Parte I

Analisi del contesto

Capitolo 2

Qualità nella produzione del software

*La metodologia **APM (Application Performance Management)** è un insieme di principi, processi, tecniche e linee guida per gestire le prestazioni delle applicazioni web. Per poter esser essere applicata in modo corretto e completo, necessita di integrarsi le altri fasi che compongono il ciclo di vita del software (dallo studio di fattibilità alla fase di manutenzione).*

*Allo scopo di comprendere come definire e posizionare la metodologia APM, questo capitolo presenta sia l'insieme di attività per la conduzione dei progetti software e i criteri con cui queste vengono intraprese (**cicli di vita**), sia le **metodologie di sviluppo** da seguire per ottenere dei prodotti software che rispondo a requisiti di Qualità. Gli obiettivi del capitolo sono duplici: **1)** definire e strutturare, a diversi livelli di astrazione, una **terminologia comune** per il processo di produzione del software e applicata poi coerentemente nella metodologia APM; **2)** illustrare i principali aspetti legati alla **Qualità del software (di prodotto e di processo)** per apprendere sia i vantaggi nell'adottare questi principi sia come contribuisce in merito l'introduzione dei processi di Testing e Monitoring, caratteristici della metodologia APM.*

2.1 Il software: prodotto e processo

L'Ingegneria del software è una disciplina tecnologica e gestionale che ha lo scopo di fornire formalismi, metodi e strumenti per la produzione sistematica e il mantenimento di software efficiente, anche con l'obiettivo di svilupparlo e modificarlo entro i tempi e i costi preventivati[3]. Il termine *Ingegneria del software* fu coniato durante la conferenza NATO organizzata a Garmisch (Germania) nel 1968, per testimoniare l'esigenza che la produzione di software diventasse una disciplina ingegneristica, con basi teoriche e metodologiche.

[3] IEEE Standard Glossary of Software Engineering (http://standards.ieee.org)

Risulta opportuno evidenziare, oltre ai campi della matematica, informatica ed economia, l'importanza dell'aspetto organizzativo, come complemento importantissimo per ottenere e far funzionare un buon prodotto software [1].

La comparsa in letteratura e nella pratica dello sviluppo del software dei concetti di *ciclo di vita* e di *processo software* si può far coincidere con la nascita dell'ingegneria del software, in quanto rappresenta un passaggio storico dallo sviluppo del software inteso come attività "artigianale" ad un approccio industriale, in cui la creazione di programmi e sistemi software viene considerata come un processo complesso che richiede pianificazione, controllo e documentazione appropriati.

I problemi che la comunità scientifica di ingegneria del software si trova a dover risolvere sono quelli classici del mondo produttivo: tempi di produzione, costi, e non ultimo il livello di Qualità dei prodotti. L'idea di studiare e migliorare il processo di produzione del software per migliorarne il prodotto non è nuova e deriva da antiche teorie socio-economiche. I problemi e le relative soluzioni somigliano a quelli generali dei processi di produzione: il processo deve essere definito come composizione ricorsiva di sotto-processi pianificati, assegnati e misurabili.

Come anni di ricerche e studi hanno dimostrato, il software è un'entità per certi versi unica; si differenzia dal risultato delle attività produttive classiche per la sua natura *immateriale*. In pratica non esiste una fase di progettazione separata da quella di produzione in senso stretto e il problema di costruire differenti copie di uno stesso prodotto non si pone. Il software è *malleabile*, cioè è tecnicamente semplice modificarlo ma è estremamente difficile gestirne i cambiamenti in modo da conservarne la consistenza. Inoltre, l'ambiente in cui il processo di produzione del software avviene è costituito da calcolatori, quindi è possibile automatizzare parti del processo programmando opportunamente gli stessi calcolatori.

Un secondo aspetto distintivo del software è costituito dalle difficoltà incontrate nel doverne definire e valutare la Qualità. Diversi sono i parametri utilizzati per valutare la bontà di un prodotto o di un processo (si veda il paragrafo 2.2). Certamente, la Qualità dipende da una miriade di fattori che vanno dalla capacità di soddisfare in modo semplice e conforme ai bisogni dell'utente, alle caratteristiche di affidabilità, sicurezza ed efficienza mediante i quali tali bisogni sono raggiunti.

Al fine di ottenere prodotti di Qualità rispettando tempi e costi pianificati, bisogna definire il processo di produzione software tenendo conto che anch'esso, essendo il prodotto di un'attività ingegneristica, può essere soggetto a modifiche. Uno dei problemi principali di questa disciplina è appunto gestire la corretta evoluzione del prodotto software e dei processi sottostanti che lo regolano. Ne consegue che è necessario perseguire il miglioramento continuo del processo di produzione software per rendere la produzione più adeguata e conforme alle esigenze del mercato. I processi di miglioramento riguardano sia la capacità che l'efficienza dell'organizzazione, perché da una parte rendono il processo più

prevedibile e controllabile, dall'altra contribuiscono al miglioramento della Qualità dei prodotti software.

2.1.1 Terminologia

In questa sezione sono riportate le definizioni dei principali termini utilizzati in questa tesi, soprattutto per la definizione della metodologia di *Application Performance Management* (*APM*); il significato di questi termini deriva per lo più da testi e articoli di Ingegneria del Software ([2][3][4][5]) e dal Framework di LONCHAMP [6].

Come precedentemente indicato, le organizzazioni moderne, per poter essere competitive in un mercato sempre più complesso ed esigente, si concentrano sul Cliente e quindi sulla Qualità del prodotto o servizio offerto. Questo motivo ha spinto le aziende a semplificare la struttura gerarchica tradizionale, organizzata per funzioni che tagliano verticalmente l'organizzazione, affiancando trasversalmente un "approccio per processi"[4]. Le strutture gerarchiche tradizionali spesso sono strutture troppo pesanti con attività ridondanti e presentano dunque diversi limiti:

- Mancata condivisione di obiettivi: la maggior parte delle decisioni vengono prese al vertice e questo comporta una responsabilità puramente esecutiva ai livelli più bassi. Inoltre ogni funzione è focalizzata su se stessa e non sugli obiettivi generali aziendali.
- Scarso flusso di comunicazioni, puramente formali e con canali di diffusione limitati e determinati dalle rigide regole della gerarchia.
- Limitata percezione del Cliente: le esigenze ed il grado di soddisfazione del Cliente sono limitate alle sole funzioni che hanno rapporti diretti con lui, a tutto il resto dell'organizzazione il Cliente è sconosciuto.

Sebbene la struttura funzionale consentisse, nel recente passato, di raggiungere livelli di efficienza elevati all'interno delle organizzazioni, con l'aumentare nel tempo del numero di funzionalità e della loro complessità, si è sentita l'esigenza di ottimizzare il risultato globale dei singoli processi. I principi chiave di una gestione per processi sono:

- Orientarsi al Cliente.
- Identificare sia il Cliente finale che i Clienti interni.
- Innescare un processo interno di apprendimento continuo.
- Attuare il coinvolgimento totale delle Risorse Umane.

[4] "*Approccio per processi*" secondo l'ISO 9004:2000 è l'adozione, nell'ambito di un'organizzazione, di un sistema di processi, unitamente alla loro identificazione, interazione e gestione.

- Razionalizzare attività, processi e procedure operative esistenti.
- Disegnare i nuovi processi in modo ottimizzato.
- Focalizzarsi sul miglioramento continuo.
- Implementare un sistema di monitoraggio continuo delle performance di processo.

- **PROCESSO SOFTWARE:** o processo di sviluppo o produzione del software, è un insieme strutturato di flussi di lavoro e attività interdipendenti del mondo reale, coinvolte nello sviluppo di un sistema software; utilizza risorse ed è progettato per poter generare prodotti o servizi ad un determinato Cliente o a gruppi di Clienti che hanno gli stessi bisogni.

  È caratterizzato da:

  - Un insieme di Input e un insieme di Output (Risultato atteso)
  - Un Inizio e una Fine (in termini di attività)
  - Una finalità prestabilita o intenzionale (Obiettivo o Goal)
  - Una serie di attività interfunzionali note
  - Un valore aggiunto
  - Una o più responsabilità ben chiare e definite del "Proprietario del processo" (*Process Owner*).
  - Un ciclo di vita

  Per tenere sotto controllo i processi dell'IT (Information Technology) si possono utilizzare le seguenti pratiche:

  - **CSF (Critical Success Factor):** forniscono l'indicazione su cosa fare per implementare i controlli a livello di processo. Definiscono indicazioni esplicite su ciò che deve essere fatto per fare in modo che gli obiettivi di processo siano raggiunti, dal punto di vista strategico, tecnico, organizzativo e procedurale.
  - **KPI (Key Performance Indicator):** sono indicatori quantitativi (misurabili) che segnalano come il processo si sta evolvendo per raggiungere il requisito di business del processo. Generalmente sono una misura dei fattori critici di successo e, quando monitorati e rielaborati, individuano le opportunità per il miglioramento del processo.
  - **KGI (Key Goal Indicator):** rappresentano l'obiettivo del processo da raggiungere. Mente il KPI serve a monitorare progressivamente l'attività collegata al processo, il KGI segnala il raggiungimento o meno dell'obiettivo (deciso a priori dall'organizzazione).

  I CSF sono elementi o azioni che devono essere applicati per avere il controllo del processo, i KPI descrivono come il processo si sta comportando, infine i KGI

indicano risultati visibili dall'esterno del processo, ovvero il soddisfacimento dei requisiti aziendali.

- **ATTIVITÀ:** rappresenta un'unità di lavoro del processo che deve essere eseguita da uno specifico *Ruolo*. La sua descrizione implica gli strumenti da utilizzare (Tool Mentor). Essa può produrre qualcosa o anche semplicemente avere funzioni di coordinamento all'interno del processo. Ogni singola attività può essere suddivisa in un insieme di operazioni elementari o *task*. Ogni attività è caratterizzata da un insieme di manufatti in input e da uno o più manufatti di output.

  Se il processo da gestire è complesso, è buona norma suddividerlo in processi elementari più piccoli o **sottoprocessi ('Sub-process')**, che a loro volta sono formati da un insieme di **fasi** per completare le quali sono necessarie delle attività.

  Il flusso logico che attraversa tutto il processo, attraversando sottoprocessi, fasi ed attività è denominato **Workflow**.

  Una mera lista dei ruoli, attività ed elaborati non costituisce un processo; è necessario un modo per descrivere una sequenza significativa di attività che produce un risultato di valore, e mostra le interazioni tra i ruoli. Un workflow è appunto una sequenza di attività che produce un risultato di valore osservabile dove tipicamente sono richiesti diversi ruoli.

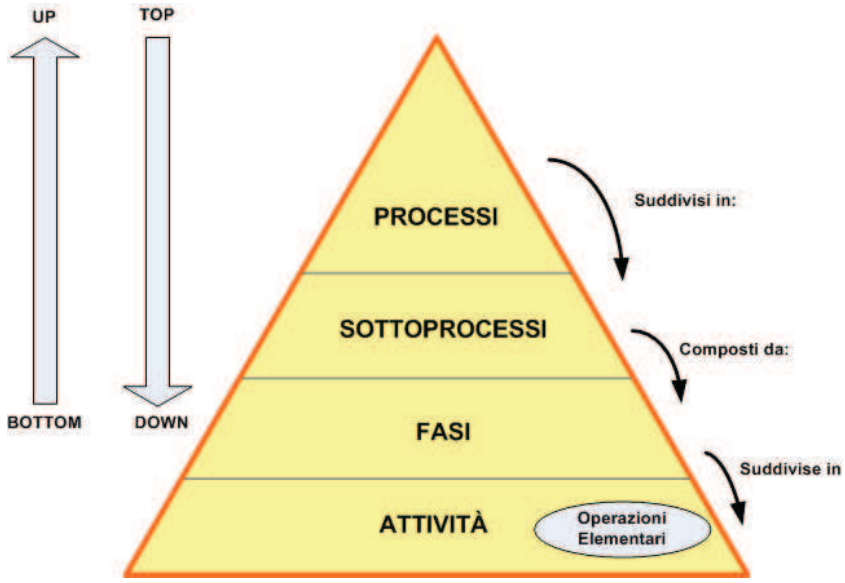

Figura 2.1: Struttura gerarchica dei processi

- **PRODOTTO SOFTWARE**: insieme di tutti i manufatti che permettono l'utilizzo di un programma da parte di un utente. È composto di codice, documentazione,

prodotti intermedi come casi di test e specifiche di progetto (*manufatti*). Il prodotto software è legato ad uno specifico processo di sviluppo.

- **MANUFATTO:** prodotto del processo, è un'entità creata o modificata da un'attività durante un processo, o perchè è un risultato richiesto, oppure perché è utile a facilitare il processo stesso. Può essere rappresentato da: modelli, database o tipologie alternative di repository, codice sorgente ed eseguibile, documenti. I manufatti sono di responsabilità di un singolo *ruolo*, rendendo tale responsabilità facile da identificare e comprendere, e promuovendo l'idea che ogni informazione prodotta in un processo richiede un insieme di competenze (d'ora in avanti si utilizzerà il termine *skill*) appropriato. Vista la complessità degli elaborati prodotti e la durata nel tempo delle attività necessarie al loro sviluppo, si rende necessaria l'adozione di tecniche e strumenti di *Configuration Management (CM)*. Il Configuration Management è un insieme di metodologie e tecniche per la gestione delle versioni dei documenti e delle configurazioni dei prodotti e di tutti i manufatti manipolati nel corso di un progetto. L'adozione di queste tecniche consente di ridurre gli errori e contribuisce a mantenere la coerenza e l'integrità tra i vari manufatti rilasciati in un progetto software. Inoltre, concorre al miglioramento dell'intero processo di sviluppo, perché la gestione facilitata dei manufatti, incrementa la capacità di riutilizzarli in altre fasi del processo.
- **RISORSA**: indica un bene messo a disposizione di un'*attività* che ne ha bisogno per essere eseguita. Questi beni possono essere di diversa natura: tempo, risorse finanziarie, strumenti automatici, Risorse umane, una risorsa fondamentale in quanto il lavoro di squadra (*team*) costituisce l'essenza di un progetto. Per definizione, per realizzare un progetto occorre una squadra che vi lavori [7].
- **RUOLO:** definisce il comportamento e le responsabilità di un individuo o gruppo di individui che lavorano assieme come team, nel contesto di un'organizzazione di sviluppo software. Si noti che i ruoli non sono individui; descrivono invece come gli individui devono comportarsi nel processo, e ne identificano le responsabilità. Ciascun membro dell'organizzazione può assumere diversi ruoli. La mappatura tra ruoli e individui, permette ad un individuo di assumere più ruoli, e a un ruolo di essere assunto da più individui. Generalmente i ruoli sono raggruppati in 6 macrocategorie: Manager, Analisti, Sviluppatori, Sistemisti, Tester e Ruoli di Supporto.
- **TOOL:** sono strumenti che supportano il programmatore e gli altri ruoli coinvolti nel processo di sviluppo software, in una particolare fase della produzione del software. Esistono strumenti utilizzati per la descrizione del problema e la sua analisi, di programmazione, di verifica e validazione, di reverse engineering (estrarre informazioni dal codice esistente per facilitare attività di manutenzione) e strumenti per la gestione delle configurazioni.
- **METODOLOGIA**: un insieme di principi base su cui si fonda la realizzazione del prodotto software (es. principi di Qualità discussi in seguito), tecniche di sviluppo (metodiche sviluppate per guidare il processo di sviluppo e possono

essere informali come l'Extreme Programming, semi-formali come l'approccio object-oriented e formali come le tecniche di model checking), standard (linee guida), skill (intesi come schemi che descrivono diverse tipologie di problemi e soluzioni consolidate nell'ingegnere del software) da seguire per raggiungere certi obiettivi. Più formalmente consiste in uno o più processi da seguire e documenti o altri manufatti da produrre usando più o meno dei linguaggi standard. Ai processi, sperimentati e ripetibili, sono applicate un insieme di conoscenze, tecniche di gestione e strumenti di lavoro specifici.

- **PROGETTO**: è un'attività lavorativa unica e ben definita, con una data di inizio, una data di fine, un obiettivo chiaramente individuato e vincoli di tempo, risorse, costi e Qualità ben determinati. Il progetto dunque per poter essere concretamente applicato, richiede un *processo software* che, a partire da una esigenza di un Cliente, ha l'obiettivo di produrre come risultato un prodotto o servizio che soddisfi completamente tale esigenza. Nella realtà, più che come un processo unico, il progetto è percepito come una serie di processi interagenti, formato da quattro momenti fondamentali: Definizione, Pianificazione, Realizzazione e Controllo del cambiamento. Come si può notare la distinzione tra progetto e processo è molto sottile: il progetto è uno sforzo temporaneo che ha l'obiettivo di creare un prodotto o un servizio unico, con risorse ben determinate.

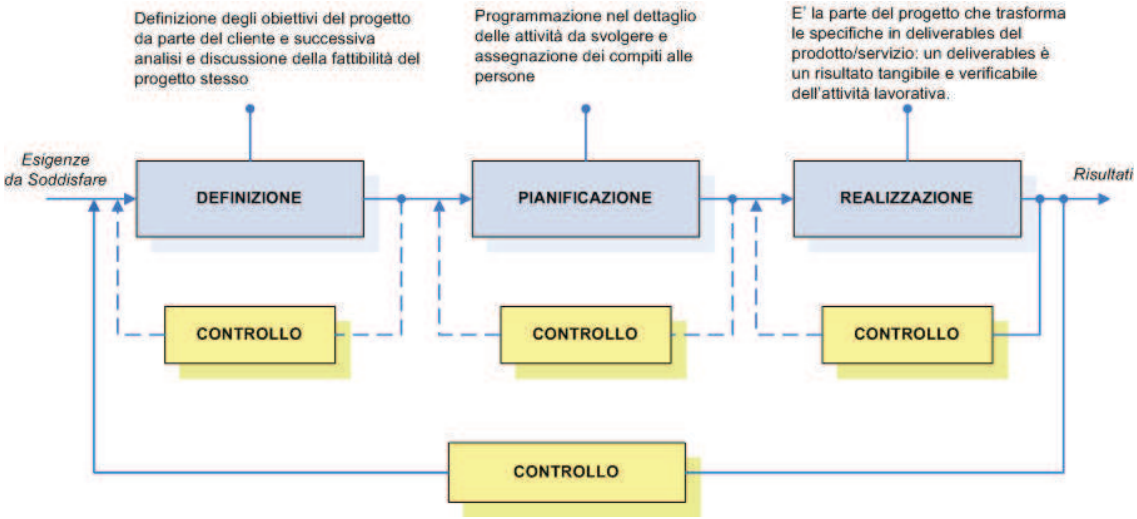

Figura 2.2: Il Progetto come processo

- **ARCHITETTURA**: una descrizione di un sistema definita da (1) **aspetti strutturali** che definiscono a loro volta la scomposizione del sistema in un insieme di componenti correlate e le relazioni tra queste componenti (connettori e interfacce); (2) **aspetti funzionali**, le responsabilità e le funzionalità allocate a ciascuna componente; e (3) **aspetti comportamentali**, le interazioni tra le componenti in specifiche situazioni elaborative o scenari d'esecuzione

- **INFRASTRUTTURA**: le scelte tecnologiche che occorrono a definire l'architettura e che si concretizzano una particolare installazione per la messa in esercizio dell'architettura definita allo scopo di "ospitare" applicazioni software (in termini di componenti e le interazioni che occorrono tra di essi).
- **SISTEMA**: è la composizione dell'infrastruttura e delle applicazioni software.

2.1.2 Il Modello come strumento di comunicazione

La comunicazione è una parte fondamentale del processo e uno degli aspetti fondamentali del modello è appunto permettere la comunicazione. Un modello è una rappresentazione astratta della realtà per comprendere in profondità il problema da analizzare o più specificatamente il sistema da costruire o già costruito. Concretamente un modello è formato da una collezione di elaborati, ognuno dei quali esprime una determinata vista sul sistema, risolve un determinato obiettivo, fornisce informazioni su aspetti statici e dinamici del sistema, nonché sui flussi di dati che fluiscono da un attore ad un altro.

Essendo utilizzato sia dai committenti che dagli implementatori deve includere tutti quegli elementi che permetto di esplorare in profondità il sistema nel tentativo di formulare parecchie soluzioni ed escludere gli elementi irrilevanti ad un dato livello di astrazione.

Pertanto modellare,

- aiuta a visualizzare un sistema com'è o come vorremmo che fosse;
- aiuta a comunicare con il Cliente che, notoriamente, usa un linguaggio spesso molto diverso da quello dell'analista e non sa esattamente cosa desidera finché non osserva un primo prototipo del software in funzione;
- permette di specificare la struttura o il comportamento del sistema e guida nella costruzione di quest'ultimo,
- documenta le decisioni che sono state prese,
- permette di ottenere un certo grado di formalismo e allo stesso tempo l'intuitività che un diagramma, seppur complesso, può fornire.

Quindi costruiamo dei modelli di sistemi complessi perché non possiamo comprendere facilmente un tale sistema in tutta la sua interezza. Esistono dei limiti umani invalicabili nel capire la complessità di un sistema ed è proprio attraverso la modellazione che si riesce a presentare il problema oggetto di studio, focalizzandosi sui diversi aspetti con diagrammi, testo e quant'altro può essere utile a rappresentare il sistema da realizzare.

La modellazione si serve di **linguaggi di descrizione** (o modellazione), che rendono possibile la creazione di descrizioni univoche e sintetiche. Le descrizioni basate su testi e diagrammi informali possono presentare problemi di imprecisione e

ambiguità: diverse persone possono dare interpretazioni differenti della stessa descrizione, soprattutto quando trasmettono poca informazione, magari anche spesso in modo confuso. Per questo si preferiscono linguaggi di tipo formale o semiformale in quanto esiste anche la possibilità di costruire strumenti automatici che analizzano la descrizione prodotta al fine di valutarne Qualità e coerenza e provarne le proprietà.

Un esempio di linguaggio di descrizione piuttosto diffuso nell'ingegneria del software è **UML (*Unified Modeling Language*)**, una collezione di diverse notazioni grafiche, alcune delle quali semiformali, cioè che hanno una sintassi ben definita ma mancano di una semantica precisa utilizzata per la descrizione degli elementi che compongono un prodotto software, relativi ad uno specifico progetto. Per maggiori dettagli si veda il paragrafo 2.4.1.1.

I linguaggi di descrizione devono essere in funzione delle esigenze del particolare progetto e non è detto che esista un linguaggio unico per tutte le soluzioni. In alcuni casi UML potrebbe essere la soluzione più conveniente per una certa tipologia di progetto; in altri, dove magari è necessario studiare e verificare altre proprietà, potrebbe risultare più opportuno l'uso di altri linguaggi formali che rendano possibile questo tipo di controlli.

2.1.3 Modello di processo: struttura e vantaggi

Con *modello di processo* s'intende la descrizione astratta di un processo software o di una classe di essi. A volte viene indicato anche con il termine *framework* di processo, per enfatizzare il suo obiettivo: offrire un quadro di riferimento da cui si può partire per creare processi specifici, adattati in una particolare realtà aziendale. La necessità di definire, nel corso degli anni, un unico modello di processo universale ha portato alla definizione di differenti modelli noti, i quali sono adottati dalle singole organizzazioni secondo le specifiche esigenze. Il modello di processo può essere definito tramite un linguaggio informale (es. gli standard della serie ISO 9000 sono un esempio di modello di processo definito in lingua scritta) oppure con un linguaggio formale che gode del vantaggio di poter analizzare staticamente e automaticamente il modello, per rivelare errori e inconsistenze.

Un modello di processo consiste di almeno cinque sottomodelli [8]:

- **modello di prodotto:** si occupa della descrizione dei componenti software, prodotti durante lo svolgimento di un processo software e delle loro relazioni reciproche. Sono degli oggetti passivi manipolati dalle attività. Il modello di prodotto include la descrizione dei documenti dei requisiti, di analisi e di progettazione, il codice sorgente, il codice oggetto, i programmi eseguibili e i dati per la verifica.
- **modello delle attività:** descrive l'insieme delle attività eseguite durante il processo di sviluppo software, dove ogni attività può essere vista come la creazione o la trasformazione di uno o più manufatti. Le attività possono

essere atomiche che scomponibili in operazioni elementari. Un'attività atomica è eseguita da un *agente*, che può essere uno strumento automatico, oppure un essere umano o entrambi.

- **modello degli strumenti:** descrive gli strumenti applicativi di supporto alle attività del processo. Si dividono in strumenti interattivi (la cui esecuzione necessita dell'intervento umano) o in strumenti di derivazione che sono completamente automatizzabili (es. un compilatore).
- **modello organizzativo:** il comportamento umano viene generalmente espresso dal concetto di ruolo che si occupa di descrivere i diritti e le responsabilità del singolo. I ruoli possono essere assegnati alle persone per eseguire compiti specifici e ciascuna persona può avere diversi ruoli. Un modello organizzativo specifica le responsabilità per le attività specifiche, le risorse impiegabili, il bilancio preventivo, i vincoli di tempo e di allocazione.
- **Meta-modello:** descrive come i modelli possono essere creati, raffinati, cambiati, istanziati ed eseguiti.

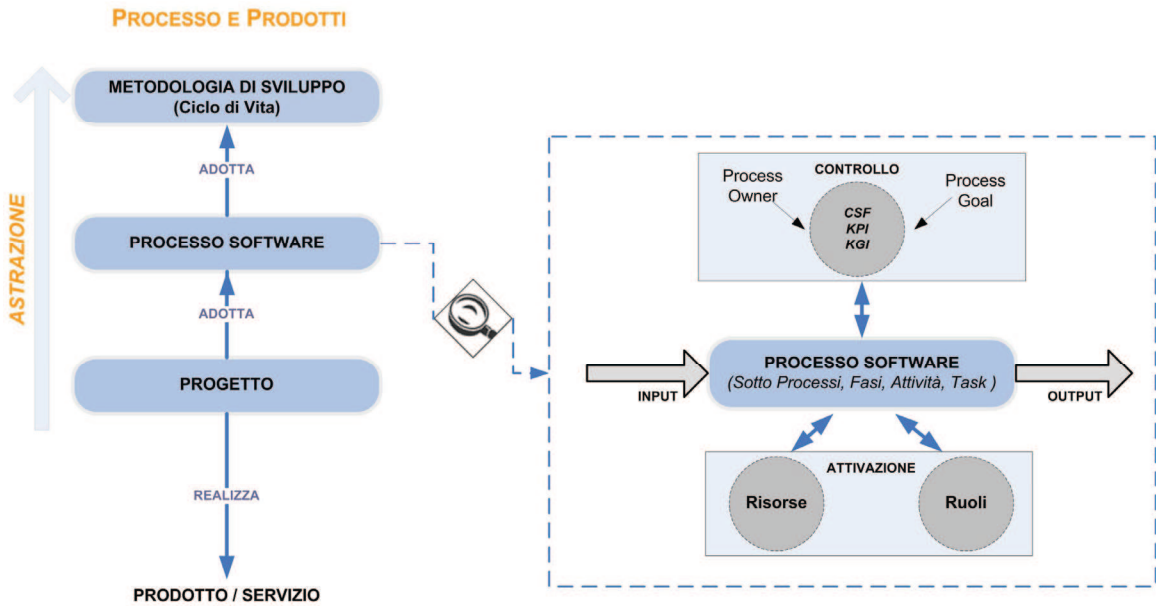

Figura 2.3: Astrazione del processo di sviluppo software

Lo schema di figura 2.3, rappresenta una sintesi strutturale dei concetti finora esposti: il **processo software** è la concretizzazione dell'adozione di un **framework di processo** in una specifica organizzazione (specializzandosi per livelli di competenza, aree applicative e dimensione dei progetti), diversamente un **processo** è richiesto ogni qualvolta si deve attuare uno specifico progetto. Nella realtà aziendale, lo stesso processo software è di norma seguito da gruppi distinti che

hanno il compito di sviluppare diversi prodotti (o suoi componenti), ognuno dei quali è descritto in un progetto.

Il **ciclo di vita** costituisce il "nucleo" del modello di processo in quanto ne definisce struttura di massima e criteri di successione delle fasi. In alcuni casi il modello di processo coincide con la mera definizione di un ciclo di vita (Es. il ciclo di vita a cascata), in altri, maggiormente strutturati, ai principi del ciclo di vita si aggiungono ulteriori dettagli metodologici per fasi e attività (Es. *Unified Software Process*).

Il legame esistente nei concetti di ciclo di vita del software, processo e progetto è analogo alla relazione esistente tra stile architetturale, architettura e installazione. Così come l'architettura di un sistema informatico può essere basata su uno o più *stili architetturali* (client-server, peer-to-peer), un processo software può essere organizzato utilizzando diversi *cicli di vita* (modelli del processo). Un *processo*, dunque, è una precisa e ordinata organizzazione di attività, così come una *architettura* è identificata da particolari componenti software (es. Web Server, Application Server e Database Server). L'installazione di un particolare prodotto software è paragonata al progetto, in quanto costituisce la messa in esercizio di una particolare architettura: ci possono essere molteplici installazioni (si differenziano per parametri e caratteristiche offerte dalle componenti, proprie di uno specifico *vendor*) della stessa soluzione e quindi di una stessa architettura.

L'altra componente principale nello sviluppo del software sono l'utilizzo dei linguaggi. Gli esempi più noti sono i **linguaggi di programmazione** per la creazione vera e propria del codice sorgente dei programmi. Seguono i **linguaggi di modellazione** (o descrizione): notazione usata per esprimere le caratteristiche di progetto e di processo. Il linguaggio UML è un linguaggio utilizzato per modellare lo spazio del problema concordato con il cliente che va dalla modellazione del progetto (es. dell'architettura e dei requisiti) sino alla modellazione del processo (es. posso usare i diagrammi UML per descrivere un ciclo di vita). Un altro linguaggio di modellazione utilizzato per pianificare i tempi di realizzazione di un progetto sono i diagrammi di Gantt (le diverse attività vengono ordinate secondo una precisa progressione temporale). Con l'avvento di Internet sono emerse i **linguaggi di Mark-up** per descrivere tipologie di informazioni e concetti ed infine è opportuno ricordare i **linguaggi domain-specific**, usati per uno specifico domino o applicazione (es. SQL per interrogare le basi di dati). [9].

Ogni metodo e linguaggio si serve di **strumenti di supporto** per l'effettiva implementazione del sistema informatico da realizzare [9]. È opportuno distinguere tre tipologie di strumenti:

- **Tool di processo e workflow** per descrivere il processo di sviluppo e per la creazione di appositi ambienti e toolset[5] (Generatori)**.** Un

[5] Il termine *Toolset* viene normalmente utilizzato per identificare un insieme integrato di tool orientati ad una specifica fase del processo (*es. strumenti di programmazione e debugging per la fase di codifica*).

esempio di generatore è il componente per lo sviluppo di particolari plug-in (moduli di estensione delle funzioni base di una applicazione).

- **Tool di gestione del processo** che consentono l'esecuzione e il coordinamento efficiente delle attività presenti nel processo di sviluppo software. Appartengono, per esempio, a questa categoria gli strumenti di **Project Management** (includono la possibilità di creare diagrammi di Gantt), che supportano il manager nella pianificazione e controllo delle tempistiche, dell'effort speso e degli obiettivi da raggiungere nel corso del progetto; software di supporto per la gestione del sistema Qualità e della documentazione.
- **Tool per lo sviluppo del software** e di gestione dell'applicazione in esercizio (Ambienti di sviluppo o *Software Development Kit* e Tool di monitoring).

Il modello oltre ad essere un efficiente mezzo di **comunicazione** tra le persone che producono e gestiscono l'evoluzione di un prodotto software, offre i seguenti vantaggi:

- **Previsione:** strumento con il quale stimare le risorse e i tempi necessari per portare a termine il processo in questione. Se il modello è interpretabile è possibile simulare l'effetto delle decisioni possibili, per scegliere quella migliore.
- **Completezza:** deve considerare tutte le fasi del ciclo di vita, le attività necessarie per l'attuazione del processo, gli strumenti software, le risorse hardware e le persone investite dai loro ruoli.
- **Evoluzione:** mentre adattare significa aggiungere informazioni al modello, fornendo valori ai suoi parametri e scegliendo determinate alternative, evolvere significa cambiare parti del modello. Le motivazioni che possono rendere necessarie modifiche ad un modello di processo sono tre: il modello contiene errori (**evoluzione correttiva**), il modello è divenuto inadatto in seguito a cambiamenti nel mondo reale modellato (**evoluzione adeguativa**), o il modello può essere migliorato attraverso l'aggiunta o l'eliminazione di nuove funzionalità (**evoluzione perfettiva**). I cambiamenti possono avvenire prima o durante l'esecuzione del modello stesso.
- **Riutilizzo:** un vantaggio della modellazione dei processi sta nel fatto che non è il caso di definire ogni volta un modello a partire dal nulla, ma si possono riutilizzare modelli, o parte di essi, già definiti e dimostratisi efficaci.
- **Verifica:** la modellazione dei processi di produzione del software permette di svolgere attività di verifica della bontà del processo, ad esempio mediante la simulazione dello stesso.

- **Valutazione:** la definizione esplicita del processo deve rendere possibile la misurazione e la valutazione, sia del modello del processo adottato che del prodotto in via di sviluppo.
- **Attuazione:** s'intende l'interpretazione del modello di processo e l'adattabilità di quest'ultimo ad un contesto concreto. Indipendente da come i modelli vengano interpretati (manualmente o dai calcolatori), si distinguono due tipologie di modelli di processo. I modelli *prescrittivi* definiscono quali attività e in che ordine devono essere portate a compimento e i *proscrittivi* non fanno altro che evidenziare le azioni da non intraprendere affinché il processo avanzi secondo lo schema prescelto.

  Inoltre un modello di processo, dopo essere stato interpretato correttamente, deve poter essere implementato per situazioni differenti (*customization*). Questa peculiarità evidenzia che non è possibile definire un modello di processo universale, cioè utilizzabile per qualsiasi organizzazione e qualsiasi prodotto.

2.2 Qualità del software

2.2.1 Le diverse accezioni della Qualità

Negli ultimi anni, un fattore che ha assunto un ruolo sempre più importante nelle strategie e nella vita quotidiana di molte organizzazioni è la "Qualità". La società è entrata in una fase che è contraddistinta da alcuni fenomeni importanti e il soddisfacimento dei bisogni primari del consumatore ha reso prioritarie nuove necessità e richieste. La competizione è oggi incentrata sulla capacità di un bene o di un servizio di soddisfare aspettative dell'utente sempre più diversificate e articolate. Inoltre, la complessità dei prodotti e servizi è cresciuta in modo impressionante. Molti prodotti oggi sul mercato, come per esempio l'elettronica di consumo o le stesse autovetture, hanno raggiunto livelli elevati di sofisticazione e lo stesso vale per i processi di servizio, in particolar modo lo sviluppo di Internet ha segnato la crescita di servizi Web (*es. e-commerce, home banking, trading on-line*).

Nel caso dei servizi Web, la necessità di fornire servizi con continuità, efficienza e sicurezza, rende la Qualità dei servizi un'esigenza di business e una caratteristica distintiva per il successo nel mercato. Ma per poter soddisfare bisogni sempre più complessi mediante beni e servizi sempre più sofisticati è necessario prevedere processi di produzione e di servizio in grado di rispondere adeguatamente alla sfida posta dal mercato. In questo scenario, la "Qualità" dei prodotti e dei processi diviene necessariamente il punto centrale delle strategie di molte organizzazioni che operano nella nostra società . Ma che cosa s'intende con il termine **"Qualità"**?

Comunemente, la Qualità può essere definita come le caratteristiche o proprietà positive proprie di un prodotto/servizio, oppure quell'insieme di

proprietà che rendono quel prodotto/servizio particolare. Infatti le seguenti definizioni evidenziano questa affermazione:

- ISO 8402: *la totalità delle caratteristiche di un prodotto o di un servizio che lo rendono conforme con i relativi requisiti espliciti ed impliciti.*
- P. Crosby: *Conformità alle specifiche*
- J. Juran: *Adeguatezza all'uso*
- Ishikawa: *La soddisfazione delle esigenze dei clienti*
- *R.Pressman, Ingegneria del software, McGraw Hill, 2000:* conformità con i requisiti funzionali e prestazionali espiciti, con standard di sviluppo documenti e con le caratteristiche implicite che è lecito attendersi.
- *UNI ISO 9004-4:* conseguenza dell'efficacia e della efficienza dei processi che li hanno generati e li supportano.
- *C.Jones, Applied software measurement, 2°ed. McGraw Hill, 1996:* assenza di difetti che potrebbero bloccare l'uso del prodotto o condurlo a fornire risultati e/o prestazioni inaccettabili.

Quindi un possibile modo "operativo", per definire la Qualità è il rapporto tra le prestazioni offerte e i bisogni e le aspettative del Cliente:

- Q = P/A
- P = prestazioni o funzioni realizzate da un prodotto
- A = prestazioni o funzioni attese dall'utente del prodotto

Dunque il Cliente è soddisfatto quando le prestazioni sono maggiori o uguali alle aspettative, viceversa, il Cliente è insoddisfatto e deluso del servizio offerto.

Certamente è possibile menzionare una serie di requisiti generali che definiscono le proprietà desiderabili (attributi) di un servizio ma non definiscono appieno tutte le possibili accezioni del termine Qualità : in alcuni casi la stessa definizione può avere significati diversi a seconda degli attori coinvolti nella fornitura del servizio. Ad esempio per l'utente, l'attributo "efficienza" può voler dire soprattutto velocità nell'ottenere i risultati attesi (tempo di attesa). Per il gestore del servizio, diventa importante la valutazione anche del costo che deve essere sostenuto per fornire il servizio (giorni-uomo e costi di struttura addebitabili alla realizzazione e manutenzione del prodotto o servizio). Questi differenti punti di vista sono correlati ai diversi obiettivi che ciascun attore ha nell'osservare il servizio. L'utente è interessato ad ottenere risultati corretti in tempi brevi. Il gestore deve tenere conto dell'economia complessiva del processo che viene a trovarsi sotto il suo controllo e gestione. Ma quanto in più è disposto a pagare il Cliente per avere una Qualità più elevata? Come sempre, è il rapporto *costi/benefici* che detta le regole.

Figura 2.4: Relazione Costi e Benefici della Qualità

Il costo della Qualità nel settore ICT è costituito dalla somma di tutti gli sforzi per realizzare un prodotto o servizio di Qualità e comprende tutte le attività per garantire la conformità ai requisiti, oltre a quelle derivanti dalla gestione della non conformità. In modo particolare: creazione e gestione del piano Qualità, revisione dei deliverables, audit interni ed esterni, test di conformità agli standard di Qualità, costo di valutazione dei fornitori (è necessario definire una metodologia, basata su metriche oggettive, che faciliti la gestione dei contratti di fornitura, intervenendo nelle fasi di definizione e controllo del processo), costo di valutazione dei possibili problemi (attività di testing). I costi della Qualità vanno sempre pesati in rapporto ai benefici che possono derivare dall'adozione di una soluzione di Qualità. Mentre i costi si collocano temporalmente lungo il ciclo di vita del progetto, i benefici sono presenti lungo tutto il percorso temporale dell'intera soluzione (finché il prodotto o servizio rimane in vita). Tra i benefici più significativi emerge una maggior soddisfazione del Cliente, maggiore produttività (minori ricicli di lavorazione) e conseguente diminuzione dei costi complessivi di progetto, minor costo di assistenza e manutenzione in quanto il prodotto che viene realizzato presenta un minor numero di difetti e maggiore motivazione e soddisfazione nei team di lavoro.

Come si può vedere dalla figura 2.4, se una ipotetica Azienda ha un livello qualitativo molto basso perché non ha messo in piedi nulla per garantire la Qualità dei prodotti, ha solo costi della "Non Qualità" derivanti da scarti, da re-work, da perdita di clienti, ecc. Se poi inizia a fare qualcosa (es. corsi al personale, documentazione, controlli,ecc.) inizierà anche a spendere (costi per la Qualità): fortunatamente nel frattempo i costi della non Qualità iniziano a scendere perché ci

si preoccupa di fare le cose meglio. Il costo complessivo è dato dalla somma delle due componenti (curva rossa), il costo residuo della non Qualità, sempre più piccolo man mano che si investe sulla Qualità e i costi del Sistema di Gestione per la Qualità (vedi paragrafo 2.2.2) che man mano diventa sempre più sofisticato, ma anche più costoso.

Un'altra dimensione che deve essere considerata per comprendere il concetto di Qualità è il livello di astrazione al quale ci si pone nel studiare gli attributi di un servizio. In generale, posso definire l'efficienza di un servizio come il tempo necessario alla ricezione dei risultati di una richiesta di servizio. Tale "metrica" di Qualità può essere declinata in modi diversi una volta considerati specifici processi di servizio: nel caso di un servizio di sportello è il tempo medio di interazione tra sportellista e utente. Nel caso di un procedimento più complesso, tale durata può includere più interazioni, anche in diverse date, tra diversi impiegati e lo stesso utente, ed implica quindi modalità e costi di misura significativamente diversi.

La definizione di una qualsiasi metrica di Qualità deve essere quindi adattata alla luce dello specifico contesto considerato. Una metrica è una definizione operativa che descrive, con termini estremamente specifici, un elemento del prodotto o servizio e il modo in cui questo elemento viene misurato dal processo di controllo della Qualità [10]. In generale le metriche sono degli standard di misura e in modo particolare esistono due tipi di metriche utilizzate nello sviluppo del software:

- **Metriche di prodotto:** quantificazione degli attributi dimensionali e qualitativi dei prodotti software. (Es. utilizzo di LOC (Line of Code) o Function Point per misurare le dimensioni del software realizzato).
- **Metriche di processo e progetto:** offrono un punto di osservazione strategico per la valutazione dell'efficacia di un progetto software. Le metriche per la stima dei costi nei progetti software rientrano in questo contesto.

Le precedenti osservazioni dimostrano che il concetto di Qualità non è facilmente definibile una volta per tutte. Ne consegue che la Qualità non è un concetto assoluto, o un attributo intrinseco di un prodotto o di un servizio, ma è relativo all'uso del prodotto o del servizio e raggiunge il massimo quando coincide con le attese e le aspettative dell'utente del prodotto o del servizio. Quindi è un concetto che varia nel tempo.

2.2.2 Aspetti organizzativi, tecnologici e standard

Ottenere la Qualità nella produzione di software dipende da diversi aspetti: organizzativi, metodologici e tecnologici.

ORGANIZZAZIONE e METODOLOGIA

L'organizzazione della produzione di software è un processo specifico nel quale deve essere definita una politica di gestione del prodotto. Quindi,

l'incremento della Qualità nel software richiede innanzitutto la definizione rigorosa della politica della gestione del prodotto, in relazione alle tipologie organizzative; successivamente, attraverso la misura dei risultati ottenuti, si possono valutare gli effetti della politica adottata ed eventualmente cambiarla. È necessario, quindi, dapprima adottare un sistema nel quale possa essere controllata la Qualità in modo da consentirne la gestione e successivamente certificare il proprio sistema.

Un **Sistema di Gestione per la Qualità (SGQ)** può essere in generale definito come l'insieme di struttura organizzativa, procedure, processi e risorse messi in atto per assicurare un sistematico approccio alla politica per la Qualità decisa dall'Azienda e finalizzato alla soddisfazione delle esigenze del Cliente. Certificare un sistema Qualità, da parte di un ente terzo imparziale ed abilitato (*ente certificatore*), significa che il sistema aziendale è in grado di dare al produttore e al mercato di riferimento garanzie sul soddisfacimento sistematico dei requisiti dei processi utilizzati per creare un prodotto/servizio, nell'ambito delle condizioni contrattuali stabilite tra Cliente e Fornitore.

I vantaggi che portano una Azienda ad adottare un SGQ ed eventualmente a certificarsi sono innanzitutto la riduzione degli scarti di produzione e delle rilavorazioni e il miglioramento continuo dei processi e dei prodotti, che a regime portano anche ad una riduzione complessiva dei costi. L'ottimizzazione dell'organizzazione e la definizione di processi standardizzati, controllati e documentati non possono che migliorare i risultati dell'Azienda, ed essere quindi un importante elemento di competitività in un mercato sempre meno disponibile ad accettare improvvisazione e disorganizzazione.

In termini metodologici, la Qualità del software nel prodotto finale deriva dall'aver stabilito per ogni fase del processo di produzione del software: cosa e come deve essere fatto in termini di attività elementari e la sequenza di tali attività. Un processo speciale, applicato trasversalmente durante l'intero processo di sviluppo software, è la **QUALITY ASSURANCE (QA)**.

La *Quality Assurance* consiste nell'intraprendere un insieme di attività pianificate e sistematicamente attuate di gestione Qualità per garantire e dimostrare (ai Clienti e potenziali Clienti) la capacità di fornire un prodotto conferme ai requisiti desiderati. L'assicurazione Qualità fornisce anche un supporto per il miglioramento continuo dei processi, che costituisce un mezzo iterativo per ridurre gli sprechi e le attività senza valore aggiunto e consente di incrementare il livello di efficienza ed efficacia operativa dei processi [11]. La QA dovrebbe essere eseguita da persone esterne al progetto, che hanno il compito di validare, fase per fase, attività per attività l'adeguatezza di quella fase o di quella attività rispetto al Sistema di Gestione per la Qualità aziendale, al piano di Qualità del progetto, ad altri standard qualitativi aziendali, ai requisiti utente e agli indicatori e metriche che si è deciso di adottare. Ogni momento di revisione, verifica o validazione dei requisiti, della documentazione, dei processi interni o dei fornitori, della progettazione, dello sviluppo e del prodotto realizzato, degli indicatori di Qualità, è considerato un'attività di QA (vedi figura 2.5). Quindi anche le fasi di test funzionale e prestazionale appartengono al processo di QA del progetto: l'attività di testing è una

delle attività più significative e costose volte a garantire la Qualità del prodotto o servizio software [12]. L'output della QA consiste in una serie di proposte di miglioramento o di azioni correttive finalizzate ad incrementare l'efficacia del progetto ed a portare benefici aggiuntivi alle persone interessate al progetto stesso.

Figura 2.5: Posizionamento del Processo di Quality Assurance

In questa tesi ci focalizzeremo su come contribuire al miglioramento della Qualità del prodotto o servizio offerto attraverso l'introduzione di una specifica tipologia di test: il ***Performance Testing*** (vedi paragrafo 4.5).

TECNOLOGIA

Considerare gli aspetti tecnologici della Qualità del software, significa stabilire come deve essere realizzato il risultato di una o più attività delle fasi del processo di produzione. Questo significa scegliere ed utilizzare strumenti che adottino le tecniche in grado di supportare opportunamente la realizzazione del risultato di una o più attività . L'obiettivo da raggiungere è utilizzare gli strumenti che consentano di produrre risultati con la miglior efficacia ed efficienza possibile.

Alcune delle tecniche normalmente supportate ed abbinate a strumenti sono:

- tecniche ed ambienti di sviluppo del prodotto;
- tecniche ed ambienti di testing del prodotto.;
- tecniche ed ambienti per la definizione di metriche, sia relative al prodotto che al processo di produzione.

STANDARD DI RIFERIMENTO

Diversi enti di standardizzazione (es. ISO) hanno cercato di integrare vari approcci alla definizione della Qualità , partendo dalla consapevolezza che la Qualità è un attributo che varia in funzione del percettore, del contesto di percezione, dello scopo e del costo del prodotto.

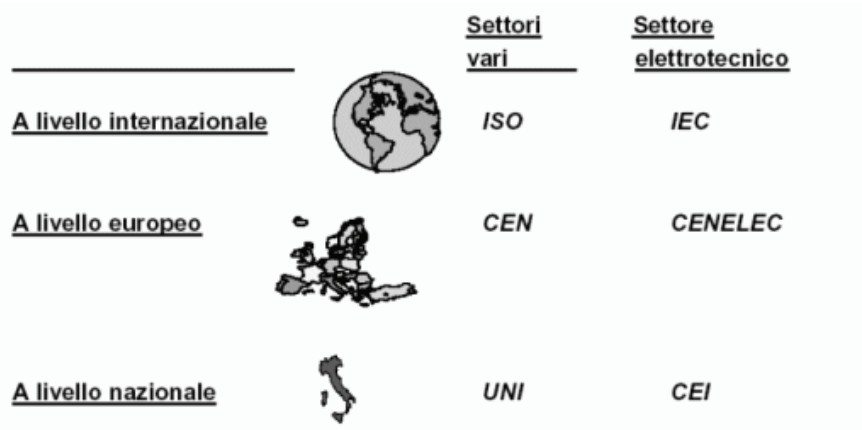

Figura 2.6: Enti di Standardizzazione

ISO (International Organization for Standardization) è la più grande organizzazione internazionale che si occupa di definire e diffondere standard[6] in tutti i processi produttivi. Gli standard che sviluppa sono in base alle richieste del mercato e vengono sviluppati da comitati di tecnici provenienti dal settore tecnico e industriale di tutti i paesi mondiali. Obiettivi della standardizzazione sono: facilitare le attività produttive, facilitare la comunicazione, migliorare gli interessi dei consumatori, facilitare scambi internazionali di beni e servizi, aumentare la sicurezza e migliorare la Qualità di vita. Generalmente l'adesione agli standard è su base volontaria, tranne per le norme cogenti, ovvero adottate come base di legislazione di alcuni paesi.

Nel momento in cui si scopre un processo migliore, questo diventa una best practice che, a lungo andare, presa in mano da un comitato tecnico ISO, può portare ad una nuova norma o al miglioramento di una esistente. Le norme sono specifiche tecniche (o altri documenti), messe a punto con la cooperazione ed il consenso di tutte le parti interessate, fondate sui risultati congiunti della scienza, tecnologia ed esperienza, miranti al vantaggio ottimale della comunità nel suo insieme ed approvate da un organismo qualificato sul piano nazionale, regionale o internazionale (ISO, 1978).

[6] Insieme di requisiti che definiscono proprietà di un prodotto o di un processo. Se quest'ultimi rispettano lo standard si dicono "conformi" allo standard.

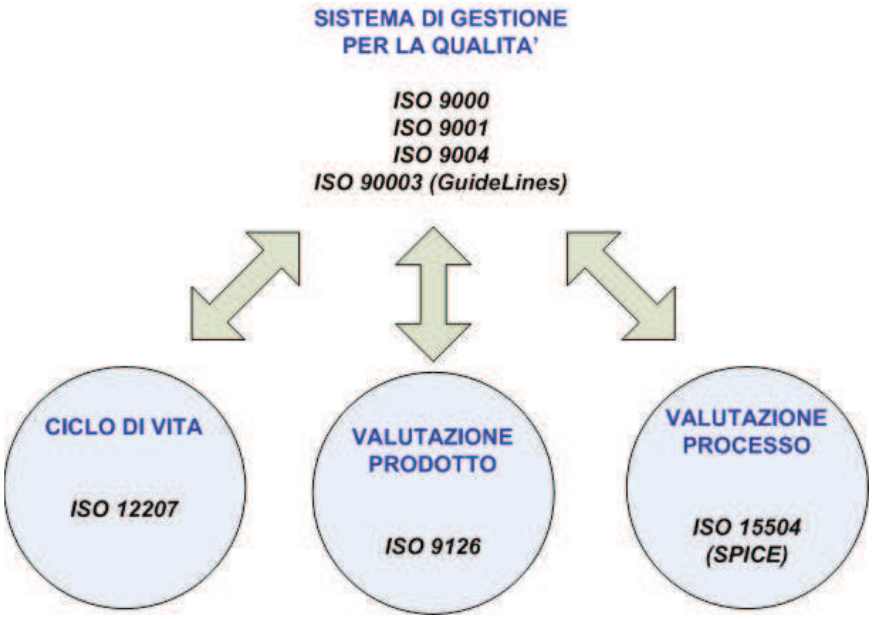

Figura 2.7: Principali normative per la Qualità del software

Vengono ora illustrati alcuni contributi sul tema della Qualità, ponendo l'enfasi sul concetto di Qualità di prodotto e di processo. Per descrivere questi contenuti si fa riferimento ai metodi e standard più conosciuti ed utilizzati del settore. In particolar modo si farà riferimento alle normative della famiglia ISO per la Qualità del software (figura 2.7): la serie **ISO 9000** che si occupa di definire le caratteristiche ed i requisiti di un sistema di Qualità aziendale, lo standard **ISO 9126**, che definisce principi generali ed un modello di Qualità del prodotto software, lo standard **ISO 12207** che determina i processi adottati lungo il ciclo di vita del software e la **norma ISO 15504**. Sebbene quest'ultima norma, nata dal lavoro del progetto internazionale **SPICE** (Software Process Improvement and Capability Method) non verrà illustrata in seguito è opportuno citarla in quanto contribuisce alla definizione Qualità del processo: infatti regolamenta in modo molto esteso tutti i passi e le attività necessarie per effettuare la valutazione dei diversi processi e per attribuire loro un "voto di maturità" , denominato "livello di capacità".

2.2.3 Requisiti qualitativi del prodotto

Un prodotto software è costituito da una *soluzione software* (codice) e dall'insieme di tutta la *documentazione di prodotto*, che comprende tutte le descrizioni realizzate durante le varie fasi del processo di sviluppo software: dalla descrizione del problema sino alla stesura del manuale utente, tecnico e di installazione. Inoltre è importante considerare la *documentazione di supporto* al processo di sviluppo: documento di pianificazione, piano della Qualità, descrizione dei casi di test utilizzati.

La Qualità di un prodotto software non dipende solo dalla soluzione software, ma anche dalla Qualità di tutti gli altri elementi di cui si compone il prodotto ed è intrinseca alla **Qualità del processo di sviluppo** (paragrafo 2.2.5). Poiché abbiamo visto che un prodotto si ottiene seguendo un determinato processo, la modalità con cui quest'ultimo viene eseguito influenza la Qualità del risultato finale.

Esiste un standard, denominato **ISO/IEC 9126** ("Information Tecnology - Software product evaluation - Quality characteristics and guidelines for their use"), pubblicato nella sua prima versione nel 1991, che definisce il modello dei requisiti qualitativi del Prodotto Software. Nella sua più recente versione, è descritta nelle pubblicazioni [13][14][15][16].

Con **"modello di Qualità del software"** si fa riferimento ad un insieme di caratteristiche (attributi) e di relazioni tra di esse, che possono essere usate per specificare i requisiti del software e valutarlo rispetto al grado di conformità a tali requisiti. Secondo il modello proposto dalla norma, la Qualità del prodotto si divide in (figura 2.8):

- **Qualità interna** (o intrinseca)**:** insieme di caratteristiche intrinseche del prodotto software, cioè a tutte quelle proprietà qualitative che lo caratterizzano durante le fasi di sviluppo e manutenzione del software stesso.
- **Qualità esterna:** descrive tutte le proprietà del software che lo caratterizzano durante la sua esecuzione. Corrisponde alla Qualità del prodotto che può venire testata in un ambiente che simula quello di reale utilizzo, inclusa una simulazione della base di dati con cui deve lavorare l'applicazione.
- **Qualità in uso** (o percepita): esprime il grado di raggiungimento delle aspettative utente nel contesto di utilizzo. Misura la distanza tra il modello cognitivo dell'applicazione che ha l'utente e quello che lo sviluppatore ha recepito e trasformato in prodotto. Tanto più questi modelli si avvicinano (attraverso tecniche di prototipazione, vedi paragrafo 2.2.4) tanto più il prodotto sarà in grado di raggiungere la soddisfazione dell'utente.

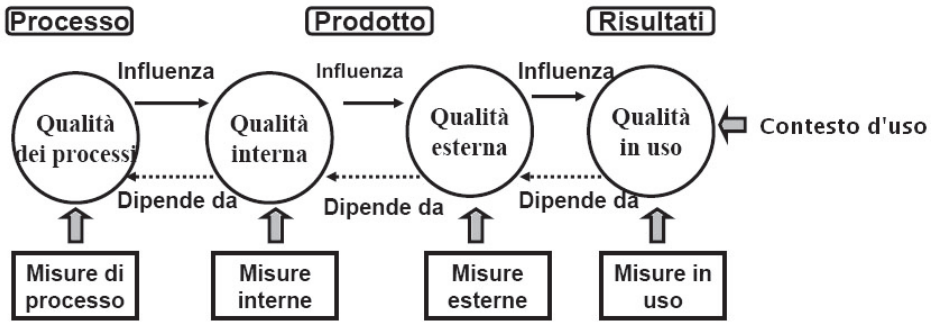

Figura 2.8: Approccio alla Qualità dei prodotti software

I diversi stakeholders[7] del progetto sono interessati a diversi aspetti della Qualità:

- **Utente:** interessato alla Qualità esterna, relativa alla modalità d'uso, alle prestazioni e agli effetti derivanti dall'uso del software.
- **Sviluppatore:** interessato alla Qualità interna, relativa agli aspetti tecnici legati allo sviluppo e alla manutenzione del software.
- **Gestore del sistema:** interessato agli aspetti di Qualità relativi all'installazione, configurazione ed esercizio del sistema.
- **Committente:** interessato alla Qualità globale, considerata come media pesata dei diversi aspetti di Qualità.

Tali viste di Qualità sono applicabili a qualsiasi tipologia di software. Per descrivere le tre forme di Qualità, lo standard ISO/IEC 9126 fornisce due modelli: uno per la Qualità interna ed esterna e uno per la Qualità in uso. Di seguito verrà descritta la prima parte dello standard relativa al modello di Qualità interna ed esterna [13], mentre per approfondimenti riguardanti il le metriche utilizzate per misurare la presenza delle caratteristiche di Qualità fare riferimento direttamente agli standard [14][15][16].

MODELLO DI QUALITA' INTERNA ED ESTERNA

Questo modello stabilisce che la Qualità del software dipende dalla presenza delle seguenti sei caratteristiche di Qualità: **funzionalità, usabilità, affidabilità, efficienza, manutenibilità** e **portabilità**. Un possibile schema che evidenzia il livello di interesse (alto, medio, basso), da parte degli stakeholders, sugli aspetti della Qualità elencati dalla norma è il seguente:

	Utente	Sviluppatore	Gestore	Committente
Funzionalità	Alto	Basso	Basso	Medio
Affidabilità	Medio	Medio	Alto	Medio
Usabilità	Alto	Basso	Medio	Medio
Efficienza	Medio	Medio	Alto	Medio
Manutenibilità	Basso	Alto	Medio	Medio
Portabilità	Basso	Alto	Alto	Medio

Tabella 2.1: Livello di interesse della Qualità da parte degli stakeholders

[7] È l'insieme delle persone che hanno interesse nel fatto che il progetto abbia successo (committente, utenti, amministratori dell'azienda, management aziendale, funzioni aziendali, fornitori) .

Vediamo il significato di ogni caratteristica attraverso le sottocaratteristiche, che meglio descrivono i diversi aspetti della Qualità.

- **FUNZIONALITÀ**: capacità di fornire funzioni tali da soddisfare, in determinate condizioni, requisiti funzionali espliciti o impliciti (il software fa quello per il quale è stato sviluppato). Per meglio definire che cosa s'intende per funzionalità del software lo standard indica cinque sottocaratteristiche.
  - **Adeguatezza:** capacità di svolgere un insieme di funzioni che risultino appropriate per un compito o un obiettivo specifico.
  - **Accuratezza:**capacità del prodotto software di fornire i risultati e gli effetti corretti e attesi.
  - **Interoperabilità:** capacità di interagire con altri sistemi.
  - **Conformità (o aderenza agli standard):** capacità del software di rispettare protocolli, leggi, convenzioni, prescrizioni e standard. Molti degli standard esistenti, in particolare quelli relativi alla comunicazione, sono proprio finalizzati a facilitare l'interoperabilità.
  - **Sicurezza:** capacità di prevenire accessi non autorizzati e di resistere ad attacchi intenzionali mirati ad accedere ad informazioni confidenziali.

- **AFFIDABILITÀ**: capacità di mantenere un livello adeguato di prestazioni sotto determinate condizioni e per un periodo di tempo fissato (il software reagisce bene a variazioni esterne).
  - **Maturità:** o robustezza, capacità di evitare blocchi dell'applicazione a seguito di errori nel software.
  - **Tolleranza ai guasti:** capacità del prodotto software di mantenere un determinato livello di prestazioni in presenza di guasti nel software. Si pensi a quei programmi che anziché interrompersi continuano a operare con funzionalità limitate.
  - **Ripristinabilità:** capacità di ripristinare livelli di prestazione predeterminati e di recuperare i dati a seguito di errori.

- **USABILITÀ:** capacità del software di essere compreso, appreso, usato con soddisfazione dall'utente in determinate condizioni d'uso (il software gestisce bene l'interazione con gli utenti).
  - **Comprensibilità :**impegno richiesto agli utenti per capire il funzionamento del software e la sua applicabilità.
  - **Apprendibilità**: impegno richiesto agli utenti per imparare ad usare il software.

- **Operabilità**: disponibilità delle funzioni essenziali per utilizzare correttamente il SW.
- **Attrattività /Piacevolezza**: capacità del software di essere piacevole per l'utente.

- **EFFICIENZA:** misura il rapporto tra il livello delle prestazioni del prodotto e la quantità di risorse che devono essere impiegate per raggiungere tali prestazioni (il software usa bene le risorse disponibili).
  - **Comportamento temporale:** si riferisce ai tempi di risposta e di elaborazione, nonché al tempo necessario per smaltire le richieste di esecuzione delle sue funzionalità.
  - **Consumo di risorse:** indica la quantità, il tipo di risorse utilizzate e la durata del loro impiego necessari per l'esecuzione delle funzionalità di un prodotto software.

- **MANUTENIBILITÀ:** capacità del software di essere modificato con un impegno contenuto (il software segue l'evoluzione dell'organizzazione).
  - **Analizzabilità:** impegno richiesto per diagnosticare carenze o cause di fallimento, o per identificare parti da modificare.
  - **Modificabilità:** impegno richiesto per modificare, rimuovere errori o sostituire componenti.
  - **Stabilità:** capacità di ridurre il rischio di comportamenti inaspettati a seguito di modifiche.
  - **Provabilità:** impegno richiesto per validare le modifiche apportate al software.

- **PORTABILITÀ:** facilità con cui il software può essere trasferito da un ambiente operativo ad un altro (il software segue l'evoluzione tecnologica).
  - **Adattabilità :** capacità da parte del software di adattarsi a nuovi ambienti operativi.
  - **Installabilità :** impegno richiesto per installare il software in un particolare ambiente.
  - **Coesistenza:** capacità del software di coesistere con altri software nel medesimo ambiente, condividendo risorse.
  - **Sostituibilità :** capacità di essere utilizzato al posto di un altro software per svolgere gli stessi compiti nello stesso ambiente.

Oltre che nella caratteristica di funzionalità, in ogni altra caratteristica del modello è presente l'attributo di conformità, inteso come il rispetto di regole e standard dettati da quella particolare caratteristica.

2.2.4 Qualità di processo

Il processo che porta alla realizzazione di una soluzione informatica complessa è esso stesso un'entità molto articolata. Descrivere il problema, progettare e sviluppare la soluzione, verificarne le caratteristiche di Qualità sono attività che richiedono grandi capacità personali, un'organizzazione adeguata e tecnologie appropriate ed efficaci. Risulta evidente che la Qualità del processo influenza la Qualità del prodotto realizzato: più il processo è di scarsa Qualità, maggiori saranno i rischi che la soluzione prodotta possa essere inadeguata rispetto agli obiettivi e ai bisogni per i quali è stata concepita.

Fondamentalmente esistono tre soggetti interessati a valutare la Qualità del processo software:

- **Committente:** ovvero chi acquista il prodotto utilizzando il budget a lui assegnato.
- **Utente finale:** spesso chi compra non è la stessa persona che usa il bene acquistato. Ad esempio la Direzione Sistemi Informativi di una grande Azienda acquista i prodotti che verranno utilizzati da tutti gli impiegati.
- **Fornitore:** attento ai bisogni della propria Clientela sarà per primo interessato a far sì che il proprio processo di sviluppo sia conforme ad elevati standard qualitativi.

In generale la Qualità di un processo viene valutata per l'affidabilità di un Fornitore oppure per la sua capacità di sviluppare software. Per valutare la bontà di un processo si identificano quattro attività principali:

- **Valutazione di maturità:** si studia il processo al fine di determinare il livello di Qualità attuale e definire dunque in quale stato il processo si trovi (*Assessment*).
- **Definizione di un programma di miglioramento:** alla valutazione di maturità segue la definizione di un programma di miglioramento per porre rimedio alle carenze precedentemente identificate. A prescindere dai diversi modi di miglioramento è essenziale il pieno coinvolgimento e impegno del management aziendale.
- **Attuazione del programma di miglioramento:** esecuzione e "messa in pratica" di quanto previsto dal programma di miglioramento.
- **Verifica dei risultati** raggiunti, consolidamento ed eventuale industrializzazione del nuovo processo.

Cambiare le modalità secondo la quale una organizzazione conduce le proprie attività è uno dei problemi considerati più critici perché richiede risorse

altamente qualificate e il pieno coinvolgimento di tutte le persone dell'organizzazione. Per raccogliere con successo questa sfida così complessa nel corso degli anni sono stati sviluppati una serie di metodi, approcci e linee guida.

Nel seguito saranno presentate le caratteristiche principali, gli approcci su cui si fondano e la loro utilità di due tra i metodi più conosciuti e utilizzati: la famiglia degli standard ISO 9000 e il CMMI.

STANDARD ISO 9000

La famiglia ISO 9000 è una collezione di documenti che definiscono le caratteristiche di un sistema di Qualità aziendale. Divisa in linee guida e standard veri e propri, viene applicata ad un vasto insieme di settori industriali e di servizi, tra cui alle imprese che progettano, sviluppano e gestiscono software. Viene utilizzata per verificare se il processo di sviluppo dell'Azienda considerata è conforme ai requisiti di Qualità contenuti nello standard.

Vi sono più enti di certificazione abilitati a poter verificare la conformità di un processo, o di un insieme di processi, allo standard ISO 9000. Normalmente, la certificazione avviene attraverso una visita ispettiva che verifica la presenza di un sistema di Qualità conforme allo standard e la sua piena applicazione all'interno dei processi aziendali. Nel caso di non conformità gravi, la certificazione viene negata o decade. In ogni caso la certificazione non ha durata illimitata, ma deve essere rinnovata nel tempo con visite ispettive a cadenza tipicamente annuale.

Gli standard che fanno parte della famiglia ISO 9000 si sono evoluti e modificati nel tempo: negli anni che vanno dal 1996 al 2000 è stata effettuate una revisione della struttura e dei contenuti della versione del 1994 e la versione più recente delle norme è la ISO 9000: 2000. In figura 2.9 è riportata la struttura delle norme ISO 9000:2000.

La **ISO 9001** [17], conosciuta anche con il termine VISION 2000, specifica i requisiti per un sistema di gestione per la Qualità che possono essere utilizzati sia in ambito interno alle organizzazioni, sia per la certificazione, sia nell'ambito di rapporti contrattuali. Essa focalizza l'attenzione sull'efficacia del sistema di gestione per la Qualità nel soddisfare i requisiti del Cliente. Uno dei principali requisiti che un'Azienda deve soddisfare per poter rendere i propri processi conformi allo standard ISO 9001 è la creazione del **Manuale della Qualità**. Tale manuale descrive tutte le procedure che l'Azienda si impegna a utilizzare per soddisfare i requisiti della norma ISO 9001. Devono inoltre essere definiti gli standard per i documenti utili per la realizzazione ed assistenza al prodotto software come: procedure, modelli standard dei documenti di progetto, guide, metodologie. Oltre ai requisiti di documentazione, il Manuale della Qualità definisce anche come l'Azienda si è organizzata per applicare, nei propri processi, gli altri punti della norma: Gestione delle risorse, Responsabilità della direzione, Pianificazione della realizzazione del prodotto, Processi relativi al Cliente, Progettazione e sviluppo, Approvvigionamento, Produzione e erogazione dei servizi, Monitoraggi e misurazioni, Miglioramento.

Figura 2.9: Struttura delle Norme ISO 9000:2000

Rispetto alla ISO 9001, la ISO 9004 [18] fornisce orientamenti per un più ampio raggio di obiettivi del sistema di gestione per la Qualità, in modo particolare per il miglioramento continuo delle prestazioni complessive e dell'efficienza dell'organizzazione, oltre che della sua efficacia. La ISO 9004 è raccomandata anche come guida per le organizzazioni la cui alta direzione desidera andare oltre i requisiti minimi della ISO 9001, per perseguire il miglioramento continuo delle prestazioni. La ISO 9004 non è tuttavia concepita per la certificazione, né per finalità contrattuali.

Il testo della ISO 90003 [19] è costruito come "precisazione" puntuale ai requisiti della ISO 9001. Questa norma non è utilizzabile come riferimento diretto per certificazioni del Sistema di Gestione per la Qualità di un'organizzazione che opera nel settore ICT. La ISO 90003 fornisce indicazioni riguardo i processi di acquisizione, fornitura, sviluppo, gestione operativa e manutenzione del software. È indipendente dalla tecnologia, dai modelli del ciclo di vita, dai processi di sviluppo, dalla sequenza delle attività e dalla struttura organizzativa utilizzata da un'organizzazione. La ISO 90003 contiene requisiti utili non solo allo sviluppatore ma anche all'acquirente (per definire i requisiti contrattuali) ed agli *auditor* (per valutare i processi delle organizzazioni).

CMMI (Capability Maturity Model Integration)

Definisce un modello di riferimento per analizzare lo stato del processo di un'Azienda ed in particolare il suo livello di maturità. Inoltre, fornisce supporto per la selezione delle aree critiche e dei punti in cui intervenire all'interno del processo considerato allo scopo di migliorarne il livello di maturità. Contrariamente allo

standard ISO 9000, in cui un processo può solo essere o meno conforme allo standard e quindi certificato, il CMMI va oltre il puro Sistema Qualità, che da quasi per scontato, ed esprime una gradualità di giudizio per la ricerca progressiva dell'eccellenza.

È stato sviluppato dal Software Engineering Institute (SEI) della Carnegie Mellon University di Pittsburgh (USA). Il CMMI attualmente pubblicato è la versione 1.1 del 2002, ed è prevista per Agosto 2006 la pubblicazione della versione 1.2 (operativa da gennaio 2007) [20].

Questo modello permette di definire il **livello di maturità** di un processo software esaminandone diverse caratteristiche. Il livello di maturità di un processo è valutato osservando il grado di reale applicazione di una serie di pratiche dell'ingegneria del software raggruppate in alcune macro aree:

- Gestione del processo: studio del processo e del suo miglioramento.
- Gestione del progetto: attività di pianificazione e controllo di un progetto e la gestione del rischio.
- Ingegnerizzazione: attività di sviluppo del prodotto software, come la definizione dei requisiti, la progettazione della soluzione, la verifica e la validazione
- Supporto: comprende pratiche che non rientrano nella realizzazione diretta del prodotto, come il Configuration Management.

La maturità di un processo è espressa attraverso una serie di livelli (per la precisione 6) che misurano la vicinanza dell'Azienda ad un modello ideale di processo (vedi figura 2.10). Ogni livello rappresenta uno stato raggiunto da parte della organizzazione, che corrisponde alla capacità di attuare in modo controllato determinate pratiche.

La definizione del livello di maturità avviene attraverso interviste che valutano il grado di esecuzione delle diverse pratiche ed in funzione delle informazioni ricavate si definisce con il management dove e come intervenire per migliorare le aree ritenute più deboli. Il modello CMMI può essere utilizzato per:

- riesame dei processi lavorativi di un'organizzazione per identificarne i punti di forza e di debolezza ed impostare azioni di miglioramento;
- valutazione di organizzazioni al fine di selezionare un Fornitore di beni e servizi;
- valutazione preventiva della capacità di un'organizzazione di rispettare dei requisiti contrattuali o di portare a termine con successo un progetto;

Oltre al sito Web, il modello CMMI è discusso in AHERN et al. [21].

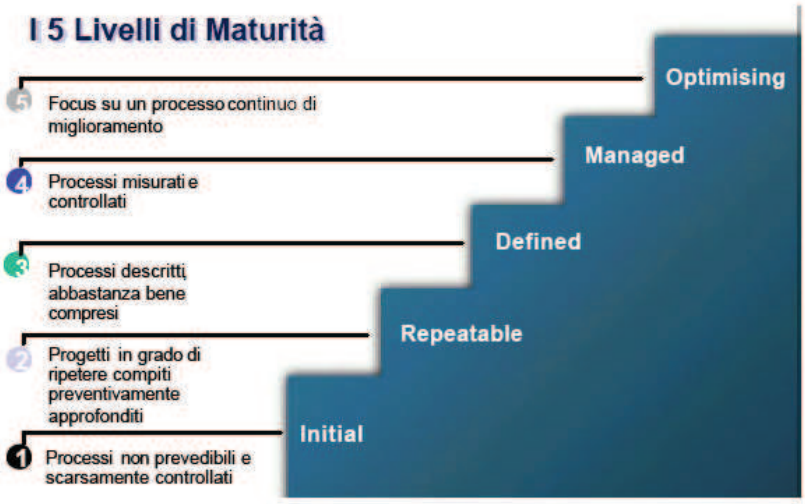

Figura 2.10: CMMI: Livello di maturità

2.3 Modelli di ciclo di vita di un prodotto software

Le situazioni in cui si trova a realizzare un progetto software sono le più varie, caratterizzate da diversi vincoli come per esempio: l'obbligo di terminare entro un tempo prefissato, obiettivi innovativi (e rischiosi), specifiche e caratteristiche difficili da definire, necessità di una realizzazione anche parziale ma rapida.

Per garantire il successo del progetto è opportuno adeguarne la conduzione alle diverse situazioni. A tal proposito l'esperienza ha definito un insieme di modelli per la conduzione dei progetti: i **cicli di vita**.

Il ciclo di vita rappresenta lo schema di massima di un processo software, caratterizzato da una strutturazione in fasi secondo le quali deve articolarsi e dai criteri, che definiscono le modalità di successione tra le varie fasi. Un ciclo di vita non definisce nel dettaglio come devono svolgersi le singole attività, ma descrive una logica di conduzione di quest'ultime. Nei paragrafi che seguiranno verranno illustrati e discussi i principali tipi di cicli di vita: a cascata, modello a V, modello incrementale, la prototipazione e il modello a spirale di BOEHM.

2.3.1 Ciclo di vita a cascata

Il primo modello di ciclo di vita strutturato è senz'altro il modello a cascata o *waterfall*.. Concepito negli anni '70 [22], introdusse una serie di concetti e idee che sono tuttora alla base di molti altri cicli di vita e delle tecniche utilizzate dagli ingegneri del software, malgrado i limiti che lo rendono poco utilizzabile in molte situazioni.

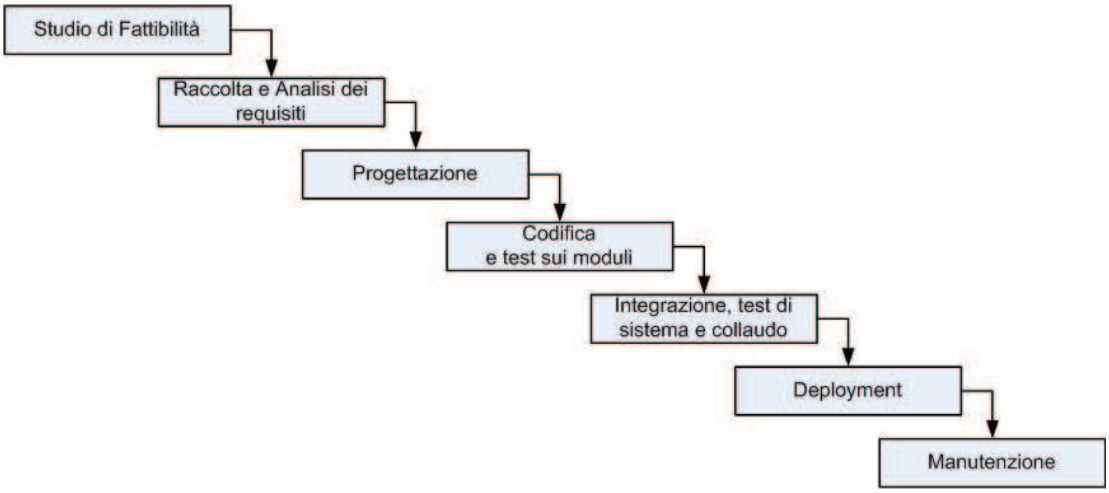

Figura 2.11: Modello a cascata

Il ciclo di vita a cascata è strutturato secondo una serie di fasi organizzate in modo sequenziale (figura 2.11); ogni fase genera un output che diventa l'input per la successiva. Nonostante in letteratura si possano trovare diverse descrizioni del ciclo di vita a cascata, che differiscono per numero e natura delle fasi, è possibile identificare uno schema di massima ragionevolmente condiviso. È importante evidenziare che la caratteristica del modello non è tanto costituita dal numero e dalla descrizione delle fasi quanto dal fatto che si passa alla fase successiva solamente quando la precedente è completamente terminata.

È buona norma che ogni fase è buona norma che termini con un documento formale approvato dal responsabile di progetto. In questo modello, eventuali ripensamenti (che implicano la messa in discussione dei risultati ottenuti da fasi precedenti) hanno un impatto molto rilevante sui tempi e sui costi e quindi sono da prevenire. Si elencano di seguito le fasi di un modello a cascata.

- **Studio di fattibilità:** si identificano le caratteristiche generali del progetto, si analizzano e verificano le sue potenzialità, limiti e vincoli e le possibili strategie di soluzione**.** Inoltre per ogni soluzione proposta si valutano i rischi e si effettua una valutazione di massima sui tempi e i costi richiesti. In questa fase l'interazione tra committenti e progettisti è la più intensa possibile. Il progetto può terminare in questa fase quando si prende la decisione di "non fare nulla"; contrariamente i dati raccolti devono consentire una pianificazione del lavoro da svolgere.
- **Raccolta dei requisiti e Analisi**: in questa fase viene dato forma ai requisiti funzionali e non funzionali del sistema (compresi i requisiti tecnologici), raccogliendoli in modo più o meno rigoroso, avendo, in ogni caso, cura di associare a ciascuno di essi una descrizione completa di tutte le condizioni che il sistema da realizzare deve soddisfare. Alla raccolta delle aspettative e richieste da parte degli utenti segue l'analisi, tipicamente da parte di esperti informatici e delle problematiche coinvolte, in cui vengono descritti i requisiti

che il sistema deve offrire in termini di funzionalità, dati manipolati e vincoli sul sistema e sul processo di sviluppo (o requisiti non funzionali)

- **Progettazione**: in questa fase viene prodotto il documento a maggior contenuto tecnologico, nel quale viene descritta in ogni dettaglio, l'architettura hardware e software del sistema da realizzare e quella del sistema di sviluppo (*System Design*). Per quanto riguarda la fase di progettazione del software, rispettando le funzionalità descritte nella fase precedente, sono documentati dettagliatamente tutti i moduli software da implementare (*Program Design*).
- **Codifica e test sui moduli**: i vari moduli dell'architettura vengono sviluppati in un opportuno linguaggio di programmazione prescelto e sulla base di un "piano" vengono eseguiti tutti i test di unità, per verificare la loro adeguatezza rispetto alla specifica di progetto. In questa fase l'output prodotto è rappresentato da programmi, documentazione per la manutenzione e manuale per l'utente.
- **Integrazione e test di sistema e collaudo**: questa fase si pone come l'obiettivo di integrare tutti i moduli sviluppati e testati separatamente e di verificare, mediante opportuni test, le funzionalità del sistema nella sua interezza e compattezza. Al test di sistema segue lo *User Acceptance Test*, ovvero il collaudo del prodotto realizzato effettuato direttamente dal Cliente in ambiente di produzione o del tutto simile a quest'ultimo.
- **Deployment:** distribuzione e installazione del software presso l'utente. In questa fase, conosciuta anche con il termine "messa in esercizio", si completa l'installazione dell'hardware e del software. Infine vengono addestrati e resi operativi gli utenti finali.
- **Manutenzione:** questa fase, che ha luogo dopo l'entrata in produzione del prodotto software realizzato, è molto critica all'interno di un processo software; in molte organizzazioni il costo complessivo speso in manutenzione di tutto il parco applicativo è pari, e talvolta supera, il 50% degli investimenti in nuovi progetti. Gli interventi di manutenzione sono dovuti per la maggior parte o al cambiamento delle esigenze degli utenti o ad adeguamenti a nuove infrastrutture. La manutenzione dovuta al cambio di esigenze dell'utente potrebbe venire limitata, sia nei tempi che nei costi, da una migliore progettazione.

Il modello "a cascata" prevede oltretutto delle attività trasversali di *gestione* che non sono circoscrivibili all'interno di un singola fase, ma che risultano sostanzialmente ortogonali all'intero ciclo di vita. Queste attività riguardano il *controllo della Qualità* in ogni fase per assicurare che il lavoro proceda con quanto pianificato e concordato. La verifica e la validazione sono attività che vanno condotte lungo tutto l'arco del processo produttivo ed evolutivo (vedi paragrafo 4.1); sarebbe sbagliato ridurle alle sole fasi di test di unità, integrazione e test di sistema.

Altra attività cardine per la riuscita di un buon progetto, prevalentemente organizzativa, è la *gestione del progetto:* si occupa della pianificazione delle attività, del dimensionamento delle Risorse Umane ed economico-finanziare e del controllo dello sviluppo temporale del progetto.

Il ciclo di vita a cascata nasce come reazione alla mancanza di struttura e organizzazione che ha caratterizzato l'attività di sviluppo software negli anni '60. Per questo motivo si presenta come un modello rigido e altamente strutturato. Alcune caratteristiche del ciclo di vita a cascata sono ormai superate e non si adattano più alla realtà odierna. Uno dei problemi è la mancata interazione con l'utente finale durante il processo di produzione. Spesso è difficile enunciare esplicitamente tutti i requisiti e il modello a cascata esige il contrario: non è in grado di governare in modo adeguato l'incertezza che si accompagna in particolare nelle fasi iniziali di un progetto.

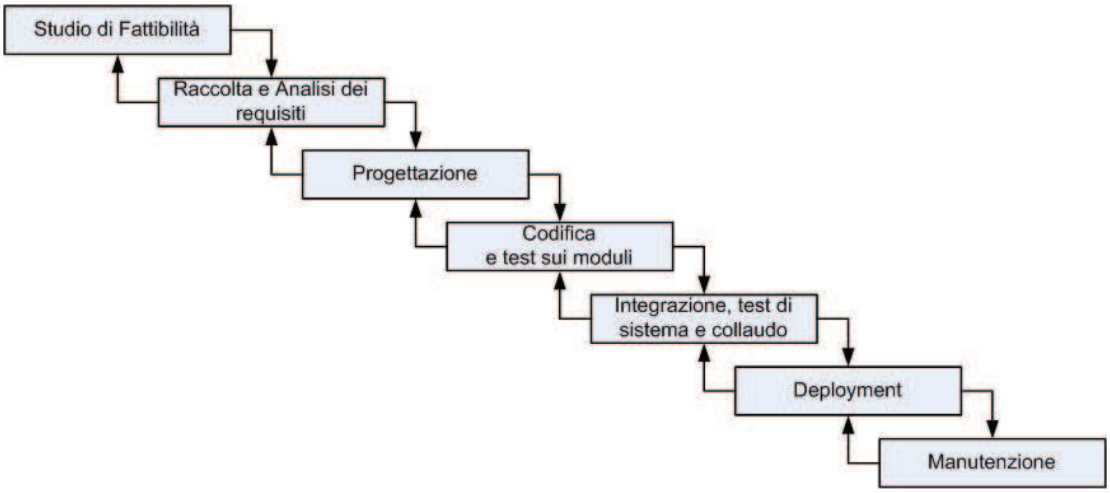

Figura 2.12: Modello a cascata con "ricicli locali"

Un'alternativa al modello standard è il **ciclo di vita con "ricicli locali"** (figura 2.12), che prevede la possibilità, ad ogni fase, di tornare alla precedente. Tale modello permette di verificare, alla fine di ogni fase, completezza e coerenza dell'output di quella fase e di intraprende le attività necessarie prima di proseguire. Il vantaggio di adottare questo modello è che gli eventuali ricicli locali, generalmente, richiedono poco lavoro aggiunto, quindi le previsioni sui tempi e sui costi sono molto attendibili.

Con l'evoluzione del mondo del software, sempre più dinamico e soggetto a rigidi cambiamenti, i metodi tradizionali sequenziali hanno dimostrato di avere parecchie lacune e di non essere più completamente adatti a soddisfare i requisiti moderni dello sviluppo software. Il problema principale è legato alla sequenzialità del processo che richiede che la definizione delle specifiche sia contenuta in un'unica fase confinata in un ben determinato intervallo all'inizio del progetto.

Il modello si applica bene quando i requisiti dell'utente sono strettamente determinabili prima dell'inizio dell'implementazione, così come l'architettura del sistema. Inoltre il committente accetta che l'insieme di funzionalità realizzate vadano in esercizio in blocco, alla fine del progetto.

Un caso tipico in cui è ragionevole implementare il modello a cascata è il rifacimento di un sottosistema del sistema informativo di una Azienda: in questo caso gli utenti sanno cosa aspettarsi (le funzioni del vecchio sistema più quelle richieste, provenienti dall'esperienza maturata).

È evidente che in molti contesti sia utopistico pensare di congelare la definizione delle specifiche e la progettazione del sistema in una fase predefinita del processo. Nella realtà accade che è necessario effettuare manutenzione aggiuntiva per correggere errori ed introdurre ulteriori fasi di requisiti, non pianificate in partenza, per tenere conto delle mutuate esigenze del committente e del dominio applicativo (**change management**). La criticità dei metodi tradizionali ha indotto la nascita di metodi di sviluppo iterativi, ovvero di modelli flessibili in cui viene abbandonata l'idea di un processo monolitico che porti al rilascio del software.

2.3.2 Modello a V

Il **modello a V** è nato per dare una ulteriore rappresentazione, a partire dal classico modello a cascata, delle relazioni che intercorrono tra le varie componenti del ciclo di vita del software. In questo modello sono dunque possibili dei ricicli non solo alla fase precedente, ma a quelle alle fasi alla stessa altezza o simmetriche.

In questo modello si pone l'enfasi sulle attività di **Verifica** e **Validazione** di ogni fase, considerate attività essenziali per garantire la Qualità del prodotto software che viene realizzato: ogni "strato" produce dei risultati che devono essere verificati in una specifica fase di verifica o di validazione. Inoltre è particolarmente adatto quando il committente accetta, come per il modello a cascata, che i risultati siano consegnati congiuntamente alla fine, però non è in grado di esplicitare in modo definitivo i requisiti degli utenti e la tecnologia di sviluppo ed esercizio non è stabile oppure è poco conosciuta.

Come nel modello a cascata, il modello a V stabilisce l'obiettivo del progetto attraverso lo studio di fattibilità; in questo caso la fase è particolarmente delicata perché i requisiti degli utenti e/o la tecnologia non sono completamente noti e quindi, le conseguenti decisioni non possono essere ritenute definitive. È anche importante in questa fase e nelle tre successive (analisi dei requisiti e specifica, progettazione del sistema e dei moduli software) che le decisioni prese siano tra loro il più possibile indipendenti in modo tale che, quando si ricicla, si possano realizzare facilmente solo le correzioni conseguenti alla decisioni modificate.

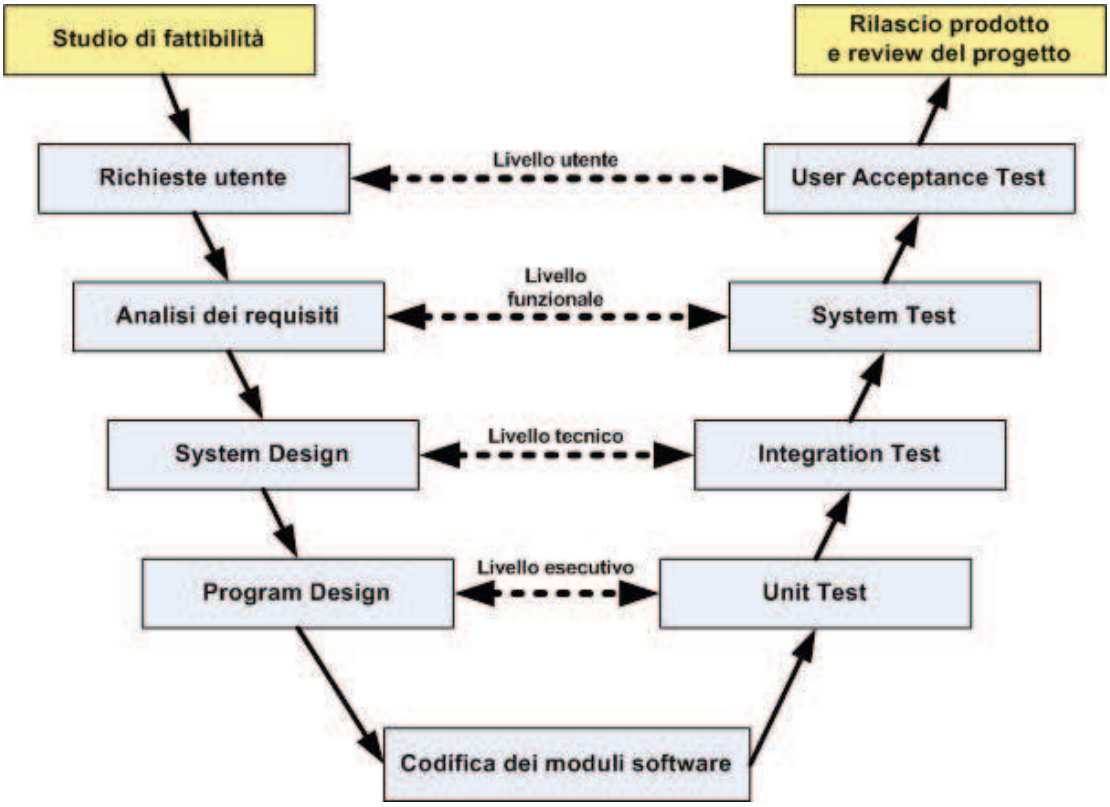

Figura 2.13: Modello a V

Come si può vedere dalla figura 2.13, ad ogni fase del modello a cascata corrisponde una determinata tipologia di test: dal test di ogni singolo modulo sino al test di accettazione da parte del Cliente ed il rilascio del prodotto realizzato. Durante la fase di *Unit Test* sono possibili non solo interventi a livello di programmazione o codifica, ma anche ripensamenti all'accorpamento in moduli delle varie funzionalità e quindi un riciclo verso la fase di progettazione dei singoli moduli software. Per ogni modulo accettato viene predisposta la relativa documentazione utente. Terminato il test di tutti i programmi, viene eseguito un test di integrazione per controllare se l'esecuzione dell'insieme dei moduli prodotti è correttamente supportato dall'architettura hardware e software del sistema, definita durante la fase di *System Design*. Segue la verifica di proprietà globali del sistema, sia funzionali che non funzionali, attraverso il *System Test.* Le richieste dell'utente sono confermate o meno attraverso un test di accettazione finale (collaudo), eseguito da persone interne, oppure, preferibilmente da un insieme di utilizzatori pilota, a cui viene rilasciato il prodotto in prova (*alfa e beta test*). Il flusso termina con una fase finale di revisione critica di tutto il progetto, necessaria affinché Fornitore, committente ed utilizzatori inneschino un processo di "apprendimento" globale che permetterà loro di gestire i progetti futuri con la consapevolezza e le conoscenze acquisiti dall'elaborazione dell'esperienza (*lesson learned*). Per ulteriori dettagli

relativi alla diverse tipologie di test si veda il paragrafo 4.2. Un modello di questo tipo si adatta particolarmente bene a gestire i progetti di sviluppo software nei quali la responsabilità del ciclo di vita è ripartita tra Azienda committente (che si occupa del ramo ascendente del modello a V) e Azienda fornitrice (alla quale è affidato il ramo discendente).

2.3.3 Modello incrementale

Come è stato definito precedentemente, il modello a cascata si basa su due presupposti fondamentali:

- il committente accetta che tutto l'output del progetto venga messo in esercizio in blocco nell'ultima fase temporale del progetto;
- i requisiti degli utenti e la tecnologia sono ben noti e accettati sia dal committente che dallo sviluppatore.

Quando non è valido solo il primo dei due presupposti si adotta il modello "a consegna incrementale", che consente al committente di avere, entro tempi brevi, una prima consegna dei risultati, anche se parziali ma funzionanti, e consegne successive durante lo svolgersi del progetto. Questa situazione si può presentare quando esigenze estranee al singolo progetto e alla sua economicità impongono che alcuni sottosistemi siano messi in esercizio prima di altri.

In questo modello (figura 2.14) si distingue un blocco di fasi iniziali, dette di *setup*, e un ciclo iterativo di consegne che viene ripetuto più volte finché l'intero progetto non è terminato (con l'ultima consegna).

Dal punto di vista globale, si differenzia dai modelli precedenti, perché prima occorre determinare l'ambiente tecnologico ed il piano delle consegne e solo successivamente viene fatta l'analisi dettagliata dei problemi da risolvere.

SETUP

- **Studio di fattibilità:** oltre ad osservare gli stessi obiettivi definiti per il ciclo di vita a cascata, questa fase deve prevedere i rischi conseguenti all'eventualità di dover modificare non solo parti del sistema appena realizzate (come nei modelli precedenti), ma anche parti del sistema già in esercizio.
- **Raccolta dei requisiti degli utenti e identificazione dei prodotti da consegnare o *deliverable*:** stesura di un documento che descrive in maniera dettagliata i requisiti degli utenti e fissa i gruppi di funzionalità associati a ciascuna delle singole consegne in cui si dovrà articolare il progetto.
- **Pianificazione delle consegne:** differentemente dai modelli precedenti, una pianificazione è possibile solo in questa fase, perché solo alla fine della fase precedente sono stati definiti il numero e la complessità dei cicli di consegna.

- **System design e Macroanalisi:** l'obiettivo principale di questa fase è definire l'ambiente tecnologico e l'architettura opportuni per il progetto.

CICLO DI CONSEGNE

- **Increment design:** analisi dei problemi connessi con le funzionalità associate alla consegna in esame. Corrisponde alla fase di analisi degli altri modelli.
- **Increment building:** in questa fase si fa la progettazione dei componenti e la codifica di tutte le funzionalità in esame; si prepara la documentazione di manutenzione e la formazione per l'utente e vengono eseguiti i test.
- **Increment deploy:** viene eseguito il test d'integrazione dei nuovi moduli con quelli già rilasciati in produzione, l'installazione, l'addestramento e la messa in esercizio delle funzionalità in esame e si prepara la documentazione di gestione.
- **Valutazione dei risultati:** a seguito della consegna appena fatta occorre rivedere il contenuto e la pianificazione delle altre consegne, l'architettura dell'intero sistema e la macroanalisi.

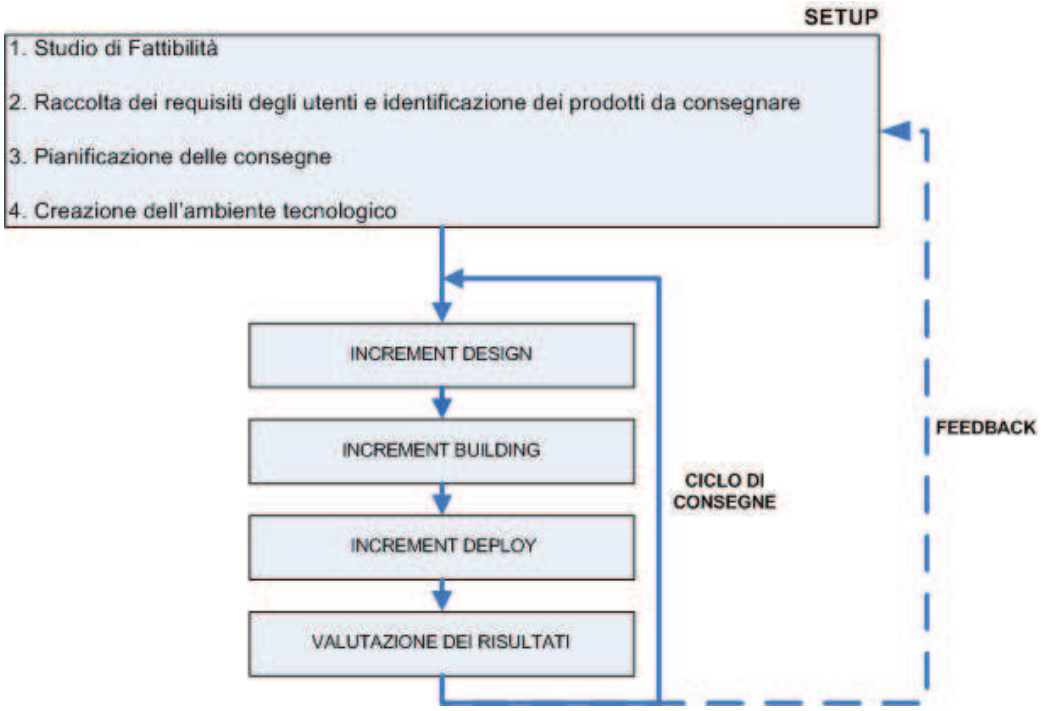

Figura 2.14: Modello incrementale

Alla fine di ogni ciclo, l'ultima fase permette di decidere se riciclare per produrre nuove consegne, ritornare a una delle fasi di *setup* per una modifica o ritenere il progetto terminato. Ovviamente, anche con questo modello, è assolutamente necessario che l'output di ogni fase sia adeguatamente documentato.

I vantaggi di questo modello sono dovuti principalmente alle consegne distribuite nel tempo e dal fatto che il *feedback* delle prime consegne influenza le successive, ciascuna consegna è piccola e facile da gestire, c'è una migliore interazione con l'utente perché ottiene immediatamente dei risultati tangibili e se il progetto dovesse essere sospeso, interrotto o cancellato, resterebbero comunque dei risultati utili (le consegne già effettuate).

Condizioni sfavorevoli che si possono presentare nell'adottare questo modello sono: il *breakage*, cioè una consegna può richiedere il rifacimento di consegne precedenti (o parti di esse), ed una produttività (quantità di prodotto/durata del progetto) inferiore del modello a cascata.

2.3.4 Prototipazione

Il paradigma prototipale è un primo esempio di modello flessibile in cui il processo di sviluppo procede per tentativi, in modo da adattarsi in maniera flessibile all'incertezza dei requisiti utente e architetturale.

Questo approccio permette al progettista di considerare ogni volta un sottoinsieme nuovo di requisiti da prendere in considerazione, basandosi sui risultati ottenuti. Vale a dire che occorre procedere con piccole realizzazioni, coinvolgendo eventualmente un campione di utenti ridotto, per poi discutere su quanto realizzato e mettere a punto le funzionalità necessarie.

Si possono adottare due approcci:

- **THROW-AWAY:** consiste nella creazione di prototipi "usa e getta" come una modalità per rappresentare le funzionalità del sistema e ridurre i rischi di errori che si verificano durante l'analisi dei requisiti. Il prototipo è sviluppato a partire da una specifica iniziale, rilasciato per farne delle prove e ragionare con l'utente e alla fine eliminato. Le iterazioni terminano quando è stato individuato l'insieme dei requisiti necessari per l'implementazione del sistema finale.
- **EVOLUTIVO:** lo sviluppatore implementa i requisiti inizialmente stabiliti in modo grezzo. Il prototipo, nel quale inizialmente sono implementate solo alcune funzionalità, viene presentato ed eventualmente rilasciato all'utente per verificare ciò che è stato realizzato e per definire ulteriori requisiti. In questo caso il ciclo riprenderà dalla fase di raccolta dei requisiti. Differentemente dal modello incrementale, che imponeva di implementare in progressione i requisiti conosciuti, questo approccio suppone che l'utente non conosca realmente tutte le esigenze e quindi debba interagire con il sistema per identificarle.

I principali vantaggi di questo modello sono riassunti dalla formula *learning by doing* e si possono elencare come segue:

- migliore comunicazione con gli utenti e coinvolgimento di quest'ultimi nel progetto
- i requisiti vengono definiti e validati operativamente
- si raggiunge facilmente la completezza e soprattutto la compatibilità e coerenza delle specifiche funzionali e tecnologiche
- i cambiamenti sono quasi tutti evidenziati durante la fase di sviluppo e non quando il sistema è diventato operativo

Gli svantaggi del modello sono una mancanza di controllo sugli obiettivi del progetto, in quanto l'utente può essere tentato ancor di più di chiedere "troppo" ed è difficile mantenere il controllo sui costi e sui tempi ipotizzati.

2.3.5 Ciclo di vita a spirale

Il modello a spirale è un modello di processo del software che abbina la natura iterativa della prototipazione e gli aspetti controllati e sistematici del modello sequenziale lineare, ma cerca di ridurre la possibilità di errori durante il processo software e, nel caso in cui si verifichino, di contenerne i danni.

In questo modello, proposto da BOEHM nel 1988 [23], il software viene sviluppato in una sequenza di versioni crescenti; le attività che portano alla creazione di un rilascio, sono ciclicamente ripetute per ogni versione successiva, secondo una spirale. Al termine delle prime iterazioni, i rilasci prodotti saranno documenti o prototipi; al termine delle ultime iterazioni saranno le versioni sempre più complete del prodotto finale. Ad ogni iterazione del ciclo di vita è legata una struttura comune detta *task region*. Ogni task region è costituita da un insieme di attività che possono variare in funzione del grado di avanzamento dell'intero processo.

Nella versione originale (figura 2.15) le *task region* sono quattro:

- **Definizione degli obiettivi, delle alternative e dei vincoli**
- **Valutazione delle alternative, identificazione e risoluzione dei rischi,** dove la novità consiste nell'introduzione dei concetti di *Risk Analysis*.
- **Sviluppo e verifica**, le attività che caratterizzano questa *task region* variano a seconda dell'iterazione considerata (dalla definizione delle specifiche sino ai test sul prodotto realizzato).
- **Pianificazione della fase successiva**, ripetere la pianificazione ad ogni iterazione permette di adattare le iterazioni successive alla reale evoluzione del progetto.

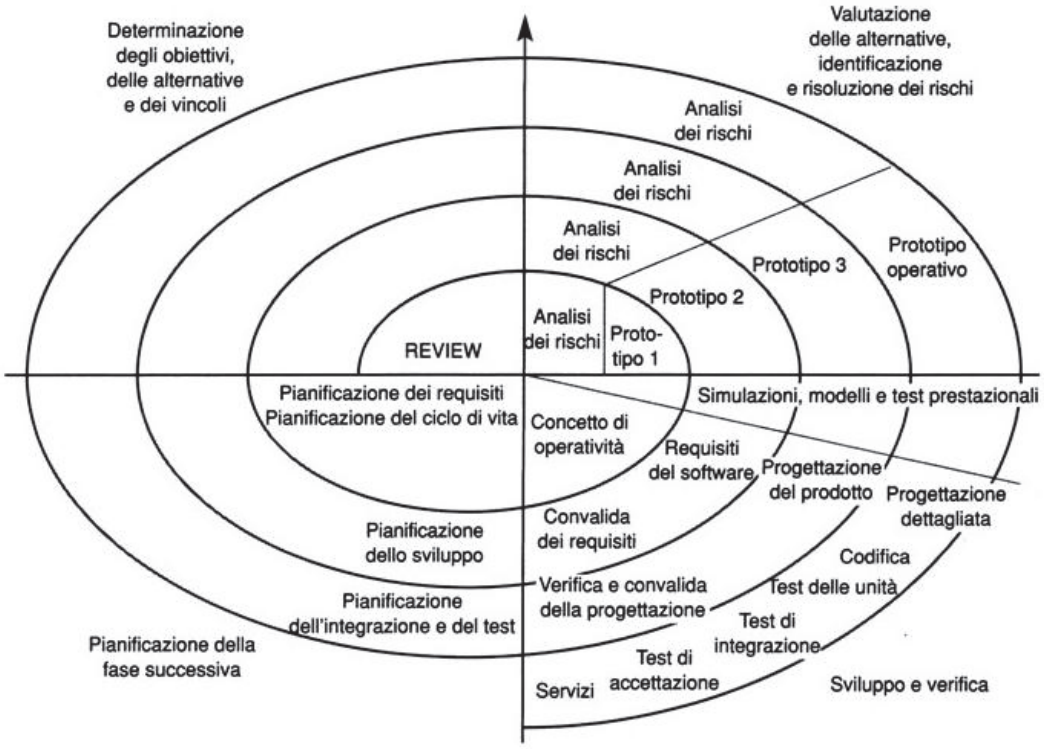

Figura 2.15: Ciclo di vita a spirale di BOEHM del 1988

Dal ciclo di BOEHM definito originariamente ne sono derivate molte varianti che differiscono per la struttura originale e per le attività che formano le task region. Lo stesso BOEHM ha modificato il suo modello per gestire i cambiamenti e le evoluzioni del processo di produzione di un prodotto software (figura 2.16), ottenendo le seguenti *task region* [24]:

- **Comunicazione con il Cliente:** all'inizio di ogni iterazione si discutono i requisiti desiderati dal Cliente.
- **Pianificazione:** vengono pianificate le attività da svolgere, tempistiche, identificazione e allocazione delle risorse da usare nella iterazione considerata. La pianificazione ad ogni iterazione permette di gestire le risorse in modo efficiente al variare delle caratteristiche del progetto.
- **Analisi dei rischi:** viene effettuata la stima e la prevenzione dei rischi tecnici e di gestione.
- **Progettazione:** task region in cui si progetta il nuovo rilascio a seguito di nuovi confronti con il Cliente emersi durante l'ultima iterazione (re-ingegnerizzazione della soluzione).

- **Realizzazione e rilascio:** viene effettuata l'implementazione ed il test degli elementi concordati in fase di progettazione.
- **Valutazione da parte del Cliente:** vengono rilevate le reazioni da parte del Cliente per ridurre e prevenire possibili problemi di fraintendimento e di incomprensione degli obiettivi.

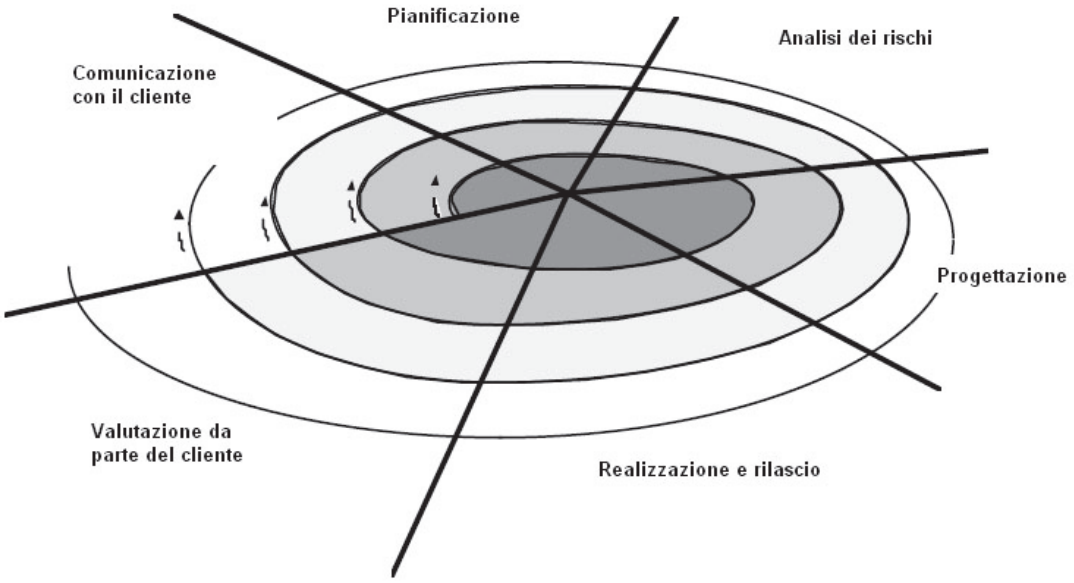

Figura 2.16: Ciclo di vita a spirale di BOEHM del 1998

Il modello di ciclo di vita a spirale descrive al meglio il processo di realizzazione di prodotti software complessi e la presenza di *task region* mantiene la sistematicità propria del modello a cascata. I vantaggi che il modello presenta sono: introduzione dell'analisi del rischio, riusabilità del codice, integrazione di sviluppo e manutenzione. Per contro, uno degli svantaggi di questo modello è la necessità di un buon grado di esperienza nella valutazione dei rischi; se eseguita correttamente, essa influenza in modo significativo le decisioni prese nel progetto, e non solo a livello implementativo.

2.3.6 Qualità nel ciclo di vita

La norma ISO/IEC 12207, pubblicata nel 1995 [25], è il primo esempio di definizione rigorosa dei Processi per un particolare settore merceologico, al di là della generica struttura dei processi aziendali, riportata nelle ISO 9001. Lo standard ISO/IEC 12207, opera di 6 anni di lavoro, supera l'approccio modellistico di un unico livello di processo (descritto nei cicli di vita del software) approdando alla definizione di una molteplicità di processi correlati tra loro.

La norma si occupa di definire un modello dei processi, attività, task e risultati (*outcomes*) che compongono il ciclo di vita di un prodotto software. Comprende tutte le fasi del ciclo, dalla analisi dei requisiti, fino ai servizi post-rilascio, di manutenzione e gestione, ed al ritiro della versione obsoleta. Inoltre descrive i processi che devono essere messi in atto per gestire, controllare, documentare e migliorare il ciclo di vita. L'ISO 12207 non si occupa, al contrario, di definire nel dettaglio cosa deve essere fatto per eseguire i processi e non prescrive l'utilizzo di un particolare ciclo di vita rispetto ad un altro. L'obiettivo dunque è di modellare i cicli di vita classici, definiti precedentemente utilizzando come componenti i processi e le attività descritte dall' ISO 12207.

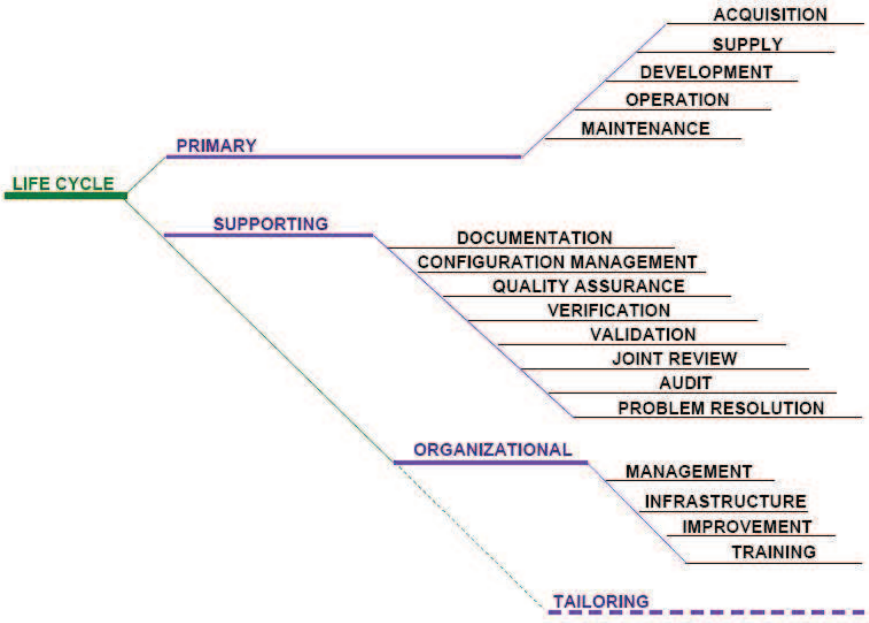

Figura 2.17: Struttura della norma ISO/IEC 12207:1995

La norma definisce 3 categorie di processi:

- **Primari**, processi orientati alla generazione di prodotti e servizi diretti all'utente finale. Riguardano sia il Cliente che il Fornitore e coprono l'intero ciclo di vita del software. Questa tipologia è costituita da 5 processi: *Acquisizione*, *Fornitura*, *Sviluppo*, *Esercizio (Operation)* e *Manutenzione*.

- **Supporto**, sono otto: *documentazione*, *gestione della configurazione*, *assicurazione Qualità*, *verifica*, *validazione*, *riesami* e *ispezioni* e *risoluzione dei problemi*. I processi di supporto sono attuati dal Fornitore e contribuiscono al successo e alla Qualità di un progetto; non sono dipendenti da una fase specifica, poiché sono trasversali a quelli primari e distribuiti lungo l'intero ciclo di vita.

- **Organizzativi:** gestione dello sviluppo, gestione delle infrastrutture, gestione del miglioramento e formazione. coprono gli aspetti manageriali ed organizzativi di un progetto, quali la pianificazione ed il controllo, ma includono anche l'addestramento delle risorse e la gestione delle infrastrutture, nonché quanto viene messo in atto per migliorare e controllare gli altri processi.

Per ognuno di tali processi devono essere chiaramente evidenziati: l'obiettivo e le responsabilità, la lista delle attività che lo compongono, i singoli compiti (task) nei quali è suddivisa ogni attività. Ogni compito o task deve indicare cosa una azione deve fare (*"what to do"*) e non come la deve fare (*"how to do"*). Inoltre la norma per ogni processo fa propria la sequenza **Plan-Do-Check-Act (PDCA)**, conosciuto in letteratura anche con il termine *ciclo di Deming*: pianifica, controlla, esegui, e intervieni per opportune correzioni (figura 2.18).

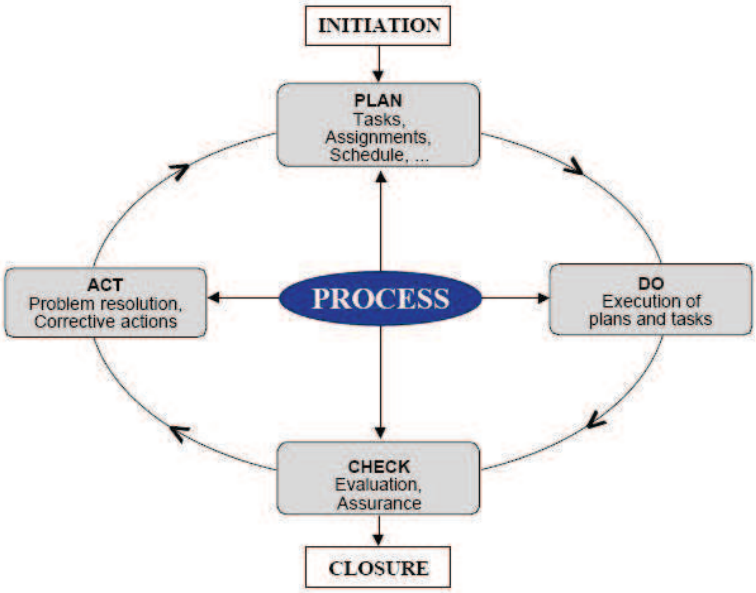

Figura 2.18: Plan-Do-Check-Act (PDCA) per i processi ISO 12207

I processi, le attività ed i tasks dello standard sono stati progettati in modo da poter essere adattati (*tailored*) a specifiche esigenze (ambienti, progetti, domini applicativi, etc.). Per questo è stato definito il processo speciale di **Tayloring**, come si nota in figura 2.17, che descrive cosa fare per "adattare"gli altri processi software. Il tayloring consiste sia nella cancellazione di processi, attività e tasks non applicabili al caso specifico, sia nell'aggiunta di processi, attività tasks eccezionali o speciali che debbono essere esplicitamente definiti (ad esempio, nei contratti).

Questo framework è stato pensato per poter essere un riferimento unico e condiviso per i managers, clienti, fornitori, sviluppatori e addetti alla Quality Assurance. Aiuta gli acquirenti di servizi IT a definire nei documenti contrattuali le

specifiche dei processi da richiedere ai fornitori (maggiore affidabilità), permette di comparare le prestazioni di più fornitori rispetto ad un modello comune (maggiore efficienza) e i fornitori trovano a loro volta gli elementi necessari per definire i contenuti di una offerta tecnica basata su una terminologia comune con gli acquirenti.

2.4 Metodologie di sviluppo

Molte organizzazioni aziendali si sono progressivamente rese conto di quanto sia importante possedere un processo per la produzione di software, definito in modo rigoroso e ben documentato. Seguire una particolare metodologia di sviluppo consente di ottenere dei prodotti software che rispondano a precisi requisiti di Qualità. Nella pratica accade che le aziende accumulano conoscenze sui propri processi di produzione (*know-how*), che però vengono in genere abbandonate e dimenticate quando i vari progetti sono portati a termine. Di conseguenza la necessità di possedere un processo razionale, definito ed utilizzato in tutti i progetti aziendali, è una priorità molto sentita nella maggior parte delle aziende, soprattutto in quelle di dimensioni medio-grandi.

2.4.1 Unified Software Process

Lo ***Unified Software Process*** [26] , o più comunemente ***Unified Process*** (d'ora in avanti si utilizzerà l'acronimo UP) , nasce nel 1999 ed è un framework generale di processo (modello), universalmente riconosciuto, da adattare alle diverse tipologie di progetto. In questo modello ricopre un'importanza rilevante lo strumento di visualizzazione UML, un linguaggio semiformale e grafico per specificare, visualizzare, realizzare e documentare le caratteristiche di un sistema software. La versione commerciale di UP è un prodotto della IBM/Rational: ***RUP (Rational Unified Process)***, utilizzato in contesti di business estremamente competitivi e in ambiti applicativi eterogenei. Risponde agli obiettivi primari di time-to-market, controllo del rischio e visibilità degli stati di avanzamento.

UP è basato su quattro assiomi fondamentali:

- Guidato dai **casi d'uso** e dai **fattori di rischio**: le modalità d'uso del sistema (casi d'uso) costituiscono la base per la definizione, negoziazione dei requisiti e validazione di quest'ultimi da parte del committente. Inoltre è importante individuare il prima possibile gli eventuali rischi che si possono dover affrontare nel processo di sviluppo e predisporre le modalità con cui possano essere superati.
- Incentrato sull'**architettura**: l'architettura applicativa e tecnologica di un sistema costruisce il fondamento tecnico dell'applicazione e del progetto. Rappresenta gli aspetti strategici di un sistema (come è scomposto in componenti e come questi componenti interagiscono); fintantochè che l'architettura non è consolidata, non esistono elementi sufficienti per

determinare con precisione (nella definizione del contratto) tempi, costi e rischi dell'intervento progettuale.

- **Iterativo** e **incrementale**: il progetto viene scomposto in sottoprogetti di dimensioni e complessità minori da svolgere in singole iterazioni (periodi di lavoro) e, a sua volta, i sottoprogetti in componenti e sottocomponenti, ognuno dei quali porterà al rilascio di nuove funzionalità. Con componente s'intende una unità di *deployment* indipendente, unica, e di "composizione con terzi", ovvero nella maggior parte dei casi utilizzata con altri componenti. In ogni iterazione si svolgono in misura e percentuale diversa le tipiche attività di sviluppo software (es. gestione dei requisiti, design, implementazione, test). Al termine di ogni iterazione è necessario un momento di verifica, e se l'esito è positivo, il progetto passa alla iterazione successiva. Inoltre è fondamentale avere un *feedback* di quanto fatto all'iterazione precedente, in modo da favorire eventuali cambiamenti occorsi durante il progetto (obiettivi e requisiti) al fine di andare incontro alla richieste del Cliente. Ogni iterazione si conclude con la produzione di un *deliverable* frutto del lavoro svolto (es. documento, prototipo).
- Basato sulla programmazione **object-oriented** e programmazione a **componenti**.

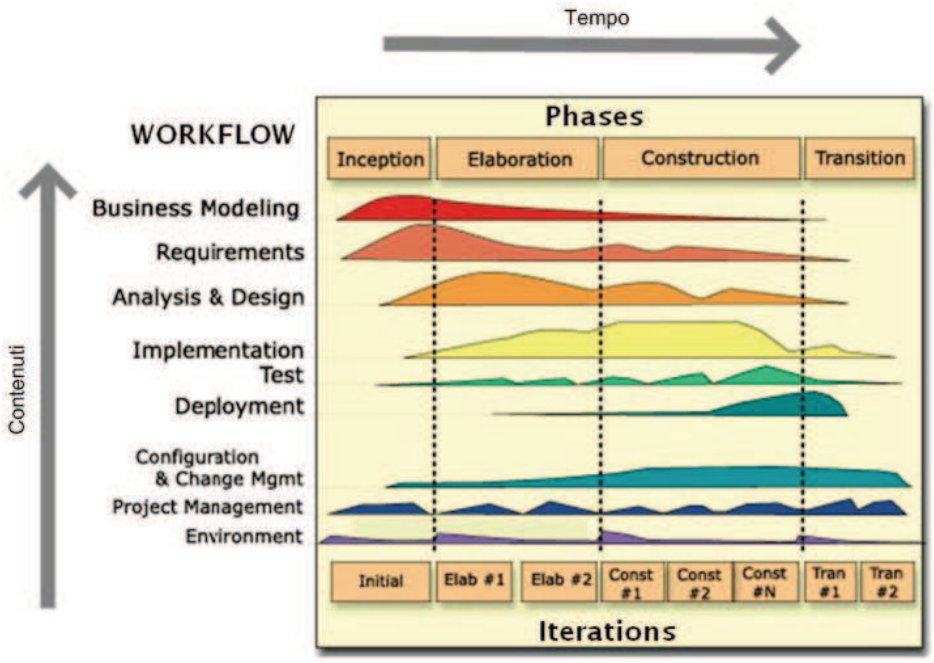

Figura 2.19: Metodologia Unified Process (UP)

Lo sviluppo avviene dunque tramite fasi successive, ognuna delle quali composta da un insieme di iterazioni. Ci si è resi conto che il processo di produzione

software è più complesso rispetto ad un ordinario processo produttivo, in quanto può essere necessario, finita una fase, tornare indietro ad una fase già terminata, apportare delle modifiche ed infine rivalidare il tutto. Diversamente dal modello a cascata, nell'UP l'integrazione non è un grosso processo affrontato unendo componenti monolitici, ma avviene in modo progressivo, quasi continuo.

Il modello UP è strutturato in due dimensioni:

1. una dimensione temporale, suddivisa in fasi, che termina con la vita e la maturità del progetto (sequenza di iterazioni citate in precedenza);
2. una dimensione di *workflow* del processo, articolata in attività ripetute in ogni iterazione di ciascuna fase (struttura).

Le fasi del modello UP sono le seguenti (indicate nella figura 2.19):

- **INCEPTION:** le diverse iterazioni che si susseguono in questa fase definiscono l'ambito e gli obiettivi del progetto. In particolare sono identificate le entità e le interazioni attraverso le quali le entità stesse interagiscono con il sistema. Al termine della fase, i risultati prodotto dalle iterazioni saranno: il *documento di vision* (composto dai primi requisiti del progetto, le caratteristiche chiave e i vincoli principali), un primo modello dei casi d'uso e il documento di *business case* (prima analisi dei rischi, prima visione del piano di progetto e una previsione dei costi per la realizzazione della proposta).

  Alla fine di questa fase si ha un'idea macroscopica del sistema, di ciò che si vuole realizzare e dei costi. L'idea macroscopica dei costi può venire posta come tetto massimo per la realizzazione; in questo caso bisogna analizzare con cura i costi, al fine di evitare discrepanze durante la realizzazione del progetto.

- **ELABORATION**: in questa fase si stabilizzano i requisiti e di conseguenza la descrizione dei confini del problema; questa fase deve terminare con la risposta a due domande: che cosa si costruisce e come viene costruito. Per rispondere a queste richieste viene fatta una analisi approfondita dei requisiti, condotta dagli analisti dei requisiti: persone che ascoltano il committente, ne analizzano la richiesta e stendono una relazione concordata col committente stesso. Per fare tutto ciò ci viene in aiuto uno strumento: lo *Use Case Diagram*. Inoltre, si definisce l'architettura di base del sistema e si minimizzano i rischi derivanti dai requisisti, per essere sicuri di ottenere alla fine dello sviluppo il prodotto richiesto. I risultati di questa fase sono un modello di use case completo, un'analisi dei requisiti supplementari (non funzionali), una descrizione dell'architettura del software, un eventuale prototipo funzionante che rappresenti l'architettura, una lista rivisitata dei rischi ed un piano di sviluppo.

- **CONSTRUCTION**: questa fase è la più consistente e quella che prevede un maggior numero di iterazioni: comprende, infatti, la realizzazione del prodotto software e l'eventuale evoluzione dell'architettura fino al rilascio

finale. La prima parte della fase di Construction è lo studio del *Release Plan*, o piano di rilascio, ove si stabilisce la ripartizione dei compiti tra i vari elementi del gruppo di sviluppo, la divisione in fasi del progetto, la formazione delle squadre, la previsione dei tempi e così via. Come unità si utilizzano i casi d'uso, in particolare: numero casi d'uso, tempo (giorni uomo) per caso d'uso. Questi elementi forniscono un'indicazione di quanto tempo sarà necessario per realizzare il progetto.

Alla fine di questa fase una prima deve essere disponibile una prima *release* del prodotto. Come risultati di questa fase avremo, inoltre un adeguato manuale ed una descrizione delle caratteristiche offerte dalla release corrente.

- **TRANSITION**: è l'ultima fase dell'UP in cui si svolgono le ultime operazioni di completamento del prodotto. In particolare, le iterazioni eseguite durante la fase di *Transition* prevedono la correzione dei difetti, le ottimizzazioni al sistema, l'avviamento e collaudo (eseguito con i veri utenti ed il reale carico di lavoro). Si conclude con la consegna del prodotto e la formazione degli utenti che dovranno interagire con il prodotto stesso.

Le attività individuate ad ogni iterazione sono le seguenti:

- *Business modeling*: comprensione dell'organizzazione del progetto e definizione dei requisiti di sistema ad alto livello.
- *Requirements*: analisi, definizione e formalizzazione dei requisiti in modo strutturato, permettendo di metterli in chiara relazione con i sistemi che li implementano e permettendo la verifica della corrispondenza e la tracciabilità delle variazioni funzionali.
- *Analysis & Design*: trasformazione dei requisiti di business in un disegno del sistema, individuando le opzioni più adatte ad implementare i requisiti, tenendo in considerazione le tematiche di riutilizzo e le performance.
- *Implementation*: implementazione dei componenti software definiti dal design, integrando i prodotti dei singoli sviluppatori in un sistema eseguibile e installabile.
- *Test*: la disciplina si propone di verificare la Qualità di diverse tipologie di oggetti (prototipo, modulo, codice integrato) in varie fasi o livelli operativi.
- *Deployment*: rilascio e installazione presso il Cliente della soluzione software sviluppata.
- *Configuration and Change Management, Project Management ed Environment*: sono workflow di supporto all'intero progetto.
    - *Configuration & Change Management:* permette di mantenere l'integrità dei manufatti del progetto, gestendo in modo controllato le richieste di modifiche e preservandone la tracciabilità nel tempo.
    - *Project Management:* la disciplina si focalizza sugli aspetti di gestione dell'intero progetto attraverso la gestione del rischio, la pianificazione

di un progetto di sviluppo iterativo attraverso il suo ciclo di vita, e il monitoraggio dello stato di avanzamento.

- *Environment*: la disciplina si propone di eseguire tutte le attività di set-up e configurazione dell'infrastruttura di un progetto applicativo e di supportare la manutenzione di un ambiente idoneo ad operare efficientemente.

Nel processo UP non esiste un'unica figura del responsabile della Qualità in quanto ogni singola persona coinvolta nel processo è responsabile della Qualità del proprio operato, sia Qualità di processo che di prodotto.

QUALITA' DEL SOFTWARE IN UP

UP è un processo di sviluppo del software progettato per i grandi team di sviluppo e fa una distinzione tra la Qualità del prodotto e la Qualità del processo. Qualità del processo è il livello con cui un processo che è stato progettato ha aderito, in fase di realizzazione, alle aspettative emerse durante lo studio. Invece la Qualità del prodotto è una misura del livello qualitativo di ciò che viene realizzato (software o sistemi software) e di tutti gli elementi contenuti in esso (componenti, sotto-sistemi architettura e così via). UP principalmente si preoccupa di verificare che un qualsiasi prodotto raggiunga il valore di Qualità atteso. A questo proposito, UP fornisce un test workflow che assicura un fissato livello di Qualità. Questo workflow, di solito, va affiancato al workflow di implementazione. A livello generale, UP garantisce la Qualità del software proponendo uno sviluppo iterativo per ogni fase distinta del processo di sviluppo del software stesso.

2.4.1.1 Modellare il software in modo visuale con UML

I modelli sono una semplificazione della realtà, aiutano a capire i problemi e ad individuarne le soluzioni. Una grossa parte del UP è dedicata allo sviluppo e alla manutenzione di modelli che rappresentano il sistema che si vuole sviluppare.

Lo ***Unified Modeling Language* (*UML*)** è un formalismo grafico per la rappresentazione, la specifica, la costruzione e la documentazione dei sistemi basati sul software, utilizzando i concetti ed i principi del paradigma *object-oriented*. Oltre a fornire un approccio sistematico per la raccolta, organizzazione e comunicazione dei requisiti, fornisce un'interpretazione standard delle parti fondamentali del sistema sino a scendere a descrizione dettagliate (schemi di database e componenti software).

UML fornisce un vocabolario per esprimere vari tipi di modelli, senza porre vincoli e indicazioni su come sviluppare il software. Differentemente UP è una guida all'uso della modellazione con UML nei progetti software: descrive i modelli di cui necessita e spiega le motivazioni per cui sono utili. Sebbene UML sia nato con il processo UP, essendo un linguaggio e non una metodologia, può essere utilizzato da team che seguono approcci diversi, come Extreme Programming (XP) (vedi paragrafo 2.4.2)

La prima versione di UML è stata sviluppata nel 1994 con lo scopo di unificare tra loro diversi formalismi per la modellazione. Prima del 1994 vi erano

infatti differenti proposte di linguaggio di modellazione basate sul paradigma object-oriented, ma erano tra loro incongruenti. UML è continuamente aggiornato e arricchito: differente da UP è uno standard OMG (Object Management Group) dal 1997 e la versione più recente è la 2.0 del 2005 [27].

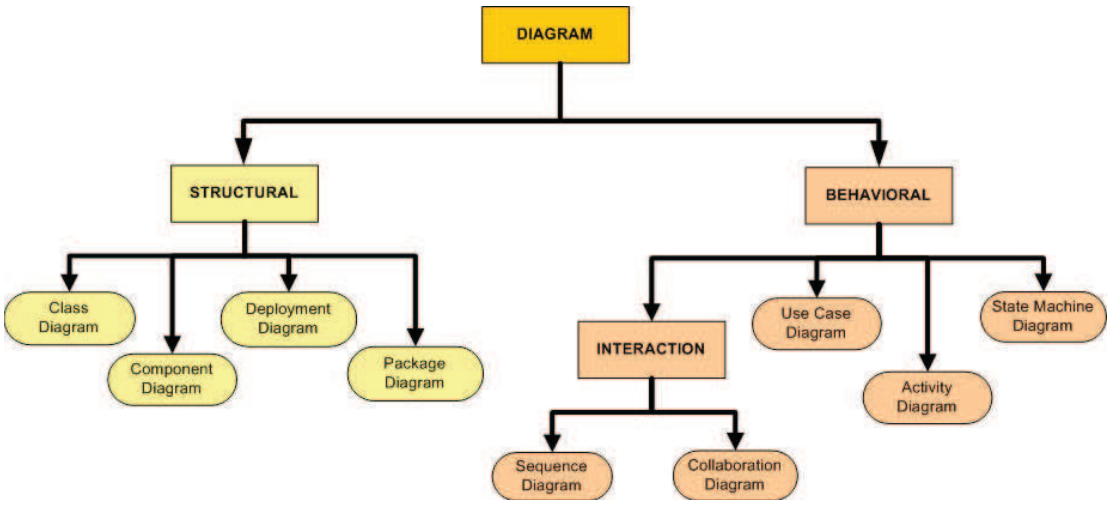

Figura 2.20: Classificazione dei diagrammi UML

Nel seguito verranno classificati i diagrammi UML principali. Per una trattazione più estesa, oltre al sito dell'OMG, si consigliano i testi di: ARLOW e NEUDSTATD [28], che contiene una descrizione di UP e della relazione con UML, di DAMIANI e MANDRAVIO [29] per una introduzione sintetica ma completa alla progettazione del software utilizzando UML.

UML definisce diversi tipologie di diagrammi (figura 2.20), che si dividono in:

- **Structural**: illustrano la staticità del sistema, gli elementi strutturali che compongono un sistema o una funzione.
- **Behavioral**: i diagrammi "comportamentali" (o di funzionamento dinamici) illustrano le caratteristiche comportamentali di un sistema o di un processo.

BEHAVIORAL

- **Use Case Diagram:** mostrano delle associazioni tra un insieme di entità esterne, dette *attori*, e delle funzionalità, dette *use case*, che il sistema dovrà implementare. L'interazione tra gli attori e i casi d'uso viene espressa per mezzo di una sequenza di messaggi. L'obiettivo dei diagrammi dei casi d'uso è definire un comportamento coerente senza rivelare la struttura interna del sistema.

- **Activity Diagram**: sono usati per modellare i comportamenti del sistema e il modo in cui tali comportamenti sono correlati in un flusso generale. Descrive il flusso di azioni necessarie per eseguire una particolare procedura. Una funzionalità complessa non è costituita da una semplice successione di azioni in sequenza e per questo motivo sono gestiti anche i comportamenti condizionali e paralleli. Un diagramma di attività può riferirsi indistintamente a una classe, un caso d'uso, un package. Lo scopo di questo diagramma è quello di concentrarsi sui flussi guidati dai processi interni (intesi come quelli opposti agli eventi esterni causati da altri oggetti), dunque tali diagrammi vanno usati nei casi in cui molti eventi rappresentano il completamento di azioni causate internamente, invece nel caso di eventi asincroni conviene utilizzare i diagrammi a stati.
- **State Machine Diagram:** rappresentano le caratteristiche di una qualche entità attraverso la descrizione degli stati in cui può venirsi a trovare e dei relativi passaggi di stato. Uno stato descrive una possibile situazione/configurazione in cui si trova l'ambito della descrizione o un suo elemento. Gli stati sono collegati tra di loro attraverso delle transazioni (attività eseguita al verificarsi di un particolare evento) che cambiano lo stato corrente.
- **Sequence Diagram**: all'interno della famiglia dei diagrammi di interazione (modelli che descrivono come dei gruppi di oggetti possono collaborare nello svolgimento di un'attività) è definito il diagramma di sequenza. I diagrammi di sequenza focalizzano l'attenzione sull'ordine temporale dello scambio di messaggi e vengono maggiormente utilizzati per descrivere scenari di successo (esempi di casi d'uso in cui tutto funziona in modo corretto). In tale contesto il livello di astrazione risulta essere notevolmente elevato: ciò non significa che si ignorano i dettagli del funzionamento del processo, bensì sono omessi dettagli implementativi. Il sequence diagram può essere utilizzato anche in fase di progettazione, mostrando effettivamente come gli oggetti collaborano tra loro per realizzare un determinato servizio (per esempio i metodi di una classe da invocare).
- **Collaboration Diagram**: sono sostanzialmente analoghi ai diagrammi di sequenza; infatti alcuni strumenti CASE consentono persino la conversione tra i due diagrammi. Questi due diagrammi, seppur semanticamente uguali, esprimono le informazioni con una vista differente. Mentre il diagramma di sequenza si concentra sulla dimensione temporale, nei diagrammi di collaborazione l'attenzione è posta sulla organizzazione degli oggetti che si scambiano i messaggi. Per conservare la dimensione temporale ogni associazione tra oggetti viene numerata.

STRUCTURAL

- **Class Diagram:** illustrano una raccolta di elementi che forniscono una formalizzazione dichiarativa del sistema. Descrivono la struttura logica del sistema in termini di classi, tipi, contenuti e relazioni. Il diagramma è una

evoluzione dei diagrammi entità-relazione dei database relazionali, ma differentemente da quest'ultimi all'interno di un oggetto è definito anche il suo comportamento.

- **Component Diagram**: descrive un'entità complessa scomponendola in parti distinte in modo da evidenziarne strutture e legami. Da un punto di vista formale, UML prevede che i componenti siano un sottotipo di classe e quindi possono essere istanziati.
- **Deployment Diagram:** illustrano l'architettura di esecuzione del sistema. Descrivono il sistema come una rete comunicante di nodi, i quali rappresentano device hardware oppure ambienti software di esecuzione.
- **Package Diagram:** permette di raggruppare insieme più entità mostrandone le mutue relazioni. Grazie al meccanismo di raggruppamento, è possibile operare ad un livello di astrazione superiore rispetto ai digrammi in cui si considera la singola entità. Un package contiene un numero arbitrario di elementi UML e permette di trattarli come un'entità unica.

UML include tre meccanismi che consentono l'estensione dei diagrammi predefiniti da parte dell'utente mediante: *stereotipi*, *tagged values* e *profili*. Questi strumenti possono essere usati nel contesto di un modello per esprimere concetti altrimenti non rappresentabili (o in modo poco chiaro) attraverso lo standard UML.

Lo stereotipo è un elemento di UML che viene qualificato in modo particolare al fine di evidenziarne proprietà o caratteristiche peculiari, addizionando dei **vincoli** all'esecuzione di un caso d'uso base mediante l'introduzione di *extension point*, oppure specificando **nuove proprietà** attraverso i *tagged values.* Sono a tutti gli effetti delle specificazioni, degli elementi costituiti dalla coppia attributo-valore; qualora sia necessario definire dei tagged value, è buona norma definire uno stereotipo che li raggruppi.

I profili sono collezioni di stereotipi, tagged values e constraints che specializzano il linguaggio per particolari domini applicativi o per l'uso di UML in congiunzione con particolari tecnologie. Fra i profili riconosciuti ufficialmente da OMG si trovano profili per CORBA, per l'*integrazione di applicazioni (EAI)* e *UML-SPT (Schedulability, Performance and Time)*, un primo tentativo di estendere UML con concetti temporali e di concorrenza e di esprimere requisiti e proprietà per analisi della schedulability e delle performance. *WAE (Web Application Extension)* è un recente lavoro di J.CONALLEN [30], non standard OMG ma riconosciuto dalla *Rational Software Corporation*, che estende la notazione UML con semantica e vincoli che permettono agli sviluppatori di modellare sistemi architetturali specifici del Web utilizzando UP o una metodologia alternativa.

2.4.2 Metodi agili e Extreme Programming

La criticità dei metodi sequenziali ha indotto la nascita di **modelli flessibili**, in cui vengono considerati approcci iterativi e incrementali che adattano lo sviluppo

ai requisiti man mano che questi vengono esplicitati. Rientrano in questi modelli: il ciclo di vita incrementale, la prototipazione, il modello a spirale e la metodologia di sviluppo UP.

Questi cicli di vita, sebbene meno rigidi, presentano delle caratteristiche comuni:

- prevedono una serie di deliverables intermedi per descrivere i risultati ottenuti durante il processo;
- sono presenti e acquistano un'importanza di rilievo le attività di pianificazione e gestione del progetto che affiancano quelle orientate alla realizzazione del software.

Queste caratteristiche sottolineano come un progetto software non si possa ridurre alla sola implementazione di codice. A volte, tuttavia, queste fasi assumono un peso ritenuto fin troppo rilevante. Per questo motivo negli ultimi anni sono stati definiti nuovi cicli di vita, complessivamente identificati con l'espressione *Agile Software Developments*.

L'approccio dei metodi agili punta l'attenzione verso la capacità di rispondere in modo veloce ed efficace ai cambiamenti che caratterizzano il processo di sviluppo software. Nel 2001 è stato pubblicato un manifesto dell'Agile Software Development che ne descrive i principi fondanti e i valori. Il termine *agile*, fa riferimento alla capacità di reazione al cambiamento, come ad esempio la capacità di adattare la soluzione software in base ai cambiamenti di requisiti richiesti dall'utente e alla flessibilità del lavoro del team durante l'intero progetto. I principi dei metodi agili, indicati nel manifesto, sono i seguenti:

- lo sviluppo agile si focalizza sulle persone e sull'interazione piuttosto che sul processo e i tool. Le metodologie agili puntano la propria attenzione sulle competenze delle singole persone che compongono il team e non tanto sul ruolo che queste assumono all'interno del gruppo.
- L'enfasi è sulla realizzazione del software piuttosto che sulla redazione di una documentazione esaustiva. La misura del raggiungimento del risultato è data dall'esecuzione del software e non dalla documentazione prodotta.
- L'aspetto essenziale è la collaborazione con l'utente e non la negoziazione del contratto. Questo aspetto mette in evidenza l'importanza di una costante comunicazione tra team e Cliente durante lo sviluppo del sofware.
- È essenziale rispondere al cambiamento piuttosto che seguire in modo rigido un piano di progetto prestabilito. Il team deve seguire con la dovuta flessibilità i cambiamenti dettati dall'evoluzione del progetto o dalla variazione dei requisiti.

Operativamente, quindi, le metodologie agili ridimensionano l'importanza e lo sforzo impiegato nelle attività di analisi, progettazione e stesura della documentazione, per porre maggiore enfasi alla fase di codifica. Questo non significa che queste attività non vengano svolte, ma che il tempo e lo sforzo dedicati sono minori rispetto ai modelli di processo presentati precedentemente.

Un esempio di metodologia agile è l'**Extreme Programming** (XP), formulata e proposta da Kent Beck nel 1999 [31], con la promessa di mantenere la controllabilità del processo pur riducendo il lavoro di supporto e convogliando il massimo dello sforzo sulla mera produzione dell'applicazione. Il metodo è caratterizzato da quattro valori chiave: *semplicità, feedback, comunicazione e coraggio*. Questo approccio, infatti, si focalizza su una progettazione della soluzione il più semplice possibile, comprendente le sole funzionalità che di volta in volta il team deve rilasciare al committente e non l'intera progettazione del sistema.

L'ideazione e la pianificazione avvengono attraverso una *user story*, una descrizione semplice e breve di una particolare funzionalità richiesta dall'utente. Il team di sviluppo analizza tutte le use story e decide in che ordine implementarle, rispettando le priorità richieste dal Cliente. Il feedback è garantito dal testing continuo eseguito sul codice e dai continui e frequenti rilasci all'utente; il team ricevendo frequenti feedback dall'utente, può adattare i rilasci successivi in modo flessibile anche in una fase avanzata di progetto.

In un processo agile la comunicazione assume un ruolo fondamentale: non solo quella continua e diretta con l'utente, ma anche la comunicazione tra i partecipanti al team di sviluppo. Infatti secondo una delle pratiche richieste dalla metodologia, la pair programming, due programmatori devono lavorare sulla stessa macchina in modo da realizzare una sorta di controllo e testing durante l'implementazione stessa del codice, in quanto due persone che operano sulla stessa porzione di codice possono essere molto più produttive di due che lavorano in modo indipendente.

Si compone di 12 pratiche fondamentali:

- **Planning game** - identificazione delle "storie" da realizzare nelle successive iterazioni; definizione delle priorità da parte del Cliente. L'attività di planning viene effettuata prima dell'inizio di ogni iterazione, e ne stabilisce gli obiettivi.
- **Small releases** - iterazioni brevi (15-60 gg.), ciascuna delle quali ha il compito di realizzare un insieme di storie concordato con il Cliente
- **Metaphor** - definizione e condivisione di una "metafora" architetturale, che guida la strutturazione del sistema.
- **Simple design** - attitudine a scegliere costantemente la soluzione "più semplice" ad un dato problema, senza porsi il problema di anticipare i futuri cambiamenti.
- **Testing** - preparazione (progettazione e definizione) dei test, effettuata in modo sistematico prima della codifica. Utilizzo sistematico di tecniche di automazione dei test, per verifiche di non regressione.
- **Refactoring** - ri-strutturazione del codice esistente, per migliorarlo; seguita dalla verifica di non regressione (esecuzione dei test precedenti).
- **Pair programming** - programmazione in coppia .

- **Collective code ownership** - proprietà collettiva del codice da parte dell'intero gruppo di lavoro: chiunque può modificare e migliorare una qualsiasi parte del sistema .
- **Continuous Integration** - le modifiche effettuate sul sistema vengono sottoposte (almeno giornalmente, ma tipicamente con più frequenza) a test di integrazione.
- **40-hour week** - divieto di lavoro straordinario (che nel medio termine diminuisce la produttività del gruppo di lavoro).
- **On-site customer** - il Cliente (o un suo rappresentante) deve essere sempre disponibile per chiarimenti sulle storie da realizzare.
- **Coding standards** - gli standard di codifica vengono rispettati in modo ferreo.

XP si propone come soluzione ottimale da adottare in un terzo scenario: quando i requisiti non sono noti a priori e non sono presenti le risorse per avviare un processo a spirale. Implementare l'XP significa integrare le 12 pratiche di cui si compone, o un suo sottoinsieme, nel processo di sviluppo aziendale.

VANTAGGI

Tra le caratteristiche più interessanti dell'XP, ritengo utile annoverare le seguenti:

- in ogni momento la squadra di sviluppo dispone di un software funzionante anche se non completo di tutte le sue funzionalità;
- la squadra di sviluppo collabora attivamente con il committente, al fine di rilevare i feedback sul prodotto già durante il processo di sviluppo e non quando il prodotto è ultimato;
- è il collaudo a guidare lo sviluppo e non il contrario: in tal modo si ha una costante consapevolezza di cosa funziona correttamente.
- Fornisce risultati migliori, in termini di Qualità del prodotto, rispetto a molti progetti condotti con altri approcci, grazie alle pratiche sistematiche e quotidiane di testing e di continuous integration.
- Porta ad una assunzione di responsabilità collettiva da parte del gruppo di lavoro, e favorisce la creazione di team coesi.

Se XP, viene eseguita nella maniera corretta, produce notevoli vantaggi per chi produce software: ottenere, fin dalle prime fasi del processo di sviluppo, un software funzionante, ridurre la densità dei difetti nel codice, maggiore coinvolgimento del committente nello sviluppo, dominare il cambiamento riducendo il rischio, e in ultimo, fattore importante ma difficilmente misurabile, aumentare la soddisfazione degli sviluppatori nel loro lavoro.

SVANTAGGI

XP è inadatto a contesti progettuali in cui sia richiesta una tracciatura sistematica dei requisiti. L'adozione di XP (dell'insieme completo delle pratiche XP) richiede trasformazioni profonde degli assetti organizzativi e delle politiche del personale, in quanto modifica in modo sostanziale ruoli consolidati a livello organizzativo. Può anche richiedere una trasformazione degli aspetti logistici, in quanto richiede che il gruppo di lavoro (tipicamente costituito da circa una decina di persone) operi in un singolo locale. È molto più semplice adottare XP per una software house (in particolare, per una software house appena nata) che non per il settore sistemi di una grande organizzazione.

XP richiede che le persone siano capaci di lavorare in gruppo: non tutti sono in grado di farlo. Richiede inoltre l'utilizzo di strumenti di sviluppo evoluti, per programmare, testare, ristrutturare il codice in modo estremamente produttivo.

QUALITA' DEL SOFTWARE IN XP

Con la Qualità avviene un fenomeno strano: spesso, aumentare la Qualità significa completare il progetto in meno tempo. Il fatto è che appena il gruppo di progetto si abitua a fare test intensivi (pietra miliare di XP) e gli standard di codifica vengono seguiti, gradualmente il progetto inizierà a progredire più velocemente di quanto facesse prima. La Qualità del progetto verrà assicurata - grazie ai test – e questo a sua volta infonderà maggiore confidenza nel codice e, quindi, maggiore facilità nel far fronte al cambiamento permettendo alle persone di programmare più velocemente .

L'altra faccia della medaglia è la tentazione di sacrificare la Qualità interna del progetto – quella percepita dai programmatori – per ridurre il tempo di rilascio del progetto, ritenendo che la Qualità esterna, quella percepita dai clienti – non sarà eccessivamente deteriorata. Tuttavia, questa è una scommessa a brevissimo termine, che tende ad essere un invito al disastro, perché ignora il fatto basilare che ognuno lavora meglio se gli è permesso di fare un lavoro di Qualità. Ignorare questo renderà il gruppo demoralizzato e, nel lungo termine, il progetto rallenterà, e verrà perso molto più tempo di quello che si sperava di guadagnare tagliando sulla Qualità.

2.4.3 Design Patterns

I design pattern, sono stati introdotti, per quel che concerne la progettazione di sistemi informatici, con la pubblicazione nel 1995 del testo: "**Design Patterns: Elements of Reusable Object-Oriented Software"** (Design Patterns: elementi per il riuso di software ad oggetti).

I quattro ricercatori (GAMMA, HELM, JOHNSON E VLISSIDES), pionieri di questa metodologia e autori del testo appena citato, descrivono soluzioni semplici, generiche ed eleganti per problemi specifici nel contesto della progettazione object-oriented.

L'uso di questa metodologia permette di raggiungere diversi obiettivi, tutti ugualmente interessanti, tra i quali l'impiego di soluzioni e modelli consolidati, il riuso del codice e la scrittura di codice più generale, elegante e riusabile.

"Progettare software ad oggetti è difficile; riuscire a renderlo riusabile è ancora più difficile. Occorre trovare gli oggetti giusti, fattorizzarli in classi con giusta granularità, definire interfacce e gerarchie d'ereditarietà e stabilire fra queste le relazioni fondamentali. La progettazione dovrebbe essere specifica per il problema che si sta affrontando, ma anche sufficientemente generica per problemi e requisiti futuri". [32]

Comunque il termine fu inizialmente introdotto in architettura in un celebre saggio di Christopher ALEXANDER; in seguito, proprio l'opera di ALEXANDER ispirò la nascita di un settore dell'ingegneria del software dedicato all'applicazione del concetto di design pattern alle architetture software. Il tema dei pattern viene oggi considerato una delle linee principali di sviluppo dell'ingegneria del software object-oriented. Esso trova applicazioni in tutta una serie di contesti di grande interesse per l'industria del software, e la maggior parte dei linguaggi di programmazione moderni, e di tecnologie correlate, sono stati progettati (o modificati) tenendo conto anche dell'obiettivo di essere coerenti con questo approccio emergente allo sviluppo del software.

Un design pattern può essere definito *"una soluzione progettuale generale a un problema ricorrente"*. Esso non è una libreria o un componente di software riusabile, quanto una descrizione o un modello da applicare per risolvere un problema che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software. La differenza tra un algoritmo e un design pattern è che il primo risolve problemi computazionali, mentre il secondo è legato agli aspetti progettuali del software.

Generalmente un pattern consta di quattro elementi essenziali:

- **nome**, costituito da una o due parole che siano il più possibile rappresentative del pattern stesso;
- **problema**, ovvero la descrizione della situazione alla quale si può applicare il pattern. Può comprendere la descrizione di classi o di problemi di progettazione specifici, come anche una lista di condizioni perché sia necessario l'utilizzo del pattern;
- **soluzione**, che descrive gli elementi costitutivi del progetto con le relazioni e relative implicazioni, senza però addentrarsi in una specifica implementazione. Il concetto è di presentare un problema astratto e la relativa configurazione di elementi adatta a risolverlo;
- **conseguenze**, sono i risultati e i vincoli che si ottengono applicando il pattern. Sebbene le conseguenze siano spesso non rivelate quando descriviamo delle decisioni di progetto, queste diventano critiche per

valutare soluzioni alternative, e per capire e stimare i costi e i benefici dell'applicazione di un pattern.

Nessun pattern può risolvere da solo tutti i problemi in un contesto: solo una combinazione di pattern può riuscirci, così come non basta una parola per parlare: ci vuole un linguaggio. Perciò una combinazione di pattern, che collaborano per risolvere i problemi di un contesto, è chiamata linguaggio di pattern (*Pattern Language*).

Come esempio, viene presentato il pattern architetturale **model-view-controller (MVC)** particolarmente noto e diffuso. Esso si occupa di descrivere la tipica struttura di un programma che permette ad un utente di modificare in modo interattivo un insieme complesso di informazioni, visualizzato secondo diverse modalità. È il caso di tutti gli strumenti interattivi come wordprocessor, spreadsheet, grafica, ecc. In tutti questi casi, lo strumento abilita la creazione e/o modifica di una struttura informativa complessa, memorizzata secondo formati non direttamente manipolabili dall'utente. La struttura informativa viene resa visibile e/o manipolabile dall'utente secondo diverse viste (*view*).

MVC è formato da tre tipologie di oggetti (figura 2.21):

- **Model:** è il componente che contiene tutti i dati gestiti dall'applicazione, incapsula una logica di business.
- **View:** si occupa della rappresentazione dei dati verso l'utente. Poiché ci potrebbero essere diverse viste, in una applicazione può esistere più di una componente di questo tipo.
- **Controller:** è il componente che definisce le reazioni dell'interfaccia grafica all'input dell'utente.

Il Model è il "cuore" dell'applicazione: definisce i dati e le operazioni che possono essere eseguite su essi. Fornisce delle funzioni per l'accesso e l'aggiornamento. Può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View stesse di presentare agli occhi degli utenti dati sempre aggiornati. View è l'interfaccia grafica con cui l'utente interagisce. Il controller ha la responsabilità di trasformare le interazioni dell'utente in azioni eseguite dal Model. Ma il Controller non rappresenta un semplice "ponte" tra View e Model. Realizzando la mappatura tra input dell'utente e processi eseguiti dal Model e selezionando la schermate della View richieste, il Controller implementa la logica di controllo dell'applicazione.

Le view possono essere aggiornate in modo **passivo**, quando l'utente richiede di aggiornare la vista mediante una *Query* oppure in modalità **attiva**, se è il modello che invia una richiesta alla view ogniqualvolta vi è un cambiamento nei dati del modello stesso (*Notify View*). Le operazioni di modifica dei dati sono generate da un evento invocato dall'utente attraverso il controller che ha il compito da un lato di

aggiornare il modello (*Change Model*) e dall'altro di provocare l'aggiornamento dirette delle viste relative (*Update View*).

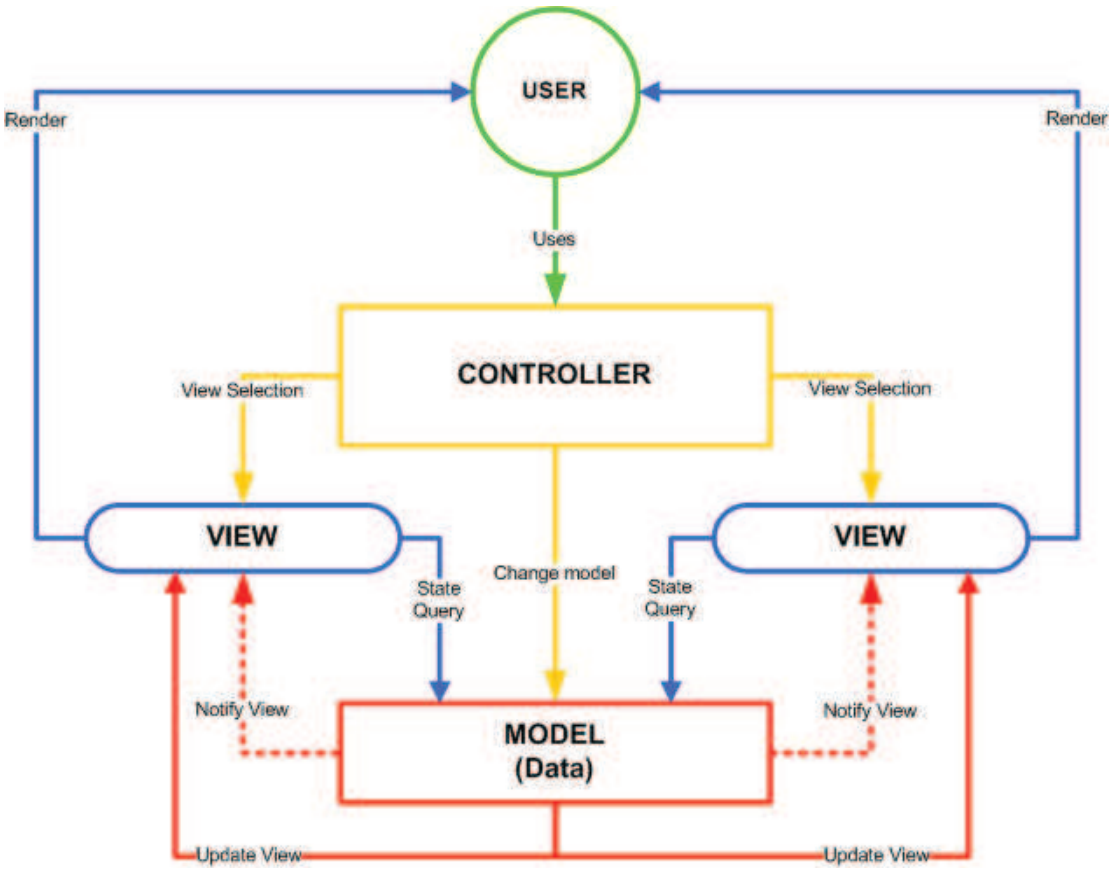

Figura 2.21: Pattern Model-View-Controller

Originariamente impiegato dal linguaggio Smalltalk, il pattern è stato esplicitamente o implicitamente sposato da numerose tecnologie moderne, come framework basati su Java, su Objective C o su .NET. A causa della crescente diffusione di tecnologie basate su MVC nel contesto di framework o piattaforme middleware per applicazioni Web, l'espressione framework MVC o sistema MVC sta entrando nell'uso anche per indicare specificatamente questa categoria di sistemi (che comprende per esempio Struts e Tapestry). Si osservi che quanto presentato è un esempio piuttosto semplice di pattern model-view-controller al fine di comprendere il significato del termine pattern.

I design pattern facilitano il riuso efficace di progetti e architetture, rendendo tecniche ormai consolidate più accessibili agli sviluppatori di nuovi sistemi. I design pattern aiutano a scegliere tra le alternative di progettazione che rendono un sistema riusabile e a scartare quelle che comprometterebbero la riusabilità. Inoltre, possono anche migliorare la documentazione e la manutenzione dei sistemi esistenti,

descrivendo esplicitamente le classi, le collaborazioni fra oggetti e le loro finalità. Per concludere, i design pattern aiutano ad arrivare ad un buon progetto più velocemente.

2.4.4 Contratti e Scenari nello sviluppo del software

La Qualità del software può essere interpretata, secondo la metodologia suggerita da Plösch [33], come elemento di accordo tra le esigenze del Cliente e le aspettative economiche del Fornitore. In generale, diventa molto importante fornire al Cliente (sia che acquisiti o che si impegni in un progetto software), concetti e strumenti che rendano disponibili metriche di valutazione dirette sui servizi attesi e indirette sui costi associati (valutazione costi / benefici).

L'approccio di Plösch combina il concetto di contratti e scenari in una consistente metodologia che può essere utilizzata, nello sviluppo di una applicazione software, per fornire al Cliente i benefici attesi malgrado il diverso linguaggio utilizzato (linguaggio naturale vs. linguaggio di programmazione).

La metodologia proposta da Plösch è una estensione del principio ***Design By Contract (DBC)***. L'idea centrale del DBC è che la progettazione delle entità software avviene definendo in modo organico e coerente le specifiche funzionali che ciascuna porzione di codice si impegna a rispettare nei confronti dei propri utilizzatori. Una specifica funzionale, o "contratto", viene creato per ogni modulo del sistema prima che sia caricato; l'esecuzione del programma è quindi vista come l'interazione fra i vari moduli vincolati da questi contratti. A questo principio, Plösch unisce il concetto di scenario, una tipica interazione tra un computer-system ed i suoi client, dove quest'ultimo può essere sia una persona fisica, sia un altro computer-system.

Fattori	Criteri	Metriche
Requisiti di Qualità	Ambiguità	Frasi non ben formulate
	Completezza	**Specifica dei requisiti dettagliata**
	Comprensibilità	Struttura del documento
	Volatilità	Modifiche frequenti
	Tracciabilità	Num. Di requisiti non tracciati nel codice
Qualità del codice prodotto	Architettura	Complessità Logica
	Manutenibilità	Dimensioni/complessità
	Riusabilità	Dimensioni/complessità
	Documentazione Interna	Codice Commentato
	Documentazione Esterna	Manuale utente
Efficacia dell'implementazione	Utilizzo di risorse	Macchine utilizzate nella realizzazione
	Termini di consegna	Fine (presunta) di ogni task
Efficacia del testing	**Correttezza**	**Errori Critici**

Tabella 2.2: Il modello SATC

Come è stato evidenziato precedentemente, per essere in grado di misurare la Qualità del software, è necessario individuare un approccio sistematico al fine di

individuare le metriche delle proprietà generali del software. Oltre al modello **ISO 9126**, per la misurazione della Qualità del prodotto sono stati proposti ulteriori modelli basati sull'approccio **FCM (Factor – Criteria - Metrics)**, tra cui il modello **SATC (Software Assurance Technology Center)** [34] illustrato nella tabella 2.2.

Gli attributi della Qualità (fattori) vengono identificati all'interno di un software. Per rendere ogni fattore misurabile è necessario raffinare i fattori utilizzando i "criteri di Qualità". I fattori rappresentano una visione più orientata all'utente, mentre i criteri di Qualità rispondono più ad una visione del software. Il processo di raffinamento continua finché gli "indicatori di Qualità", che sono misurabili e ben determinati nelle metriche, vengono individuati per ogni criterio di Qualità.

Sebbene tutti i criteri sopra citati siano importanti per la Qualità di un prodotto software, in questo approccio assumono per il Cliente un rilevante significato gli attributi di :

- **Completezza:** un documento di specifica dei requisiti `e completo se contiene tutti i requisiti, specificati con una adeguata precisione per ognuno di essi in modo da agevolare le fasi di design ed implementazione.
- **Correttezza:** un prodotto `e corretto se risponde in maniera soddisfacente a tutti i requisiti emersi nel documento che ne raccoglie la specifica.

Il passaggio da problema a soluzione vede come elemento di congiunzione il concetto di descrizione dell'interfaccia, cioè la specifica. Essa rappresenta il contratto tra chi ha un problema da risolvere e lo sviluppatore che ne progetta e implementa una soluzione. Un **contratto** è solitamente un documento che esplicita obblighi e benefici per i contraenti. Nella vita di tutti i giorni, i contratti sono sottoscritti da due parti. Dalla stipula di un contratto, ogni parte si aspetta dei benefici ed accetta alcuni obblighi a cui sottostare. Solitamente ciò che per una parte costituisce un obbligo, per l'altra parte costituisce un beneficio.

Nell'ingegneria del software, le clausole e gli impegni si traducono in:

- **Precondizioni:** espressione logica che deve essere verificata affinché una certa porzione di codice possa essere eseguita.
- **Postcondizioni:** indica la condizione che deve essere verificata al termine di un'elaborazione.

Il termine asserzione, in questo contesto, è usato come sinonimo di precondizione o postcondizione, mentre il termine contratto racchiude un insieme di asserzioni di un programma o parte di esso.

Uno **scenario** è una tipica interazione tra un computer-system ed i suoi client. Un client può essere sia una persona fisica, sia un altro sistema. Gli scenari sono utili per individuare e documentare i requisiti di progetto, facilitano il lavoro in quanto permettono di suddividere il progetto in sottoprogetti più piccoli, sono di facile comprensione e approvazione da parte del committente e utili durante la fase di testing per verificare che tutti i requisiti siano soddisfatti.

Indipendente dal tipo di metodologia scelto per lo sviluppo del software, ci sono delle operazioni (task), che devono essere eseguite durante le fasi di analisi, design, implementazione e testing. I contratti e gli scenari influiscono su queste fasi principali fornendone del valore aggiunto:

- **ANALISI:** lo scopo dei **contratti** è quello di aggiungere, tramite le asserzioni, elementi di specifica addizionali ai prodotti software e di imporre degli "obblighi" ai client (entità fisiche e non) ed agli implementatori di questi prodotti. Gli scenari hanno un ruolo molto importante nella fase analisi. Il framework che si deve seguire per utilizzare gli **scenari** nella fase di analisi è: raccolta dei requisiti del committente, sviluppo ed adattamento delle degli scenari secondo le necessità raccolte, presentazione al Cliente dei risultati ottenuti e decidere se è il caso di re-iterare il processo di analisi dal momento che non sono state recepite alcune esigenze. Uno dei diagrammi UML più utilizzato per rappresentare gli scenari è lo *use case diagram*.
- **DESIGN:** anche la fase di design, come quella di analisi, è di tipo iterativo. Partendo con la fase di analisi, essa viene trasformata con dei graduali passi in un modello di design che contiene i vari nomi delle diverse classi, le interfacce e le funzionalità. Ogni modifica che verrà effettuata nel modello sviluppato nella fase di design, comporterà una modifica anche nei contratti. I **contratti** assicurano che il design model rimanga sempre compatibile con quello pensato nella fase di analisi. Invece gli **scenari** permettono di verificare che il design model sviluppato possa essere utilizzato nel contesto di un'applicazione specifica, dal momento che il core e le varie funzionalità sono illustrate e chiaramente raccolte all'interno di essi.
- **IMPLEMENTAZIONE:** i **contratti**, in questa fase, hanno ruolo essenziale, dal momento che guidano il programmatore durante l'implementazione ed il testing delle unità. Nella fase di implementazione, le invarianti e le postcondizioni sono molto importanti, dal momento che devono essere rispettate e soddisfatte. Gli **scenari** assumono un ruolo marginale e diventano esclusivamente fonte di chiarimento, in quanto originano una serie di precisazioni per quanto riguarda i requisiti funzionali che andranno rispettati.
- **TESTING:** se nelle fase di implementazione i **contratti** vengono accuratamente introdotti nel codice, come gia visto negli esempi precedenti, con appositi tool si può verificare che tutti i requisiti funzionali, siano stati rispettati. Questo metodo assicura una correttezza nel codice sviluppato, agevola la fase di testing ed illustra al committente che effettivamente tutte le sue richieste e necessità sono state rispettate. Gli **scenari** sono la risorsa primaria per generare gli *interaction test*. Gli scenari, infatti devono essere utilizzati per costruire e formulare gli *scripting* di testing funzionale.

JASS [35], è un esempio di tool per la valutazione dei contratti per applicazioni sviluppate in linguaggio Java.