CICLO DI VITA E INGEGNERIA DEL SOFTWARE

Problema:

Fino a 20 anni fa circa ¼ di tutti i progetti software (d'ora in poi scriverò sw per indicare software e hw per indicare hardware) veniva interrotto prima della conclusione e la metà veniva terminato con tempi e costi molto maggiori di quelli preventivati. Perché? perché non si applicavano le metodologie dell'**ingegneria del sw** durante la realizzazione del progetto sw.

Ingegneria del sw: disciplina tecnologica e gestionale che si occupa della realizzazione e della manutenzione di un prodotto o servizio sw **RISPETTANDO I TEMPI E I COSTI PREVENTIVATI**.

La progettazione di costruzioni civili o edilizie ha una lunga storia, quindi i progettisti sono ormai consapevoli della necessità di applicare delle metodologie **SISTEMATICHE** (ossia applicate in modo ordinato, organizzato, preciso, regolare, metodico, coerente) nella progettazione, mentre nella progettazione sw questo inizialmente non accadeva.

Sottolineo inoltre che la progettazione rispettando i criteri indicati dalle varie metodologie porta a produrre del software di qualità. Gli sviluppatori inesperti ritengono che l'oggetto del proprio lavoro sia il file eseguibile dell'applicazione che stanno realizzando. In realtà il prodotto del loro lavoro è il codice sorgente. Quindi è importante non solo che un'applicazione funzioni ma anche che il codice sia ben scritto. Infatti si stima che l' 80% del tempo di lavoro su un'applicazione è dedicato ad interventi di manutenzione e un software scritto male, senza il rispetto delle regole stabilite internamente all'azienda, senza i commenti, senza documentazione, porta ad incrementare questo tempo di lavoro generando, in alcuni casi, l'antieconomicità del prodotto software.

Come si realizza un prodotto o servizio software? Innanzitutto chiariamo la differenza fra prodotto software e servizio software. Un prodotto sw è un software che viene acquistato da un cliente il quale paga per avere il **possesso** di tale prodotto ed installarlo sul proprio dispositivo. Indichiamo con **servizio software** una particolar forma di distribuzione del software, generalmente in abbonamento, in cui il cliente non paga per il possesso di un software, ma per il suo utilizzo online. Ad esempio un sw che fornisce servizi online: online banking, motore di ricerca, social network, registro elettronico, Office 365 (non richiedono installazione). Quest'ultima modalità di fruizione di un software è indicato con il termine di **Saas** (**Software as a service**).

Le fasi della realizzazione di un software sono 5 e sono riportate nel modello a cascata del ciclo di vita di un sw: analisi, progettazione, sviluppo o implementazione, verifica o test, manutenzione.

CICLO DI VITA DEL SW: insieme delle attività ed azioni da intraprendere per la realizzazione di un progetto sw

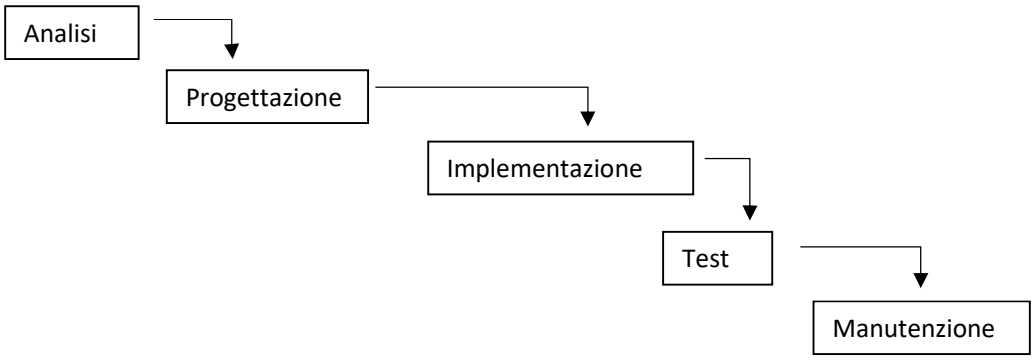
Il ciclo di vita viene rappresentato attraverso il MODELLO A CASCATA nel quale la realizzazione del sw è suddivisa in fasi da realizzarsi in sequenza e nel quale ciascuna fase (tranne l'ultima) produce un output che è l'input della fase successiva.

Il modello a cascata, seppur ormai superato, è importante per capire quali sono le fasi che sono SEMPRE PRESENTI, in un processo di realizzazione del software.

FASE	SCOPO	OUTPUT	NOTE
Analisi	Identificare i **requisiti** del prodotto sw da realizzare **Cosa** deve fare il sistema informatico, **non come!**	Requisiti (specifiche) Verrà prodotto un vero e proprio documento (o elenco) dei requisiti (o specifiche)	Il progettista (o staff di progettazione) deve collaborare in maniera molto stretta con il committente (l'ente o persona che commissiona il sw). L'analista si interfaccia generalmente con una figura esperta del settore in cui il software andrà a funzionare, chiamato **esperto del dominio** (non è un informatico). • Generalmente parlano linguaggi diversi. • L'esperto del dominio può dare per scontato molte cose. Con dominio si intende l'ambito di funzionamento del sw. Esempio: per il registro elettronico il dominio è il mondo della scuola.
Progettazione	Si occupa di **come** verranno realizzati i requisiti richiesti. Detta anche design. Per semplificare la realizzazione del prodotto, esso viene suddiviso in **componenti.** L'insieme di tali componenti e le relazioni fra di loro formano l'architettura del software	Architettura software: ossia la descrizione dei **componenti software di cui** sarà costituito il programma. Esempi di componenti possono essere: le strutture dati (array, grafi…), le classi (nella programmazione ad oggetti), le librerie da utilizzare.	I componenti dovrebbero essere il più possibile indipendenti fra loro e testabili autonomamente. Il paradigma della OOP (Object Oriented Programming) favorisce questa indipendenza dei componenti.
Implementazione (sviluppo) (realizzazione)	**Realizzazione** dei singoli moduli nel linguaggio di programmazione scelto.	Codice sorgente/eseguibile Versione da testare. (Da verificare poi nella fase successiva) **Documentazione del codice**	
Verifica (test)	Verifica, con procedure **opportunamente predisposte**, della corretta	Codice sorgente/eseguibile da rilasciare	Risponde alla domanda: il programma svolge **correttamente, completamente efficientemente** il compito per cui è stato sviluppato?

	integrazione dei componenti e del soddisfacimento dei requisiti.		Ci sono apposite tecniche di testing che studieremo.
Manutenzione	Può essere sia correttiva sia evolutiva. **Correttiva**: correzione di eventuali errori (bug) non rilevati in fase di test. **Evolutiva**: aggiunta di ulteriori funzionalità al prodotto	Nuove versioni (release) del codice sorgente/eseguibile	La manutenzione evolutiva può essere resa necessaria sia per non aver rilevato alcune esigenze in fase di analisi, sia perché sono emerse nuove esigenze durante lo sviluppo o durante l'utilizzo del prodotto. Sia perché il contesto organizzativo per cui è nato il software è cambiato. Si stima che la fase di manutenzioni occupi circa l'80% del ciclo di vita di un software.

Il modello a cascata del CICLO DI VITA del software e le sue fasi *(software waterfull life-cycle)*



Perché è superato?

> perché le problematiche che emergono in fase di test del codice portano a dover ripetere le fasi di implementazione (sviluppo) e, addirittura, possono richiedere di svolgere nuovamente le fasi di analisi e progettazione. Spesso durante la valutazione di un prototipo il committente individua nuove specifiche che richiedono una nuova fase di analisi, progettazione, sviluppo, test (o verifica).

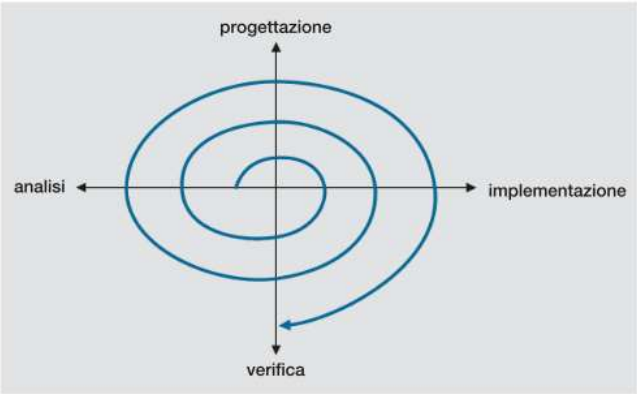
E allora?

Il modello a cascata è stato sostituito da un modello chiamato **modello a spirale** del ciclo di vita del software. In questo modello, dopo la fase di test, la fase di manutenzione coincide con una nuova fase di analisi e quindi viene ripetuta l'esecuzione delle varie fasi. Il modello a spirale, rispetto al modello a cascata è incrementale e iterativo, vediamo cosa significa:

incrementale: il sw viene prodotto in versioni successive ciascuna delle quali è un'evoluzione della precedente che ne migliora e ne accresce le funzionalità.

iterativo: (ciclico) perché le fasi di analisi, progettazione, implementazione (o sviluppo) e test, vengono svolte ripetutamente.

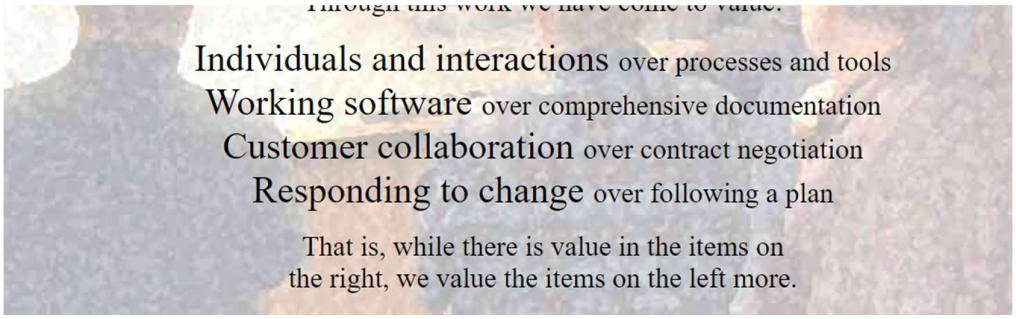
Il ciclo di vita del software può dunque essere rappresentato con il grafico a spirale che ne evidenzia l'aspetto iterativo.



METODOLOGIE AGILI

Negli ultimi anni si sono affermate **metodologie di sviluppo** sw meno strutturate chiamate **metodologie agili**.

La parola agile enfatizza la capacità di tali metodologie di rispondere rapidamente ai **cambiamenti di specifiche**. Le metodologie agili si riconoscono in un manifesto (https://agilemanifesto.org/) e in alcuni princìpi. I più interessanti sono:



Le metodologie agili enfatizzano il ruolo del **testing** e del **refactoring** per assicurare la qualità del codice rilasciato. Si arriva addirittura, in alcuni casi, a teorizzare la predisposizione del codice di test prima ancora dello sviluppo dell'applicazione (*test-driven developement*). Quindi prima vengono creati i test automatici e poi si sviluppa il software in modo che superi i test. Con il termine **refactoring** si intende la modifica frequente del software **già testato, funzionante e conforme ai requisiti,** per migliorarne la flessibilità in fase di manutenzione sia correttiva sia evolutiva. Le due attività (testing e refactoring) sono legate fra loro perché per testare le frequenti modifiche ai sw introdotte con il refactoring, è opportuno avere dei test di semplice attuazione ed automatici. Vedremo appositi strumenti software per l'automatizzazione dei test

Inoltre le metodologie agili, basandosi su continui e frequenti rilasci del codice, non sarebbero applicabili se non grazie alla diffusione della modalità di distribuzione di servizi sw online. Tale modalità non richiede l'installazione di sw in locale ma la fruizione, da locale attraverso dei browser, di servizi sw posti su server. Tale modalità è indicata con il termine **Saas (Software as a Service).** La modalità Saas, infatti, consente alla software house di rilasciare frequenti versioni dei software senza obbligare il cliente a continui download ed installazioni di patch software. La modslità Saas viene utilizzata, ad esempio nel registro elettronico mastercom o nei wordprocessor e spreadsheet di Google e di Microsoft.

IL FRAMEWORK SCRUM

SCRUM è una piattaforma (framework) di progettazione Agile. Il nome SCRUM identifica il pacchetto di mischia del gioco del rugby.

Scrum viene utilizzato principalmente nell'ambito dello sviluppo software ma non solo. Infatti è una metodologia che può essere utilizzata per guidare un qualsiasi team di lavoro che deve risolvere in maniera iterativa e incrementale un progetto complesso.

Lavoro di approfondimento:

Guardare i seguenti 4 video di presentazione di SCRUM

1 https://www.youtube.com/watch?v=afj_tAuCMBA
2 https://www.youtube.com/watch?v=ypjUq2hKAQE
3 https://www.youtube.com/watch?v=qmYtilbg1rA
4 https://www.youtube.com/watch?v=NesNtCrjL6g

Bella e chiara presentazione su come funziona SCRUM

1. https://www.youtube.com/watch?v=wYNSqxpan_U (presenta Scrum)

Domande guida :

Cosa sono gli stackholder?

Cosa sono le user story?

Cosa è il product backlog?

Cosa fa lo scrum master? Rimuove gli ostacoli, facilita una corretta esecuzione del processo.

Cosa è uno sprint? L'unità di base dello sviluppo SCRUM dura da 1 a 4 settimane, alla fine c'è un rilascio del prodotto (il prodotto rilasciato alla fine dello sprint è chiamato incremento)

Sprint planning?

Sprint backlog?

daily scrum?

Sprint review?

Sprint retrospective?

Interessante è il l'evento daily scrum (cosa ho fatto ieri? Cosa farò oggi? Quali sono gli impedimenti che hanno ostacolato il mio lavoro'?)

LINGUAGGIO UML (Unified Modeling Language)

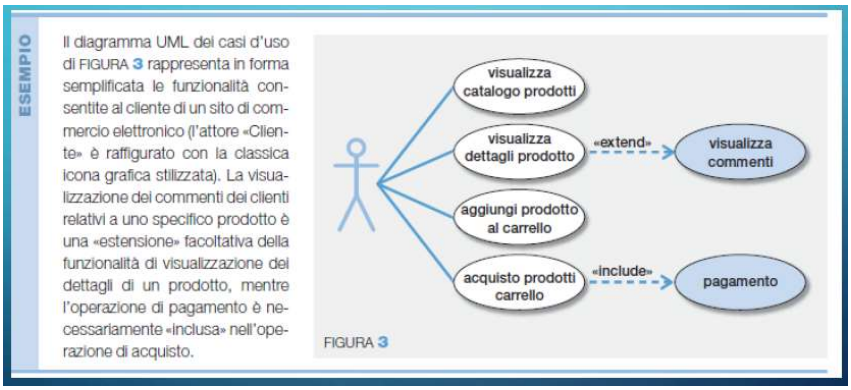
Cosa è?

E' un linguaggio di modellizzazione. Consiste in un insieme di formalismi grafici (diagrammi), che consente di definire e descrivere un software da diversi punti di vista. E' stato ideato nel 1996. L'obiettivo dei diagrammi è quello di costruire, con un linguaggio grafico ben determinato, un modello del sw da realizzare, visto da più punti di vista diversi. L'insieme di tali punti vista costituirà quello che viene definito **Visual Modeling**. Questo linguaggio è particolarmente utile nella OOP (Object Oriented Programmig, la programmazione ad oggetti).

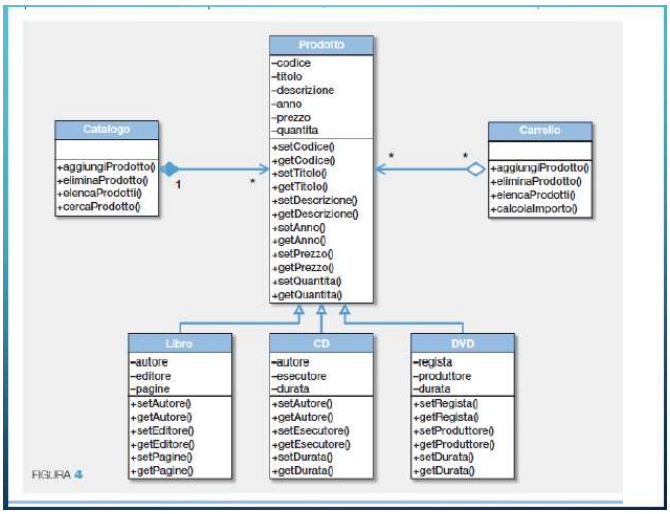
I diagrammi previsti dal linguaggio UML sono 14 (nella sua ultima versione 2.5.1 del 2017, le specifiche si possono scaricare dal sito: https://www.uml.org/), quelli che utilizzeremo sono 7, e si classificano in: diagrammi **comportamentali** (rappresentano dei comportamenti, una sequenza di avvenimenti), diagramma **strutturali** (rappresentano il modello del sw in maniera statica) e un **diagramma di interazione.**

Ogni diagramma appartiene ad una di queste tre tipologie. Presentiamo sinteticamente i diagrammi principali. In seguito, durante l'anno studieremo l'utilizzo di ciascuno di questi diagrammi.

- **Diagramma dei casi d'uso**: è un diagramma *comportamentale* che descrive le **funzionalità** (cosa fa) di un sistema software **dal punto di vista dell'utente** del sistema. La descrizione è in termini di **utenti** ed **azioni eseguibili**. Gli utenti possono essere sia persone che altri sistemi informatici e sono indicati con il termine di "*attori*" (attore è "chi usa il sw").



- **Diagramma delle classi**: è un diagramma *strutturale* che descrive **l'architettura** del software. Indica come sono realizzate le **classi**, con i relativi **attributi** e **metodi**, e le **relazioni** fra di esse.

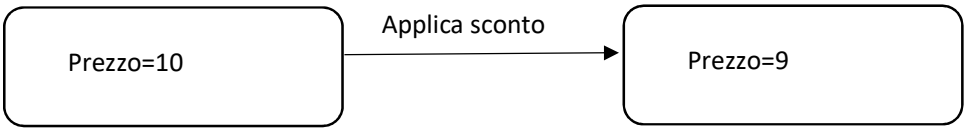


- **Diagramma degli oggetti**: è un digramma *strutturale* che descrive lo **stato** di tutto o solo di una parte del software, in un determinato istante dell'esecuzione. Con stato si intende "i valori assunti dagli oggetti", ossia dalle istanze delle classi. (Nella programmazione imperativa lo **stato** di un programma l'avevamo definito come "il valore di tutte le variabili in un determinato istante", nella OOP lo **stato** è l'insieme degli oggetti istanziati in un determinato istante). Il diagramma degli stati rappresenta delle istanze del diagramma delle classi. Si usa generalmente per mostrare degli esempi di classi. Ad esempio in una istanza della classe Libro del diagramma precedente, in un certo istante può essere questa:

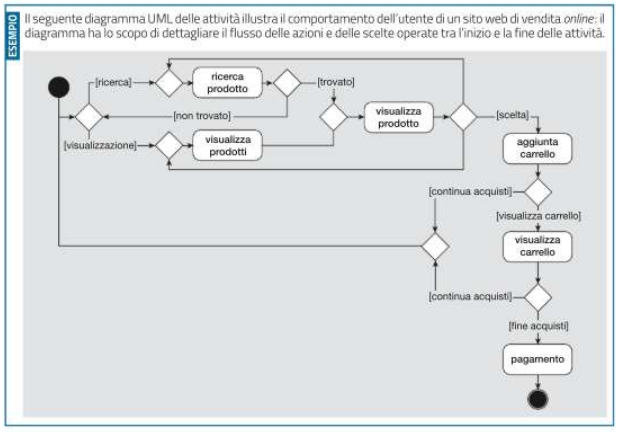
Libro1
Codice = 0010
Titolo = Zanna Bianca
Descrizione = Versione economica
Anno = 1925
Prezzo = 10
Quantita= 1
Autore = Jack London
Editore = Mursia
Pagine = 312

- **Diagramma degli stati:** è un diagramma *comportamentale*, rappresenta le transizioni di stato di **un** componente software (un oggetto) **in seguito ad un evento** che provoca un cambiamento di stato. (non necessario per tutti gli oggetti, si usa solo se serve)

  Esempio, diagramma di stato dell'oggetto libro1



- **Diagramma delle attività**: è un diagramma *comportamentale*, illustra le fasi del flusso di controllo di un sistema sw, ossia la sequenza degli eventi che accadono fra l'inizio e la fine delle attività. Diversamente dal diagramma dei casi d'uso, in cui viene mostrato tutto ciò che l'utente può fare, senza dire "cosa deve fare prima e cosa deve fare dopo", in questo diagramma vien mostrata la sequenza di operazioni che l'utente può svolgere dall'inizio alla fine delle attività.



- **Diagramma delle sequenze**: è un diagramma *di interazione*, descrive la **comunicazione** fra gli oggetti di un sistema sw. Con "comunicazione" si intende lo **scambio di messaggi** ossia **l'invocazione di metodi** di un oggetto da parte di un altro oggetto. In un sistema sw vengono costantemente fatte richieste e inviate risposte fra gli oggetti. La linea verticale tratteggiata rappresenta il passare del tempo, ciò che è rilevante non è il tempo in valore assoluto ma l'ordine in cui i messaggi sono inviati, ossia è importante mostrare che un messaggio deve essere inviato dopo un altro. Il rettangolo bianco indica la "vita" (timelife) di un oggetto.

  I diagrammi di sequenza UML sono utili se si desidera rappresentare graficamente **operazioni complicate** per renderle più comprensibili.

  Nei diagrammi di sequenza Generalmente non si rappresenta l'intero sistema informatico ma una sua parte.

ESEMPIO

Un diagramma UML di sequenza come quello riportato di FIGURA 5 illustra l'interazione nel tempo (che trascorre dall'alto verso il basso) tra i componenti di un sistema software, in particolare – se riferito a un linguaggio di programmazione orientato agli oggetti – mostra la sequenza temporale di invocazione dei metodi tra oggetti di classi diverse.

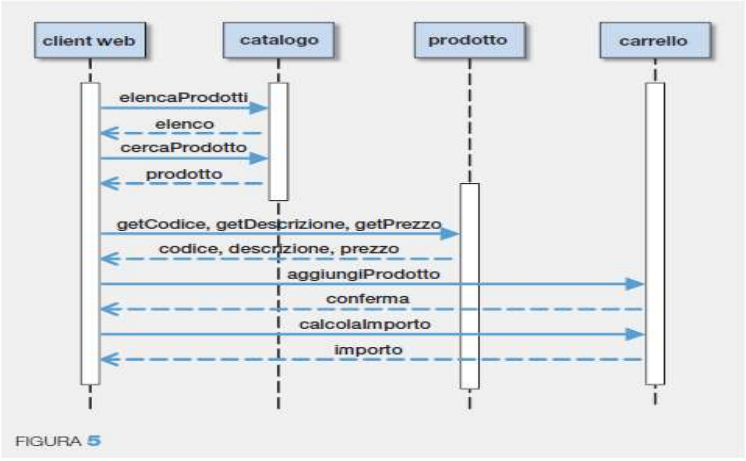
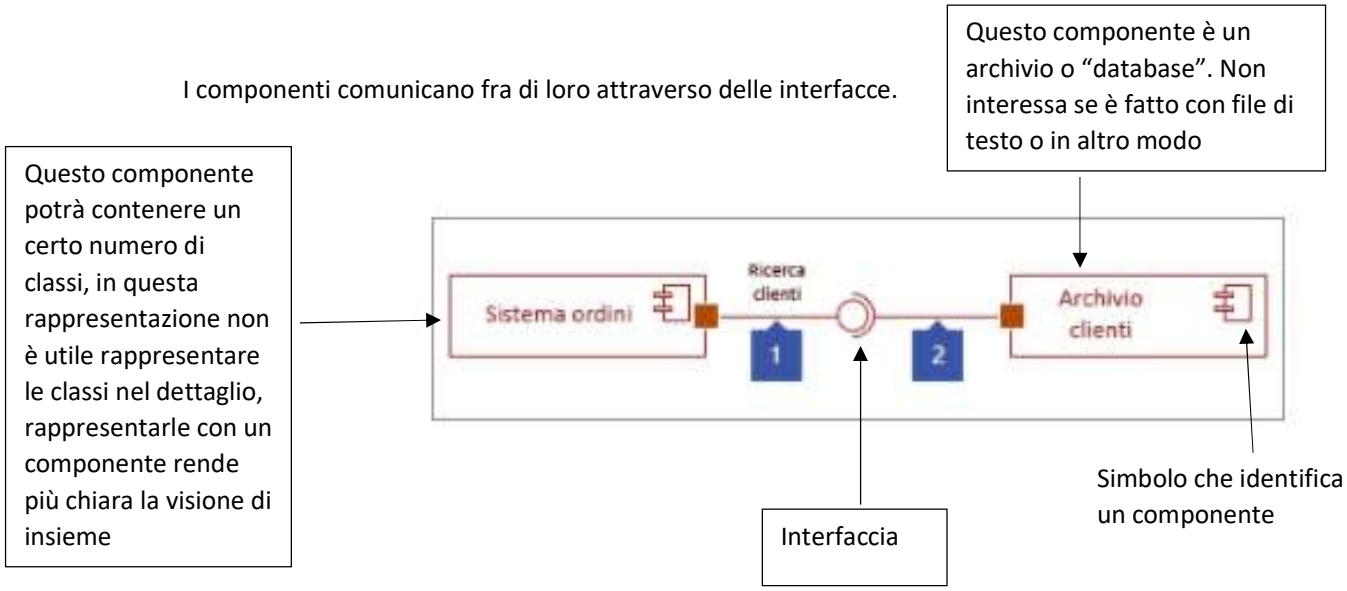


FIGURA 5

- **Diagramma dei componenti**: serve a rappresentare l'architettura di un software in termini di **relazioni fra i componenti.**
  Abbiamo già visto un diagramma che ci consente di rappresentare l'architettura del software, il diagramma delle classi. Tale diagramma fornisce una visione dettagliata delle classi (con i propri metodi e attributi), mentre il diagramma dei componenti fornisce una visione più ampia dell'architettura, infatti:

1. Un componente non è solo una classe, è un concetto più ampio che include sia componenti logici (**le classi**), sia componenti fisici (**file, librerie, basi di dati)**
2. Un componente può includere al suo interno più componenti, per questo il diagramma dei componenti fornisce una visione d'insieme più ampia dell'architettura del software.

I componenti comunicano fra di loro attraverso delle interfacce.



QUALITA' DEL SOFTWARE

Per molti prodotti industriali, è abbastanza intuitivo confrontare la qualità di due prodotti. Ad esempio per un'automobile: affidabilità (un'auto con i ricambi garantiti per 10 anni, è più affidabile di una con i ricambi garantiti un anno), sicurezza (un'auto con 10 airbag e abs è più sicura di una senza airbag e abs) , prestazioni (si confrontano velocità, accelerazione), efficienza (3 litri di carburante per 100 km anziché 4 litri per 100 km). Definire cosa sia un sw di qualità è un po' meno intuitivo.

Per ogni prodotto industriale l'ente **ISO (International Standard Organization)** produce delle norme che definiscono degli standard di qualità. Molto nota è la norma ISO 9001 che non si riferisce ad uno specifico prodotto ma si riferisce in generale alle organizzazioni (generalmente alla aziende). In pratica, tale norma, definisce delle procedure che devono essere messe in atto all'interno dell'azienda, per garantire la qualità della produzione. Un'azienda può ottenere una certificazione ISO 9001 (per farlo deve rivolgersi, a pagamento, ad una società che rilascia la certificazione), e in questo modo potrà presentare ai suoi clienti una certificazione, riconosciuta in tutto il mondo, della qualità del proprio lavoro.

Anche per il software, l'ISO ha stabilito, con una sua norma, quali sono i requisiti di qualità di un software, e un modo per misurare tali requisiti attraverso degli algoritmi. Tale norma è la **ISO/IEC 25010 del 2011. (IEC significa International Electrotechnical Commition).**

Nell'ultima versione della norma ISO/IEC 25010 le caratteristiche di qualità elencate sono 14, vediamo le più significative.

Compatibilità: livello di coesistenza e interoperabilità con altri software, ad esempio in MS Word posso importare un file di testo creato con Notepad ma non posso fare il viceversa. Il livello di compatibilità di Word è maggiore di quello di Notepad.

Funzionalità: presenza delle funzionalità che soddisfano i requisiti. "Il software fa quello per cui è stato realizzato? Per il sw non è così facile dimostrarlo, come invece può esserlo per altri prodotti industriali (automobile, sedia…), si dimostra attraverso dei test ed è necessario conoscere i requisiti richiesti.

Affidabilità: la norma la definisce come la capacità del prodotto software di mantenere uno specificato livello di prestazioni quando usato in date condizioni per un dato periodo. Per un software embedded che gestisce sempre lo stesso hardware l'affidabilità è verificabile in maniera più semplice (misurando il tempo medio che passa fra il verificarsi di due errori) rispetto all'affidabilità di software che richiedono interazione con l'utente.

Usabilità: stabilisce quanto è facile il software da utilizzare (quanto è "user friendly"), quanto è più o meno intuitivo. Ad esempio l'interfaccia GUI di Windows è più usabile del prompt dei comandi.

Efficienza: indica la relazione fra la quantità di risorse utilizzate (quanta CPU e quanta Memoria) e le prestazioni. Ad esempio due software di elaborazione grafica diversi possono occupare una quantità di memoria centrale diversa per modificare la stessa immagine nello stesso modo.

Manutenibilità: Indica quanto il sw è semplice da modificare, per ottenere la manutenibilità è fondamentale la modularità, ossia costruire classi il più possibile autonome e indipendenti. Se devo modificare una parte di codice di una classe, potrò modificarlo solo una volta, non dovrò andare a modificare il codice in diversi punti.

Portabilità: Facilità di esecuzione su piattaforme diverse. Il sito è visualizzabile in maniera corretta su tablet, smartphone, pc? Oppure: il software è eseguibile sui diversi sistemi operativi? La macchina virtuale di JAVA, ad esempio, favorisce estremamente la portabilità rispetto ai software compilati in maniera tradizionale.

Sicurezza: Capacità di garantire la conservazione di dati corretti e di verificare gli accessi ad essi.

DESIGN PATTERN

E' una soluzione (generica) per un problema ricorrente nell'ambito della progettazione (del software, ma non solo).

L'idea di pattern è nata in ambito urbanistico dall'architetto Cristopher Alexander.

In ambito di progettazione del software, il riferimento per i progettisti è dato dall'insieme di 23 pattern indicati dai ricercatori Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (***the gang of four***) nel loro testo pubblicato nel 1995 ***Design Patterns: elementi per il riuso di software orientato agli oggetti*.**

A cosa serve un design pattern? Grazie ad un design pattern, il progettista non deve più pensare a COME risolvere un problema progettuale ma A QUALE PATTERN RICONDURRE il suo problema. Il design pattern fornirà la soluzione generale, che il progettista dovrà adattare al proprio caso. Il progettista risparmierà molto tempo nella progettazione perché le soluzioni alle problematiche più ricorrenti sono già pronte.

Ad esempio: in un software molto spesso si ha la necessità di "scandire" una serie di elementi (una lista di studenti, una lista di prodotti, una lista di fatture) ed accedere ai dati di uno di questi elementi. Esiste un pattern chiamato **Iterator** che consente di accedere ad una "lista" di elementi qualunque essi siano (studenti, prodotti, fatture..)

Lo schema con cui si documentano i pattern è il seguente (p 13. del libro).

PATTERN

Nome	*Make it run, make it right, make it fast, make it small.*
Problema	Quando ottimizzare un progetto software?
Contesto e forze	Si sta affrontando un nuovo progetto e la soluzione richiesta deve normalmente essere «migliore, più veloce e più economica» ed è necessaria una strategia di sviluppo che bilanci le necessità del cliente o del mercato e organizzative. L'ottimizzazione ha costi sia a breve termine sia a lungo termine. L'ottimizzazione precoce porta con sé il rischio di alti costi per risultati limitati: gli sviluppatori normalmente non sono in grado di prevedere quali saranno gli *hot-spot* del progetto e posticipare l'ottimizzazione fino al momento in cui essi non sono noti si è spesso dimostrato saggio («Molti errori di progettazione e programmazione sono stati compiuti in nome dell'efficienza più che per ogni altro singolo motivo, compresa la stupidità cieca»; «Il miglioramento dell'efficienza spesso viene ottenuto a costo di sacrificare qualsiasi altra desiderabile caratteristica del prodotto»). Lo sviluppo incrementale ha un effetto estremamente positivo sul morale degli sviluppatori: ottenere qualcosa che funziona in una fase iniziale del processo di sviluppo e che cresce progressivamente mantiene l'entusiasmo e la visione ad alti livelli. La correttezza dei prodotti è una caratteristica estremamente desiderabile, prodotti software veloci ottengono una valutazione positiva da parte del committente e i progetti «piccoli» sono più facilmente distribuibili e mantenibili. ▶
Soluzione	Adottare un ciclo di sviluppo iterativo che ordini gli obiettivi come di seguito: 1. *make it run* (garantire il funzionamento); 2. *make it right* (garantire il rispetto dei requisiti); 3. *make it fast* (ottimizzare le prestazioni); 4. *make it small* (ottimizzare la struttura interna).
Contesto risultante	La ricerca dell'efficienza non impatta eccessivamente sulla buona progettazione dell'architettura; il morale del gruppo di sviluppo rimane elevato e solo le ottimizzazioni realmente necessarie sono effettuate.