

IAN SOMMERVILLE- SOFTWARE ENGINEERING

NONA EDIZIONE

Disclaimer *I termini progetto, sistema, software e applicazione vengono molto spesso intercambiati, questo poichè l'autore tende a riferirsi allo stesso concetto evitando di usare sempre la stessa parola*

L'autore del libro introduce il libro rispondendo ad alcune domande frequenti riguardanti cosa dobbiamo aspettarci dallo studio del software applicato all'ingegneria, i costi di produzione, le sfide che ogni ingegnere del software deve affrontare e le caratteristiche che un buon software deve avere.

What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, evolution costs often exceed development costs.

Perchè la produzione del software diventa ottimale quando viene affiancata ad una buona pratica ingegneristica?

Gli approcci ingegneristici servono a far 'funzionare le cose' tramite l'applicazione di teorie, metodi e strumenti adatti per la risoluzione dei vari problemi che la progettazione software richiede.

Si possono identificare 4 caratteristiche predominanti usati come 'unità di misura' della bontà del software:

1. Maintainability, ovvero la capacità del software di evolversi nel tempo, con l'idea di rispettare le aspettative dei clienti, che diventano più esigenti via via che nascono nuovi requisiti applicativi.
2. Dependability: un software 'dependable' dovrebbe essere in grado di saper badare a se stesso, ovvero essere sicuro e non portare malfunzionamenti fisici alla macchina, o perdite economiche agli sviluppatore.
3. Efficiency, ovvero la capacità del software di non produrre sprechi, o , piu realisticamente, di non produrne troppi
4. Acceptability, ovvero la qualità intrinseca del sistema di essere leggibile e fruibile a tutti gli utenti interessati, nonché compatibile con tutte i dispositivi a cui è destinato.

Generalmente la produzione software passa per 4 step principali:

1. Software specification, ovvero lo studio di fattibilità e l'analisi dei requisiti che introducono il progetto in sé
 2. Software development, il 'coding' e lo sviluppo del progetto avvengono in questa fase
 3. Software validation, la fase di testing del progetto, che garantisce la funzionalità del sistema
 4. Software evolution, la fase in cui nuove versioni del software vengono rilasciate, correggendo errori e migliorando l'applicazione grazie (anche) al feedback dei clienti
- Un aspetto messo in risalto da Sommerville è che uno sviluppatore non è un semplice tecnico dell'informatica ma deve essere attento anche alle questione etiche

Modelli di produzione software

Nella produzione software ci sono alcuni modelli (archetipi, framework) che vengono presi in considerazione con l'idea di dare ordine alla produzione stessa, non sono modelli molto rigidi, servono più che altro come linee guida per orientarsi nei vari step, alcuni tra i più noti riguardano:

Modello a cascata (waterfall method)

Ognuno dei processi software menzionato in precedenza viene trattato come un blocco separato, generalmente dipendente a catena l'uno rispetto all'altro. Si è soliti identificare 5 step distinti:

1. Analisi dei requisiti, ovvero comprendere gli obiettivi del software, le richieste del cliente, i tool e i costi necessari per l'implementazione, eventuali scadenze.
2. Design software, studio dei requisiti hardware e software, insieme alle descrizioni dei tool da applicare e le connessioni fra le varie parti del software

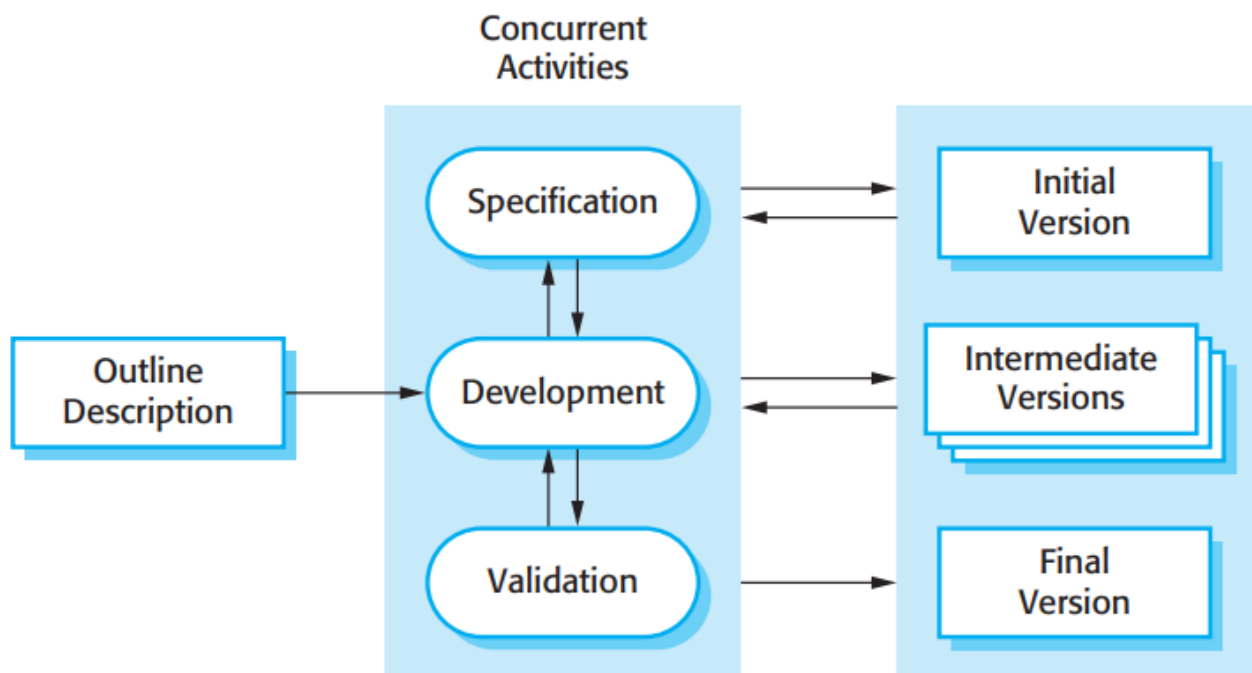
3. Implementazione e unit testing, implementazione a blocchi (unità) e successivo testing
4. Integrazione e testing del sistema, merging delle unità e successivo testing del sistema, che include la messa in opera e la consegna al cliente
5. Manutenzione, la fase più lunga e spesso più costosa, che richiede continui aggiornamenti e supporto sia ai clienti che al sistema

Generalmente il modello a cascata è molto robusto, documentato e ampiamente utilizzato in vari ambiti

Sviluppo incrementale (incremental development)

Lo sviluppo incrementale si basa sull'idea di fornire al cliente una prima versione, esponendola ai feedback e evolvendola in base alle richieste dell'utente e a vari problemi riscontrati.

Le attività di *specifica*, *sviluppo* e *validazione* sono fortemente interfogliate tra loro e tra le versioni incrementali rilasciate nel tempo, sottolineando una forte connessione tra le varie versioni rilasciate



Di seguito i benefici di un approccio incrementale:

1. i costi di manutenzione ed evolvibilità sono ridotti e la documentazione è meno rigida
2. La probabilità di venire incontro alle esigenze del cliente è molto alta grazie alla comunicazione diretta che si crea tra sviluppatore e committente
3. le delivery del progetto sono molto rapide in quanto non si deve attendere il rilascio di versioni definitive

Alcune problematiche da tenere in considerazione

1. Il metodo incrementale richiede molta attenzione da parte del manager di progetto in quanto la documentazione è spesso insufficiente per stare al passo con l'evolubilità del sistema
2. La struttura rischia di perdere di coesione in quanto il rilascio continuo di versioni potrebbe far perdere la capacità di comprendere la struttura iniziale

Software orientati al riuso

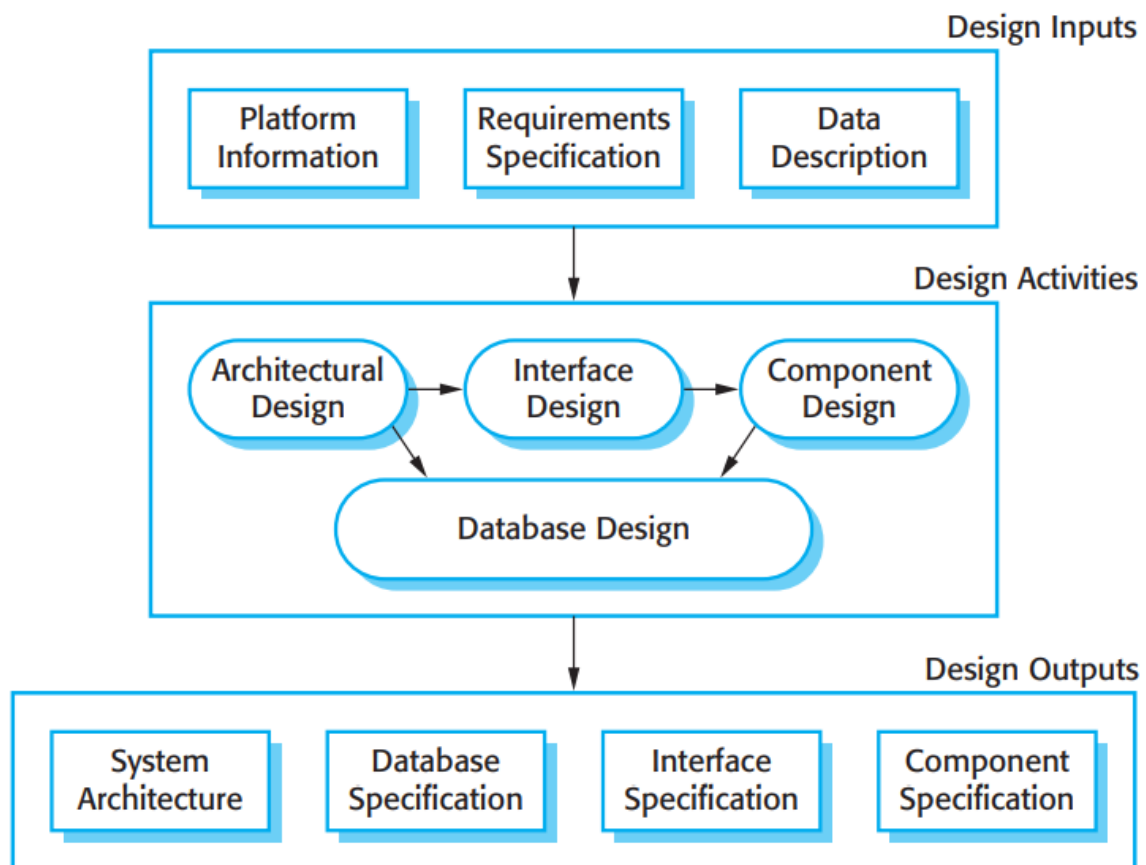
I software orientati al riuso sono molto comuni nella comunità informatica, molto spesso uno sviluppatore si rende conto che del codice di cui è familiare può essere utile nella stesura di un nuovo progetto, quindi lo prende in prestito, lo modifica, adattandolo alle sue esigenze e lo riusa,

un modello che si basa su questa strategia consta di 4 punti chiave:

1. Analisi dei componenti
2. Modifica dei requisiti
3. Design tramite il riuso
4. Sviluppo e integrazione

Software design e implementazioni

Il design di un software ha il ruolo di descrivere l'implementazione del progetto, tramite data models e strutture, la stesura di un design coerente e robusto non è fatta a priori ma in maniera iterativa man mano che nuove funzionalità vengono implementate e nuove problematiche emergono.



Guardando la figura in alto notiamo che il processo di modellazione del design segue una logica Ingresso-Stato-Uscita dove gli input sono di tipo descrittivo e riguardano i dati che abbiamo a disposizione per modellare il progetto, il core del design, ovvero la parte 'tecnica' richiede competenze di design vere e proprie, in questa parte vediamo come diverse strategie implementative vengono interfogliate per dare vita al progetto, una volta pronto il design possiamo stilare una documentazione di quanto fatto contenente le specifiche descritte nel design.

Validazione software

La validazione software è un insieme di tutte quelle azioni post-implementazione che servono a garantire che il sistema funzioni come dovrebbe, potrebbe sembrare la parte più banale nell'implementazione di un progetto, ma spesso è quella che si protrae per più tempo, in quanto eventuali errori spesso compaiono molto tempo dopo il rilascio del software, inoltre per sistemi molto complessi, la fase di testing richiede molta attenzione perché devono essere svolti tutti i test relativi alle varie funzionalità del software, sia nei casi standard, che nel funzionamento nei casi limite.

Ci sono 3 tipi di testing che vengono effettuati:

1. Testing di sviluppo: ogni unità viene testata dagli sviluppatori indipendentemente dalle altre.
2. Testing di sistema: le varie unità vengono integrate e il funzionamento del sistema viene testato come un unico grande blocco.
3. Testing di accettazione: il test finale che viene fatto prima di mettere in uso il sistema, può essere considerato come un test di rifinitura.

Banale aggiungere che in casi di sistemi complessi le varie fasi di testing sono interfogliate tra di loro, rendendo il testing più elastico.

Evoluzione software

A proposito di elasticità, la produzione di un buon software deve essere elastica al cambiamento, capace di evolversi in base alle necessità dei clienti, e capace di implementare nuove funzionalità.

Lo studio di evolvibilità del sistema è molto importante poiché un sistema evolvibile richiede sia dei costi, sia la problematica di dover riscrivere codice, in modo tale da potersi adattare alle nuove funzionalità, per questo motivo l'evolvibilità deve essere pianificata con attenzione.

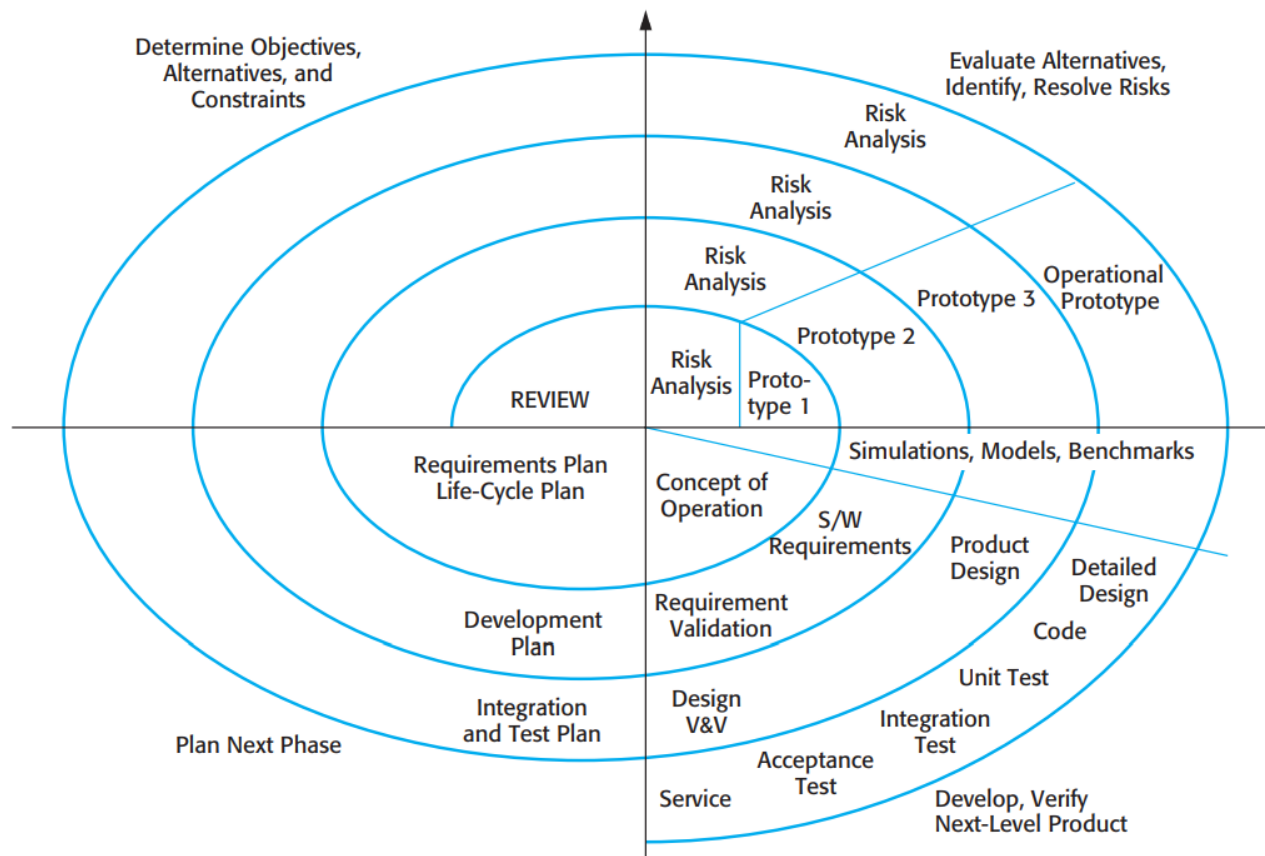
Incremental delivery (di nuovo (?))

La consegna incrementale è quel modello di sviluppo software in cui la versione messa a disposizione al committente è ancora in fase di sviluppo, una volta pianificati gli incrementi vengono integrati nella versione parziale e rilasciati con tramite una nuova versione del sistema al committente.

Questa strategia mette in comunicazione diretta il cliente con lo sviluppatore, tramite feedback sui prototipi che vengono utilizzati, inoltre ha il vantaggio di poter essere rilasciata in molto meno tempo, non dovendo aspettare la versione definitiva.

Essendo la consegna incrementale un processo iterativo soffre di tutti quei problemi di gestione, documentazione, riuso del codice e dispendiosità dei testing, nonché una coordinazione tra cliente e committente nel ricevere e inviare feedback.

Modello a spirale



Questo modello è caratterizzato da cicli (loop) che si compongono da 4 fasi:

1. Impostare gli obiettivi: Ovvero concepire strategie, valutare rischi, e porre in considerazione piani dettagliati
2. Analisi dei rischi: per ogni rischio va fatta un'analisi, e creare un sistema prototipo che li rimuova (o riduca)
3. Sviluppo e validazione: si sceglie un modello di sviluppo e lo si esegue fino a fine ciclo
4. Planning: step finale per decidere se proseguire con un altro loop o procedere alla messa in opera del sistema

Questo modello di sviluppo ha la particolarità di dedicare un'intera fase allo studio dei rischi

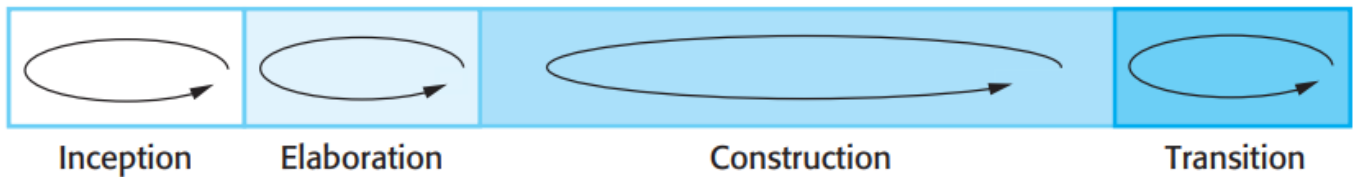
Rational Unified Process (RUP)

Il RUP nasce dall'idea di non considerare un singolo modello per la progettazione di un sistema, in quanto quest'ultimo presenta un singolo punto di vista, che molto spesso non è abbastanza per uno sviluppo efficiente e robusto.

Il modello RUP viene descritto tramite 3 prospettive:

1. Prospettiva dinamica
2. Prospettiva statica

3. Prospettiva pratica



Dall'immagine invece notiamo che le fasi vere e proprie del modello RUP sono 4 e non sono strettamente legate ai processi (come invece avveniva nel modello waterfall) ma piuttosto a dei concetti da seguire, vediamo meglio:

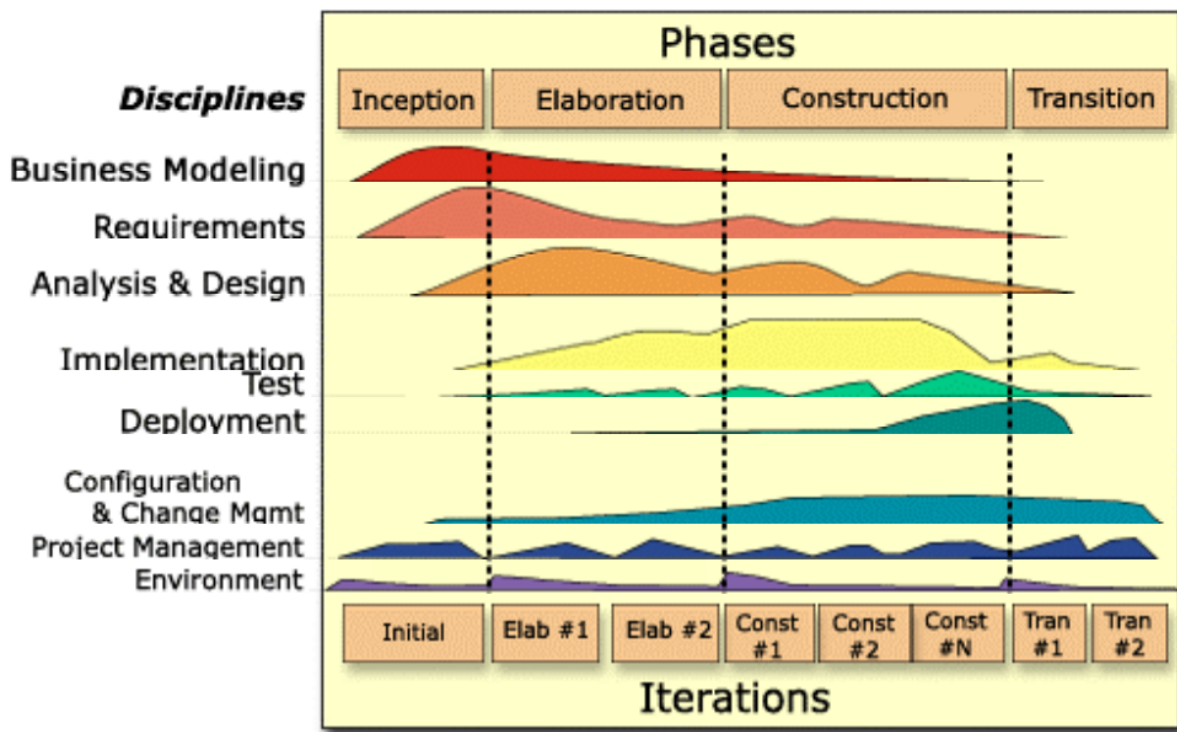
-*Inception*, questa fase si concentra nella scelta di casi in cui il sistema potrebbe essere utile, se da questa fase ci si accorge che il progetto ha poca utilità, il progetto potrebbe venire abbandonato

-*Elaborazione*, vengono definiti i requisiti del sistema

-*Costruzione*, la fase vera e propria di design, sviluppo e programmazione

-*Transizione*, l'ultima fase dove il sistema viene rilasciato al pubblico, viene inoltre monitorato l'andamento del sistema e viene stilata una documentazione

Fasi e tipologie di attività



Lo schema rappresenta i vari effort per ogni fase.

MODELLO AGILE

Il modello agile nasce dall'esigenza di rilasciare le applicazioni in tempo utile, poichè il tempo affinché diventino obsolete si è contratto con il passare degli anni.

Come si può intuire il modello agile utilizza un approccio incrementale, in quanto il sistema deve essere rilasciato rapidamente, sacrificando la robustezza iniziale nei primi rilasci. Il modello agile divenne negli anni 90 così popolare che venne addirittura stilato un manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

I metodi agili sono comunemente preferibili quando le compagnie software non sono molto grandi in quanto la documentazione non è troppo dettagliata e tutti devono essere in grado di capire in che direzione si sta andando, a motivo di ciò, generalmente il metodo agile sfrutta anche l'interazione con il committente, per adeguarsi meglio agli obiettivi che si vogliono raggiungere.

Si potrebbe pensare che il metodo agile sia semplice da implementare, ma molto spesso sviluppare del buon software in poco tempo è estremamente complicato, in particolare le principali difficoltà riguardano.

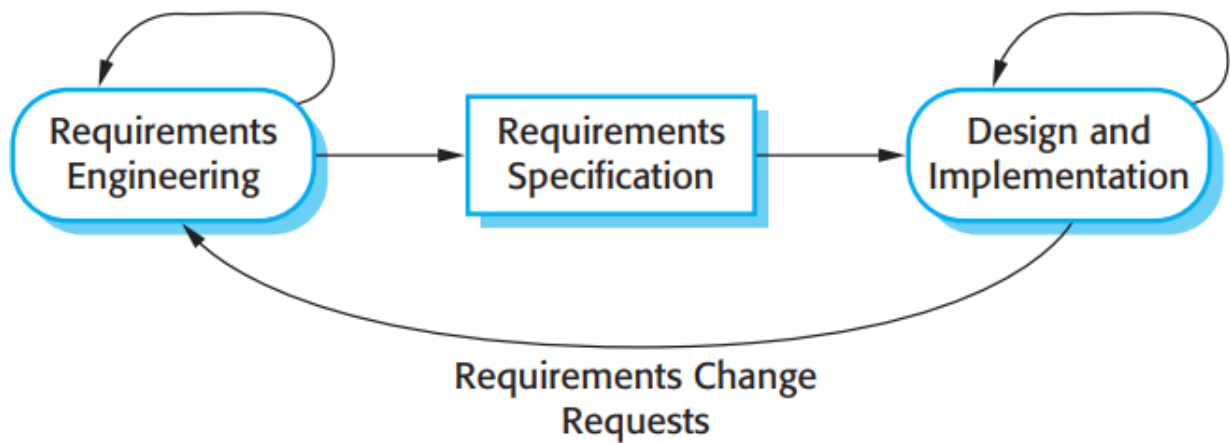
1. La difficoltà di interagire con un committente che, molto spesso, è addirittura poco consapevole di quello che vuole.
2. Il metodo agile richiede una forte cooperazione tra colleghi, una skill umana che spesso non tutti hanno.
3. Se un sistema ha molti stakeholders, ognuno potrebbe avere preferenze diverse, riducendo l'agilità di sviluppo
4. Mantenere un ambiente di lavoro semplice e agile quando il sistema da progettare è molto semplice richiede del lavoro extra.

Modelli Plan-driven o Metodi agili?

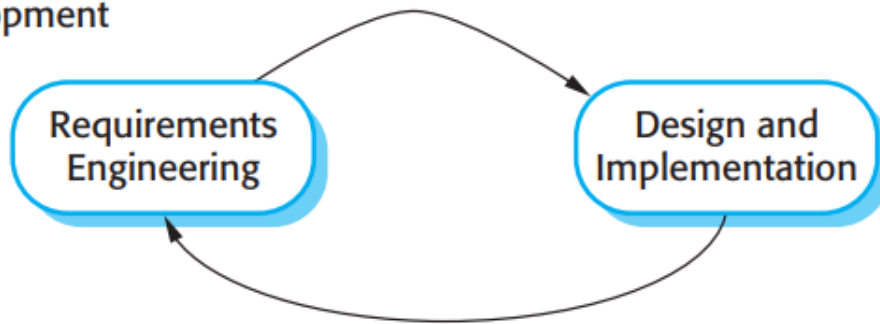
Essendo entrambe le strategie ampiamente utilizzate viene da sé affermare che entrambi i

metodi possono essere efficientemente implementati, resta da chiedersi quando optare per l'uno e quando per l'altro.

Plan-Based Development



Agile Development



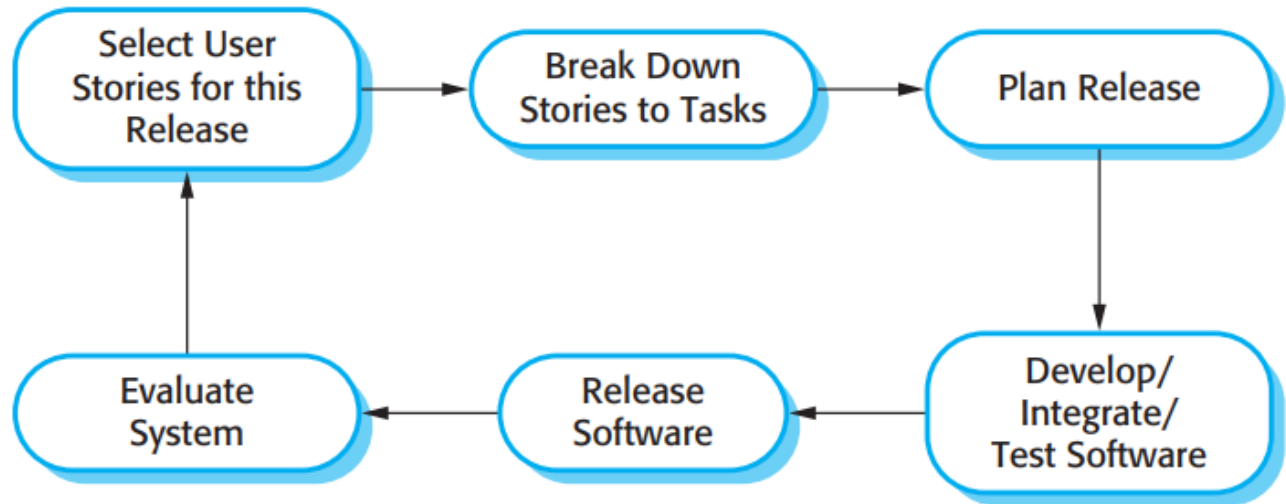
La scelta del modello di sviluppo è uno dei passi iniziali nella costruzione del sistema, il modo migliore per scegliere quello più adatto e rispondere a poche e semplici domande:

1. E' importante avere una documentazione dettagliata e un design molto complesso? Se sì, un approccio plan-driven è d'obbligo.
2. E' utile (e possibile) fare uscire delle versioni incrementali del progetto? Se sì, è fortemente consigliato un approccio agile
3. Quanto grande è il sistema da implementare? I metodi agili sono più efficaci quando il sistema non ha dimensioni eccessiva, in caso contrario è utile pianificare in maniera più accurata le varie fasi.
4. Quanta vita avrà il sistema dopo il rilascio? Se il ciclo di vita del sistema è stimato in anni, la documentazione e la manutenzione è indispensabile, un approccio pianificato è quindi preferibile.
5. Come è organizzato il team di sviluppo? I metodi agili funzionano quando il team si conosce bene ed è consapevole del progetto che si andrà ad implementare

Altre considerazioni riguardano le skill degli sviluppatori, questioni legali e culturali ma girano sempre intorno ai concetti di documentazione e tempi di vita, sono comunque considerazioni qualitative facilmente intuibili anche al di fuori all'ambiente ingegneristico.

Un metodo 'super-agile: l'Extreme Programming (XP)

L'Extreme Programming è il caso più emblematico di metodologia agile, nonché quello più ampiamente utilizzato, il nome riconduce ad un livello di 'agilità estrema', e molto del carico di lavoro viene svolta anche in un giorno lavorativo.



Dallo schema in alto si nota l'introduzione di una espressione che ancora non abbiamo visto negli altri modelli: le User Stories; questa espressione fa riferimento agli scenari d'uso, queste storie sono fondamentali poichè danno un'indicazione su come deve essere implementato il progetto.

Le principali tematiche presenti nell'Extreme Programming si possono riassumere in pochi concetti:

1. Lo sviluppo incrementale è caratterizzato dal rilascio di piccole e frequenti versioni del sistema
2. L'interazione con il cliente è fortemente richiesta, in quanto molto spesso un rappresentante degli stakeholders è presente alle riunioni del team di sviluppo
3. Molto spesso viene consigliato il Pair Programming, ovvero la programmazione di coppia
4. Essendo il numero di versioni rilasciate molto vasto, tipicamente il refactoring e l'integrazione è qualcosa da richiedere continuamente, per evitare degenerazioni del codice.
5. Mantenere la semplicità, anche stavolta grazie al refactoring

Piccola digressione: il refactoring

Dato che il concetto di refactoring è essenziale nella produzione software, richiamo brevemente una definizione di Wikipedia per chi ancora non ha afferrato il concetto:

Con **refactoring** (o **code refactoring**), nell'[ingegneria del software](#), si indica una "*tecnica strutturata per modificare la struttura interna di porzioni di [codice](#) senza modificarne il comportamento esterno*" applicata per migliorare alcune caratteristiche non funzionali del [software](#) quali la [leggibilità](#), la [manutenibilità](#), la [riusabilità](#), l'[estensibilità](#) del codice nonché la riduzione della sua complessità, eventualmente attraverso l'introduzione a posteriori di [design pattern](#). Si tratta di un elemento importante delle principali metodologie emergenti di sviluppo del software (soprattutto [object-oriented](#)), per esempio delle [metodologie agili](#), dell'[extreme programming](#), e del [test driven development](#).

Il refactoring è un elemento integrante di molti processi di sviluppo fortemente basati su [test automatici](#); per esempio, lo [sviluppo basato su test](#) (TDD) prevede una fase (obbligatoria ed esplicita) di refactoring al termine di ogni ciclo di modifica. Fra i due concetti esiste infatti un legame molto stretto: rieseguire eventuali test automatici al termine di ogni micromodifica fornisce infatti un più alto grado di confidenza che non siano stati introdotti errori; questo consente di prendere in considerazione anche modifiche particolarmente pericolose (come lo spostamento di codice fra classi o la modifica delle relazioni di [ereditarietà](#)).

Il Testing nell'Extreme Programming

Chi sceglie l'Extreme Programming non deve ignorare la fase di testing, senza la quale questo approccio risulterebbe inefficace, a differenza di altri metodi la fase di testing avviene (anche) prima di aver scritto del codice, in questo caso si parla di Test First Development, gli altri testing riguardano l'applicabilità degli scenari, e i classici test su sviluppo e produzione.

Pair programming

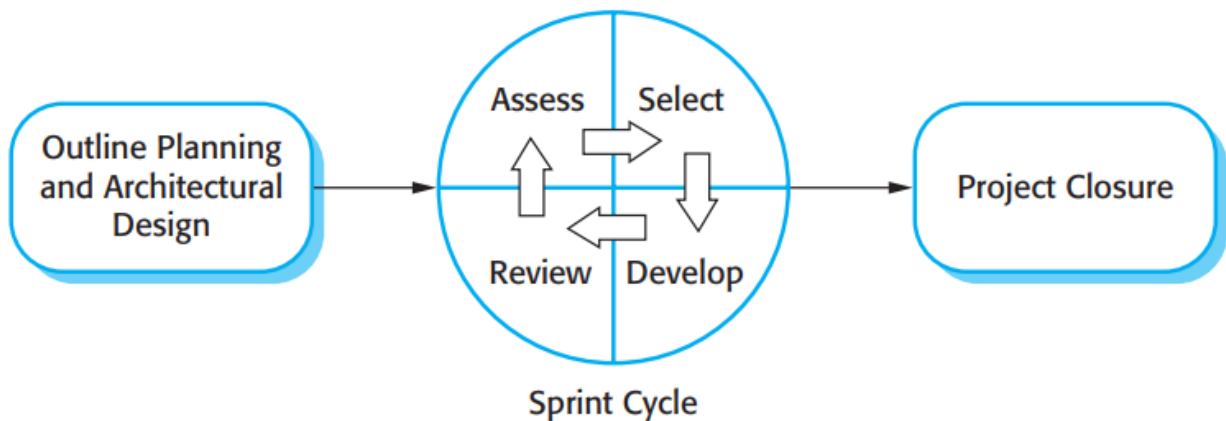
Abbiamo visto che la programmazione estrema incoraggia alla programmazione di coppia, ma come funziona, e soprattutto, perché funziona? Vediamo di riassumere le tematiche più importanti:

1. Il Pair Programming abbraccia la filosofia di 'collective ownership' che presuppone che ognuno è responsabile e proprietario del codice scritto
2. Il codice scritto in 2 è molto più correggibile poichè psicologicamente una persona è più incline a criticare il codice altrui piuttosto che il proprio
3. E' più veloce ed efficiente poichè la collaborazione aiuta a sbrogliare situazioni di stallo.

Gestire i processi agili: lo Scrum

Quando si pensa ai processi agili spesso si pensa a qualcosa di poco organizzato, che non richiede competenze manageriali e che viene improvvisato alla giornata.

L'approccio Scrum è un processo agile in grado di organizzare il lavoro agile in step prestabiliti, evitando l'uso dell'Extreme Programming e del Pair Programming



La prima fase organizza il lavoro ponendosi degli obiettivi e organizzando il design.

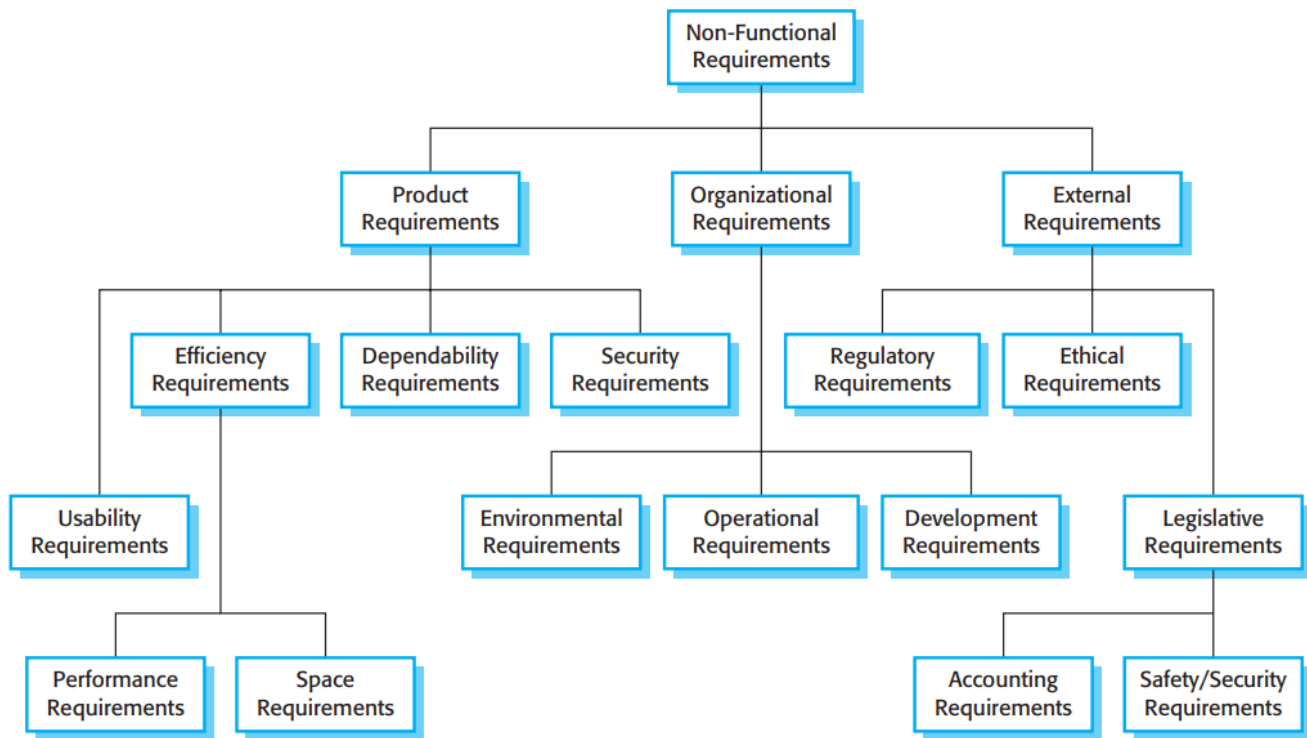
La seconda fase, detta sprint cycle, è il lavoro vero e proprio dove avviene lo sviluppo software, uno sprint dura in genere uno o due settimane al seguito del quali si organizza un meeting per discutere i progressi fatti, una volta stabilito che il lavoro è completo si passa alla fase finale dove si ha la messa in opera del progetto.

Uno sprint è caratterizzato dalle solite 4 fasi in cui si discute, seleziona, sviluppa e testa il progetto e alla fine di ogni ciclo lo stakeholder viene aggiornato sui progressi.

Dal punto di vista organizzativo non esiste il 'project manager', ma piuttosto si introduce il termine di Scrum Master, che è un coordinatore che facilita l'organizzazione tra i membri del team.

Una peculiarità di questo approccio è che i meeting sono giornalieri, durano poco e convenzionalmente si sta in piedi.

**I requisiti di un software



Vediamo sopra un esempio di requisiti non funzionali, e notiamo che la numerosità di questi requisiti indica che tralasciarli in toto comporta uno sviluppo disastroso del software, a questi requisiti, che sono chiamati così perché non dipendono dalle caratteristiche intrinseche del sistema, si associano i requisiti funzionali, che invece descrivono come quel particolare sistema dovrebbe funzionare.

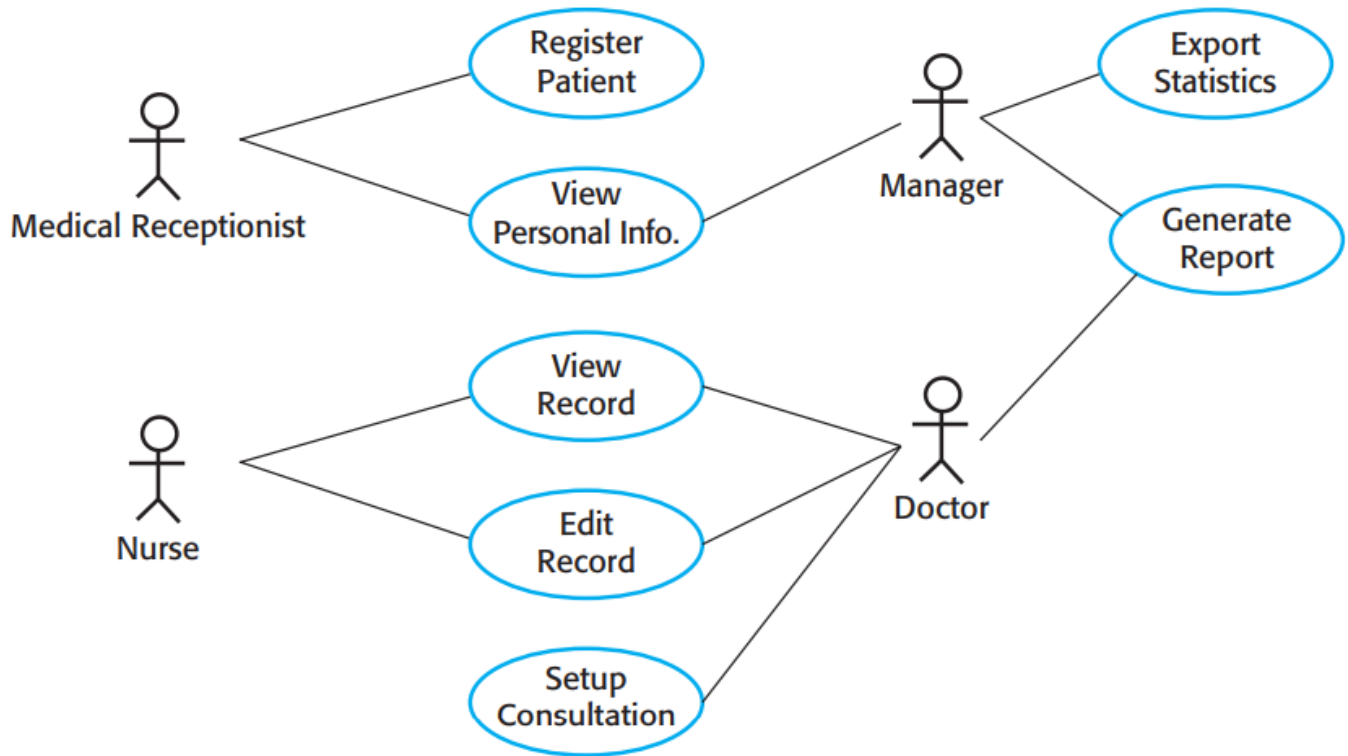
Specifiche dei requisiti, scenari e interviste

Per indicare i requisiti che un sistema deve avere, va innanzitutto scegliere un linguaggio che possa racchiudere i concetti da esprimere in modo tale che siano comprensibile a tutto il team di sviluppo. A questo scopo si creano degli scenari d'uso e si provano a descrivere usando un linguaggio che incorpora notazioni matematiche, operazioni matematiche e linguaggio naturale, cercando quando possibile di mantenere dei toni semplici.

Gli scenari assumono esempi di vita reale, in cui si cerca di descrivere le funzionalità del sistema, il dominio applicativo, cosa potrebbe andare storto, e in quali modi il cliente potrebbe voler utilizzare il sistema. Per quest'ultima ragione sono a volte richieste interviste agli stakeholders.

Casi d'uso

I casi d'uso sono fondamentali nella stesura dei requisiti che un software corretto dovrebbe avere, si è soliti rappresentare i casi d'uso attraverso il linguaggio UML, in particolare attraverso gli Use Case Diagram.



I protagonisti dei casi d'uso, generalmente le persone che devono utilizzare il sistema, vengono detti attori, ed ogni attore può dover utilizzare più casi d'uso.

Possiamo quindi definire i casi d'uso come l'interazione tra attore e sistema.

System modelling

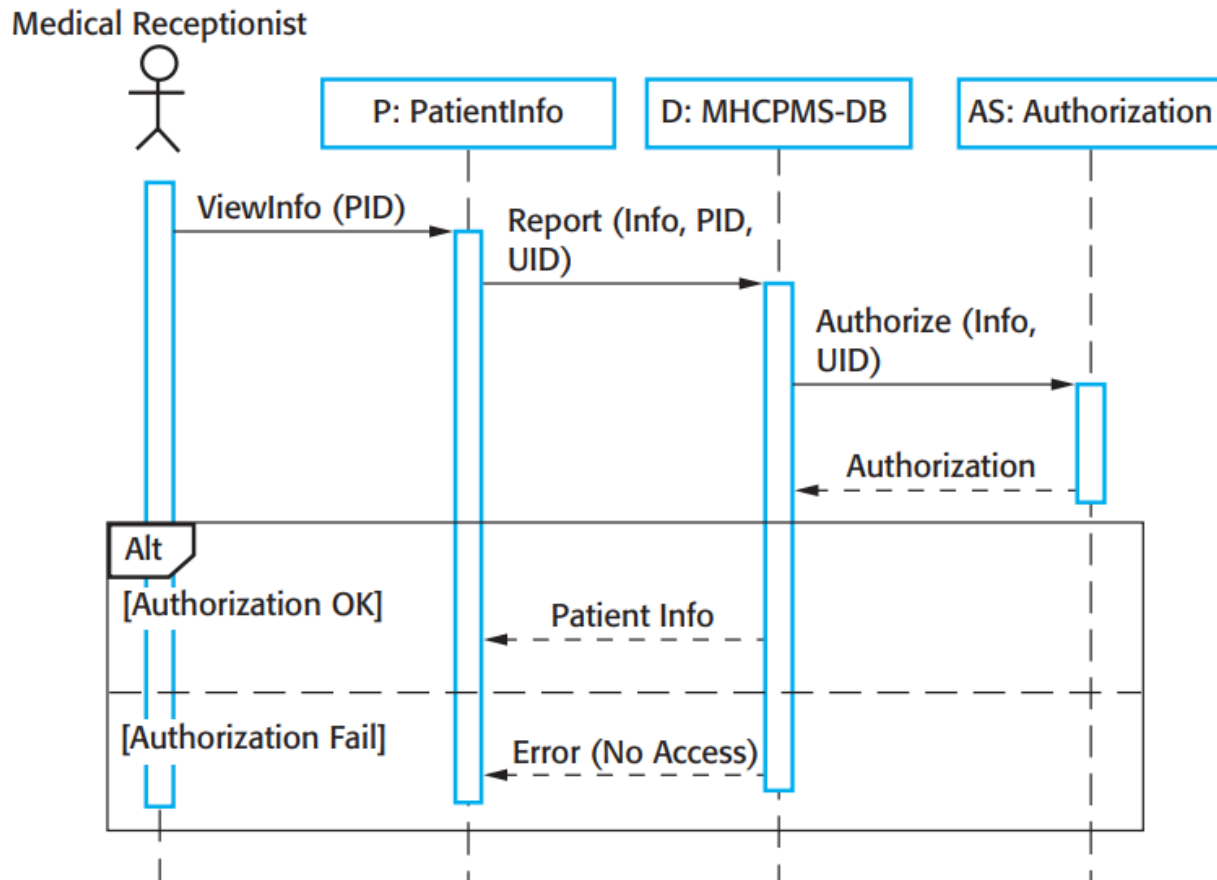
Il system modelling è il processo che mira ad astrarre il sistema da sviluppare in dei modelli comprensibili e validi. La stragrande maggioranza di modellazione di sistemi avviene grazie ad UML, un linguaggio molto versatile in grado di rappresentare tutto ciò a che fare con del software.

UML ha infatti così tanti diagrammi che un sondaggio del 2007 ha indicato come la maggior parte dei sistemi viene descritto con una piccola percentuale dei diagrammi UML disponibili, in particolare i diagrammi più utilizzati sono:

1. Activity diagrams: che rappresenta le attività che intercorrono in un processo
2. Use case diagrams: come abbiamo visto, rappresenta come un utente interagisce con il sistema
3. Sequence diagrams: che rappresenta sia come l'utente interagisce con il sistema, che come i componenti del sistema interagiscono tra di loro
4. Class diagram, che mostra come gli oggetti e le classi sono associati tra di loro
5. State diagrams, che mostra come il sistema reagisce a eventi (sia interni che esterni)

Più in dettaglio : i Sequence Diagrams

Come abbiamo già detto i sequence diagram mostrano interazioni tra componenti, e interazioni tra componenti e utenti, su un particolare caso d'uso.



Vediamo in questo caso come modellare il caso d'uso del receptionist di uno studio medico che deve poter visualizzare le info del paziente e autorizzare eventuali azioni (ricovero, rilascio, visite).

Vediamo come è possibile modellare anche situazioni non previste e generare messaggi d'errore.

Come per quasi tutti i diagrammi UML, non è sempre utile rappresentare ogni singola interazione, considerando che il compito del design ha il ruolo di snellire la visualizzazione del progetto, e mettere troppa carne al fuoco potrebbe appesantire di troppo la fase di progettazione.

Più in dettaglio: i Class diagrams

I Class Diagrams ricoprono il 90% dell'uso che si fa di UML, poichè sono principalmente utilizzati nel rappresentare i pattern usati nella programmazione.

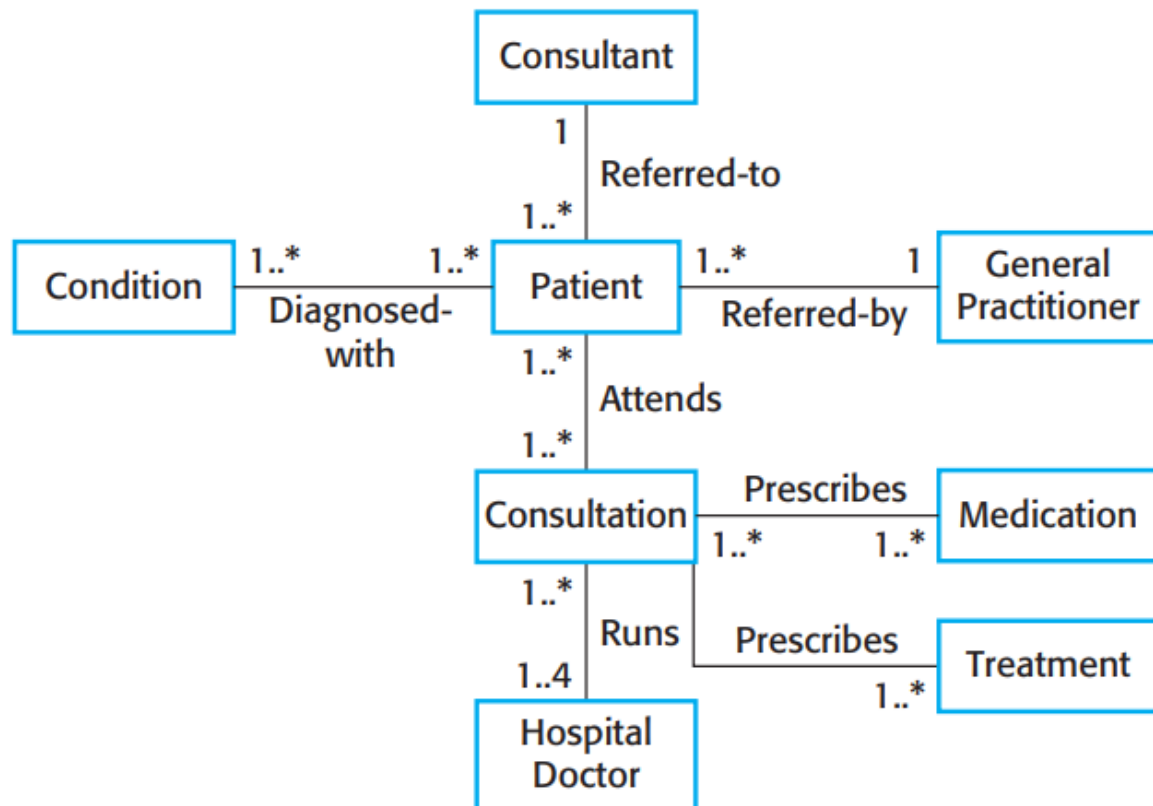
I class diagram sono un tipo di modellazione strutturale, che descrive quindi l'interazione tra componenti, in questo caso specifico si parla di interazioni tra classi e interazioni tra oggetti. Ciò che viene fatto quando si sceglie di utilizzare i Class Diagram è rappresentare gli oggetti 'del mondo reale' in classi e vedere come questi oggetti dovrebbero interagire tra di loro.

Vediamo il caso più semplice di Class Diagram che rappresenta la relazione che intercorre tra il

paziente e il suo record.

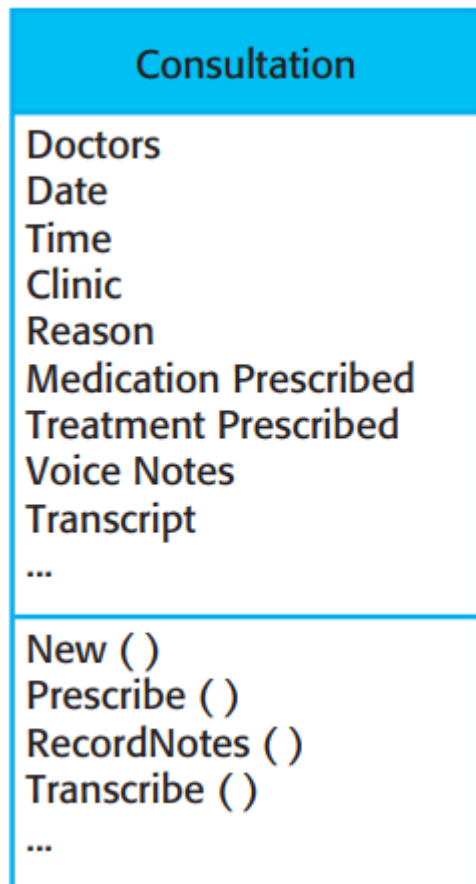


Vediamo, con un caso più complicato, notando come il Class Diagram è in grado di rappresentare dipendenze e relazioni multiple



Quindi è tutto qua?

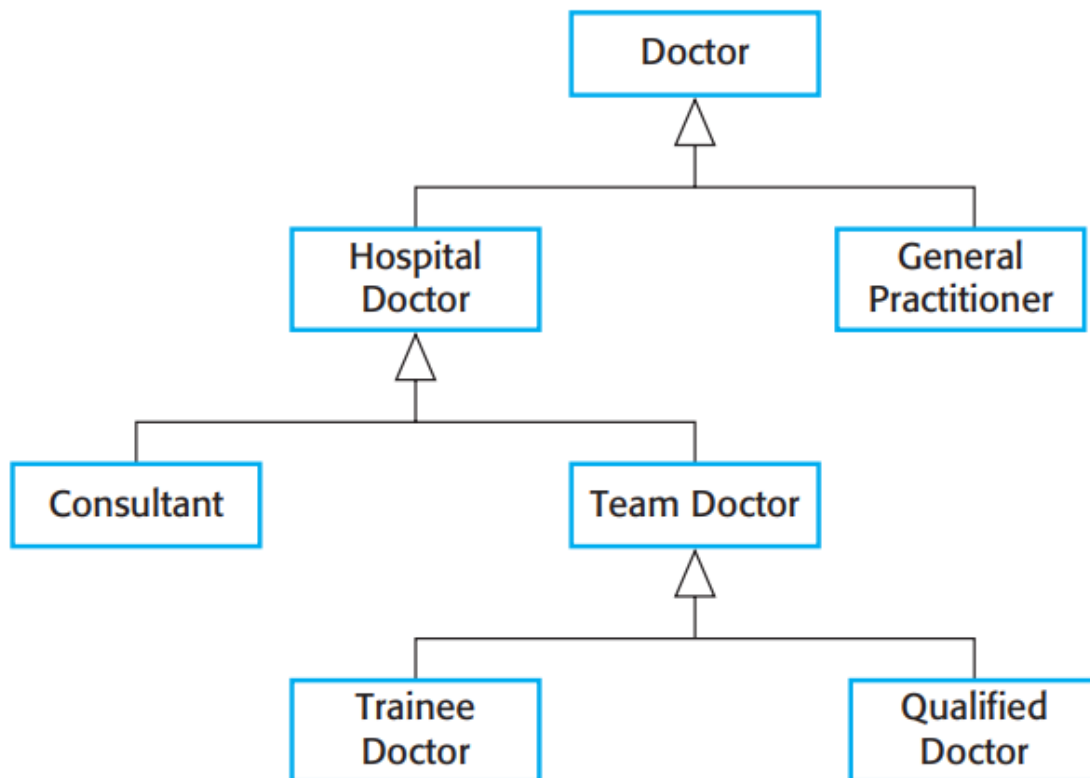
In realtà la rappresentazione tramite Class Diagram avviene attraverso diversi livelli di dettaglio, ad esempio un classe può essere rappresentate semplicemente con il nome dell'oggetto che sta cercando di rappresentare, o molto più comunemente, con l'insieme di attributi e metodi (operazioni) che porta con sè. Vediamo un esempio.



Alcuni standard di un Class Diagram

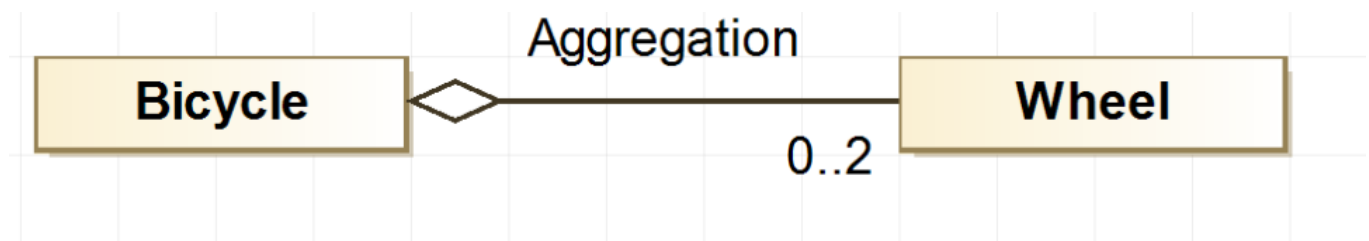
Generalizzazioni

La generalizzazione viene rappresentata con una freccia che punta alla classe generale, è utile quando si vuole rappresentare una gerarchia tra le classi ed è una tecnica ampiamente utilizzata sia nel design sia nella programmazione. Vediamo un esempio.



Le aggregazioni

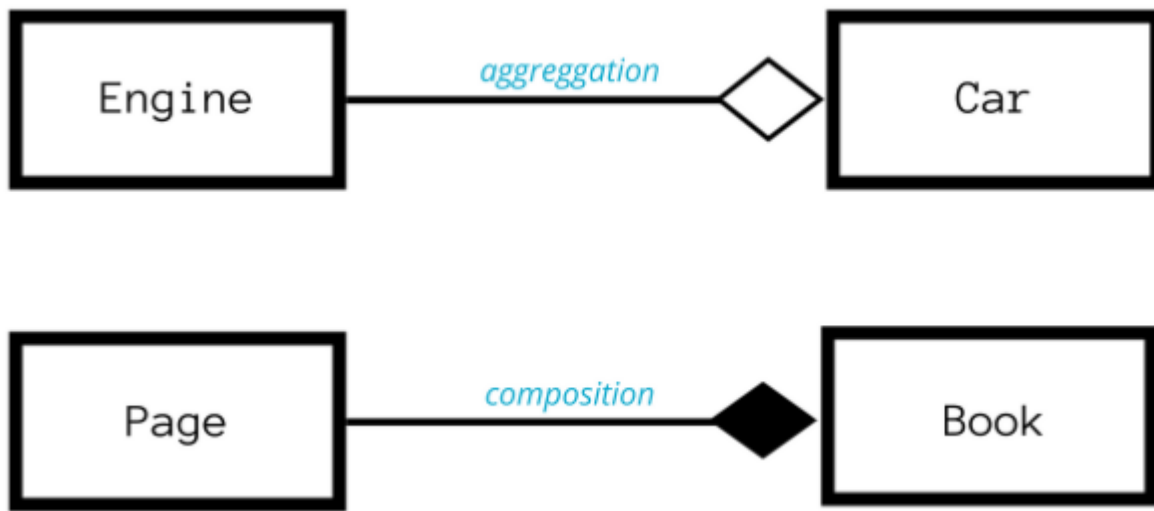
Generalmente per spiegare il concetto di aggregazione si utilizzano diversi esempi pratici di vita reale: le dita che compongono una mano, i punti di un cerchio, le slide che fanno parte di un PowerPoint, in tutti questi esempi abbiamo l'intero (the whole) e le sue componenti (the parts). In UML si è solito rappresentare le aggregazioni in questo modo.



In questo caso si sta modellando implicitamente che la ruota fa parte della bicicletta indipendentemente dall'esistenza della bicicletta.

La composizione

Quando non si suppone l'esistenza di un componente quando l'intero non esiste si parla di una composizione, che può essere considerata quindi come la versione forte dell'aggregazione. Vediamo che la differenza di notazione consiste graficamente nel riempimento del rombo nel caso della composizione.



State diagram

Gli State Diagram modellano processi event-driven attraverso l'uso di state charts, gli State Diagram risultano quindi molto utili quando stiamo modellando un sistema che cambia ad ogni azione compiuta. Un esempio molto utile è riportato di seguito e rappresenta una modellazione di un forno a microonde che cambia il suo stato in base ad alcuni eventi elementari come la fine cottura o l'apertura - chiusura del forno.

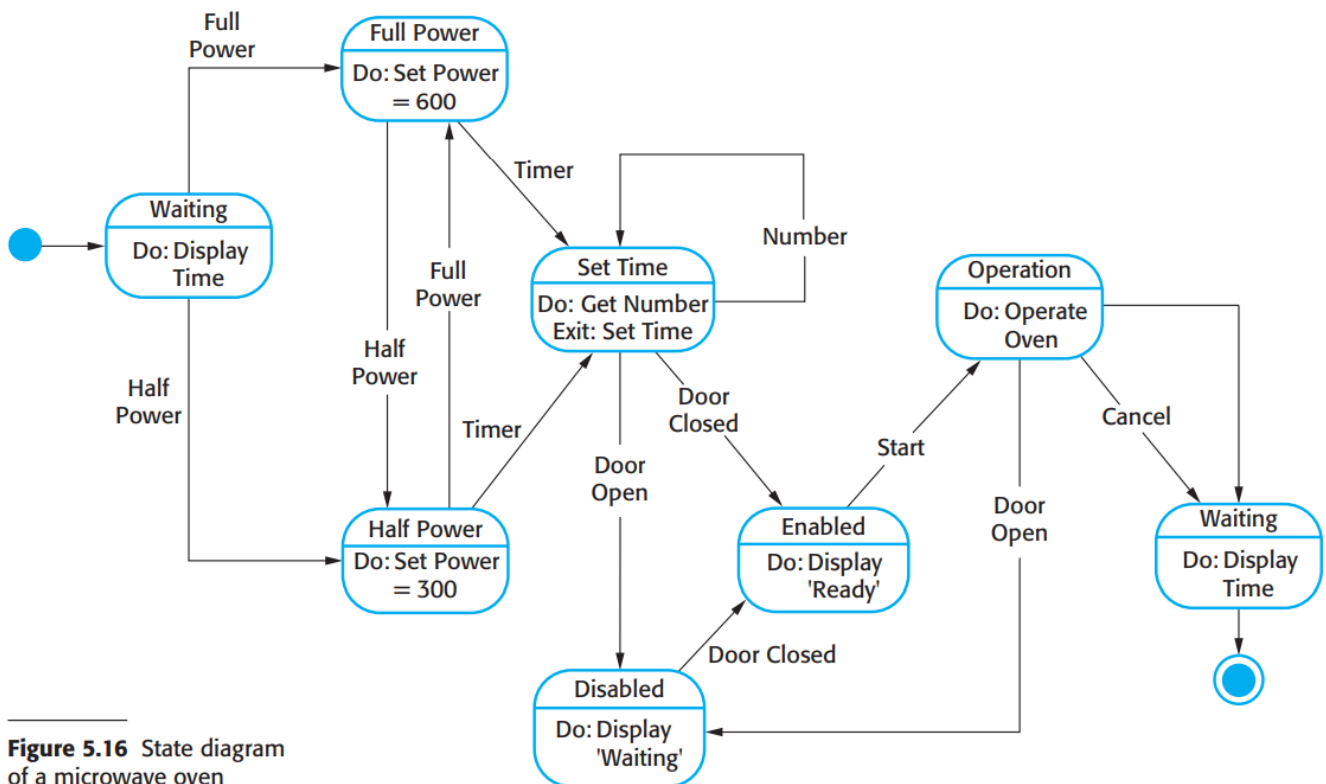


Figure 5.16 State diagram of a microwave oven

Si nota che quello che si sta modellando è una concezione personale di come dovrebbe funzionare il forno, in particolare si vuole settare la potenza massima, il tempo di waiting il display del tempo rimanente ecc...

L'importanza del design architetturale nella progettazione software

Il design architetturale si preoccupa di descrivere come un sistema dovrebbe essere organizzato e strutturato, una buona architettura del software incide su caratteristiche non funzionali essenziali per la progettazione quali performance, robustezza, e manutenibilità. Esplicitando l'architettura del progetto servendosi di un design ben realizzato offre diversi vantaggi tra i quali:

1. Comunicazione con gli stakeholder
2. Analisi del sistema
3. Riutilizzo a larga-scala

Durante il processo di costruzione di un'architettura software il progettista incorre a decisioni importantissime per l'esito del progetto che vengono formulate in base all'esperienza del progettista stesso e alla sua conoscenza sul progetto che dovrà realizzare. Alcune domande che un buon progettista dovrebbe saper porre all'attenzione sono:

1. Esiste già un'architettura generica che funge come template che si vuole realizzare?
2. Come verrà distribuito il sistema attraverso i core e i processori?
3. Quali stili e pattern dovrebbero essere utilizzati?
4. Come posso decostruire i vari componenti del sistema in sottocomponenti?
5. Qual è il modo migliore per documentare il progetto?

Lo stile architetturale del progetto deve inoltre tenere in considerazione alcuni requisiti per i quali si vuole/si dovrebbe dare priorità come:

1. Performance
2. Sicurezza
3. Disponibilità (Availability)
4. Manutenibilità (Maintainability)

Intro ai Design Patterns

Con la necessità di implementare un design orientato agli oggetti, magari tramite l'ausilio di UML, ci sono alcuni passi essenziali da tenere in considerazione:

1. Definire il dominio applicativo
2. Elaborare il design architetturale
3. Capire quali oggetti considerare nel sistema
4. Sviluppare modelli di design
5. Definire le interfacce necessarie

Principali problemi di implementazione

L'implementazione software è la fase cruciale in cui si crea del software che deve essere eseguito

(ed eseguibile). Alcuni aspetti da tenere considerati durante la programmazione, ma che vengono sistematicamente ignorati da molti programmatori riguardano:

1. Il riuso: il codice che un buon programmatore scrive deve essere quanto più possibile riutilizzabile
2. Gestione delle configurazioni: Durante il processo di sviluppo vengono create diverse versioni per uno stessa unita, se non si tiene traccia delle varie versioni potrebbe inserire configurazioni non sincronizzate di unita diverse del progetto
3. Sviluppo host-target, capita molto spesso che il sistema funzioni sulla macchina dello sviluppatore ma non su quella del cliente, questo perchè le specifiche del target di riferimento vengono trascurate

Più in dettaglio: il riuso

Il riuso del software è apprezzabile a livelli molto diversi tra loro, abbiamo:

1. Il livello di astrazione: a questo livello ciò che non viene riutilizzato non è il codice in sè, ma la conoscenza maturata da esso
2. Livello oggetti: in questo caso riusiamo gli oggetti presenti nella nostra libreria
3. Livello componenti: I componenti sono collezione di oggetti, con la stessa logica possiamo utilizzare collezione di oggetti memorizzate nelle librerie
4. Livello sistema: riutilizzare il sistema significa riutilizzare l'intera applicazione, solitamente ciò che appare utile è riusare configurazioni che, con qualche modifica potrebbe fare molto comodo al nostro progetto.

Piu in dettaglio: la gestione delle configurazioni

La gestione delle configurazione gioca un ruolo essenziale nei sistemi per le quali sono necessari continue modifiche e continui rilasci di versioni differenti, con il termine di gestione di configurazione si intende la configurazione di 3 attività principali

1. Gestione della versione
2. Integrazione del sistema che aiuta lo sviluppatore a tenere traccia di quale configurazione è usata in uno specifico sistema
3. Problemi di tracking, la fase di supporto che tiene traccia di tutti gli errori e i bug nelle varie configurazioni

Più in dettaglio: sviluppo host target

La maggior parte del software è basato sui modelli host-target, la strategia per sincronizzare il software su una macchina diversa da quella che su cui si sta programmando consiste nell'uso dei simulatori che hanno il ruolo di simulare l'hardware su cui dovrebbe girare l'applicativo.

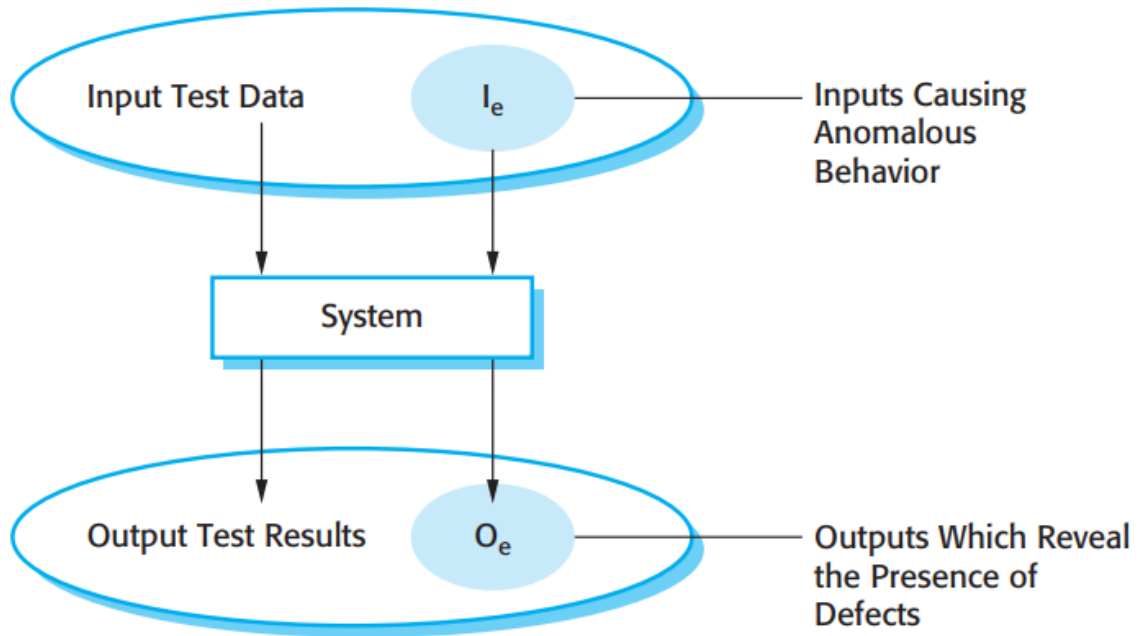
Il testing software

Il testing software è progettato con lo scopo di valutare se il sistema funziona come dovrebbe o presenta errori di struttura, bug di codice o pecca di feature che avrebbero dovuto fare parte del sistema, il processo di testing aiuta anche il committente a capire come utilizzare il sistema.

Una frase celebre di Dijkstra afferma:

Testing can only show the presence of errors, not their absence

Questo a significare come non si può mai essere completamente sicuro di aver rimosso totalmente gli errori del sistema, soprattutto quando questo risulta fortemente interfogliato.



Rimanendo in tema di citazioni Boehm chiarisce un dubbio sulla differenza tra validazione e verifica

- Validation: Are we building the right product?'
- Verification: Are we building the product right?'

Test di sviluppo

I test di sviluppo vengono generalmente svolti dai programmatori che scrivono il codice, poiché sono i candidati migliori a conoscere le eventuali criticità del sistema, la granularità del sistema si esplica sia a livello unitario (Unit Testing), sia a livello delle singole componenti (Component Testing), sia a livello dell'intero sistema (System Testing).

Come abbiamo visto i test sono fondamentali, ma molto costosi in termini sia di budget che di tempo, è quindi utile prefissarsi alcuni unit test cases, stabilendo per ogni test cosa ci aspettiamo che debba succedere, e fare una stima di ciò che potrebbe andare storto, ben consapevoli che una percentuale di incertezza sarà sempre presente.

Gli errori più comuni riguardano il testing sulle componenti, in particolare su sistemi complessi, supponendo che due o più componenti interagiscono fra di loro in un'interfaccia comune gli errori che possono presentarsi sono:

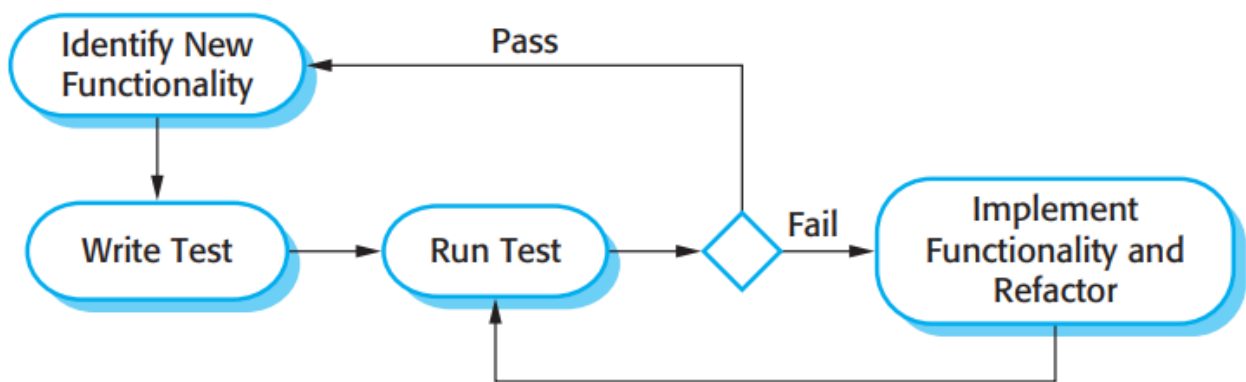
- Uso improprio dell'interfaccia
- Errori di timing

Per quando riguarda il system testing, il passo fondamentale è capire come integrare le varie componenti e inferire il comportamento del sistema, questo passaggio non è semplicissimo poichè generalmente le varie componenti vengono gestite da team di sviluppo differenti.

Un caso particolare: lo sviluppo test-driven

Lo sviluppo basato sui test è un approccio che interfoglia la scrittura del test e il suo testing, questa situazione si presenta molto spesso nello sviluppo incrementale, dove ogni incremento viene prima testato, e solo successivamente si riprende con la scrittura del codice.

Lo schema che segue illustra molto bene questa situazione



Testing di rilascio

Normalmente questo tipo di test è inteso per gli utenti finali, che fruitori del sistema finale, in sistemi molto critici comunque il testing può essere svolto in contemporanea da un team estraneo allo sviluppo del sistema, ma che ne conosce le funzionalità.

Lo scopo principale del testing di rilascio è capire se il sistema è pronto o meno all'uso, supponendo ovviamente che tutti i test precedenti non abbiano prodotto errori significativi.

Test sulle performance

Una volta superati i vari test, e integrate correttamente le componenti di un sistema, il test sulle performance si basa sull'eseguire molte volte l'applicativo, con carichi diversi, per verificarne l'affidabilità e la robustezza.

La filosofia è quindi quella di stressare il sistema in maniera incrementale, per comprenderne i limiti e cercare ulteriori ottimizzazioni.

User testing

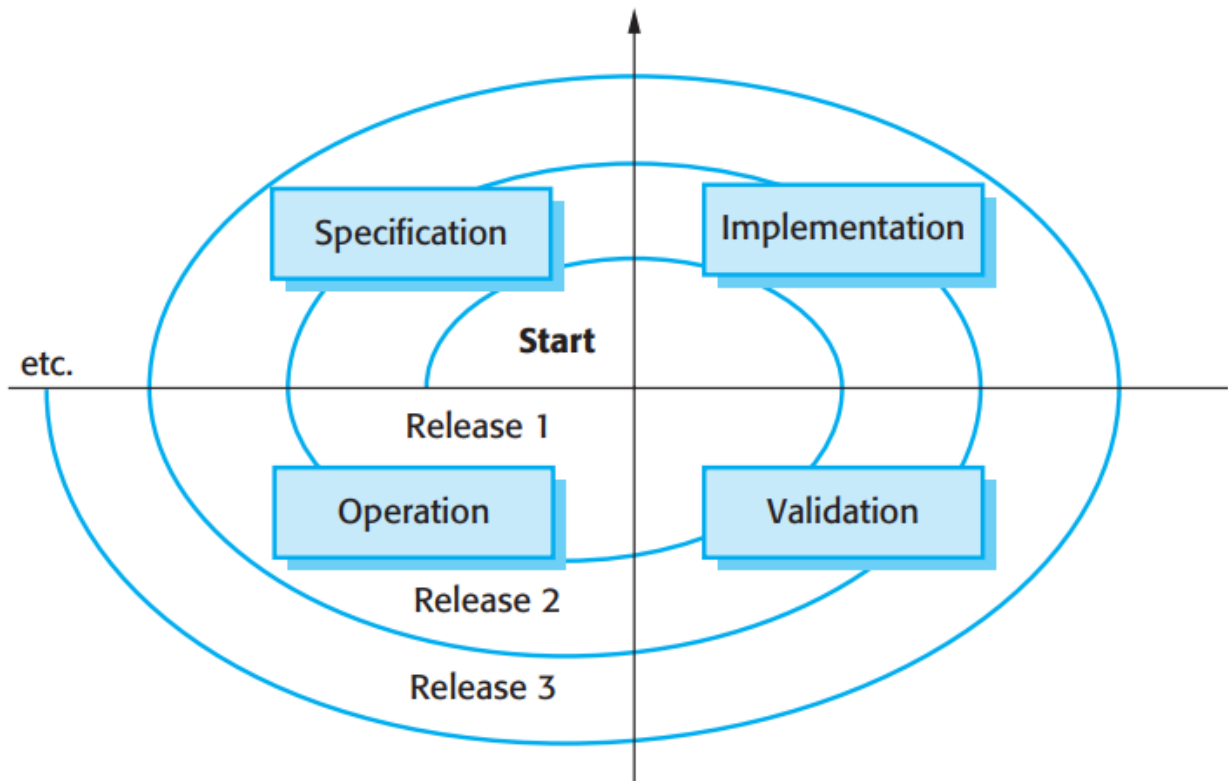
L'user testing si compone in 3 fasi

- Alpha test, riservato ad una cerchia ristretta di tester
- Beta test, che avviene con il primo rilascio agli utenti finali
- Acceptance test, riservato ai clienti, che danno un feedback sull'affidabilità del sistema

Evoluzione del software

Come abbiamo visto, lo sviluppo del software non si ferma una volta rilasciato, ma molto spesso viene modificato nel tempo, per meglio rispondere alle esigenze dei clienti.

Si potrebbe pensare che il budget sia prevalentemente destinato al processo di sviluppo, invece si stima che circa il 90% dei costi risieda proprio nell'evoluzione del software post-rilascio questo perchè generalmente un buon software ha un ciclo di vita che arriva fino a 30 e più anni.

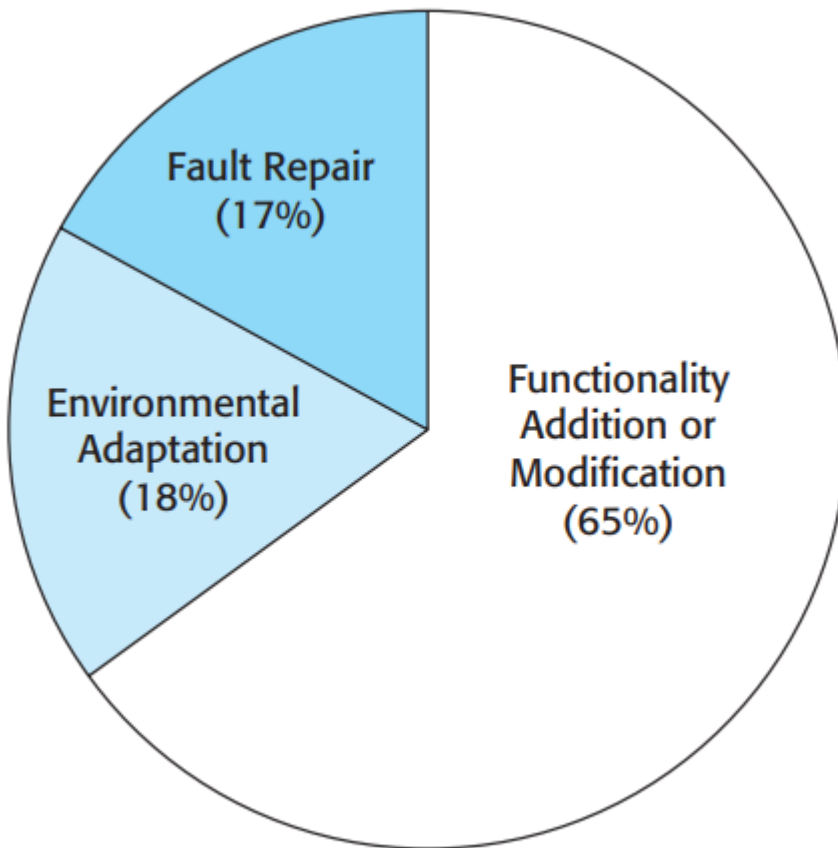


Da questo grafico a spirale notiamo come, nonostante l'applicazione sia stata già rilasciata, le successive versioni richiedono ancora lo studio di specifiche implementazioni, validazione ecc...

Molto spesso ciò che viene richiesto dagli sviluppatori nel post-rilascio dell'applicazione è la gestione di cambiamenti sul business dell'applicazione ed errori improvvisi che possono occorrere da un momento all'altro, in questi casi bisogna essere 'agili' a rispondere a situazioni non pianificate.

Manutenzione software


Strettamente correlato con il concetto di evoluzione software, la manutenzione è il processo di gestione del sistema una volta rilasciato. Anche in questo caso quando si parla di manutenzione ci si riferisce alla risoluzione di problemi che occorrono nel sistema (faults repair), adattamento all'ambiente in continua evoluzione e aggiunta di funzionalità richieste dal mercato (o dal cliente).



Dal grafico a torta notiamo che l'effort principale è rivolto all'aggiunta di nuove funzionalità, in quanto la gestione dei guasti potrebbe essere già stata 'minimizzata' nel processo di sviluppo pre-rilascio.

Una giustificazione da fare sul costo così alto di manutenzione, oltre quella relativo al lifespan di un buon software, risiede nel fatto che, una volta terminato il progetto, il team di sviluppo si divide, e concentra le proprie attenzioni su un altro progetto, facendo venire meno la coordinazione e il tempo messo a disposizione per il sistema già rilasciato.

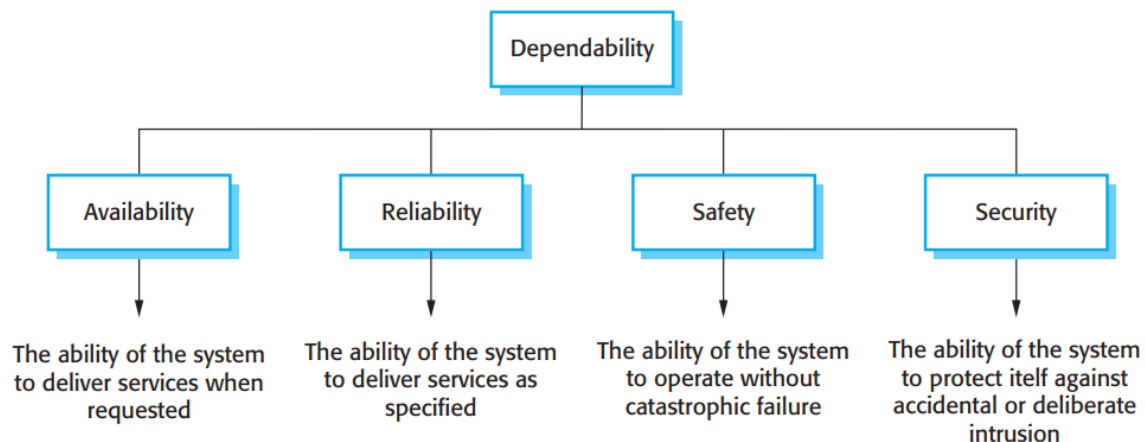
Anche il refactoring entra in gioco, usato spesso per prevenire i costi di manutenzione, cito interamente Sommerville che spiega molto bene questo passaggio

 Refactoring is the process of making improvements to a program to slow down degradation through change (opdyke and johnson, 1990). it means modifying a

program to improve its structure, to reduce its complexity, or to make it easier to understand. refactoring is sometimes considered to be limited to object oriented development but the principles can be applied to any development approach. when you refactor a program, you should not add functionality but should concentrate on program improvement. you can therefore think of refactoring as 'preventative maintenance' that reduces the problems of future change.

Affidabilità (dependability) del software

L'affidabilità è un concetto molto vasto e sfocato nella produzione di software e può essere riassunto scomponendolo in 4 componenti distinte:



A queste 4 fondamentali caratteristiche descrivono l'affidabilità del sistema, se ne possono affiancare quelle relative alle proprietà del sistema come:

- Survivability
- Maintainability
- Repairability
- Error tolerance

Più in dettaglio: Availability e Reliability (Disponibilità e Attendibilità)

Queste due caratteristiche del sistema sono facilmente quantificabili, in quanto, in seguito a dei test, si valuta una percentuale per entrambe; ad esempio, un sistema avrà un availability del 90% se 90 volte su 100 è in grado di offrire del servizio richiesto, analogamente un sistema sarà affidabile al 90% se 90 volte su 100 produrrà l'output richiesto

Più in dettaglio: Safety

Nei sistemi safety-critical le operazioni devono sempre essere sicure, in quanto operazioni non sicure in questi sistemi potrebbero danneggiare le persone che le usano (Software per gestire voli, macchine, medicine ecc...).

Distinguiamo due casi quando si parla di safety:

- Malfunzionamento hardware che potrebbe provocare infortuni a chi utilizza il servizio
- Malfunzionamenti di secondo tipo, ad esempio quando il software genera una prescrizione di un medicinale con un dosaggio estremamente basso/alto

Piu in dettaglio: Security

La sicurezza del sistema indica la predisposizione a difendersi da attacchi esterni, anche in questo caso l'importanza dell'affidabilità riguarda il tipo di sistema che si vuole implementare, la sicurezza è fondamentale nei sistemi di tipo militare o tutti quei sistemi che presuppongono lo scambio di informazioni sensibili, le minacce alla sicurezza si classificano in 3 tipi:

- Minacce alla confidenzialità del sistema (password e dati sensibili)
- Minacce software (virus)
- Minacce alla disponibilità del sistema (DDOS)

Sistemi embedded

Cos'è un sistema embedded? Da Wikipedia:

Un **sistema *embedded*** (lett. "incorporato" o "incassato") o **sistema integrato** identifica genericamente tutti quei [sistemi elettronici di elaborazione](#) a [microprocessore](#) progettati appositamente per un determinato utilizzo ([special purpose](#)), ovvero non riprogrammabili dall'utente per altri scopi, spesso con una [piattaforma hardware](#) *ad hoc*, integrati nel sistema che controllano e in grado di gestirne tutte o parte delle funzionalità richieste.]

I sistemi embedded sono importantissimi poiché adesso alla parte hardware viene spesso affiancata una parte software, infatti esistono molti più sistemi embedded rispetto ad altri tipi di sistema.

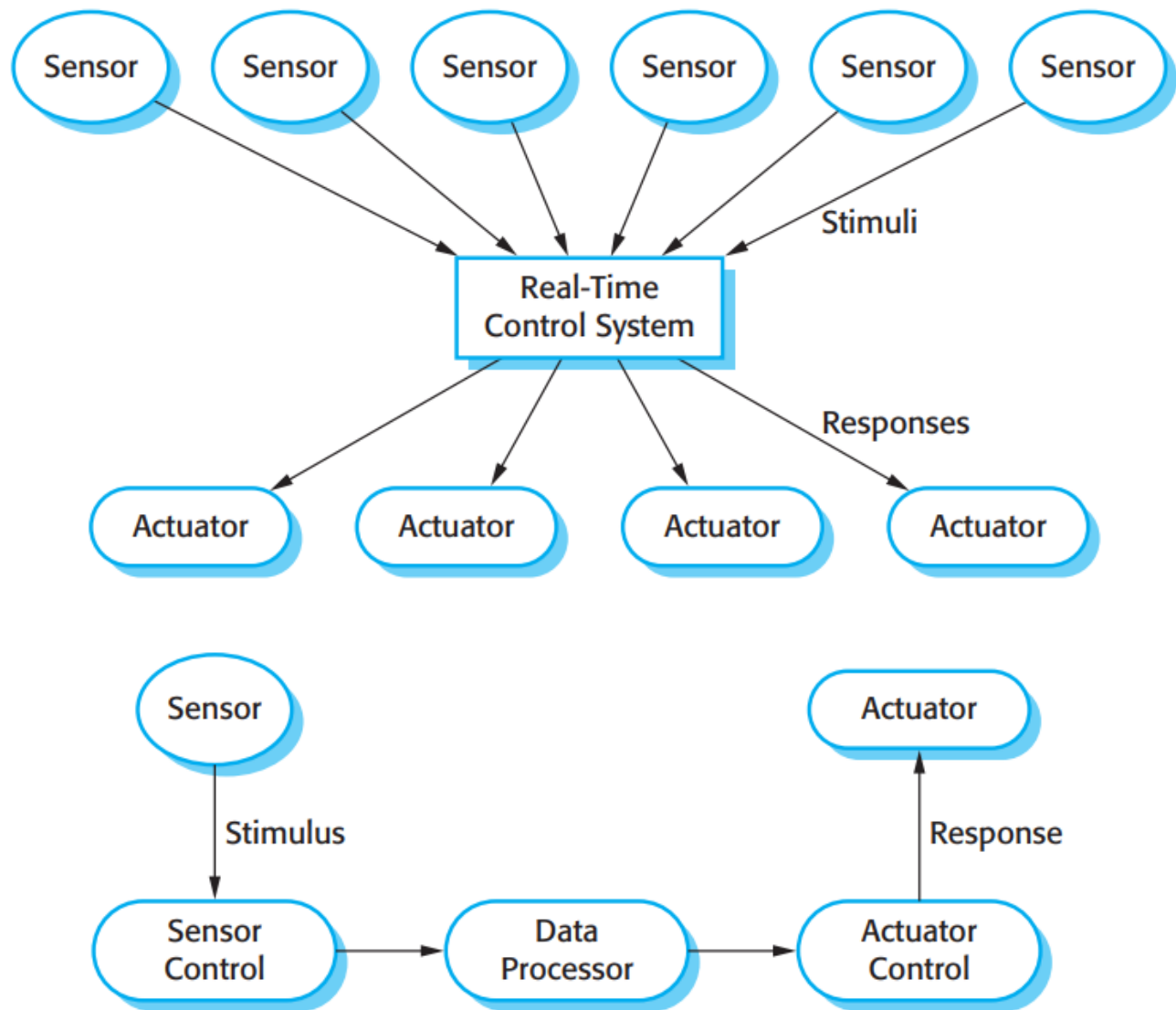
Di seguito alcune caratteristiche dei sistemi embedded:

1. I sistemi embedded generalmente sono sempre in azione quando l'hardware è acceso
2. Le interazioni con l'ambiente sono spesso incontrollabili, quindi una buona pianificazione dei sistemi embedded tiene traccia delle varie situazione ed è pronta a gestire in maniera concorrente le varie situazioni, con bari processi che vengono svolti in parallelo
3. Interazioni dirette con l'hardware potrebbero essere necessari
4. Il design dei sistemi embedded molto spesso prioritizza questioni di sicurezza e attendibilità, per questo motivo si tende a riciclare modelli già ampiamente testati piuttosto che rischiare di introdurre 'caos' proponendo modelli nuovi.

I sistemi embedded possono essere considerati come sistemi reattivi, infatti devono reagire all ritmo dell'ambiente; un sistema embedded in real-time viene realizzato a partire da alcuni input iniziali, detti stimoli (Stimuli). Gli stimoli possono essere di due tipi:

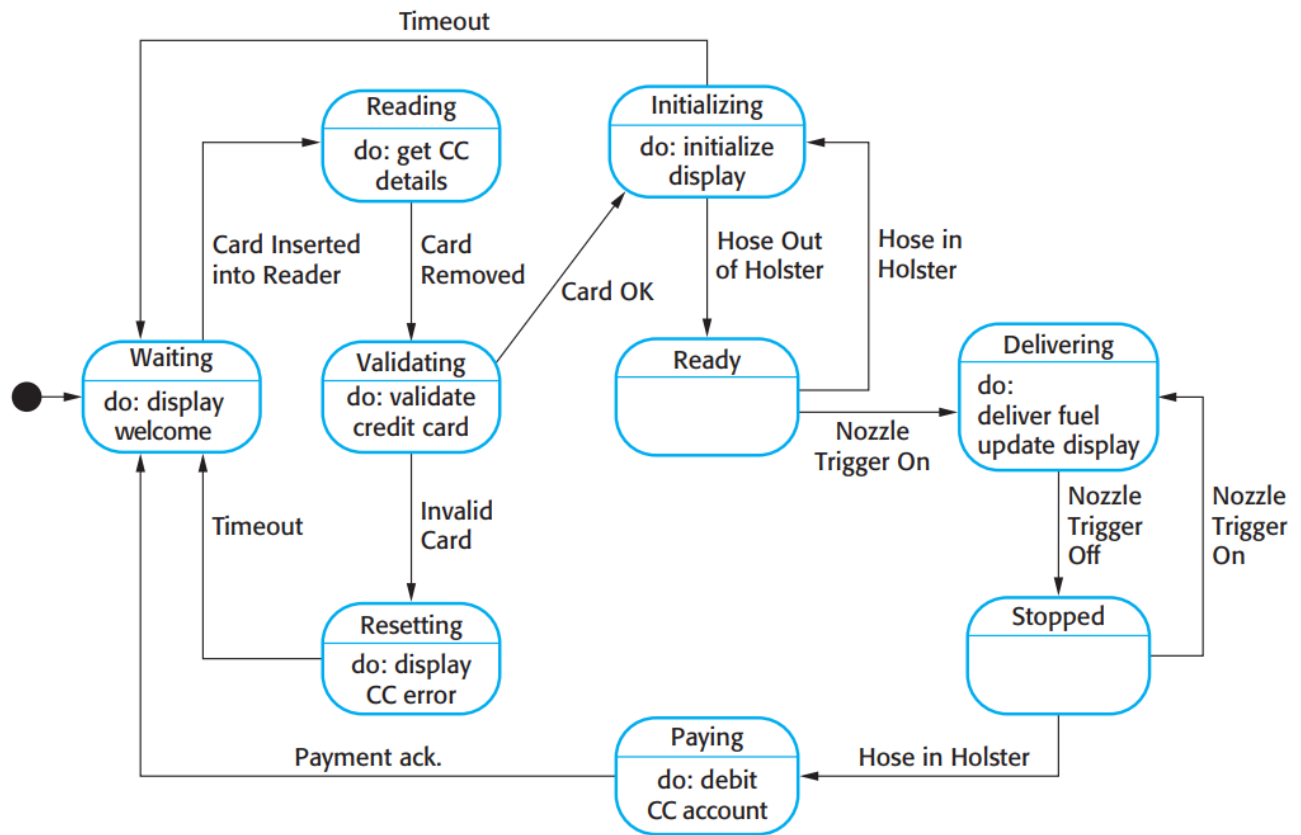
1. Periodici: quando avvengono in lassi di tempo prevedibili
2. Aperiodici: quando lo stimolo non può essere previsto dal sistema

Un sistema embedded real-time è composto da sensori e attuatori separati tra loro, possiamo vedere questi sistemi come processi MIMO, le figure che seguono mostrano come sono correlati tra loro sensori e attuatori



Modellazione real-time

Sfruttando i diagrammi di stato di UML, possiamo rappresentare i processi che avvengono in sistemi real-time, la supposizione necessaria riguarda il fatto che, in un dato istante, il processo si trova in uno dei possibili stati, e che, alla ricezione di uno stimolo, lo stato potrebbe variare. Vediamo un esempio di un sistema embedded (una pompa di benzina) rappresentata tramite uno state diagram:



Programmazione real-time

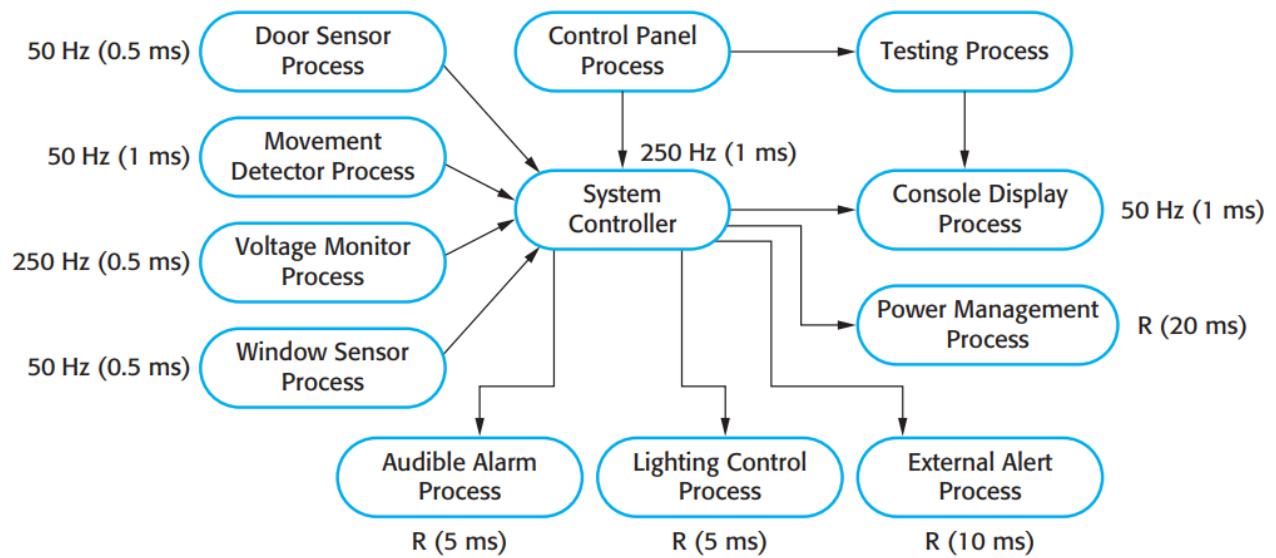
Per quanto riguarda la programmazione di questi sistemi generalmente si utilizza Assembly e C, poiché garantiscono una maggiore velocità di risposta, il problema principale risiede nel fatto che questi due linguaggi non garantiscono la concorrenza tra i processi, si potrebbe optare quindi per linguaggi orientati agli oggetti, una versione modificata di Java è stata proposta con lo scopo di interfacciarsi meglio alla programmazione di questi sistemi.

Analisi della tempistica

Nei sistemi real-time la correttezza dell'output non basta, l'output prodotto deve essere anche prodotto in un tempo significativamente basso, l'analisi della tempistica si basa proprio sullo stabilire quanto spesso un dato sistema embedded real-time dovrebbe produrre un output. Questo tipo di analisi prende in considerazione tre fattori chiave:

1. **Deadline:** Il tempo entro il quale il sistema deve necessariamente produrre uno stimolo
2. **Frequenza:** quante volte un processo deve essere eseguito in dato istante di tempo (ad esempio 1 secondo)
3. **Il tempo richiesto ad un singolo processo per reagire ad uno stimolo**

Vediamo un esempio di un sistema embedded real-time su cui è stata fatta un'analisi dei tempi



Sistemi operativi real-time

Un sistema operativo real time ha il compito di gestire i vari processi che avvengono nei sistemi real time, fornendo un file system e gestendo i processi a runtime.

I classici sistemi operativi non riescono a gestire questi sistemi poiché richiedono troppo spazio, quindi per i sistemi real-time vengono creati ad-hoc, che solitamente includono

- Un clock real-time
- Un gestore delle interruzioni
- Uno scheduler
- Un dispatcher