Il codice, i linguaggi, le strutture dati e gli algoritmi sono solo una minima parte di quando si va a sviluppare del software, ma dato che il codice verrà prodotto in team e non sarà composto soltanto da poche linee di codice, la vera implementazione sarà preceduta da una serie di passaggi, quali analisi dei requisiti e progettazione. La progettazione può essere fatta in vari modi ed è essenziale per la produzione di un codice in team.

Due parti importanti nella produzione di un codice sono il cliente e l'analisi dei requisiti. L'analisi dei requisiti non è un processo che dura poco e va avanti anche dopo l'inizio dello sviluppo del codice. Infatti è molto importante ascoltare le esigenze del cliente e avere un dialogo con esso per poter fornire un software con la miglior qualità possibile. Il cliente può richiedere modifiche durante lo sviluppo del software o dopo la prima release e in base alle esigenze si valutano costi e tempi di produzione. Le modifiche all'interno dello sviluppo possono arrivare anche da cause al di fuori delle richieste del cliente. Proprio per questo è importante fornire un supporto per il proprio software, in maniera tale da non farlo diventare obsoleto alla prima modifica.

Uno dei requisiti per dire che il software è di qualità è la sua modificabilità, infatti se un codice è scritto male vi verrà difficile capire ciò che fa,fare operazioni di testing e modificarlo per successive variazioni. In questi casi sarebbe più conveniente scrivere un nuovo software da 0, dato che per ogni modifica avremo bisogno di molto tempo e di un costo non indifferente. E' importante quindi scrivere un software quanto più leggibile e modificabile possibile, in maniera tale da implementare le modifiche in maniera semplice e veloce

Un altro aspetto importante è la correttezza del software. La correttezza del nostro software passa soprattutto dalle richieste del cliente, infatti per definire un software corretto dobbiamo soddisfare tutte le sue richieste. Oltre a dover soddisfare tutte le sue richieste dobbiamo dar modo al cliente di esprimere al meglio il suo desiderio, perchè altrimenti arriviamo a soddisfare le sue richieste ma il software non gli sarà utile dato che non è stato in grado di esprimere le sue richieste al meglio. Per agevolare il cliente in questa fase è bene rilasciare delle demo del nostro software, in maniera tale che il cliente possa farsi un'idea di ciò che stiamo andando a sviluppare.

AltrI requisiti per valutare un software

- Efficienza: L'utilizzo delle risorse in modo parsimonioso è importante per avere un software reattivo e che non vada a sovraccaricare inutilmente le macchine su cui gira. Per risorse si intendono memoria e CPU

- Manutenibilità: Se ci sono modifiche da effettuare all'interno del nostro software devono essere semplici da implementare e non deve richiedere troppo tempo. Avere un alto grado di manutenibilità ci consente di mantenere il nostro software aggiornato nel tempo, rendendolo in grado di adattarsi alle modifiche che ci saranno nel corso del tempo e che non possono essere previste in fase di progettazione

- Dependability (Sicurezza e affidabilità)
  La sicurezza può essere distinta in 2 categorie
  - Security: Un sistema rispetta la caratteristica di security se consente l'accesso ai dati solo ai rispettivi proprietari e non consente ad utenti non proprietari di accedere ad essi.
  - Safety: Un sistema rispetta la caratteristica di safety se non mette a rischio l'hardware su cui gira o la sicurezza di persone che dipendono da esso

  Per garantire ciò, in caso di malfunzionamento del sistema software, vanno attuati meccanismi di sicurezza per evitare che il malfunzionamento causi danni all'utilizzatore

  Affidabilita: Posso misurare l'affidabilità del sistema software tramite i malfunzionamenti che si sono manifestati. Quindi un sistema software è affidabile se si provocano pochi malfunzionamenti durante la sua esecuzione

- Usabilità: Facilità d'uso del software. Per l'utente deve essere semplice utilizzare il software, avere tutti i dati a portata di mano quando richiesti e non deve richiedere all'utente dati o informazioni che non ha. L'interfaccia grafica gioca un ruolo fondamentale in questo ambito. Da non confondere con accessibilità, infatti essa prevede l'utilizzo di tecniche specifiche, per persone appartenenti a specifici gruppi, di utilizzare comunque il software.

Spaghetti code
Il codice spaghetti code è un codice che non si basa sulla filosofia della programmazione ad oggetti. In questo tipo di programmazione abbiamo metodi lunghi, senza parametri e tante variabili globali. CI sono interazioni minime fra gli oggetti e inoltre il flusso di esecuzione viene deciso dai metodi e non dal chiamante. Questo tipo di approccio è contro la programmazione ad oggetti, infatti in OOP, è il chiamante a decidere il flusso di esecuzione e non i metodi. Già dai nomi delle classi e dei metodi posso individuare lo spaghetti code

Metodo di programmazione command and query: Con questo paradigma di programmazione vado a dividere quelli che sono i metodi che restituiscono valori dai metodi che servono ad aggiornare lo stato del processo. In questo modo riesco a ottenere i dati sullo stato del processo senza rieseguire tutte le operazioni, risparmiando così cicli di CPU.

Scrivere un codice con poche linee di codice per metodo aiuta l'individuazione degli errori e aiuta a correggerli senza dover analizzare necessariamente tutto il codice. I progetti software di grandi dimensioni, nonostante siano di grandi dimensioni, hanno veramente poche linee di codice per metodo. In questa manierà è semplicissimo aggiungere funzionalità e testare il software.
Andando a implementare il nostro sistema software tramite metodi piccoli e semplici ci da la possibilità di adottare un approccio bottom up. Partendo da metodi basilari è possibile andare a costruire metodi più complessi andando a richiamare i vari metodi più piccoli, in maniera tale da ottenere operazioni e risultati più complessi.

Oltre a questo ci è più semplice andare a verificare la correttezza del nostro sistema software, andando a implementare classi di test per il nostro sistema software. Le classi di test non sono altro che classi contenenti metodi che emulano il funzionamento del software in varie situazioni, più test si fanno e più è probabile che il software si comporti come ci aspettiamo.

I test devono sempre partire da una sitazione nota, cioè la situazione in cui il software è libero da precedenti test o residui di vecchie esecuzioni, per questo prima di ogni test le variabili vanno settate ai valori di default o svuotate nel caso di liste dinamiche.

I test devono essere autovalutanti, cioè ogni test deve sapere se il risultato ottenuto è quello desiderato. Il risultato desiderato viene indicato da un oracolo, un qualche cosa che genera il test e conosce a priori il risultato, dando così per scontato di conoscere a priori il risultato del test.

Per evitare errori all'interno del codice di test, esso deve essere semplice da scrivere e da correggere in caso di errori, per evitare di andare ad effettuare test falsati.

E' molto utile andare a stampare il risultato del test, sia che il risultato sia positivo o negativo. Questo ci permette di non andare a controllare a mano tutti i risultati andando a semplificare la vita all'esperto di testing.
A volte può essere utile andare ad utilizzare delle approssimazioni invece che andare ad utilizzare valori esatti, in maniera tale da avere tolleranza sul risultato ottenuto.

Mentre si va a scrivere il codice del sistema software è indispensabile andare a testare continuamente ciò che facciamo, ma oltre a testare ciò che stiamo scrivendo è utile andare a testare continuamente l'intero

sistema, in maniera tale da andare a limitare errori in altre parti del sistema mentre andiamo a modificare o implementare una nuova classe

Un altro metodo di implementazione del sistema software è il **test driven development**. In questo paradigma ci è vietato andare a scrivere codice prima di scrivere il test per esso. Prima di implementare delle nuove funzioni o andare a modificare la classe dobbiamo prima scrivere un test che ci darà per forza esito negativo. In questa maniera andiamo a dare importanza alla parte di progettazione del sistema, perchè mentre scriviamo il test ci facciamo un'idea di ciò che dovrà fare la classe che stiamo andando a scrivere andando a limitare codice in eccesso e funzioni non necessarie

Esistono 2 tipi di test, i **test whitebox e i blackbox**. I primi sono dei test dove chi scrive il test può leggere e conosce il codice su cui va ad implementare il test, mentre nei blackbox il creatore del test non sa cosa fa effettivamente il codice ma si basa sui requisiti che il codice deve rispettare. I blackbox ci permettono di andare a creare test a prescindere dal codice, in maniera tale da andare a prevedere più flussi di esecuzione del software

Nel paradigma a oggetti è importante che ogni metodo abbia un proprio ruolo e una propria responsabilità, in maniera tale da dividere in maniera semplice i ruoli tra i metodi e andare a risolvere il problema generale, passando da metodi a basso livello a metodi ad alto livello
Refactoring
E' il processo che cambia un sistema software, in maniera tale da lasciare il comportamento invariato ma cambiandone e migliorandone la struttura interna.

Grazie al refactoring è possibile migliorare la struttura del software a poco a poco, infatti a tempo di progettazione è difficile ottenere una progettazione che vada bene per tutta la durata dell'implementazione. Il refactoring ci consente di adattarci alle varie esigenze durante l'implementazione del sistema software.

Grazie al refactoring è possibile incorporare più facilmente nuovi requisiti o requisiti che cambiano, dato che andando a migliorare la struttura vado ad ottenere codice meno corposo e strutture complicate divise in strutture più semplici.

Molte volte, inoltre, succede che a causa di tempi ridotti viene data meno attenzione alla fase di progettazione, introducendo però un debito tecnico, dato che prima o poi il codice andrà sistemato. Il refactoring a tempo di implementazione evita questo debito tecnico, andando a dare una buona struttura al sistema software.

Grazie al refactoring viene rimosso codice duplicato e aiuta ad individuare eventuali bug nel sistema. Molte volte si ottiene un sistema software più veloce e reattivo, dato che vengono rimossi passaggi inutili dalla sua esecuzione.

Tecniche di refactoring
Estrai metodo
Nel caso in cui avessimo metodi lunghi, che svolge più di un compito, è bene dividere le responsabilità di quel metodo in più metodi

I metodi lunghi hanno bisogno di commenti per essere comprensibili, mentre se ho piccoli metodi, mi è possibile descrivere il metodo tramite il nome mi è più semplice capirne il funzionamento

I metodi a più alto livello risultano più comprensibili, visto che saranno formati da tanti piccoli procedimenti

Sostituisci temp con query

Supponiamo di avere una variabile temporanea, assegnata all'interno di un metodo. E' possibile racchiudere il risultato assegnato in un metodo esterno, in questo modo il risultato ottenuto non sarà disponibile solo a quel metodo ma sarà disponibile in tutto il programma.

Calcolare il risultato ogni volta che c'è bisogno di quel valore è sintomo di metodi lunghi, inoltre permette una maggior leggibilità del codice, visto che tramite il nome del metodo, abbiamo un'indicazione sull'utilizzo di quel valore

Dividi Variabile Temp
Nel caso in cui avessimo una variabile temporanea che ha più di una responsabilità è bene dividerla in 2 variabili temporanee distinte. Questo rende il codice più leggibile, dato che la responsabilità di quella variabile è descritta tramite il suo nome.

Non bisogna mai utilizzare variabili temporanee per più scopi, confonde il lettore

Ereditarietà

Nella programmazione a oggetti uno degli obiettivi è il riutilizzo di classi e di codice. Per andare a riutilizzare una sola parte di una classe, con ciò che abbiamo studiato ora, dovrei andare a importare l'intera classe, introducendo parti inutili di essa, una cosa che sarebbe preferibile evitare. Per rendere più snello e leggibile il codice è utile andare a utilizzare l'ereditarietà

L'ereditarietà ci permette di creare sottoclassi che ereditano le caratteristiche della classe padre. La classe padre in questo caso si chiamerà superclasse e deve essere la classe generale da cui andare a derivare le altre sottoclassi. Le sottoclassi saranno specializzazioni della classe padre

Esempio: Basti pensare al caso in cui io abbia una classe persona e una classe studente. Nel caso di programmazione a oggetti senza ereditarietà dovrei andare a implementare campi uguali in entrambe le classi, quali nome cognome o codice fiscale
Se vado a definire la classe studente come una sottoclasse di persona, vado a implementare questi campi sulla superclasse persona e vado a implementare i campi specializzati quali matricola o corso di laurea direttamente nella sottoclasse studente.

Questo va a evitare ridondanze nel codice e di poter importare sottoclassi invece che la classe intera nel caso in cui avessi solamente bisogno di una parte del codice e non del codice intero

Nota: In java non esiste l'ereditarietà multipla.

Nella classe studente posso andare ad aggiungere funzionalità o modificare le funzionalità della classe persona, ma non posso andare a rimuovere metodi implementati nella superclasse. Per modificare un comportamento mi basta andare a implementare un metodo nella sottoclasse con lo stesso nome (Override). Nelle sottoclassi posso richiamare metodi della superclasse, andando volendo, ad aggiungere funzionalità prima e dopo il richiamo del metodo della superclasse. Il richiamo della funzione dichiarata nella superclasse avviene tramite il metodo super().

Interfacce

Le interfacce sono dei tipi proprio come le classi, ma a differenza di quest'ultime esse non implementano I metodi dichiarati, ma lasciano alle sottoclassi il compito di farlo.
Se una classe va a legarsi ad un'interfaccia invece che ad una classe avrà l'obbligo di implementare i metodi dichiarati nell'interfaccia, in questo modo avremo sempre la sicurezza di avere un'implementazione completa della sottoclasse.
Nel caso in cui io voglia cambiare il comportamento dei metodi implementati posso banalmente creare un'altra implementazione dell'interfaccia e far puntare tutto il programma alla nuova implementazione

Classi Astratte

A primo impatto interfacce e classi astratte sono molto simili ma le classe astratte permettono l'implementazione di alcuni metodi a differenza delle interfacce. Le classi abstract non sono istanziabili, quindi i suoi metodi sono utilizzabili solamente se esistono implementazioni di una qualche sottoclasse.

C'è una differenza sostanziale tra i metodi abstract e i metodi di default. I metodi di default non possono cambiare lo stato della classe mentre i metodi abstract si

Nota Off topic: Overriding è l'implementazione di un metodo già dichiarato e implementato in una superclasse. Serve a specializzare il metodo per la sottoclasse. Con overloading invece andiamo a indicare più metodi con lo stesso nome, ma una volta richiamato il metodo, l'implementazione viene distinta tramite il tipo di parametri che stiamo passando alla funzione

Compatibilità fra tipi
L'ereditarietà in java permette di definire una classificazione di tipi. Una sottoclasse è un tipo compatibile con la sottoclasse, infatti è possibile contenere un sottotipo in una variabile di del tipo della superclasse. Ovviamente nel caso in cui andiamo ad assegnare un sottotipo ad una variabile del tipo della superclasse non ci è possibile richiamare funzioni dichiarate nella sottoclasse. Questo perché la variabile creata è un contenitore più piccolo della sottoclasse e il compilatore si aspetta di trovare le funzioni dichiarate nella superclasse
Quando andiamo a fare questo tipo di assegnazione e proviamo a richiamare una funzione non viene richiamata la funzione implementata nella superclasse ma viene richiamata quella del sottotipo.
Quando andiamo a richiamare una funzione, l'interprete controlla a runtime se esiste la funzione nell'istanza dell'oggetto. Se non è presente richiama quella della superclasse. Risale nella gerarchia fino a raggiungere alla sommità, la classe Object a cui tutti fanno riferimento

Nota: La ricerca e la scelta dell'implementazione del metodo da utilizzare viene chiamato dispatch

Late Binding
Il late binding è un meccanismo essenziale nel paradigma a oggetti. Il meccanismo che ci permette di scegliere quale implementazione della funzione eseguire a runtime viene chiamato late binding. Il comportamento della funzione implementata invece viene detto Polimorfo, cioè che può cambiare il suo funzionamento in base alle condizioni in cui ci troviamo a runtime. Senza polimorfismo e late binding dovremmo controllare il tipo dell'istanza tramite uno switch e scegliere manualmente l'implementazione della funzione da utilizzare
I processi evolutivi definiscono come e in che modo sono condotte le 5 fasi della progettazione di un sistema software
Le 5 fasi principali nella progettazione di un sistema software sono

- Analisi dei requisiti
- Progettazione
- Implementazione
- Convalida o testing
- Manutenzione

Analisi dei requisiti
L'analisi dei requisiti è il processo che serve a definire le specifiche del programma e stabilisce I vincoli software
Definisce:

- Requisiti: Ciascuna delle caratteristiche che il software deve avere
- Specifiche: Descrizione delle caratteristiche del software
- Feature: Un set di requisiti correlati tra loro. Una feature permette il raggiungimento di un obiettivo

Ci sono 2 tipi di requisiti

- Funzionali: Cosa il sistema deve fare

- Non-Funzionali: Come il software implementa I requisiti funzionali

Fase di progettazione e implementazione
Serve a progettate la struttura che avrà il software, a individuare le dipendenze e la responsabilità che avrà ogni parte del sottosistema software. Vengono progettate le interfacce, le strutture dati e gli algoritmi.

Ogni attività nella fase di progettazione produce una documentazione che descrive il modello

Nella fase di implementazione vengono implementate le strutture dati, gli algoritmi e le classi a partire dal modello descritto nella fase di progettazione. Oltre all'implementazione della struttura definita nella documentazione nella fase di implementazione vengono individuati e rimossi gli errori nella documentazione

Le fasi di progettazione e di implementazione sono correlate e spesso alternate tra loro

Fase di convalida o testing
La fase di convalida è la parte dove si intende mostrare che il sistema software è conforme alle richieste definite nell'analisi dei requisiti.
Per fare ciò viene fatto testing sul sistema software, eseguendolo con condizioni e dati stabiliti dal cliente

Ci sono diverse parti nella fase di testing
- Unit test: Vengono testate singole classi o singoli componenti, con test stabiliti e scritti dal programmatore
- Test di sistema: L'intero sistema è testato
- Test di accettazione: L'intero sistema è testato con dati forniti dal cliente. Si va a verificare se le specifiche sono state rispettate
- Beta test: Il sistema quasi completo viene fornito a un piccolo gruppo di utilizzatori che avranno il compito di testare il sistema software e di segnalare eventuali problemi

Processi evolutivi
Waterfall
Il processo evolutivo waterfall è un processo statico e con tanta documentazione. Si passa alla fase successiva solo se quella precedente è completa.
E' caratterizzata da poca interazione con i clienti e tempi lungi di produzione. C'è molta difficoltà ad applicare le modifiche proposte dal cliente, visto che tutta la progettazione viene fatta all'inizio e sarà difficile aggiungere nuove funzionalità
Viene in genere utilizzata per gestire sistemi grandi e critici. Produce un'ampia documentazione e il codice avrà un'ottima qualità

Evolutivo
Abbiamo 2 varianti del processo evolutivo
- Sviluppo per esplorazione: Gli sviluppatori lavorano con i clienti, che gli fornisce delle specifiche di base e per mezzo di trasformazioni si arriva al sistema software finale
- Bug and fix: Documentazione inesistente. Si passa alla fase di implementazione senza quella di design. Produco una versione e la modifico fino a quando il cliente non è soddisfatto. Di conseguenza il codice prodotto è di scarsa qualità

Un problema di questo processo è sicuramente la difficile comprensione dei codici prodotti e la scarsa modificabilità. Va utilizzato per sistemi di piccole dimensioni o per piccole parti di sistemi grandi

A spirale
Le attività sono organizzate su una spirale. Ogni loop della spirale è una fase
Ogni loop consiste in 4 settori:

- Obiettivi per la fase corrente
- Valutazioni rischi del progetto
- Produzione di una parte e convalida
- Revisione e pianificazione fase successiva

Il processo a spirale è un processo agile, serve poco tempo per la prima versione del prodotto e c'è la possibilità di interagire col cliente, chiedendo modifiche e supportandole.
Ogni fase nel processo a spirale produce del codice testato e integrato nel sistema

Extreme programming

Sviluppo Agile
Nello sviluppo agile viene data priorità all'auto organizzazione, alla collaborazione e alla comunicazione invece che alla struttura della progettazione stessa. Viene data più importanza all'individuo che al progetto in sè.

Nello sviluppo agile viene preferita l'implementazione di un sistema software semplice e funzionante il prima possibile, per fare ciò il cliente è al centro del progetto e può proporre le sue modifiche quando gli pare. La scelta del cliente di apportare modifiche viene supportata

Extreme programming
L'extreme programming si basa sullo sviluppo e la consegna di piccoli incrementi di funzionalità
Si hanno solo 2 settimane per lo sviluppo degli incrementi, ci sono piccoli gruppi di sviluppatori e la documentazione si basa su storycard. Vengono coinvolti tutti nello sviluppo, pure il cliente, che avrà il compito di fornire supporto, fare richieste e proporre cambiamenti in loco. I prodotti con questo approccio sono perennemente sotto testing

XP è fortemente adattivo, quindi è il giusto approccio per progetti dove I requisiti non sono stabili ma cambiano nel tempo

Vengono fissate 40 ore di lavoro settimanali per I dipendenti, questo gli consente di essere più tranquilli alla stesura del codice e previene errori gravi durante l'implementazione dei task

Story Card
Le story card sono richieste e feature che il sistema software dovrà avere. Il cliente scrive le story card contenente la richiesta, da essa potrà essere ricavata una stima sul tempo di sviluppo. Il cliente scrive story card per riempire 3 settimane. A ogni story card possono essere assegnate delle priorità

Le storie devono essere semplici, se gli sviluppatori non riescono ad ottenere una stima la storia viene rimandata al cliente che avrà il compito di dividerla in più storie
La creazione delle storie è esente da dipendenze funzionali

La gestione del progetto a release incrementali consente di pianificare grossolonalemente l'intera release, ma non ci consente di pianificare troppo avanti

Per l'attuale release gli sviluppatori dividono le storie In task e ciascuno si impegna per realizzarne uno. Vengono svolti prima I task più rischiosi

Piccole release
XP si basa sul rilascio di piccole release più semplici possibile. Ogni release ha un tempo di sviluppo di 2 settimane e deve essere stabilito un design semplice e funzionale per la release corrente
Piccole release forniscono agli sviluppatori un feedback rapido su ciò che stanno creando, toccando con mano I frutti del loro lavoro. Andando a sviluppare delle release si ottiene la fiducia del cliente

Il giusto design si ha quando passa I test per le feature previste per quella release, non ha parti duplicate e esprime qualsiasi funzione importante prevista dagli sviluppatori
Le CRC card aiutano a tenere traccia dello stato del sistema software

Testing
In XP il sistema software è perennemente sotto testing. Non appena si produce del codice nuovo viene subito testato
Ci sono 2 tipi di test

- Unit test: Scritti dal programmatore, servono per testare ciò che ha scritto con dati e test scritti da lui
- Test funzionali: Scritti dal cliente servono a testare le feature del sistema software.

Integrazione continua
Integrazione del codice testato ogni poche ore. Tutti gli unit test devono essere superati, se non li supera la coppia che ha scritto il codice deve ripararlo, se non può ripararlo si ricomincia da zero

Altre caratteristiche di XP sono la codifica comune dei vari task e il refactoring.
Tutto il codice è scritto nello stesso stile, seguendo le convenzioni stabilite dal linguaggio. Questo rende il codice più leggibile e coerente. Il refactoring è importante in XP. Si tende a fare refactoring in coppia e a piccoli passi, supportarti dai test e dal design semplice delle varie release. Si punta a codice senza ripetizioni