nonostante tutti questi accorgimenti per migliorare la qualità del prodotto siano perfettamente logici, rappresentano un problema: sono troppo pochi i progetti che arrivano in fondo!

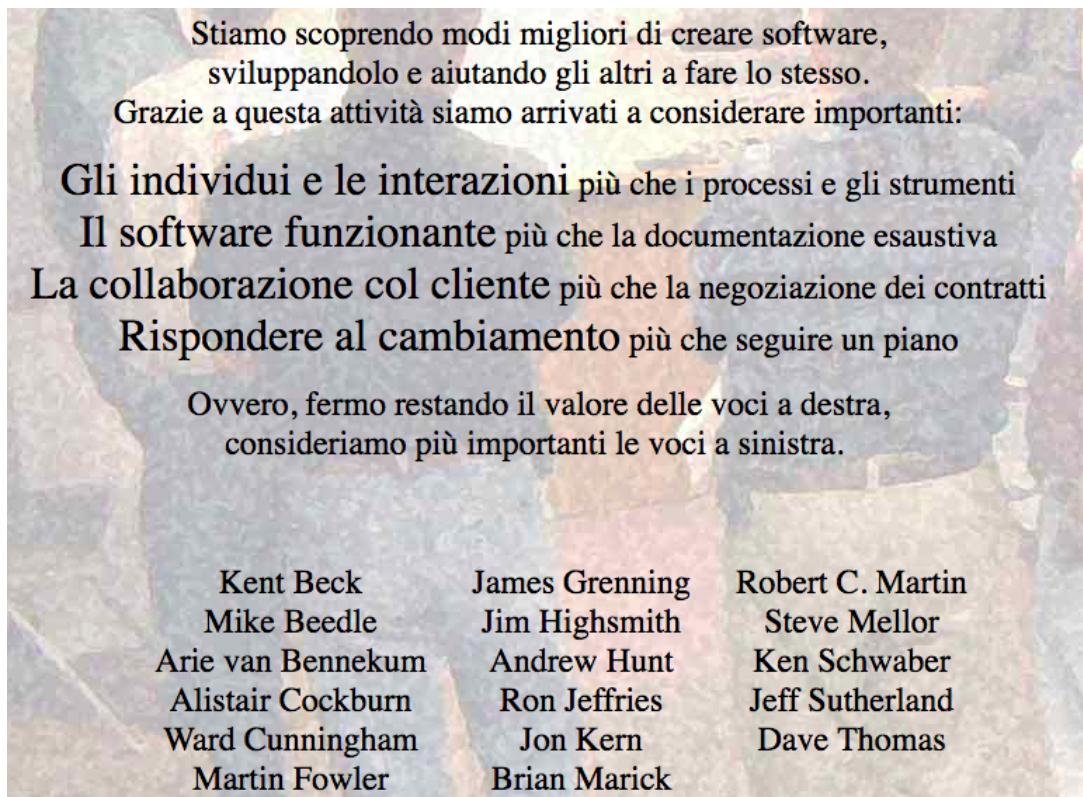
Bisogna migliorare il processo di miglioramento del prodotto, per essere più rapido e soprattutto lavori bene quanto le altre branche dell'ingegneria software.

Ecco che nasce la necessità di creare un insieme di pratiche di sviluppo software più "agili", facili da usare quando si sviluppa un sistema.

AGILE SOFTWARE DEVELOPMENT

Nell'ingegneria del software, l'espressione metodologia agile (o sviluppo agile del software, in inglese *agile software development*) si riferisce appunto a un insieme di pratiche di sviluppo del software emerse a partire dai primi anni 2000 e fondate su un insieme di pratiche comuni, direttamente o indirettamente derivate dai principi del "Manifesto per lo

sviluppo agile del software" (*Manifesto for Agile Software Development*) pubblicato nel 2001 da Kent Beck, Robert C. Martin, Martin Fowler e altri.



Chiarimenti: AGILE non è

- un processo
- un metodo di progettazione, ma piuttosto

è una collezione di pratiche guidate da un insieme di principi*.

*I principi di AGILE

1. La più alta priorità è quella di soddisfare il cliente coinvolgendolo nell'intero processo
2. le modifiche devono essere le benvenute anche in tardo sviluppo, poiché soprattutto rappresentano per il consumatore un vantaggio competitivo
3. Il software va rilasciato frequentemente, con intervalli di circa 2 settimane o al massimo al mese
4. costruire il progetto con persone motivate individualmente
5. Ad essi bisogna dare un ambiente e un supporto di cui hanno bisogno, ma soprattutto bisogna credere in loro per fare in modo che il loro compito sia svolto
6. il modo più efficiente per convogliare le informazioni è quello per cui il team debba fare conversazione faccia a faccia
7. Il lavoro sul software è la prima misura di progresso
8. Il processo AGILE promuove uno sviluppo sostenibile

9. Gli acquirenti, gli sviluppatori e gli utenti dovrebbero mantenere un rapporto di pace tra di loro
10. una continua attenzione per un'eccellente tecnica e una buona progettazione migliora l'agilità
11. La semplicità è essenziale
12. le migliori architetture e progettazioni emergono da un team in grado di auto organizzarsi
13. a intervalli regolari il team dovrebbe riflettere su come diventare più produttivo

Pratiche di AGILE

Ma quali sono le effettive e concrete pratiche che vengono messe in atto? Ce ne sono moltissime noi ne vedremo alcune:

- 1) **Test-Driven Design** - È uno stile di programmazione dove le principali attività di scrittura del codice vengono invertite:
 - codice
 - test
 - design
 Ovvero scrivo un singolo e unico test che descrive le specifiche richieste, su questo costruisco il programma che aggiusterò mano a mano.
 Procedura assicuramente il vantaggio che il codice esegue esattamente quello che deve fare e ovviamente il test è stato correttamente costruito.
- 2) **Code review** - Con questa pratica ovviamente intendiamo il fatto che un codice vada rivisto è corretto prima di essere consegnato. Questo permette dei benefici dal punto di vista della conoscenza, aumenta la consapevolezza del team, ma soprattutto la creazione di alcune possibili soluzioni alternative al problema.
- 3) **Pair programming: extreme code review** - Consiste in due programmatore che condividono una singola postazione di lavoro, ma il programmatore che lavora sulla tastiera è chiamato "driver", mentre l'altro comunque attivamente coinvolto Ma di più sulla direzione globale del lavoro è detto "navigator"; Ovviamente questi due devono cambiare di ruolo ogni tot minuti.

31/10/2019

- 4) **User stories** - Queste storie coincidono con delle frasi scritte nel dominio di linguaggio che catturano le aspettative dell'utente:
 As a <role>, I want <goal> so that <benefit> (sono di questo tipo) ->
 Essendo un impiegato voglio acquistare un pass così posso arrivare al lavoro guidando.

Ma sono di buona qualità le user stories?

Invest è un set di criteri, o una checklist, per valutarne la qualità:

- Sono indipendenti
- sono negoziabili ovvero sono facili da modificare per l'utente
- sono valutabili ovvero hanno un valore che possono offrire all'utente
- sono stimabili, ovvero Per quanto riguarda gli sviluppatori è necessario uno sforzo per realizzare quella determinata funzionalità delle user stories
- sono piccole
- sono testabili, fattibili e controllabili

Certo è che lo sviluppo è soltanto una parte del ciclo di vita del software, l'evoluzione di quest'ultimo è molto più importante:

- nell'approccio AGILE viene prodotta davvero poca documentazione

5) Extreme programming - XP che si basa invece su alcuni principi visti precedentemente

comprende 4 attività:

- codice
- test
- ascolto
- progettazione

comprende 5 valori:

- comunicazione
- test
- semplicità
- feedback
- coraggio
- rispetto

comprende 3 principi:

- feedback
- svolgere in semplicità
- abbracciare il cambiamento

comprende 4 gruppi di pratiche:

- 1) fine scale feedback
 - pair programming
 - planning game

- test driven development (prima di scrivere il test si scrive il codice, che ci permette di controllare se quello che c'era prima continua a funzionare)
- whole team (team unico "tutto è di tutti", molto vantaggioso)

2) continuous process

- continuous integration - si controlla il codice di qualcun altro per evitare bug
- design improvement (Re-factoring): noi abbiamo codice per far funzionare il test, per questa ragione design implementation è fondamentale: si ristruttura ogni volta che c'è uno smell (modifico il codice, ma non la funzionalità)
- small releases

3) shared understanding

- coding standard
- collective code ownership: il codice è di tutti per questo ha una sua sintassi standard
- simple design
- system metaphor: siccome la comunicazione è orale, si introducono dei nomi o si usano metafore per capire di cosa si sta parlando.

4) programmer welfare- utilizzare un passo sostenibile, come in una maratona

- sustainable pace

Sempre in XP si trova un processo di pianificazione: **Planning game**
È il processo di pianificazione in XP, si basa sulle storie degli utenti. Si tiene prima di ogni iterazione. Comporta 2 parti:

- I) Pianificazione di rilascio (include i clienti)
- II) Pianificazione di iterazione (solo sviluppatori).

I clienti ordinano le storie in base al valore (critico, valore commerciale significativo, bello da avere), e i programmatore in base al rischio tecnico.

05/11/2019

Apriamo argomento dei **Design Pattern**

Ogni pattern descrive un problema che si verifica ripetutamente nel nostro ambiente, e quindi descrive il nucleo della soluzione a quel problema, in modo tale da poter utilizzare la sua soluzione un milione di volte, senza mai farlo allo stesso modo due volte.

I modelli sono generalmente raggruppati in una struttura coerente con il proprio vocabolario, sintassi (contesto) e grammatica (uso). I modelli sono collegati, quindi l'adozione di uno suggerisce che l'altro deve prendere in considerazione.

Esistono diversi cataloghi esistenti per i pattern nell'ingegneria del software. *Ci concentreremo sui pattern di progettazione.*

Esistono diversi cataloghi esistenti anche per motivi di design. Presenteremo quelli del libro: Elements of Reusable Object-Oriented Software di Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the Gang of Four) - 1994.

Documentazione

- Applicabilità: situazioni in cui questo pattern è utilizzabile; il contesto per il modello.
- Struttura: una rappresentazione grafica del motivo. A tale scopo è possibile utilizzare diagrammi di classe e diagrammi di interazione.
- Partecipanti: un elenco delle classi e degli oggetti utilizzati nel pattern e dei loro ruoli nel progetto.
- Collaborazione: una descrizione di come le classi e gli oggetti utilizzati nel pattern interagiscono tra loro.
- Conseguenze: una descrizione dei risultati, degli effetti collaterali e dei compromessi causati dall'uso del modello.
- Implementazione: una descrizione di un'implementazione del modello; la parte soluzione del modello.
- Codice di esempio: illustrazione di come è possibile utilizzare il modello in un linguaggio di programmazione.
- Usi noti: esempi di utilizzi reali del modello.
- Pattern correlati: altri pattern che hanno qualche relazione con il pattern; discussione delle differenze tra il modello e modelli simili.

La maggior parte dei pattern in realtà è pure fabrication.

I pattern secondo la gang si dividono in 3 grandi famiglie:

- 1) pattern creazionali - viene reso astratto il processo di istanziazione
- 2) pattern strutturali - riguarda il come le classi e gli oggetti sono composti per formare strutture più larghe
- 3) pattern comportamentali - riguardano gli algoritmi e l'assegnamento di responsabilità tra oggetti

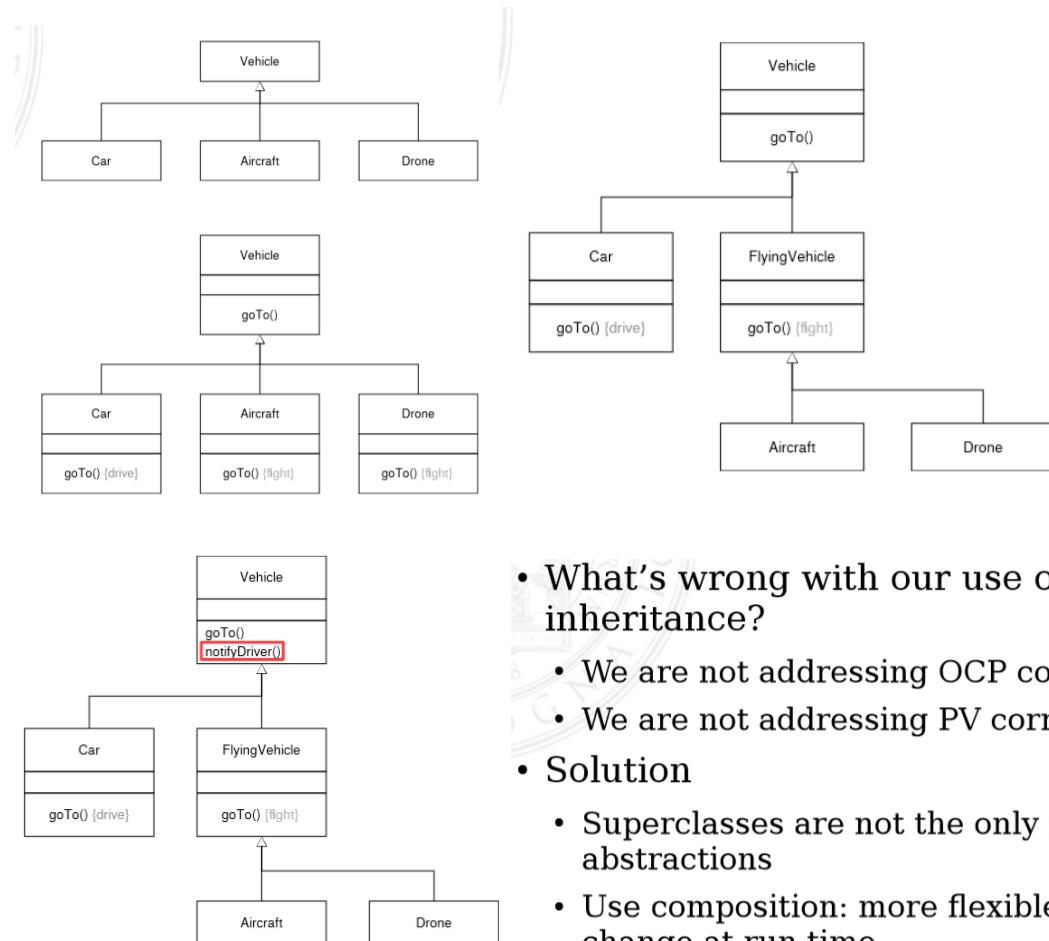
Potrei anche dividerli per scope o purpose e se da soli (formano classi) o in più gruppi (object):

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy		Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

totale = 23

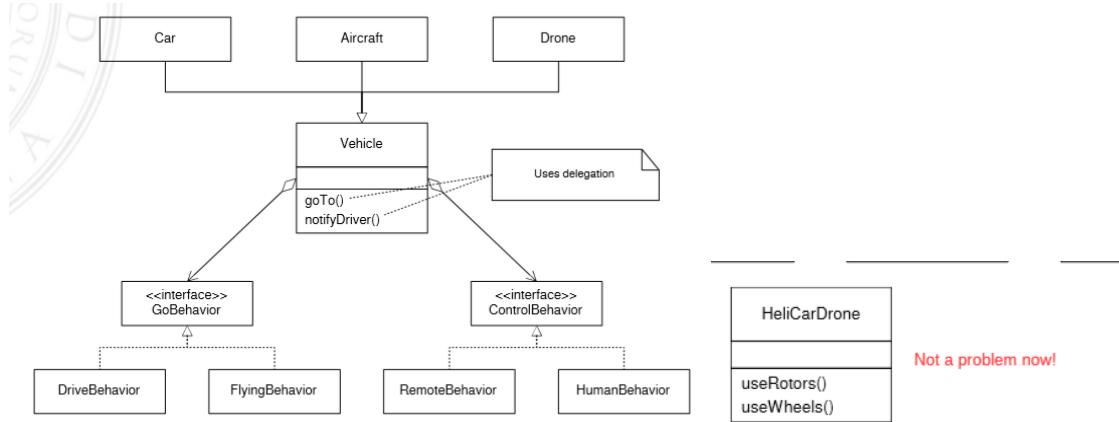
Non li faremo tutti. Ma comune a tutti è un principio a cui la gang resta fedele è usare composizione invece che ereditarietà.

Favorire la composizione degli oggetti rispetto all'ereditarietà delle classi aiuta a mantenere ogni classe encapsulata e focalizzata su un compito.



- What's wrong with our use of inheritance?
 - We are not addressing OCP correctly
 - We are not addressing PV correctly
- Solution
 - Superclasses are not the only possible abstractions
 - Use composition: more flexible and can change at run time

Allora è meglio fare:



Codice:

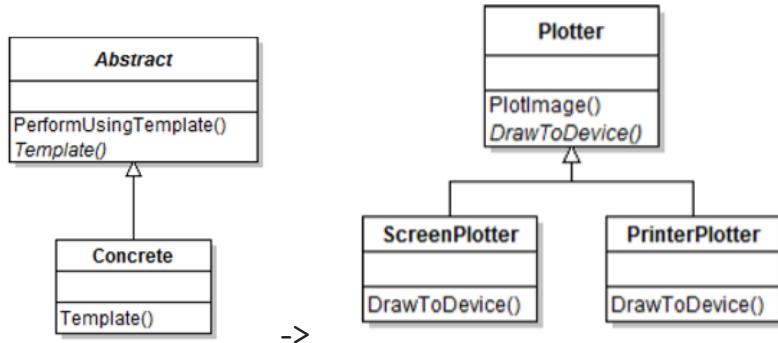
```

1  Class Vehicle {
2      GoBehaviour goBehaviour;
3      ControlBehaviour controlBehaviour;
4  }
5
6  void goTo() {
7      goBehaviour.goTo();
8  }
9
10 void notify() {
11     ControlBehaviour.notify();
12 }
13
14
15 Class Car extends Vehicle {
16     costruttore Car(){
17         this.goBehaviour = new DriveBehaviour();
18         this.controlBehaviour = new HumanBehaviour();
19     }
20 }
```

1) Pattern: Template Method - Comportamentale

Definire dei metodi astratti lasciando alle sottoclassi il compito di ridefinirli.

Esempio. Quicksort che deve ordinare non numeri semplici ma oggetti complessi. Scrivo quicksort poi avrò una classe che prende A e prende B esegue una comparazione, ma sarà astratta, la sottoclasse che compara farà in modo che a seconda dell'oggetto che passo la comparazione avverrà in modo diverso. Ridefinisco solo il suo di comportamento.

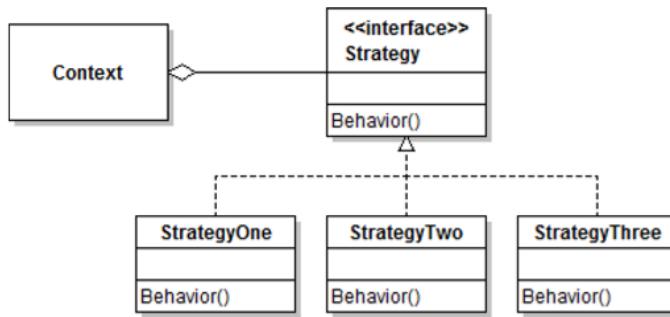


Plotter è specializzato da Screen e Printer, deleghiamo a Draw il disegno, che sarà specifico per ogni tipo di device.

2) Pattern: Strategy - Comportamentale

In certi casi si ha bisogno di oggetti che hanno comportamenti simili, ma con piccole modifiche ognuno.

Uso le interfacce e classi concrete: che ridefiniscono i comportamenti in maniera diversa.



Context se vuole attivare un comportamento delega a Strategy. Cambia oggetto a seconda della strategia che si deve usare.

QUA C'ERA IL TESTING

07/11/2019

Torniamo ai Design pattern.

Premessa per il pattern Factory:

New considered harmful

Partiamo dal presupposto che il new è pericoloso (se ho bisogno di una istanza di una classe l'idea è di generarla noi stessi, non col new).

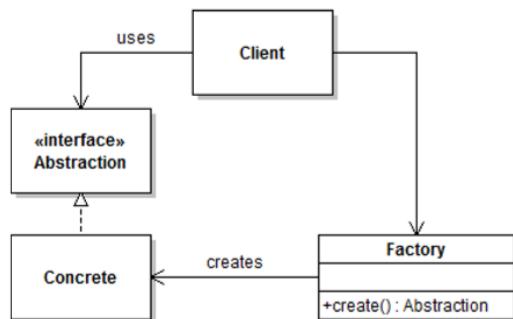
Magari perchè voglio riusarla, se non ho il garbage collector come in java, chi la distrugge se non mi serve più? Mi sta riempiendo memoria.

In generale è facile violare DIP, creare dipendenza tra classe e costruttore.

Molto spesso sono le classi di alto livello che, avendo bisogno di oggetti di basso livello, violano DIP.

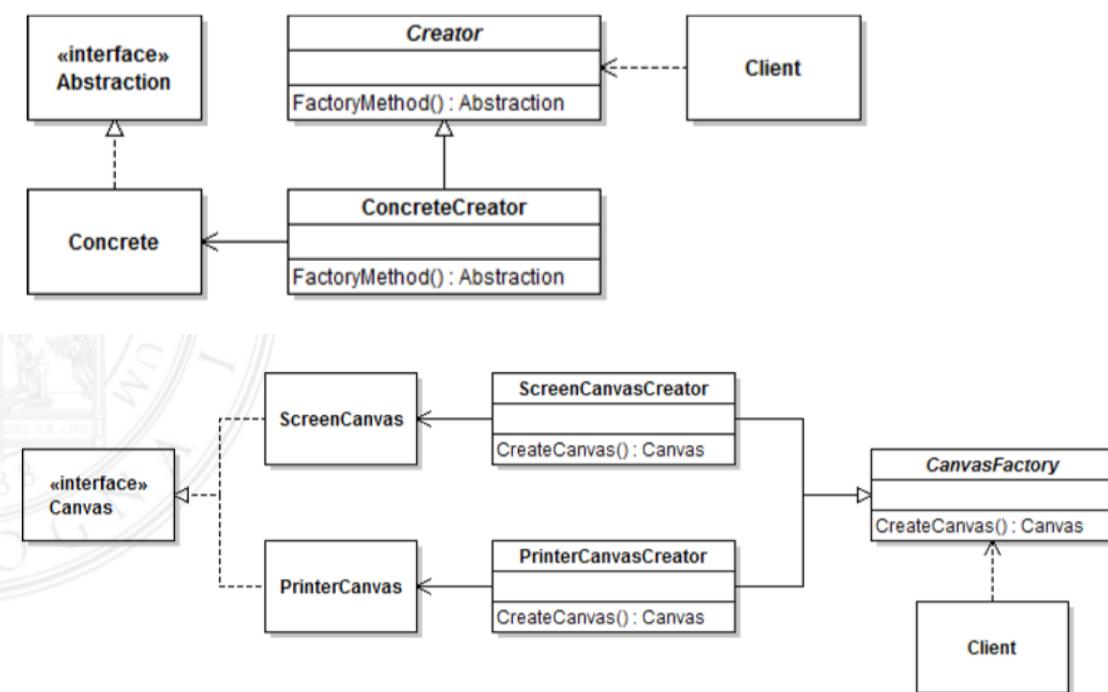
Al suo posto uso:

3) Pattern, ma non proprio, è un idioma: Factory - Creazionale



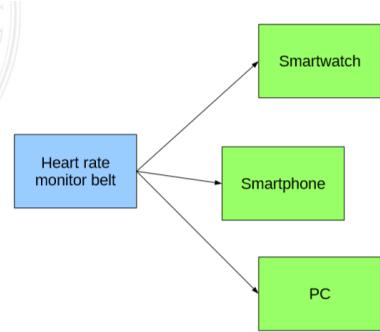
- Disaccoppia il client (colui che ha bisogno dell'oggetto) dall'instanziazione
- il client non dipende dalla classe dell'oggetto, ma da un'astrazione intermedia

Il factoryng può anche essere familiare (quindi il creator può essere concretizzato da più factory):



Quando ha senso "sforzarsi" per creare factory? Non ha senso usarlo se la classe di basso livello non cambia mai (se invece è piuttosto volatile sì).

Premessa per il 4 pattern: Notification Problem



Ho delle componenti del software interessate al cambiamento di stati di altre componenti.

Il numero degli “osservatori” potrebbe variare in numero o in tipo!

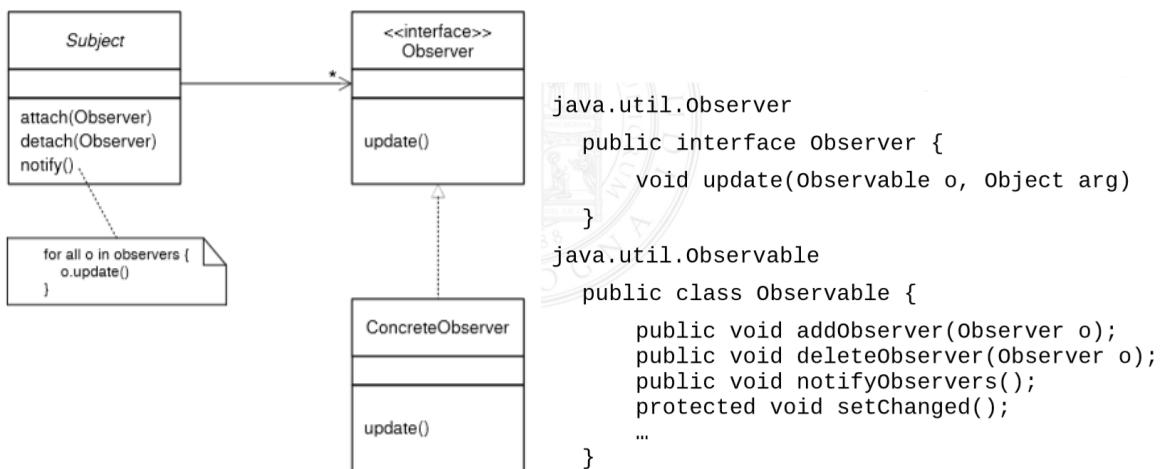
Se la fascia cardiaca va continuamente ad aggiornare il pc ecc. vuol dire che ogni volta richiama i metodi di ognuna delle classi interessate, che potrebbero avere nomi diversi! Ma soprattutto uno ad uno.

La fascia dipende dalle altre, ma alla fine chiama solo quel piccolo metodo: soluzione è metterlo in una interfaccia... problema? Sì, la mia dipendenza si è spostata.

Ecco che la soluzione è proposta dal 4 pattern:

4) Pattern: Observer - Comportamentale

Definire una dipendenza uno-a-molti tra gli oggetti in modo tale che quando un oggetto cambia stato, tutti i suoi dipendenti vengano notificati e aggiornati automaticamente. Questo usando una interfaccia semplicissima, dove però ad essa non interessa il tipo.

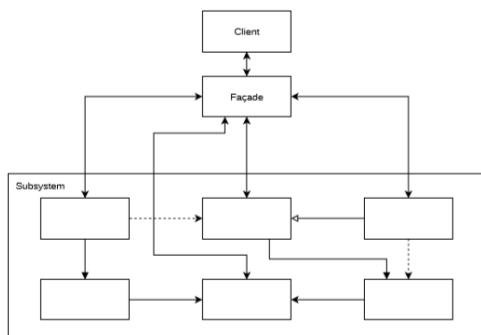


Se aggiungo un nuovo device non accade nulla, perchè definisco subject come classe astratta che ha dei metodi per gestire gli osservatori (attach() e detach()) anche se sono interessati all’aggiornamento.

Chiarimento: la cintura estende il subject, gli osservatori richiamani i due metodi attach e detach. Subject è astratta, non è la “cintura”, “cintura” è una specializzazione.

5) Pattern: Facade (leggi Fassade) - Strutturale

Fa riferimento ai problemi riguardanti i sottosistemi:



Di solito il client per dialogare con il sottosistema ne deve conoscere la struttura ("quale classe devo usare?"). Questo va contro l'idea di isolamento del sistema stesso.

Il client dovrebbe dialogarci tramite un mediatore: la classe Facade, è come una interfaccia unificata di alto livello, ma è una classe strutturata. Essa prende le invocazioni del client e le rimanda (richiama) all'oggetto opportuno), espone e basta non realizza (la realizzazione è nelle classi opposte) = ergo, sto violando SRP solo di facciata. Ha pochissime righe di codice al suo interno.

E' certamente vero che ho più dipendenze in questo modo, ma sono di qualità migliore e inoltre devo pensare al caso in cui io abbia più client.

Se anche cambia il sottosistema, è vero che la modifica a Facade è obbligatoria, ma si tratta di poche righe.

12/11/2019

6) Pattern: Singleton - Creazionale

Assicurati che una classe abbia solo un'istanza e fornisci un punto di accesso globale ad essa, questa viene creata dinamicamente la prima volta che faccio getInstance() della classe singleton.

E' facile sbagliarne l'uso:

- potrebbe essere un code smell
- essere sicuri che ci sia solo una istanza
- fornire un unico punto di accesso

Cosa può essere singleton? Candidati

- I. factories = oggetti che creano altri oggetti di diverso tipo
- II. loggers = componenti che tengono traccia dell'andamento del programma

- III. classi di configurazione = complesse
- IV. tutte le classi senza variabili di stato sono ragionevoli candidati, poiché contengono solo comportamenti
Cosa NON può essere singleton? NON Candidati
- I. Classi per le quali una singola istanza fa parte della specifica ma non è intrinseca al problema di dominio
- II. oggetti facilmente accessibili

Nella realtà si usa singleton solo per factories e loggers, che sono entità concrete che esistono in quantità limitate...

7) Pattern: Proxy - Strutturale

Si tratta di un pattern strutturale che viene utilizzato per accedere ad un oggetto complesso tramite un oggetto semplice.

Questo pattern può risultare utile se l'oggetto complesso:

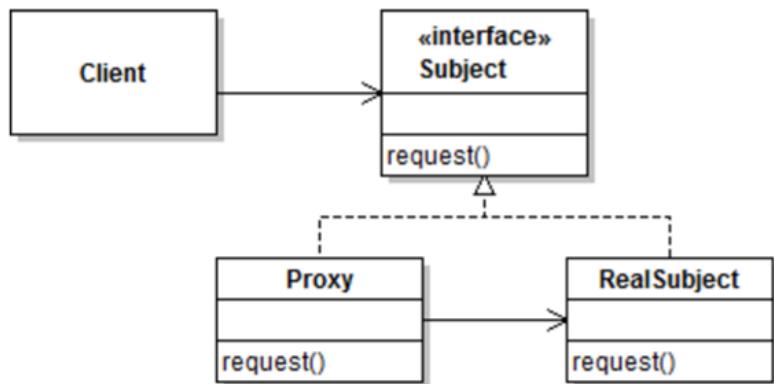
- richiede molte risorse computazionali
- richiede molto tempo per caricarsi
- è locato su una macchina remota e il traffico di rete determina latenze ed overhead
- non definisce delle policy di sicurezza e consente un accesso indiscriminato
- non viene mantenuto in cache ma viene rigenerato ad ogni richiesta

Come risolvo?

Questo pattern è composto dai seguenti partecipanti:

1. **Client**: colui che effettua l'invocazione all'operazione di interesse
2. **SubjectInterface**: definisce l'interfaccia utilizzata dal Client che viene implementata dal Proxy e dal RealSubject
3. **RealSubject**: definisce l'oggetto reale di cui il Proxy avrà il compito di surrogare.
4. **Proxy**: definisce la classe che avrà il compito di surrogare l'oggetto reale mantenendo una Reference a tale oggetto, creando e distruggendo l'oggetto ed esponendo gli stessi metodi pubblici dell'oggetto reale definiti dall'interfaccia.

Voglio che la comunicazione avvenga attraverso la mediazione di un surrogato, di un oggetto che sia invisibile agli occhi del client, un "intermediario". Il client è convinto di operare direttamente sull'oggetto che fornisce la funzionalità, ma in realtà dialoga con un intermediario che la stessa interfaccia dell'oggetto.



Perchè è utile?

- Controllo accessi
- Contatore accessi
- Accesso logger
- Accesso ad oggetti remoti (possibilmente con memorizzazione nella cache)
- Riferimenti intelligenti (contatore di riferimento, caricamento di un soggetto persistente su richiesta, controllo blocco, ...)

Client

```

aggiungiImpiegato (List<Impiegato> impiegati, nome, ... ) {
    Impiegato i = ... ;
    impiegati.add(i);
}

uso il proxy che manda alle client interfacce
ma richiamata es clone cachea
e fa svolgere alle clone le
funzionalita richiesta dal
client.

```

è client richiamando add, in realtà il proxy
fa svolgere l'operazione
alle cloni list.

Access logger

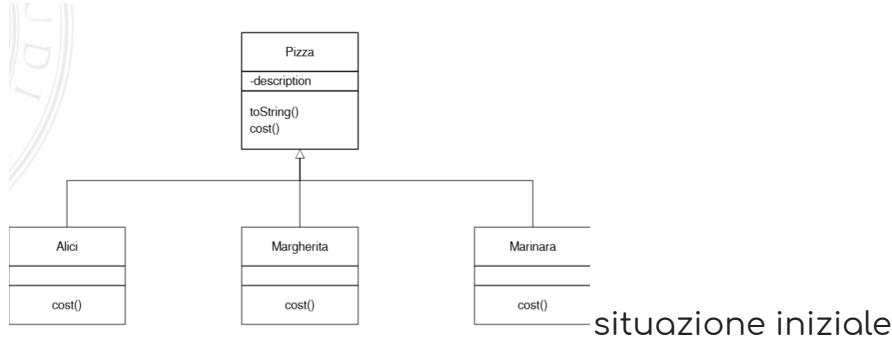
8) Pattern: Decorator - Strutturale

Premessa:

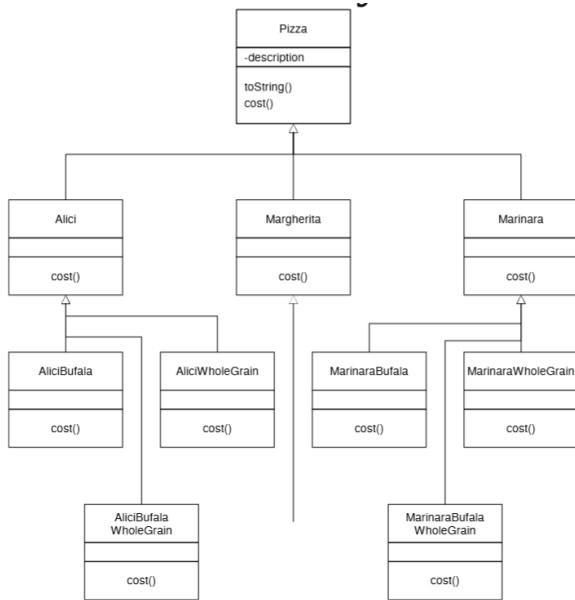
Aggiungere dinamicamente nuove funzioni, opzioni ad un oggetto.

Pensiamo alla pizza :) ci sono tanti tipi di pizza. Pizza è una superclasse che racchiude un comportamento comune che il costo. Il costo deve esistere per ogni pizza.

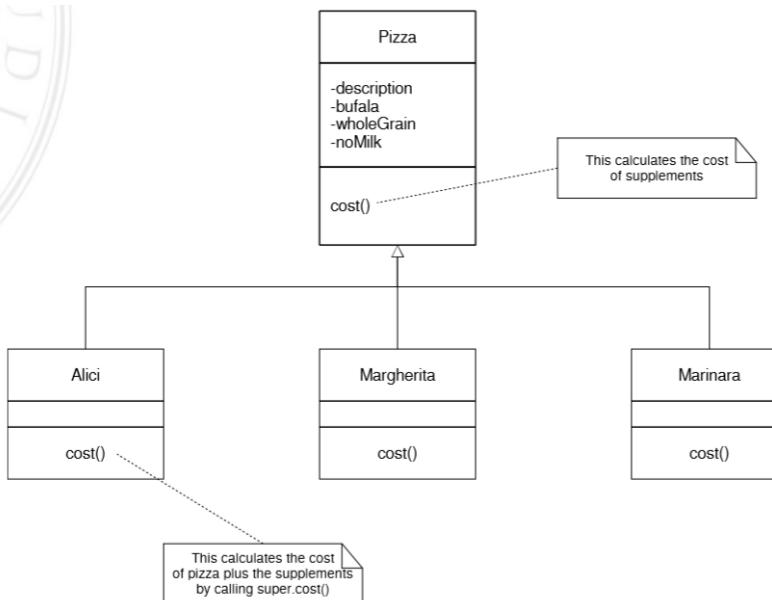
Il problema è che pizza potrebbe differenziarsi per impasto, ingredienti cosa che complicherebbe all'infinito una gerarchia!



situazione iniziale



che degenera. Posso risolvere?



Provo a cambiare:

pizza è la classe base, ma che si specializza in funzione del tipo di pizza e basta. La gestione dei supplementi avviene definendo le variabili all'interno della classe base e che modificano di conseguenza il costo. Non così perchè non riesco a creare nuove opzioni in modo semplice e

cambiare un prezzo per un'opzione ma neanche salvare casi tipo "doppia bufala", inoltre sto violando OCP.

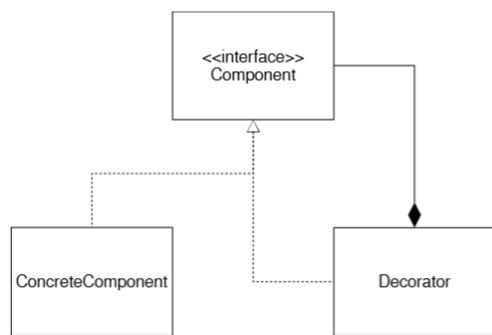
In Java, e più in generale nella programmazione ad oggetti, per aggiungere delle funzionalità ad una classe viene utilizzata l'ereditarietà che prevede la creazione di classi figlie che specializzano il comportamento della classe padre, tutto ciò a compile Time!

Pertanto se in sede di definizione della struttura delle classi non vengono previste delle specifiche funzionalità, queste non saranno disponibili a RunTime. Al fine di superare questo limite, attraverso la decorazione è possibile aggiungere nuove funzionalità senza dover alterare la struttura delle classi ed i rapporti di parentela in quanto è possibile agire a RunTime per modificare il comportamento di un oggetto.

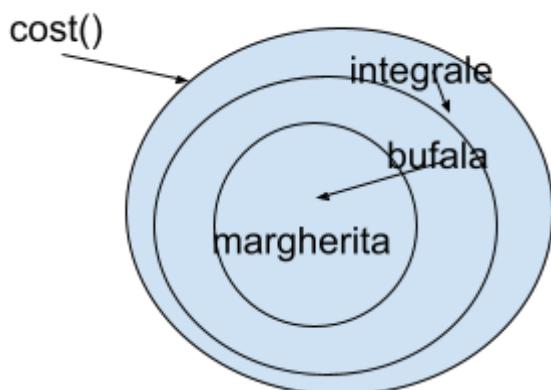
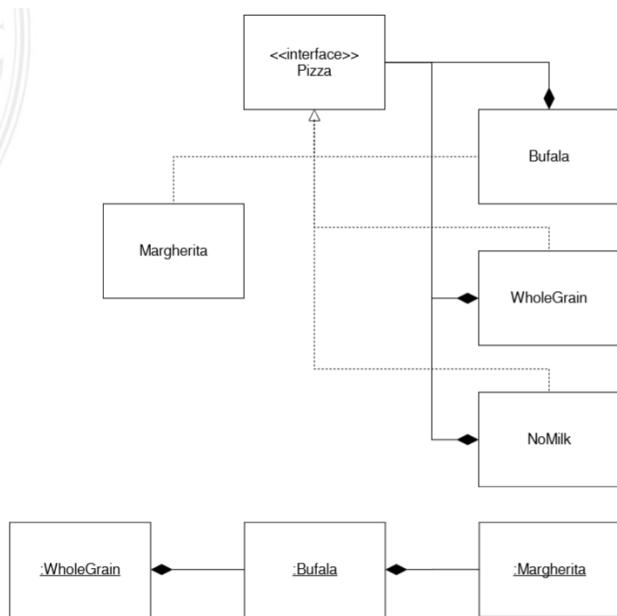
Decorator offre un'alternativa flessibile alla sottoclasse per estenderne la funzionalità.

Questo pattern è composto dai seguenti partecipanti:

1. **Client**: colui che effettua l'invocazione alla funzionalità di interesse
2. **Component**: definisce l'interfaccia degli oggetti per i quali verranno aggiunte nuove funzionalità
3. **ConcreteComponent**: definisce un oggetto al quale verrà aggiunta una nuova funzionalità
4. **Decorator**: definisce l'interfaccia conforme all'interfaccia del Component e mantiene l'associazione con l'oggetto Component
5. **ConcreteDecorator**: implementa l'interfaccia Decorator al fine di aggiungere nuove funzionalità all'oggetto.



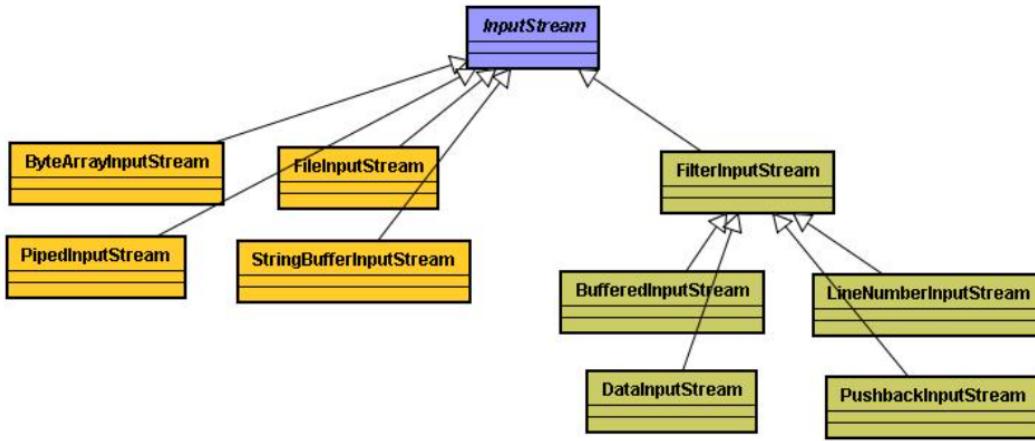
ecco che la Pizza diventa:



pizza è la nostra interfaccia, abbiamo decorators per ogni opzione (bufala, integrale e margherita) chiamo la funz. cost() sull'oggetto più esterno, che è integrale (client vede il decorator più esterno).

Tale pattern presenta i seguenti vantaggi/svantaggi:

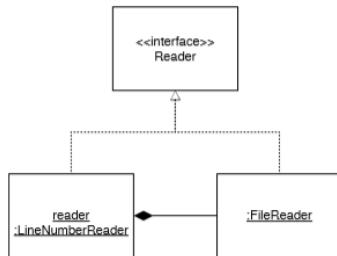
- maggior flessibilità rispetto alla eredità:** permette di aggiungere funzionalità in modo molto più semplice rispetto all'ereditarietà
- funzionalità solo se richieste:** consente di aggiungere delle funzionalità solo se occorrono realmente senza ereditare una struttura di classi che prevede un insieme di funzionalità di cui se ne utilizzeranno solo una parte.
- aumento di micro-funzionalità:** la presenza di molte classi Decorator di cui ognuna di esse aggiunge una micro funzionalità, può creare problemi in fase di comprensione o di debug del codice.



java.io.Reader

```

Reader reader =
new LineNumberReader(
    new FileReader("myfile"));
  
```



Un esempio noto del pattern Decorator lo troviamo nelle librerie java ed esattamente nelle classi di `java.io.InputStream` in cui i partecipanti sono così suddivisi:

- **Component:** la classe astratta `InputStream`
- **ConcreteComponent:** le classi `ByteArrayInputStream`, `FileInputStream`, `ObjectInputStream`, `PipedInputStream`, `SequenceInputStream` e `StringBufferInputStream`
- **Decorator:** la classe `FilterInputStream`
- **ConcreteDecorator:** le classi `BufferedInputStream`, `DataInputStream`, `LineNumberInputStream` e `PushbackInputStream`

L'utilizzo di questo pattern consente di poter scegliere la funzionalità di nostro interesse quando occorre leggere uno stream di dati, per esempio utilizzando `BufferedInputStream` è possibile bufferizzare lo stream.

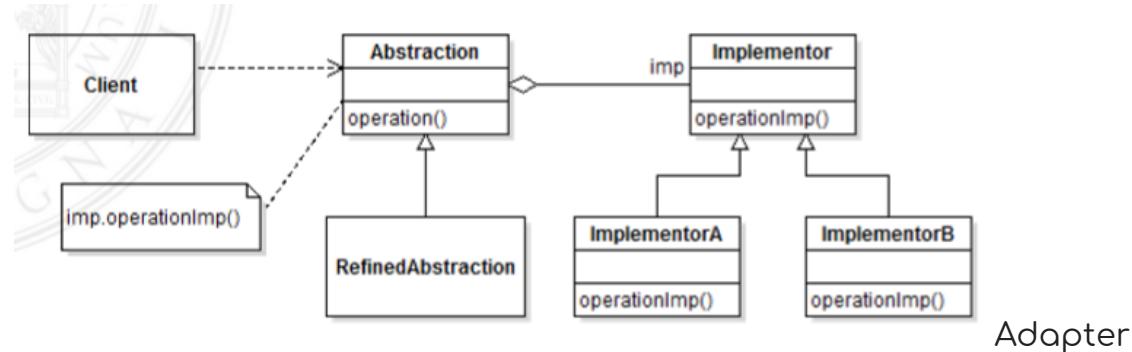
13/11/2019

9) Pattern: Bridge - Strutturale

Disaccoppia un'astrazione dalla sua implementazione in modo che i due possano variare in modo indipendente. Viene introdotto all'inizio dell'implementazione così da permettere a tutti di variare e non creando situazioni di dipendenza di basso livello. Possiamo dire che Bridge annulla la "tirannia" del client, inserendo un punto di discontinuità, per cui la classe di basso livello avrà la sua interfaccia e anche la classe di alto livello la sua. Bridge quindi si pone in mezzo,

come un ponte tra le due interfacce, mettendole in comunicazione e non violando DIP.

E' comunque un elemento volatile (cambia ogni volta che una delle due interfacce cambia).



makes things work after they're designed; Bridge makes them work before they are.

QUA C'ERA SCRUM E KANBAN

19/11/2019

10) Pattern: State - Comportamentale

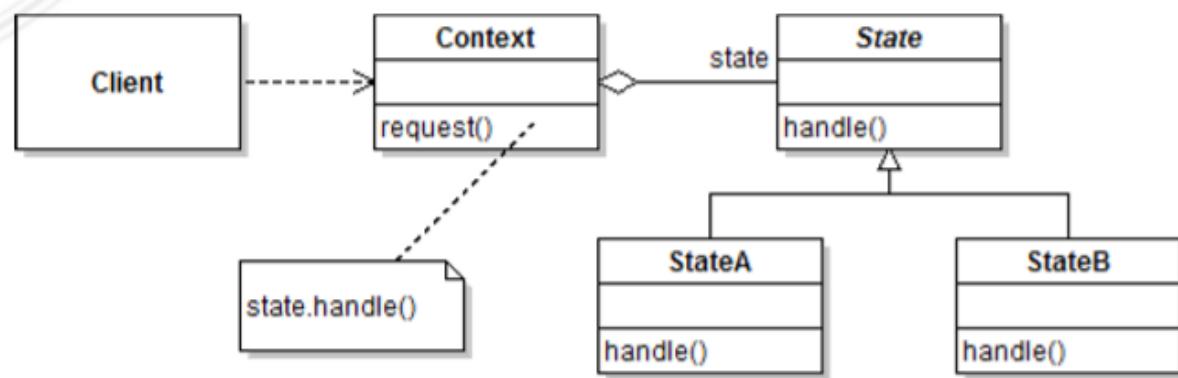
Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando il comportamento di un oggetto deve cambiare dinamicamente in base al suo stato.

Solitamente quindi, il codice è pieno di istruzioni condizionali che esaminano lo stato degli oggetti, in particolare esaminano il valore di variabili e costanti al fine di prendere delle decisioni. Molto spesso queste decisioni sono molto complesse e dipendono da molti valori, tutto ciò determina la generazione di grossi blocchi IF-ELSE/SWITCH che spesso contengono anche logica di business. Questo comporta seri problemi di comprensione, manutenzione ed evoluzione del codice.

L'utilizzo di questo pattern permette di scorporare i grossi blocchi condizionali ed inserirli negli oggetti di stato, in modo da associare il comportamento di un oggetto al suo stato.

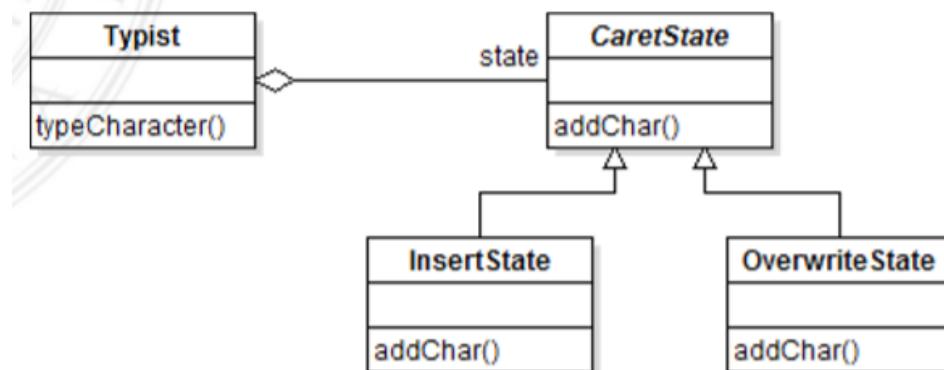
La request del client viene delegata ad un oggetto detto state. Ma la selezione della specifica classe concreta è in funzione dello stato dell'oggetto.

Vedi esempi:



- **Context:** definisce l'interfaccia di interesse del Client e mantiene una istanza della ConcreteState che definisce lo stato corrente
- **State:** definisce un interfaccia per incapsulare il comportamento associato con un particolare stato.
- **ConcreteState:** ogni sub-classe che implemento un comportamento associato con uno stato.

Esempio con dati:



Esempio con editor di testi: permette la sovrascrizione o l'inserimento.
L'oggetto realizza comportamenti diversi in dipendenza dello stato di typist, se insert o overwrite.

11) Pattern: Mediator - Comportamentale

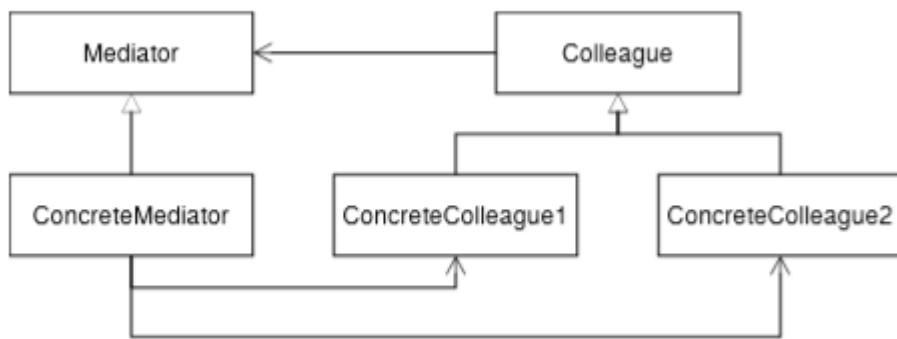
Pattern utilizzato per gruppi di oggetti che dialogano in maniera immediata, ma che possono implementare classi differenti.

Viene utilizzato per permettere lo scambio di messaggi tra diversi attori tramite un intermediario. In questo modo gli attori sono collegati indirettamente tramite un intermediario.

Ogni collega non ha reference diretta, ma fa riferimento a mediatore.

Questo pattern è composto dai seguenti partecipanti:

- **Mediator:** definisce una interfaccia per comunicare con i Colleague
- **ConcreateMediator:** mantiene la lista dei colleghi e implementa lo scambio di messaggi tra di loro
- **Colleague:** definisce l'interfaccia dei Colleague
- **ConcreateColleague:** implementa il singole collega e le modalità di comunicazione con il Mediator.



Tale pattern presenta i seguenti vantaggi/svantaggi:

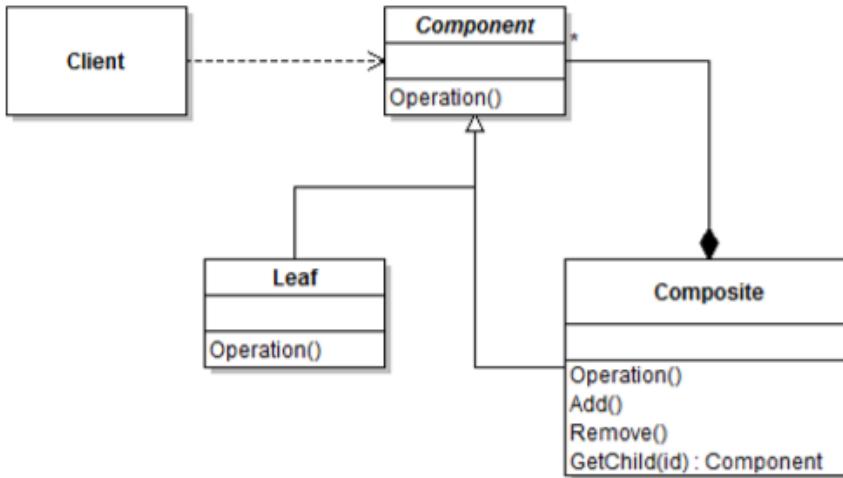
- Disaccoppiare i colleghi: i colleghi dialogano tra di loro in modo indiretto passando per il Mediatore e questo facilita la gestione delle comunicazioni
- Semplificare le connessioni: il Mediatore consente di ridurre le connessioni dei Colleghi da many-to-many a one-to-many
- Controllo centralizzato: il controllo delle comunicazioni è centralizzato e questo consente di avere una visione complessiva del sistema ed una gestione più efficiente delle modifiche
- Single Point of Failure: nel caso di malfunzionamento del Mediatore l'intero sistema sarà coinvolto ed in caso di fermo del Mediatore, i Colleghi resteranno isolati

12) Pattern: Composite - Strutturale

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato quando si ha la necessità di realizzare una gerarchia di oggetti in cui l'oggetto contenitore può detenere oggetti elementari e/o oggetti contenitori. L'obiettivo è di permettere al Client che deve navigare la gerarchia, di comportarsi sempre nello stesso modo sia verso gli oggetti elementari e sia verso gli oggetti contenitori.

Questo pattern è composto dai seguenti partecipanti:

1. Client: colui che effettua l'invocazione all'operazione di interesse
2. Component: definisce l'interfaccia degli oggetti della composizione.
3. Leaf: rappresenta l'oggetto foglia della composizione. Non ha figli. Definisce il comportamento "primitivo" dell'oggetto della composizione
4. Composite: definisce il comportamento degli oggetti usati come contenitori ed detiene il riferimento ai componenti "figli"



Esempio

classico di questo pattern è l'albero come struttura.

13) Pattern: Memento - Comportamentale

Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando si ha necessità di ripristinare lo stato di un oggetto ad un suo precedente stato. Ciò richiede di memorizzare gli stati pregressi di un oggetto per poterli eventualmente ripristinare.

Memento, in latino, vuol dire ricordare ed infatti l'obiettivo del pattern è quello di ricordare gli stati precedenti per poterli ripristinare all'occorrenza in un tempo successivo.

Senza violare l'incapsulamento, voglio catturare ed esternalizzare uno stato interno di un oggetto così che l'oggetto possa esservi riportato più tardi.

Il punto è che se dovessi interrogare ogni oggetto, probabilmente esso non mi darà queste informazioni, perchè per incapsulamento espone solo alcune cose.

Allora:

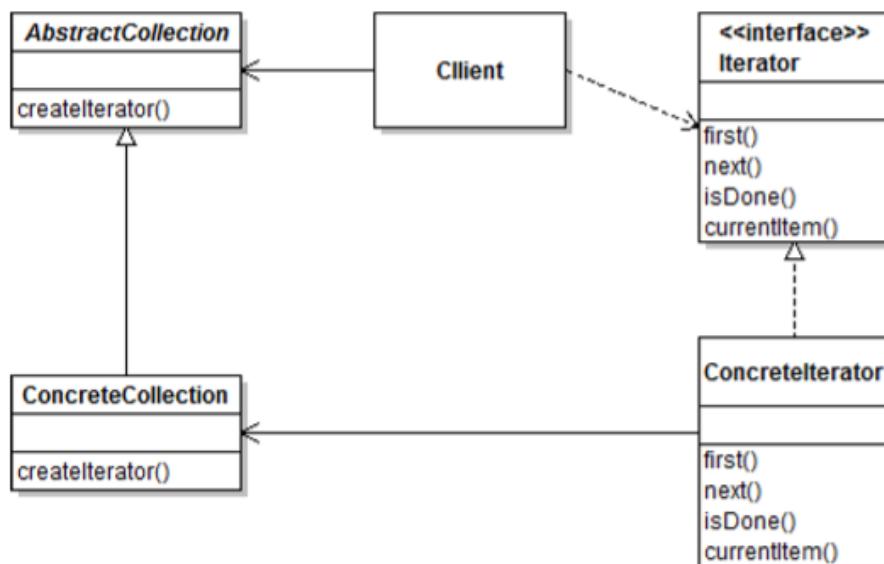
- vado dall'oggetto e gli delego questa responsabilità? No, funziona per incapsulamento, ma violo SRP.
- vado dall'oggetto, gli chiedo una rappresentazione del suo stato, un blob -> stringa o array opaca che codifica lo stato e così riesco a bypassare SRP (lo violo, però non è oneroso - non scrivo su disco).

14) Pattern: Iterator - Comportamentale

Si tratta di un pattern comportamentale basato su oggetti e viene utilizzato quando, dato un aggregato di oggetti, si vuole accedere ai suoi elementi senza dover esporre la sua struttura.

L'obiettivo di questo pattern è quello di disaccoppiare l'utilizzatore e l'implementatore della collezione di dati, tramite un oggetto intermedio che esponga sempre gli stessi metodi indipendentemente dalla collezione di dati.

Collezioni diverse hanno iteratori diversi, ma tutti quanti realizzano la stessa interfaccia.



Sto rispettando

OCP cambiando pezzi di codice senza che gli altri lo vedano (se il client non fa usi specifici, ma solo rassegna) il client rimane invariato.

Questo pattern è composto dai seguenti partecipanti:

- Iterator: colui che espone i metodi di accesso alla struttura dati
- Concreteliterator: implementa l'Iteratore e tiene il puntatore alla struttura dati
- Aggregator: definisce l'interfaccia per creare un oggetto di tipo Iteratore
- ConcreteAggregator: implementa l'interfaccia di creazione di un oggetto Iteratore

15) Pattern: Visitor - Comportamentale

Voglio poter operare su tutti gli elementi della collezione di dati, ma che l'operazione agisca su tutti gli elementi.

Questo pattern è basato su Inversional Control (ovvero in iterator sono io che chiamo hasNext() per esempio: qui definisco un'operazione che lo faccia per me).

Oggetti diversi della collezione possono avere visitors differenti.

```

class java.nio.file.Files {
    public static Path walkFileTree
        (Path start, FileVisitor<? super Path> visitor)
    ...
}

interface java.nio.file.FileVisitor {
    visitFile(T file, BasicFileAttributes attrs)
    ...
}

```

walk per ogni
elemento richiama visit file che poi aggiusto io, ma è lui che lo fa!

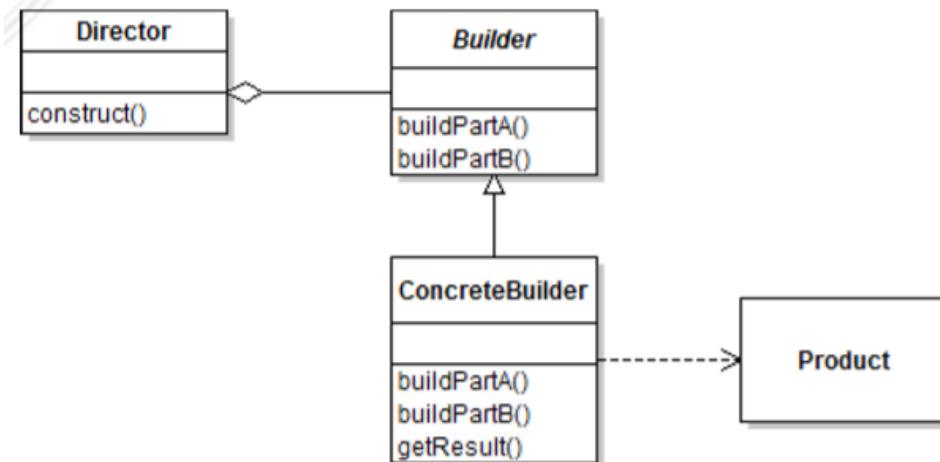
16) Pattern: Builder - Creazionale

Si tratta di un modello che viene utilizzato per creare un oggetto senza dover conoscere i suoi dettagli implementativi.

Fa riferimento a brutti costruttori:

`Foo foo = new Foo(a, b, null, null, c, null, d)` sono null perchè magari inizialmente sono inizializzati a null o perchè uso un valore di default.

Questo modello fornisce un client che non deve essere una conoscenza dei passi necessari per la creazione di un oggetto ma tali passaggi vengono delegati e un Director che sa cosa e come fare.



L'oggetto

specializzato inizializza lui altri oggetti, mette insieme i parametri necessari:

`Foo foo = Foo.Builder.createBuilder().setA(a).setB(b).setC(c).setD(d).build()`

17) Pattern: Command - Comportamentale

Viene usato quando si ha la necessità di disaccoppiare l'invocazione di un comando dai suoi dettagli implementativi, separando colui che invoca il comando da colui che esegue l'operazione.

Tale operazione viene realizzata attraverso questa catena:

Client-> Invocatore-> Ricevitore

- Il client non è tenuto a conoscere i dettagli del comando ma il suo compito è solo quello di chiamare il metodo dell'invocatore che si occuperà di intermediare l'operazione.
- L'invocatore ha l'obiettivo di encapsulare, nascondere i dettagli della chiamata come nome del metodo e parametri.
- Il Ricevitore utilizza i parametri ricevuti per eseguire l'operazione

Ma tra l'invocatore e il Ricevitore viene posto il Command ovvero il comando da eseguire. Il command è una semplice interfaccia che viene implementata da una o più classi concrete che invocano il ricevitore.

18) Pattern: Abstract Factory - Creazionale

Viene utilizzato per creare delle famiglie di oggetti interconnessi senza avere la necessità di conoscere dettagli implementativi.

L'intento del pattern è quello di creare delle interfacce che, attraverso l'implementazione di classi concrete, consentano di lavorare con una varietà di elementi che presentano le stesse funzionalità ma con diverse implementazioni.

19) Pattern: Prototype - Creazionale

Viene utilizzato per creare un nuovo oggetto clonando un oggetto già esistente detto prototipo. Questo pattern risulta utile affinché il Client possa creare nuovi oggetti senza conoscerne i dettagli implementativi ma avvalendosi della clonazione. La creazione del clone avviene a RunTime e non a CompileTime pertanto il clone viene creato in sede di esecuzione.

In Java c'è Clone

20) Pattern: Flyweight - Strutturale

Viene utilizzato per ottimizzare l'utilizzo delle risorse ed evitare la presenza di oggetti duplicati, spesso viene associato a richieste di performance in quanto il suo utilizzo può migliorare le prestazioni di una applicazione.

In particolare in Java, ogni qual volta che viene utilizzato l'operatore new, la JVM alloca nell'heap uno spazio in memoria di 32 bit ed una eccessiva generazione di oggetti può saturare le risorse del sistema che, anche se elevate, sono pur sempre limitate. Molto spesso la generazione di nuovi oggetti non è motivata da reali esigenze mentre invece è dovuta a superficialità oppure a errata analisi degli impatti prestazionali. Da qui l'esigenza di riutilizzare gli oggetti precedentemente creati ai fini del loro riutilizzo.

21) Chain of Responsibility - Comportamentale

Viene utilizzato quando si ha la necessità di disaccoppiare il mittente di una richiesta dal destinatario.

Il destinatario prevede che le richieste debbano essere gestite da una serie di attori, ognuno con diversa responsabilità e tra loro collegati in modo gerarchico = catena di responsabilità. Ogni elemento della lista è collegato con dei puntatori uno al successivo.

A fronte della ricezione di una richiesta, il destinatario gestirà la risposta propagando la richiesta nella catena fino ad individuare il responsabile.

Il mittente non è tenuto a conoscere chi materialmente dovrà gestire/eseguire la richiesta, l'unica cosa che dovrà sapere è a chi dovrà inviare la richiesta. Sarà cura del destinatario organizzarsi in modo efficiente per recuperare il responsabile.

Pensiamo per esempio ad un call-center che deve gestire le richieste delle proprie utenze, qualora il personale non è in grado di risolvere il problema, propaga la richiesta al servizio di secondo livello che proverà a gestire/risolvere il problema altrimenti propaga a sua volta il problema.

In Java la propagazione delle eccezioni è un esempio di catena di responsabilità. Quando si verifica un errore, il gestore dell'errore, se non riesce a gestire l'eccezione in corso, propaga l'errore nella catena.

22) Pattern: Interpreter - Comportamentale

Data una lingua, definire una rappresentazione per la sua grammatica insieme a un interprete che utilizza la rappresentazione per interpretare le frasi nella lingua.

23) Pattern: Adapter - Strutturale

Si tratta di un pattern strutturale basato su classi o su oggetti in quanto è possibile ottenere entrambe le rappresentazioni. Viene utilizzato quando si intende utilizzare un componente software ma occorre adattare la sua interfaccia per motivi di integrazione con l'applicazione esistente.

14/11/2019

SCRUM

E' un framework all'interno del quale le persone possono affrontare complessi problemi di adattamento, offrendo in modo produttivo e creativo prodotti con il massimo valore possibile.

Viene usato in molti contesti ma soprattutto per lo sviluppo di prodotti software.

Scrum fa parte del mondo agile. Non è un metodo di sviluppo completo, tuttavia: il focus principale è sulla gestione del progetto

- Scrum può (e di solito è) adottato CON altri metodi di sviluppo (integrazione o sostituzione di pratiche relative al progetto)

la parola scrum = vuol dire mischia nel rugby, i componenti si supportano a vicenda per l'avanzamento.

La sua struttura:



- iterazioni time-boxed chiamate sprint di 2-4 settimane
- all'interno di sprint si realizzano dei task definiti all'interno del ciclo. Vengono determinati popolando lo sprint backlog a partire dal product backlog.
- si produce un PSPI, che mano a mano viene decrementato

Per portare avanti il ciclo si prevede una struttura a 3 elementi principali:

1. Ruoli

Si differenziano in:

- *core* (amministrativi, sempre presenti: master owner e team development) - pigs
- *additional* (clienti e manager esecutivo) - chickens

In particolare:

- il product owner rappresenta lo stakeholder, ne riporta la voce, decide le priorità scadenze e stabilisce le funzionalità. Può accettare o rifiutare il lavoro consegnato mano a mano.
- il team è piccolo, ma cross funzionale, tutti sono esperti di tutto. Si auto-organizza, non c'è un gestore o coordinatore di progetto. Sono i singoli membri del team a prendersi il loro lavoro.
- il scrum master, è un leader che controlla il team, ma è solo responsabile per la corretta applicazione di scrum e ogni tanto fa delle riunioni.

2. Eventi

Esistono 4 tipi di eventi:

- sprint planning - evento all'inizio di ogni sprint: Il proprietario del prodotto definisce un obiettivo e presenta gli elementi correlati più importanti dal backlog del prodotto e ciascuno viene discusso per dettagliarlo e stimare lo sforzo richiesto. Gli articoli selezionati vengono suddivisi in attività e il backlog di sprint viene popolato.
- daily scrum - breve riunione per aggiornare sullo stato di avanzamento. le domande da porsi:
 - What did I do yesterday that helped the Development Team meet the Sprint Goal?
 - What will I do today to help the Development Team meet the Sprint Goal?
 - Do I see any impediment that prevents me or the Development Team from meeting the Sprint Goal?
- sprint review - Viene presentato l'incremento insieme a problemi e soluzioni. Viene discusso il portafoglio ordini del prodotto; cronologia, budget e capacità sono rivisti.
- sprint retrospective - Vengono discussi i miglioramenti del processo per i prossimi sprint.

3. Artefatti

Si differenziano in:

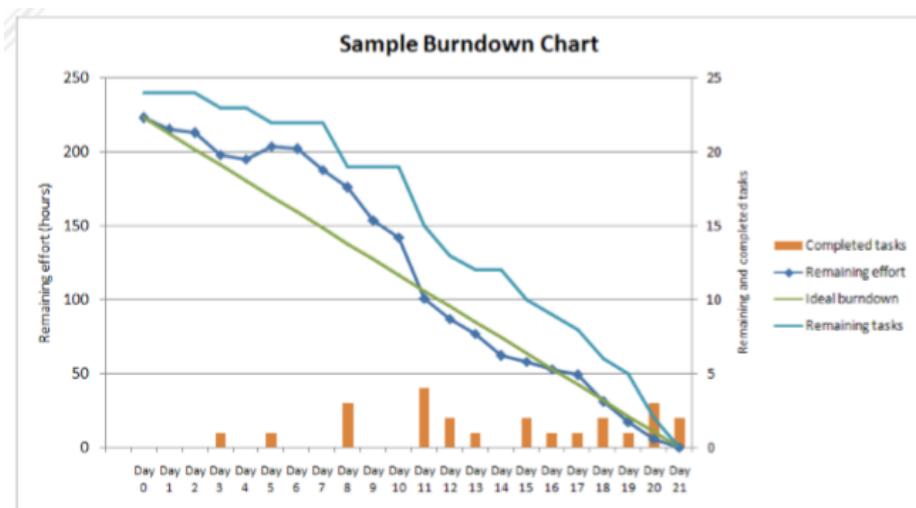
- *product backlog* - meccanismo per la gestione dei requisiti.
Lista ordinata di cose da fare:
 - requisiti funzionali (user stories)
 - requisiti non funzionali
 - elementi che hanno a che fare con correzioni
 - elementi legati alle tecnologie
 tutti questi concorrono per il backlog, che in realtà per la maggior parte è composto da user stories.

 user stories = vengono raggruppate in epiche e poi mano a mano vengono raffinare
- *sprint backlog* - a partire dal precedente si arriva allo sprint. Insieme di task posti in funzione degli elementi con più alta priorità dichiarati nel b.l. Ovviamente da svolgere nella durata dello sprint. Ogni attività è associata a uno sforzo (ore necessarie per completare l'attività); lo sforzo dovrebbe

essere inferiore a un giorno lavorativo. Se più grande, poi dividerlo.



- *burn down chart* - è la rappresentazione grafica dell'effort in termini di ore.



Stimare la durata scrum:

- task: durano ore
- user stories: si valutano in termini di valore aggiunto nel momento in cui vengono realizzate = storie point (i punti storia di solito progredivano come una sequenza di Fibonacci); assegnate con la planning poker (letteralmente esistono delle carte per stimare) ovvero quante story point dare ad un task?

Ma date queste metriche, quanta roba tiro fuori dal backlog iniziale?

Ci sono 2 diversi approcci:

1. **Capacity driven** - se tiro fuori dal b.l. cerco di capire quante ore ci metteranno i task singoli (ne tiro fuori finchè arrivo). Ovviamente la stima non è precisa. Parliamo di tempo quando ancora sto valutando in termini di complessità.
2. **Velocity driven** - fa riferimento alla velocità di un team (numero di stories per ciclo): + ci si lavora + la velocity è costante, molto spesso la costanza non si verifica.

La cosa migliore è fare un misto tra le due.

La metodologia scrum si adatta bene a gruppi di 7-9 persone massimo, più gente c'è meno funziona. Una caratteristica fondamentale è la produttività individuale (quindi ci vuole un'eccellenza tecnica imprescindibile).

Se ho molto personale, e decido di frammentare i team il problema diventa coordinarli: LeSS = large scale scrum è un framework destinato a molti team che lavorano insieme su un unico prodotto, già creato appositamente: il product backlog è unico, ma poi gli sprint si dividono.

KANBAN

Dal giapponese, tabellone. E' un metodo nato non per lo sviluppo software ma per modelli di produzione tipo Toyota o Pirelli.

L'obiettivo è controllare la produttività just in time, di prodotti che evolvono nel tempo.

Principi:

- (non si parte da 0) c'è già una realtà con struttura. Qui si cerca di facilitare i cambiamenti, indirizzamenti per migliorare processi continui ed incrementali (non ci sono cicli time boxed).
- rispetta il corrente processo, regole e responsabilità
- tutto il lavoro è diviso in unità: attraverso delle pipeline composte di stadi.

La visualizzazione è un concetto importante: la kanban, ovvero la pipeline è rappresentata.

Kanban cerca di far corrispondere il work-in progress (WIP) alla capacità del team. A tale scopo, le fasi della pipeline sono limitate a un limite: il processo procede spostando gli elementi di lavoro attraverso la pipeline, dal modulo "da fare" a "fatto", il tempo di ciclo è la metrica di base utilizzata per valutare i progressi del team: in particolare i cycle time (dall'inizio alla fine) a differenza delle velocity di scrum.

Workflow -->	Inbox		Specification		Ready for Development		Development (e.g. using Scrum and XP)			Code Review		Test on Local System		Test on Pre-Production System	
	WIP Limit -->	5	2	2	3	2	2	2	2	2	2	2	2	2	2
		Feature	In progress	Done	Planned	In Progress	Done	In progress	Done	In progress	Done	In progress	Done	In progress	Done
Login	User Story 567 User Story 214		User Story 857				User Story 654				User Story 75				
Register				User Story 244		User Story 751									
Password Recovery	User Story 624					User Story 245		User Story 782							
...	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
Billing			User Story 657	User Story 38				User Story 858							

Interrompiamo un attimo i pattern e parliamo brevemente di **Software Testing**

Il software testing fa parte delle attività di:

- verifica: valutare gli artefatti se fanno quello che ci si aspetta
- validazione: se il sistema è corretto ma in che senso? -> miglioriamo la sua affidabilità rispetto alla correttezza

di attività.

I software non godono di continuità tra i diversi stati fisici che lo caratterizzano, sono sistemi particolari.

Se volessi avere un insieme di test per verificare un software dovrei quindi farne tantissimi! Allora si procede invece per test fatti spesso, per più tempo rimane un bug, nascosto, più sarà costoso rimediargli.

Parole chiave:

- Bug = manifestazione dell'errore (che si può verificare o no)
- Failure = quello che accade se c'è un bug
- Issue = bug report
- Test case = singole unità dei test. Creo un'istanza del sistema che deve essere creato, gli mando degli input e descrivo quali sono i suoi comportamenti attesi.
- Test set = l'insieme delle unità test case

I test avvengono a vari livelli:

- | | |
|---------------|----------------------------------|
| • Unit | • Class/method |
| • Integration | • Group of software modules |
| • System | • The whole system
(in vitro) |
| • Acceptance | • The system in use |

Indipendentemente da essi abbiamo comunque:

2 tipi di testing:

1. **STATICO** il codice è analizzato senza essere eseguito
Si procede con teorie e modelli matematici. Un sistema molto usato è Polyspace
2. **DINAMICO** il codice viene eseguito controllandone l'output

*Come faccio a capire qual è l'insieme di test case che posso usare per un intero sistema?

Approccio BLACK-BOX	Approccio WHITE-BOX
<p>Dato un sistema costruisco un insieme di test senza vedere il codice, conosco solo come dovrebbe comportarsi.</p> <p>Cosa valuto?</p> <ul style="list-style-type: none"> • Equivalence partitioning • Boundary value analysis • Decision table testing • All-pairs testing • State transition tables <ol style="list-style-type: none"> 1. quali sono i valori critici che potrebbero mandarlo in crisi (bound) 2. Se funziona per il 5 funzionerà anche per il 7. scelgo dei gruppi di valori più strutturati 3. se prendo dei valori con logica, devo prenderne tutte le combinazioni possibili 4. semplifico: non prendo tutte le combinazioni, solo quelle in cui almeno 2 valori sono diversi 5. Costruisco uno state machine e su di esso ricavo i test possibili (metto dei test per ogni transizione) 	<p>I test devono attraversare tutto il codice che è stato scritto (cosa diversa dal black box, che invece potrebbe non entrare mai in certe choice).</p> <p>Cosa valuto?</p> <ul style="list-style-type: none"> • Control flow / data flow • Branch testing (decision coverage) • Path testing (code coverage) <ol style="list-style-type: none"> 1. trasformo il codice in un grafo e con il test viaggia al suo interno 2. se ho dei flussi decisionali cambio le variabili e vedo cosa succede 3. modifico dei valori

	BLACK	WHITE
VANTAGGI	<ul style="list-style-type: none"> • I testers possono anche non essere coders • Siamo molto più vicini ai requisiti 	<ul style="list-style-type: none"> • La conoscenza del codice viene acquisita durante la creazione dei test case • Alta coprenza
SVANTAGGI	Non testerò mai in ampiezza di codice (via branch)	Molto complesso

Il punto è anche non volere troppi test! Non voglio doverci perdere tempo sopra! Per evitarlo c'è

Mutation testing (per Java c'è PIT)

Se un test set è fatto bene come lo capisco?

Creo dei mutanti del codice (ovvero copie del sistema con specifici errori). Runno il test sui mutanti e se il test passa vuol dire che c'è per forza un problema.

*Come faccio a capire qual è l'insieme di test case che posso usare per una sola classe o metodo?

Faccio Unit Testing (per Java c'è JMock)

Ovvero gli elementi vengono testati in isolamento.

Ma se nei miei metodi si richiamano altri metodi di altre classi? Come faccio a sapere dov'è l'errore? L'isolamento riesce lo stesso, ma si utilizzano dei fake/mock object, creo in sostanza delle controfigure già pronte delle mie dipendenze.

Questa procedura è automatizzata da alcuni software.

(Ma c'è anche un framework specifico per unit che è XUnit (per Java JUnit) che si basa su alcuni elementi:

- test runner - va creata una istanza dell'elemento che devo testare
- usa asserzioni (chiamo metodo mi aspetto un risultato, una per caso)

- il sistema va portato ad un certo stato per essere eseguito
- collezioni di test case
- ambiente specifico per il test

)