Ciclo di vita

Con ciclo di vita si intende un insieme di azioni, con un inizio e una fine, per progettare un progetto software. Esistono diversi modelli:

- A cascata: il primo modello, prevede una serie di fasi. Conclusa una, si va avanti ma non prevede di poter tornare indietro. Si parte con la raccolta dei requisiti, un'analisi di tempi e costi, per poi terminare con codifica e test. Prevede la manutenzione del sistema in 3 diverse tipologie:
  - Adattativa → il sistema si riadatta a nuove esigenze
  - Correttiva → si risolvono problemi riscontrati in seguito
  - Perfettiva → si apportano migliorie
- A cascata con backtrack: simile alla precedente ma con possibilità di tornare indietro.
- Iterativo: un approccio che divide il problema in diversi sottoproblemi da realizzare separatamente. Ognuno di essi porta ad un prodotto parziale e funzionante che verrà a mano a mano integrato con le parti successive
- A spirale: composto da 4 parti:
  - Analisi → determinare obiettivi e vincoli
  - Progetto → valutazione di alternative e rischi
  - Sviluppo in codice e test
  - Collaudo per il rilascio

  Prevede diversi vantaggi, tra cui ovviamente la possibilità di limitare gli errori, ma richiede una sofisticatezza molto accurata

Metodologie agili

I modelli sopra indicati, detti anche "plan-based", sono tra i più classici per lo sviluppo di software. Nel tempo, però, sono nate metodologie che rendono la realizzazione più rapida e semplificata, solitamente più efficiente per sistemi di piccole dimensioni: le metodologie agili. Conservano con le precedenti la iteratività, ma non prevedono tecnologie codificate nello specifico, lasciando al programmatore libertà. Tra le metodologie più note eXtreme Programming, Scrum e Kanban

eXtreme Programming (XP)

Nasce con l'idea di mantenere la controllabilità del processo tenendo minimi i lavori di supporto e documentazione. Si basa sull'assenza di semilavorati, sulla non analisi e pianificazione dell'applicazione ma su una serie di attività da decidere a mano a mano. La sequenza di azioni prevede:

1. Codice
2. Testing
3. Consegna al committente (in caso di rifiuto si torna indietro)
4. Implementazione grafica

La programmazione avviene a coppie, scambiate frequentemente, di persone con esperienza diversa

Scrum

Dal mondo del rugby (pacchetto di mischia), è una metodologia iterativa, adattativa e incrementale. Si basa su tre concetti:

1. Sprint → iterazione di circa un mese alla fine della quale si consegna un prototipo funzionante
2. Backlog → lista di funzionalità da sviluppare
   - Product Backlog → lavoro da svolgere nel complesso
   - Sprint Backlog → lavoro da svolgere nel singolo Sprint

3. Meeting → riunione degli sviluppatori, prevede un responsabile (Product Owner), circa 10 persone tra programmatori, grafici etc. (Scrum Team) e il responsabile del team di sviluppo (Scrum Master)

Per tenere traccia del corretto andamento del progetto, lo Scrum Master può usare strumenti grafici come la Sprint Burndown Chart.

Ha il limite di definire solo procedure di Project Management, e va dunque opportunamente integrato con altri approcci (es. RUP, XP…)

Kanban

Dalla cultura giapponese (Miglioramento continuo), garantisce la possibilità di continuare a ritmi sostenibili e consentendo un miglioramento continuo del singolo e del team

Fondamenti di Kanban

Si deve attenere ad alcune regole filosofiche:

1. Visualizzare i flussi di lavoro (o workflow)
   I Workflow sono sequenze di passi svolte dall'inizio alla fine. La visualizzazione di tutti i flussi rende più chiaro cosa fare anche al Project Manager, che sa come e dove vengono investite le risorse. La visualizzazione consiste in 7 passi:
   - Identificazione del flusso di lavoro
   - Identificazione dell'ambito di lavoro, e su quale parte del flusso si ha controllo
   - Mappare le fasi del flusso in colonne su una lavagna
   - Definire i tipi di lavoro e quando i singoli sono completi, *done*.
   - Decidere un modello di scheda per ogni tipo di lavoro
   - Posizionamento degli elementi sulle carte, identificando cosi una priorità
   - Tracciamento del flusso di e, periodicamente, revisione del processo
2. Limitare il lavoro in corso (work-in-progress WIP)
   Inteso come limitare il numero di compiti da svolgere contemporaneamente, riducendo cosi ritardi e tempi del singolo ciclo, ridurre i costi, minimizzare i rischi e massimizzare il rendimento
3. Misurare e gestire il flusso
   Il flusso è il modo in cui una informazione si muove attraverso un processo aziendale. Si analizza come visualizzazione e limitazione del flusso impattano sul sistema e come migliorarli. Una metrica molto diffusa è il Cumulative Flow Diagram, che mostra la quantità di lavoro in ogni fase del sistema. Una seconda metrica è il lead & cicle time, confronta quanto tempi gli articoli passano nel processo e in quanto vengono attivamente lavorati. Il tempo necessario totale è detto throughput, indica l'analisi di questi due parametri
4. Esplicitare le politiche di processo.
   Questo rende più chiaro come una certa azienda si muove in un dato contesto
5. Riconoscere le opportunità di miglioramento
   Ciò avviene attraverso cinque fasi:
   - Identificare il vincolo
   - Sfruttarlo a proprio vantaggio
   - Minimizzare ulteriori impedimenti
   - Eliminare il vincolo
   - Cercare eventuali altri vincoli

   A questo concetto è allineato il ciclo Plan-Do-Check-Act, quindi cosa fare, come, valutazione delle

azioni e valutazione dei cambiamenti

Scrumban

Ibrido che unisce Scrum e Kanban, più vicino al Kanban meno restrittivo

DevOps

Acronimo di Development Operation, quindi programmazione + gestione post-codice, è una metodologia che pone le sue basi sulle metodologie agili ma che evolve con una filosofia tutta sua. È ampiamente diffuso in diversi settori informatici, tra molti Amazon, Netflix, lo stesso Google etc.

Prevede una serie di piccoli team autonomi e multidisciplinari che collaborano tra loro. DevOps prevede un sistema di miglioramento continuo proprio grazie a queste collaborazioni tra team diversi. La filosofia è tale che gli sviluppatori controllano fin da subito il funzionamento del codice ad ogni iterazione di compilazione. I team sono inoltre tenuto a ridurre i tempi medi di risoluzione, permettendo più rilasci di funzionalità velocemente, piuttosto che ottimizzare i tempi di gestione dei guasti, che non è altro che una riduzione dei problemi (in realtà, i problemi in un software raramente vengono eliminati del tutto).

SI elencano le pratiche base di DevOps:

1. Continuous integration → gli sviluppatori integrano frequentemente le modifiche al codice
2. Continuous delivery → modifiche al codice portano ad un rilascio immediato degli aggiornamenti. Solo dopo vengono eseguiti i test per trovare eventuali bug
3. Continuous testng → il testing constante aiuta nella ricerca di potenziali rischi
4. Continuous monitoring → un continuo controllo di metriche quali infrastruttura, utilizzo della CPU...
5. Infrastructure as Code (IaC) → le infrastrutture son configurate e gestite usando il codice
6. Microservizi → insieme di piccoli servizi separati che comunicano attraverso interfacce, API o altri metodi. Esse, tra le altre cose, permettono ai team DevOps di aggiungere componenti con funzionalità diverse senza modificare i già esistenti

Topologie

A seconda delle esigenze della specifica azienda, può essere adottata una delle nove topologie di DevOps tra le seguenti, dove è possibile e rende più semplice la realizzazione del sistema:

- Dev & Ops collaboration → la più teorica, prevede collaborazione intermittente tra i team
- Fully-shared Ops responsibilities → vi è una minima separazione tra i due team
- Op sas Infrasctructure-as-Service → Adatta ad organizzazioni con servizi in cloud
- DevOps as external service → team Dev ottiene supporto di un fornitore di servil che gestisce le responsabilità degli OPS
- DevOps team with an expiry date → collaborazione temporanea tra Dev e Ops
- DevOps advocacy team → utile per team solitamente distanti
- SRE team → team di Site Realibility Engineering, I Devs devono fornire la prova che il software è conforme agli standard richiesti
- Container-driven collaboration → incapsulamento dello sviluppo di un'applicazione in un "contenitore"
- Dev & DBA collaboration → più legata al mondo dei DataBase, i ruoli di Dev e delle DBA sono integrati aiutando a superare i limiti delle singole

Il software come prodotto

Come in un qualsiasi ambito ingegneristico, anche l'informatica produce prodotti. In particolare software, che hanno la difficoltà di essere quantificati temporalmente ed economicamente. Ad occuparsi di questi aspetti, o meglio di una stima di tempi e costi, è il Project Manager.

La difficoltà nella stima iniziale dei tempi è dovuta ad una carenza di informazioni iniziali, ed è importante sapere come valutarli e, in caso siano stime ottimistiche (quasi sempre), saper difendere le proprie stime col cliente.

Un altro aspetto importante che deve avere un Project Manager è mettere gli sviluppatori in condizione di poter sfruttare al meglio tutti i componenti disponibili, tant'è che si parla dell'ingegneria del software come ingegneria del riuso: spesso e volentieri ci si affida a componenti già esistenti riadattandoli al proprio specifico contesto.

È altresì importante sapere come gestire le risorse umane: assicurarsi che il singolo dipendente faccia progressi e il gruppo sia coeso sia in compiti che in ruoli gerarchici, specie per gruppi numerosi. Fondamentale diventa anche il ruolo del leader, non necessariamente coincidente con il manager: la leadership è spesso una caratteristica innata in una persona e deve focalizzarsi sugli obiettivi a breve termine. Si può dire che leader e manager siano ruoli complementari.

Un compito importante è anche la valutazione della qualità di un software. Solitamente si utilizza un sistema di gestione della qualità, che normalmente prescrive alcuni aspetti di progetto. A volte è richiesta per legge una particolare certificazione, ma è in ogni caso importante che la documentazione delle attività svolte e l'analisi della loro efficacia forniscono le basi per migliorare il software. Un altro metodo di controllo è il Total Quality Management, secondo cui la qualità dipende più dall'impegno delle persone coinvolte che non da un processo preapprovato.

Il software come prodotto presenta una serie di caratteristiche da tenere sempre in considerazione, tra cui:

- Design
- Malleabilità
- Manutenzione correttiva e adattiva
- Qualità interna ed esterna

A seconda del caso, può sussistere uno dei 4 possibili rapporti tra committente e sviluppatore:

1. Autoconsumo → committente = sviluppatore
2. Sviluppo interno → committente e sviluppatore fanno parte della stessa azienda
3. Sviluppo su commissione → il committente può decidere di pagare integralmente i costi di produzione ottenendo tutti i diritti o pagare una parte, lasciando i diritti allo sviluppatore
4. Software shrink – wrapped → lo sviluppatore rimane l'unico detentore dei diritti ma fornisce licenze d'uso

Una unità di musira fondamentale per lo sforzo espresso dal singolo soggetto nella produzione di un sistema è il mese uomo. Nel mondo reale, l'aumento delle risorse non implica quasi mai la riduzione dei tempi poiché i singoli vanno poi coordinati. Esiste una equazione che consente di stimare il tempo di rilascio

di un software in cui partecipano n sviluppatori che tiene conti di fattori quali la dimensione del lavoro in righe di codice, tempo di coordinazione etc.

È molto difficile studiare, invece, il costo di un software poiché si devono esaminare diversi aspetti. Tra di diversi modelli si distinguono costi diretti (personale tecnico, informatico etc.), indiretti (ristrutturazione edificio etc.) e altri fattori organizzativi. Un programma può essere misurato con due diverse metriche:

- Dimensionali → numero di righe (escluse le righe di commento e di codice ripetuto), tiene conti di diversi indici quali produttività, qualità, costo unitario
- Funzionali → misura indirettaq delle funzioni che svolge il sistema

Tecniche di stima del costo

COCOMO2

È una tecnica di tipo dimensionale che fornisce diversi sottomodelli a seconda dello scopo. Ad esempio, per costruire modelli da zero si sfrutta l'Early Design Model che calcola l'effort in mesi-uomo tenendo conto di tutto lo scenario circostante

Diagramma di Grant

Diagramma che mette in funzione del tempo inizio e fine dell'attività. Consente di vedere quali attività possono essere svolte in parallelo, quali richiedono recupero di versioni precedenti etc.

Function point Analysis

Tecnica che misura il valore del software tramite dei criteri come:

- Confronto richiesta del cliente vs software prodotto. Non vengono pagate cose in più rispetto a quelle richieste
- Scopo del software (non si analizza come è stato sviluppato ma solo cosa fa)

Un'applicazione viene misurata basandosi su die aree di valutazione:

- Unadjusted Function Point, che riflette le funzionalità fornite all'utente
- Value Adjustment Factor, che riflette la complessità delle funzionalità

Il software come processo

Verifica-Validazione-Test VV&T

Tre attività fondamentali nel processo di sviluppo di un software. Seppur nel linguaggio comune sembrano sinonimi, bisogna distinguerli in tre diverse operazioni:

- Verifica → rispetto dei requisiti
- Validazione → rispetto delle richieste dell'utente, solitamente riguardo la qualità
- Test → strumento per effettuarle

Solitamente, i programmi di successo durano molto più del previsto e cambiarli, per una azienda, può risultare molto dispendioso.

Verifica

Intesa come confronto tra prodotto e specifiche, prevede alcuni “check iterativi” quali la compatibilità del codice e la sua esecuzione senza errori. Possono essere svolti in diverse maniere, dalle più informali (e diffuse, valutare due oggetti separatamente ed individuarne le corrispondenze) alle più formali (meno diffuse, con approcci più matematici). La tecnica di verifica più diffusa è la compilazione, da cui si individuano più facilmente errori di sintassi e logici

Validazione

Viene eseguita correttamente coinvolgendo i clienti.

Un sistema di alta qualità deve poter essere utilizzato al meglio. Seppur sembra banale, spesso è un aspetto secondario poiché chi progetta non si mette nei panni dell’”ignorante di turno”. Risulta utile il parere di un esperto di usabilità e, come già detto, il coinvolgimento dell’utente

Test

Si pone come obiettivi principali la ricerca di bug (si dice che un test di successo è quello che identifica i bug), mascherarli al cliente e fornire informazioni che aiutano l’evoluzione del sistema. Esistono diverse tipologie di testing, tra cui gli Unit Test e i test di regressione (si verifica che modifiche applicate al sistema non alternino il funzionamento di caratteristiche precedentemente testate e funzionanti).

Si distinguono test che guardano la specificità dell’entità e quelli che guardano la loro struttura interna

Un cliente percepisce un sistema come di qualità quando svolge il comportamento richiesto (senza interessarsi al comportamento dei singoli componenti), eventuali modifiche non alterano le funzioni già supportate ed i casi tipici sono facilmente individuabili e gestibili.

Può essere necessario automatizzare il processo di testing, anche se l’automazione non individuerebbe una pianificazione errata di testing.

Un problema tipico degli Unit Test è il paradigma OOP, in particolare riferimento ad incapsulamento ed ereditarietà.

Seppur l’attività è importante, viene spesso trascurata per ragioni economiche e temporali.

Manutenzione ed evoluzione del software

Rappresentano la maggior parte dei costi di un software (natura tecnica e non), oltre a richiedere maggior tempo. Con manutenzione si intende la modifica post distribuzione, ad esempio modificando i componenti esistenti o aggiungendone altri. I cambiamenti possono essere dovuti da diversi fattori, come la presenza di nuovi requisiti, individuazione di guasti. Esistono quattro tipologie di manutenzione:

- Perfettiva → aggiunta di nuove operazioni
- Adattativa → adattamento a nuove politiche
- Correttiva → correzione di errori nel programma
- Preventiva → modifiche per migliorare la futura manutenibilità

Esistono, inoltre, sistemi vecchi ma ancora in uso. Tali sistemi prendono il nome di Sistemi legacy, e sono molto difficili da cambiare per le aziende.

Configurazione del software

Si tratta di un processo per stabilire i legami tra gli attributi di un prodotto. La gestione della configurazione tiene traccia dei singoli elementi di configurazione di un sistema IT, i cui asset sono rappresentati da un pezzo di software, un server etc.

La gestione della configurazione traccia e monitora le modifiche ai metadati di configurazione di un sistema software.

Qualità del software

Può essere intesa sia come proprietà intrinseche del software (qualità interna) sua come ciò che percepisce il committente (qualità esterna). Le prime proprietà ricercano affidabilità, efficienza, facilità di modifica e minime difficoltà nel trasferire il software in diversi ambienti. Le seconde ricercano la soddisfazione dei fabbisogni del committente. Si distinguono tre categorie:

- Qualità interna → caratteristiche interne
- Qualità esterna → proprietà del software che lo caratterizzano durante la sua esecuzione
- Qualità in uso → proprietà del prodotto in uno specifico ambiente

Qualità interna ed esterna

Dipende da sei caratteristiche:

- Funzionalità → svolgere i compiti assegnati
- Usabilità → l'utente riesce ad utilizzarlo senza difficoltà
- Affidabilità → possibilità di mantenere un livello di prestazioni adeguate sotto determinate condizioni
- Efficienza → rapporto tra prestazioni del prodotto e quantità di risorse da impiegare
- Manutenibilità → capacità di modificare il software senza alterare il corretto funzionamento
- Portabilità → possibilità di trasferire da un ambiente all'altro

È importante, in genere, mantenere una certa aderenza agli standard e la sicurezza da accessi non autorizzati

Qualità in uso

Si valuta per efficacia di raggiungere i propri obiettivi con piena accuratezza, produttività e limiti di risorse e garanzie di accettabilità di livelli di rischio di danni a persone e software

Riuso del software

Si tratta di una pratica ormai molto diffusa oggigiorno nella produzione di software. Si può parlare di livelli che vanno dal riuso di sistemi per intero, ad alcune applicazioni, componenti fino a semplici oggetti e funzioni Ha diversi vantaggi e svantaggi:

- ☑ Sviluppo più rapido
- ☑ Uso efficace degli specialisti
- ☑ Maggiore affidabilità
- ☑ Costi di sviluppo ridotti
- ☑ Rischio di processo ridotto
- ☒ Mantenere ed usare una libreria di componenti
- ☒ Adattare i componenti riutilizzati
- ☒ Aumento dei costi di manutenzione
- ☒ Mancanza di supporto degli strumenti

Quando si lavora al riuso di software, si considerano fattori quali il programma di sviluppo software, la sua durata, la piattaforma di esecuzione…

Il riuso di framework applicativi, entità molto grandi composte da una collezione di classi astratte, è tra le pratiche di riuso più diffuse. Si basa sul pattern MVC, che verrà spiegato prossimamente

Un altro aspetto riguarda il riuso di linee di prodotto, applicazioni con funzionalità generiche e composte da un insieme di componenti condivisi poi adattabili al singolo committente. Adattare una linea di software può comportare la configurazione di componenti, l'aggiunta di nuovi componenti e la loro modifica per soddisfare nuovi requisiti.

Infine, si dia uno sguardo agli application system: sistemi software adattabili a diversi clienti senza dover cambiare il codice del sistema. Ciò rende più rapida la realizzazione di sistemi affidabili e la valutazione delle funzionalità fornite. Tuttavia, i requisiti devono essere riadattati e, a volte, il prodotto può essere basato su presupposti difficili da cambiare. Uno dei sistemi più utilizzati è l'ERP, sistema che supporta processi di business comuni ed è infatti utilizzato dalle più grandi aziende al mondo.

I Pattern

Si tratta di un particolare riuso del software. In particolare, prevede un riuso di progetto già esistenti per risolvere problemi di progettazione. Per la loro specificità in tema riuso, sono molto flessibili e facilmente adattabili al contesto richiesto. Esistono tre diverse tipologie di pattern, dal più generico al più specifico:

- Pattern architetturali → descrivono lo schema strutturale di un software
- Design pattern → legati a problematiche di progettazione orientata ad oggetti. Ricerca quale, tra le soluzioni incontrate, meglio rispetta il proprio progetto.
- Pattern di implementazione → pattern specifici per una particolare tecnologia

Sono spesso confusi con gli algoritmi, ma va fatta attenzione. Un algoritmo è una sequenza predefinita di passi, mentre i pattern indicano una serie di azioni comunque da contestualizzare caso per caso.

Seppur non fondamentali, risultano importanti perché rispettano codici standard e, soprattutto, consentono di velocizzare i tempi di produzione.

Pattern architetturali

Model-View-Controller MVC

Come suggerisce il nome, il pattern separa il codice in tre classi distinte:

- Model → gestione stato e logica
- View → interfaccia utente in risposta ad aggiornamenti generici dal model
- Controller → traduce l'input dell'utente in aggiornamenti poi passati al model

Questa suddivisione consente rappresentazioni multiple della stessa informazione, oltre a rendere più facile la risposta agli input e, di conseguenza, modificare le interfacce utente. Soprattutto, consente allo sviluppatore di lavorare ad un singolo aspetto per volta. Le tre classi comunicano tra loro nel seguente modo:

1. La View riceve l'input dall'utente e lo passa al Controller
2. Il Controller modifica il Model in risposta all'input
3. Il Model cambia sulla base di aggiornamenti e modifica la View
4. La View aggiorna l'interfaccia utente

Layered Architecture

Le funzionalità del sistema sono organizzate in strati separati che si basano su funzioni e servizi offerti dallo stato precedente. Inoltre, modificando il comportamento di uno strato solo quello a lui adiacente ne viene influenzato. Questo approccio supporta lo sviluppo incrementale. Un esempio può essere pensato come lo strato basso a rappresentare il supporto del sistema fino allo strato più alto ad indicare l'interfaccia utente

DAO

Consente di disaccoppiare l'accesso ai dati dal database in utilizzo. Questo perché, altrimenti, se si cambia il database di riferimento tutte le informazioni vengono perse e il programma stesso va riprogettato in buona parte.

Repository

È intesa come database condiviso, la maggior parte dei sistemi si appoggia su una repository. Questo agevola la condivisione di grandi quantità di dati. Questo, però, richiede una manutenzione continua di più copie di dati

Client-Server

Un sistema che segue lo schema client-server è organizzato sottoforma di una serie di servizi associati (server) ed utenti che ne usufruiscono (client). A far da tramite tra i due è una rete browser che permette al client di accedere ai servizi. Il modello logico dei servizi va implementato in un singolo computer, e le loro modifiche non intaccano altre parti del sistema

Pipe-and-filter

Si utilizza quando realizziamo una serie di programmi in sequenza in cui l'output di uno è l'input di quello successivo. Sono molto diffuse per le attività che si svolgono periodicamente (es. backup, detti programmi batch.

Design pattern

Tra i più noti i pattern GoF, distinti in:

- Creazionali → creazione di istanze
- Strutturali → composizione di classi ed oggetti
- Comportamentali → composizione dei metodi

Pattern creazionali

Factory Method

Si pone di utilizzare, in nuovi contesti, attività inizialmente realizzate per altri scopi. Evitando di riscrivere tutti i metodi da zero, si pensa di realizzare una superclasse/interfaccia con dei metodi generici poi sovrascritti grazie al meccanismo di overriding.

Il meccanismo di ereditarietà è comunque molto rigido, ed è preferito dunque l'uso di un'interfaccia.

In generale, è utile quando si vuole realizzare una linea di prodotti con molte caratteristiche in comune

Abstract Factory

Considera scenari leggermente più complessi del Fsctory Method: ci si aspetta di lavorare con una serie di oggetti correlati da alcune caratteristiche comuni e con stili diversi. Ad esempio un oggetto scarpa con stile "da calcio" o "stivale".

Si crea una famiglia di software correlati con stili diversi: per il singolo oggetto applichiamo l'interfaccia da cui l'oggetto implementato presenta le stesse operazioni con gli altri. Le differenze vere e proprie sono solo di tipo "stilistico". Permette, indipendentemente dallo stile da seguire, di procedere senza problemi.

Builder

Si pone l'obiettivo di costruire oggetti diversi scomponendolo in più sottoparti da completare iterativamente passo dopo passo. Ad esempio, in un'idea di personalizzazione dell'oggetto risulta complesso ragionare con ereditarietà o interfacce poiché il risultato sarebbe molto scomodo. Una soluzione è creare un unico oggetto ricco di attributi per ogni tipo di situazione. La cosa è comunque inefficiente poiché ci sarebbero diversi attributi null. La soluzione pensata dal pattern Builder definisce una classe per costruire i vari attributi a mano a mano con dei metodi opportuni. Così facendo si evita la realizzazione di attributi "inutili". Il problema è che il client non sa a priori quali metodi gli servono. La versione più sofisticata del pattern è l'utilizzo di una classe "director" che funge da guida al client.

Prototype

Consente di copiare degli oggetti esistenti senza conoscere in dettaglio le classi. La soluzione è l'utilizzo di un'interfaccia: la classe da clonare deve rispettare una interfaccia con un metodo clone che poi clonerà la classe interessata (anche lei deve presentare il metodo clone).

Singletone

Fa si che una classe abbia una sola istanza e si può accedere ad essa da un solo punto di accesso. Ci si muove rendendo il costruttore della classe privato. Successivamente, si utilizza un metodo statico che lo usa. Così facendo è accessibile dall'esterno ma non modellabile dall'esterno e mantenendo così un buon livello di sicurezza. È molto diffuso nella gestione condivisa di oggetti.

Pattern strutturali

Adapter

Come suggerisce il nome, utilizza una interfaccia che consente la collaborazione tra oggetti che seguono interfacce diverse. Prende i dati da un primo oggetto, li rielabora per poi restituirli all'altro con una scrittura a lui compatibile.

Bridge

Consente di mettere in correlazione due dimensioni indipendenti (come se un ponte – bridge li collegasse): piuttosto che creare 2^n sottoclassi, si hanno due classi (con le rispettive sottoclassi) associate tra loro grazie alla composizione degli oggetti. Una delle due classi si chiama Astrazione e l'altra Implementazione. Risulta utile quando si realizzano interfacce grafiche (es: una per Windows, una per Linux etc.) o nella gestione di API.

Composite

Costruisce una gerarchia ad albero: un oggetto può suddividersi in due o più oggetti che a loro volta possono ulteriormente dividersi fino a raggiungere il caso minimo detto oggetto terminale. Il pattern ci consente di applicare ricorsivamente queste operazioni e, a prescindere dal livello in cui ci si trova, restituisce sempre lo stesso risultato. Questo strumento è utile per gestire le gerarchie

Decorator

Consente di implementare un oggetto partendo da un livello base e, a mano a mano, viene sempre più arricchito con nuove funzionalità arricchendolo. Onde evitare di lavorare con n comandi, si associa all'oggetto altri oggetti che contengono tutte quante le funzionalità richieste

Facade

È fondamentale per sistemi legacy: fornisce un'interfaccia semplificata per un insieme complesso di dati senza dover fornire tutti i dettagli del caso. Sono molto diffusi, ad esempio, nel mondo dell'elettronica.

Flyweight

Permette di inserire più oggetti nella quantità di RAM disponibile condividendo parti comuni dello stato tra più oggetti. Così facendo, pur avendo migliaia di oggetti diversi, memorizzo una singola volta tutti i dati simili in modo da avere più memoria libera una volta completato il processo.

Proxy

Si tratta di un pattern che definisce, per un oggetto, un segnaposto (sostituto) che fa da tramite tra il client e l'oggetto (solitamente molto pesante), che prende proprio il nome di Proxy. È molto vantaggioso perché, in caso si ricercano dati già cercati in precedenza (anche da altri utenti), il proxy li individua subito nel suo magazzino e rende, così, il processo molto più veloce

Pattern comportamentali

Chain of Responsibility

Consente di legare una serie di attività da far delegare da un oggetto ad un altro. È molto diffusa in ambito di telecomunicazioni: un pacchetto di dati viaggia attraverso Internet fino ad arrivare alla destinazione giusta. Inoltre, in caso stia viaggiando un messaggio obsoleto può essere interrotta la catena

Command

Trasforma un comando in un oggetto contenete tutte le informazioni necessarie. Solitamente, si può chiamare un relativo metodo per effettuare l'operazione. I parametri possono però essere diversi e quindi il comando può risultare complesso. L'utilizzo di un oggetto al posto di un metodo minimizza il problema

Iterator

È un pattern che gestisce le operazioni iterative più complesso. Posso scegliere tra diverse tipologie di iterazioni, al pattern non importa. È chiaro, però che il modo in cui si itera implica fortemente il modo di lavorare.

Mediator

Ha lo scopo di ridurre le dipendenze caotiche tra oggetti. I vari oggetti sono fortemente correlati tra loro, infatti una modifica in uno può portare ad alterare tutto il funzionamento. Entra in gioco una classe Mediator che fa appunto da mediatore. Così facendo, modifico solo l'elemento e il modo con cui esso comunica con il mediatore. È fondamentale, oggigiorno, in ambito di Big Data Analytics.

Memento

Dal latino “ricordo”, comporta di ripristinare lo stato precedente di un oggetto in caso si voglia tornare indietro per annullare delle attività. Per fare ciò, occorre memorizzare tutti i dati relativi all’attività. Lo stato, però, riguarda centinaia di oggetti a loro volta composti da diversi attributi privati. Questo può implicare o l’utilizzo di diversi metodi Get oppure bisogna rendere gli attributi pubblici rendendo più vulnerabile il sistema. Il Pattern memento prevede la creazione di una “versione virtuale” in cui si memorizza il valore dello stato e, all’occorrenza, si va a pescare da questo.

Observer

È un pattern che presenta un oggetto principale che deve inviare un messaggio a tutti gli oggetti collegati, evitando di entrare in contatto con eventuali altri oggetti presenti nel sistema ma non interessati all0attività dell’oggetto di interesse.

State

È un tipo di pattern che permette ad un oggetto di modificare il suo comportamento in funzione dello stato. Questo fa si che lo stesso comando, a seconda dello stato, si comporta in un certo modo. Evita l’inserimento di una serie infinita di switch.

Strategy

Permette di definire una serie di algoritmi, mettere ognuno in una classe separata e rendere i loro oggetti intercambiabili. In altre parole, per uno stesso problema ho diverse strategie e, a seconda del contesto e delle circostanze, si sceglie la soluzione più opportuna.

Template Method

Definisce la struttura di un algoritmo nella superclasse ma viene implementata solamente nelle sottoclassi in modo che, a seconda del grado di difficoltà, si può scegliere di pescare dalla sottoclasse più o meno complesso

Visitor

Prevede di utilizzare comportamenti completamente diversi a seconda dello scopo per cui viene chiamato in causa. Questo può risultare complicato poiché, di volta in volta, cambia del tutto lo scopo.

L’utilizzo del pattern permette di separare gli algoritmi dagli oggetti su cui si operano. Così facendo posso svolgere più compiti contemporaneamente e, a seconda del client, viene eseguita una certa attività.

Ingegneria del software avanzata

CBD

La Component-based Development Paradigm, CBD, è la metodologia affrontata finora per cui un software può essere progettato prendendo pezzi di sistemi già esistenti e contestualizzandoli in modo che comunichino tra loro senza problemi grazie a delle opportune interfacce.

Distributed computing

Il distributed computing, o calcolo distribuito, prevede che i vari sistemi che comunicano tra loro e ragionano in maniera indipendente l’uno con l’altro. Inoltre, i sistemi devono essere attendi ad aspetti di eterogeneità: sono comunque sistemi realizzati da persone che, ad esempio, possono aver pensato per una stessa sigla due significati completamente diversi.

Un grosso vantaggio è la tolleranza ai guasti grazie ad un forte utilizzo della ridondanza.

Spesso viene confuso con il “calcolo parallelo”. Questo sfrutta, però, la potenza di processori che lavorano sugli stessi dati (in parallelo) per risolvere problemi difficili. Per i calcoli distribuiti, ogni processore ha una sua RAM. In parallelo, come suggerisce il nome, ci si appoggia alla stessa.

Service-Oriented Architecture

È uno stile che supporta l’orientamento ai servizi. Con il recente avvento del cloud, si è pensato che i vari componenti possono essere trasformati in servizi da cui si accede tramite un server. Troviamo forti connessioni proprio con il cloud, da cui poi ne sfrutta diversi servizi. Il loro utilizzo più comune è mediante i web-service.

Real-time Computing

È importante che un sistema non solo sia funzionante ma anche realizzata nel tempo previsto. Sono fondamentali soprattutto per sistemi Embedded, quasi tutti funzionanti in tempo reale (ad esempio sistemi realizzati a scopo medico).

Cloud Computing

Con cloud si intende il trasferimento di grandi quantità di dati su server opportuni. Risultano molto vantaggioso poiché riducono i costi e velocizzando i tempi di lavoro. Ci sono tre dievrse tipologie di cloud:

- Pubblico, i più diffusi. Vengono forniti al pubblico (es. Amazon, Google)
- Privato, gestiti per la singola azienda
- Ibrido, generalmente pubblici tranne una sezione relativa alla sicurezza

Esistono diversi contratti stipulabili con i cloud. Tra i più comuni:

- ❖ Infrasctructure as Service → si compra solo l’infrastruttura e poi la si gestisce autonomamente.
- ❖ Platform as Service → il fornitore offre sia l’infrastruttura sia alcuni servizi utili per un certo scopo
- ❖ Software as Service → vengono forniti anche alcuni programmi (es. Office 365)

Architettura a microservizi

Nasce seguendo la filosofia dei Service-Oriented Architecture, supera i problemi di server condivisi. Si prevede, onde avere un solo server che in caso di danno influenza tutto il sistema, di avere tanti server quanti servizi. Si parla allora proprio di microservizi poi compattati per formare il software finale.

Molto diffuso, allo scopo, il Docker, che agevola fortemente il riuso, e Kubernetes.

I microservizi possono essere pensati come pattern, detti pattern di composizione.

Dependability

È chiaro che, realizzando un software, sia quanto più possibile affidabile e capace di trasmettere fiducia al compratore, oltre chiaramente a garantire una ottima sicurezza.