APS - Q&A

Usateli solo come ripasso e non per studiare

Parlami dei processi software.

Un software è un *applicativo generico o personalizzato* per un cliente eseguibile su un computer. Si può costruire in diversi modi, ma *bisogna seguire delle linee guida* per garantire l'applicazione delle best practice. Il **processo software** è l'approccio riguardante la costruzione, il rilascio e la manutenzione del codice, e indica come definire correttamente i *ruoli del team* e come organizzarsi nel tempo. Più in generale, è **l'ingegneria del software** la disciplina che produce questi software.

Quali sono le attività del processo software?

Innanzitutto si parte con l***'analisi dei requisiti***: li ricaviamo dal linguaggio naturale, e sono fondamentali per definire le necessità. Si passa poi alla ***progettazione***, ossia *trovare la soluzione che soddisfa questi requisiti*. Il passaggio successivo è lo ***sviluppo*** di queste soluzioni, che vanno poi ***convalidate*** nello step dopo: il prodotto 'finito' soddisfa i requisiti? Se sì, si può procedere col ***rilascio*** e ***l'evoluzione***, cioè aggiornamenti del prodotto in base a nuove esigenze o possibilità.

E in ottica Object Oriented, come si affrontano l'analisi e la progettazione?

In fase di **analisi** si definiscono classi di dominio che descrivono *concetti e oggetti del problema presi dal mondo reale*, mentre in **progettazione** si definiscono classi software per modellare i concetti in classi e oggetti software utilizzabili nel codice.

Cos'è un oggetto nell'analisi OO? Un'istanza di una classe concreta del mondo reale (per esempio, una persona specifica della classe Persona)

I processi SW
Qual è il processo software meno flessibile?

Quello in **cascata**. Si eseguono le *attività predefinite in modo sequenziale* (requisiti->design->test->uso), ma il problema è che *se i requisiti non sono stabili fin dall'inizio si deve tornare ripetutamente alla prima fase* e procedere di nuovo in sequenza. Per i progetti complessi il rischio di fail è alto in quanto *flessibilità praticamente assente.*

Come funziona lo sviluppo iterativo, incrementale, evolutivo?

Si sviluppa in multiple **iterazioni** di durata fissa (*time-boxed, dura in settimane*) e variabili in base al carico di lavoro. In un **iterazione** si svolgono tutte le attività di processo e si produce il software eseguibile, viene usato e viene fornito feedback.
Il materiale prodotto sarà la base per l'iterazione successiva (ecco perché **incrementale**), che verrà influenzata dalla precedente per ridefinire passo passo gli obiettivi e come deve **evolvere** il progetto. Questo sviluppo è flessibile alle modifiche, stabilizza i requisiti ed è più adatto ai progetti complessi. Bisogna prestare particolare attenzione ai *rischi del progetto*

per pianificare le prossime mosse. Si ha continuo feedback ed è fondamentale per l'iterazione successiva e per come devono cambiare i requisiti.

E Unified Process?

Non è altro che un *processo iterativo, incrementale ed evolutivo* per lo sviluppo *orientato agli oggetti.* Si possono integrare altri metodi iterativi (Agile UP) al suo interno, e i pilastri di questo processo sono i *casi d'uso e i rischi.*
UP ha quattro fasi: ***Ideazione*** (avvio, studio e stime sui costi e tempi), ***elaborazione*** (sviluppo del nucleo e revisione dei requisiti), ***costruzione*** (preparazione al rilascio) e ***transizione*** (completamento). Ogni fase è costituita da più iterazioni.
Un'interazione è una sorta di *mini progetto* (quindi analisi, costruzione, test…) in una finestra di tempo.

Metodo Agile?

Uno sviluppo iterativo che promuove l'agilità: risposte rapide e flessibili ai cambiamenti, consegne incrementali e semplici. Si modellano in UML *solo le parti complesse*, lo sviluppo è guidato dai test (TTD) e si applica spesso il refactoring per alleggerire il più possibile.
UP Agile è un 'fork' di UP in pieno stile Agile: poche (ma mirate) attività/elaborati, requisiti completati DURANTE le iterazioni e non prima di implementare, UML solo strettamente necessari.

Metodo SCRUM?

Un *metodo dallo spirito Agile* che consente il rilascio dei prodotti più 'importanti' per i clienti nel minor tempo possibile. Nello specifico lo ***SCRUM è un incontro giornaliero del team*** dove si stabiliscono le priorità della giornata guardando il **backlog** (arretrati da fare). Ogni iterazione è uno ***sprint*** di 2-4 settimane e la quantità di lavoro fattibile in uno sprint è detta velocity.
Figure chiave: - Scrum Master: fa applicare lo SCRUM e fa da intermediario tra team
- Team di sviluppo (non più di 7 per scrum)
- Product owner: cliente

Differenze tra UP e SCRUM?

In UP la struttura generale è molto più *definita e rigida* ed è concentrato sui rischi e l'architettura, mentre SCRUM cerca di *promuovere flessibilità e adattabilità* come cardini insieme alla collaborazione tra team. UP è più indicato per gestire le complessità, SCRUM è *adatto a tutto, anche i progetti piccoli.*

Unified Process - Ideazione
Come si svolge la prima fase, ideazione? (risposta generale non specifica)

Una sola iterazione (la 0), in media di qualche settimana: è il primo passo per definire la visione del progetto, si raccolgono le informazioni necessarie e si decide la fattibilità. Si stimano i costi, tempi e il 10% dei requisiti FUNZIONALI (casi d'uso, servizi e risposte che

deve fornire agli input) e non funzionali (proprietà che deve rispettare il progetto, non funzioni) più critici. Si definiscono gli attori principali per determinare la portata del progetto. Gli elaborati mantengono un linguaggio naturale comprensibile a tutti, quindi da evitare il gergo tecnico. Si pianifica la fase successiva qui.

Cosa sono i requisiti?
I requisiti sono la capacità o condizioni a cui il progetto deve essere conforme per la sua riuscita: si devono evitare le ambiguità, ogni requisito deve essere completo e coerente. Talvolta si introducono nuovi requisiti funzionali per soddisfare le specifiche introdotte dai non funzionali. I requisiti si possono acquisire in diversi modi: direttamente interrogando i clienti, organizzando meeting tra gli sviluppatori e i clienti oppure raccogliendo feedback dai clienti a fine iterazione.

Elaborati dei requisiti? Gli elaborati in merito ai funzionali sono i casi d'uso, mentre per i non funzionali le Specifiche Supplementari.

Differenza tra fase e disciplina? Una fase sono più iterazioni insieme, le discipline sono le attività che si fanno in quelle iterazioni

Quali sono le discipline dell'ideazione? Modellazione del business (visione di progetto, stime economiche, team..), Requisiti

Disciplina dei Requisiti

Come definisco un caso d'uso? Scegliere i confini del sistema -> Identifico gli attori primari e i loro obbiettivi -> Definiscono i casi d'uso che soddisfano questi obbiettivi (un caso d'uso per ogni obbiettivo utente)

Definisco un caso d'uso quando: Non è banale (e corto, quindi dalle 3 alle 10 pag), supera il test del capo (il tuo capo è contento quando gli riferisci di averci lavorato tutto il giorno) e genera valore al business misurabile (cioè, utile ad un evento del business).

Quali elaborati/documenti si producono in Ideazione?
Casi d'uso, diagramma casi d'uso

Com'è fatto un caso d'uso?

Rappresentazioni di dialoghi tra uno o più attori e un sistema che svolge un compito. In pratica, descrivono il comportamento del sistema in diverse situazioni (ma NON i dettagli implementativi). Gli attori possono essere coinvolti direttamente ma anche esterni, offrendo supporto o interesse dentro il caso d'uso. I casi d'uso sono una collezione di azioni e interazioni, cioè scenari! Quindi un insieme di percorsi di successo/fallimento. Un caso d'uso può essere breve (un solo paragrafo che riguarda solo il successo), informale (n paragrafi per n scenari) oppure dettagliato (dalle 3 alle 10 pagine); può essere di livello utente, una sotto-funzione di sistema o il sommario di un obiettivo più ampio.
Il sistema viene descritto a scatola nera: non si descrive nello specifico chi fa cosa, ma come lo fa (le sue responsabilità).

Da cosa è formato il Diagramma dei casi d'uso?

Si modellano in UML i casi d'uso a livello utente, ci sono associazioni tra casi e attori e le associazioni tra casi d'uso possono essere: include (uno include l'altro), extend (uno estende l'altro soddisfatta una condizione e salta ad un extension point) e la generalizzazione (questa vale anche per gli attori).

NB: Modello dei casi d'uso comprende Diagramma casi d'uso, SSD e Contratti. Questi ultimi due non si fanno in Ideazione

Unified Process - Elaborazione
Come si svolge la seconda fase, l'elaborazione? (risposta generale non specifica)

Durante l'elaborazione si esegue un'indagine seria e completa di due o più iterazioni, ognuna dalle due alle 6 settimane. Si produce l'eseguibile del nucleo dell'architettura, che viene già testato e certificato di qualità.

Cosa si fa alla prima iterazione (1)? Centrata su l'architettura, è guidata dal rischio e affronta tutti gli aspetti più difficili e rischiosi del progetto. Si realizzano e implementano parte dei casi d'uso dettagliati durante l'iterazione, nello specifico quelli scelti nella pianificazione. Per le iterazioni successive si pianificano i requisiti da realizzare in base al rischio, alla copertura e alla criticità per il cliente.

Quali sono le discipline dell'elaborazione?

Modellazione del business, Requisiti, Progettazione, Implementazione.

Quali documenti si producono?

- Modello di dominio (informazioni da gestire)
- Ampliato il modello dei casi d'uso: aggiornato il diagramma, aggiunti SSD (funzioni), aggiunti Contratti (cambiamenti di stato come conseguenza delle funzioni).
- Documento dell'architettura: Architettura logica + varie viste
- Modello di progetto: Diagramma delle classi sw, Diagrammi di Interazione, Diagramma dei package

Disciplina del Business

Cos'è e perché definiamo il modello di dominio?

Si tratta di una rappresentazione visuale delle classi concettuali, ossia i concetti reali del progetto che stiamo trattando. Il dominio è lo spazio in cui il sistema opera ed è il cardine per tutti gli altri elaborati! Viene consultato prima in fase di analisi per comprendere il sistema da realizzare, e in progettazione come spunto per modellare lo strato di dominio a livello sw (ad esempio negli SD).

Le classi sono concettuali, ma esistono anche di descrizione: contengono informazioni correlate a qualcos'altro nel dominio (datatype). Ogni classe ha i suoi attributi, ovvero proprietà dei concetti con valore e visibilità (public, private…).
Le classi possono essere in associazione tra loro, sono relazioni concettuali e non software: aggregazioni (un intero e tante parti indipendenti), composizioni (una parte ad UN intero e l'intero è responsabile delle parti), generalizzazioni.

Disciplina dei Requisiti - ampliamento

Cosa sono e perché definiamo gli SSD?

Un elaborato che mostra gli eventi di I/O tra attori e sistema: l'utente genera gli eventi e il sistema esegue le operazioni nell'ordine definito dal diagramma di sequenza di sistema. Il sistema è inteso a scatola nera, cioè una grande entità che opera nel dominio (non ci sono quindi suddivisioni). Si modella prima lo scenario di successo e poi tutte le alternative. Sono importanti per determinare le operazioni di sistema nel dominio, come reagisce e poi successivamente per modellare le procedure all'interno del sistema in progettazione (non più a scatola nera).

Cosa sono e perché definiamo i contratti?

I contratti descrivono nel dettaglio i cambiamenti agli oggetti del dominio come conseguenza delle operazioni di sistema (messe a disposizione dall'interfaccia pubblica). Non sono obbligatori ma complementano i casi d'uso, quindi si stilano solo i necessari. Si definiscono per operazione, casi d'uso coinvolti, pre e post condizioni (cioè il cambio di stato dell'oggetto, come nuove istanze o modifiche agli attributi).

Differenza tra la disciplina di requisiti e progettazione?

L'analisi dei requisiti serve a fare la cosa giusta, progettare serve a farla bene (e che soddisfi i requisiti). I passaggi tra queste due sono molto frequenti ad ogni iterazione.

Nell'elaborazione cosa si deve definire per costruire il documento di architettura?

L'architettura software, ossia l'insieme delle decisioni significative sull'organizzazione di un sistema sw. Si scelgono gli elementi strutturali, come sono composti e le interfacce.
Si può stratificare la vista dell'architettura guardando solo determinati aspetti dell'architettura: in questo caso, la più utile è la vista logica.

Cos'è l'architettura logica?

Vista che divide le classi sw in strati/package/sottosistemi del software, è logica in quanto si modella cosa deve fare il sistema e non come è fatto fisicamente. Si rappresenta in UML ed è importante per identificare le dipendenze tra i package e sottopackage.
Si divide in più criteri: livelli in base al nodo fisico di elaborazione (pc, server…), strati in base alle sezioni verticali e le loro responsabilità e partizioni orizzontali che separano sottosistemi o gruppi di funzionalità.

Cos'è l'architettura logica a strati e perché usarla?

Uno stile di architettura logica in cui ogni strato è un gruppo grosso di package/classi e sottosistemi che hanno responsabilità coese e condivise nel sistema. Gli strati più in basso sono i servizi a basso livello e generali; gli strati in alto sono i servizi applicativi specifici e questi ricorrono ai servizi dei sottostanti. Come lo fanno dipende dall'architettura: una strati stretta può richiamare solo quelli immediatamente sottostanti, quella rilassata di strati ancora più giù.
Esempi di strati: UI, logica applicativa, servizi tecnici.
Si stratifica perché: consente modifiche locali e indipendenti, si separano la logica applicativa e quella di interfaccia e si possono utilizzare le funzioni più in basso.

Uno strato importante?

Logica applicativa: riguarda gli oggetti del dominio, e si divide in due strati a sua volta: strato del dominio, riguardante gli oggetti del dominio (i componenti principali), e strato application dove ci sono gli oggetti che gestiscono il workflow da e verso gli oggetti del dominio (come interagiscono tra loro nelle operazioni i vari oggetti).

Disciplina della Progettazione

Come ci si può approcciare alla progettazione?

- Durante la programmazione: si progetta testando (TTD) e facendo refactoring
- Disegno -> Codice: si converte UML a codice mediante reverse engineering
- Disegno: si converte in qualche modo (e decisamente costoso) il disegno in codice

Modello di progetto: da cosa è formato?

- Statici: Diagramma delle classi/package
- Dinamici: SD, Diagrammi di comunicazione, Macchine a stati (non necessarie in UP) e Diagrammi di attività.

I modelli dinamici sono quelli in cui applichiamo i pattern e la progettazione guidata dalla responsabilità.

Parlami dei diagrammi di sequenza SD. A cosa servono, differenze SSD e perché usarli?

La differenza dagli SSD è che non si ragiona più a scatola nera: il livello software necessita di esplicitare tutti i componenti del sistema coinvolti. Assistiamo a creazione e distruzione di oggetti, e allo scambio di messaggi tra lifeline (oggetti).

Cosa sono i messaggi?

Una comunicazione o interazione tra oggetti. Il messaggio mostra come gli oggetti collaborano tra loro per svolgere una specifica funzione o processo.

I messaggi possono essere inviati e di risposta (penso sia abbastanza autoesplicativo), e il messaggio che dà il via alla sequenza è detto found message nei diagrammi iterativi.

Cosa sono i Diagrammi di comunicazione (CD)?

Sono uguali agli SD ma le lifeline sono create nel momento in cui c'è un messaggio che ne invoca la creazione (grafo a rete), e dato che si sviluppano in verticale si affiancano i numerini per distinguere l'ordine delle azioni.
CD e SSD sono iterativi per come illustrano le iterazioni.

Cos'è il Diagramma delle classi software e perché lo usiamo?

Illustra le classi, interfacce e relazioni tra classi: è composto dagli oggetti ossia istanze di classe, ognuno di essi ha attributi e comportamenti (le operazioni); seguono i classificatori, ossia caratteristiche comportamentali e strutturali degli elementi. Un classificatore è una classe, un'interfaccia, un attore…
Le parole chiavi invece sono indicatori delle proprietà di un modello, come per esempio l'essere astratto, un enum o interfaccia.
Un'interfaccia è un insieme di funzioni pubbliche esposte in modo da separare funzionalità e implementazione: ciascuna di esse ha un contratto (serie di condizioni, non il documento) e chi la usa deve rispettarlo.
Si rappresenta con UML e valgono tutte le regole della generalizzazione di oggetti (totale/parziale, congiunta/disgiunta).

Cosa sono le Macchine a Stati in ottica di progettazione?

Un documento non necessario in UP ma gradito, modellano il comportamento dinamico dei classificatori del dominio: in particolare, il ciclo di vita di un oggetto rappresentato tramite stati, transizioni ed eventi.
Lo stato di un oggetto è la combinazione di valori degli attributi, delle sue relazioni e delle attività in corso. Un oggetto può essere dipendente dallo stato, cioè fornisce risposte diverse in base al proprio stato, oppure semplicemente non esserlo (risponde sempre allo stesso modo).

Gli stati possono essere ancora più complessi: quali tipi ci sono?

- Compositi: stati che includono una o più regioni con al loro interno sotto macchine; quando una regione raggiunge lo stato finale termina la regione (in caso tutte le regioni abbiano stati finali terminano in modo sincrono), e quando viene raggiunto lo stato finale del composito termina l'intero composito. Le regioni possono comunicare tra loro tramite meccanismi di flag o synch.
- Pseudostati:
    - Giunzione: ricongiunge più archi in uno stesso stato
    - Selezione: un input e vari output, l'output ottenuto è quello che soddisfa la condizione corrispondente (una sorta di switch)
    - Ingresso/Uscita: si entra/esce nel composito

- Memoria: di tipo semplice, ovvero ricorda solo l'ultimo stato del composito, oppure multilivello in cui ricorda tutti gli ultimi stati dei sotto macchinari del composito.
- Finale: termina il composito.

Che tipo di eventi si possono verificare?

- chiamata: chiamata di un'operazione o sequenza
- segnale: comporta l'invio o attesa di ricezione di un segnale
- variazione: una condizione soddisfatta fa scattare un'azione
- temporale: verificano condizioni temporali, dopo() o quando()

Cosa mostrano i diagrammi di attività?

Le attività di un processo e il flusso di esso attraverso nodi e collegamenti; molto utili (ma non necessari in UP) per mostrare il workflow di un processo/oggetto. I nodi si possono trasmettere tra loro dei token, che possono essere oggetti e dati oppure dei token per il controllo del flusso.
I nodi possono essere di:

- Azione: invocano attività, comportamenti e operazioni; invia un segnale d'azione o emette un evento ricevuto il token opportuno in entrata
- Controllo: alterano il flusso (inizio, fine, decisione)
- Oggetto: pile o code di token dell'attività
    - I pin sono dei nodi oggetto di input/output

Le azioni possono essere raggruppate in base alla loro correlazione tramite partizioni.

RDD Responsibilty Driven Design
Come si progetta guidati dalle responsabilità? (incoraggiata da UP)

Innanzitutto, la responsabilità è l'astrazione di quello che un oggetto fa o rappresenta, e progettare guidati dalle responsabilità vuol dire vedere il progetto come una comunità di oggetti responsabili e collaborativi tra loro al fine di fornire funzionalità.
La loro implementazione è tale per adempiere alle loro responsabilità.
Bisogna trovare le responsabilità, chiedersi a quale oggetto appartengono e come fa quell'oggetto a soddisfarla.

GRASP
Le responsabilità si assegnano in base ai Pattern GRASP, principi base di progettazione e responsabilità (sono consigli e non metodi fissi, ma sono problemi ben noti e identificati da un nome). Di seguito, tutti i pattern GRASP spiegati.

Information Expert? Se una classe ha le informazioni necessarie per adempiere ad un compito, allora dovrebbe essere quella la classe che lo esegue (ad esempio i calcoli di dati).
Creator? Se una classe ha le informazioni necessarie per creare un'istanza di classe o banalmente farà molto uso di questa nuova istanza, allora deve essere lei quella che la istanzia (ne è responsabile).

Controller? Se ho necessità di gestire gli eventi di sistema e coordinare le risposte alle richieste degli utenti (quindi instradare ai gestori appropriati), mi è utile creare un oggetto Controller che se ne occupi.

Low coupling? Mantenendo l'accoppiamento basso tra le classi (cioè evitando la dipendenza stretta), si riesce ad aumentare flessibilità e manutenibilità. Per far ciò si rivedono le responsabilità di ciascuna classe.

High Cohesion? Per mantenere le classi chiare, avere responsabilità precise e sostenere il low coupling, si cerca di tenere le funzioni necessarie solo alla sua area funzionale, e per altre aree collabora con le altre classi (alta coesione).

Polimorfismo? Data una classe base con un comportamento base, possono esserci variazioni come alternative sul tipo (if,then,switch) che modificano il comportamento. Piuttosto che gestirle nella classe base e avere ripercussioni sul sistema, si usa l'ereditarietà e il polimorfismo per assegnare la responsabilità delle variazioni a delle classi derivate, così che in base al tipo venga chiamato il servizio adatto pur in realtà apparendo con lo stesso nome.

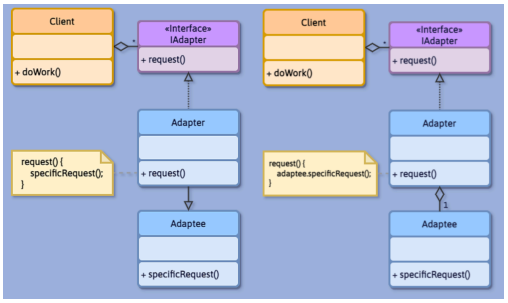
Pure Fabrication? Quando vogliamo garantire sia low coupling (evitare la dipendenza stretta tra classi) che high cohesion (tenere ben separate le responsabilità) ma non è appropriato applicare l'information expert (cioè, chi ha le info è chi ha la responsabilità) poiché violerebbe questi due principi, si introduce una classe artificiale di convenienza, che non fa parte del dominio e che ha queste responsabilità.

Indirection? Voglio sostenere low coupling e ridurre l'accoppiamento diretto tra due classi/componenti, quindi introduco un intermediario tra le due (un esempio è l'adapter).

Protected Variations? Voglio evitare che le variazioni o le instabilità influiscano sul funzionamento complessivo del sistema: si cerca di proteggere le parti del sistema soggette a variazioni tramite un'interfaccia stabile che le esponga tenendole "protette".

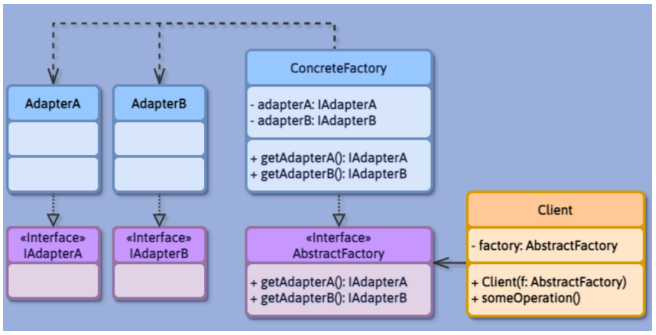
Design Pattern
Dei problemi comuni software ben conosciuti e di cui si conosce anche uno schema risolutivo da applicare. Ogni pattern ha un proprio scopo. Di seguito, tutti i pattern design spiegati.

Adapter? (strutturale: gestire le informazioni delle classi)



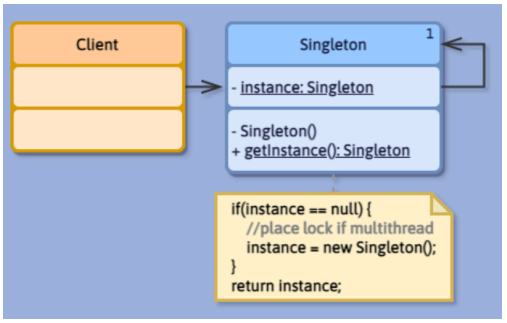
Il suo compito è rendere compatibili tra loro le interfacce che non lo sono (ad esempio server e client): per farlo si converte l'interfaccia di un componente (nella foto client) in un'altra che sarà compatibile grazie ad un oggetto o classe Adattatore (Adapter) con l'altra interfaccia.

Factory? (creazionale: riguarda la creazione di uno/più oggetti)



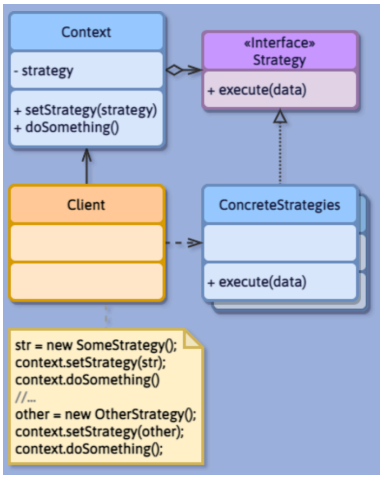
Quando esistono delle condizioni come una logica di creazione complessa, si vogliono separare le responsabilità di creazione per una migliore coesione ecc, è utile definire un oggetto fattoria che gestisca la creazione di questi oggetti. La Concrete Factory (singleton) è responsabile degli oggetti e la loro creazione, l'interfaccia Abstract Factory consente di accedere agli oggetti in questione dal client. In questo esempio la concrete factory è una, ma possono essercene di più che implementano la stessa Abstract Factory

Singleton? (creazionale)



Garantisca che esista UNA sola istanza di una classe e che esista un unico punto di accesso globale ad essa (il get). Si inserisce un lock in multithreading per evitare più istanze.
esempio banale: logger di un applicazione che agisce in più parti di essa
usato per: Factory (esiste una sola) e Facade

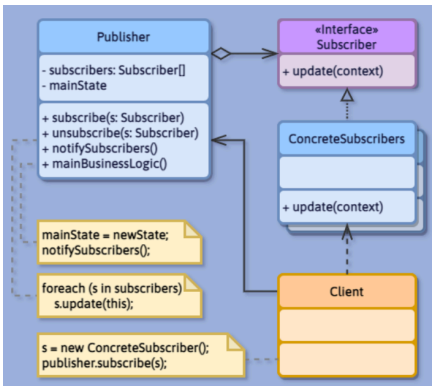
Strategy? (comportamentale: l'interazione tra classi e oggetti per ottenere un certo comportamento)



Quando voglio gestire degli algoritmi o politiche variabili in modo da poterli modificare singolarmente, è utile definire ciascuno in classi separate Concrete Strategy con un'interfaccia Strategy comune da cui accedervi. Il client confeziona un oggetto Strategy con la strategia che gli serve e lo manda al Context, settando la strategia necessaria a runtime tramite il setter esposto. Infine, il context esegue la strategy indicata e la restituisce.

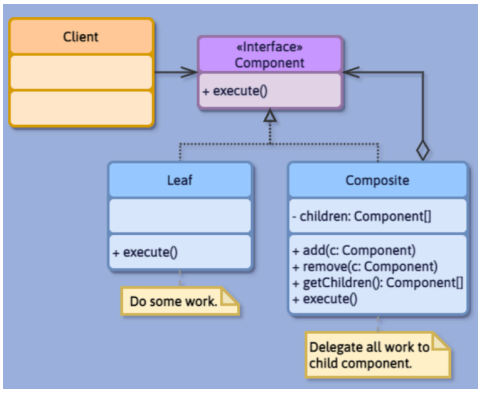
Si basa sul polimorfismo, fornisce Protected Variations rispetto agli algoritmi implementati e spesso si creano con una factory.

Observer? (comportamentale)



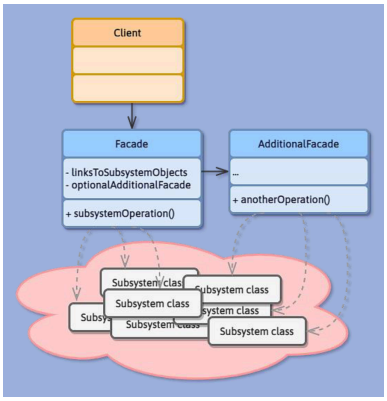
Si usa quando vogliamo notificare degli oggetti di un evento o situazione particolare: si definisce un publisher che notifica tramite un'interfaccia comune Subscriber tutti gli oggetti interessati, che poi reagiranno a loro modo alla notifica e indipendentemente. Il client è il responsabile dell'aggiornamento del publisher. Viene sostenuto il low coupling in quanto i publisher conoscono i subscriber tramite l'interfaccia ed essi possono registrarsi o cancellarsi dinamicamente (semplicemente implementando l'interfaccia), si basa sul Polimorfismo e fornisce Protected Variation nei confronti dei subscriber (modifiche indipendenti).

Composite? (strutturale)



Quando si vuole trattare un gruppo o una struttura composta di oggetti dello stesso tipo come se fossero un unico oggetto atomico (un albero con radice e foglie), si definiscono delle classi per i vari oggetti composti in modo che abbiano la stessa interfaccia. L'interfaccia Component descrive le operazioni comune agli elementi del composto, i vari Composite delegano i lavori alle foglie e raccolgono i risultati da restituire al client.
Usato a braccetto con Strategy, si basa sul polimorfismo e fornisce Protected Variations nei confronti dei componenti.

Facade? (strutturale)



Ci sono tante implementazioni e interfacce (quasi un sottosistema intero) da gestire, la scelta più appropriata è un oggetto Facade che copra tutto quanto e funga da punto di contatto per i componenti e li assista nella collaborazione. Ci si può servire di un additional Facade di supporto.
Il Facade è un singleton, garantisce protected variations verso i componenti ed è inoltre un oggetto indirection che sostiene il low coupling.
esempio banale: un controller Facade

Cosa vuol dire progettare la visibilità?

Significa determinare la capacità di un oggetto di vedere o avere riferimento ad altro oggetto: prendiamo il caso di A e B. La visibilità di B in A può essere gestita con un attributo (B attributo di A), per parametro (B è un parametro di un metodo), localmente (B è oggetto locale di A) e globalmente visibile.
Ovviamente parametro e locale sono visibilità temporanee, le altre due permanenti.

Come si passa dalla progettazione all'implementazione?

Si prendono gli elaborati della progettazione come input per generare il codice OO, in particolare partiamo dalle classi meno accoppiate alle più accoppiate. Si definiscono quindi classi, metodi, costruzioni e variabili. Se una classe UML implementa un'interfaccia su una variabile, si dichiara l'interfaccia su quella variabile anche nel codice

TEST nello sviluppo

Come funziona il Test Driven Development?

Il senso è che si scrivono prima i test immaginando il codice completo e poi si scrive il codice completo. Può essere utile per capire meglio lo scopo del programma (e non programmare bismillah), rendere chiaro il comportamento e verificare in modo veloce se il codice rispecchia il comportamento atteso: i cambiamenti possono essere fatti con più consapevolezza se qualcosa non va come ci si aspetta.

Quanti tipi di test conosci?

- Unitari: test per le singole classi e metodi, ragioniamo per piccole parti del sistema (è importante liberare i test vecchi quando si eseguono i nuovi per evitare errori strani)
- Integrazione: test per vedere se le parti comunicano tra loro
- Sistema: test per vedere se il sistema è collegato bene
- Accettazione: il test dal punto di vista dell'utente, verificando funzioni tutto

Come si svolge il ciclo di test in generale?

Scrivo test unitario che fallisce, scrivo il codice per validare questo test e se tutto funziona controllo il codice e applico eventuali refactoring. Questo è il ciclo base, ma esiste anche il doppio che dopo aver verificato il test unitario prova anche un test di Accettazione per verificare che ora funzioni tutto il complesso di unità.

Refactoring

Cos'è e perché usarlo?

Un metodo disciplinato e noto usato per riscrivere o ristrutturare del codice esistente SENZA cambiare il comportamento esterno, a scopo di migliorarlo (per esempio rimuovendo duplicati e compattando codici lunghi) e predisporlo a nuove modifiche.

Quando dobbiamo usarlo?

- Regola del 3: se una porzione si ripete almeno 3 volte nel programma
- Aggiunta una nuova funzione
- Quando correggiamo un bug
- In generale in fase di revisione del codice

Quali sono i passaggi generali?

Passano i test? Sì -> Ci sono dei problemi (code smell) nel codice? Sì -> Ho trovato il code smell, che refactor applico? -> Ho trovato e applicato il refactor, funziona? -> No --^
-> Sì -> Apposto

Quali refactor esistono?

- ***Extract***
  - ***Variable:*** tante espressioni complesse in un if, faccio variabili per ogni condizione
  - ***Method:*** prendo del codice esistente da un altro metodo e creo un nuovo metodo
  - **Class**: estraggo parte della logica o comportamento di una classe per metterla in una nuova
  - **Subclass**: si prendono i campi usati più raramente e si mettono in una sottoclasse
  - **Superclass**: si prendono i campi in comune tra più classi e si mettono in una superclasse

- **Inline class:** si incorpora una classe quasi vuota in una classe che la sfrutta (Person ha solo il nome e riferimento a Telephone Number, si incorpora tutto in Person)

- **Move/rename method:** si sposta un metodo da una classe ad un'altra/si rinomina un metodo

- **Preserve Whole Object:** anziché recuperare informazioni da un metodo con le get, si passa come parametro l'intero metodo

- **Replace temp with Query:** anziché definire una variabile temporanea per un'operazione, definisco un metodo che restituisce quell'operazione alla sua chiamata (double basePrice= … -> basePrice())

- **Replace method with Method Object:** si prende un metodo esistente, si estrae e si porta in un nuovo oggetto, e poi si chiama quel metodo dall'altro oggetto.

- **Replace Parameter with Method Call:** anzichè recuperare parametri da B e poi chiamare B con quei parametri, si passano solo i parametri che già possiede A e B recupera il resto

- **Replace Inherit with Delegation:** Sostituisco l'eredità con un'operazione di delega
- **Replace conditional with Polymorphism:** rimpiazzo un lungo if/switch dividendo le condizioni per sottoclassi

- **Decompose conditional:** evito gli operatori logici per più espressioni, così ho solo una via di if e un else

- **Introduce Parameter Object:** converto degli attributi passati come parametro in un oggetto (...(datestart, dateend)->...(dates))

- **Introduce Assertion:** condizione necessaria per il proseguimento

Code Smell

Cosa sono?

Cattive pratiche di programmazione, ciascuna è riconoscibile e risolvibile tramite refactor

Quali code smell esistono?

- **Bloaters:** codici troppo lunghi

    - Long method: troppo codice in un metodo
        - posso ridurre la lunghezza senza problemi o ci sono dei cicli : estraggo il codice in nuovi metodi (Extract Method)
        - ci sono problemi con variabili locali: uso un refactor per le variabili (Replace temp, Introduce param…)
        - ci sono condizionali: li scompongo con nuove funzioni (decompose conditional)

    - Large class: troppi metodi in una classe
        - si può separare in più classi: Creo nuove classi (extract class)
        - si può implementare parte del comportamento in altri modi o in generale è poco usata quella parte: definisco una sottoclasse con quella parte (extract subclass)
        - uno o più metodi si possono spostare altrove: extract method

    - Long param objects: un metodo ha troppi parametri
        - se alcuni parametri sono presi da return di un altro oggetto: chiamo quell'oggetto direttamente (Replace with method call)
        - se ci son tanti parametri relativi ad un oggetto: passo il metodo come parametro
        - se non sono correlati: introduco un oggetto che contiene i parametri (introduce param object)

- **OO Abusers:** implementata male la logica OO
    - Switch: c'è uno switch molto lungo (o tanti if-else)

        - posso sfruttare il polimorfismo: divido le condizioni in classi (Replace with Poli)
        - posso isolare lo switch: estraggo e sposto il metodo in un'altra classe (Extract + move)

    - Refused bequest: Alla sottoclasse non servono le proprietà ereditate completamente
        - se non ha senso l'eredità: la rimpiazzo con la delega (Replace delegation)
        - l'eredità va bene ma si possono togliere metodi dalla superclasse per metterli in un'altra: extract superclass

- **Change Preventers:** ci sono parti di codice molto accoppiate che impediscono cambi rapidi
    - Shotgun Surgery: i cambi sono piccoli ma da fare in tante classi diverse
        - posso riorganizzare il comportamento raggruppando gli elementi altrove: move method & fields
        - se rimangono classi vuote: accorpo ad altre classi (inline class)

- **Dispensables:** codice inutile da spostare/rimuovere altrove
    - Duplicated code: codice duplicato molte volte
        - ha senso renderlo un metodo chiamabile: extract method
    - Data class: fa solo da contenitore per i dati
        - aggiungo metodi per manipolare i dati nella classe (move method)
    - comments: troppi commenti
        - se il commento spiega un'espressione lunga: estraggo l'espressione in una variabile (extract variable)
        - …spiega una porzione intera di codice: estraggo metodo
        - …afferma regole necessarie a far funzionare il codice: introduco asserzione
- **Couplers:** troppi accoppiamenti o deleghe
    - Feature envy: un metodo accede di più ai dati di un'altro che ai propri
        - il metodo chiaramente non deve stare lì: spostalo
        - solo una parte del metodo accede ai dati: estrailo
        - il metodo usa tante funzioni di classi e si può smembrare: extract + move