APPUNTI
Fondamenti di Ingegneria del Software

Università degli studi di Verona

Antonio Panfili

22 febbraio 2022

Indice

1 Introduzione

Il software è un elemento indispensabile per le moderne economie, per gli stati e le organizzazioni. Il problema del software è che è astratto ed intangibile quindi è difficile identificarne le proprietà di qualità e robustezza.
Di fatto quando si sviluppa il team è composto da persone ed ognuna di esse conosce molto bene una parte e poco tutte le altre parti. Questo solitamente causa grossi problemi che riguardano principalmente la qualità del software che può degenerare velocemente trasformandolo in qualcosa di molto difficile da mantenere, gestire ed utilizzare.

L'ingegneria del software è una serie di tecnologie e teorie per supportare lo sviluppo professionale di software.
Se parliamo di quanto costa il software cioè quanto costa svilupparlo, sappiamo che il costo è di vari ordini di grandezza superiore rispetto al costo dell'hardware. Tanto più ora che, nella maggior parte dei casi, il costo dell'hardware è diventato il prezzo dell'utilizzo di un servizio cloud.
Dalle statistiche si evince che il costo per la manutenzione del software è di gran lunga superiore al costo per il suo sviluppo. La manutenzione significa anche aggiornamenti che si rendono necessari per l'evoluzione delle normative e delle leggi alle quali il software deve sottostare.
Ci possono anche essere problemi legati al cambiamento del contesto nel quale il software è eseguito per il quale si necessita anche del cambiamento del sistema software.
Un esempio di software in esecuzione da decenni sono i software legacy delle banche; il costo di manutenzione in questo caso è di gran lunga più alto rispetto a quello della sua produzione.
La disciplina dell'ingegneria del software ha come obiettivo quello di supportare uno sviluppo software che sia cost-effective, cioè che possa soddisfare gli obiettivi di costo che possono essere sostenuti dal cliente.
Secondo una ricerca, solo circa il 29% dei progetti vengono terminati con successo (entro i tempi ed entro il budget). La maggioranza dei progetti, invece, ha raggiunto dei compromessi di tempi e costi. In alcuni casi si è ritardata la scadenza, in altri si sono accettati costi maggiori, in altri ancora si è accettato di avere un sottoinsieme di funzionalità rispetto a quelle richieste inizialmente.
I problemi tipici dei progetti software sono:

- **Stima dei costi**: i costi effettivamente sostenuti risultano superiori a quelli stimati all'inizio. Questo perché stimare i costi del software è impossibile data la natura intangibile dello stesso. A meno che non si abbia l'esperienza di realizzazione di un sistema molto simile da utilizzare come confronto.
- **Stima del tempo**: i costi possono anche essere stimati correttamente, ma il software viene consegnato oltre i tempi definiti dal contratto. In questo caso il committente potrebbe anche venire meno al pagamento del contratto.
- **Errata interpretazione**: un altro problema è che il software consegnato potrebbe essere disallineato rispetto alle richieste fatte dal committente.
- **Qualità del codice**: il software potrebbe anche avere una scarsa qualità, in termini di scrittura del codice (che pone il problema della manutenzione

successiva, oppure in termini di utilizzo (occupa troppa memoria, si blocca spesso, ecc.).
- **Poor design**: l'ultimo problema è che gli utenti finali potrebbero non essere pienamente soddisfatti dal software che è stato consegnato per via dell'usabilità, della velocità o della user-experience.

Le motivazioni per cui queste problematiche insorgono sono legate alla difficoltà intrinseca della complessità del problema, come appunto la stima dei costi, ma anche dei problemi tecnologici come il linguaggio di programmazione da utilizzare, l'infrastruttura su cui girerà il software (cloud o altro).
Ci sono tuttavia ragioni tangibili che rendono difficile lo sviluppo del software. Una delle prime motivazioni è la comunicazione. Tipicamente ci sono dei problemi di comunicazioni tra il tecnico e il committente che detta i requisiti del software. Questo per via della discrepanza di conoscenze tra lo stakeholder e l'ingegnere del software incaricato della creazione del prodotto. Questo è il problema più importante che provoca l'inutilità del software per via dei requisiti raccolti in maniera incorretta e sommaria.
Un altro problema è il capire quali sono le vere esigenze dell'utente finale, che potrebbe non volere un'interfaccia raffinata, ma piuttosto un sistema che risolva i suoi problemi di produttività quotidiana.
Ci possono essere poi difficoltà nel comprendere qual è il dominio dell'organizzazione all'interno della quale l'ingegnere va ad operare.
Infine altri problemi possono essere legati all'organizzazione e alla gestione delle persone che lavorano su di esso come la difficoltà da parte di alcune risorse umane di lavorare in team.

Il termine **ingegneria del software** è stato coniato alla Nato Conference, l'attuale International Conference of Software Engineering (ICSE)[1], uno dei più grandi incontri sull'ingegneria del software. In queste conferenze vengono presentati nuovi strumenti e metodologie per la gestione del software. Negli ultimi anni, in questi incontri, è tema comune quello dell'intelligenza artificiale legata alla gestione e manutenzione del software.
Il termine ingegneria è legato alla volontà iniziale di strutturare la creazione del software come un processo produttivo ingegneristico ad esempio la creazione di un'automobile. L'idea era quella di tradurre ciò che era già noto in meccanica, edilizia ecc nel mondo dello sviluppo software. In questo modo vengono elencati tutti gli aspetti che sono legati allo sviluppo del software, dalla raccolta dei requisiti, all'architettura ad alto livello, alla produzione vera e propria del software.
Se nelle progettazione di automobili si guardano tutti i dettagli, anche in questo caso si devono guardare i dettagli di sviluppo per evitare possibili futuri errori.
Quindi l'obiettivo dell'ingegneria del software è quello di ottenere un software che abbia la qualità richiesta, che rispetti le scadenze ed i vincoli di costo prefissati.
Ovviamente questo impone di dover raggiungere dei compromessi ad esempio per essere in linea con tempi e costi si scenderà a compromessi su qualità e performance.
I professionisti attivi nell'ambito dell'ingegneria del software devono essere in grado di produrre sistemi affidabili e funzionanti in maniera rapida. Queste due cose sono

[1] http://www.icse-conferences.org

contrastanti, infatti più brevi sono le tempistiche di produzione e più bassi sono i costi, più il codice sviluppato sarà difettoso e di bassa qualità. L'importante sempre e comunque è soddisfare le richieste dal committente.
Ci sono varie metodologie di sviluppo per riuscire, con i giusti compromessi, a produrre un software stabile scegliendo di volta in volta quella giusta in base a ciò di cui si necessita.
Una volta raccolti i requisiti si deve progettare il software e discutere il progetto con il team per assicurarsi che la conoscenza sia condivisa.
Come professionisti dell'ingegneria del software si viene a contatto anche con gli aspetti etici di questa professione. Ad esempio, ci si aspetta che l'ingegneria del software abbia un comportamento etico e responsabile. Prima di tutto la confidenzialità, anche senza che venga richiesta esplicitamente con la stipulazione di un accordo di riservatezza. Come ingegneri del software non si devono ventilare competenze che non si possiedono, si è tenuti a rispettare le leggi della proprietà intellettuale del software e si è tenuti a non abusare delle risorse informatiche alle quali si ha accesso. Ci sono associazioni internazionali che raccolgono vari ingegneri del software nel mondo.

2 Software Process

L'ingegneria del software vuole portare tutti i vantaggi che ci sono negli altri ambiti ingegneristici anche nello sviluppo software. Questi principi sono stati creati intorno agli anni 70-80 in cui chi scriveva software avrebbe dovuto creare un programma senza delle linee guida condivise. Anche nello sviluppo software si prende un prodotto grezzo e lo si lavora per ottenere un prodotto finito da consegnare.
E perché è particolare dotarsi di un processo software?
È utile per poter mettere ordine e pianificare in anticipo le varie fasi dello sviluppo così da avere un'idea chiara di degli obiettivi raggiunti e di quelli da raggiungere.
L'obiettivo del processo software è quello di migliorare la produttività dei dipendenti (evitando sprechi di tempo) e soprattutto controllare la qualità del prodotto (sapere in ogni momento se quello che ho davanti è un artefatto finale o parziale).
Solitamente più il processo software è di qualità, più sarà di qualità il prodotto finale. Per processo software si intende una serie di attività condotte e finalizzate allo sviluppo del prodotto finale.
Ci sono differenti processi, ma tutti hanno in comune alcune attività.

- **Specifica**: definire come dovrebbe funzionare il sistema
- **Progettazione e implementazione**: definire l'organizzazione e implementare il sistema
- **Validazione**: controllare che rispetti i requisiti del cliente
- **Evoluzione**: cambiare il sistema in base ai cambiamenti del cliente

Il processo software è diviso in attività condotte per portare avanti lo sviluppo del software (es: creare il design dell'interfaccia utente, specificare il modello dei dati ecc...). All'interno del processo ci sono anche gli output del processo stesso (tipicamente sono gli output di un'attività che vengono prodotti anche come input per l'attività successiva), i ruoli per ogni persona che lavora al processo software,

le pre e post condizioni (le prime devono essere vere prima dell'inizio e le seconde dopo la fine).
La prima grande distinzione che facciamo tra i processi software:

- **Processi plan-driven**: hanno una struttura fissa nel tempo e sono rigidi. Tutte le attività sono pianificate in anticipo prima di iniziare a sviluppare
- **Processi AGILE**: hanno una struttura flessibile e sono adatti al cambiamento dei requisiti in corso d'opera. La pianificazione è fatta a grandi linee e viene precisata nel dettaglio man mano che il prodotto sta venendo costruito. I processi AGILE sono nati perché spesso i requisiti da implementare non sono molto chiari all'inizio ma diventano man mano più chiari in corso d'opera
- **Processi ibridi**: molti processi pratici prendono spunto da entrambe i modelli a seconda delle esigenze

Il sistema **Code and Fix**, che non è definibile modello, è un processo abbastanza semplice.
Per prima cosa è necessario capire le necessità che devono essere implementate. Dopodiché si inizia l'implementazione di ciò che è stato richiesto. Ci sono delle condizioni da superare, dei testi. Se questi non passassero dovremmo fare delle azioni di fix. Quando i test passano, allora si può consegnare il programma finale. Il programma viene scritto per interazioni successive ed è consigliabile solo per progetti con meno di 2000 righe di codice e per team sotto i 5 dipendenti. Di fatto questo non è un processo software. Alcuni modelli di sviluppo software invece sono quelli visti nelle sottosezioni successive.

2.1 Waterfall model

Il modello **Waterfall** è uno dei primi processi proposti in risposta allo sviluppo del software artigianale e deriva dal processo di lavorazione manufatturiero. Negli anni 70 è stato il processo software più utilizzato poiché utilizzato per mettere ordine e capire quali fossero le attività da condurre per produrre software. Il suo obiettivo è quello di trasformare una produzione artigianale di software in una produzione industriale.
La caratteristica di questo processo è che il è scomposto in fasi e gli output di una fase sono gli input della fase successiva. Quindi la fase successiva può partire solo se quella precedente è stata completata.
La prima fase è quella della definizione dei requisiti. In questa fase si parla con i clienti o con gli stakeholders per capire quali siano le caratteristiche da implementare. Questa fase è importantissima perché gli errori commessi durante questa fase si protraggono per tutto lo sviluppo poiché sono diretta conseguenza di un'errata interpretazione dei requisiti. Tipicamente gli utenti non hanno la comprensione completa dei loro problemi perché spesso non li conoscono tutti. Una sola intervista, di solito, non è sufficiente per avere una visione completa.
La seconda fase è quella in cui si progetta il sistema software, cioè si cerca di definire le linee guida ad alto livello e l'architettura che può essere adottata. Questa parte può iniziare solo quando i requisiti sono pronti, vengono scritti e sono immutabili. Attraverso i requisiti sono attribuite le varie responsabilità ai progetti software.

La terza fase è quella dell'implementazione. Una volta definito il progetto si comincia a implementare il progetto. Vengono scritti anche dei test (unit test) per verificare se i singoli componenti sono corretti o meno.
Quando l'implementazione di tutti i componenti è completa e testata, si può passare alla fase di integrazione in cui tutti i componenti vengono uniti attraverso interazioni. Da questo momento è altresì possibile testare il sistema nella sua interezza e verificare se tutti i componenti comunicano in maniera corretta tra di loro. Una volta che il sistema è stato testato nella sua interezza è possibile consegnarlo al committente. Tuttavia il lavoro non è concluso ma si passa alla fase di manutenzione e aggiornamento del software.
L'attività di manutenzione può essere correttiva, di miglioramento (per aumentare velocità o prestazioni) e per aggiunte (da fare quando eventualmente sorgono nuovi requisiti).
La peculiarità di questo modello è che le fasi sono fisicamente separate nel tempo e ogni fase deve essere completata prima di passare alla fase successiva. Questa tipologia di processo è indicata per progetti che prevedono anche una componente hardware, per lo sviluppo di sistemi safety-critical (software degli aerei o dei treni) e quando il progetto è portato avanti da diverse aziende che lavorano ognuna a una parte diversa del software.
Non è invece indicato per quei progetti in cui le fasi hanno un feedback verso una fase precedente. Tipicamente è difficile anche adottarlo laddove i requisiti non siano esattamente chiari all'inizio.
Il vantaggio è di dare una grandissima importanza ai requisiti del software poiché si ritarda l'implementazione fino a quando i requisiti non sono chiari e molto ben definiti.
Gli svantaggi principali di questa tipologia di sviluppo sono che una fase deve essere completata prima di potersi spostare alla fase successiva, perciò qualsiasi cambiamento è qualcosa di molto costoso. Se, ad esempio, un committente cambia un requisito, si deve tornare indietro attraverso le fasi già concluse.
È bene dunque utilizzare questo processo di sviluppo in ambienti in cui i requisiti sono chiari e il contesto non è dinamico. Di solito questo modello è adottato per commesse governative in cui i requisiti sono messi a contratto.

2.2 Processo incrementale

Se nel processo a cascata le varie fasi sono chiaramente suddivise nel tempo e solo alla fine si arriva ad avere qualcosa di eseguibile, con il processo incrementale si cerca di arrivare quanto prima ad avere una versione eseguibile del sistema anche se questa avrà solo un sottoinsieme di funzionalità. In questo modo è possibile fornire una demo con un sottoinsieme delle funzionalità e man mano aggiungere le restanti. Il committente potrà avere aggiornamenti testabili sullo sviluppo con il vantaggio che potrà subito dare un feedback con conseguente modifica graduale funzionalità per funzionalità.
Questo approccio è usato solitamente quando in un progetto i requisiti non sono chiari dall'inizio e potrebbero cambiare ed evolvere durante lo sviluppo del progetto. Questo tipo di processo permette di accomodare anche i cambiamenti dei requisiti

perché di fatto l'attività di analisi dei requisiti perdura durante tutto lo sviluppo.
Questo tipo di sviluppo ha un costo molto più basso per rispondere ai cambiamenti, sia dei requisiti che dell'implementazione; inoltre la documentazione è più concisa ed è molto più facile avere feedback dal committente e dall'utente finale che non devono immaginare il prodotto leggendo documenti ma provarlo fisicamente con la demo.
Questo tipo di approccio ha anche degli svantaggi tra cui il management aziendale. Con il processo a cascata è tutto ben chiaro e si riesce a tracciare meglio lo stato di avanzamento del progetto. Con questo tipo di approccio invece i manager perdono un po' il controllo perché fanno fatica a capire a che punto del progetto si è arrivati. È anche più difficile capire se si è in anticipo, giusti o in ritardo nei tempi ed è complicato mantenere una documentazione chiara poiché non è scritta ed aggiornata rigorosamente come nel processo a cascata. Spesso infatti bisogna riscrivere parti di codice con la conseguenza che la documentazione non è quasi mai allineata.
Un altro svantaggio è legato al fatto che si ammetta la possibilità di apportare cambiamenti in corso d'opera necessari ad integrare altre parti di codice che vengono aggiunte. È quindi facile che il codice sorgente degradi molto più facilmente se non vengono intraprese delle azioni per ripristinare la qualità del codice. La qualità può degradare talmente tanto che l'aggiunta di una nuova funzionalità potrebbe essere molto difficile e pertanto richiedere molto tempo.

2.3 Processo di integrazione e configurazione

Questo processo prevede che lo sviluppo del software si avvalga di codice di terze parti che deve solo essere riconfigurato per far funzionare il sistema. È un approccio basato sul riuso di codice già sviluppato (bagaglio aziendale) e sull'utilizzo di codice terzo già presente sul mercato.
I componenti utilizzati hanno la necessità di essere configurati per rispondere al problema specifico oppure devono essere modificati perché rispondono solo parzialmente al bisogno.
Tipicamente si inizia anche in questo caso dalla specifica dei requisiti perché i problemi da risolvere guidano la scelta dei componenti da utilizzare.
Dopodiché inizia una fase di indagine alla ricerca dei componenti già disponibili sul mercato. Si possono trovare diversi componenti per una stessa funzionalità e va quindi selezionato quello che più si addice al caso specifico. Terminata questa fase è necessario passare ad una fase di raffinazione dei requisiti nella quale si valutano i componenti trovati anche in base alle pre-condizioni che gli stessi richiedono. Se si riesce a raggiungere un compromesso allora si può proseguire con tali componenti altrimenti dovranno essere adottate delle strategie alternative.
Il vantaggio principale di questo metodo è il basso costo perché si massimizza il riutilizzo ma unito a questo c'è anche un basso rischio perché i componenti utilizzati sono per la maggior parte già testati e funzionanti.
Questo permette di ottenere una consegna veloce dell'artefatto finale.
Solitamente però la qualità è inferiore a software prodotti completamente con codice interno poiché i componenti vengono adattati e non nascono con il preciso scopo per cui vengono in realtà utilizzati.

È anche possibile che ci si trovi nella situazione di scendere a compromessi con i requisiti perché non è detto che i componenti necessari siano già stati sviluppati da terzi e che rispondano al 100% alle richieste del committente. Lo svantaggio principale è che si perde il controllo sull'evoluzione del software perché di fatto il lavoro consiste più sull'integrazione dei componenti già esistenti che non sullo sviluppo.
Potrebbe inoltre succedere che la manutenzione di un componente utilizzato sia discontinua o che addirittura venga abbandonata.
Questo approccio è consigliabile in casi particolari come ad esempio una start-up nella quale si necessita di un prodotto in tempi brevi da poter mostrare ai potenziali investitori.

2.4 Attività comuni tra i modelli

Tutti e tre i processi hanno quattro attività/fasi in comune:

- Specifica
- Design e implementazione
- Validation
- Evolution

Considerando lo sviluppo prolungato del software, con un contratto di manutenzione del codice o di aggiornamento, queste fasi sono da considerare come cicliche e pertanto vanno ripetute spesso.

2.4.1 Specifica

L'obiettivo di questa fase, in qualsiasi modo essa venga eseguita, mira a capire quali siano le necessità e le funzionalità che il software deve avere. Questa è un'attività critica perché, se non si comprende a pieno il problema da risolvere, si finisce per risolverne uno diverso da quello richiesto.
La raccolta dei requisiti è dunque uno passo importante e da non sottovalutare.
Una volta raccolti i requisiti è bene fare uno studio di fattibilità cioè uno studio sulla fattibilità del progetto stesso sulla base dei dati a disposizione ed anche dell'esperienza pregressa.
Comprovata la fattibilità del progetto, quello che si ottiene è un documento dei requisiti in cui sono descritti in modo chiara le funzionalità del software ed il modo in cui esse andranno implementate.
La raccolta dei requisiti è portata avanti in tre fasi:

- Prima fase: analisi dei requisiti ed elaborazione di una descrizione del sistema da implementare. Può essere fatta sia prendendo come esempio il sistema attualmente in uso, sia tramite una discussione con gli utenti finali. Può contenere dei modelli, delle astrazioni, dei prototipi e tutto ciò che aiuta a capire quello che si dovrà implementare.
- Seconda fase: è la specifica dei requisiti, cioè si scrive nero su bianco quello che si è raccolto. Ci sono due macro-suddivisioni di requisiti:
    - user-requirements: più astratti e discorsivi

    - requisiti di sistema: più dettagliati e specifici delle funzionalità da implementare
- Terza fase: è quella della validazione dei requisiti. Si controlla cioè la consistenza, se ci sono delle contraddizioni o delle lacune che vanno colmate. È importante identificare eventuali errori prima di procedere.

2.4.2 Design e implementazione

Per design si intende la progettazione mentre per implementazione la scrittura del codice.
L'obiettivo di questa specifica è tradurre ciò che si è carpito nella fase di raccolta dei requisiti in un programma.
La fase di design consiste nella creazione di un modello in modo da avere presente come comunicano i componenti tra di loro, tra i sistemi, l'organizzazione dei dati, il database, le API. Una volta definita la struttura, bisogna passare alla fase di implementazione cioè allo sviluppo vero e propria.
Ovviamente il design e l'implementazione cambiano in base al tipo di sistema che si decide di implementare.
Per prima cosa bisogna decidere l'architettura del sistema cioè la struttura ad alto livello, i componenti e come essi comunicano tra loro, dopodiché si progetta il design del database per capire quali strutture dati principali verranno utilizzate. Infine vanno definite le interfacce dei componenti e come esse comunicano tra di loro.
Senza una chiara interfaccia definita prima, quando si dovranno integrare i componenti tra di loro, dovranno essere apportate delle modifiche e delle integrazioni e non sempre questo è facile.
Fatto ciò è possibile definire i dettagli del design dei singoli componenti e i dettagli per l'implementazione.

2.4.3 Validation

La validazione è divisibile in due macro-tipologie; la prima è nota come **check** ed è di fatto una verifica del codice. Il verificatore legge il codice per cercare eventuali problemi o errori all'interno dello stesso. Questo sistema è utilizzato anche per vedere se sono presenti vulnerabilità all'interno del codice.
La seconda è definita **program-testing** cioè la creazione di scenari di test per comparare i risultati che il componente elabora con le assunzioni fatte a priori. Il testing viene condotto in vari momenti:

- Component o unit testing: è condotta durante lo sviluppo del software. Ogni componente è testato come qualcosa di a sé stante
- System-testing: prevede di integrare tutti i componenti in un sistema finale per testare le interazioni tra i componenti
- Acceptance testing: è il test che viene condotto quando il sistema software è nelle condizioni operative finali. Si controlla come il sistema si comporta in produzione poco prima di rilasciarlo al cliente

2.4.4 Evolution

Questa attività si concentra sul garantire al cliente la correzione di eventuali bug, ma anche l'evoluzione dovute a nuove richieste.
Il processo di evoluzione prevede un controllo del sistema in esecuzione che in alcuni casi è di un'azienda terza, la proposizione di nuove modifiche e la loro attuazione.
La nuova versione rilasciata con le modifiche apportate diventerà così la versione corrente ed in esecuzione.

3 Agile software development

Negli anni 80-90 quando sono nate le tecniche di gestione di un processo software, l'obiettivo era quello di sviluppare un software di alta qualità, quindi è stata prestata attenzione ad una pianificazione dettagliata con controlli rigorosi sia alla parte di design, sia alla parte di avanzamento. Inoltre, sono anche stati creati dei tool per visualizzare graficamente lo stato di avanzamento del processo.
Veniva utilizzato un approccio plan-driven, il più appropriato per il software del tempo in quanto commissionato da agenzie governative per il controllo di infrastrutture safety-critical.
Software di questo tipo richiedono una valutazione a lungo termine; ad esempio il software utilizzato negli aerei deve ipotizzare l'hardware presente tra una decina di anni in modo da poter continuare ad essere utilizzato dato che servono molti anni per scrivere software di questo tipo.

Alla fine degli anni 90 il mercato è evoluto passando dalla quasi totalità di commesse governative a richieste da parte di aziende in cerca di software a supporto del loro business.
Questo tipo di software ha caratteristiche molto diverse rispetto alle commesse governative. Per questo motivo si è resa necessaria una nuova metodologia di sviluppo che prevedesse delle consegne durante la fase di costruzione e non solo alla fine del ciclo di sviluppo.
Il costo per l'approccio a piani su progetti di piccola media grandezza era inoltre troppo alto ed il modello adottato rende difficile rispondere velocemente al cambiamento dei requisiti.

Sono emerse così delle metodologie di sviluppo software pensate per rispondere a queste esigenze. La nuova metodologia è stata definita **AGILE** poiché è snella e pronta al cambiamento.
Nello sviluppo agile le varie fasi proseguono contemporaneamente e terminano solo nel momento in cui il progetto software viene concluso. Perciò le fasi di raccolta di requisiti, valutazione e comprensione di essi viene svolta in parallelo alle altre e non prima dello sviluppo.
Il sistema è sviluppato con una metodologia basata su incrementi successivi. Si stabilisce un ritmo all'inizio e lo si mantiene durante tutto lo sviluppo. Il ritmo tra i vari tempi di sviluppo intermedi può variare dalle 2 alle 4 settimane ed allo scadere del tempo prestabilito viene rilasciato un prodotto (una nuova versione, funzionalità, aggiornamento...).
Le fasi vengono sviluppate anche con l'aiuto dello stakeholder poiché servono dei

feedback sul lavoro svolto che eventualmente comporti il cambio dei requisiti.
La documentazione è ridotta al minimo e viene promossa una comunicazione più informale tra i partecipanti al progetto, riducendo le noiose riunioni formali degli anni 80.
Il terzo fattore importante di questo modello di sviluppo è l'utilizzo di strumenti automatizzati per alleggerire il carico dello sviluppatore da azioni manuali e noiose che potrebbero portare ad uno spreco di tempo (es. sistemi automatici di test, sistemi che creano l'interfaccia grafica, sistemi di continuos integration...). Nel manifesto che definisce le linee guida della metodologia agile sono presenti quattro punti di rilievo:

- Le persone e la loro interazione rispetto al processo di sviluppo ed agli strumenti
- Meglio un software funzionante che una documentazione esaustiva
- Collaborazione con il committente
- Rispondere ai cambiamenti in modo rapido e veloce (agile)

Ed alcuni principi fondamentali:

- **Coinvolgimento del cliente**: permette di comprendere i requisiti in modo accurato e corretto oltre che valutare quelli che sono i rilasci intermedi
- **Consegna incrementale**: ad ogni step il software può essere installato ed utilizzato dal cliente
- **Meglio le persone che i processi**: le skills dei programmatori sono molto importanti. Essi vengono spesso lasciati liberi di adottare le proprie preferenze di scrittura del codice
- **Cambiamento dei requisiti**: è importante che l'architettura sia abbastanza agile da riuscire a cambiare i requisiti e l'implementazione in corso d'opera
- **Mantenere la semplicità**: perché le cose semplici sono quelle che solitamente creano meno problemi

Le aziende che adottano le tecniche agili sviluppano software di piccole o medie dimensioni. Perché queste tecniche possano essere implementate, deve esserci la volontà del cliente di far parte del team di sviluppo. Senza questo il progetto deraglierà.
È importante anche che ci siano poche entità esterne che pongano dei vincoli. Ad esempio, per il software medicale lo sviluppo agile non funziona.
Lo sviluppo agile è tipicamente condotto insieme a tecniche agili di project-management come lo **SCRUM**. Lo SCRUM è una metodologia di gestione del processo che può essere adattata anche a progetti diversi da quello software.
Dopo la loro definizione, i processi agili hanno avuto una radicalizzazione che ha condotto a quella che viene definita programmazione estrema, dove le caratteristiche dell'agile vengono portate all'estremo. Ci sono varie versioni del software rilasciate ogni giorno, tutti i test devono ritornare esito positivo perché il software possa essere compilato e consegnato e c'è una continua integrazione tra i componenti del software anche quotidianamente. Altre caratteristiche della programmazione estrema sono il design molto semplice, un tipo di sviluppo test-first ed il pair programming.
Le **user-story**, descrizioni di scenari nei quali il software una volta sviluppato sarà

integrato, sono delle brevi narrazioni meglio comprensibili sia dall'utente finale che dallo sviluppatore.
Sono scritte dal punto di vista dell'utente che racconta ciò che vorrà fare con il suo prodotto.
Ogni user-story è messa in quella che viene definita **user-card** cioè una breve storia nella quale sono integrati i vari bisogni degli utenti. A partire dalle user-story vanno sviluppate delle **task** che spezzano la user-story in diverse feature da implementare all'interno del software. Questo lavoro è fatto dal team di sviluppo supportato dallo stakeholder che diventa parte del team di sviluppo stesso. Una volta definite le task si deve stimare l'effort necessario (es: ore o giorni) necessarie ad uno sviluppatore per risolvere una task.
È importante avere anche una valutazione dell'importanza delle task, per concentrarsi prima su quelle che hanno un punteggio più alto. Fatto ciò è si può pensare allo sviluppo.

3.1 Test-driven

Una delle modalità di sviluppo utilizzate è detta test-driven, sviluppo guidato dai test. In passato i test venivano scritti dopo il software, in questo contesto tuttavia si inverte la relazione temporale ed i test sono scritti prima. È così possibile utilizzarli durante lo sviluppo del codice per trovare gli errori già durante lo sviluppo. Fintanto che le funzionalità correnti non sono tutte corrette lo sviluppo non può proseguire per evitare situazioni di contrasto successive generate degli errori e dalla loro risoluzione. Posticipare la correzione significa faticare di più per trovarne la causa e risolverlo.
Il test incrementale va di pari passo con lo sviluppo incrementale di uno scenario. Il punto critico è che per scrivere un test si deve avere ben chiaro il requisito poiché il test specifica quali input servono e quali output devono essere generati. Scrivere i test permette di verificare se lo sviluppatore ha compreso o meno il requisito e quindi è molto probabile che l'implementazione del codice sarà corretta con meno ambiguità e problemi di comprensione.
Durante lo sviluppo test driven lo stakeholder va comunque coinvolto poiché sarà lui a specificare se i test stanno venendo eseguiti nei contesti giusti di operatività del software.
Adottare questa metodologia significa anche avere a disposizione degli strumenti automatizzati per l'esecuzione dei test in quanto gli scenari per i casi di test sono molti.
In caso di cambiamenti in corsa alcune funzionalità possono essere sviluppate come implementazioni di altre funzionalità, quindi sono necessari dei componenti per agevolarne l'implementazione.
Inevitabilmente, quando si sviluppa per incrementi successivi, la qualità del codice può andare scemando ed i cambiamenti che si dovranno affrontare in futuro saranno sempre più costosi e difficili da gestire. È importante dunque intraprendere delle attività di miglioramento del codice durante la produzione, cambiandone l'architettura e riportando la qualità ad un livello accettabile. Un'attività di questo tipo fatta a software completato può essere molto costa, mentre fatta volta per volta risulta

essere più facile ed economica.

3.2 Pair programming

Un altro concetto importante è il pair programming, cioè due sviluppatori che lavorano su un solo computer ad una stessa funzionalità. Nel manifesto agile i ruoli non sono ben definiti ed il lavoro è alternato tra i due programmatori considerando però che le coppie sono molto variabili e dinamiche.
Ci sono numerosi vantaggi nell'utilizzare questa tecnica. Il codice, o una parte di esso, non è di un solo sviluppatore, ma di più dipendenti e conseguentemente tutti hanno interesse che il codice venga scritto in maniera corretta e comprensibile. Un altro vantaggio è che il programmatore che non sta scrivendo può fornire un aiuto a chi sta scrivendo ed un punto di vista diverso che può aiutare a non commettere errori. Si promuove, inoltre, la condivisione della conoscenza che a sua volta ha numerosi vantaggi. Innanzitutto, se c'è da apportare una modifica possono farla più persone e se un membro del team dovesse cambiare, la sua conoscenza non andrebbe persa, ma rimarrebbe comunque all'interno del team attraverso gli altri componenti.

3.3 Agile progect management

L'agile project management può essere usato in vari contesti. Nel nostro caso ci si concentrerà sul contesto dello sviluppo del software. Ci sono alcune fasi che caratterizzano lo sviluppo tramite metodologia agile:

- **Fase iniziale**: si esegue il setup del progetto in cui si definiscono gli obiettivi generali e si abbozza l'architettura del software anche se a grandi linee
- **Fase centrale**: lo sviluppo avanza per cicli (sprint). Durante uno sprint si sviluppa un incremento che alla fine dello sprint andrà consegnato
- **Fase finale**: alla fine degli sprint c'è la conclusione del progetto. Durante questa fase si scrive la documentazione e viene controllato l'intero progetto.

Il **product backlog** è la "to-do list" che contiene tutte le funzionalità che dovranno essere implementate. Il backlog inizialmente viene riempito con le user-story, oppure con il sistema attualmente in uso ed il product-owner (committente o stakeholder), seleziona quali sono le funzionalità che vanno inserite nello sprint successivo. Questo è sufficiente per iniziare il primo sprint.
Per iniziare vengono selezionate le entry con priorità più alta. I task sono selezionati fintanto che non viene saturata la velocity del team. All'inizio la velocity sarà una misura imprecisa poiché è molto difficile da misurare; viene tuttavia raffinata man mano con l'avanzare degli sprint.
Ogni giorno in mattinata il team si riunisce per lo SCRUM e gli sviluppatori aggiornano il team sui progressi del giorno prima, sui task su cui lavoreranno durante la giornata e gli eventuali problemi che hanno riscontrato.
La durata dello sprint è fissa e nel caso in cui una funzionalità sia stata sottostimata e non sia stata conclusa viene rimossa dal backlog così da rispettare la scadenza (anche con un sottoinsieme di obiettivi).

Durante lo sprint, il team è isolato e tutte le comunicazioni sono affidate allo scrum master così da evitare di disturbare il team e permettergli di raggiungere gli obiettivi prefissati.
Ogni sviluppatore prende delle task dal backlog e le lavora poi, quando il task è completato, viene spostato nella colonna "completato" e lo sviluppatore può cominciare a lavorare su un altro task. Alla fine dello sprint si deve consegnare l'incremento. Nel farlo viene organizzata una sprint review durante la quale si pensa anche ad eventuali miglioramenti in vista degli sprint successivi. Dopodiché vengono inserite altre task nello sprint backlog ed il ciclo ricomincia. Durante l'ultimo meeting è presente anche il product owner che testerà il prodotto finale.

I benefici dello scrum sono molteplici tra cui il fatto che il prodotto è diviso in parti gestibili e comprensibili sia dal product owner che degli sviluppatori, eventuali requisiti che cambiano o su cui non sono state ancora prese delle decisioni non ritardano lo sviluppo, il team ha una conoscenza del sistema nella sua interezza e il cliente può fornire feedback sulle consegne di modo che lo sviluppo possa essere gestito con maggiore velocità.
Nel corso del tempo si è pensato di estendere la metodologia scrum anche ad aree per cui non era mai stata utilizzata come, ad esempio, i sistemi distribuiti.
La metodologia scrum è pensata per situazioni in cui tutti gli sviluppatori sono nella stessa sede fisica. Quando questo non succede è importante che il product owner faccia visita fisicamente al team di sviluppo. Se gli sviluppatori sono distribuiti geograficamente è comunque importante mantenere una comunicazione informale (es: video call o instant messaging) ed avere un sistema di integrazione continua così che tutti gli sviluppatori abbiano cognizione del sistema che sta stanno creando.
Alcune aziende hanno provato a capire se la metodologia agile potesse scalare così da essere utilizzata anche in progetti di grandi dimensioni con team composti da molte persone o in progetti di lunga durata (non solo mesi, ma anche anni).
Sono state così formulate due dimensioni di scala:

- Scala nelle dimensioni del software
- Scala nelle dimensione dell'organizzazione

Tutte le proposte formulate per risolvere il problema della scalabilità concordano sul fatto che va mantenuta la flessibilità, che i rilasci debbano essere molto frequenti e che la comunicazione sia importante.
Un problema tipico che può presentarsi quando si utilizza la tecnica agile è quello della manutenzione. Questa tecnica è infatti molto performante quando si tratta di sviluppo ma poco quando si tratta di mantenimento del software creato cosa che dovrebbe comunque essere supportata.
Sintetizzando si possono individuare due problemi fondamentali sul problema del mantenimento:

- Il software è difficile da mantenere perché non c'è una scrupolosa documentazione e se un membro del team originale dovesse lasciare il progetto porterebbe con sé varie conoscenze riguardo le decisioni prese durante lo scrum. Questo può capitare quando i progetti sono lunghi

- Il cliente dovrebbe essere comunque coinvolto nel processo di manutenzione e già è difficile convincerlo a partecipare durante la creazione del software

Da qui la domanda: meglio Agile o Plan-driven? Per rispondere vanno fatte delle considerazioni:

- È importante avere una specifica e un livello di dettaglio alto prima dell'implementazione?
- È realistico avere un approccio basato sui rilasci incrementali?
- Sono anche da valutare le dimensioni del sistema software. Se il sistema può essere sviluppato da un team abbastanza piccolo, si può scegliere una metodologia agile. Viceversa, se il team è molto grande e se si preferiscono comunicazioni formali è meglio un approccio plan-driven

È quindi importante capire il tipo di sistema che si è chiamati a sviluppare; per esempio se la vita attesa del sistema è lunga, cioè se un sistema deve funzionare per molti anni, l'approccio agile non è il migliore.
Bisogna anche capire quante inferenze esterne ci potrebbero essere sul prodotto software e conoscere il proprio team di dipendenti oltre al supporto tecnologico disponibile dato che il metodo agile funziona meglio laddove ci sono degli automatismi che risparmiano tempo ad azioni manuali.
È importante definire il livello di skill dei componenti del team così che essi vengono lasciati liberi di prendere alcune decisioni. Alcune problematiche potrebbero emergere nel caso in cui il team sia distribuito geograficamente.
In alcune organizzazioni si adottano dei processi di qualità che vincolano lo sviluppo perché si impone di motivare le decisioni prese e di scrivere i requisiti dettagliatamente prima di iniziare lo sviluppo. Questi vincoli vanno tenuti in considerazione per la scelta del metodo da adottare.
Infine otrebbero esserci problemi nell'introdurre la metodologia agile in aziende abituate al modello plan-driven.
La metodologia agile ha incontrato e incontra tutt'ora varie resistenze al momento della sua introduzione in azienda perché spesso i project manager non ne hanno una conoscenza approfondita e fanno fatica ad abbandonare qualcosa che conoscono bene per rischiare con un approccio innovativo.
Inoltre spesso le grandi organizzazioni hanno delle procedure interne per il controllo e la verifica della qualità che di fatto sono in contrasto con le metodologie agili.
In ultimo le metodologie agili prevedono la possibilità che i membri dei team, a seconda delle loro capacità, possano prendere delle decisioni. A questo si unisce anche l'attenzione alle resistenze di tipo culturale a livello dirigenziale nell'azienda del tipo "si è sempre fatto così e non c'è motivo di cambiare".

4 Requirement engineering

Una delle prime task che sono da condurre per creare il software è il requirement engineering. Per requirements/requisiti si intende la descrizione delle caratteristiche e dei servizi che il sistema dovrà fornire agli utenti finali durante la sua esecuzione. Ognuno dei requisiti risponde ad uno o più problemi degli utenti finali.
I requisiti si possono scomporre attraverso una macro-suddivisione iniziale:

- **Requisiti utente**: sono delle frasi in linguaggio naturale che descrivono le necessità che l'utente ha e che si aspetta che il sistema risolva. Sono abbastanza astratte perché definiscono a grandi linee ciò che l'utente vorrebbe vedere
- **Requisiti di sistema**: hanno una struttura un po' più dettagliata, non sono in linguaggio libero. Di fatto non definiscono i bisogni dell'utente, ma il modo in cui gli addetti ai lavori comprendono i bisogni dell'utente. È necessario tradurre i requisiti utente in requisiti di sistema per poter essere usati in una progettazione e per validare se ciò che è stato chiesto è stato realizzato. Nei requisiti di sistema noi dobbiamo specificare bene quello che è stato messo nei requisiti utente. Solitamente il requisito da parte degli utenti è molto basico e generico, mentre i requisiti di sistema sono dettagliati e quanto più improntati all'implementazione

Gli **Stakeholders** sono coloro che hanno un qualche interesse per il sistema, che usano il sistema o le cui attività sono influenzate dal sistema. Lo sono gli utenti finali, i manager, i regolatori esterni.

Un'altra suddivisione macroscopica per i requisiti è quella tra requisiti funzionali e requisiti non funzionali.

4.1 Requisiti Funzionali

Sono quei requisiti che descrivono le funzionalità che il sistema deve avere. Quindi c'è un mapping 1:1 tra requisito funzionale e funzionalità. Ad esempio, un requisito funzionale riguarda come il sistema dovrebbe rispondere a un determinato input o come il sistema si dovrebbe comportare in determinate situazioni.
Ci sono diversi livelli di astrazione sui requisiti non funzionali; possono essere molto generici o un po' più specifici. Essi possono anche essere ambigui e prestarsi a più interpretazioni.
L'obiettivo dei requisiti funzionali è quello di essere completi, cioè tutte le funzionalità presenti all'interno del sistema devono essere catturate nei requisiti, inoltre devono essere consistenti. Di fatto, però, quello che succede è che gli stakeholder hanno necessità differenti e spesso sono già inconsistenti le richieste iniziali quindi i requisiti potrebbero essere imprecisi. Alcune inconsistenze possono essere già identificate nella fase di raccolta dei requisiti, altre potrebbero essere più subdole e comparire solo al momento dell'implementazione.

4.2 Requisiti non funzionali

Sono dei vincoli sul sistema, dei vincoli sui servizi offerti dal sistema nella sua interezza, quindi sono delle proprietà del sistema. Per esempio, potrebbero esserci dei vincoli sui tempi di risposta, sul processo che dobbiamo adottare per sviluppare il sistema (un particolare linguaggio di programmazione), standard che devono essere utilizzati perché sono prescritti dalla normativa o dal committente.
Il punto fondamentale è che questi sono vincoli sul sistema nella sua interezza, quindi i vincoli non funzionali potrebbero a loro volta porre dei vincoli sull'architettura globale del sistema.

Un requisito non funzionale potrebbe generare vari requisiti funzionali. Alcuni esempi di requisiti non funzionali.

- **Requisiti di prodotto**: potrebbero esserci dei requisiti di usabilità anche per persone ipovedenti o daltoniche. Potremmo volere una maggiore affidabilità o dei tempi di risposta molto brevi
- **Requisiti di sicurezza**: chi può accedere al sistema, con quali permessi, quali sono i metodi di autenticazione
- **Requisiti di organizzazione**: a seconda del tipo di cliente o di sviluppatore potrebbero esserci dei vincoli sul processo produttivo. Ad esempio, vincolo sul linguaggio di programmazione o sugli ambienti sul quale il software deve girare
- **Requisiti esterni**: provengono dal mondo esterno come entità governative che regolano un particolare dominio. Oppure ci sono requisiti di natura etica o normativi (es. su software avionico o ferroviario)

È importante che i requisiti non funzionali siano quantitativi e verificabili; se così non fosse sarebbe difficile verificare se un requisito è stato soddisfatto o meno.
Le misurazioni citate nei requisiti non funzionali possono essere la velocità del sistema (numero di transazioni al secondo, refresh dello screen), la dimensione, l'usabilità (può essere quantificata in numero di ore utilizzate per la formazione), l'affidabilità (tempo medio tra un fallimento e il successivo; probabilità che il sistema non sia raggiungibile), la robustezza o la portabilità.

4.3 Raccolta dei requisiti

Il processo di cui dobbiamo dotarci per raggiungere questi obiettivi. Tipicamente non è semplice comprendere i requisiti. Per minimizzare il numero di errori, serve una metodologia. Tanto più è seguito il metodo scelto, quanto più si ottiene un approccio ingegneristico. La raccolta dei requisiti si articola in tre macro-passi: elicitazione e analisi dei requisiti, specifica dei requisiti e validazione. Queste tre fasi non sono lineari, assomigliano più a passi di raffinamento successivo.

4.3.1 Elicitazione dei requisiti

L'obiettivo è capire cosa gli utenti devono fare con il sistema, quali sono le funzionalità che il sistema dovrà fornire, come gli utenti interagiranno col sistema e come essi lavorano.
In questa fase, stakeholders e ingegneri del software devono collaborare per capire qual è il dominio all'interno del quale si sta lavorando (es. medico, bancario...). Devono comprendere le funzionalità che il sistema deve fornire, se ci sono dei vincoli sulle performance o sull'hardware o dei vincoli tecnici che vanno presi in considerazione quando passeranno alla fase di implementazione del sistema.
Questa prima fase non è banale e ci sono ostacoli che la rendono piuttosto complicata e prona ad errori. Gli stakeholders non conoscono esattamente quello che vogliono e per questo motivo i requisiti raccolti con un'intervista sono parziali o parzialmente corretti. Si deve quindi lavorare assieme a loto per sviscerare correttamente il problema senza assumere che lo stakeholder lo conosca esattamente.

Gli stakeholders parlano dei requisiti con una terminologia propria e quindi le parole utilizzate potrebbero nascondere delle insidie e della conoscenza implicita. Inoltre, se si parla con diversi stakeholder, potrebbero emergere dei requisiti in conflitto per via di punti di vista divergenti.
Non sono da sottovalutare inoltre i fattori "politici", ad esempio alcuni manager potrebbero voler aggiungere dei requisiti che li aiutino dal punto di vista politico (cioè maggior controllo o maggior potere su decisioni che possono essere prese all'interno dell'azienda).
Infine rimane da considerare che i requisiti potrebbero mutare durante l'analisi dei requisiti.
Una delle modalità più utilizzate per raccogliere i requisiti è quello dell'intervista; si parla con gli stakeholder.
Le interviste possono essere a domande chiuse, oppure l'intervista può essere aperta senza domande ma con una traccia di argomenti dei quali l'ingegnere del software necessita di discutere. L'intervista aperta è più difficile da condurre, ma le informazioni raccolte sono spesso più ricche. Tipicamente, quello che succede è un mix tra le due tipologie di interviste.
Le interviste però vanno condotte ponendo attenzione a non inserire nelle domande il suggerimento alla risposta, influenzando così lo stakeholder.
Durante le interviste gli stakeholder descrivono quali sono i problemi da risolvere e, a meno che l'intervistatore non sia in grado di fornire un primo prototipo anche su carta, non è facile condurre interviste efficaci. Lo stakeholder tende ad utilizzare un gergo ed una terminologia tecnica ma potrebbe far fatica a trasmettere alcuni concetti all'intervistatore oppure potrebbe non essere disposto a parlare ed esprimersi durante un'intervista.

Ci sono tuttavia delle metodologie alternative all'intervista. Una di queste è nota come **studio etnografico** cioè considerare che le risposte delle persone dipendono dalla conoscenza che hanno.
Gli stakeholder parlano del proprio lavoro, ma magari non conoscono le relazioni con il resto delle altre persone quindi descrivono ciò che sarebbe loro utile nel sistema, ma non hanno idea di quale sia l'operatività degli altri.
Questo metodo ha come obiettivo, oltre a quello di carpire i requisiti, anche di capire quali sono le relazioni tra gli intervistati ed i futuri utenti del sistema.
In questo metodo, l'ingegnere del software si immedesima nel ruolo di lavoratore all'interno dell'azienda e cerca di comprendere meglio il processo lavorativo dei futuri utenti.
Questo sistema porta a evitare le domande dando spazio all'osservazione degli utenti durante l'operatività quotidiana.
Uno studio di questo tipo è molto potente ma anche molto costoso ma possono essere condotti solo quando lo scopo è sostituire il software in uso con uno nuovo.
Una volta che raccolte le necessità, per rappresentare questi requisiti si possono usare quelle storie e scenari. Sono un esempio di come il sistema dovrebbe funzionare ed è facile, a partire da queste, ottenere un feedback. La loro concretezza aiuta l'ingegnere del software a parlare con lo stakeholder.
La storia è scritta in forma narrativa e racconta di come il sistema dovrebbe essere

utilizzato.
Lo scenario invece è un po' più strutturato poiché fornisce delle informazioni specifiche (input/output).
Da una storia si ricava un numero superiore di scenari poiché la storia è generica e lo scenario specifico.

4.3.2 Specifica dei requisiti

Una volta raccolte le necessità degli utenti è necessario descriverle in un documento dedicato.
A volte la specifica dei requisiti è parte del contratto e quindi la fase di raccolta dei requisiti non rientra nel conto della commessa del software perché condotta solo per presentare un preventivo.
Nel secondo caso è da considerare come un investimento per competere all'assegnazione di una commessa. È quindi importante che i requisiti raccolti siano il più completi possibile. In linea di principio i requisiti dovrebbero dire "cosa" il sistema dovrà fare e non "come" il sistema lo dovrà fare. Questa divisione netta in teoria non lo è nella pratica poiché il tutto dipende anche dal sistema che si pensa di implementare.
La specifica dei requisiti è fatta in linguaggio naturale in quanto è comprensibile da tutti e molto espressiva. Ciò però può rendere il requisito interpretabile in modo ambiguo. Ci sono quindi delle linee guida per limitare il problema dell'ambiguità:

- È importante adottare un formato standard da definire all'inizio e da utilizzare fino alla fine
- Il linguaggio dovrebbe essere consistente cioè le forme verbali dovrebbero essere utilizzate in modo consistente lungo tutto il documento (es. Shall viene usato per qualcosa di obbligatorio – Should per qualcosa che non è del tutto obbligatorio)
- È utile sottolineare o evidenziare le parti chiave del progetto
- Sarebbe importante anche evitare di utilizzare terminologia specifica informatica in favore di parole del dominio in cui andiamo ad operare
- È bene anche aggiungere delle motivazioni per spiegare perché il requisito è importante. Questo torna utile nel caso in cui si debba ripensare al requisito in modo da controllare se lo si è compreso

Ai requisiti va data una struttura predeterminata e non troppo libera per diminuire i gradi di libertà che l'ingegnere ha nello scriverli.
Una trattazione molto strutturata ha senso nei domini applicativi in cui si ha bisogno di uno sforzo intenso per l'analisi, come ad esempio avviene nei sistemi medicali.
La tecnica da adottare per modellare i requisiti dipende anche dal progetto che si deve affrontare.
È da considerare anche lo **use case** che definisce come il sistema si relaziona con gli attori. Uno use case è uno scenario ed identifica gli attori e le loro interazioni. Si possono anche utilizzare degli **sequence diagram** per specificare ogni singolo use case.
L'obiettivo finale della raccolta dei requisiti è quello di scrivere il documento dei requisiti che espone in modo definito quelli che sono tutti i requisiti che devono essere

implementati.
La scrittura di questo documento è variabile e dipende dal sistema software che si deve implementare; se si utilizza un sistema guidato da piani il documento sarà molto dettagliato; in caso di un sistema a sviluppo incrementale il documento sarà meno strutturato e più agile con un minore livello di dettaglio.
Alcuni organismi hanno cercato di definire degli standard per questa tipologia di documenti come Institute of Electrical and Electronics Engineers (IEEE), ma tipicamente sono applicabili a processi di grandi dimensioni e guidati dal modello a piani.
Questo documento può essere utilizzato sia dagli utenti finali, dai manager che lo utilizzano per una stima del costo che il progetto avrà, ma anche dagli ingegneri del sistema e dagli ingegneri dei testi. Tornerà utile inoltre a coloro che dovranno fare la manutenzione del sistema.

4.3.3 Validazione dei requisiti

L'obiettivo di questo processo è cercare di trovare tutti i potenziali problemi, errori e inconsistenze. Correggere l'errore a questo punto richiede parecchio sforzo ma il costo per correggerlo più avanti sarebbe ancora maggiore. Per controllare la presenza di errori si può utilizzare una sequenza di controlli di questo tipo:

- **Validità**: controllare se le funzioni che sono state definite supportano nel miglior modo i bisogni del cliente
- **Consistenza**: controllare se ci sono delle inconsistenze o dei conflitti tra i requisiti
- **Completezza**: controllare se tutte le funzionalità richieste dal cliente, sono prese in considerazione
- **Realismo**: controllare se è tutto realizzabile con le tecnologie attuali
- **Verificabilità**: controllare se tutti i requisiti così come scritti sono verificabili

Per controllare la validazione dei requisiti invece si possono seguire varie modalità:

- **Controllo manuale**: un gruppo di persone esegue i controlli leggendo il documento e controllandone la consistenza
- **Tramite un prototipo**: per verificare se si è capito correttamente quello che è stato richiesto dal cliente
- **Tramite scenari di test**: scrivere degli scenari di test ancor prima di creare il sistema. Questo impone di pensare a delle condizioni che vanno verificare e a controllare se tutte le casistiche sono state prese in considerazione. Se scrivere questi scenari di test risulta difficile, significa che il requisito non ha catturato bene ciò che il sistema dovrebbe supportare

4.3.4 Cambiamento dei requisiti

A meno di aver scelto un processo che congela i requisiti per l'intera durata dello sviluppo, essi sono soggetti a cambiamento. Per questo motivo si deve trovare un metodo gestire richieste di cambiamento.
Per software di grandi dimensioni è da aspettarsi che i requisiti cambino costantemente. Tipicamente il motivo per il quale i requisiti cambiano sono la difficoltà

nel definire bene e per sempre tutti i requisiti, perciò man mano che si procede e si conosce meglio il problema viene più facile dettagliare i requisiti in modo migliore. Altri motivi sono ad esempio modifiche all'hardware o richieste di integrazione con altri sistemi software del cliente oppure per via di nuove normative.
I requisiti potrebbero anche dover essere modificati per via della differenza tra chi compra il sistema e chi lo utilizza alla fine.

Con il termine **requirement management** si intende il processo adottato per apportare le modifiche. Innanzitutto è importante identificare i requisiti e per farlo ogni requisito deve essere associato a un numero univoco. Dopodiché si definiscono le attività da condurre per accettare un cambiamento e propagarlo all'interno del documento dei requisiti. Se infatti viene modificato un requisito collegato ad altri è necessario propagare il cambiamento a tutti i requisiti ad esso collegati.
Esistono degli strumenti con lo scopo di facilitare il tenere traccia dei collegamenti tra requisiti.
In ingegneria del software è necessario avere un corretto supporto allo sviluppo e dotarsi di strumenti che automatizzano delle operazioni che potrebbero essere fonte di errore.

5 UML

Così come i requisiti anche il design deve essere documentato. Lo standard Unified Modeling Language (UML) è nato appunto con questo obiettivo.
Negli anni 80 è nata l'esigenza di avere un linguaggio per la modellazione ma il tentativo di creare un linguaggio condiviso da tutti è stata fallimentare.
L'UML può essere utilizzato in vari contesti, ad esempio per abbozzare soluzioni, ma anche per fornire il dettaglio implementativo di un sistema.
Ci sono varie tipologie di diagrammi UML, non tutti vengono utilizzati correntemente, ma è importante conoscerli. I diagrammi si possono dividere in tre categorie:

- **Modelli strutturali**: descrivono l'organizzazione e le relazioni tra le varie parti del sistema. Ci sono delle strutture più statiche e dinamiche che descrivono lo stato a run-time. Il class diagram è sicuramente quello più utilizzato e descrive le varie entità del sistema, con le relative relazioni. Ogni classe è rappresentata da un rettangolo. All'interno ci sono il nome della classe, i vari attributi e metodi. Ogni attributo o metodo ha una visibilità (+ pubblica, - privata, # package, $\sim$ protected). Le classi rappresentano una vista statica del sistema, mentre gli oggetti sono le istanze di una classe e sono dinamici. Le classi possono anche essere in relazione tra di loro (associazione, generalizzazione, aggregazione, composizione). Il class diagram è utilizzato quando si vuole rappresentare la natura statica del sistema e le relazioni tra le varie componenti. Si possono anche attribuire le responsabilità alle varie classi; se si nota che una classe ne ha troppe è da valutare l'idea di spezzarla in più classi. Allo stesso modo, se una classe non ha responsabilità, è da valutare l'idea di rimuoverla
- **Modelli di interazione**: descrivono le interazioni tra le varie parti del sistema
- **Modelli di comportamento**: descrivono come il sistema deve agire

6 Architectural desing

Dopo aver raccolto i requisiti, le funzionalità ed i servizi che il progetto dovrà fornire si passa alla progettazione del software. L'obiettivo della progettazione dell'architettura è quello definire come il software debba essere organizzato. È una fase intermedia che tra la definizione dei requisiti e la progettazione nel dettaglio del sistema. È importante fare queste valutazioni ad alto livello per vari motivi:

- Migliore comunicazione con lo stakeholder. Parlare ad alta voce della struttura ad alto livello può aiutare a non scordare dei dettagli
- Evidenziare i problemi a livello architetturale così da tenerne conto già in principio
- Concordare il riutilizzo di software già disponibile o utilizzato in precedenza

L'utilizzo della progettazione architetturale si concentra nel facilitare la discussione sul design del sistema e la sua documentazione ad alto livello.
La sua produzione non è tanto un passo tecnico, quanto un processo che richiede di prendere delle decisioni. Per esempio, come i vari artefatti software verranno distribuiti nell'hardware, l'utilizzo di pattern architetturali che vengono riutilizzati nel tempo per realizzare determinati tipi di sistemi, la discussione su queste architettura permette inoltre di capire se è possibile soddisfare i requisiti non funzionali. Il sistema viene scomposto nei suoi componenti principali e così da capire se ognuno di essi ha una responsabilità definita, ambigua o distribuita su più componenti (questo causa problemi).
Le conseguenze che una scelta di un'architettura può avere sono:

- Performance
- Sicurezza
- Safety del sistema finale
- Affidabilità
- Manutenibilità

Scelta l'architettura è possibile fornire diverse rappresentazioni/viste della stessa.

- **Vista logica**
- **Vista di processo**: mostra quali sono e le interazioni tra i vari processi
- **Vista di sviluppo**: scompone il sistema in parti che verranno assegnate a team o sviluppatori distinti
- **Vista fisica**: permette di capire come i componenti potranno essere installati sui diversi dispositivi o i diversi host

È importanti rifarsi a dei pattern architetturali per avere una terminologia condivisa e universalmente accettata perché sono soluzioni buone a problemi ricorrenti. Avere dei pattern che rappresentano le best practice nelle scelte architetturali aiuta molto nella fase di progettazione. I pattern possono anche non fornire una soluzione adatta al problema da risolvere, ma a quel punto è compito dell'ingegnere integrarlo con qualche funzionalità in più per adattarlo meglio alle esigenze del caso. Nella realtà infatti i problemi non sono così semplici e ci si trova spesso a dover più che altro prendere spunto dai pattern adattando l'architettura scelta al caso in oggetto.

6.1 Model-View-Controller

È uno dei pattern più utilizzati per la programmazione web ed è abbreviato con l'acronimo MVC. Il pattern architetturale in questione prevede tre ruoli:

- **Model**: responsabile della gestione e della persistenza dei dati. Si interfaccia col DataBase (DB)
- **View**: è responsabile di costruire l'interfaccia grafica e presentarla all'utente finale
- **Controller**: è responsabile di implementare la logica di business. Deve essere in grado di catturare le azioni dell'utente e rispondere alle richieste che arrivano

Per comprendere meglio si consideri ad esempio una webapp. Il web browser riceve una pagina html costruita dal componente view e tutte le interazioni che l'utente con il web browser rappresentano un evento grafico che il web browser invia al controller il quale riceve gli input e li elabora per poi eseguire l'update sul DB attraverso il componente model.
L'MVC è utilizzato in Java anche per applicazioni desktop con grafica e per applicazioni Android.
L'MVC andrebbe usato quando i modi di interagire o visualizzare i dati sono molteplici (varie schermate) poiché è meglio disaccoppiare i dati dalla loro presentazione così che si possano creare più modalità di visualizzazione dei dati sfruttando la stessa interfaccia con il DB. Tuttavia il disaccoppiamento introduce un po' di complessità nel codice.

6.2 Layered architecture

Questo pattern architetturale prevede la presenza di più strati uno sopra l'altro.
Si parte dal livello "core", centrale, si costruisce sopra di esso cosicché lo strato sottostante fornisca dei servizi allo stato sovrastante.
Le comunicazioni sono solo tra strati adiacenti in modo da permette di utilizzare una strategia di sviluppo incrementale. Per cambiare uno strato bisogna però implementare tutte le interfacce del precedente così come sono implementate dallo strato da sostituire. Si prenda come esempio un sistema informativo; il primo stato comprende il sistema operativo ed il database, sopra c'è lo strato che implementa il core business, poi uno strato che si occupa dell'autenticazione e del management della user interface ed infine lo strato più in alto che gestisce l'interfaccia grafica.
Tale archawitettura a strati è utilizzata dal sistema ISO/OSI per le comunicazioni di rete.
Anche il sistema operativo Android utilizza un'architettura a strati.
Questo tipo di architettura è molto utilizzata soprattutto nella costruzione di servizi sopra altri servizi pre-esistenti ma si potrebbe anche utilizzare quando lo sviluppo dei vari strati è in capo a team di sviluppo diversi. È invece opportuno utilizzarla in caso di requisiti di sicurezza a più livelli poiché possono essere eseguiti controlli sulla sicurezza a più livelli, ed è possibile rimpiazzare uno o più layer fintanto che soddisfano la stessa interfaccia.
Ci sono tuttavia degli svantaggi; non è sempre possibile raggiungere un livello di isolamento totale, perciò bisogna permettere dei "buchi" per fare dei salti di layer.

Un altro problema tipico riguarda le performance perché, ad esempio, per fare una chiamata dal layer più in alto a quello più in basso viene eseguita una lunga catena di chiamate.

6.3 Repository architecture

Il pattern architetturale di tipo repository prevede un componente centralizzato che mantiene una versione autorevole dei dati e vari componenti satelliti che interagiscono solo col componente centrale.
I dati possono essere generati anche dai componenti satelliti e vengono salvati nel repository centrale che si occupa di aggiornare anche gli altri componenti satelliti.
Un esempio di repository è quello usato dai sistemi di version control (Git). I repository sono utilizzati quando ci sono grandi volumi di dati che devono rimanere consistenti a lungo nel tempo cioè nei sistemi **data-driven**.
Il vantaggio è che può esserci totale indipendenza tra i vari componenti satellite. I dati sono tipicamente consistenti perché c'è un solo punto in cui sono mantenuti ma di contro c'è il fatto che esiste un singolo point-of-failure ed anche uno svantaggio legato all'inefficienza derivante dal fatto che tutte le comunicazioni devono passare per il componente centrale il quale potrebbe trasformarsi in un collo di bottiglia.

6.4 Client-Server

Il pattern client-server prevede che ci siano uno o più componenti ad avere il ruolo del server e che forniscono servizi ai vari client. I server sono connessi a una rete attraverso la quale i client richiedono i servizi. Le richieste e le risposte sono gestite da un protocollo di comunicazione che, in quasi tutti i casi, è l'Hypertext Transfer Protocol (HTTP).
Questa struttura è consigliata quando si hanno dati messi a disposizione da una serie di servizi dislocati sul territorio.
Avere più server permette di replicarli scalando in modo da prevedere un re-indirizzamento e gestire meglio il carico. Tra gli altri vantaggi c'è il fattore della scelta della posizione dei server che per ridurre la latenza della comunicazione possono essere scelti il più vicino possibile al client, in caso in cui uno dei server avesse problemi e non fosse più disponibile il servizio non verrebbe negato ai client.
Tra gli svantaggi si considerino la difficoltà di prevedere le performance perché dipendono dal tipo e dal carico della rete ed il fatto che alcuni server potrebbero essere gestiti da terze parti.

6.5 Peer-to-peer

L'architettura peer-to-peer generalizza l'architettura client-server "mescolandola". Ogni componente dell'architettura gioca il ruolo sia di client che di server nel senso che entrambi i ruoli sono giocati dallo stesso componente e tutti i componenti sono sullo stesso livello.
Un vantaggio di queste reti è che scalano molto bene. Se uno dei peer non dovesse funzionare, il servizio verrebbe richiesto a un altro componente senza intermediazioni

o necessità di operazioni manuali. Un esempio di questa architettura è il protocollo **bitTorrent**.
Oltre ai client-server è necessario anche un **tracker** che mantiene traccia di quali **peer** afferiscono all'architettura.
Un altro vantaggio è la velocità perché dipende dalla struttura della rete cioè un file è suddiviso in varie parti e si può richiedere una parte da ciascun peer. L'architettura peer-to-peer è utilizzata anche da bitcoin.

6.6 Pipe and filter

Pipe and filter è un'altra architettura che prevede la partecipazione di vari componenti ad una pipeline di generazione e gestione dei dati.
Un componente genera i dati, li passa ad un altro che li filtra e poi ad un terzo che li analizza. Infine l'ultimo componente visualizza i dati elaborati.
Tipicamente questa architettura è utilizzata quando si devono gestire ed analizzare dei dati e non è appropriata per attività interattive in cui è previsto l'intervento dell'utente.
È inoltre opportuno utilizzarla quando l'input ha bisogno di vari step di processing prima di essere trasformato in output. Questa architettura rende molto facile capire le responsabilità dei componenti ed è facile anche modificarle perché è molto intuitivo aggiungere nuovi filtri sia in maniera sequenziale che in maniera concorrente.
Ci sono però anche degli svantaggi; il formato dei dati deve essere deciso prima di iniziare l'implementazione del sistema, ogni componente deve leggere i dati e farne il parsing per estrarre le informazioni prima di poterle elaborare e successivamente deve ricomporli in un formato di scambio (es. Extensible Markup Language (XML) o JavaScript Object Notation (JSON)).

7 Testing

In caso di testing all'inizio del processo di sviluppo, si tratta di "fase di validazione". Una volta implementato il software è necessario accertarsi della coerenza e del corretto funzionamento del sistema sia per quanto riguarda errori di programmazione sia per quanto riguarda errori dovuti al disallineamento tra i requisiti e il codice implementato.
L'obiettivo del testing è quello di trovare eventuali errori, verificare se il software rispetta e fornisce un'implementazione corretta dei requisiti raccolti in precedenza e permette di mantenere alta la qualità del codice. La soluzione idilliaca sarebbe la presenza di un test per ogni funzionalità o per combinazioni di funzionalità se queste comunicano tra loro.
È da aspettarsi anche la presenza di problemi di difformità dai requisiti oppure errori sul codice che provocano il blocco del sistema o generano errori inaspettati. Questi errori vengono tipicamente chiamati **bug**.
Quando si testa un pezzo di software, il programma soggetto a testing è detto System Under Testing (SUT). A questo sistema viene fornito un input e si attende la produzione dell'output; se l'output che è quello atteso, allora il test passa, altrimenti il test fallisce.

C'è tuttavia una terza condizione; se durante l'esecuzione accade un errore critico (es: divisione per zero), il sistema non produce l'output, ed in questo caso si parla di errore e non di fallimento.
L'obiettivo principale del testing viene fatto risalire alle considerazioni di Dijkstra. Egli disse che il testing non può dimostrare la correttezza di un programma in quanto il programmatore potrebbe anche essersi dimenticato un test, tuttavia può controllare la presenza di alcuni errori nel programma.
Per avere la certezza che il sistema sia corretto al 100% bisognerebbe scrivere tutti i casi di test cioè $\lim_{x\to\infty}$ dove x = Numero di casi di test.
Con il testing possiamo quindi cercare di trovare errori, ma non assicurarci che il programma sia corretto.
Non potendo avere la certezza della correttezza, ci si affida alla confidenza nell'aver testato la maggior parte delle combinazioni di input prima di consegnare il prodotto al cliente.
La confidenza non si può testare ma potrebbe essere guidata dalle funzionalità presenti nel software; la confidenza varia in base a qual è l'obiettivo del software (un software avionico richiede una maggior confidenza), anche gli utenti finali possono aiutare a capire quale deve essere la confidenza e lo stesso anche gli ambienti commerciali nei quali si sta operando (spesso è necessario uscire presto sul mercato anche con versioni poco testate).

Esistono due tipi diversi di verifica sulla correttezza e sulla presenza di errori. Uno è il **software testing** che prevede l'esecuzione del programma attraverso diversi scenari, l'altro è la **code inspection** o review che prevede di controllare il codice e la documentazione visivamente per scovare eventuali errori. La seconda ha alcuni vantaggi in quanto non è limitata al codice, ma si estende anche alla documentazione e non soffre il mascheramento degli errori perché non si esegue il codice.
La prima invece può anche essere applicata a versioni incomplete del software e può essere estesa anche alla qualità, ai commenti, al controllo delle inefficienze ed in generale a controllare di aver correttamente seguito una particolare convenzione nella codifica.

7.1 Development testing

Si possono identificare tre fasi di software testing; la prima è il development testing condotto durante lo sviluppo dagli sviluppatori stessi. L'obiettivo è scoprire difetti all'interno del software attraverso l'utilizzo di strumenti come debugger.
L'approccio adottato è abbastanza semplice; viene localizzato il punto di errore (fault), corretto e rimosso.
Il development testing viene condotto in maniera gerarchica e in diverse fasi.

7.1.1 Unit testing

La prima fase è detta unit test (test dell'unità) e si concentra sui singoli metodi o singoli oggetti o singole classi.
Ogni parte viene testata in isolamento. Data una classe, se ne testa ogni contenuto e si esercitano gli oggetti in tutti i loro stati possibili.
Tuttavia eseguendo i test in modo isolato è difficile avere totale completezza.

Uno unit test è un pezzo di codice composto da una parte iniziale che rappresenta il setup del test e porta la classe da testare in uno stato "testabile" (es. accende il dispositivo e riempie il DB), una parte di codice che esercita la funzionalità da testare ed una parte finale assertiva in cui sono formulate della asserzioni sullo stato raggiunto e sull'output prodotto.
Se tutte le asserzioni sono verificate, il test passa; se anche una singola asserzione non è verificata il test fallisce.
È auspicabile che i test di unità siano eseguiti in modalità automatizzata e batch. Il mock è un sostituto dell'unità che presenta la stessa interfaccia e viene richiamato esattamente nello stesso modo, ma i metodi ritornano dei valori costanti. Utilizza la stessa interfaccia degli oggetti che sta sostituendo, ma non ritorna i veri dati.
È meglio utilizzare un mock anche se si possiede una classe sensore già pronta perché così è più facile isolare l'errore. Ad esempio si possono usare dei mock per i dati del DB così che si possa escludere un problema nel database in caso di fallimento del test.
Gli scenari che vanno testati sono quelli che non provocano errori così da controllare se il flusso di esecuzione è corretto. Dopodiché si passa al testing di valori in input scorretti o che il sistema non si aspetta per controllare se il sistema risponde con l'errore corrispondente.
Lo spazio per le prove sull'input è molto grande, quindi si devono scegliere i valori da utilizzare e per farlo viene utilizzato l'**equivalence partitioning** cioè si partiziona l'input in classi di equivalenza e si assume che tutti i valori in quelle classi abbiano lo stesso comportamento.
Una volta partizionato lo spazio degli input si devono scegliere i valori da selezionare ed anche per questo esistono delle linee guida. Solitamente si scelgono dei valori agli estremi del dominio ed un valore centrale.
Quando gli input sono sequenze o array si devono testare sequenze con differenti numeri di elementi, le sequenze con un elemento e con zero elementi.
Infine bisogna forzare l'input per generare almeno una volta i casi di errore previsti, per verificare che sia tutto corretto (testing delle eccezione e dei casi d'errore).

7.1.2 Component testing

Durante la fase di component testing le classi, dove necessario, vengono messe in relazione tra loro e viene testata l'interfaccia di comunicazione e come si relazionano tra di loro.
Si deve verificare che il comportamento di una classe sia corretto nel momento in cui viene messa in relazione con altre classi.
Questa fase ha senso solo nel quando la fase di unit testing è stata eseguita in maniera abbastanza esaustiva e completa.
Il component testing coinvolge le modalità in cui una classe utilizza i servizi delle altre e come avviene lo scambio dei dati. Dunque bisogna testare i dati che vengono comunicati da una classe all'altra sia passandoli come parametri, sia usando l'eventuale memoria condivisa, sia usando l'interfaccia procedurale (protocollo implicito che definisce l'ordine con cui i metodi vanno richiamati).
Il primo ordine di errore è legato ad errori nell'utilizzo dell'interfaccia, come ad esempio dati di tipo sbagliato o passati nell'ordine sbagliato. Possono esserci anche

errori legati ad assunzioni sui dati ricevuti, come ad esempio aspettarsi che la lista fornita dalla classe mittente sia ordinata. Infine ci potrebbero essere anche errori legati al timing.
Per testare l'interfaccia è bene utilizzare i valori estremi nei loro intervalli di validità. Nel caso di puntatori a memoria, si può testare il passaggio di puntatori nulli.
I test devono tentare di mandare in fail uno dei due componenti oppure eseguire degli **stress-test** inviando dati con una frequenza molto più alta del normale.

7.1.3 System testing

Infine, il system testing è la fase di test a livello di sistema che si concentra sull'interazione tra tutti i componenti che concorrono all'esecuzione del software. Il focus in questo caso è sull'interazione tra i vari componenti per implementare le funzionalità ad alto livello richieste dagli stakeholders. Si testano quindi le funzionalità utili per l'utente finale ad esempio l'interfaccia web con il backend di un'applicazione web.
Per prima cosa si deve verificare la compatibilità dei componenti e se interagiscono e trasferiscono correttamente i dati tra loro. Si parte quindi dagli use-case che rappresentano una sorgente di informazione per verificare il comportamento del sistema.
Non potendo testare tutto ci si focalizza sulle cose più importanti richiedendo che tutte le istruzioni del sistema siano eseguite almeno una volta al termine dell'esecuzione del caso di test oppure richiedendo che ogni funzionalità del menù esposto all'utente venga eseguita almeno una volta oppure che vengano testate delle combinazioni di queste funzionalità.
Si deve anche verificare che la gestione degli errori sia implementata correttamente quindi deve essere prevista l'esecuzione delle funzionalità sia su input corretti che su input scorretti per controllare se il sistema risponde col messaggio di errore corretto.

L'approccio di programmazione **test-driven** consiste nello scrivere i casi di test ancor prima di aver implementato le varie funzionalità.
Il processo produttivo in questo caso è: aggiunta di una funzionalità vuota, creazione del caso di test che se eseguito sicuramente fallirà perché il corpo della funzione è vuoto, implementazione della funzione che soddisfa il caso di test, esecuzione del test e se fallisce iterazione del processo perché vuol dire che la funzione non è ancora implementata correttamente.
L'obiettivo di questo approccio è molteplice; scrivere il caso di test obbliga a pensare alla funzionalità in termini di cosa deve fare, invece del come deve farlo.
Tipicamente sviluppando il caso di test a posteriori si implementa la funzionalità solo dalla parte "come" e non dalla parte "cosa".
L'utilizzo di questa metodologia ha numerosi vantaggi. Innanzitutto, i casi di test coprono bene il codice dell'applicazione, si possono inoltre eseguire i regression test cioè esecuzione dei test man mano che si sviluppa una nuova funzionalità complessa così da accertarsi che le nuove implementazioni non abbiano compromesso funzionalità già implementate, si semplifica anche l'attività di debugging perché ogni caso di test si riferisce ad una singola funzionalità e se fallisce si sa con precisione quale controllare.
I casi di test a prescindere da tutto fungono anche da documentazione per il sistema.

7.2 Release testing

Una volta che è stata implementata la release (parziale o finale) da consegnare, il team deve testarla prima di consegnarla al cliente.
Tipicamente vengono eseguiti test differenti rispetto al development testing per controllare eventuali errori di diversa natura riguardanti i requisiti richiesti.
Per questa fase si utilizza un approccio **black-box** cioè ci si limita ad un comportamento input/output con asserzioni che vengono formulate solo sull'output che viene prodotto e si prescinde dai componenti che compongono il sistema.
Le differenze tra il system testing ed il release testing sono:

- Il release testing è affidato a un team esterno
- Il system testing è incentrato sulla ricerca di difetti
- Il release testing verifica che il sistema risponda in maniera corretta ai requisiti

Un modo per condurre questo test è quello basato sugli scenari; seguendo gli scenari, ci si basa su casi di utilizzo realistici. Un secondo ordine di test si può condurre sulle performance del sistema attraverso stress-test per il sovraccarico deliberato.
Se il sistema supera il release testing, può essere consegnato al cliente.

7.3 User testing

È l'ultima fase di testing, che prevede il coinvolgimento del cliente ma non è detto che venga fatta poiché dipende molto dalle politiche aziendali. L'obiettivo è testare il software nell'ambiente in cui viene utilizzato perché l'ambiente di esecuzione potrebbe influenzarne l'utilizzo.
Vengono coinvolti utenti e stakeholder prima con un **alfa-test** composto da un numero molto limitato di utenti, poi con un **beta-test** che invece coinvolge un numero maggiore di utenti tenuti a segnalare la presenza di errori. Alla fine si esegue l'**acceptance-test** che coinvolge gli utenti finali.
È quindi l'utente finale che decide se la versione del software è adatta alla produzione o meno. Se mancano funzionalità o sono presenti errori il software deve essere rielaborato.
All'inizio dello sviluppo vengono definiti i criteri di acceptance-test, dopodiché viene definito un piano di acceptance-test che può essere descritto ad alto livello (descrive quali requisiti sono ricoperti dall'acceptance-test e quali no). Quando la release è pronta per essere consegnata, il cliente prima di saldare il pagamento, esegue i test di accettazione. Se falliscono si apre una contrattazione tra il cliente e lo sviluppatore per capire se il software può comunque essere accettato dal cliente oppure se c'è bisogno di una revisione del codice e del sistema.
Per quanto riguarda il metodo agile il test ha un ruolo molto importante perché gli utenti/clienti fanno parte del team di sviluppo e partecipano attivamente alle discussioni è quindi essenziale che questi utenti/clienti rappresentino tutte le categorie destinatarie del software.

8 Refactoring

Il refactoring è un cambiamento alle strutture interne di un programma per renderle facili da comprendere, più economiche da mantenere e di qualità maggiore ed è eseguito a beneficio degli sviluppatori senza che l'utente finale se ne accorga.
Viene attuato per contrastare la naturale degradazione del software durante il suo ciclo di vita e si può considerare come una sorta di manutenzione preventiva per evitare che a lungo andare ci siano dei deterioramenti e delle complicazioni (es: mancanza di supporto per alcuni web browser).
Principalmente mira a modificare la struttura del codice per ridurne la complessità così da renderlo più facile da comprendere. Quando si affronta l'attività di refactoring del codice si modifica la struttura del codice senza agire sulle funzionalità. Questa fase prevede che l'attività di aggiunta di funzionalità venga sospesa.
Un altro motivo rilevante per eseguire il refactoring oltre che per aumentare/ripristinare la qualità del codice è il facilitare la scrittura dei casi di test oltre ad agevolare i futuri cambiamenti.
Un concetto vicino a quello del refactoring è quello del **reengineering**. Quest'ultima è un'attività svolta su software di dimensioni elevate e con una vita molto lunga. Il reengineering consiste nel riprogettare il sistema per una nuova architettura. Si riscrivono quindi diverse parti del software.
Il refactoring, invece, è un'attività condotta in parallelo durante tutto lo sviluppo del software, al contrario del reengineering che, invece, è un'attività puntuale.
Esistono delle linee guida per aiutare a capire quando è il caso di eseguire il refactoring.

- Prima di aggiungere una nuova funzionalità perché è più facile rispetto a farlo dopo
- Quando si corregge un bug o ci si trova in presenza di code-smell
- In generale va eseguito il più spesso possibile

8.1 Code-smell

Uno dei modi per identificare la necessità di eseguire il refactoring è quella basata sul code smell. La definizione comprende varie casistiche tra cui:

- Codice duplicato che limita la correzione degli errori in quanto va fatta per ogni parte duplicata
- Metodi composti da molte righe di codice ai quali tipicamente corrispondono troppe responsabilità (un metodo non dovrebbe essere più lungo di una schermata e non dovrebbe essere pieno di commenti)
- Presenza di costrutti switch-case da sostituire con il polimorfismo nei linguaggi object-oriented
- Data-clumping cioè un gruppo di dati che vengono acceduti sempre insieme e che quindi dovrebbero essere messi in una classe con dei metodi dedicati all'accesso
- Peculative generality cioè quando lo sviluppatore aggiunge una funzionalità molto generica per eventuali sviluppi futuri con il risultato di avere codice non utilizzato nel software

- Dead-code
- Long parameter list
- Blocco try-catch con vuota la parte del catch

Particolare attenzione va data al codice clonato cioè codice riutilizzato attraverso il copia-incolla.
Di solito lo si attua anche per la creazione di funzionalità simili apportando quindi leggere modifiche al codice copiato.
Questo sistema è perché prono alla propagazione di bug molto difficili da correggere (ci si deve ricordare tutti i punti in cui si è fatto il copia-incolla). Fortunatamente esistono dei software che aiutano a trovare le parti clonate.

8.2 Processo e tipologie di refactoring

Il processo di refactoring parte quando i casi di test non contengono nessuno errore. Successivamente si identificano i code smell, si decide quale tecnica di refactoring adottare ed infine si eseguono nuovamente i test per confermare che la modifica non abbia generato errori.
Avere pochi test o non averne rende questa operazione difficile perché insicura. I casi di test infatti proteggono anche da errori che si possono inserire con l'attività di refactoring.
È consigliabile svolgere l'attivita di refactoring per piccoli passi; piccoli cambiamenti per volta e verificando ad ogni passo le modifiche eseguendo i test.
Il refactoring può essere fatto manualmente oppure utilizzando degli strumenti automatici che ne velocizzano il processo. Gli Integrated Development Environment (IDE) più comuni implementano un catalogo di refactoring per automatizzare questa operazione ed i refactoring più semplici è bene farli con tali strumenti automatici (es: rinominare file, classi, metodi e variabili). Tuttavia il refactoring automatico può causare degli errori, ed i casi di test aiutano a tutelarsi anche da queste problematiche. I tipi di refactoring più comuni sono:

- **Extract interface**: una classe A fa riferimento ad una struttura dati (es: ArrayList), però potrei voler introdurre un'interfaccia e così disaccoppiare l'utilizzo che la classe A fa della struttura dati, dall'implementazione. Introducendo un'interfaccia che mette a disposizione dei metodi per gestire la struttura dati ottengo il vantaggio di poter in futuro cambiare la struttura dati di riferimento senza dover modificare la classe A
- **Pull-up**: data una serie di classi con una classe che estende una super classe che a sua volta estende un'altra classe, pull-up è la tecnica che permette di spostare dei componenti da una classe a quella superiore per astrarli maggiormente
- **Extract method**: se alcune parti di uno o più metodi sono uguali si può prendere la parte di codice duplicata e creare un metodo a sé stante, così da non avere codice duplicato
- **Move method**: spostare il metodo da una classe all'altra perché sfrutta maggiormente i metodi della seconda

- **Replace temp with query**: invece di utilizzare variabili temporanee è preferibile utilizzare metodi che ne calcolano e ritornano il valore così da poterli utilizzare più volte in vari metodi.
- **Replace parameter with method**: invece di calcolare un risultato su una variabile temporanea e poi passare tale valore come input di un metodo, si può passare ottenere quel valore all'interno del metodo invece di riceverlo come parametro
- **Extract class**: se una classe ha troppi metodi o troppi dati vuol dire che ha troppe responsabilità. Tipicamente queste classi hanno delle responsabilità che sono poco coese e connesse tra di loro, per questo sarebbe bene spezzarle per avere delle responsabilità ben definite di modo che siano riutilizzabili più facilmente mantenibili

Altri tipi di refactoring più complessi che solitamente non possono essere fatti da uno strumento automatico a meno che non vengano eseguiti come sequenze di refactoring semplici sono:

- **Replace inheritance with delegation**: si consideri una classe Stack e una classe Vector. La prima classe implementa le funzioni push e pop. Queste funzionalità possono essere implementate come un vettore. Un modo per farlo è quello di far estendere la classe vector dalla classe stack. Se stack estende vector però espone tutte le funzionalità che sono disponibili nel vector e vector contiene anche funzionalità che non hanno senso per stack. Per questo è meglio sostituire l'ereditarietà con la delega. La classe stack non estende la classe vector, ma utilizza il vector come una struttura dati. In questo modo le funzionalità di vector che si vuole esporre anche su stack devono essere fatte attraverso delega
- **Replace conditional with polymorphism**: si consideri una classe Client che fornisce una funzionalità per calcolare l'area di varie figure implementata con uno switch-case. Tipicamente il refactoring prevede di utilizzare una classe astratta Shape che mette a disposizione un metodo per calcolare l'area, ma senza implementarlo. Il metodo concreto viene poi implementato nelle varie classi che estendono la classe shape.
- **Separate domain from presentation**: quando si inizia ad implementare l'interfaccia grafica che contiene anche la logica di business, senza aver condotto un passo di design precedentemente, l'interfaccia grafica finisce per essere modificata sia da chi si interessa dell'aspetto grafico, sia da chi si occupa della logica di business. Quindi è importante che i vari domini dell'applicazione siano separati. Per passare da un design monolitico ad uno modulare vanno fatti una serie di design passo-passo partendo da un'extract-class, per poi passare ad un refactoring move-field e move-method ed infine un'extract-method così da distribuire le funzionalità nelle varie classi

9 Risk

È un argomento trasversale, ma molto importante per l'ingegneria del software. I progetti software sono difficili da condurre e da portare a termine con successo perché

devono soddisfare i vincoli di organizzazione, di budget e di tempo.
Per raggiungere questi obiettivi ci sono delle metodologie che permettono di agevolare il lavoro.
Sicuramente se non si adotta una buona strategia di management, il progetto è destinato a fallire, tuttavia non si può essere al 100% sicuri che adottando un buon management il risultato sarà positivo, quanto meno però si è sulla buona strada. A differenza di altri tipi di progetti, il software è intangibile, quindi è difficile capire quale sia esattamente lo stato di avanzamento oltre al fatto che ogni progetto software è di per sé unico e richiede la risoluzione di nuovi problemi e che il processo adottato può variare in base all'organizzazione nella quale si opera.
Ci sono numerosi fattori che possono influenzare il progetto e il tipo di project management che si decide di implementare:

- **La dimensione dell'azienda**: in aziende piccole le persone sono più controllate ed è possibile anche una comunicazione informale tra le persone
- **Il tipo di stakeholder**: il software viene scritto in base allo stakeholder di riferimento. Se il software da realizzare andrà utilizzato all'interno della stessa azienda sarà molto più facile comunicare con gli stakeholder
- **La dimensione del software**
- **Il tipo del software**: software critico ha tendenzialmente bisogno di un processo di sviluppo abbastanza specifico in cui ogni scelta deve essere motivata
- **La cultura dell'organizzazione**: l'azienda è disposta a prendersi dei rischi oppure no
- **Il processo di sviluppo**: avrà un impatto sul tipo di gestione del progetto

Per gestire un progetto si seguono alcune fasi:

- **Project planning**: definizione di una pianificazione per il progetto, date delle release ecc, quando verrà consegnato il progetto
- **Risk management**: definizione delle procedure da adottare in caso si manifestino dei problemi
- **People management**
- **Reporting**: consegna dei report sullo stato di avanzamento del progetto
- **Proposal writing**: definizione di una proposta di progetta, una volta finita la progettazione, da consegnare al decision maker

Per quanto riguarda il Risk management si possono individuare varie fasi.

9.1 Risk identification

È la parte più corposa dal punto di vista temporale, perché bisogna identificare quali sono i rischi che il progetto può incontrare.
I rischi possono essere di stima sull'effort, sul tempo, sulle persone necessarie e possono compromettere l'intero progetto perché comportano errori di stima del costo. Vanno considerati anche i rischi organizzativi (potrebbero derivare dall'azienda, come problemi finanziari, problemi di budget), rischi legati al personale (potrebbe essere difficile acquisire le persone con la competenze necessarie a lavorare al progetto oppure alcuni sviluppatori potrebbero andare in malattia, rischi legati ai requisiti

(mancanze nella raccolta dei requisiti che quindi vanno rivisti e conseguentemente va riscritto anche il codice corrispondente), rischi legati alla tecnologia (ad esempio l'utilizzo di un DB che non è in grado di soddisfare tutte le richieste ed infine rischi legati agli strumenti che vengono utilizzati per facilitare il lavoro come generatori di codice o altro.

9.2 Risk analysis

Una volta identificati i rischi, essi vanno analizzati perché non sono tutti ugualmente importanti.
Sono quindi da assegnare le priorità in base all'importanza del rischio. La criticità viene rappresentata in due dimensioni: la probabilità che il rischio si manifesti e la **seriousness** cioè l'impatto che avrebbe sul progetto. Tipicamente si utilizza la scala (molto bassa, bassa, alta, molto alta) per rappresentare le criticità ed essendo una scala qualitativa è abbastanza soggettive, pertanto spetta all'ingegnere essere il più oggettivo possibile.
Per ogni rischio quindi si deve associare un valore per la probabilità e la seriousness.

9.3 Risk planning

Dopo aver analizzato i rischi è bene cercare di capire quali sono le azioni da intraprendere nel momento in cui questi rischi si dovessero manifestare.
Per prima cosa bisognerebbe cercare di evitare il rischio, cambiando il progetto e cercando di evitare che si manifesti.
Se non è possibile evitarlo, si cerca di minimizzarne l'impatto. Importante è la creazione di un piano di contingenza (es. rischi finanziari), se non è possibile minimizzare il rischio. Questo è un piano che entra in gioco solo nel momento in cui il rischio si manifesta e serve per evitare che lo stesso crei troppi danni al progetto.

9.4 Risk monitoring

I rischi vanno costantemente monitorati per essere pronti ad intervenire qualora si manifestassero ed ancor prima per prevenirli nel caso in cui aumentasse la probabilità del loro manifestarsi.
Per monitorare i rischi si possono utilizzare degli indicatori di rischio, indicatori quantitativi che permettono di capire quando un rischio si sta avvicinando. Ad esempio, se si è in ritardo in ogni consegna, vuol dire che si sono stimati male i tempi di sviluppo e di consegna.
Il monitoraggio va fatto ovviamente durante lo sviluppo del progetto e non alla fine così da avere il tempo di risolvere l'eventuale problematica venutasi a creare.

9.5 Gestione del personale

Gli ingegneri del software sono molto costosi poiché sono pochi e la richiesta è molto alta.
La gestione scorretta del personale porta inevitabilmente al fallimento del progetto.

Uno degli obiettivi è dunque quello di scegliere un gruppo di perone abbastanza coeso in modo da favorire l'apprendimento tra le persone del team possano e lo stimolo a vicenda a creare software di qualità.
Per comporre il team in modo corretto bisogna conoscere le personalità dei dipendenti ed abbinarle in modo bilanciato.
Alcune delle personalità sono:

- **Task-oriented**: trovano motivazioni nel lavorare e sono perfezioniste. Sono persone che lavorano bene anche da sole. La maggior parte degli ingegneri del software ha questa personalità
- **Self-oriented**: lavorano per raggiungere degli obiettivi personali che sono esterni al lavoro (es. la volontà di avere una bella macchina, casa, poter viaggiare...). Queste persone, tipicamente, vorrebbero avere il comando
- **Interaction-oriented**: sono stimolate ad andare al lavoro per le interazioni sociali. Il problema di queste persone potrebbe essere quello di perdere molto tempo a parlare e scambiare idee

Anche per quanto riguarda l'assegnazione del personale e le modalità di pianificazione esiste il processo **plan-driven** cioè a piani. Tipicamente, la gestione del progetto per piani consiste nel definire degli step che definiscono cosa deve essere fatto, a chi deve essere assegnato il task, quali task ci sono e quali sono i termini temporali dei task.
Questa è la modalità più facile che permettere di stimare i tempi e quindi capire se si è in ritardo rispetto a quanto pianificato oltre a dare la possibilità al management di prendere delle decisioni informate in base al livello di sviluppo del progetto.
È importante che vengano fatte delle review regolari per controllare se si è in linea con quanto pianificato ed in caso quali sono le motivazioni che hanno comportato il non rispetto della pianificazione.
Le attività del progetto vanno divise in task che contengono alcune informazioni tra cui il tempo necessario per essere eseguite, il giorno in cui la si può iniziare a sviluppare, eventuali dipendenze tra le task (cioè se delle task vanno eseguite prima e altre dopo) e l'effort necessario per portarle a termine.
Si possono anche definire delle **milestone** all'interno del progetto cioè punti in cui un particolare obiettivo dovrebbe essere stato raggiunto.
Il momento del raggiungimento della milestone permette di avere una chiara idea sul rispetto delle tempistiche pianificate. Lo stesso principio può essere applicato ai **deliverable** cioè tutti quei documenti o demo che possono essere consegnati al cliente.

La pianificazione del progetto può essere rappresentata in una tabella, dove ogni task è una riga ed ogni colonna rappresenta sia l'effort che la durata in giorni che eventuali dipendenze.
La rappresentazione migliore per la progettazione del progetto è il diagramma di Gantt, diagramma che rappresenta il flusso temporale del progetto. Sulle ascisse è rappresentato il tempo, dal giorno zero al termine mentre sulle ordinate sono rappresentate le task. Nel diagramma è presente una barra continua per ogni task, la barra inizia il giorno in cui la task dovrebbe cominciare e termina il giorno in cui dovrebbe finire perciò la lunghezza rappresenta la durata.

Sono rappresentate anche le milestone con dei rombi e spesso si trovano all'inizio o alla fine delle task e si possono rappresentare esplicitamente o meno anche le dipendenze con delle frecce che iniziano dalle task che sono la dipendenza e terminano nella task che richiede la dipendenza.
Questo diagramma permette di ragionare su come si possono ordinare le task per liberare risorse ed utilizzarle in modo migliore e più efficiente.
Un altro diagramma meno utilizzato è lo staff-allocation che, una volta fatto il digramma di Gantt, permette di allocare le persone alle varie task. Anche in questo caso l'asse delle ascisse rappresenta il flusso temporale e quello delle ordinate rappresenta le persone.
La barra piena rappresenta una persona allocata full-time mentre la barra a metà rappresenta una persona allocata part-time. Questo diagramma permette di capire quanto stanno lavorando i dipendenti e quando entrano ed escono dal progetto.
Questo diagramma fornisce anche una visuale diversa sulle relazioni di dipendenza perché rende evidente quante e quali persone stanno lavorando ad una task, per quanto tempo e cosa provocherebbe un ritardo in una task per le task successive.
Questo tipo di gestione è utilizzata anche su progetti molto lunghi (anche di anni) dove spesso la progettazione definita all'inizio non è rispettata del tutto e dovrà quindi essere cambiata ed adattata nel tempo.

L'alternativa alla gestione a piani di un progetto è la gestione agile che ricalca un po' l'idea dello sviluppo agile del processo. Il sistema agile pone meno enfasi sulla progettazione dettagliata e più enfasi sulla mutabilità del progetto rispetto al cambiamento delle condizioni esterne. È sempre presente una fase di **release planning** in cui si pianificano le release ed un'**allocation planning** che ha un tempo molto più corto di esecuzione.
Per questo tipo di pianificazione si usa il cosiddetto **planning game** che ha alcune fasi. Si inizia con l'identificazione di storie che rappresentano le feature e ad ognuna di esse si assegna l'effort che serve per portare a compimento la storia così da poter stimare anche l'effort complessivo dell'intero progetto. Dopodiché bisogna pianificare le release scegliendo in che ordine vanno implementate le feature e il tempo totale di ogni ciclo.
Si utilizza la velocity per capire quale delle storie inserire nella release ed all'inizio di ogni iterazione viene assegnata l'allocazione delle task in cui gli sviluppatori scelgono su quali task lavorare.
La scelta è fatta su base volontaria ma tutte le task devono essere implementate per completare le storie.
Circa a metà dell'iterazione e consigliabile fare una review per valutare se si è in orario con lo sviluppo o se si è in ritardo e nel caso riorganizzare le task rimanenti per poter consegnare un prodotto stabile seppur parziale al termine delle release. In questo caso il termine "task" è diverso da quello usato precedentemente nel diagramma Gantt perché in questo caso le task sono precise e non generiche.
La scadenza non può mai essere spostata quindi l'unica possibilità in caso di ritardi è la riduzione delle storie o task incluse nella release.
Questo planning ha dei vantaggi tra cui l'incentivare la comunicazione e la conoscenza delle dipendenze, la possibilità di scelta delle proprie task per i programmatori che quindi sentiranno un senso di appartenenza alla task e saranno più motivati a

portarle a termine.
Tuttavia se il cliente non è disponibile a partecipare al planning game sarà difficile implementare questa metodologia. Inoltre la metodologia agile funziona bene dove la comunicazione può essere informale e quindi è più faticosa da utilizzare in team numerosi o molteplici.

10 Domande

Cosa si intende per digramma di Gantt?
Cosa si intende per gestione del rischio? Perché è importante? Come si gestisce?
Cosa si intende per Selenium?
Cosa si intende per Unit Test?
Cosa si intende per Git clone? Quali sono i branch principali?
Cosa intendiamo con processo software?
Perché è importante dotarsi di un processo software?
Descrivi il processo guidato a piani e come si compone
Quali sono gli svantaggi del modello guidato a piani?
Se le fosse assegnato un processo software e dovesse dotarsi di un team per sviluppare e consegnare il progetto, in quali casi si doterebbe del processo software a cascata?
Che vantaggi ha Selenium rispetto all'uso di jwebunit per quanto riguarda gli acceptance test?
Descrivere il processo di sviluppo incrementale
Quali sono gli svantaggi dello sviluppo incrementale?
Se si utilizza il sistema plan-driven che cambiamenti bisogna fare per passare al sistema agile? Che spiegazione si può dare al team?
Deduzione dei requisiti plan-driven, come si mappa su un processo agile?
Come protrebbero venire trattai i requisiti nello sviluppo agile?
Cos'è lo scrum?
Come si reagisce se un task è stato sottostimato in scrum?
Descrivere la gestione della manutenzione con lo scrum
Come scala il sistema agile?
Che opposizioni ci possono essere alla produzione agile?
Se l'azienda adotta procedure di quality enforcement come potrebbe reagire allo scrum?
Ci sono estensioni di scrum con più team di sviluppo?
Che vincoli ci sono sulla consegna in scrum multi team?
Le consegne devono essere allineate?
Cosa rappresenta e a cosa serve l'activity diagaram?
Quali sono le differenza tra decision e fusion/merge?
Descrivere l'MVC
Quali sono le caratteristiche dell'architettura a strati?
Si descriva l'architettura pipe and filter
Descrivere il modello di sviluppo basato sul riutilizzo

Quali sono le specifiche nel configuration and reutilisation?
In una startup di 1, 2 persone si potrebbe adottare l'integration?
Quali sono i problemi dell'ingegneria dei requisiti?

Acronimi

DB DataBase. 24, 28, 35

HTTP Hypertext Transfer Protocol. 25

ICSE International Conference of Software Engineering. 4
IDE Integrated Development Environment. 32
IEEE Institute of Electrical and Electronics Engineers. 21

JSON JavaScript Object Notation. 26

SUT System Under Testing. 26

UML Unified Modeling Language. 22

XML Extensible Markup Language. 26