Università degli Studi di Roma La Sapienza
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Quinquennale in Informatica

INGEGNERIA DEL SOFTWARE 1

Rielaborazione del corso tenuto dal professor Bottoni

Autore: Andrea Saporito
Vers.: 1.2

Indice

Introduzione

Questa è la rielaborazione del corso di Ingegneria del Software 1 tenuto dal professor Bottoni. Ho pensato di realizzare questo lavoro per mettere insieme, in modo ordinato e scorrevole, tutte le informazioni presentate durante il corso. Alcuni punti sono stati approfonditi rispetto a quanto fatto a lezione, in quanto mi sembrava doveroso approfondire tali argomenti. Esistono argomenti che non ho approfondito o per mancanza di informazioni specifiche o perché si allontanavano troppo dagli argomenti trattati durante il corso.

Il lavoro è idealmente diviso in quattro parti: nella prima tratto la tecnologia ad oggetti, in modo da fissare una volta per tutte la terminologia che utilizzo, nella seconda introduco l'ingegneria del software, nella terza parlo delle attività ausiliarie, cioè di quelle attività che si svolgono parallelamente alle attività principali nel processo di ingegneria del software (ho ritenuto opportuno anticipare questa parte perché alcuni settori dello Unified Process che andrò ad analizzare nella parte successiva, si appoggiano sugli argomenti qui trattati), nella quarta parlo dello Unified Process cioè di un modello per l'ingegneria del software (si è scelto questo argomento perché rappresenta lo stato dell'arte nella progettazione orientata agli oggetti, si avvale del linguaggio UML ed è una sintesi di vari modelli). All'interno dello Unified Process parlerò anche delle attività di ombrello, cioè di quelle attività che iniziano nelle primissime fasi del processo e si svolgono parallelamente a quelle principali. In ultimo darò una breve introduzione del Refactoring.

PARTE PRIMA

Tecnologia ad oggetti

Cos'è un oggetto?

Un oggetto è un qualcosa che possiede uno **stato**, degli **attributi** e dei **comportamenti**; usando la similitudine con la realtà, possiamo pensare ad una tazza: essa ha uno stato (piena o vuota), degli attributi (forma e colore) ma nessun comportamento (è un oggetto passivo: viene riempito e viene vuotato, ma non lo fa automaticamente); pensando invece ad un cane abbiamo vari comportamenti: mangia, beve, abbaia, si muove, ecc.

Nei computer ovviamente un oggetto non è qualcosa di reale, ma, al più, una rappresentazione: così la tazza riprodotta sullo schermo non è una tazza reale! Proprio per questo motivo la tazza sullo schermo può avere dei comportamenti: riempi per farla riempire, svuota per farla svuotare.

Chiamo **dominio degli attributi** l'insieme degli attributi caratteristici di un oggetto.

Tutti gli oggetti hanno un'**identità**: essa rappresenta in modo univoco un oggetto; nell'esempio delle tazze, due tazze possono essere uguali (stesso stato), ma non hanno la stessa identità: se prendo una tazza in mano, l'altra rimane sul tavolo; sono dunque in grado di distinguere le due tazze pur essendo esse uguali. Ovviamente l'esempio è valido solo se nel dominio degli attributi delle tazze si trascura la posizione sul tavolo.
Per ovviare a questo inconveniente possiamo fare un esempio che non richiede di ignorare nel dominio degli attributi nulla: nel campo sub-atomico ogni particella può essere perfettamente uguale ad altre e, pur non potendo conoscere la posizione di una particella, siamo in grado di bombardarla senza toccare le altre particelle. Di conseguenza due particelle uguali sono in qualche modo distiguibili e quindi non identiche.

Nella programmazione ad oggetti gli attributi vengono anche chiamati **proprietà**, mentre i comportamenti sono anche chiamati **metodi** od **operazioni**.

Classi ed istanze

Un oggetto, normalmente, è un'**istanza** di una "cosa" e può essere una delle istanze della stessa "cosa". Ad esempio una tazza da caffè è un elemento dell'insieme delle possibili tazze. La descrizione della "cosa" è chiamata **classe**. Un oggetto è un'istanza di una classe: la classe definisce come sono fatti gli oggetti da essa derivati, quindi l'insieme dei possibili stati, attributi e comportamenti, l'oggetto stabilisce i valori posseduti dagli attributi. È importante notare che i comportamenti sono uguali tra tutti gli oggetti: l'azione riempi della tazza sarà realizzata in modo identico indipendentemente dall'istanza della tazza in quel momento.

Come collaborano gli oggetti?

Normalmente è assurdo pensare che un oggetto viva da solo senza comunicare nulla agli altri (se non interagisce non ha motivo di esistere: pensate ad un programma che vi occupi memoria, senza fare assolutamente nulla!). La comunicazione tra due oggetti avviene per mezzo di messaggi: questi messaggi attivano una o più operazioni dell'oggetto destinatario del messaggio. Ad esempio ad un oggetto Spedizione viene chiesto di spedire della merce, tale oggetto manderà dunque un messaggio all'oggetto Magazzino richiedendo di ridurre la quantità disponibile della detta merce; in altre parole Spedizione attiva un comportamento di Magazzino: riduciMerce!

Come gli oggetti si identificano tra di loro?

Immaginiamo che Spedizione abbia a che fare con due Magazzini, uno per la merce A ed uno per la merce B; come fa a sapere a quale oggetto inviare il messaggio?

Ad ogni oggetto viene assegnato un OID (Object Identifier o Identificatore dell'oggetto), questo identificatore è unico per ogni oggetto e non ce ne sono mai due uguali. Un OID è per l'esattezza un riferimento: se Spedizione ha accesso agli OID dei suoi due Magazzini, ha accesso ai due Magazzini. Quindi per poter spedire un messaggio ad un suo Magazzino, Spedizione deve conoscere il suo OID.

L'OID viene anche detto legame o, appunto, riferimento.

Questi legami possono essere persistenti o transitori. Un legame è persistente se esso è memorizzato da qualche parte (probabilmente un attributo) e tale parte viene salvata alla chiusura del programma e può essere ripristinata successivamente. Un legame è transitorio se l'OID non viene salvato.

Ma se il legame è transitorio, come si fa a conoscerlo? Esso può essere: recuperato da un altro oggetto che lo contiene in maniera persistente, essere passato con un messaggio, od essere creato sul momento. La creazione di un OID avviene quando si istanzia un oggetto di una classe (il costrutto new del Java restituisce appunto un OID).

Una volta che Spedizione conosce l'OID dei suoi magazzini, come fa ad inviargli un messaggio? Supponiamo che Spedizione abbia memorizzato, transitoriamente o persistentemente, gli OID in due attributi merceA e merceB. Vuole inviare un messaggio al Magazzino contenente la merceA, dicendogli di ridurre la disponibilità della sua merce? Userà dunque merceA.riduciMerce(). Con questa sintassi si indica di prendere l'OID memorizzato in merceA e di inviare un messaggio all'oggetto con quell'OID dicendogli di eseguire il comportamento riduciMerce.

Questo sistema è detto più brevemente: Invocare il metodo riduciMerce dell'oggetto Magazzino riferito da merceA (od ancora più brevemente Invocare il metodo riduciMerce di merceA).

Classe

Abbiamo detto che la classe è il descrittore di un insieme di oggetti e di essi specifica i possibili stati, attributi e comportamenti. Specifichiamo un po' meglio il concetto.

Possiamo considerare una classe come uno stampo per la creazione degli oggetti. La classe indica gli attributi che un oggetto deve possedere (dominio degli attributi) nonché il loro tipo; specifica altresì tutti i comportamenti dell'oggetto. Dato che uno stato normalmente è definito attraverso i valori possibili degli attributi, la classe normalmente non specifica gli stati in maniera esplicita.

Attributi

Un attributo è identificato da un nome e da un tipo e può contenere un valore; le classi definiscono i tipi, mentre gli oggetti assegnano il valore agli attributi stessi. Un tipo può essere primitivo o una classe; nel primo caso esso viene fornito dal linguaggio ad oggetti (stringhe, numeri, booleani) mentre nel secondo significa che l'attributo dovrà contenere un OID di un oggetto utente istanziato dalla classe definita come tipo (es.: merceA è di tipo Magazzino, significa che l'attributo merceA dovrà contenere un OID di un oggetto che sia un'istanza di Magazzino).

Gli attributi hanno anche una **visibilità** cioè viene definito quali oggetti possono vedere l'attributo.

Visibilità **Pubblica**: l'attributo di un oggetto può essere visto e manipolato da qualsiasi altro oggetto che abbia accesso all’oggetto che lo contiene.

Visibilità **Privata**: l'attributo di un oggetto può essere visto e manipolato solo da un oggetto che sia istanza della classe che ha dichiarato l'attributo.

Visibilità **Protetta**: l'attributo di un oggetto può essere visto e manipolato da un oggetto che sia istanza della classe che ha dichiarato l'attributo e da tutti gli oggetti istanze di classi "figlie" (vedi più avanti la Generalizzazione).

Esistono anche altri tipi di visibilità, definite dai linguaggi di programmazione, come **Package** (l'accesso è possibile anche a tutte le classi appartenenti allo stesso "insieme logico") o **Friend** (l'accesso è possibile anche alle classi definite come "amiche").

A volte si parlerà di attributi **derivati**: essi in realtà non esistono come attributi fisici, vengono invece calcolati di volta in volta, attraverso operazioni su altri attributi; ad esempio un attributo "tempo trascorso" può essere derivato: viene calcolato in base alla differenza tra la data/ora corrente ed una iniziale.

Operazioni

Le operazioni specificano gli algoritmi che manipolano i dati dell'oggetto (attributi). L'operazione può contenere una lista di parametri in ingresso (**parametri formali**), cui saranno assegnati valori specifici nella chiamata (**parametri attuali**) e può restituire un valore all'oggetto chiamante.

Il nome dell'operazione insieme alla lista dei tipi dei parametri formali è chiamata **firma** (o **signature**, in inglese) di un'operazione. La firma deve essere unica all'interno di una classe: questo significa che una classe può avere più operazioni con lo stesso nome, purché la lista dei tipi dei parametri formali sia sempre diversa.

La visibilità delle operazioni segue le stesse regole della visibilità degli attributi.

Prassi della programmazione ad oggetti è quella di rendere gli attributi non pubblici e permettere agli altri di modificare o leggere gli eventuali dati necessari, attraverso appositi metodi pubblici (in modo da poter effettuare maggiore controllo e/o aggiungere solo apposite operazioni alla lettura/variazione di un dato)

Associazione

Quando due oggetti devono comunicare tra di loro, significa che tra le due classi vi è un'associazione. L'associazione indica, dunque, il fatto che gli oggetti delle due classi devono, in qualche modo, comunicare.

L'associazione può avere un nome e le classi che ne fanno parte un *ruolo* (cioè un nome), semplicemente per ricordare che tipo di associazione hanno le due classi ed il loro ruolo. Ad esempio due classi, Ordine e Spedizione, potrebbero avere un'associazione "Spedizione dell'ordine" con i loro ruoli di ordine e di spedizione, ad indicare che le due classi devono collaborare per spedire un ordine e che, rispettivamente, Ordine si preoccuperà di fornire gli ordini a Spedizione che provvederà a spedirli.

L'associazione può avere una molteplicità, cioè ad ogni capo di un associazione si stabiliscono quanti oggetti della classe possono partecipare alla collaborazione. Normalmente le molteplicità comuni sono:

0..1	al massimo un oggetto può partecipare all'associazione
0..*	quanti oggetti si vuole (compreso zero!) possono partecipare all'associazione
1..1	uno ed un solo oggetto può partecipare all'associazione (può essere indicata solo con 1)

1..*	da uno a quanti se ne vuole possono partecipare all'associazione
*	Nessun limite al numero di oggetti che possono partecipare all'associazione

Ovviamente possono esisterne anche altre.

Così l'associazione “Spedizione di un ordine” avrà da entrambi i lati * ad indicare nessun limite sul numero di ordini che possono far parte di una spedizione e nessun limite al numero di spedizioni attraverso cui possono essere serviti gli ordini.

Le associazioni che hanno ad entrambi i capi * sono dette **molti a molti**, quelle dove una lato vi è 1 e dall'altro vi è * sono dette **uno a molti** o **molti ad uno** (a seconda da che “lato” si guarda l'associazione).

Le associazioni possono avere una **navigabilità** esplicita cioè un'indicazione che indica da che lato è possibile percorrere l'associazione; ad esempio se abbiamo un'associazione tra Utenti (che contiene i nomi degli utente) e Password (che contiene le relative password) è assai probabile che dagli utenti vogliamo risalire alle password, ma non viceversa; quindi l'associazione avrà una navigabilità da Utenti a Password. Se la navigabilità non è indicata, il più delle volte, indica una navigabilità bidirezionale.

Classe di associazione

Benché normalmente non esistano in alcun linguaggio di programmazione le classi di associazioni, esse vengono usate quando si vuole assegnare attributi e/o comportamenti ad un'associazione stessa. Ad esempio un'associazione tra Studenti e Corsi potrebbe avere una classe di associazione (Valutazione) che mantiene i voti degli studenti rispetto ai corsi seguiti.

Nella pratica tale classe sarà poi tradotta in una classe reale che avrà due associazioni: una con Studenti ed una con Corsi.

Aggregazione e Composizione

L'aggregazione e la composizione sono due tipi di particolari di associazioni. Con questi due “costrutti” si indica che una classe è “composta” da altre classi. In altre parole un oggetto che contiene tanti oggetti di una determinata classe viene detto contenitore, mentre gli oggetti che fanno parte dell'oggetto contenitore sono detti aggregati o componenti (anche se quest'ultimo termine è poco usato perché potrebbe causare ambiguità con un termine omonimo, ma dal significato completamente diverso, usato dallo Unified Process).

L'aggregazione è un costrutto che implica transitività e asimmetria: se A contiene la classe B e B contiene la classe C, allora A contiene la classe C; A non può però contenere se stessa, B non può contenere A (oltre a se stessa) e C non può contenere né A né B (oltre a se stessa).

La composizione aggiunge la dipendenza esistenziale: se A contiene B attraverso la composizione, significa che se A viene distrutto, allora anche B deve esserlo e B non può essere creato se non viene creato A.

Generalizzazione

La generalizzazione è una relazione “tipo-di” tra una classe più generica (**superclasse**) ed una più specifica della precedente (**sottoclasse**). La sottoclasse è una tipologia della superclasse. Un oggetto della sottoclasse può essere usato ovunque sia permesso l'uso di un oggetto della superclasse.

La generalizzazione permette di non ridefinire proprietà già definite. Gli attributi e le operazioni già

introdotti nella superclasse possono essere riusati nella sottoclasse (si dice che la sottoclasse **eredita** gli attributi ed i metodi della classe genitrice). Essa facilita la specifica incrementale, l'uso delle proprietà comuni tra classi ed una migliore localizzazione delle modifiche.

Anche la generalizzazione è transitiva ed asimmetrica.

Si noti che di ereditarietà si parla in relazione alle classi, non agli oggetti (si applica ai tipi non ai valori).

Polimorfismo

Un metodo ereditato da una sottoclasse è spesso usato così com'è. Tuttavia esso può essere modificato o riscritto completamente. Quando questo avviene si parla di metodo polimorfo: quale implementazione del metodo verrà eseguita, dipenderà da qual'è l'istanza su cui quel metodo viene chiamato. Al chiamante quale sia l'implementazione non importa, proprio perché le sottoclassi possono essere adoperate ovunque si possa adoperare una superclasse (le diverse versioni del metodo accettano gli stessi tipi e restituiscono lo stesso tipo, quindi per la classe chiamante non cambia nulla).

Ereditarietà

L'ereditarietà è alla base della generalizzazione, senza ereditarietà le classi non potrebbero usare attributi od oggetti della superclasse, né ridefinire metodi della superclasse.

A volte si parla di ereditarietà multipla; questo avviene quando una classe è sottoclasse di più superclassi. Benché in certi frangenti possa essere utile essa introduce anche problematiche legate all'ereditarietà: se due superclassi hanno due attributi o due metodi con la stessa firma, la sottoclasse quale eredita? Come fa ad usare l'una o l'altra? Per questo motivo alcuni linguaggi (come il Java) permettono solo l'ereditarietà singola.

Classe astratta

Una classe astratta è una classe genitrice che non ha direttamente istanze: solo le sue sottoclassi possono essere istanziate. Generalmente una classe è astratta perché almeno uno dei suoi metodi è astratto: cioè definisce la firma, ma non ha implementazione.

Le classi astratte non possono venir istanziate ma sono molto utili nella modellazione. In un certo senso, esse definiscono un “vocabolario” di alto livello che arricchisce il linguaggio della modellazione.

Una classe astratta permette di definire alcune operazioni comuni, lasciando alle sue sottoclassi l'implementazione di operazioni specifiche. Ad esempio una classe astratta Video (per una videoteca) potrebbe fornire i metodi generici dei supporti visivi, lasciando astratta un'operazione spesaAffitto che indica il costo di affitto di un video. Le sue sottoclassi Videocassetta e DVD implementeranno il metodo.

Oggetto Classe

Abbiamo detto che le classi sono gli “stampi” per gli oggetti. In certi casi, però, bisogna riferirsi alla classe come un oggetto. Ciò avviene quando vi è bisogno che alcuni attributi siano comuni a tutte le istanze o dei metodi agiscano globalmente. In tal caso si parla di Oggetto Classe.

Un esempio di attributo comune è un contatore che conta il numero di istanze: essa non può essere parte di un oggetto, perché l'oggetto non sarebbe a conoscenza di altre istanze.

Un esempio di metodo globale, legato al precedente, è un metodo che restituisce il numero di istanze create fino a quel momento.

PARTE SECONDA

Ingegneria del Software

Una definizione generale dell'ingegneria del software, può essere: il processo da applicare al prodotto (il software) per portare a termine con successo, in breve tempo, a costo contenuto ed ad alta qualità, il progetto.

Una definizione esatta è data dall'IEEE, come l'unione dei seguenti due punti: “1) Applicazione di una strategia sistematica, disciplinata e misurabile allo sviluppo, esercizio e manutenzione del software. 2) Studio delle strategie di cui al punto 1)”.

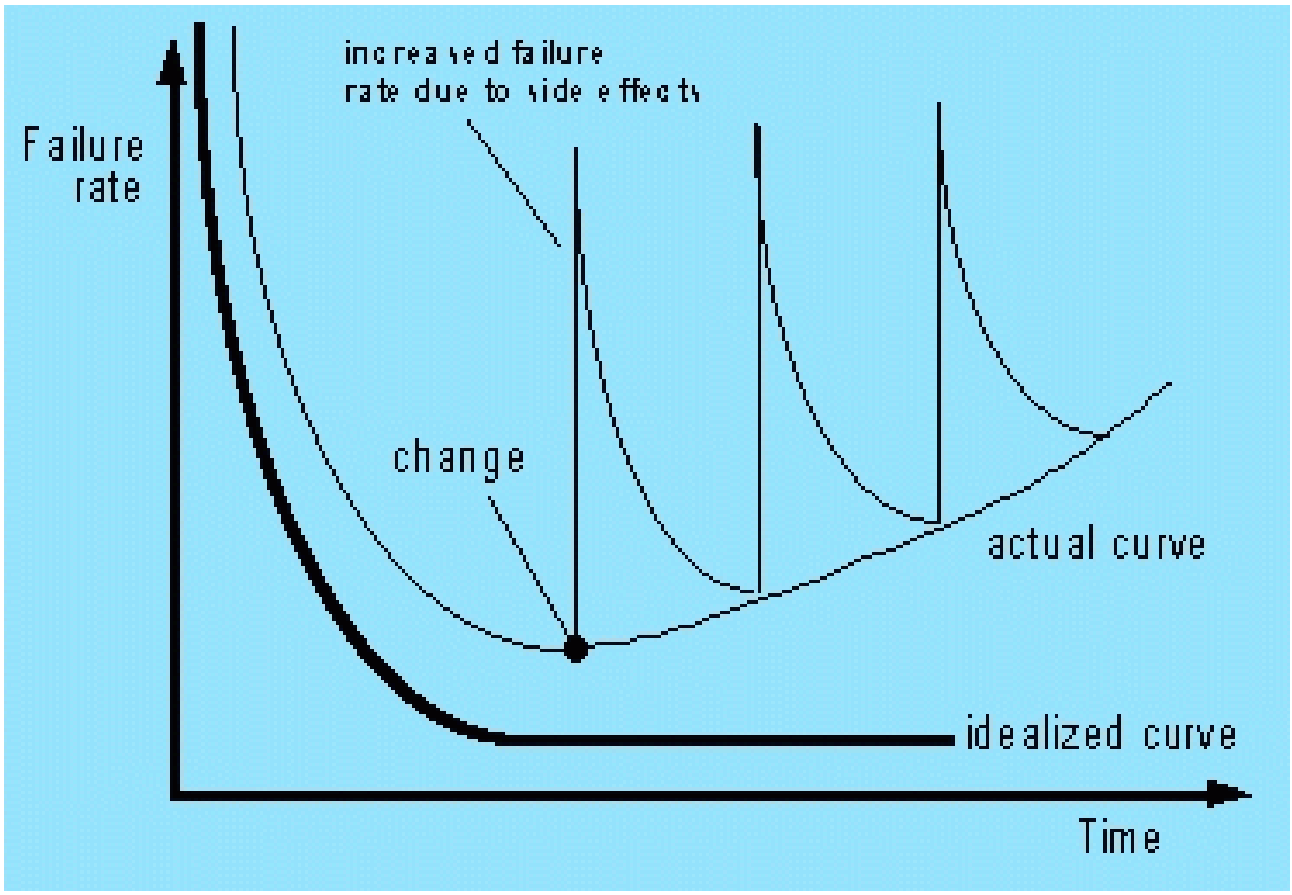
Dunque dobbiamo definire prodotto, processo e progetto.

Prodotto

Nel nostro caso il prodotto è il software che vogliamo realizzare. Andando nello specifico possiamo dire che il software ormai ci circonda: esistono numerosi sistemi manovrati dai computer e su di essi gira il software (basta pensare anche a quanti tipi di software esistono: di sistema, gestionali, embedded, real time, scientifico, per database, per videogiochi, per l'elaborazione di testi, per l'intelligenza artificiale, basato sul web, multimediale, ecc.). Normalmente quando si cerca di pensare al software (inteso come il prodotto di un certo processo), viene da paragonarlo alla produzione di un qualche oggetto. La similitudine però ha molte pecche:

1. Il software si sviluppa e si struttura, non si fabbrica: i metodi per fabbricare oggetti reali sono nettamente diversi dai metodi per sviluppare un software
2. Il software non si consuma: l'hardware tende a danneggiarsi nel corso del tempo, il software no. Tuttavia anche il software può essere soggetto a difetti, ma mentre nell'hardware il difetto corretto non provoca altri problemi (non verissimo, ma corretto dal nostro punto di vista), nel software la modifica può ripercuotersi su altre parti aggiungendo nuovi difetti. La curva di “deterioramento” può essere rappresentata dal successivo diagramma:

Dove la idealized curve è la curva ideale che accadrebbe se le correzioni e le nuove aggiunte non inserissero nuovi difetti. L'actual curve mostra la curva reale.



3. La costruzione di materiali usa pezzi già pronti e li combina solamente, il software è spesso realizzato ad hoc. Questo in parte è dovuto al fatto che non sempre il collegamento di sistemi già pronti porta al sistema finale; oltre a questo è decisamente più difficile sviluppare pensando al riuso ed i tempi stretti fanno sì che spesso le soluzioni siano cercate solo nel range interessato (e di conseguenza non riutilizzabili). L'ingegneria del software cerca di invertire la tendenza e cerca di estendere a tutto il prodotto quello che succede adesso nelle interfacce utente (pulsanti, caselle di testo, ecc. sono elementi standard riutilizzati).

I miti

Prima di vedere che cos'è il processo, vediamo di sfatare alcuni miti.Questi miti sono nati agli inizi degli anni '70, ma alcuni di essei potrebbero essere ancora validi.

Miti del management

Mito: abbiamo già interi volumi di standard e procedure da seguire nello sviluppo. Non c'è forse tutto l'indispensabile?

Realtà: Quanti di questi standard e procedure sono conosciute ed applicate? Sono seguiti in maniera completa ed ottimizzata? Sono usati standard e procedure che riflettano i metodi moderni di sviluppo del software? La risposta a tutte queste domande è NESSUNA.

Mito: Abbiamo i più moderni strumenti di sviluppo. Dopo tutto acquistiamo sempre i computer più recenti.

Realtà: Non è vero che al computer più potente corrisponde un miglior sviluppo del software. È come dare una formula 1 ad uno che deve imparare a guidare... Spesso sarebbe più conveniente

spendere i soldi in strumenti adatti allo sviluppo, piuttosto che potenziare i computer, ma costringere i programmatori a lavorare a mano (per mancanza di strumenti).

Mito: se siamo in ritardo possiamo recuperare aumentando il numero di programmatori.

Realtà: Sviluppare un software non è, come abbiamo detto, costruire un oggetto. Immettere "un'orda mongola" all'interno del processo di sviluppo rallenterà ancora di più lo sviluppo, in quanto la nuova gente dovrà essere addestrata, dovrà capire cosa hanno fatto gli altri programmatori, ecc.

Mito: se decido di far realizzare un progetto software ad una terza parte, posso stare tranquillo perché tutto il lavoro verrà svolto esternamente.

Realtà: Le "terze parti" non hanno, spesso, idea del funzionamento interno; se non si sa gestire i propri processi interni, è assai difficile riuscire a gestire il processo svolto da uno esterno.

Miti della clientela

Mito: un'affermazione generica degli scopi è sufficiente per cominciare a scrivere i programmi; i dettagli si possono trattare in seguito.

Realtà: un'insufficiente descrizione preliminare è la causa principale del fallimento del progetto. Senza un'accurata descrizione, non si riesce a definire i confini del progetto.

Mito: I requisiti di un progetto mutano di continuo, ma i mutamenti si gestiscono agevolmente grazie alla flessibilità del software.

Realtà: Il software non è flessibile, né tanto meno lo è svilupparlo. È vero, i requisiti cambiano, ma apportare il cambiamento durante le prime fasi del progetto è più facile e meno costoso, che non quando il software è pronto. Su questo argomento avremo modo di tornare per un approfondimento.

Miti del programmatore

Mito: Una volta scritto e messo in opera un programma, il nostro lavoro è finito.

Realtà: La maggior parte del lavoro avviene una volta che il software è stato rilasciato.

Mito: Fino a quando il programma non è in condizioni di essere eseguito, non c'è modo di valutarne la qualità.

Realtà: Uno dei metodi più efficaci di esame della qualità del software, la revisione tecnica formale, si applica già dall'inizio di un progetto. Avremo modo di approfondire l'argomento.

Mito: il solo prodotto di un progetto concluso è il programma funzionante.

Realtà: un programma funzionante è solo una parte del software. La documentazione è il fondamento di uno sviluppo riuscito e, cosa ancora più importante, è la guida dell'attività di manutenzione.

Mito: l'ingegneria del software ci farà scrivere un'inutile e voluminosa documentazione che inevitabilmente rallenterà le cose.

Realtà: l'ingegneria del software punta sulla qualità, la migliore quantità porta ad un minor lavoro, un minor lavoro porta a tempi di consegna più rapidi. Quello che porta via tempo è imparare l'ingegneria del software (insomma, deve essere un investimento). Quando si pensa che non vi è tempo per svolgere un'ingegnerizzazione del software, ci si provi a chiedere: "Avremo tempo di rifarlo da zero?".

Processo

Il processo software è un quadro di riferimento entro il quale si svolgono le attività necessarie alla realizzazione di software di alta qualità.

Abbiamo definito l'ingegneria del software come processo; in realtà essa include i metodi, le tecniche e gli strumenti.

Nonostante questo, il processo rimane l'ambito chiave su cui ci si muove, infatti in esso si stabiliscono le aree chiave, il contesto in cui si applicano i metodi tecnici, si creano i prodotti intermedi (modelli, documenti, dati, resoconti, formulari, ecc.), si stabiliscono i punti di controllo (o pietre miliari), si garantisce la qualità e si governano in modo opportuno le modifiche.

I metodi costituiscono il sapere tecnico relativo alla costruzione del software. Essi investono una vasta gamma di attività, che comprende l'analisi dei requisiti, la progettazione, la costruzione dei programmi, il collaudo ed il supporto. I metodi dell'ingegneria del software poggiano su una famiglia di principi di base che governano ciascuna delle sue aree e comprendono le attività di modellazione ed altre tecniche descrittive.

Gli strumenti dell'ingegneria del software forniscono al processo ed ai metodi un supporto automatizzato, in tutto od in parte.

Dando una rapida panoramica, possiamo dire che l'ingegneria del software tende a dare una risposta a queste domande:

- Qual'è il problema da risolvere?
- Quali caratteristiche delle entità vengono utilizzate per risolvere il problema?
- In che modo l'oggetto (e quindi la soluzione) sarà realizzato?
- In che modo l'oggetto sarà costruito?
- In che modo si condurrà la ricerca degli errori compiuti della progettazione e costruzione dell'oggetto?
- Quale supporto si darà all'oggetto a lungo termine, quando i suoi utenti richiederanno correzioni, adattamenti e miglioramenti?

Indipendente dall'area applicativa, dalle dimensioni del progetto e dalla sua complessità, le operazioni coinvolte nella realizzazione del software si articolano in tre fasi generali, ciascuna delle quali è investita da una o più delle domande indicate in precedenza.

La fase di definizione si occupa del che cosa. In questa fase lo sviluppatore cerca di determinare quali siano le informazioni da elaborare, quali siano le funzioni e prestazioni attese, quale debba essere il comportamento del sistema, quali interfacce si debbano definire, quali siano i vincoli progettuali ed i criteri di validazione in base ai quali valutare la riuscita del risultato. Si devono determinare i requisiti fondamentali del sistema e del software.

La fase di sviluppo si occupa del come. In questa fase un ingegnere del software definisce come strutturare i dati, come implementare le funzioni entro una data architettura software, come implementare i dettagli procedurali, come strutturare le interfacce, come tradurre un progetto complessivo in un linguaggio di programmazione e come svolgere i collaudi.

La fase di supporto si occupa delle modifiche legate alla correzione degli errori, agli adattamenti che si sono resi necessari per l'evoluzione dell'ambiente del software e delle modifiche tese a soddisfare nuove esigenze della clientela. In questa fase si applicano nuovamente le attività di definizione e sviluppo, ma nel contesto del software esistente. Si possono individuare quattro tipi di modifiche:

- **Correzioni** – Anche in presenza delle migliori tecniche di controllo della qualità, è probabile che gli utenti scoprano difetti nel prodotto software. La *Manutenzione correttiva* modifica il software allo scopo di eliminare i difetti.
- **Adattamenti** – Con il passare del tempo, è probabile che si evolva l'ambiente (CPU, sistema operativo, regole aziendali, caratteristiche esterne del prodotto, ecc.) per cui il software è stato sviluppato. La *Manutenzione adattiva* consiste in modifiche al software tese ad adattarlo al nuovo ambiente operativo.
- **Migliorie** – Con l'utilizzazione del software, emerge l'utilità di nuove funzioni. La *Manutenzione perfettiva* estende il software oltre i requisiti funzionali originari.
- **Prevenzione** – Il software si deteriora a causa delle modifiche; perciò allo scopo di garantire che il software svolga il proprio compito rispetto ai suoi utenti, si rende necessaria la *Manutenzione preventiva*, detta anche *software reengineering*. In sostanza, la manutenzione preventiva consiste in modifiche che rendano più semplici le correzioni, gli adattamenti e le migliorie.

Le fasi e le relative attività descritte in questa panoramica sono completate da attività ausiliarie, che si svolgono durante tutto il processo:

- Controllo e guida del progetto
- Revisioni tecniche formali
- Garanzia di qualità del software
- Gestione delle configurazioni software
- Preparazione e produzione di documenti
- Gestione del riutilizzo
- Misurazioni
- Gestione dei rischi.

Negli ultimi anni il SEI (Software Engineering Institute) ha ideato un modello a cinque livelli che misura l'efficacia globale dell'applicazione di tecniche di ingegneria del software da parte di un'azienda:

Livello1: *Iniziale* – Il processo software è definito di volta in volta ed in alcuni casi risulta confuso. Pochi processi sono definiti e la riuscita dipende dall'impegno individuale.

Livello2: *Ripetibile* – Presenza di processi basilari di gestione di progetti, volti a sorvegliare costi, tempi e funzionalità. La disciplina di processo necessaria ha lo scopo di ripetere il successo di progetti precedenti per applicazioni simili. Questo livello comprende:

- la gestione delle configurazioni software;
- la garanzia di qualità del software;
- la gestione dei sottocontratti per il software;

- il controllo e la sorveglianza del progetto;
- la gestione dei requisiti

Livello3: *Definito* – Il processo software, sia per le attività di gestione sia per quelle di realizzazione, è documentato, conforme ad uno standard ed inserito in un processo software su scala aziendale. Ogni progetto adotta una versione documentata ed approvata del processo aziendale per lo sviluppo e la manutenzione del software. Questo livello comprende tutte le caratteristiche definite per il Livello2. Questo livello comprende:

- le revisioni;
- il coordinamento dei gruppi;
- l'ingegneria del prodotto software;
- la gestione integrata del software;
- i programmi di addestramento;
- la definizione del processo aziendale;
- l'obbiettivo primario del processo aziendale

Livello4: *Gestito* – Si raccolgono misure dettagliate relative al processo software ed alla qualità del prodotto. Sia il processo software sia i prodotti sono valutati quantitativamente e controllati mediante misure dettagliate. Questo livello comprende tutte le caratteristiche definite per il Livello3. Questo livello comprende:

- la gestione della qualità del software;
- la gestione quantitativa del processo

Livello5: *Ottimizzato* – Il continuo miglioramento del processo è consentito dalla raccolta di dati quantitativi e dal collaudo di idee e tecnologie innovative. Questo livello comprende tutte le caratteristiche definite per il Livello4. Questo livello comprende:

- la gestione del cambiamento del processo;
- la gestione del cambiamento tecnologico;
- la prevenzione dei difetti

Modelli di processo

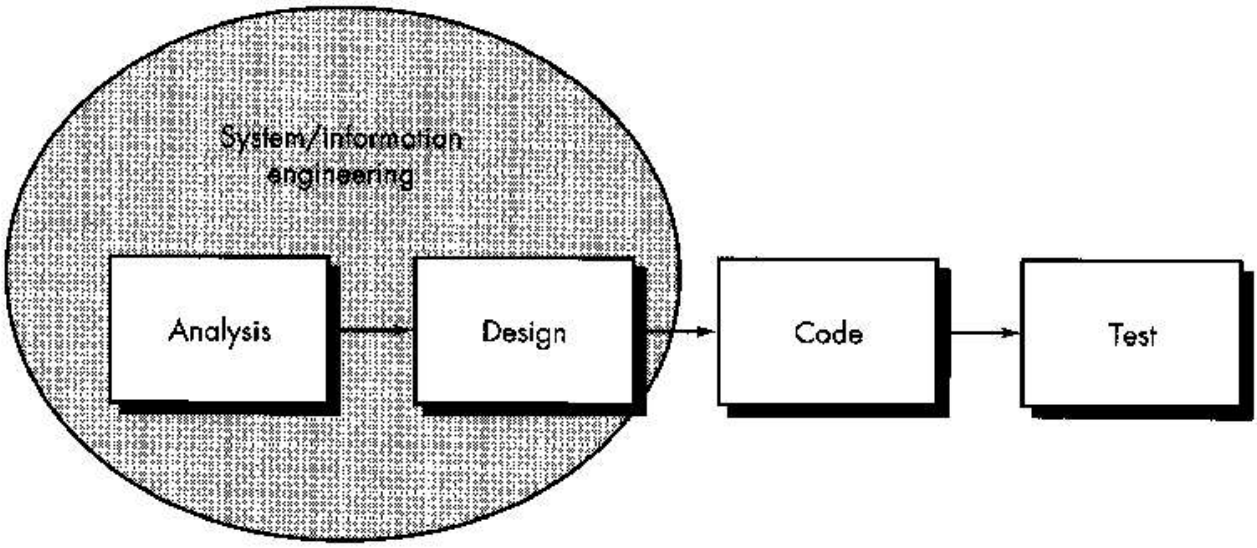
Lo sviluppo del software possiamo vederlo come un'attività divisa in quattro fasi: Status quo, Definizione del problema, Sviluppo tecnico, Integrazione della soluzione; se vediamo queste attività come l'una il seguito dall'altra allora dall'ultima si può ritornare alla prima e ricominciare il processo. Lo status quo rappresenta lo stato delle cose presente, la definizione del problema individua lo specifico problema da risolvere, lo sviluppo tecnico lo risolve, l'integrazione della soluzione consegna i risultati. Vedere così lo sviluppo del software è un po' semplicistico, per due motivi:

- ✔ Ogni fase è in realtà una macro fase, che può essere a sua volta divisa nelle stesse fasi, che a loro volta possono essere divise, ecc.
- ✔ Le fasi non sono così distinte, ma è difficile separarle: infatti ogni fase ha più di un punto in comune con tutte le altre fasi.

Per questi motivi, lo sviluppo del software è considerata un'attività caotica. I modelli che qui vengono presentati cercano di porre ordine a tale caos.

Modello sequenziale lineare

È stato il primo modello che fu costruito: esso si presenta come in figura:

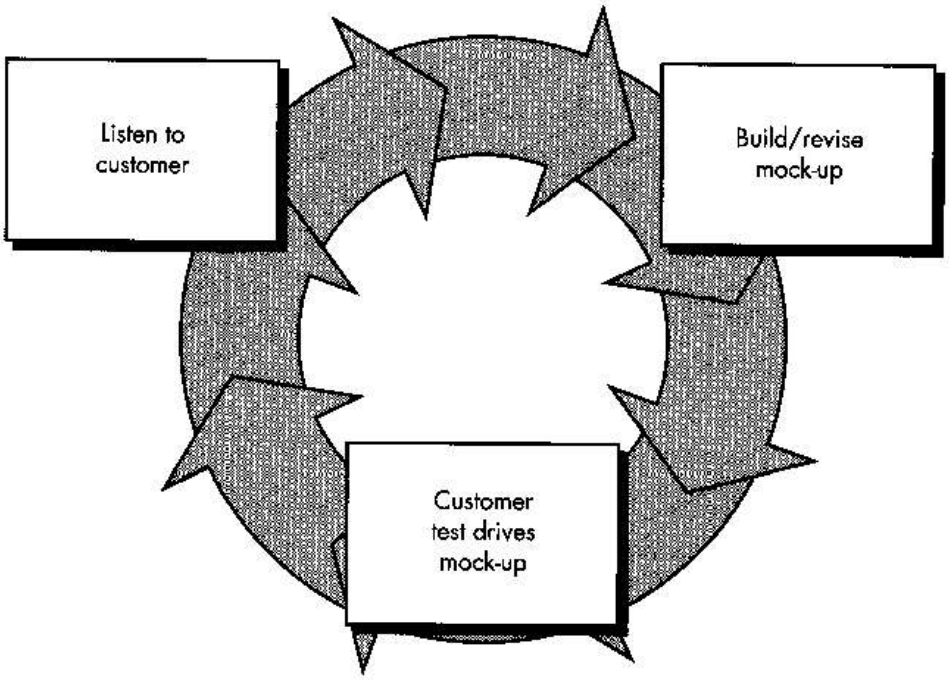


In questo modello la fase di analisi precede quella di design che precede quella di scrittura del codice, che precede il collaudo. Una fase non può cominciare prima che sia finita l'altra, una volta conclusa una fase non ci si mette più mano.

I problemi si vedono subito: infatti i progetti reali seguono raramente un flusso sequenziale (i cambiamenti causano confusione), vi è difficoltà per il cliente di formulare i requisiti esplicitamente, il cliente deve avere pazienza (e spesso non ne ha), gli sviluppatori attendono inutilmente ‘stati bloccanti’ (come abbiamo detto una fase non inizia se l’altra non è portata a termine).

Modello del Prototyping

Questo modello lo possiamo riassumere come in figura nella pagina successiva:



Come si vede è ciclico: si parte dall'ascoltare l'utente, si prosegue costruendo un prototipo, lo si fa testare al cliente, lo si riascolta, si modifica il prototipo, lo si fa ritestare e così via fin quando il software non è pronto.

I prototipi vengono costruiti in fretta, con scelte inadeguate, algoritmi inefficienti ed altre cose che rendono funzionante il prototipo in tempi brevi, ma tenuto insieme "con lo scotch". Quando si vuole modificare, molte scelte devono essere gettate via, rielaborate, trovati algoritmi più efficienti, ecc.

I problemi che sorgono possono essere tre:

1) Il fenomeno del throw-away: si ha spesso reticenza a gettare tutto il lavoro fatto
2) Aspettative irrealistiche da parte del cliente (vedendo un prototipo, potrebbe pensare che per il software completo manchi poco e che possa essere consegnato in tempi troppo stretti)
3) Mettendo insieme i due problemi, otteniamo spesso un prodotto il cui problema è nella manutenibilità: alla prima modifica fatta si verificano tutta una serie di side-effect che non fanno più funzionare il programma nel modo previsto.

Modello RAD

Il Modello RAD (Rapid Application Development) è un modello che prevede di suddividere il lavoro in team paralleli, ogni team segue un processo sequenziale lineare su un periodo breve; il processo sequenziale lineare è diviso nei seguenti passi:

- Business modeling (*quali dati guidano il processo? Chi li genera? Dove vanno? Chi li elabora?)*
- Data modeling (*attributi degli oggetti e loro relazioni*)
- Process modeling (*operazioni di aggiunta, modifica, cancellazione o recupero di oggetti*)
- Application generation (*utilizza componenti esistenti e crea componenti riutilizzabili*)

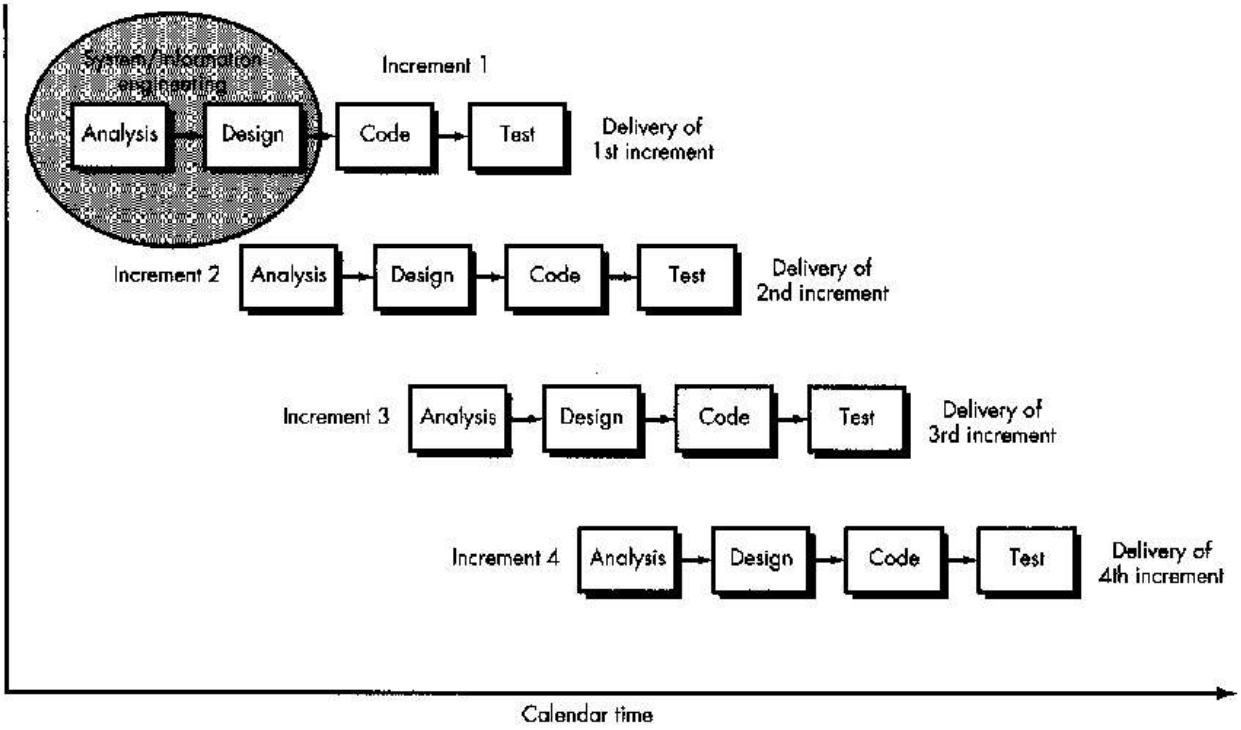
- Testing and turnover (*collaudo dei nuovi componenti*)

Il modello RAD ha i seguenti difetti:

- Per grandi progetti occorrono risorse sufficienti per creare le squadre
- Forte impegno degli sviluppatori e dei clienti
- Presuppone una soluzione modulare
- Riusabilità può implicare perdita di prestazioni

Modello Incrementale

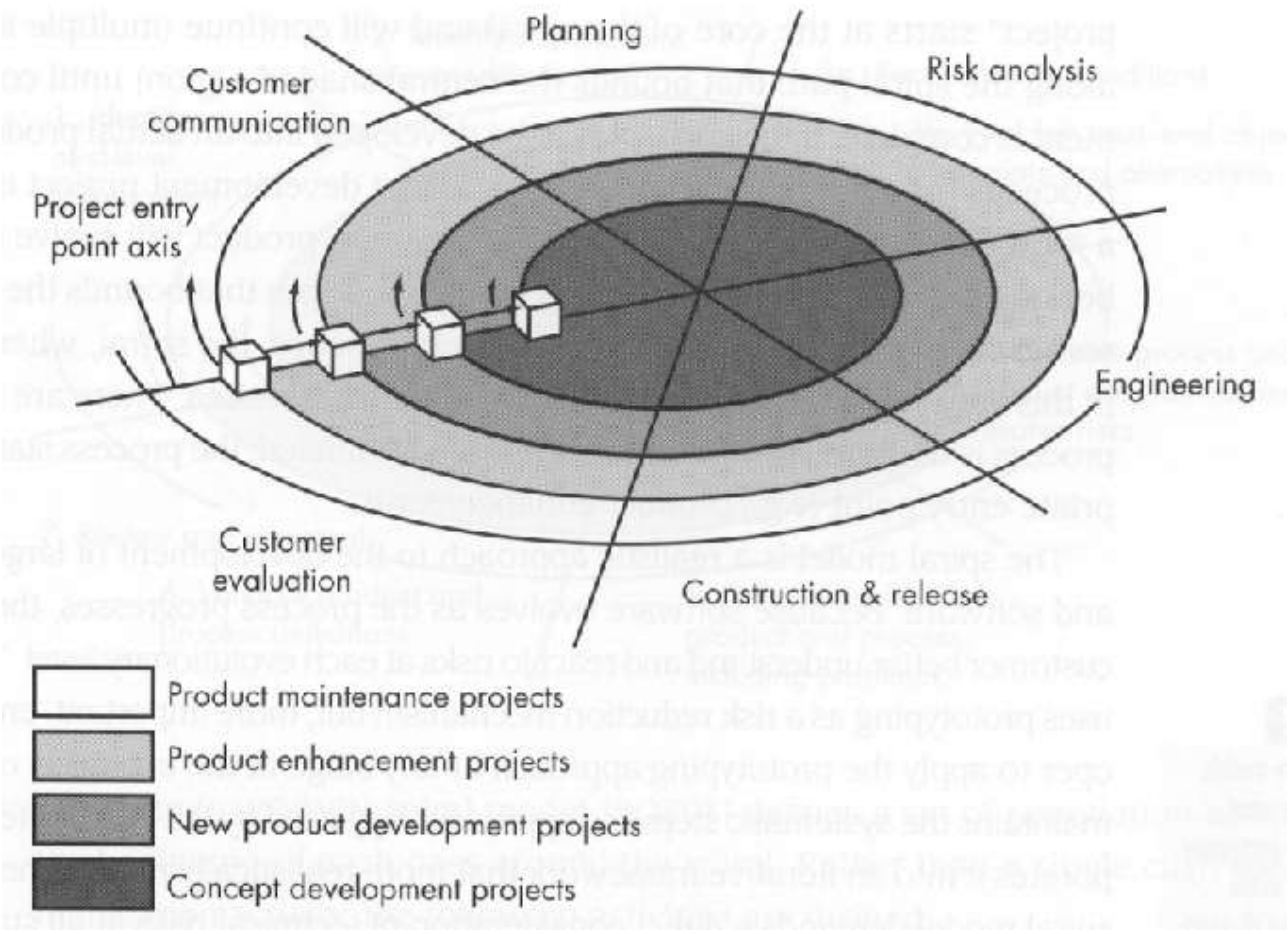
Il modello incrementale riprende il modello sequenziale lineare e vi aggiunge la possibilità di inserire incrementi. In altre parole il prodotto non viene dato immediatamente completo di ogni funzionalità, ma, ad ogni ciclo, vengono sviluppate singole funzionalità il cui insieme corrisponderà al prodotto finale. La figura rappresenta tale modello.



Ha gli stessi difetti del modello sequenziale lineare, anche se in maniera più mitigata, grazie alla realizzazione degli incrementi.

Modello a spirale

Il modello a spirale è rappresentato in figura



Con questo modello si percorre una spirale dall'inizio del progetto fino alla fuoriuscita del prodotto, ogni spira è divisa in settori: Comunicazione col cliente, Pianificazione, Analisi del Rischio, Progettazione, Costruzione e Rilascio, Valutazione da parte del cliente.

Il problema di questo modello è la difficile visualizzazione della taglia e temporizzazione di ogni spira.

Modello a spirale WINWIN

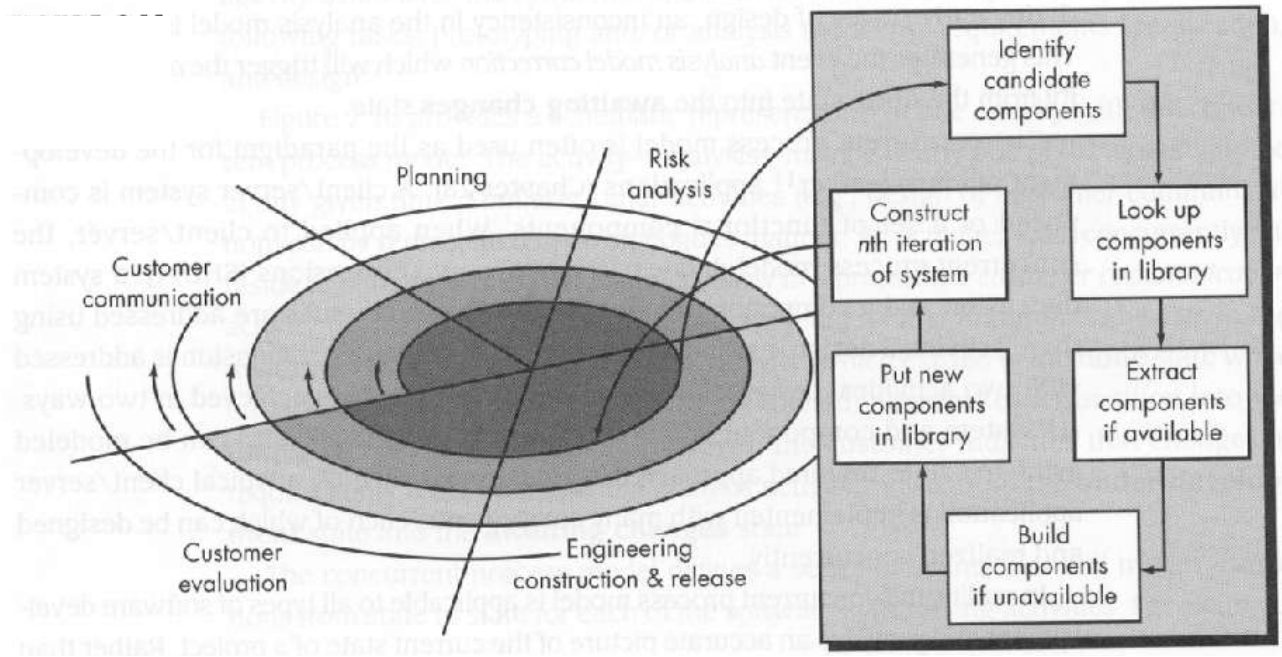
Questo è una modifica del modello a spirale; con questo modello cambiano i settori:

- Identificare gli stakeholder (con questo termine si intende chiunque abbia un qualche interesse al prodotto/processo/progetto)
- Identificare le condizioni di vittoria di ogni stakeholder (con questo si intende quando uno stakeholder si possa ritenere soddisfatto del prodotto/processo/progetto)
- Negoziazione delle condizioni di vittoria, stabilire gli obiettivi, i vincoli e le alternative
- Valutare il processo, le alternative e risolvere i rischi
- Definire il prossimo livello
- Validare il prodotto e la definizione di processo
- Commenti

Il difetto di questo modello è che non sempre le condizioni di vittoria sono facilmente trovabili ed ancor meno negoziabili.

Modello ad assemblaggio di componenti

Questo è una modifica al modello a spirale, introducendovi la riusabilità object oriented

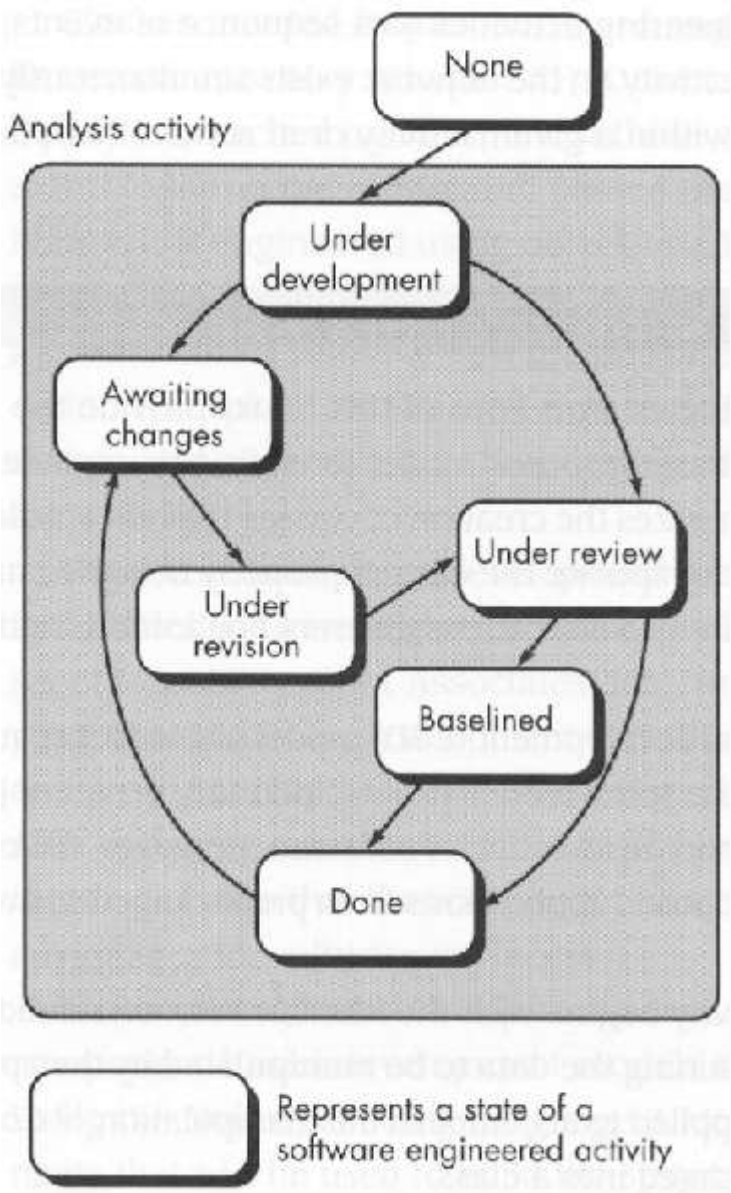


I difetti sono i seguenti:

- Riusabilità richiede pianificazione attenta
- Molti programmi esistenti non sono riusabili
- Più adatto per domini applicativi particolari (con schemi significativi di riuso)

Modello dello sviluppo concorrente

Con questo modello si usano dei diagrammi di stato per ogni attività, ogni evento fa scattare una transizione di stato. È un modello utile per sviluppo dell'organizzazione interna dell'azienda e quando c'è un alto grado di interdipendenza tra moduli diversi (es. applicazioni client-server)



Altri modelli

Nel corso del tempo sono venuti a formarsi altri modelli, li presentiamo in rapida successione in quanto o sono in corso di studio o approfondiremo più avanti:

- Specifiche Tecniche formali – basata sulla logica matematica
- Tecniche "di quarta generazione" – usando cioè linguaggi di definizione ad alto livello
- MDA (Model Driven Architetture) – l'idea è quella di sviluppare software indipendentemente dalla piattaforma (PIM – Platform Indipendent Model) avendo metodi di trasformazione per portarlo sotto piattaforme specifiche (PSM – Platform Specific Model) in breve tempo e mantenendo coerenza – Tale sistema è utile per sistemi eterogenei

- Agile Software Programming – basato sul concetto che grandi persone fanno grandi programmi, per ogni pezzo di programma vi sono due responsabili (uno propositore di soluzioni, uno critico di soluzioni, in alternanza) e che le soluzioni sono condivise da tutti i programmatori
- Unified Process – che vedremo approfonditamente più avanti

Progetto

Il progetto possiamo definirlo come quell'attività che, seguendo le linee guida date dal processo, porta alla realizzazione del prodotto.

Esistono dieci punti con i quali si può capire quando un progetto è a rischio:

1. Gli sviluppatori non comprendono le esigenze dei loro clienti
2. Il campo d'azione del prodotto è mal definito
3. Le modifiche vengono gestite in modo non appropriato
4. Cambia la tecnologia scelta
5. Le operazioni devono cambiare o sono mal definite
6. Le date di consegna non sono realistiche
7. Gli utenti sono ostili
8. Perdita delle risorse finanziarie
9. Il team del progetto non contiene persone con le conoscenze appropriate
10. Il manager ed i professionisti non mettono a frutto le lezioni apprese

Come può un manager evitare i problemi appena enunciati? Si suggerisce un approccio di buon senso costituito da cinque punti:

1. **Partire col piede giusto**: ciò si ottiene innanzitutto lavorando duramente (molto duramente) per comprendere il problema che deve essere risolto e poi definendo gli oggetti e le attese realistiche per tutti coloro che saranno interessati dal progetto. Ciò deve essere ribadito realizzando il team corretto ed assegnando al team l'autonomia, l'autorità e le tecnologie necessarie per svolgere il lavoro.
2. **Mantenere alta l'attenzione**: molti progetti partono bene ma poi si disintegrano lentamente. Per mantenere alta l'attenzione, il manager del progetto deve fornire incentivi per ridurre assolutamente al minimo il ricambio di personale; il team deve concentrarsi sulla qualità in ogni operazione svolta ed il management deve fare il possibile per rispondere alle esigenze del team (burocrazia ridotta al minimo, eliminare gli incontri inutili e le richieste di aderenza dogmatica alle regole del processo, ecc.).
3. **Verificare i progressi**: per un progetto software, i progressi devono essere controllati a mano a mano che viene creato un prodotto (ad esempio le specifiche, il codice sorgente, i casi di collaudo) e poi approvati (utilizzando revisioni tecniche formali) nell'ambito di un'attività di verifica della qualità. Inoltre le misure dei processi e del progetto software possono essere raccolte ed utilizzate per stabilire i progressi rispetto alle indicazioni sviluppate dall'organizzazione di sviluppo software.
4. **Prendere decisioni intelligenti**: in pratica le decisioni del project manager e del team software devono preferire la semplicità. Quando possibile si deve decidere di utilizzare software commerciale pronto all'uso o componenti software esistenti, decidere di evitare interfacce personalizzate

quando sono disponibili approcci standard, decidere di identificare e poi evitare i rischi più evidenti e decidere di allocare più tempo di quanto si possa pensare alle operazioni più complesse e rischiose.

5. **Condurre un'analisi post-mortem**: stabilire un meccanismo generale per trarre lezioni da ogni progetto. Valutare i tempi pianificati ed ottenuti, raccogliere ed analizzare le valutazioni metriche del progetto software, chiedere le opinioni dei membri del team e dei clienti e trascrivere queste informazioni.

La gestione efficace di un progetto software si fonda sulle 3P: persone, problema, processo.

Persone

Le persone sono spesso la chiave di un progetto software ed il più delle volte date un po' troppo per scontate. Nel processo software ed in ogni progetto di sviluppo, intervengono varie persone che si possono classificare in cinque categorie:

1. *gli alti dirigenti*: i quali definiscono il contesto aziendale, che spesso influenza un progetto in modo significativo;
2. *i capi progetto*: incaricati di pianificare, motivare, coordinare e controllare il personale tecnico;
3. *i tecnici*: dotati delle competenze necessarie per la realizzazione concreta di un prodotto od un'applicazione;
4. *i clienti*: che specificano i requisiti del software da realizzare ed altri punti di riferimento interni che hanno un interesse marginale sul risultato;
5. *gli utenti finali*: che interagiscono col software reso operativo.

Normalmente la struttura organizzativa di un'azienda è difficile che cambi, mentre i capi progetto ed i tecnici vengono di volta in volta assegnati a progetti diversi. È compito del capo progetto riuscire a far divenire un gruppo di persone, quello che viene definito un “team coeso”:

- ✔ I membri della squadra devono fidarsi l'uno dell'altro
- ✔ La distribuzione delle specifiche abilità deve essere appropriata per il problema
- ✔ È opportuno escludere dal team gli elementi particolarmente rissosi per aumentare la coesione all'interno del team
- ✔ Un team coeso è un gruppo di persone unite così strettamente che l'intero è più grande della somma delle parti.

Un team che acquista coesione permette di aumentare le probabilità di successo, non occorre gestire i suoi membri nei modi tradizionali, né motivarli. Essi hanno preso lo slancio.

Creare un team coeso, però, è più facile a dirsi che a farsi. Quello che si può provare ad evitare è un team “tossico”:

- ✗ Atmosfera di lavoro frenetica in cui i membri del team perdono energie e concentrazione sugli obbiettivi del lavoro.
- ✗ Un elevato livello di frustrazione causata da fattori personali, aziendali o tecnologici che provocano un attrito tra i membri.
- ✗ Procedure frammentate o poco coordinate od un modello di processo mal definito o mal scelto che impedisce il raggiungimento del risultato

- ✗ Una definizione non chiara dei ruoli, che provoca una mancanza di responsabilità
- ✗ Una continua e ripetuta serie di fallimenti che provoca pessimismo ed abbassa il morale.

Evitare un team tossico richiede di lavorare singolarmente su questi punti, evitando atmosfere frenetiche, abbassando la frustrazione, scegliendo le giuste procedure ed i modelli di processo, definendo chiaramente ruoli e responsabilità. Per l'ultimo punto si devono stabilire tecniche specifiche per la comunicazione e la soluzione dei problemi. Inoltre il fallimento di un membro del team deve riguardare l'intero team.

Ovviamente quando si formano i gruppi, affinché siano team, è necessario considerare anche le caratteristiche singole di ogni individuo: c'è infatti chi lavora meglio sotto stress, altri quando vi è più tempo, altri prendono decisioni seguendo un ragionamento logico, altri l'impulso del momento, alcuni raccolgono informazioni in modo intuitivo, altri in modo più organizzato, ecc. Un capo progetto che metta insieme un team deve tener conto di tutti questi fattori.

Oltre a quanto detto bisogna stabilire il tipo di struttura del team:

- **Democratico Decentrato (DD)**: Un team di questo tipo non ha un capo permanente. Di volta in volta vengono nominati coordinatori delle attività per brevi periodi, che sono poi sostituiti da altri, incaricati di coordinare attività diverse. Le decisioni sui problemi e sulle strategie sono prese per consenso. La comunicazione fra i membri è orizzontale. Tale struttura produce un morale ed un grado di soddisfazione alti, si adatta meglio a sistemi con bassa modularità a causa dell'alta quantità di comunicazioni che esige, sviluppa più difetti di altre strutture, impiega più tempo a portare a termine un progetto, ma sono i più indicati quando si richiede un alto grado di socievolezza.
- **Controllato Decentrato (CD)**: Un team di questo tipo ha un capo che coordina le attività e sottocapi responsabili delle sottoattività. Le decisioni si prendono collettivamente, ma l'implementazione delle soluzioni è distribuita fra i sottogruppi a cura del capo. La comunicazione fra sottogruppi e fra persone è orizzontale. Si ha anche una comunicazione verticale attraverso la gerarchia.Tale struttura è più adatta in caso di alta modularità e quindi relativa indipendenza fra le persone coinvolte, produce meno difetti ed in tempi più brevi rispetto ad una struttura DD, inoltre è più adatto per progetti di grandi dimensioni.
- **Controllato Accentrato (CA)**: Le decisioni ad alto livello ed il coordinamento del team sono affidati ad un capo. La comunicazione fra il capo ed i membri del team è verticale. Tale struttura è più adatta in caso di alta modularità e quindi relativa indipendenza fra le persone coinvolte, produce meno difetti ed in tempi più brevi rispetto ad una struttura DD, inoltre è più adatto per progetti di grandi dimensioni.

Detto tutto questo, la logica conseguenza è che un buon capo progetto abbia, od acquisisca in breve tempo, alcune caratteristiche:

- ✔ **Motivazione**: La capacità di incoraggiare (mediante spinte e controspinte) i tecnici a dare il meglio di sé
- ✔ **Organizzazione**: La capacità di plasmare i processi esistenti (o di inventarne di nuovi) così da tradurre un'idea iniziale in un prodotto finale
- ✔ **Idee ed innovazione**: La capacità di spingere le persone a creare ed ad essere creativi anche entro i limiti stabiliti per un prodotto od un'applicazione particolare
- ✔ **Problem Solving**: Un buon capo progetto è in grado di determinare i fattori tecnici ed organizzativi più importanti, strutturare sistematicamente una soluzione, motivare opportunamente i tecni-

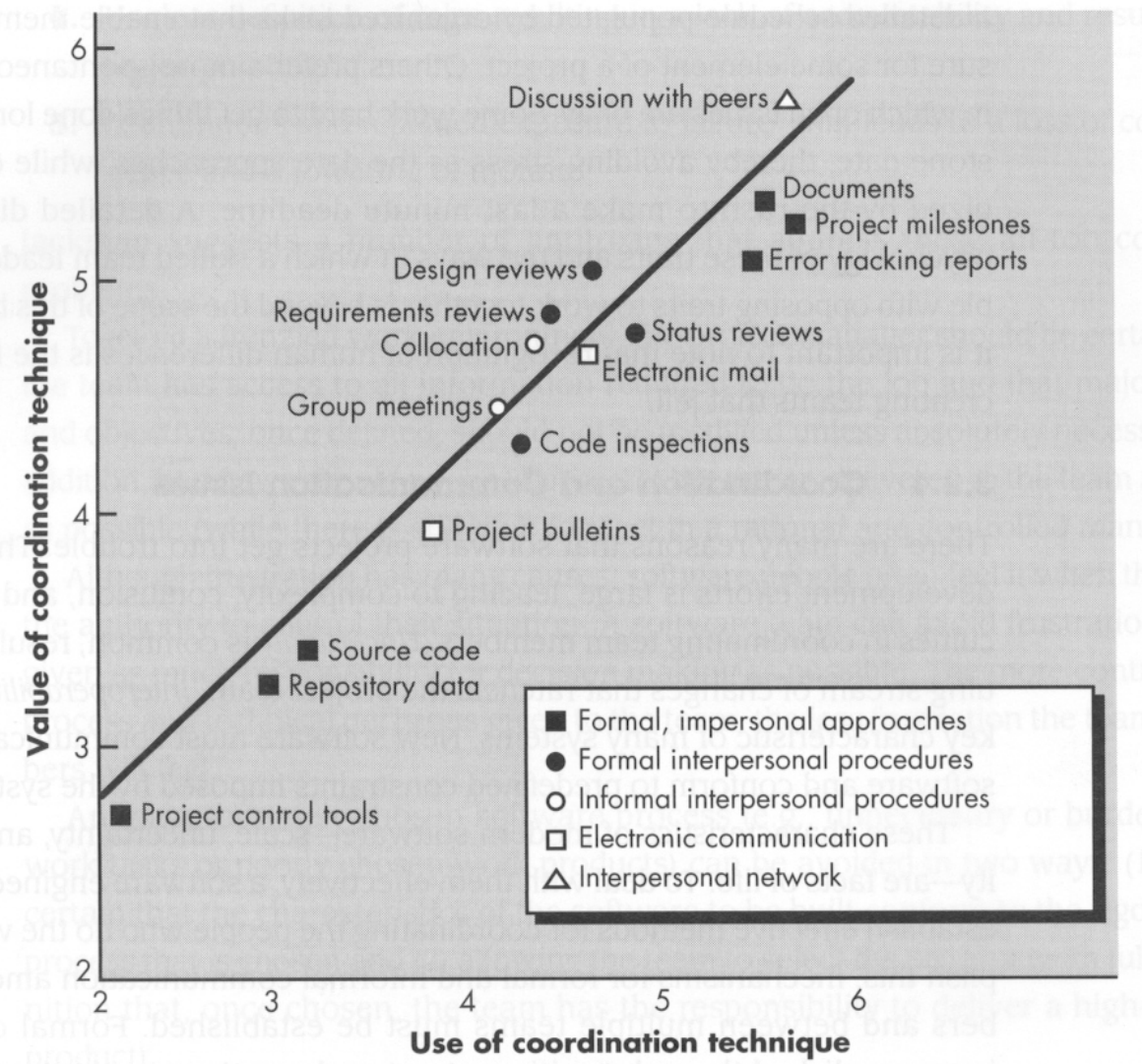
ci che devono svilupparla, applicare a nuove soluzioni le lezioni apprese nei progetti passati ed essere abbastanza flessibile da saper mutare direzione se i primi tentativi si rivelano infruttuosi

- ✔ **Identità manageriale**: Un buon capo progetto deve assumersi la responsabilità del progetto. Deve essere capace di tenere saldamente le redini quando è necessario e di lasciare che i tecnici più preparati seguano il proprio istinto
- ✔ **Incentivazione**: Per accrescere la produttività di un team, il capo progetto deve ricompensare lo spirito di iniziativa ed i risultati e deve dimostrare nei fatti di tollerare che si prendano rischi ragionevoli
- ✔ **Influenza e spirito di gruppo**: Un buon capo progetto deve "leggere" nelle persone; deve cogliere i segnali, verbali e no, e reagire alle esigenze che quei segnali suggeriscono. Deve mantenere il controllo anche in situazioni critiche.

Un altra cosa da considerare sono i metodi di comunicazione del gruppo:

- **Strategie formali, impersonali**: Comprende documenti e lavorati (ad esempio codice sorgente), note tecniche, punti di controllo del progetto, tabelle dei tempi e strumenti di controllo del progetto, richieste di cambiamenti e documenti relativi, relazioni su errori e dati d'archivio
- **Procedure formali, interpersonali**: Centrata sulle attività di garanzia della qualità, applicate ai "semilavorati" di un progetto software, fra le quali riunioni di verifica dello stato ed ispezioni del progetto e del codice
- **Procedure informali, interpersonali**: Comprende riunioni di gruppo per la diffusione di informazioni, per la risoluzione di problemi e per l'individuazione dei requisiti e la distribuzione dello staff
- **Comunicazioni elettroniche**: Comprende la posta elettronica, le banche dati elettroniche, i siti web e, per estensione, le videoconferenze
- **Rete interpersonale**: Includono discussioni informali con i membri del team e con coloro che sono all'esterno del progetto e possono avere esperienza od un punto di vista utile per i membri del team

Il grafico seguente mostra una ricerca fatta nel 1995 in cui fa vedere l'utilizzo delle tecniche citate ed il loro valore percepito. Le tecniche sotto la retta sono state giudicate meno valide.



Prodotto

Proprio all'inizio del progetto, il capo progetto si trova di fronte ad un dilemma. Egli deve definire stime quantitative ed un piano organizzativo, ma non dispone di dati certi. Un'analisi approfondita dei requisiti del software fornirebbe i dati necessari per le stime, ma spesso una tale analisi richiede settimane o mesi. Peggio ancora, i requisiti sono a volte fluidi e soggetti a cambiamenti nel corso del progetto. Tuttavia è necessario predisporre immediatamente un piano.

Di conseguenza, il prodotto ed il problema devono essere esaminati senza indugio all'inizio del progetto, determinando almeno la portata del progetto e dei suoi confini.

La portata si definisce rispondendo alle domande seguenti:

- **Contesto**: in che modo il software in costruzione si inserirà in un sistema, prodotto o contesto aziendale esistente e quali vincoli impone il contesto?
- **Obbiettivi relativi ai dati**: Quali dati visibili all'utente deve produrre il software? Quali dati formano il suo input?
- **Funzionalità e prestazioni**: Quali funzioni svolge il software per trasformare i dati di input in quelli di output? Si devono considerare aspetti specifici legati alle prestazioni?

La portata del software deve essere espressa con precisione ed in modo comprensibile sia a livello manageriale sia a quello tecnico. L'*enunciato della portata* deve precisare dei *limiti*. In altre parole devono essere indicati esplicitamente i dati quantitativi (ad esempio numero di utenti simultanei, dimensioni di una mailing list, massimo tempo di risposta accettabile); devono essere indicati i vincoli e le limitazioni (ad esempio limiti di memoria imposti dai costi); devono essere descritti i fattori facilitanti (ad esempio il fatto che gli algoritmi necessari siano ben noti e disponibili).

Trovare tutti questi aspetti in maniera completa, richiede troppo tempo. La soluzione migliore è riuscire a scomporre il problema ed estrarre le funzioni più importanti (senza le quali il prodotto non è quello voluto), i limiti più importanti (senza i quali il prodotto non è soddisfacente), ed i dati principali. L'obbiettivo è quello che trovato il contesto si riesca a risalire ai dati, funzionalità e prestazioni senza le quali il prodotto non è quello aspettato. Avendo questi dati è possibile iniziare finalmente le stime.

Processo

Formato il team e trovata la portata del progetto è giunto il momento di mettere in moto il processo che porterà alla realizzazione del prodotto. Ma quale che sia il processo che uno voglia seguire, bisogna riuscire a pianificare le varie operazioni ed assegnare le persone.

Un utile sistema è prevedere una tabella dove nelle colonne vengono riportate le varie fasi e/o attività necessarie al processo, nelle righe le varie funzionalità del prodotto, nell'incrocio tra riga e colonna si inseriscono i vari compiti necessari. Per ogni casella sarà poi necessario assegnare le risorse, decidere tempi di inizio e fine, ecc. Ovviamente i compiti necessari, le risorse necessarie, ecc. variano da progetto a progetto, in quanto dipendono tanto dal tipo di processo scelto, quanto dal prodotto che si vuole realizzare.

Il principio WWWWHH

Questo principio prende il nome dalle iniziali delle 6 domande a cui un team deve saper rispondere per vedere se il progetto è ben delineato:

- **Perché (Who) si sviluppa il progetto?** La risposta a questa domanda consente a tutte le parti di stabilire la validità dei motivi pratici per cui il software deve funzionare. In altre parole, lo scopo giustifica l'impegno di persone, tempo e denaro?
- **Cosa (What) verrà eseguito ed entro quando?** La risposta a queste domande aiuta il team a stabilire i tempi del progeto identificando le operazioni principali del progetto ed i punti fermi richiesti dal cliente.
- **Chi (Who) è il responsabile di una funzione?** All'inizio si è detto che il ruolo e le responsabilità di ciascun membro del team deve essere ben definito. La risposta a questa domanda aiuta ad ottenere ciò.
- **Dove (Where) sono le responsabilità organizzative?** Non tutti i ruoli e le responsabilità risiedono nel team software. Anche il cliente, gli utenti ed i contatti interni del cliente hanno delle responsabilità.
- **In quale modo (How) il lavoro verrà eseguito sia tecnicamente che gestionalmente?** Una volta che è stato definito il campo d'azione del prodotto, occorre definire una strategia gestionale tecnica del progetto.
- **In quale quantità (How much) è richiesta ogni singola risorsa?** La risposta a questa domanda deriva dalle stime di sviluppo sulla base delle risposte alle precedenti domande.

PARTE TERZA

Metriche

Vi sono quattro motivi che spingono a misurare il processo ed il progetto software: caratterizzare, valutare, prevedere, migliorare.

Si caratterizza per acquisire conoscenze sui processi, i prodotti le risorse e gli ambienti e per stabilire indicazioni di base per il confronto con i prossimi lavori.

Si valuta per determinare lo stato rispetto ai piani. Le misurazioni sono i sensori che ci fanno conoscere quando i progetti ed i processi sono fuori controllo, in modo da poter applicare misure correttive. Inoltre si valuta per stabilire il raggiungimento degli obbiettivi di qualità e per stabilire l'impatto dei miglioramenti tecnologici e di processo sui prodotti e sui processi.

Si prevede per pianificare. La misura per scopi di previsione prevede l'acquisizione di conoscenze sulle relazioni fra processi e prodotti e la creazione di modelli per la realizzazione di tali relazioni in modo che i valori osservati per determinati attributi possano essere utilizzati per prevederne altri. Ci si comporta in questo modo perché si vogliono stabilire degli obbiettivi raggiungibili nell'ambito dei costi, dei tempi e della qualità in modo da poter applicare le risorse appropriate. Le misure di previsione fungono anche da base per determinare le tendenze in modo da poter stimare i costi, i tempi e la qualità sulla base dei valori attuali. Le proiezioni e le stime basate su dati storici aiutano anche ad analizzare i rischi ed a seguire valutazioni sul progetto e sul costo.

Si misura per ottenere un miglioramento quando si raccolgono informazioni quantitative che aiutano ad identificare i punti critici, le inefficienze e le altre opportunità di miglioramento della qualità del prodotto e delle prestazioni di un processo.

Anche se i termini misura, misurazione e metrica sono spesso considerati sinonimi, è importante sottolineare le sottili differenze:

- **Misura**: fornisce una misurazione quantitativa del grado, ammontare, dimensione, capacità di un attributo di un prodotto o di un processo. Le misure possono essere *dirette* (righe di codice prodotte, velocità di esecuzione, memoria occupata, difetti rilevati in un dato intervallo temporale, ecc.) od *indirette* (funzionalità, qualità, complessità, efficienza, affidabilità, facilità di manutenzione, ecc.).
- **Misurazione**: l'atto di determinare la misura.
- **Metrica**: mette in relazione, secondo certi criteri, le singole misure.
- **Indicatore**: è una combinazione di metriche che mette in risalto certe caratteristiche di un processo, di un progetto o di un prodotto.

Il difetto delle metriche software è il fatto che esse non misurano qualcosa di tangibile e come tale non c'è accordo su cosa si deve misurare, come misurarlo, quali metriche ed indicatori adottare.

Metriche di processo

Esiste un solo modo razionale di migliorare un processo: misurare gli attributi specifici, ricavarne una famiglia di metriche significative e ricavarne degli indicatori che suggeriscano una strategia di miglioramento. Gli indicatori di processo forniscono informazioni sull'efficacia di un processo esistente. Sulla base di tali indicatori, manager e tecnici sono in grado di stabilire che cosa funziona a dovere e cosa no. La raccolta delle metriche di processo abbraccia tutti i progetti in un vasto arco di tempo. Il loro obbiettivo è quello di fornire indicatori che conducano a miglioramenti a lungo termine del processo software.

L'efficacia di un processo software si misura indirettamente. A partire dalle informazioni derivate dal processo si ricava una famiglia di metriche. Tali informazioni comprendono il numero di errori scoperti prima della distribuzione del software, i difetti comunicati dagli utenti finali, i prodotti distribuiti, il lavoro speso, il tempo impiegato, il grado di conformità alle scadenze prefissate ed altre misure. Le metriche di processo si ricavano anche dalla misura delle caratteristiche di specifiche operazioni ingegneristiche. Si possono ad esempio misurare il lavoro ed il tempo spesi nelle attività ausiliarie e nelle altre attività generiche.

Le metriche di processo possono rivelarsi significativamente benefiche per un'azienda impegnata ad accrescere il proprio livello di maturità di processo. Tuttavia, come tutte le metriche, anche queste sono soggette ad abusi e per questo possono creare più problemi di quanti ne risolvano. Un “galateo delle metriche software”, utile per i manager e coloro che devono applicare le metriche di processo, può essere il seguente:

- Nell'interpretare le metriche utilizzare il buon senso e la sensibilità organizzativa.
- Comunicare regolarmente i risultati agli individui ed ai team che hanno raccolto le misure e le metriche.
- Non servirsi delle metriche per valutare i singoli.
- Collaborare con i tecnici ed i team per impostare obiettivi chiari e le metriche che aiutano a raggiungerli.
- Non utilizzare mai le metriche come arma nei confronti dei singoli e dei team.
- Non cosiderare negativi i dati che segnalano aree problematiche. Quei dati sono solo un indicatore utile a migliorare il processo.
- Non dare troppa importanza ad una metrica specifica, a danno di altre, pure importanti.

Un indicatore di processo utile è l'analisi dei guasti, la quale si svolge nel seguente modo:

1. Gli errori ed i difetti sono classificati secondo l'origine (ad esempio nella specifica, nella logica, nella mancanza di conformità agli standard e così via)
2. Si prende nota del costo di correzione di ogni errore o difetto.
3. Si contano errori e difetti di ogni categoria e si dispongono le categorie in ordine decrescente.
4. Si calcola il costo complessivo degli errori e dei difetti di ogni categoria.
5. Si analizzano i risultati per determinare le categorie che comportano i costi più alti.
6. Si predispone un piano di modifiche al processo tese ad eliminare la classe di errori e difetti più costosa (od a ridurne la frequenza).

Una delle cose che può aiutare nell'analisi dei guasti è costruire un grafico che riporti la distribuzione dei difetti, dividendoli in cause ed origini.

Metriche di progetto

Le metriche di progetto si utilizzano per scopi strategici; le misure di progetto servono fini tattici. In altre parole, le metriche di progetto e gli indicatori derivati sono sfruttati dal capo progetto e dal team per adattare la progressione del progetto e le attività tecniche. Gli indicatori di progetto permettono al capo progetto di stimare lo stato di avanzamento di un progetto in corso, seguire i rischi potenziali, scoprire aree problematiche prima che la situazione diventi critica, regolare il flusso del lavoro e delle operazioni e valutare la capacità del team di controllare la qualità dei prodotti.

In genere la prima applicazione delle metriche di progetto ha luogo nella fase di stima. Le metriche raccolte dai processi passati formano una base dalla quale ricavare stime dell'impegno e del tempo necessari per il progetto in corso. Nel corso del progetto, le misure di impegno e tempo spesi vengono confrontati con le stime (e con la tabella dei tempi del progetto). Munito di questi dati, il capo progetto sorveglia e pilota l'avanzamento del progetto.

Lo scopo delle metriche di progetto è duplice: anzitutto ridurre i tempi di sviluppo, suggerendo gli aggiustamenti utili ad evitare ritardi, a mitigare problemi e rischi potenziali; in secondo luogo, stimare via via la qualità del prodotto e, se necessario, correggere l'approccio tecnico per migliorarla.

Metriche dimensionali

Le metriche dimensionali si ottengono normalizzando le misure di qualità e produttività sulla base delle dimensioni del software prodotto. Esse si basano sul numero di linee di codice prodotto e sono:

- errori per KLOC (migliaia di linee di codice)
- difetti per KLOC
- costo per riga di codice
- pagina di documentazione per KLOC
- errori/mese-uomo
- LOC per mese-uomo
- costo per pagina di documentazione

Le metriche dimensionali non sono da tutti accettate in quanto esse non possono misurare:

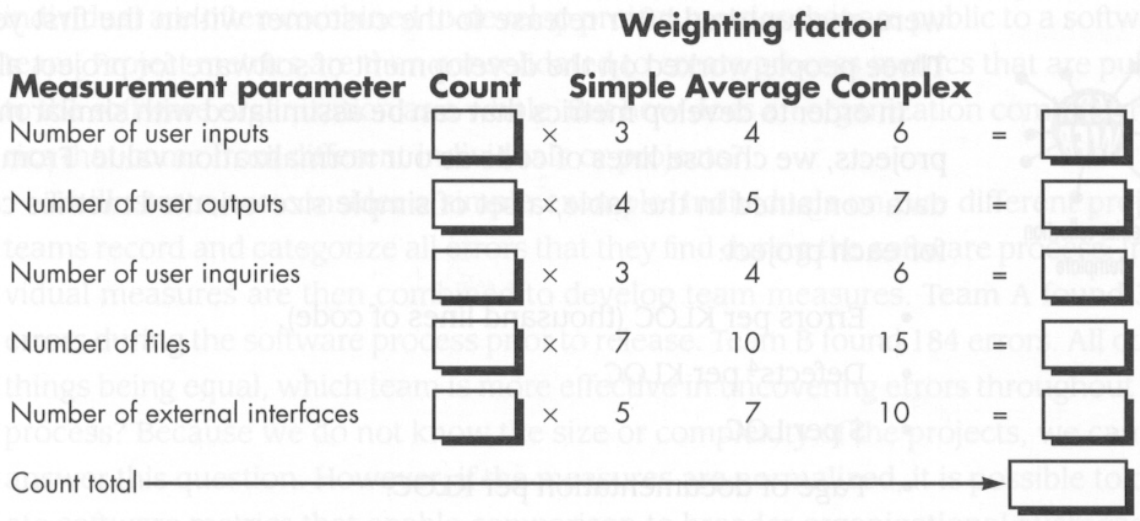
- Fattori umani (dimensione del gruppo, competenze)
- Fattori problematici (complessità, cambiamento)
- Fattori di processo (tecniche, strumenti)
- Fattori di prodotto (affidabilità, prestazioni)
- Fattori di risorse (persone, strumenti)

inoltre tutte le misure basate sulle LOC dipendono pesantemente dal linguaggio di programmazione adottato e che il loro impiego nelle stime esige un livello di dettaglio non sempre facile da raggiungere (ad esempio, il pianificatore deve stimare il numero di LOC ben prima delle fasi di analisi e progettazione).

Metriche basate sulle funzioni

Le metriche basate sulle funzioni adottano una misura della funzionalità fornita dall'applicazione. Tale misura è denominata *function point* (o *punti funzione*).

Per calcolare i punti funzione si riempie la seguente tabella:



Measurement parameter	Count		Weighting factor: Simple	Average	Complex	
Number of user inputs		×	3	4	6	=
Number of user outputs		×	4	5	7	=
Number of user inquiries		×	3	4	6	=
Number of files		×	7	10	15	=
Number of external interfaces		×	5	7	10	=
Count total						

I parametri di misurazione sono i seguenti:

Numero di input utente: si conta ogni input dell'utente che fornisce al software dati applicativi distinti. Occorre distinguere gli input dalle interrogazioni, che devono essere conteggiate a parte.

Numero di output utente: si conta ogni output destinato all'utente che fornisce dati applicativi. Per output si intendono resoconti, schermate, messaggi d'errore e così via. I singoli dati contenuti in un resoconto non sono conteggiati separatamente.

Numero di interrogazioni utente: un'interrogazione è definita come un input in linea che suscita la generazione di una risposta immediata del software, nella forma di output in linea. Si conta ogni interrogazione distinta.

Numero di file: Si conta ogni file logico principale (cioè un raggruppamento logico di dati che possa far parte di un database oppure un file separato).

Numero di interfacce esterne: Si contano tutte le interfacce leggibili dalla macchina (ad esempio file di dati su supporti di memorizzazione), utilizzate per trasmettere informazioni ad un altro sistema.

Raccolti questi dati, si associa ad ogni conteggio un indice di complessità. Le aziende che si servono dei punti funzione sviluppano criteri per determinare se una certa voce è semplice, media o complessa. In ogni caso la determinazione della complessità resta in parte soggettiva.

Per calcolare i punti funzione (FP), si usa la relazione seguente:

$$FP = totale \times [0{,}65 + 0{,}01 \times \sum F_i]$$

"totale" è la somma delle voci ricavate dalla tabella soprastante.

Gli F_i (i=1;...;14) sono valori di aggiustamento della complessità, basati sulle risposte alle seguenti domande:

1. Il sistema richiede salvataggi e recuperi affidabili?
2. Si richiedono comunicazioni di dati?
3. Ci sono funzioni di elaborazione distribuita?
4. Le prestazioni sono un fattore critico?
5. Il sistema funzionerà in un ambiente operativo esistente, utilizzato a fondo?

6. Il sistema richiede inserimenti di dati in linea?
7. L'inserimento di dati in linea richiede che la relativa transazione si articoli in più schermi o più operazioni?
8. I file principali sono aggiornati in linea?
9. I dati d'ingresso o di output, i file e le interrogazioni sono complessi?
10. L'elaborazione interna è complessa?
11. Il codice è stato progettato per essere riutilizzabile?
12. La progettazione richiede anche conversione ed installazione?
13. Il sistema è stato progettato per essere installato in più organizzazioni?
14. L'applicazione è stata progettata per facilitare le modifiche e per essere di semplice uso da parte dell'utente finale?

La risposta ad ognuna di queste domande può andare da 0 (non importante o non applicabile) a 5 (assolutamente essenziale).

Le costanti ed i pesi visti precedentemente sono determinati empiricamente.

I punti funzione così calcolati si utilizzano per delle misure di produttività e qualità:

- errori per FP
- difetti per FP
- costi per FP
- pagine di documentazione per FP
- FP per mese-uomo

I function point permettono di confrontare lo sforzo stimato su due sistemi diversi, o per progetti nel tempo. Tuttavia i punti funzione sono una misura **astratta** e **relativa** (non concreta o assoluta!) e devono essere adattati per ogni organizzazione o dominio.

Feature Points

I feature points sono un adattamento dei function points, a cui viene aggiunta un altro parametro di misurazione: gli algoritmi!

Un **Algoritmo** è definito come un problema computazionale delimitato, incluso per un determinato programma per computer. L'inversione di una matrice, la decifrazione di una sequenza di bit ed il trattamento di un interruzione sono tutti esempi di algoritmi.

Object Points

Gli object points sono una metrica basata sulle funzioni, ma, a differenza dei function points, essi si incentrano sui tool di ausilio che permettono di creare velocemente schermate e report. I parametri di configurazione che prende in esame sono i seguenti:

- **Schermate**: Tutto ciò che deve essere visualizzato sullo schermo
- **Report**: I vari report che devono essere generati
- **Componenti 3GL**: Tutto ciò che deve essere costruito “a mano”

Ad ognuno di questi parametri viene associato un peso:

Object Type	*Complexity-Weight*		
	Simple	***Medium***	***Difficult***
Screen	1	2	3
Report	2	5	8
3GL Component			10

I valori vengono poi sommati insieme ed utilizzati per ottenere i NOP, cioè gli object points che tengono conto anche della percentuale di oggetti riutilizzabili

$$NOP = OP \cdot [\frac{(100 - reuse)}{100}]$$

Dove: OP è il valore trovato dalla somma precedente, reuse è la percentuale di riutilizzo.

Trovato il NOP viene calcolato il PROD:

Developers experience and capability ICASE maturity and capability	**Very low**	**Low**	**Nominal**	**High**	**Very high**
PROD	4	7	13	25	50

Attraverso il NOP ed il PROD si può calcolare lo sforzo richiesto in mesi-uomo:

$$sforzo = \frac{NOP}{\text{PROD}}$$

Anche qui, come per i function points, i valori numerici sono dati attraverso dati empirici.

Modello COCOMO

Il modello COCOMO è un modello che cerca di aiutare il capo-progetto a:

a) calcolare lo sforzo in mesi-uomo previsto per un progetto, nonché la sua durata

b) seguire il progetto durante tutte le sue fasi

La formula principale è quella del calcolo per lo sforzo in mesi-uomo:

$$sforzo = A \times \text{size}^{B}$$

Dove:

- ➔ A è una costante moltiplicativa usata per calcolare gli effetti moltiplicativi quando aumenta la grandezza del progetto
- ➔ size è la grandezza del progetto in migliaia di linee di codice
- ➔ B è il fattore di scala che indica le economie e le diseconomie in base alla grandezza del progetto.

Size può essere calcolata anche attraverso i punti funzione tramite la seguente tabella:

Language	SLOC / UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic - Compiled	91
Basic - Interpreted	128
C	128
C++	29
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

Tali trasformazioni sono state realizzate attraverso diversi studi (tutt'ora in corso).

A è una costante empirica, mentre B è calcolata con la formula:

$$B = 0{,}91 + 0{,}01 \times \sum W_i$$

Dove i W_i sono dati da 5 fattori:

- Lavori precedenti
- Flessibilità di sviluppo
- Risoluzioni di Architettura/rischi
- Coesione del Team
- Maturità del Processo

Ognuno di questi fattori è pesato in base al loro grado di scala (Molto basso, basso, medio, alto, molto alto, estremamente alto). Stabilire qual'è il loro grado di scala lo si fa rispondendo a delle domande, mentre a che valore corrisponde il grado di scala è un calcolo empirico.

Calcolato lo sforzo esso viene aggiustato in base ai *Cost Driver* che indicano quanto incidono le varie parti del progetto sullo sforzo. Questi Cost Driver sono 21 durante le fasi del progetto, ma di essi

si usano solo 7 (in realtà alcuni di questi vengono combinati) quando si ci trova all'inizio del progetto e bisogna stimare i costi.

I *Cost Driver* sono i seguenti:

- **ACAP** Analyst Capability
- **AEXP** Applications Experience
- **CPLX** Product Complexity
- **DATA** Database Size
- **DOCU** Documentation to match lifecycle needs
- **FCIL** Facilities
- **LTEX** Language and Tool Experience
- **PCAP** Programmer Capability
- **PCON** Personnel Continuity
- **PDIF** Platform Difficulty
- **PERS** Personnel Capability
- **PEXP** Platform Experience
- **PREX** Personnel Experience
- **PVOL** Platform Volatility
- **RCPX** Product Reliability and Complexity
- **RELY** Required Software Reliability
- **RUSE** Required Reusability
- **SCED** Required Development Schedule
- **STOR** Main Storage Constraint
- **TIME** Execution Time Constraint
- **TOOL** Use of Software Tools

La seguente tabella indica quali sono i *Cost Driver* usati durante le stime iniziali:

Early Design Cost Driver	Counterpart Combined Post-Arch. Cost Driver
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

Avuti i Cost Driver, si calcola lo sforzo modificato:

$$sforzoModif = sforzo \times (\prod_i EM_i)$$

Dove EM_i sono i Cost Driver calcolati.

Ottenuto lo sforzo così modificato, lo si può usare per calcolare il tempo:

$$TDEV = [3 \times sforzoModif^{(0,33+0,2\times(B-0,01))}] \times (\frac{SCEDPercentage}{100})$$

Dove TDEV è il tempo, mentre SCEDPercentage è la percentuale del Cost Driver SCED.

Esistono poi delle modifiche al modello per prendere in cosiderazione anche il riuso, la conversione e la reingegnerizzazione ed il mantenimento.

Metriche per la qualità

Lo scopo primario dell'ingegneria del software è quello di produrre sistemi, applicazioni o prodotti di alta qualità. Per raggiungerlo, gli ingegneri del software devono applicare metodi efficaci abbinati a strumenti moderni, nel contesto di un processo software maturo. Un buon ingegnere del software (ed un buon manager del settore software) deve inoltre misurare se l'alta qualità sarà ottenuta.

Le misure che si possono prendere in questo frangente possono essere:

- **Correttezza**: un programma che non operi correttamente è di ben poco valore per i suoi utenti. La correttezza è il grado in cui il software svolge la funzione richiesta. La misura di correttezza più comune è il numero di difetti per KLOC, intendendo per difetto un'accertata mancanza di conformità ai requisiti. Quando si considera la qualità globale del prodotto software, i difetti sono quei problemi evidenziati da un utente del programma dopo che il programma stesso è stato rilasciato per l'utilizzo generale. Per gli scopi di definizione della qualità, i difetti vengono contati dopo un periodo di tempo standard, in genere un anno.
- **Facilità di manutenzione**: la manutenzione è l'attività più costosa in termini di lavoro fra quelle del software; è l'indice della facilità con cui un programma può essere corretto nel caso di rilevi un errore, adattato nel caso in cui muti il suo ambiente o ritoccato nel caso in cui un utente desideri una modifica dei requisiti. Non c'è modo di misurare direttamente la facilità di manutenzione, di conseguenza si deve ricorrere a misure indirette. Una semplice metrica orientata al tempo è il "tempo medio per la modifica" (MTTC, Mean-Time-To-Change), cioè il tempo necessario ad

analizzare la richiesta di cambiamento, progettare, implementare, collaudare e distribuire agli utenti l'opportuna modifica. In media i programmi di facile manutenzione esibiscono un valore inferiore di MTTC (per modifiche affini) rispetto a quelli di difficile manutenzione.

- **Integrità**: questo attributo misura la capacità di un sistema di resistere ad attacchi (sia accidentali, sia intenzionali) alla sua sicurezza. Gli attacchi possono essere rivolti a tre componenti del software: programmi, dati e documenti. Per misurare l'integrità, si devono defininire due altri attributi: la minaccia e la sicurezza. La minaccia è la probabilità (stimata o ricavata da dati empirici) che un attacco di un certo tipo avvenga in un certo dato spazio di tempo. La sicurezza è la probabilità (parimenti stimata o ricavata da dati empirici) che l'attacco sia respinto. L'integrità di un sistema si può allora definire così:
  $integrità = \sum [1 - minaccia \times (1 - sicurezza)]$
  dove la somma si estende a tutti i tipi di attacchi
- **Usabilità**: stabilire se un prodotto è facile da usare è tra le cose più difficili da misurare; normalmente si cerca di misurare quattro caratteristiche: le competenze fische ed intellettive necessarie per apprendere ed utilizzare il sistema; il tempo necessario ad acquisire una moderata disinvoltura nell'uso del sistema; l'incremento di produttività (rispetto alla soluzione sostituita dal sistema), misurato quando il sistema è utilizzato in modo moderatamente efficiente; una valutazione soggettiva (talvolta ricavata da un questionario) delle reazioni degli utenti.

Efficienza di rimozione dei difetti

Una metrica della qualità utile sia a livello di processo, sia di progetto è l'efficienza di rimozione dei difetti (DRE, Defect Removal Efficiency). In sostanza la DRE è una misura del potere filtrante delle attività di assicurazione e controllo di qualità applicate nel corso di tutte le attività portanti del processo.

Rispetto al processo considerato nella sua interezza, la DRE è così definita:

DRE=E/(E+D)

dove E è il numero di errori scoperti prima della consegna del software all'utente finale e D è il numero di difetti scoperti dopo la consegna.

Il valore ideale di DRE è 1, ottenuto quando il software non presenta difetti. In un caso reale il valore di D è maggiore di zero, ma quello di DRE può tendere ad 1.

All'interno del singolo progetto, la DRE può servire a valutare la capacità di scoprire errori da parte di un team prima di passare alla successiva attività portante od al compito successivo. Per fare un esempio, l'analisi dei requisiti produce un modello suscettibile di revisione per la ricerca e correzione degli errori. Gli errori non rilevati passano all'operazione di progettazione (nella quale possono essere scoperti o no). In tale contesto la DRE è ridefinita come segue:

$DRE_i = E_i / (E_i + E_{i+1})$

dove E_i è il numero di errori scoperti nel corso dell'attività i, E_{i+1} è il numero di errori scoperti nel corso dell'attività i+1 riconducibili ad errori non rilevati nell'attività i

Dal punto di vista della qualità, l'obbiettivo di un team (o del singolo ingegnere del software) è quello di avvicinarsi quanto possibile al valore 1 per DRE_i. In altre parole, gli errori devono essere filtrati prima di passare all'attività successiva.

Integrazione delle metriche nel processo software

Una delle cose più difficili delle metriche è introdurle all'interno di un processo software. Come abbiamo detto ad inizio capitolo le metriche servono a caratterizzare, valutare, prevedere e migliorare. Ma i problemi quotidiani di un progetto lasciano poco spazio a disposizione per le questioni strategiche. I capi progetto devono preoccuparsi di questioni più immediate: produrre stime significative sul progetto, produrre sistemi di alta qualità, rispettare le scadenze.

Però, quando il progetto è completato, allo sviluppatore piacerebbe poter rispondere ad alcune domande: Quali requisiti hanno più possibilità di mutare? Quali componenti del sistema sono maggiormente soggetti ad errori? Quanti collaudi si devono pianificare per ogni modulo? Quanti errori di ciascun tipo è ragionevole attendersi dai collaudi?

Si può dare una risposta a queste domande, a patto di aver raccolto le metriche opportune e di averle strutturate come guida tecnica.

Occorre dunque stabilire una piattaforma per le valutazioni metriche, tenendo presente che le informazioni raccolte non devono essere necessariamente radicalmente differenti, ma le stesse valutazioni metriche possono servire per vari scopi. Inoltre affinché sia un efficace ausilio per il miglioramento dei processi e/o per la stima dei costi e degli sforzi, i dati della piattaforma devono avere i seguenti attributi: i dati devono essere ragionevolmente precisi e devono essere evitate stime vaghe sui progetti passati; i dati devono essere raccolti per il maggior numero possibile di progetti; le misure devono essere uniformi, ad esempio il concetto di linee di codice deve essere interpretato allo stesso modo in tutti i progetti per i quali sono stati raccolti i dati; le applicazioni devono essere simili al lavoro che si deve stimare (ha poco senso utilizzare la piattaforma di un sistema di elaborazione batch per stimare un'applicazione embedded in tempo reale).

I dati raccolti vengono tanto dal processo, quanto dal progetto, quanto dal prodotto; l'insieme di questi dati ci fornisce le misure, dalle quali vengono elaborate le metriche, dalle quali vengono estratti gli indicatori. Idealmente i dati necessari dovrebbero essere raccolti via via; purtroppo questa situazione è rara, perciò la raccolta dei dati comporta un'indagine storica dei progetti passati e la ricostruzione dei dati rilevanti.

Introdurre le metriche

Quando si decide di iniziare le misurazioni è facile perdersi tra la mole delle cose misurabili. La regola numero uno è semplice: mantere le cose semplici. È inutile avere a che fare con moli di dati che non si sa o non si ha tempo di elaborare. Per inserire le metriche in un processo di sviluppo, bisogna:

1) Indentificare gli obbiettivi
2) Indentificare cosa vogliamo conoscere
3) Indentificare le entità e gli attributi necessari (che cosa misurare)
4) Formalizzare gli obbiettivi delle misurazioni (come misurare)
5) Stabilire come elaborare le misure prese (cioè scegliere la metrica da seguire)
6) Stabilire gli indicatori che si vogliono prendere in considerazione

Ad esempio poniamoci un obbiettivo: ridurre il tempo di valutazione ed implementazione delle richieste di modifica. Cosa vogliamo conoscere: il tempo trascorso dalla richiesta al termine della valutazione, l'impegno per svolgere la valutazione, il tempo trascorso dalla valutazione all'assegnazione dell'ordine di modifica al personale, l'impegno necessario ad apportare la modifica, il tempo

necessario per apportare la modifica, gli errori scoperti durante la modifica ed i difetti scoperti dopo che la modifica è stata rilasciata al cliente. Quali sono le entità e gli attributi necessari e come formalizziamo il tutto? T_{queue} è il tempo trascorso (in ore o giorni) dalla richiesta al termine della valutazione, W_{eval} è l'impegno (in persone-ora) per svolgere la valutazione, T_{eval} è il tempo trascorso (in ore o giorni) dalla valutazione all'assegnamento dell'ordine di modifica al personale, W_{change} è l'impegno (in persone-ora) necessario ad apportare la modifica, T_{change} è il tempo necessario (in ore o giorni) per apportare la modifica, E_{change} è il numero di errori scoperti durante la modifica, D_{change} è il numero di difetti scoperti dopo che la modifica è stata rilasciata al cliente. Dopo aver raccolto queste misure per una serie di modifiche, è possibile calcolare il tempo totale trascorso dalla richiesta di modifica all'implementazione della stessa e la percentuale del tempo assorbito per partire, per la valutazione, l'assegnamento e l'implementazione della modifica. Analogamente si può determinare le percentuali dell'impegno richiesto per la valutazione e l'implementazione. Queste valutazioni metriche possono essere determinate nel contesto dei dati di qualità, E_{change} e D_{change}. Le percentuali forniscono una conoscenza del punto in cui il processo di richiesta della modifica rallenta e ciò può portare a contromisure per il miglioramento del processo per ridurre T_{queue}, W_{eval}, T_{eval}, W_{change} e/o E_{change}. Inoltre si può calcolare l'efficienza della rimozione dei difetti che può essere confrontata col tempo trascorso e con l'impegno totale necessario per determinare l'impatto delle attività di controllo della qualità sul tempo e l'impegno necessario per apportare una modifica.

Garanzia di qualità del software

La SQA – software quality assurance – è un'attività che copre tutto il ciclo di vita del software, e si compone di un'attività di gestione della qualità, una tecnologia di ingegneria del software, revisioni tecniche formali svolte durante il processo, una strategia di collaudi su più livelli, una gestione della documentazione e delle modifiche, una procedura che garantisca la conformità allo standard dello sviluppo, e infine meccanismi di misurazione e stesura dei resoconti.

Cos'è la qualità?

Il controllo della qualità ha come elemento cardine il controllo delle variazioni da un prodotto all'altro: l'obiettivo di una produzione di qualità è di minimizzare le differenze tra i prodotti. Minimizzare le variazioni può significare molte cose: minimizzare la differenza tra le risorse previste e quelle effettivamente impiegate, il numero di difetti del prodotto distribuito, la variabilità dei difetti da una versione all'altra dello stesso prodotto.

La **qualità** di un oggetto è una caratteristica che si basa su proprietà misurabili del prodotto, cioè su quantità confrontabili con degli standard prefissati; nel caso del software, però, queste proprietà "misurabili" sono più difficili da quantificare rispetto agli oggetti fisici. Tuttavia, anche per il software sono state standardizzate delle metriche che riguardano la complessità ciclomatica, la coesione, il numero di function-points, il numero di righe di codice.

Nella valutazione di un prodotto si possono individuare due tipi di qualità:

- La **qualità della progettazione**, relativa all'analisi delle caratteristiche specificate dal progettista (materiali, tolleranze, prestazioni richieste) – è un parametro che riguarda i requisiti, le specifiche e la progettazione del sistema.
- La **qualità della conformità** rappresenta il grado di aderenza dell'implementazione rispetto a quanto specificato nella progettazione – è una caratteristica dell'implementazione.

Definiamo **controllo di qualità** l'insieme degli esami, delle revisioni e dei collaudi svolti nel corso del ciclo di vita del software, il cui scopo è di garantire che il prodotto soddisfi i requisiti imposti; è importante che per ogni lavorato siano definite specifiche misurabili, da usare come metro di misura per valutare gli oggetti prodotti.

La **garanzia di qualità** consiste di funzioni di auditing e di stesura dei resoconti del management; ha lo scopo di fornire al management le informazioni necessarie a garantire che il prodotto soddisfi gli obiettivi qualitativi.

Il **costo della qualità** è la somma di tutti i costi sostenuti per l'attività di SQA e attività correlate. Questi costi si possono classificare come:

- *Costi di prevenzione* – comprendono i costi per la pianificazione della qualità, per le revisioni tecniche formali, per le attrezzature necessarie per i collaudi, e per l'addestramento del personale addetto alla SQA.
- *Costi di stima* – sono sostenuti per le attività svolte per valutare la qualità di un prodotto nella prima fase di ogni processo.
- *Costi dei difetti* – questi costi scomparirebbero se il prodotto non rivelasse nessun difetto prima della consegna al cliente; si dividono in costi interni, che si verificano quando un errore viene rilevato prima della consegna (rifacimenti, riparazioni, analisi dei guasti) ed esterni; questi ultimi sono costi legati ai difetti scoperti dopo il rilascio del software al cliente (risposte ai reclami, restituzione e sostituzione del prodotto, assistenza tecnica, obblighi di garanzia).

La garanzia di qualità del software

Diamo una definizione più formale del concetto di qualità del software: la qualità è *"Il rispetto dei requisiti funzionali e prestazionali enunciati esplicitamente, la conformità a standard di sviluppo esplicitamente documentati e le caratteristiche implicite che ci si aspetta da un prodotto software realizzato professionalmente"*

Da questa definizione emergono tre punti fondamentali per lo svolgimento dell'attività di SQA:

1. i requisiti sono alla base delle misurazioni della qualità; la non conformità ai requisiti implica mancanza di qualità;
2. gli standard specificati definiscono i criteri da seguire durante lo sviluppo del software,
3. anche i requisiti impliciti devono essere tenuti in cosiderazione; un software che rispetta i requisiti espliciti ma non quelli impliciti è spesso un software di scarsa qualità.

È importante ricordare che il gruppo che si occupa dell'attività di SQA svolge il ruolo di rappresentante del cliente, e quindi deve osservare il software dal punto di vista del cliente.

La garanzia di qualità del software comprende un insieme di operazioni affidate a due distinti gruppi di lavoro: un gruppo tecnico, ed un gruppo responsabile della SQA vera e propria; quest'ultimo gruppo è responsabile della pianificazione, della sorveglianza, della registrazione, dell'analisi e della documentazione della garanzia di qualità.

Più in dettaglio, le attività svolte da un gruppo di SQA possono essere così descritte:

- *preparazione di un piano SQA per un progetto* – il piano deve specificare:
  - valutazioni da compiere
  - esami e revisioni da compiere
  - standard applicabili al progetto
  - procedure di stesura di resoconti e ricerca degli errori
  - documenti da produrre
  - grado di risposta fornita al team del progetto
- *partecipazione alla stesura della descrizione del processo software del processo*
- *revisione delle attività di sviluppo, in modo da verificarne la conformità al processo fissato*
- *esame dei lavorati software previsti, per verificarne la conformità a quelli definiti nel processo*
- *certificazione che le deviazioni nel lavoro e nei prodotti siano documentate e gestite secondo le procedure previste*
- *registrazione delle violazioni e resoconto alla direzione aziendale*

Le revisioni del software

Le revisioni dovrebbero essere effettuate in varie tappe durante il processo di sviluppo, esse hanno il compito di scoprire gli errori. Esistono diversi tipi di revisioni: una riunione informale intorno alla macchina da caffè è una forma di revisione, se si discute di questioni tecniche; una presentazione formale del progetto del software ad un pubblico di clienti, dirigenti e tecnici è una forma di revisione; noi ci occupiamo in particolare dei *walkthrough* – **revisioni tecniche formali** – che sono revisioni condotte da tecnici per altri tecnici.

Le revisioni sono una sorta di filtro per il processo di sviluppo. Come per i filtri dell'acqua, anche i filtri per il software tendono a rallentare il "flusso" delle attività di ingegneria del software. Se sono troppo pochi il flusso sarà "sporco". Se sono troppi il flusso sarà troppo lento.

Una revisione – qualsiasi revisione – è un modo di sfruttare le differenze tra le persone per:

1. sottolineare i miglioramenti necessari ai risultati di ogni persona e team;
2. ratificare quelle parti di un prodotto per le quali non è necessaria né desiderata alcuna miglioria;
3. conferire al lavoro tecnico una qualità più uniforme, od almeno più prevedibile che non senza revisioni, al fine di renderlo meglio governabile.

L'obbiettivo principale delle revisioni è quello di scoprire gli errori durante il processo di sviluppo, per impedire che diventino difetti dopo la consegna del software. L'ovvio beneficio è che gli errori non si propagano alle fasi successive del processo. Già, perché ogni errore che si propaga ha la disgraziata conseguenza di ampliarsi di un certo fattore, rendendolo così evidente, ma praticamente "incurabile".

Le revisioni tecniche formali

Gli scopi di una Revisione Tecnica Formale o FTR – Formal Technical Review – sono:

1. scoprire errori nelle funzioni e nella logica del software
2. verificare che il software soddisfi i requisiti
3. garantire che il software sia rappresentato secondo gli standard predefiniti
4. accertare che il software sia sviluppato in modo uniforme
5. rendere più governabili i progetti

Le FTR permettono inoltre di addestrare i tecnici meno esperti, che durante le revisioni hanno occasione di confrontarsi con diversi approcci all'analisi, alla progettazione e all'implementazione del software, e di garantire la sostituibilità e la continuità all'interno del team di lavoro, in quanto i tecnici vengono a conoscenza anche delle parti del software a cui non hanno lavorato.

Walkthrough, ispezioni, revisioni incrociate sono tutti esempi di FTR; ogni revisione si svolge sotto forma di riunione.

La **riunione** per una FTR dovrebbe rispettare alcune caratteristiche:

- deve coinvolgere da 3 a 5 persone
- ogni partecipante deve svolgere un lavoro di preparazione alla riunione che non duri più di un paio d'ore
- la riunione non deve durare più di due ore

Una FTR deve concentrarsi su piccole porzioni del software; normalmente l'oggetto di una FTR è un lavorato (ad esempio una parte della specifica dei requisiti, il progetto dettagliato di un modulo, il codice sorgente di un modulo). Chi ha sviluppato il lavorato informa al momento giusto il capo-progetto che il lavorato è finito e necessita di una revisione. Il capo progetto nomina un revisore capo, che si accerta che il prodotto sia pronto per la revisione e ne distribuisce la documentazione ai partecipanti alla revisione; ogni partecipante dedica non più di due ore all'esame della documentazione familiarizzando con il prodotto da revisionare e prendento appunti su eventuali osservazioni. Alla riunione partecipano lo sviluppatore del lavorato, il revisore capo e gli altri revisori; uno dei partecipanti funge da segretario della riunione, incaricato di redigere una sorta di verbale della riu-

nione, incentrato sulle questioni importanti emerse via via. L'FTR si apre con la presentazione dell'agenda e con una breve introduzione dello sviluppatore. Lo stesso sviluppatore prosegue "percorrendo" il lavorato e dando spiegazioni, mentre i revisori possono intervenire sulla base della preparazione che hanno svolto. A mano a mano che emergono errori o problemi, il segretario li registra.

Al termine della riunione i revisori decidono se accettare il lavoro così com'è, respingerlo a causa di errori gravi oppure accettarlo con riserva, perché ci sono errori non gravi da correggere (ma non occorre una nuova revisione).

Ogni revisione deve essere documentata da una relazione che dica qual è stato l'oggetto della revisione, da chi è stata compiuta, cosa si è scoperto e a quali conclusioni si è giunti. E' importante disciplinare anche come debbano essere risolte dopo la revisione le questioni in essa sollevate.

La relazione consiste in genere in un modulo di una pagina, eventualmente corredato di allegati. Essa viene a far parte della documentazione del progetto e può essere distribuita al capo progetto ed a tutti gli interessati.

L'elenco delle questioni serve a due scopi: individuare aree problematiche del prodotto e guidare operativamente lo sviluppatore nella fase di correzione. L'elenco delle questioni è di norma allegato alla relazione.

Linee guida per le revisioni

Elenchiamo alcune **linee guida** per le revisioni:

- *rivedere il prodotto, non chi l'ha sviluppato*: una revisione tecnica formale coinvolge le persone ed il loro ego. Se ben condotta, dovrebbe lasciare ai partecipanti una sensazione di soddisfazione; diversamente, può creare un'atmosfera d'inquisizione. Gli errori devono essere evidenziati con gentilezza, il tono della riunione deve essere informale ma costruttivo, la riunione non deve provocare imbarazzo o disprezzo. Il revisore capo deve condurre la riunione al fine di assicurare il mantenimento di tono e modalità appropriati e deve immediatamente arrestare una revisione di cui ha perso il controllo.
- *fissare un'agenda e rispettarla*: gli orari sono un grande problema di tutte le riunioni. Una revisione tecnica formale deve seguire la traccia prefissata e gli orari stabiliti. Il revisore capo ha la responsabilità di far rispettare gli orari della riunione e di riprendere le persone che causino ritardi ingiustificati.
- *limitare i dibattiti*: qualora il revisore faccia emergere un problema, potrebbe verificarsi un disaccordo nel merito. Invece di spendere tempo discutendo la questione, il problema dovrebbe essere annotato e ripreso per ulteriori dibattiti in altre sedi.
- *rilevare gli errori, non cercare di risolverli tutti nel corso della riunione*: una revisione non è una sessione di lavoro dedicata alla soluzione dei problemi; spesso la soluzione può essere affidata allo stesso produttore, da solo a con l'aiuto di altri. La soluzione dei problemi dovrebbe essere rimandata a dopo la riunione di revisione.
- *prendere appunti*: talvolta è una buona idea che la persona preposta alla stesura del verbale prenda appunti su una lavagna, in modo che gli altri revisori possano controllare i termini e le priorità delle informazioni che vengono registrate.
- *prevedere un numero limitato di partecipanti, che giungano alla riunione preparati*: due teste ragionano meglio di una, ma quattordici teste non sono necessariamente meglio di quattro. È importante cercare di limitare al minimo possibile il numero di persone coinvolte. Peraltro, tutti i membri del gruppo di revisione devono prepararsi prima dell'inizio della riunione. Il revisore

capo dovrebbe invitare a stendere commenti scritti (che testimoniano il fatto che il revisore ha visto il materiale).

- ✔ *allocare risorse e prevedere tempi per le TFR*: Al fine di rendere le riunioni efficaci, esse dovrebbero essere pianificate come un compito da svolgersi durante il processo software. Inoltre devono essere stabiliti i tempi di rilascio delle modifiche che sono il risultato inevitabile di una revisione tecnica formale.
- ✔ *istruire i revisori*: Per apportare un contributo significativo alle riunioni, tutti i partecipanti dovrebbero ricevere un'adeguata istruzione. Essa dovrebbe comprendere sia gli aspetti relativi al processo, sia gli effetti psicologici della revisione.
- ✔ *rivedere le revisioni passate*: l'esame a posteriori può essere utile per rilevare problemi connessi al processo stesso di revisione. Il primo prodotto da rivedere potrebbero essere le stesse linee guida per la revisione.

Tecniche per la revisione

- ➢ Walkthrough strutturato – è una tecnica informale che coinvolge l'autore del prodotto sottoposto a revisione e uno o più sviluppatori che conoscono il processo di sviluppo. La riunione dura dai 30 ai 60 minuti, e non è richiesta una documentazione formale.
- ➢ Lettura del codice – due o più sviluppatori leggono il codice separatamente, poi si riuniscono insieme all'autore per discuterlo. Non c'è un processo definito per la lettura del codice, e non è richiesta documentazione dell'esito della revisione.
- ➢ Ispezione del codice – Fagan ha definito un processo sistematico per la lettura del codice, che comprende pianificazione, preparazione, ispezione, lavorazione e documentazione dei risultati; sono specificati anche i ruoli dei partecipanti. Durante la riunione l'autore, o il lettore del codice, lo parafrasa mentre i partecipanti gli rivolgono domande che possano aiutare a scoprire errori o difetti.
- ➢ Revisioni del software – gli autori del software conducono una riunione informale con i lettori per stabilire il retroscena e distribuire il materiale; i revisori esaminano il materiale fornito singolarmente, usando una lista di punti da esaminare come guida. L'autore raccoglie le liste con i punti esaminati e consolida i risultati, che vengono poi presentati al gruppo di revisione durante una riunione.
- ➢ Revisione attiva della progettazione - ogni revisore riceve una lista di punti da esaminare; diversi revisori ricevono liste diverse, focalizzate su diversi insiemi di elementi. L'autore del lavoro sottoposto a revisione incontra i vari revisori individualmente per esaminare le loro osservazioni.
- ➢ Lettura basata sui difetti – si parte da un modello di possibili difetti nel documento dei requisiti. Per ogni classe di difetti viene sviluppato un insieme di domande per caratterizzare la classe; le domande definiscono anche un insieme di passi (scenari) che dovrebbero essere seguiti durante la lettura. Mentre si legge il documento e si seguono i diversi passi, il lettore cerca di rispondere alle domande presenti nello scenario.
- ➢ Revisioni senza riunione - possono essere altrettanto efficaci quanto le revisioni basate su riunioni; le revisioni individuali permettono di trovare più elementi da risolvere, ma tendono a trovare molti difetti che in seguito non si rivelano tali (*falsi positivi*) ed a duplicare il lavoro.
- ➢ Revisioni tecniche guidate da computer – gli sforzi sono rivolti allo sviluppo di software per il supporto alla distribuzione dei report sui difetti rilevati, e all'organizzazione di riunioni dove i partecipanti non risiedono nello stesso luogo.

Garanzia di qualità su base statistica

Un possibile controllo della qualità è quello di basarsi su dati statistici per riconoscere le aree con maggiori problemi e tentari di risolverli. Questa tecnica guarda a dodici fattori:

- Specifiche incomplete od errate
- Fraitendimento delle comunicazioni col cliente
- Deviazione intenzionale delle specifiche
- Violazione degli standard di programmazione
- Errore nella rappresentazione dei dati
- Incoerenza nell'interfaccia tra componenti
- Errore logico nella progettazione
- Collaudo incompleto od errato
- Documentazione imprecisa od incompleta
- Errore nella traduzione della progettazione in un linguaggio di programmazione
- Interfaccia uomo-macchina ambigua od incoerente
- Altro

Per ognuno di questi si raccolgono dati statistici sul numero totate di errori, su quelli gravi, quelli medi e quelli lievi. Mettendo a confronto i dati risultanti si può decidere qual'è l'area su cui agire per migliorare la qualità.

Oltre a questo si può calcolare l'*indice degli errori*; tale indice indica quanto viene migliorato nel complesso il processo software dedicato alla qualità.

Per calcolare l'indice degli errori (EI) vi è bisogno:

- E_i = numero totale di errori scoperti nell'i-esima fase del processo
- S_i = numero di errori gravi
- M_i = numero di errori di gravità media
- T_i = numero di errori meno importanti
- PS = dimensione del prodotto (in LOC, istruzioni nel progetto, pagine di documentazione) all'i-esima fase
- w_s, w_m, w_t = coefficienti di peso per le tre categorie di errori; i valori suggeriti sono $w_s = 10$, $w_m = 3$, $w_t = 1$

Avendo queste misure si può calcolare un indice di fase, PI_i

$PI_i = w_s(S_i / E_i) + w_m(M_i / E_i) + w_t(T_i / E_i)$

Ora si può calcolare l'indice di errore

$$EI = \sum (i \times PI_i) / PS$$

Il Piano SQA

Il piano SQA fornisce una guida per l'istituzione di un protocollo di garanzia di qualità. Messo a

punto dal gruppo SQA e dal team di progetto, il piano fa da quadro di riferimento per le attività di SQA previste per ogni progetto software.

La sezione *Introduzione* descrive lo scopo e la portata del documento.

Nella sezione *Compiti dell'SQA* indica quali attività del processo software sono coperte dalla garanzia di qualità.

Nella sezione *Documenti e prodotti lavorati* sono elencati tutti i documenti citati nel piano; sono inoltre annotati gli standard applicabili.

La sezione *Management* descrive il posto occupato da SQA nella struttura aziendale, le operazione ed attività di SQA e la loro collocazione entro il processo software, i ruoli organizzativi e le responsabilità relative alla qualità del prodotto.

La sezione *Documentazione* descrive (per via di rimandi) i lavorati prodotti nel corso del processo software, che comprendono:

- documenti del progetto (ad esempio il piano di progetto)
- modelli (ad esempio gli ERD, le gerarchie di classi)
- documenti tecnici (ad esempio specifiche, piani di collaudi);
- documenti per l'utente (ad esempio file di guida)

La stessa sezione definisce inoltre il minimo insieme di semilavorati in grado di garantire un'alta qualità.

La sezione *Standard, pratiche e convenzioni* elenca tutti gli standard e le pratiche applicate nel processo software (ad esempio standard per i documenti e per il codice, indicazioni per le revisioni). Sono inoltre elencate tutte le metriche di progetto, processo e (in certi casi) di prodotto, che devono essere raccolte nel corso del processo.

La sezione *Revisioni ed esami* precisa quali revisioni ed esami devono essere svolti dal team di sviluppo, dal gruppo SQA e dal cliente. Per ogni revisione ed esame deve indicare per sommi capi la tecnica da applicare.

La sezione *Test* rimanda al piano ed alle procedure per i collaudi del software. Essa definisce inoltre i requisiti di registrazione storica.

Nella sezione *Resoconto di problemi ed azione correttiva* si definiscono le procedure di stesura di resoconti, controllo e risoluzione degli errori e dei difetti, oltre ad individuare gli incaricati di tale attività.

Il resto del piano è dedicato agli strumenti ed ai metodi su cui poggiano le attività di SQA, alle procedure di gestione della configurazione software per il governo del cambiamento, alla definizione dei modi di gestione del contratto, ai metodi per assemblare, custodire e mantenere i record, all'addestramento necessario per soddisfare le esigenze del piano, ai metodi per individuare, valutare, sorvegliare e controllare i rischi.

Gestione delle configurazioni Software

Quando si realizza del software, vi sono sempre dei cambiamenti. E poiché tali cambiamenti si verificano, occorre controllarli in modo efficace. La gestione delle configurazioni software è un insieme di attività progettate con lo scopo di controllare i cambiamenti tramite l'identificazione dei prodotti che possono cambiare, la definizione delle relazioni fra di essi, la definizione dei meccanismi per la gestione delle versioni di questi prodotti, il controllo delle modifiche imposte e producendo revisioni e report sulle modifiche eseguite.

La *gestione delle configurazioni software* (**SCM**, Software Configuration Management) è un'attività ausiliaria, che abbraccia tutto il processo software. Un cambiamento può avvenire in qualsiasi momento; le attività di SCM hanno lo scopo di riconoscere il cambiamento, controllarlo, garantire che sia opportunamente implementato e riferire agli interessati l'avvenuto cambiamento.

Ma cos'è una configurazione software? Una configurazione software è un'insieme di programmi (nella forma sorgente ed eseguibile), documenti che descrivono i programmi (sia per i tecnici, sia per gli utenti) e strutture dati (contenute nei programmi od esterne). Ognuno di questi elementi può dover cambiare senza preavviso in qualsiasi momento del ciclo di vita del software.

Qual'è l'origine dei cambiamenti? La risposta a questa domanda è tanto varia quanto la natura dei cambiamenti. Si possono però individuare quattro fonti principali di cambiamento:

- nuove condizioni aziendali o di mercato, che dettano un cambiamento nei requisiti del prodotto o nelle regole aziendali;
- nuove esigenze della clientela, che richiedono una modifica nei dati prodotti da un sistema informativo, nella funzionalità di un prodotto o nel servizio offerto da un sistema basato sui computer;
- la riorganizzazione o la riduzione delle dimensioni dell'azienda, che causa un cambiamento nelle precedenze fra progetti e nella struttura dei team addetti allo sviluppo del software;
- vincoli finanziari o di tempo che costringono a ridefinire il sistema od il prodotto.

La gestione delle configurazioni software è un insieme di attività ideate per gestire le modifiche per tutto il ciclo di vita del software.

La gestione delle configurazioni software si basa sul concetto di **documento acquisito**: Una specifica od un prodotto formalmente revisionato ed accettato, che serve in seguito da fondamento di un ulteriore sviluppo e che può essere modificato solo attraverso procedure formali di controllo del cambiamento.

Per descrivere il concetto si può ricorrere ad un'analogia: Osservando le porte di una cucina di un grande ristorante, si può vedere come esse siano contrassegnate dalle scritte ENTRATA ed USCITA. Le porte hanno dei fermi che consentono la loro apertura solo nella direzione opportuna. Se un cameriere prende un piatto pronto dalla cucina, lo mette su un vassoio e subito si accorge di aver preso il piatto sbagliato, egli può sostituirlo con il piatto giusto in modo veloce ed informale prima di lasciare la cucina. Se, tuttavia, egli esce dalla cucina, consegna il piatto al cliente e solo in questo momento viene informato del suo errore, deve seguire una procedura fissata: verificare sulla nota se c'è stato un errore, scusarsi sentitamente, ritornare in cucina attraverso la porta di ENTRATA, spiegare il problema e così via.

Un documento acquisito è analogo alla porta della cucina del ristorante. Prima che un elemento della configurazione software sia diventato un documento acquisito, le modifiche possono essere apportate in modo veloce ed informale. Tuttavia, una volta stabilito un documento acquisito, è come

se si fosse passati attraverso l'equivalente di una porta a senso unico. Si possono ancora apportar modifiche, ma ora è necessario applicare una determinata procedura formale che valuti e verifichi ciascuna modifica.

Nell'ingegneria del software, definiamo un documento acquisito come un "punto critico" nello sviluppo del software, segnato dalla consegna di un elemento della configurazione software e dall'approvazione di tale elemento, ottenuta mediante una revisione tecnica formale. Ulteriori modifiche all'architettura del programma (contenuta nella specifica di progettazione) potranno essere apportate solo dopo la loro valutazione ed approvazione.

Un elemento di configurazione del software è anche detto **SCI** (Software Configuration Item) od **Oggetto della Configurazione**. Quando esso diviene un documento acquisito esso viene inserito all'interno di un database, o *repository*, dove può essere preso per essere utilizzato nelle altre parti del progetto. Tuttavia, ribadisco, tale oggetto non potrà essere più modificato, a meno che non si passi attraverso tutte le attività della gestione della configurazione del software. Proprio perché viene riutilizzato, un oggetto della configurazione ha spesso varie dipendenze con altri oggetti della configurazione.

Individuazione degli oggetti della configurazione

Per essere tenuti sotto controllo, gli elementi della configurazione devono ricevere ciascuno un nome ed essere strutturati secondo uno schema orientato agli oggetti. Si possono individuare due tipi di oggetti: oggetti di base ed oggetti aggregati. Un oggetto di base è una "unità testuale" prodotta da un tecnico nel corso di analisi, progettazione, stetura del codice o collaudo. Ad esempio, un oggetto di base può essere una parte della specifica dei requisiti, il listato di un componente od una serie di casi di prova utilizzati per collaudare il codice. Un oggetto aggregato è una collezione di oggetti di base o di altri oggetti aggregati. Ad esempio una "Specifica di progettazione" è un oggetto aggregato in quanto è in realtà un elenco, dotato di nome, comprendente Progettazione dei dati, Progettazione architetturale, Progettazione dei moduli, Progettazione delle interfacce. Esso contiene nient'altro che dei puntatori a questi altri componenti.

Ogni oggetto è individuato da una serie di caratteristiche: un nome, una descrizione, un elenco di risorse ed una "realizzazione". Il nome è una sequenza di caratteri che individua univocamente l'oggetto. La descrizione è un elenco di elementi che specificano: il tipo di oggetto (ad esempio documento, programma, dato), un'identificatore di progetto, informazioni di versioni o modifica. Le risorse sono entità fornite, elaborate, richiamate od in altro modo richieste dall'oggetto. Tipi di dati, funzioni specifiche o perfino nomi di variabili sono esempi di ciò che può fungere da risorsa per un oggetto. La realizzazione è un puntatore all'unità testuale, nel caso degli oggetti base, ed un puntatore nullo, nel caso di oggetti aggregati.

Nell'individuazione di oggetti della configurazione si deve tener conto anche delle relazioni fra gli oggetti. Un oggetto può essere "parte-di" un oggetto aggregato. La relazione "parte-di" determina una gerarchia fra gli oggetti. Non è realistico supporre che le relazioni fra oggetti si limitino a quelle gerarchiche; spesso esistono relazioni orizzontali fra oggetti in rami diversi della gerarchia. Ad esempio un modello dei dati può essere collegato ad un diagramma dei flussi di dati ed ad una famiglia di casi di prova e ciò forma una classe di equivalenza specifica. Questo tipo di relazione è una relazione di "collegato-a". La relazione "parte-di" vige all'interno di un oggetto composto, mentre il secondo collega un oggetto aggregato (il modello dati) ed un oggetto base (classi di casi di prova).

Le relazioni fra gli oggetti della configurazione si possono rappresentare mediante l'uso di MIL (Module Interconnection Language o linguaggio di connessione di moduli). Questo linguaggio indica, per ogni oggetto, quali dati, funzioni e strutture rende disponibili, di quali ha bisogno (relazioni

“collegato-a”) e da cosa è composto (relazioni “parte-di”). Anche l’uso di diagrammi di classi o di casi d’uso (come vedremo quando parleremo dell’Unified Process), ecc. può aiutare a trovare le relazioni tra gli oggetti.

Controllo delle versioni

Quando un oggetto è divenuto documento acquisito ed è stato inserito nel repository, da lì può essere prelevato per il suo utilizzo.

Normalmente i repository permettono di scegliere, oltre all’identificatore dell’oggetto, anche una serie di attributi, che possono essere semplici come il numero di versione od estremamente complicati come una serie di variabili booleane che indicano quali funzioni sono state modificate.

Questo permette una certa flessibilità; infatti si può: tener traccia delle modifiche apportate, sviluppare versioni differenti (ad esempio partendo dalla versione 1.1, si generano due versioni parallele, la 1.1.1 e la 1.2), scegliere quale versione utilizzare e costruire varianti (ad esempio si potrebbe costruire un oggetto aggregato C che usi con i monitor a colori un oggetto A, mentre con i monitor bianco e nero usi un oggetto B; in questo caso di C esisterebbero un’unica versione, ma due varianti).

Controllo del cambiamento

Il controllo dei cambiamenti è fondamentale. Ma le forze che lo rendono necessario lo rendono anche fastidioso. Ci preoccupiamo dei cambiamenti poiché una piccola perturbazione del codice può portare gravi problemi nel prodotto. Ma può anche correggere un grosso errore od attivare una nuova funzionalità. Ci preoccupiamo dei cambiamenti perché un singolo sviluppatore può mettere a rischio il progetto; ma lo stesso sviluppatore potrebbe magari avere delle idee brillanti ed il processo di controllo di un grosso cambiamento potrebbe scoraggiare questa creatività.

Ci si trova, dunque, davanti a dei fattori da bilanciare: troppo controllo sui cambiamenti e si creano problemi; troppo poco e si creano altri problemi. Fermo restando, quindi, questi due punti, un processo di controllo dei cambiamenti è così, schematicamente, rappresentato (vedi pagina seguente):

Una richiesta di cambiamento (vedi avanti per un modello formale) viene sottoposta e valutata per stabilire la sua validità tecnica, i potenziali effetti collaterali, l'impatto globale sugli altri oggetti della configurazione e sulle funzioni di sistema ed i costi previsti. I risultati della valutazione vengono presentati come un report dei cambiamenti (vedi avanti per un modello formale) che viene utilizzato dall'autorità preposta al controllo dei cambiamenti, una persona od un gruppo di persone che prende la decisione finale sullo stato e la priorità del cambiamento. Per ogni cambiamento approvato viene generato un ordine di cambiamento (ECO – Engineering Change Order, vedi avanti per vedere un modello formale). Questo ordine descrive i cambiamenti da eseguire, i vincoli da rispettare ed i criteri di revisione. L'oggetto da modificare viene estratto dall'archivio del progetto, viene eseguito il cambiamento e vengono applicate le attività appropriate di garanzia di qualità. L'oggetto viene poi reinserito nell'archivio e vengono utilizzati gli appropriati meccanismi di controllo della versione per creare la versione successiva del software.

Le operazioni di reinserimento e di estrazione coinvolgono due aspetti importanti: il controllo degli accessi e la sincronizzazione. Il *controllo degli accessi* governa i diritti di accedere ad un determinato elemento della configurazione e di modificarlo da parte dei tecnici; la *sincronizzazione* garantisce

che le modifiche svolte simultaneamente da persone diverse non si annullino l'una con l'altra. In seguito all'approvazione di una richiesta e sulla base dell'ECO relativo, un tecnico estrae un elemento della configurazione. Una funzione di controllo degli accessi garantisce che il tecnico sia autorizzato all'estrazione; il meccanismo di sincronizzazione marca l'oggetto nel database così da impedirne ogni altra modifica fino al suo reinserimento. Si noti che resta possibile estrarre delle copie, ma le modifiche sono impedite. Il tecnico modifica una copia del documento acquisito, detta "versione estratta". Dopo le operazioni di SQA ed i collaudi, l'oggetto modificato viene reinserito e la nuova versione del documento acquisito viene sbloccata rispetto ad altre richieste di modifica.

Abbiamo detto che una rischiesta di cambiamento deve essere approvata da un'autorità preposta al controllo dei cambiamenti. Dato che ogni oggetto della configurazione ha legami con altri oggetti della configurazione, l'autorità preposta al controllo dei cambiamenti varia dal semplice capo progetto, per una modifica locale che non interessa altri moduli (nome delle variabili interne, ad esempio), ad un'insieme di esperti (rappresentanti dei settori software, hardware, database, assistenza, marketing e così via), se la modifica coinvolge altri oggetti od entra in merito ad altri campi (cambiamento dell'interfaccia utente, ad esempio). Alcune domande che l'autorità si deve porre sono: Quali effetti ha la modifica sull'hardware? E sulle prestazioni? In che modo può cambiare la percenzione che il cliente ha del prodotto? Come incide sulla qualità e sull'affidabilità?

Esami della configurazione

Una volta che la richiesta di modifica è stata accettata, è stato prodotto l'ECO e si è effuttata la modifica, sorge una domanda: Come si può garantire che la modifica sia stata correttamente effettuata? In due modi: le revisioni tecniche formali, descritte nel capitolo sulla Qualità del Software, e l'esame della configurazione, che andremo a descrivere di seguito.

Un *esame della configurazione software* completa la revisione tecnica formale, analizzando le caratteristiche dell'oggetto generalmente ignorate dalla revisione. L'esame deve rispondere alle domande seguenti:

1. La modifica specificata nell'ECO è stata effettuata? Si sono aggiunte altre modifiche?
2. È stata svolta una revisione tecnica formale per valutare la correttezza sul piano tecnico?
3. Il processo software è stato seguito e gli standard di ingegneria del software sono stati applicati correttamente?
4. La modifica è stata messa in risalto nello SCI? Sono stati specificati l'autore e la data dell'operazione? Gli attributi dell'elemento della configurazione riflettono la modifica?
5. Sono state eseguite le procedure SCM per annotare, registrare e riferire la modifica?
6. Sono stati opportunamente aggiornati tutti gli elementi della configurazione correlati?

Relazioni sulla situazione

La stesura di resoconti è tra le attività più importanti del processo di controllo e gestione del cambiamento. Infatti tali resoconti servono per capire: che cosa è accaduto, chi ne è il responsabile, quando è accaduto, che cosa ne subisce le conseguenze.

Inoltre i resoconti possono essere collocati in un database in linea, cosicché sviluppatori ed addetti alla manutenzione possano accedere alle informazioni sulle modifiche per parola chiave. Inoltre periodicamente viene compilato un resoconto sullo stato della configurazione per tenere aggiornati la direzione e lo staff di ogni cambiamento importante.

I resoconti sullo stato della configurazione svolgono un ruolo vitale nella riuscita di un progetto di sviluppo di software di grandi dimensioni. L'alto numero di persone coinvolte spesso fa sì che "la mano sinistra non sappia cosa fa la destra". Può capitare che due persone tentino di modificare lo stesso oggetto con scopi contrastanti. Il team può passare mesi interi costruendo programmi sulla base di una specifica hardware obsoleta. Una persona che sarebbe in grado di rilevare le conseguenze dannose di una modifica proposta, non sa che la modifica è in effetti già in corso. La stesura di resoconti sullo stato della configurazione contribuisce ad eliminare problemi di questo genere migliorando la qualità delle comunicazioni fra il personale.

Documenti formali

Qui di seguito vengono evidenziati i vari modelli formali: la richiesta di cambiamento, il report dei cambiamenti, l'ECO, il configuration audit report, cioè il documento che viene compilato quando il cambiamento deve essere propagato, ed il resoconto sullo stato della configurazione.

Software Change Request (Richiesta di cambiamento)

Questo documento viene generato quando viene richiesto un cambiamento.

1.0 Identificazione dello SCI

Questa sezione identifica lo SCI per il quale il cambiamento è richiesto.

1.1 Nome, identificazione e descrizione dello SCI

Vengono specificati il nome, il numero di versione/revisione/variante dello SCI, nonché il numero delle pagine se vi è un documento che lo riguarda. Una breve descrizione è data.

1.2 Richiedente

Il nome della persona che richiede il cambiamento

1.3 Informazioni di contatto

Come contattare il richiedente (serve se bisogna fargli domande specifiche).

1.4 Data, luogo ed ora

Dove e quando il cambiamento è stato richiesto

2.0 Descrizione del cambiamento

Questa sezione presenta una breve descrizione della richiesta di cambiamento

2.1 Descrizione

Questa sezione presenta una dettagliata descrizione delle circostanze che hanno partecipato alla richiesta del cambiamento e la richiesta nel desiderata.

2.1.1 Circostanze

Informazioni di Background riguardanti la richiesta.

2.1.2 Esempi

Informazioni di supporto come stampe di un report contenenti errori o un'incorretta immagine nello schermo

2.1.3 Il cambiamento

Una discussione dettagliata sul cambiamento richiesto.

2.2 Ragioni e giustificazione

Questa sezione discute perché il cambiamento è richiesto e la giustificazione per la richiesta.

2.3 Interesse percepito

La percezione del richiedente sull'interesse del cambiamento.

2.4 Alternative

La percezione del richiedente su ogni possibile alternativa al cambiamento.

2.5 Priorità richiesta

La priorità assegnata dal richiedente sul lavoro, generalmente scelta da una scala di 5 possibili valori.

Software Change Report (Report dei cambiamenti)

Questo documento viene generato una volta valutato il cambiamento (cioè una volta stabilita la sua validità tecnica, i potenziali effetti collaterali, l'impatto globale sugli altri oggetti della configurazione e sulle funzioni di sistema ed i costi previsti).

1.0 Identificazione dello SCI

Questa sezione identifica lo SCI per il quale il cambiamento è stato richiesto.

1.1 Nome, identificazione e descrizione dello SCI

Vengono specificati il nome, il numero di versione/revisione/variante dello SCI, nonché il numero delle pagine se vi è un documento che lo riguarda. Una breve descrizione è data.

1.2 Richiedente

Il nome della persona che richiede il cambiamento.

1.3 Valutatore

Il nome della persona che ha valutato la richiesta di cambiamento.

1.4 Informazioni di contatto

Come contattare il richiedente od il valutatore (serve se bisogna fargli domande specifiche).

1.5 Data, luogo ed ora

Quando e dove il report dei cambiamenti è stato generato.

2.0 Valutazione del cambiamento richiesto

Questa sezione presenta una breve valutazione sulla richiesta di cambiamento.

2.1 Descrizione degli SCI interessati

Questa sezione discute gli SCI interessati dalla richiesta di cambiamento.

2.2 Categoria del cambiamento

Questa sezione classifica il cambiamento in una categorie di cambiamento (Di base, funzionale, fisica, vedi il Configuration Audit Report più sotto per le spiegazioni delle categorie).

2.3 Scopo del cambiamento

La valutazione del valutatore sullo scopo del cambiamento includendo una lista di tutti gli SCI

interessati e di tutti gli utenti che dovranno essere informati.

2.3.1 Richiesta lavoro tecnico

Una descrizione del lavoro richiesto per completare il cambiamento

2.3.2 Risorse speciali e tools

Tools od altre risorse speciali richieste sono annotate qui.

2.4 Effetti di Interoperabilità

L'impatto del cambiamento su altri sistemi.

2.5 Rischio tecnico

Sono descritti i rischi associati con la descrizione dei cambiamenti.

3.0 Valutazione del costo

Questa sezione presenta una valutazione del costo della richiesta del cambiamento includendo un stima dello sforzo e del tempo richiesti.

4.0 Raccomandazioni / Disposizioni

Questa sezione presenta la raccomandazione del valutatore riguardante il cambiamento, la sua priorità interna, e le ultime disposizione sul cambiamento, basati per una futura valutazione da parte della autorità preposta al controllo dei cambiamenti.

4.1 Raccomandazione

Il cambiamento va fatto oppure no, tenendo conto delle varie valutazioni?

4.2 Priorità interna

Quanto è importante questo cambiamento alla luce del lavoro che deve essere effettuato e sulle altre considerazioni riguardanti il business / prodotto?

4.3 Disposizioni

Se il cambiamento andrà fatto: quando?

Engineering Change Order (ECO)

Questo documento viene prodotto una volta che il cambiamento è stato approvato

1.0 Identificazione dello SCI

Questa sezione identifica lo SCI per il quale il cambiamento verrà fatto.

1.1 Nome, identificazione e descrizione dello SCI

Vengono specificati il nome, il numero di versione/revisione/variante dello SCI, nonché il numero delle pagine se vi è un documento che lo riguarda. Una breve descrizione è data.

1.2 Richiedente

Il nome della persona che ha richiesto il cambiamento.

1.3 Valutatore

Il nome della persona che ha valutato la richiesta del cambiamento.

1.4 Informazioni di contatto

Come contattare il richiedente ed il valutatore (serve se bisogna fargli domande specifiche).

2.0 Descrizione del cambiamento che deve essere fatto

Questa sezione presenta una breve descrizione del cambiamento.

2.1 Descrizione deli SCI interessati

Questa sezione discute gli SCI che saranno interessati dalla richiesta di cambiamento.

2.2 Categoria del cambiamento

Questa sezione classifica il cambiamento in una categoria di cambiamento (Di base, funzionale, fisica, vedi il Configuration Audit Report più sotto per le spiegazioni delle categorie).

2.3 Scopo del cambiamento

La valutazione del valutatore sullo scopo del cambiamento includendo una lista di tutti gli SCI interessati e di tutti gli utenti che dovranno essere informati.

2.3.1 Richiesta lavoro tecnico

Una descrizione del lavoro richiesto per completare il cambiamento

2.3.2 Risorse speciali e tools

Tools od altre risorse speciali necessarie.

2.4 Effetti di Interoperabilità

L'impatto del cambiamento su altri sistemi.

2.5 Rischio tecnico

Sono descritti i rischi associati con la descrizione dei cambiamenti.

3.0 Analisi e progettazione delle modifiche

Una breve descrizione delle modifiche al modello di analisi e progettazione che sono richieste.

3.1 Descrizione delle modifiche al modello di analisi

I cambiamenti richiesti al modello di analisi sono descritti. I modelli di dati, funzioni e comportamenti sono tutti considerati.

3.2 Descrizione delle modifiche al modello di progettazione

I cambiamenti richiesti al modello di progettazione sono descritti. I modelli di dati, architetture, interfacce e livelli di componenti sono tutti considerati.

4.0 Requisiti di validazione

Una descrizione delle attività di SQA e di approccio al collaudo richiesti per assicurare che il cambiamento sia fatto senza effetti collaterali.

4.1 Piano rivisto

Descrive il tipo di revisione che deve essere condotto.

4.2 Piano di test

Descrive i test di regressione ed i nuovi test che saranno richiesti.

Configuration Audit Report

Questo documento viene prodotto una volta che il cambiamento è stato attuato

1.0 Informazioni sull'audit

Breve descrizione dell'audit

1.1 Tipo di Audit

Viene indicato il tipo di Audit generato. Esso può essere:

- **Di base (Baseline)**: la modifica riguarda i requisiti, il modello di analisi o quello di progettazione
- **Funzionale (Functional)**: la modifica cambia funzionalità, performance, operazioni possibili, documenti supportati, sicurezza, ecc. di uno SCI
- **Fisica (Physical)**: la modifica cambia hardware, software, documentazione (tecnica, utente, strutture dati), progettazione (intesa in senso inverso: un modulo diviso in due ad esempio), ecc. di una SCI

1.2 Data dell'audit

Viene indicata la data in cui è stato generato l'audit

1.3 Responsabile dell'audit

Chi ha generato l'audit

2.0 Identificazione dello SCI

Questa sezione identifica lo SCI per il quale il cambiamento è stato fatto.

2.1 Nome, identificazione e descrizione dello SCI

Vengono specificati il nome, il numero di versione/revisione/variante dello SCI, nonché il numero delle pagine se vi è un documento che lo riguarda. Una breve descrizione è data.

2.2 Responsabile

Il nome della/e persona/e che ha curato il cambiamento.

2.3 Informazioni di contatto

Come contattare il responsabile (serve se bisogna fargli domande specifiche).

3.0 Descrizione del cambiamento

Viene fornita una descrizione esaustiva del cambiamento apportato, nonché di tutti i problemi e gli errori che questo potrebbe portare

3.1 Descrizione

Una descrizione del cambiamento

3.2 Problemi o discrepanze che potrebbero essere riscontrate (opzionale)

Viene fornita una descrizione dei problemi o delle discrepanze a cui si va incontro con questo cambiamento e vengono descritti i relativi punti a cui fare attenzione

3.3 Errori riscontrati (opzionale)

Vengono identificati gli altri SCI ed i punti che necessitano obbligatoriamente di controlli e mo-

difiche a causa del cambiamento

3.4 Raccomandazioni ed azioni (opzionale)

Vengono fornite le raccomandazioni e le azioni necessarie da attuare, nonché i responsabili.

Configuration Status Report (resoconto sullo stato della configurazione)

Questo documento viene prodotto periodicamente per informare e tenere aggiornati tutti gli interessati ai cambiamenti apportati, quelli in sospeso, ecc. Il modulo qui sotto stante è un esempio di quello che contiene un resoconto sullo stato della configurazione.

Nome		Data	
Team		Responsabile	
Processo di controllo del cambiamento: Attività			
	Settimana corrente		Ciclo alla data
Cambiamenti sottomessi			
Camviamenti approvati			
Cambiamenti rigettati			
Cambiamenti rinviati			
Cambiamenti eccezionali			
Cambiamenti invertiti			
Processo di controllo del cambiamento: Stato			
Volume prodotto sotto controllo SCM	Settimana corrente		Cambiamenti dalla precedente settimana
Pagine di testo			
Pagine di progettazione			
Linee di pseudocodice			
LOC totali			
LOC nuovi e cambiati			
LOC dei casi di collaudo			
Pagine di materiale di test			
Pagine di risultati di test			
Altri oggetti			
Altri oggetti			

Commenti

PARTE QUARTA

Unified Process

Lo Unified Process è un processo di sviluppo software iterativo ed incrementale basato sui componenti, centrato sull'architettura e guidato dai casi d'uso.

1. Con iterativo si intende un processo non sequenziale che ripassa più e più volte su alcune attività e/o fasi di sviluppo al fine di migliorare sempre di più il prodotto
2. Con incrementale si intende che ad ogni interazione viene aggiunto qualcosa al prodotto
3. Con basato sui componenti si intende che il prodotto è strutturato in modo che ogni unità sia autonoma e collabori con le altre unità; favorisce lo sviluppo attraverso la programmazione ad oggetti
4. Con centrato sull'architettura si intende che si presuppone che si scelga una volta per tutte il tipo di “sistema” su cui gira la macchina (client-server, distribuita, locale, centralizzata, ecc.), i comportamenti delle interfaccie (come devono dialogare le varie parti), ed in generale l'organizzazione del sistema
5. Con guidato dai casi d'uso si intende che il prodotto viene sviluppato guardando principalmente a cosa deve fare e secondariamente al come.

Lo Unified Process si basa sull'UML come supporto per la documentazione tecnica che viene prodotta.

Sviluppare un prodotto con l'ausilio dello Unified Process significa procedere a piccoli passi, migliorando di volta in volta i risultati ottenuti e modellare lo sviluppo seguendo una metodologia ad oggetti, cioè si separano le competenze dei singoli moduli del programma.

Fasi ed attività

Lo Unified Process divide lo sviluppo del software in 4 fasi; ogni fase è a sua volta costituita da 5 attività (o flussi di lavoro). Ogni fase si considera conclusa una volta prodotti i documenti previsti, mentre le singole attività avranno peso diverso a seconda della fase corrente. A differenza di quanto ci si possa aspettare il confine delle fasi è piuttosto blando e si passa da una fase all'altra con estrema naturalezza.

Le fasi sono le seguenti:

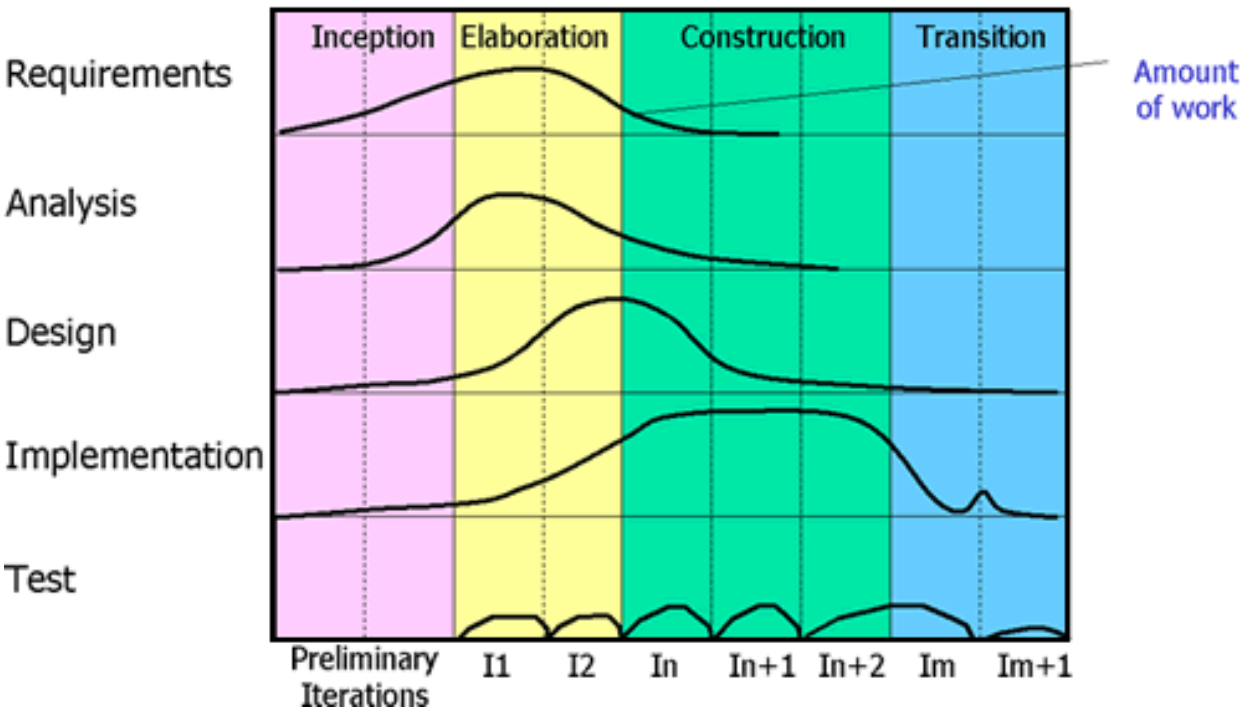
- **Inception** – Lo scopo di questa fase è trovare gli obbiettivi del ciclo di vita
- **Elaboration** – Lo scopo di questa fase è trovare l'architettura di base per il ciclo di vita
- **Construction** – Lo scopo di questa fase è costruire il software
- **Transition** – Lo scopo di questa fase è quello che viene definito come Product Release, cioè il rilascio del prodotto.

Le attività sono le seguenti:

- **Requirements** (Requisiti) – Lo scopo di questa attività è trovare cosa deve fare il sistema
- **Analisys** (Analisi) – Lo scopo di questa attività è raffinare e strutturare i requisiti
- **Design** (Progettazione) – Lo scopo di questa attività è realizzare i requisiti in un sistema architetturale
- **Implementation** (Implementazione) – Lo scopo di questa attività è costruire il software

- **Test –** Lo scopo di questa attività è verficare che l'implementazione lavori come desiderato.

Ognuna di queste attività o flusso di lavoro compare in maniera più o meno accentuata a seconda della fase in cui si ci trova. La figura seguente individua la relazione tra Fasi ed Attività:



Fase 1: Inception

Gli obbiettivi della fase di Inception sono quelli di stabilire delle possibilità su ciò che può fare o non fare il prodotto; può richiedere delle tecniche di prototipizzazione al fine di validare i requisiti richiesti. Durante questa fase viene realizzato il caso di businnes cioè una vista ad alto livello sugli scopi del sistema (approfondiremo il concetto quando parleremo dell'attività dei Requisiti). Importante notare che in questa fase si cercano di catturare i requisiti essenziali affinché il sistema funzioni. Nella fase successiva verranno approfonditi i requisiti in modo tale da soddisfare pienamente il cliente; ma è importante che il progetto abbia uno scopo ben definito.

Essendo questa fase quella iniziale in cui si stabilisce anche se far partire o no il progetto, è in questa fase che si cerca di calcolare costi e rischi.

Il fuoco di questa fase sono proprio le iterazioni preliminari, cioè tutti gli incontri necessari a capire le esigenze dell'utente e valutarne costi e rischi; L'unica attività di progettazione o di implementazione che può essere fatta è nel caso in cui si decida di realizzare un prototipo dell'interfaccia per poter dialogare meglio col cliente.

Le pietre miliari della fase di Inception sono gli obbiettivi del ciclo di vita. Ad ogni pietra miliare corrisponde una documentazione da produrre:

Condizioni di Soddisfazione	Produzioni
Gli stakeholders hanno approvato gli obbiettivi di progetto	Un documento di visione che dichiara i requisiti principali, le caratteristiche ed i vincoli del progetto
La portata del sistema è stata definita ed è stata approvata dagli stakeholders	Un modello di casi d'uso iniziale (solamente dal 10% al 20% completo)
I requisiti chiave sono stati catturati	Un glossario di progetto
Valutazioni di programma e di costo sono stati approvati dagli stakeholders	Un piano di progetto iniziale
Un caso di business è stato preparato dal responsabile di progetto	Caso di business
Il responsabile di progetto ha effettuato una valutazione di rischio	Un documento o un database di valutazione del rischio
Conferma di fattibilità con gli studi tecnici e/o prototipizzazione	Uno o più prototipi funzionanti
Un profilo di architettura è stato individuato	Un documento di architettura iniziale

Uno o più documenti li vedremo quando parleremo delle attività (sono esse che producono i documenti).

Fase 2: Elaboration

La fase di elaboration è la fase più critica, in quanto da essa dipendono tutte le altre fasi.

L'idea è quella di dare un primo taglio al sistema eseguibile, in modo tale da evolverlo fino al sistema finale. È importante: esso non è un prototipo!

Durante questa fase raffiniamo la valutazione del rischio, individuiamo le risorse, il tempo, l'apparecchiatura, il personale ed il costo necessari (e, ovviamente, si chiede conferma al cliente: se non passa il progetto abortisce), definiamo gli attributi di qualità (tasso di errori scoperti, densità di difetti accettabile), si prepara un piano di costruzione dettagliato, si catturano l'ottanta per cento dei requisiti funzionali e si crea il primo eseguibile sull'architettura di base.

Il focus di questa fase è chiaramente sui requisiti, l'analisi ed il progetto. L'implementazione diviene importante alla fine della fase quando l'architettura di base deve essere prodotta.

Ogni attività ha il suo specifico ruolo:

Requisiti – deve estrarre tutti i requisiti richiesti e raffinare quelli già estratti durante la fase di Inception

Analisi – deve stabilire cosa deve essere costruito

Progettazione – deve creare un'architettura stabile

Implementazione – deve costruire l'architettura di base

Test – deve testare l'architettura di base

Alla fine della fase di elaboration il sistema deve essere pronto per la sua costruzione.

Le pietre miliari sono l'architettura del ciclo di vita. Ad ogni pietra miliare corrisponde una documentazione da produrre:

Condizioni di soddisfazione	Produzioni
Un'architettura di base eseguibile e robusta è stata creata. L'architettura di base eseguibile dimostra che il rischio importante è stato identificato e risolto	L'architettura di base eseguibile, il modello statico di UML, il modello dinamico di UML, il modello dei casi d'uso di UML
La visione del prodotto è stata stabilizzata	Documento di visione
La valutazione del rischio è stata modificata	Valutazione del rischio aggiornata
Il caso di business è stato modificato ed è stato approvato dagli stakeholders	Caso di business aggiornato
Un programma di progetto è stato generato in dettaglio sufficiente per permettere ad un'offerta realistica, su tempo, soldi e risorse per le fasi successive, di essere formulata	Piano di progetto aggiornato
Gli stakeholders approvano il piano di progetto. Il piano di progetto è stato verificato contro il caso di business	Caso di business e piano di progetto
Un accordo è raggiunto con gli stakeholders per continuare il progetto	Documento firmato

Uno o più documenti li vedremo quando parleremo delle attività (sono esse che producono i documenti).

Fase 3: Construction

L'obbiettivo è completare tutti i requisiti, l'analisi ed il progetto ed evolvere l'architettura di base, generata nella fase di Elaboration, nel sistema finale.

Attenzione a non perdere l'architettura costruita: è tipico che quando la scadenza si avvicina si cerca di risolvere i problemi nel modo più veloce possibile, spesso sfornando sistemi con bassa qualità ed alto costo di manutenzione.

L'enfasi è sulla fase di implementazione, dato che la maggior parte di analisi e progettazione è già stata fatta nella fase precedente. L'attività di test diviene molto importante, in quanto bisogna provare le varie aggiunte e le relative integrazioni.

Durante questa fase il progetto va arrivando alla sua fase conclusiva. Le pietre miliari sono la possibilità operativa iniziale. Ad ogni pietra miliare corrisponde una documentazione da produrre:

Condizioni di Soddisfazione	Produzioni
Il prodotto software è sufficientemente stabile e sufficientemente di qualità da essere sviluppato per la comunità di utenti	Il prodotto software, il modello UML, la suite di Test

Condizioni di Soddisfazione	Produzioni
Gli stakeholders hanno approvato e sono pronti per la transizione del software nel loro ambiente	Manuali utente, descrizione della release
Le spese effettive contro le spese previste sono accettabili	Piano di progetto

Uno o più documenti li vedremo quando parleremo delle attività (sono esse che producono i documenti).

Fase 4: Transition

La fase di Transition inizia quando il beta testing è completato ed il sistema finale è sviluppato. Questa fase deve risolvere i problemi trovati durante il beta testing ed installare il software nell'ambiente degli utenti.

Sommariamente deve:

- Correggere i difetti
- Preparare il sito degli utenti al nuovo software
- Adattare il software per operare sul sito degli utenti
- Modificare il software se problemi imprevisti si presentano
- Creare i manuali dell'utente ed altra documentazione
- Fornire consulenza all'utente
- Condurre una revisione post progetto

Arrivati a questa fase, i giochi sono fatti: il software deve essere pronto e l'unica fase di implementazione è la correzione di errori trovati durante l'ultimo beta testing e la preparazione al trasferimento sul sito dell'utente. La fase di test riguarda l'installazione del sistema (controllare che tutto vada bene) e testare la soddisfazione dell'utente.

Le pietre miliari sono la Release Product. Ad ogni pietra miliare corrisponde una documentazione da produrre:

Condizioni di Soddisfazione	Produzioni
Il beta testing è completato, i cambiamenti necessari sono stati fatti e gli utenti hanno approvato il sistema	Il prodotto software
La comunità degli utenti sta usando attivamente il prodotto	
Il prodotto supporta le strategie accordate con gli utenti ed implementate	Piano di supporto all'utente, Manuali utente

Ovviamente con piano di supporto all'utente si intendono l'assistenza e la manutenzione del software, nonché l'eventualità di nuove release in tempi futuri.

Requirements (Requisiti)

Dato che i requisiti indicano cosa deve fare il sistema, il compito di questa attività è capire cosa fare, non come. Il fatto di mantenere separate le due cose, permette un migliore sviluppo iterattivo ed incrementale.

L'analisi dei requisiti riguarda sia quelli funzionali (cosa deve fare il programma) che quelli non funzionali (specifiche e vincoli sull'implementazione del sistema). Specificatamente un requisito funzionale è alla base del sistema e può, spesso, essere tradotto in un caso d'uso (vedi avanti), mentre un requisito non funzionale non è traducibile in un caso d'uso, ma vincola tutte le successive attività implementative.

Durante questa attività, viene prodotto il documento SRS (System Requirements Specification), cioè un elenco in linguaggio naturale dei requsiti specificati nella forma "Il *Sistema* fa *funzione*". Ad ogni requisito viene assegnato un identificatore univoco. A questo documento segue il Supplementary Requirements Specification (anch'esso abbreviato con SRS), in cui i requisiti sono messi in forma tabellare, con le seguenti colonne:

ID – identificatore

Dettaglio – specifica in linguaggio naturale

Tipo – è un requisito funzionale o non funzionale? Di cosa fa parte il requisito? Quest'ultima domanda dipende da quali classificazioni possono essere dati ai requisiti; ad esempio ci può essere il tipo Autenticazione che indica che è un requisito necessario all'autenticazione degli utenti, mentre il tipo Prodotto, specifica un requisito per la memorizzazione delle informazione dei prodotti, ecc. (i tipi vanno nel glossario)

Priorità – Deve Avere, Potrebbe Avere, Sarebbe meglio Avere, Facoltativo, Consigliato, Indispensabile, ecc. (le priorità vanno nel glossario).

Nella pratica il System Requirements Specification ed il Supplementary Requirements Specification vengono fusi in un unico documento (da cui la stessa abbreviazione); l'SRS è la tabella descritta sopra, che viene via via riempita a mano a mano che vengono scoperti e classificati i requisiti.

Sempre nella pratica l'SRS è completato con i *valori di pianificazione* e cioè:

- Stato (Proposto, approvato, incorporato, validato, ecc.)
- Stima dei costi di implementazione di tale requisito.
- Livello di rischio stimato.

L'attività di estrazione dei requisiti richiede di porre attenzione sui quantificatori universali ed esistenziali: a volte gli utenti usano questi quantificatori in modo inappropriato. Ad esempio: "Tutti gli utenti possono accedere al sistema senza autenticarsi (nessuno escluso)." Dopo molte altre specifiche: "Esiste un tipo di utente che deve farsi autenticare, altrimenti non può accedere." I quantificatori sono sbagliati: la specifica richiede, infatti, che esistano degli account speciali a cui si può accedere dopo l'autenticazione, se l'utente non si autentica, viene usato un account guest (ospite).

Il **glossario del progetto** è un dizionario che associa ad ogni parola, il suo significato. È usato per evitare sinonimi ed omonimi nel progetto. I primi sono pericolosi perché a nomi diversi corrispondono stesse definizioni (e si rischia di trovarsi con due classi identiche ma con nomi diversi), mentre i secondi sono pericolosi perché agli stessi nomi corrispondono significati diversi (e si rischia di creare classi sovracaricate od addirittura con funzionalità in contrasto tra loro e quindi impossibili da realizzare). Il glossario evita di inserire questi termini, rendendo univoci significati e nomi (corri-

spondenza uno ad uno), scegliendo un unico termine per i sinonimi e trovando nuovi termini per gli omonimi. Nelle descrizione possono comparire i sinonimi per dialogare con gli stakeholders (che spesso sono restii ad adottare un linguaggio diverso).

Per tutto il ciclo di vita del software si useranno le parole del glossario, quando bisogna riferirsi a funzionalità del sistema.

Prima di partire cercando gli attori (chi o cosa usa il sistema) ed i casi d'uso (cosa fa il sistema) è importante conoscere i limiti del progetto (sembra stupido, ma in certi casi risulta problematico trovare i confini del progetto; senza confini un progetto non può finire). Proprio per trovare i limiti del progetto viene introdotto il modello di business (più nello specifico i casi d'uso di business). Con esso si cerca di focalizzare lo scopo del sistema: si vede il sistema come una grande scatola nera, che riceve degli input da parte di utenti (possono essere sia persone che altri sistemi) esterni e fornisce loro i risultati aspettati. A volte questa scatola nera viene aperta ed ampliata quando è necessario per capire il contesto in cui si opera. Ad esempio supponiamo di dover realizzare un sistema per il commercio elettronico di libri e cd. Visto come scatola nera, abbiamo un utente che fornisce al sistema cosa vuole comprare ed il sistema gli risponde che la merce gli verrà inviata, oppure se c'è qualche problema. Uhm... un po' poco. Apriamo la scatola nera e vediamo che essa è formata da altre scatole nere che agiscono tra di loro: un magazzino che possiede gli articoli, un sistema di autenticazione che gestisce gli utenti, un sistema di pagamento per gestire il pagamento dei libri/cd, l'interfaccia di gestione verso il cliente. Queste scatole nere comunicano tra di loro passandosi degli input e degli output, in modo tale da realizzare la scatola nera più grande. Bene questa descrizione sarà il nostro caso d'uso di business, cioè su cosa dobbiamo lavorare; ciò che non rientra non è necessario.

Nella pratica ci può essere più di un caso d'uso di business (per questo si usa il termine Modello di Business): questo capita quando il sistema richiede di essere spezzato in più sottosistemi (nell'esempio precedente, ci poteva essere un nuovo caso d'uso di business per la reale spedizione della merce).

Non è sempre facile trovare di primo acchito il/i caso/i di business, né sapere se essi vanno divisi in più casi d'uso oppure no. Proprio per questo motivo il modello non è considerato scolpito sulla pietra, ma alle successive iterazioni può essere espanso. L'importante è ricordare di non perdere di vista lo scopo del sistema.

Abbiamo detto che scopo di questa attività è estrarre i requisiti e tradurli in casi d'uso (almeno quelli funzionali). La prima domanda da porsi è: cos'è un caso d'uso? Formalmente un caso d'uso rappresenta una funzione del business che è esternamente visibile ad un attore e può essere controllata singolarmente nel processo di sviluppo. Un attore rappresenta chiunque o qualunque cosa (persona, macchinario, ecc.) interagisca col sistema per ottenere un servizio od un qualche risultato utile dal sistema stesso.

Per visualizzare i casi d'uso UML ci fornisce i *diagrammi dei casi d'uso*: con esso gli attori sono rappresentati da omini stilizzati, mentre i casi d'uso da ellissi contenenti il nome del caso d'uso. Attori e casi d'uso sono collegati tra loro con delle linee che rappresentano l'interazione tra i soggetti. Un attore può comunicare con uno o più casi d'uso, mentre un caso d'uso può comunicare con attori e casi d'uso; questo porta al fatto che un attore non ha senso se non interagisce con un caso d'uso, mentre può aver senso un caso d'uso che non interagisca con alcun attore.

Entriamo nel dettaglio.

Diagramma dei casi d'uso

Il diagramma dei casi d'uso assegna i casi d'uso agli attori ed evidenzia le relazioni tra i casi d'uso.

Nel diagramma dei casi d'uso un attore può essere rappresentato da un omino stilizzato, con sotto il nome dell'attore (normalmente rappresenta un attore esterno), od un rettangolo contenente il nome dell'attore e la parola «actor» (per attori che rappresentano classi da implementare od altri fattori, come il tempo). Un caso d'uso è rappresentato come un ellissi con all'interno il nome del caso d'uso.

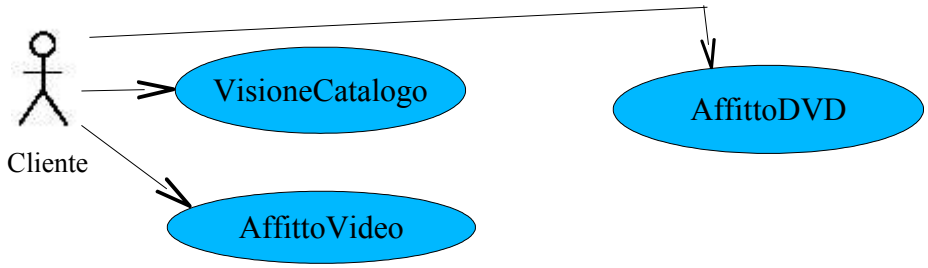
Gli attori sono associati ai casi d'uso per mezzo di frecce che puntano dall'attore al caso d'uso. Le relazioni tra casi d'uso sono rappresentate attraverso frecce tratteggiate su cui vi è assegnato un nome tra una lista predefinita. Oltre a questo è possibile la generalizzazione degli attori e dei casi d'uso.

Prima di addentrarci ancora più nei particolari, vediamo un esempio chiarificatore:

Ipotizziamo che una videoteca decida di modernizzare il suo sistema di affitto e vendita di videocassette e DVD. Il sistema da implementare deve permettere agli utenti in possesso di tessera di affittare una videocassetta od un DVD. Il sistema prevede la selezione della videocassetta o del DVD da un catalogo elettronico.

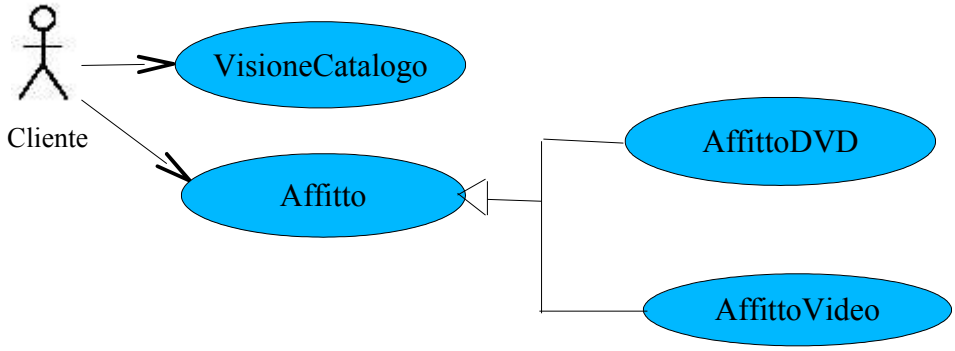
Senza addentrarci in altri particolari (la descrizione normalmente sarà ben più fornita di particolari) individuiamo l'attore cliente e vari casi d'uso: VisioneCatalogo, AffittoVideo, AffittoDVD.

Rappresentiamo questi casi d'uso:



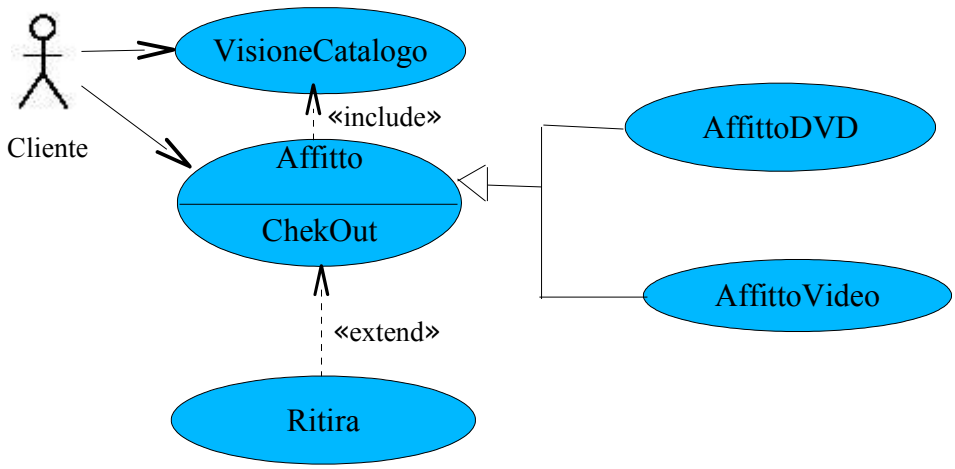
Abbiamo detto che è possibile la generalizzazione: essa, allo stesso modo delle classi, è una relazione tra due attori (o casi d'uso): un attore (od un caso d'uso) che specializza un attore (od un caso d'uso) più generale. Esso è rappresentato da una freccia con la punta triangolare bianca.

Nell'esempio fatto precedentemente, ci poteva essere il caso d'uso Affitto che veniva specializzato dal caso d'uso AffittoVideo ed AffittoDVD:



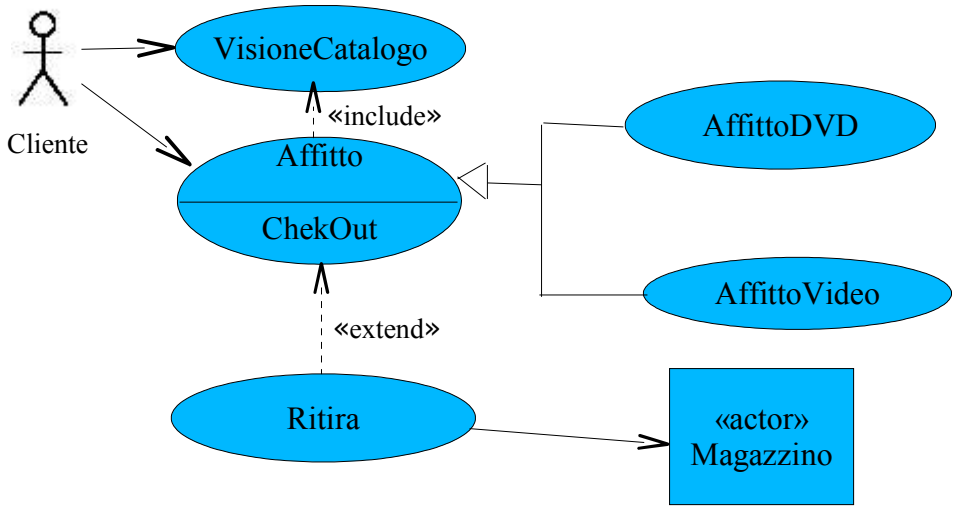
Gli altri due tipi di associazioni tra casi d'uso sono: Inclusione (rappresentato da una freccia tratteggiata con la scritta «include») e l'Estensione (rappresentato da una freccia tratteggiata con la scritta «extend»). Con la prima si identifica un caso d'uso che include sistematicamente il caso d'uso collegato con la freccia; il caso d'uso non può essere eseguito senza eseguire l'altro caso d'uso (non è vero il contrario). Con la seconda si intende che un caso d'uso estende il caso d'uso collegato per particolari “punti di estensione”; un punto di estensione indica una condizione in cui il caso d'uso base (quello puntato dalla freccia) demanda ad un altro caso d'uso la sua esecuzione. L'estensione indica comportamenti opzionali. Quando si usa l'estensione è buona norma indicare i punti di estensione nel caso d'uso base.

Nell'esempio: Aggiungiamo che per affittare un film è necessario visionare il catalogo, ma è possibile affittare più di un film e ritirarli in blocco, dopo aver fatto le proprie scelte (un po' come il carrello dei siti di e-commerce). Dunque Affitto include VisioneCatalogo, mentre il caso d'uso Ritira estende il caso d'uso Affitto. Nel primo caso si intende che ogni volta che bisogna Affittare (videocassetta o DVD non importa) bisogna Visonare il catalogo, mentre nel secondo si indica che dopo che l'utente ha deciso quale film affittare, lo ritira (non può ritirare dei film se prima non ha deciso di affittarli); tuttavia è possibile che lo affitti ma non lo ritiri, perché prima vuole Visionare nuovamente il catalogo per trovare un altro film da affittare. Prendendo spunto dalla terminologia dell'e-commerce, chiamiamo il punto di estensione CheckOut:



Importante ricordare che un caso d'uso che ne estende un altro, può essere assegnato ad un attore, in quanto il caso d'uso potrebbe fornire risultati ad un attore.

Nell'esempio l'operazione di ritiro informa il magazzio della diminuizione delle scorte. Qui rappresenteremo l'attore con l'altra rappresentazione in quanto non è un attore umano, ma una parte del sistema ad essere informato:



Non sempre è necessario usare le freccie tra attori e casi d'uso: se vengono usate linee continue senza freccia non si sta specificando se è l'attore che usa il caso d'uso, od il caso d'uso che dà informazioni all'attore (probabilmente entrambi).

Documentare i casi d'uso

Il diagramma dei casi d'uso dà una buona panoramica su cosa deve fare il sistema; tuttavia da solo non basta: ogni caso d'uso dovrebbe avere una descrizione che lo accompagna per specificare i suoi compiti, eventuali flussi alternativi (ad esempio quando qualcosa non va a buon fine), condizioni da rispettare, ecc. In più anche gli attori vanno descritti, in quanto non è sempre chiaro dal nome il ruolo che l'attore copre.

Al diagramma dei casi d'uso vengono dunque allegati due documenti: la Specifica degli attori e la Specifica dei casi d'uso.

Il primo documento è formato da una semplice tabella:

Attore	*Nome dell'attore*
ID	*Identificativo*
Genitore	*Nel caso sia sotto generalizzazione, specifica quale attore specializza*
Ruolo	
Descrizione del ruolo dell'attore.	

Ovviamente i testi in corsivo vanno sostituiti con i dati reali.

Il secondo documento è anch'esso formato da una tabella, ma più articolata:

Caso d'uso	*Nome del caso d'uso*
ID	*Identificatore*
Breve descrizione	*Breve descrizione del caso d'uso*
Attori	*Attori coinvolti*

Precondizioni	*Condizioni che devono essere verificate affinché il caso d'uso possa essere eseguito*
Flusso Principale	*Descrizione testuale delle operazioni che compiono gli attori ed il caso d'uso, nonché le risposte*
Scenari secondari	*Eccezioni che possono avvenire nel caso d'uso (la descrizione di questo scenario è riportata su un'altra tabella)*
Flussi alternativi	*Alternative al flusso principale (sia errori facilmente descrivibili che vere e proprie diramazioni del flusso principale)*
Postcondizioni	*In che stato si trova il sistema dopo l'esecuzione del caso d'uso*

A questa tabella vengono aggiunte una tabella per descrivere gli scenari secondari: essa contiene:

Caso d'uso	*Stesso nome del caso d'uso di cui questo è uno scenario secondario*
Scenario secondario	*Nome dello scenario secondario che andiamo a specificare*
Flusso Principale	*Descrizione testuale delle operazioni che compiono gli attori ed il caso d'uso, nonché le risposte in questo scenario*
Flussi alternativi	*Alternative al flusso principale (sia errori facilmente descrivibili che vere e proprie diramazioni del flusso principale) in questo scenario*

Se un caso d'uso è una specializzazione di un altro caso d'uso (cioè stiamo usando la generalizzazione), il titolo della prima riga diviene “Caso d'uso figlio” e si aggiunge una riga “Genitore”, subito sotto l'identificatore, in cui si indica il nome del caso d'uso che viene specializzato.

Se un caso d'uso ha dei punti di estensione, si inseriscono nel flusso, nel punto in cui possono essere eseguiti, i loro nomi racchiusi tra doppie parentesi angolari (« »).

I casi d'uso che ne estendono un altro hanno la tabella la cui prima riga è cambiata in “Caso d'uso di estensione”, viene aggiunta una riga “Estende”, subito dopo l'identificatore, in cui si indica il nome del caso d'uso esteso, ed il flusso principale viene chiamato “Segmento”.

I casi d'uso che includono altri casi d'uso indicano il punto in cui viene eseguito un altro caso d'uso (che sia nel flusso principale, in uno alternativo od in uno scenario secondario) con la “direttiva” *include(Nome del caso d'uso da eseguire)*.

È possibile aggiungere altre righe alle tabelle precedentemente descritte inserendo informazioni su

flussi impossibili o non permessi, sui messaggi scambiati tra gli attori ed il caso d'uso, e l'uso di oggetti, valori o risorse del sistema.

Nota importante: i casi d'uso sono atomici, scambiano informazioni con gli attori, ma non tra di loro.

I casi d'uso non possono essere usati (o è più difficile o "pericoloso" usarli) nel caso in cui:

- Il sistema è dominato dai requisiti non funzionali
- Il sistema ha pochi utenti
- Il sistema ha poche interfaccie

In tal caso è meglio cercare altre tecniche.

Attività e Stati

Altra documentazione che può essere prodotta, riguarda i Diagrammi di Attività e quelli degli Stati. Con i primi si modellano i flussi principali ed alternativi di ogni caso d'uso (danno una panoramica d'insieme sui flussi), mentre i secondi, meno usati in questo momento, permettono di modellare stati del sistema.

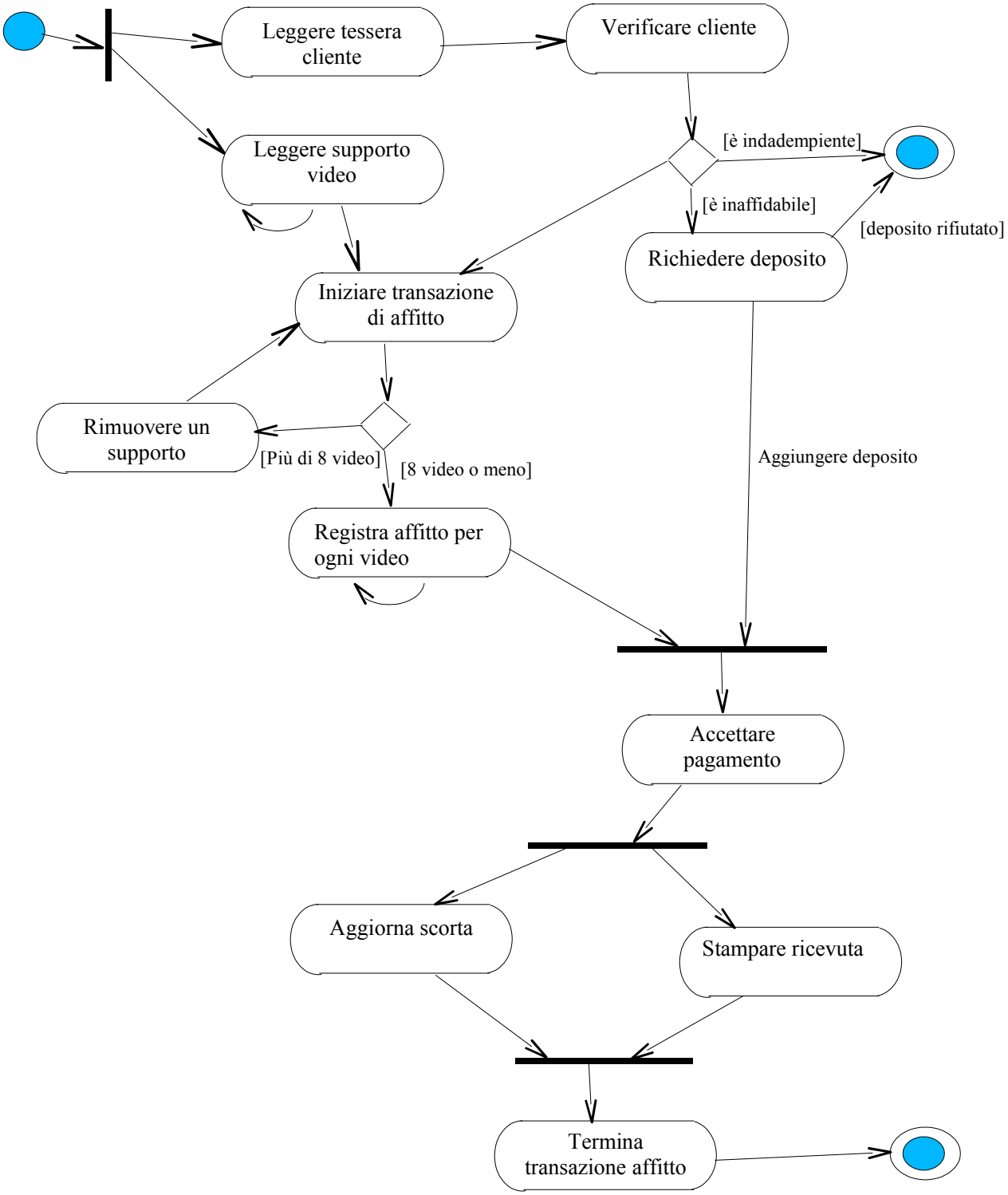
Con il primo si prendono le descrizioni dei flussi: sia quello principale che quelli alternativi sono fusi nello stesso diagramma per poter far vedere l'insieme delle azioni compiute dal caso d'uso preso in considerazione.

Per preparare un diagramma delle attività si disegna un cerchio pieno che indica il punto di inizio dell'attività; quando un caso d'uso viene attivato, si ci sposta sulla prima attività (rappresentata da un rettangolo dai bordi arrotondati), quando l'attività ha compiuto il suo compito, si ci sposta alla seconda e così via fino allo stato finale (rappresentato da un cerchio pieno inscritto in un secondo cerchio vuoto) in cui il caso d'uso termina. Nel momento in cui bisogna fare una scelta (e quindi passare ad un flusso alternativo) il collegamento avviene attraverso un rombo che permette di spezzare la linea di congiunzione tra le attività a seconda delle condizioni specificate su ogni ramo (sono racchiuse tra parentesi quadre); quando i flussi alternativi si riuniscono lo fanno attraverso un secondo rombo. Una delle potenzialità del diagramma delle attività è la possibilità di mostrare attività parallele: quando partono diverse attività in parallelo, una linea spessa spezza in più di una le linee di congiunzione tra le attività (ciò è chiamato *fork*); quando si riuniscono lo fanno attraverso una linea spessa (ciò è chiamato *join*).

Facciamo un esempio: nell'affitto video il flusso principale indica: "Un cliente dà all'impiegato la tessera ed il supporto video che vuole noleggiare. Il video e la tessera vengono letti e qualsiasi dettaglio di inadempienza o ritardo è evidenziato all'impiegato, il quale deve richiedere al cliente chiarimenti. Un cliente può prendere in affitto un massimo di otto film. Il cliente paga in contanti o carta di credito. A quel punto la scorta è aggiornata ed i film sono dati al cliente insieme alla ricevuta d'affitto. Ogni record di affitto memorizza (sotto il conto del cliente) le date di uscita e di restituzione, insieme all'identificazione dell'impiegato. Un record di affitto distinto è creato per ogni video affittato."

I flussi alternativi recitano: "Se il cliente è inadempiente, egli non può affittare nulla. Se il cliente cerca di affittare troppi film, deve rinunciare a qualcuno o non può ritirarli. Se il cliente è inaffidabile deve versare un deposito pari al costo del periodo di affitto per ciascun film."

Nell'esempio ho omesso vari altre specifiche per semplificare un po' il diagramma. Ed ecco il diagramma delle attività:



Essendo poco usato, in questo contesto, il diagramma degli stati verrà discusso successivamente.

Determinare i requisiti

N.d.A. Questo paragrafo non fa propriamente parte dello Unified Process, ma si ritiene utile per una migliore chiarificazione :)

Anche se non è possibile dare una tecnica di estrazione dei requisiti sempre valida, esistono alcune tecniche che possono aiutare:

Interviste: con queste si intendono degli incontri con il cliente; durante questi incontri vengono poste diverse domande, affinché si possa capire il punto di vista del cliente, risolvere i dubbi, e capire le sue reali necessità. Il problema riguarda il fatto che spesso un cliente ha difficoltà od è restio ad esprimere i requisiti in termini comprensibili per lo sviluppatore. Per questo è importante la capacità dell'intervistatore di comunicare e stabilire rapporti interpersonali.

Normalmente l'intervista è preceduta da un qualche tipo di documento che spiega gli obbiettivi dell'intervistatore e/o pone alcuni quesiti di base, in modo da avere una punto di partenza ed un contesto per la discussione.

Le interviste possono essere libere o strutturate (normalmente ci sono entrambi): le seconde sono una serie di domande già pronte (alcune anche a risposta chiusa), mentre con le prime le domande vengono di volta in volta formulate in base alle necessità. È meglio, tuttavia, evitare tre tipologie di domande:

- Domande nelle quali l'intervistatore esprime (direttamente od indirettamente) la propria opinione sul problema (“dobbiamo fare le cose nel modo in cui le facciamo?”)
- Domande prevenute, simili alle precedenti, eccetto che l'opinione dell'intervistatore è chiaramente espressa (“non intendi fare questo, non è vero?”)
- Domande contenenti già la risposta (“devi fare le cose in questo modo, non è vero?”)

Questionari: I questionari sono basati su dei documenti contenenti domande a risposta chiusa. A differenza dell'intervista, non c'è un colloquio tra il progettista ed il cliente. Questo permette di avere risposte in tempi brevi e di mantenere l'intervistato anonimo, ma possono nascere delle incomprensioni dovute al fraintendimento delle domande. Questo sistema raggiunge meglio i suoi scopi quando è usato come strumento in aggiunta alle interviste e le domande sono poste in maniera meno ambigua possibile.

Normalmente un questionario ha tre forme:

- *Quesiti a scelta multipla*, dove chi risponde deve scegliere una o più risposte tra un insieme di risposte date. Commenti aggiuntivi possono essere permessi.
- *Quesiti a punteggio*, dove è possibile esprimere la propria opinione su un'affermazione (ad esempio: sono pienamente daccordo, sono daccordo, neutrale, sono in disaccordo, sono totalmente in disaccordo, non so).
- *Questi con ordinamento*, dove le risposte fornite dovrebbero essere ordinate (per importanza) con numeri in sequenza, valori percentuali, od altri meccanismi di ordinamento.

Osservazioni e studio dei documenti e dei sistemi software: Queste ultime tecniche possono essere utili in certi frangenti e deleteri in altri casi.

Con l'osservazione si intende l'osservare il lavoro del/dei cliente/i al fine di scoprire le procedure che usano e di cosa hanno bisogno. Ciò è utile quando il cliente è incapace di fornire direttamente l'informazione, anche perché potrebbe avere una conoscenza frammentaria rispetto all'intero processo. Il problema è che è fastidioso essere osservati mentre si lavora e si tende a seguire procedure

molto più formali saltando qualsiasi scorciatoia (nel bene o nel male) intrapresa, oppure si tende a lavorare con maggior impegno alterando le valutazioni sulla complessità delle operazioni coinvolte.

Con lo studio dei documenti e dei sistemi software si intende lo studio dei documenti dell'organizzazione e dei sistemi che attualmente usano al fine di farsi un'idea di come sono abituati a lavorare e fargli trovare, dunque, un sistema più familiare. Il problema è che non sempre il cliente è disposto a far conoscere le proprie condizioni di lavoro e far mettere mano a documenti a volte riservati ed i sistemi usati possono essere inadatti e poco facili da usare (uno dei motivi per cui lo volevano cambiare) oppure totalmente incomprensibili a chi non ha familiarità col prodotto.

Prototipi: Sempre più usati, i prototipi non sono altro che sistemi “sporchi e veloci”. In altre parole vengono preparate interfaccie del programma con dati codificati direttamente al suo interno, con azioni già preconfigurate e non cambiabili.

Risulta utile sia per estrarre requisiti difficilmente estraibili in altri modi (ad esempio con sistemi che devono fornire nuove funzionalità di business) e per iniziare a considerare anche l'interfaccia utente fin dai primi approcci.

Il difetto principale dei prototipi è che devono essere considerati usa e getta, una volta che hanno esaurito il loro compito devono essere eliminati e non reinseriti dentro il progetto che si va a costruire, pena difficoltà di test e manutenzione. Purtroppo spesso il cliente crede che se il prototipo è pronto in breve tempo aggiungere le funzionalità reali richiederà ancor meno tempo. Infatti è tipico confondere il programma con la sua intefaccia (estremamente importante, ma non è l'unica parte del progetto).

Sviluppo cooperativo: In un certo senso alla base dello Unified Process. Lo sviluppo cooperativo richiede vari incontri con gli sviluppatori, il/i cliente/i ed in genere gli stakeholders. Ad ogni incontro bisogna discutere, approfondire, scoprire, validare, ecc. le varie fasi dello sviluppo del progetto. Se gli incontri sono ben preparati, permettono di procedere celermente; al contrario possono rallentare lo sviluppo se non si riesce a raggiungere accordi in tempi brevi.

Ricordate, infine, che ogni requisito deve essere validato, assegnato ad un contesto, capire se funzionale o non funzionale, assegnatagli una priorità, ecc. (in altre parole, il documento SRS deve essere ben compilato).

Attività di Ombrello: Stime

Durante la fase di Inception (e, come raffinamento, anche in quella di Elaboration) è importante stabilire se un progetto è fattibile o meno, quanto tempo, risorse e costi preveda, ecc. Il problema è che all’inizio non si ha chiaro né la portata, né le conoscenze necessarie e neanche tutti i requisiti. Il problema non è da poco, visto che non si possono rimandare alla fine del processo le stime necessarie.

Vediamo, come si potrebbe fare. La prima cosa da cui partire è la portata del software: Quanto è grande il progetto? Quali sono i suoi limiti? Rispondere a queste domande, non è sempre facile; tuttavia realizzare un modello di business può dare una mano a stabilire la portata del software. Ciò non basta per le stime; è importante che il capo progetto sappia, od abbia modo di avere, le risposte ad alcune domande:

- Da dove parte la richiesta di questo lavoro?
- Chi utilizzerà la soluzione?
- Quali saranno i benefici economici di una soluzione ottimale?

- La soluzione si può trovare altrove?
- Quali sono secondo il cliente le caratteristiche di un output soddisfacente?
- Quali problemi deve affrontare la soluzione?
- È possibile mostrare o descrivere l'ambiente in cui la soluzione dovrà operare?
- Ci sono questioni relative alle prestazioni o vincoli particolari che incidono sulla soluzione?
- Qual'è la persona più indicata a rispondere alle domande?
- Le risposte sono "ufficiali"?
- Ci sono altre persone che possono dare informazioni supplementari?
- Le domande sono rilevanti rispetto al problema?
- C'è qualcos'altro che bisognerebbe chiedere?
- Si stanno facendo troppe domande?

Le risposte a queste domande vengono in parte dall'analisi dei requisiti (fulcro di questa attività), altre da domande poste al cliente. Le ultime tre si rivolgono al capo progetto e dovrebbero farlo riflettere sul numero e sulla rilevanza delle domande che dovrà porre al cliente.

Una volta stabilita la portata del software, è necessario stabilire se esso è fattibile, quanto costa, quanto tempo ci vuole, le risorse necessarie, ecc. Purtroppo la maggior parte di queste cose richiede una certa esperienza, in quanto solo col tempo si è in grado di capire che problemi possa avere un team, le sue conoscenze, quanto impiegano a far qualcosa, ecc. Vediamo, comunque, come procedere.

La prima cosa che vediamo sono le risorse; è vero che esse vengono assegnate solo una volta che il progetto è stato suddiviso nelle sue parti fondamentali e si è visto di cosa c'è bisogno per ogni parte, tuttavia sapere cosa abbiamo a disposizione è utile, se non fondamentale. Le risorse che si hanno a disposizione sono le risorse umane, le risorse software riutilizzabili e le risorse ambientali. Sulle prime, a parte quanto detto per i team coesi, bisogna conoscere la loro collocazione nell'azienda (manager, ingegnere senior, ecc.) nonché la specializzazione (telecomunicazioni, database, client/server, ecc.). Le seconde le possiamo dividere in:

- **Componenti di serie**: software esistente, acquistabile dall'esterno o sviluppato internamente in progetti passati. Questi componenti sono pronti all'uso e sono già stati certificati.
- **Componenti già sperimentati**: specifiche, progetti, codice o dati sviluppati in progetti passati e simili al software che si deve realizzare. I membri del team hanno maturato una piena esperienza nell'area applicativa propria di tali componenti. Di conseguenza le modifiche da apportare comportano rischi relativamente bassi.
- **Componenti parzialmente sperimentati**: specifiche, progetti, codice o dati per collaudi sviluppati per progetti passati, affini al software che si deve realizzare, ma soggetti a modifiche sostanziali. I membri del team hanno un'esperienza limitata nell'area applicativa propria di tali componenti; di conseguenza le modifiche da apportare comportano rischi sensibili.
- **Nuovi componenti**: Componenti software che devono essere realizzati specificatamente per il progetto in corso.

Normalmente, quando si ha a che fare con questi componenti, bisognerebbe seguire questi passi:

1. Se esistono componenti di serie che soddisfano i requisiti del progetto, conviene acquistarli. Il

costo di acquisto ed integrazione di componenti di serie è quasi sempre inferiore al costo di sviluppo di software equivalente. Per di più i rischi sono relativamente bassi.

2. Se si dispone di componenti già sperimentati, i rischi associati al loro adattamento ed integrazione sono in genere accettabili. Il piano del progetto deve tener conto del loro impiego.
3. Se si dispone di componenti parzialmente sperimentati, il loro uso nel progetto va analizzato. Qualora, prima dell'integrazione ad altri componenti, essi richiedano modifiche rilevanti, si deve procedere con molta attenzione, poiché il costo dell'adattamento (a causa degli eccessivi side-effect che può generare) può superare quello di sviluppo di nuovi componenti.

Per quanto rigurada le risorse ambientali esse riguardano tutto l'hardware ed il software, nonché i luoghi di lavoro materiali, che assistono allo sviluppo. Essi possono comprendere tool, hardware specializzato, posti di lavoro particolari per il collaudo, ecc. Conoscere queste risorse è importante per evitare di mandare in parallelo attività che in parallelo non possono andare a causa delle limitazioni ambientali (se ad esempio c'è bisogno di una postazione speciale per il collaudo, e ne abbiamo una sola, non si possono collaudare conteporaneamente due componenti).

La parte principale dell'attività di ombrello del calcolo delle stime, consiste del scomporre i requisiti in problemi più piccoli e facilmente gestibili e per ognuno di essi applicare alcune misurazioni viste precedentemente: LOC, Function Points, Feature Points, Object Points, COCOMO. Ovviamente ognuna di queste misure è effettuata in base ad una stima ed andrà tenuta sotto controllo durante l'evolversi del progetto. Una volta prese queste misure, si vede quanto costa realizzare un KLOC, FP od altro, quanto tempo (in mesi uomo) ci vuole per realizzarlo e quindi quanto costa il progetto e quanti mesi-uomo ci vogliono.

Un altra tecnica consiste nel fare la stima basandola sulle fasi del processo: in altre parole si usa una tabella nelle cui righe si mettono le funzioni e nelle colonne le fasi; le colonne sono a loro volta suddivise nelle attività. All'incrocio tra riga e colonna vengono messi i mesi-uomo necessari a terminare un attività per una determinata funzione. La somma ci darà il totale dei mesi-uomo neceessari al progetto. Sapendo quanto costa un mese-uomo, sappiamo quanto verrà a costare il progetto.

Un ultimo appunto: tutte le stime dovrebbero dare risultati circa uguali (variazioni intorno al 20%) su tutte le variabili (costi e tempi in primis), altrimenti significa che: o non si è capito la portata del progetto, o non si sono capiti i requsiti!

La scelta tra sviluppo ed acquisto

Come abbiamo detto prima, ci sono dei casi in cui bisogna scegliere se acquistare un componente o svilupparlo.

Espandiamo il concetto a tutto il progetto: quando si inizia un nuovo progetto esistono quattro possibilità che possono essere applicate a tutto il progetto o solo ad una parte:

1. Sviluppo da zero
2. Servirsi di componenti già sperimentati o parzialmente sperimentati
3. Acquistare un prodotto esistente ed adattarlo
4. Affidare lo sviluppo ad un altro produttore (outsourcing)

La scelta può dipendere da vari fattori; le indicazioni qui di seguito possono dare una mano:

1. Preparare una specifica delle funzionalità e delle prestazioni attese. Quando è possibile definire caratteristiche misurabili.

2. Stimare costi e tempi dello sviluppo in proprio.
3. Selezionare le tre o quattro applicazioni candidate che meglio soddisfano i requisiti
4. Selezionare una serie di componenti software riutilizzabili atti a costruire l'applicazione richiesta
5. Selezionare i tre o quattro candidati per l'outsourcing che meglio soddisfano i requisiti
6. Vedere se la data di consegna del prodotto è anteriore alla data di completamento del software sviluppato in proprio.
7. Vedere se il costo di acquisto sommato al costo di adattamento è inferiore al costo dello sviluppo in proprio
8. Vedere se il costo del supporto esterno (ad esempio il contratto di manutenzione) è inferiore al costo del supporto interno.
9. Valutare ogni pacchetto, componente o soluzione proposta sulla base della qualità dei prodotti collaudati, del supporto al cliente, del controllo del prodotto, della reputazione, dei costi e così via.
10. Consultare gli altri stakeholder e chiedere il loro parere.

Per calcolare i costi di ogni opzione bisogna anche vedere la probabilità che possano verificarsi problemi; così per la costruzione in proprio bisogna stabilire il costo è la probabilità che lo sviluppo sia semplice o complesso. Per il riutilizzo bisogna vedere il costo e la probabilità che le modifiche sia secondarie od importanti ed in questo caso il costo e la probabilità che queste modifiche siano semplici o complesse. Per l'acquisto bisogna vedere il costo e la probabilità che le modifiche da apportare siano modifiche secondarie od importanti. Per l'outsourcing bisogna vedere il costo e la probabilità che vengano richieste delle modifiche o vada bene così com'è.

Ovviamente tale "albero di decisione" può essere tanto più profondo quante più opzioni si hanno a disposizione.

Il costo atteso è pari alla somma delle varie probabilità di scegliere un cammino per il costo stimato di quel cammino.

Attività di ombrello: Analisi dei Rischi

Durante tutto il progetto possono verificarsi vari problemi. Bene, un rischio è un problema potenziale, che può verificarsi o meno. È importante che fin dalle prime fasi si ponderi bene i rischi che il progetto protrebbe presentare e provvedere a gestirli. L'analisi dei rischi serve proprio a questo.

Definiamo un rischio un "qualcosa" che presenta due caratteristiche:

- *Incertezza*: il rischio può avvenire o no; non ci sono rischi con la probabilità del 100% (e se così fosse sarebbe un vincolo e non un rischio)
- *Perdita*: se il rischio si realizza, ne seguono conseguenze indesiderate o perdite

Una volta stabilito cos'è un rischio è necessario stabilirne il tipo. I rischi possono essere:

- *Noti*: sono quelli suscettibili di essere scoperti dopo un attento esame del piano del progetto, dell'ambiente aziendale e tecnico in cui si svolge lo sviluppo e di altre informazioni rilevanti (ad esempio data di consegna non realistica, mancanza di requisiti documentati o della dichiarazione dei portata del software, ambiente di sviluppo inadeguato).
- *Prevedibili*: sono quelli dedotti dall'esperienza (ad esempio alto ricambio del personale, difficol-

tà di comunicazione col cliente, diminuzione dell'impegno nel lavoro di fronte a richieste di manutenzione).

- *Imprevedibili*: sono quelli che possono avverarsi, e spesso lo fanno, ma sono estremamente difficili da individurare in anticipo.

Ognuno di questi può essere a sua volta diviso in:

- *Rischio progettuale*: mette a repentaglio il piano del progetto. Questo significa che se il rischio si realizza, molto probabilmente i tempi del progetto si allungheranno ed i costi cresceranno. I rischi progettuali comprendono problemi potenziali relativi al budget, alla tabella dei tempi, al personale (scelta ed organizzazione), alle risorse, al cliente, ai requisiti e gli effetti relativi sul progetto, alla complessità, alla dimensione ed al grado di incertezza del progetto.
- *Rischio Tecnico*: mette a repentaglio la qualità e la puntualità del prodotto. Nel caso in cui un rischio tecnico si avveri, l'implementazione può diventare difficile od impossibile. I rischi tecnici individuano problemi potenziali di progettazione, implementazione, interfacciamento, verifica, manutenzione, ambiguità delle specifiche, incertezza tecnica, obsolescenza tecnica, ed impiego di tecnologie di avanguardia. I rischi tecnici si avverano perché il problema è più difficile del previsto.
- *Rischio aziendale*: minaccia la sopravvivenza del prodotto. Questa categoria comprende rischi che mettono in pericolo il progetto ed il prodotto. Fra i più rilevanti: realizzare un prodotto od un sistema di alta qualità, ma che nessuno vuole (rischio di mercato); realizzare un prodotto che non rientra più nella strategia aziendale (rischio strategico); realizzare un prodotto che il settore commerciale non è in grado di vendere; perdere l'appoggio della direzione aziendale, per via dei mutamenti di interesse o delle persone (rischio direttivo); perdita di finanziamenti o di personale (rischio finanziario).

Una volta detto cos'è un rischio, e di che tipo può essere, vediamo come individuarli, valutarli e gestirli.

L'individuazione dei rischi è il tentativo sistematico di specificare le minacce al piano di un progetto (stime, tempi, uso delle risorse e così via). L'individuazione dei rischi noti e prevedibili è il primo passo verso la prevenzione, quando possibile, e la loro gestione, quando necessario.

Per ognuno dei tipi di rischio che abbiamo visto sopra, possiamo distiguere tra rischi *generici* e *specifici*. I primi sono comuni a qualunque progetto software; i secondi si possono individuare solo a patto di avere una chiara visione della tecnologia, delle persone e dell'ambiente specifici del progetto in esame.

Per stabilire i rischi generali bisogna rispondere ad alcune domande:

- I top-manager della società di software e del cliente hanno formalizzato la realizzazione del progetto?
- Gli utenti finali hanno accettato positivamente la realizzazione del sistema o prodotto?
- I requisiti sono stati ben compresi?
- I clienti sono stati interessati appieno nella realizzazione dei requisiti?
- Gli utenti finali hanno attese realistiche?
- La portata del progetto è stabile?
- Il team di ingegneria del software ha le conoscenze adeguate?

- I requisiti del progetto sono stabili?
- Il team del progetto ha esperienza nel campo della tecnologia da implementare?
- Il numero di persone che lavorano nel team del progetto è adeguato allo svolgimento del lavoro?
- Tutte le persone interessate (i clienti/utenti) sono daccordo sull'importanza del progetto e sulla necessità che venga realizzato il sistema od il prodotto?

Se una di queste domanda ha una risposta negativa, occorre svolgere gli atti di riduzione, monitoraggio e gestione dei rischi. Il grado di rischio del progetto è direttamente proporzionale al numero di risposte negative a queste domande.

Per quanto riguarda i requisiti specifici, si devono esaminare il piano di progetto e la dichiarazione di portata e rispondere alla domanda: "Quali aspetti peculiari del prodotto possono minacciare il piano del progetto?"

Al fine di individuare i rischi è utile stilare un **catalogo dei rischi**, suddiviso in varie categorie di rischi noti e prevedibili:

- **Dimensione del prodotto**: rischi associati alla dimensione complessiva del software da costruire o modificare.
- **Effetti commerciali**: rischi associati ai vincoli imposti dal management od anche dal mercato.
- **Caratteristiche del cliente**: rischi associati alla sofisticazione del cliente ed alla capacità dello sviluppatore di comunicare tempestivamente con esso.
- **Definizione del processo**: rischi associati alla precisione con cui il processo di sviluppo è definito ed è seguito dall'organizzazione.
- **Ambiente di sviluppo**: rischi associati alla disponibilità ed alla qualità degli strumenti utilizzati nello sviluppo del prodotto.
- **Tecnologia**: rischi associati alla complessità del sistema da realizzare ed alla novità della tecnologia racchiusa nel sistema.
- **Dimensione ed esperienza dello staff**: rischi associati all'esperienza complessiva, tecnica e progettuale degli ingegneri impegnati nello sviluppo.

L'implementazione del catalogo dei rischi è lasciata al team: alcuni potrebbero mettere come prima cosa la divisione tra rischi noti e probabili, sotto di essi mettere le categorie, sotto le categorie mettere i rischi individuati, con il relativo tipo (tecnico, aziendale, strategico, ecc.); altri potrebbero mettere la suddivisione per categorie e sotto di queste i relativi rischi (con indicazioni sul tipo e sul fatto se è noto o probabile); altri ancora potebbero mettere in cima il tipo di rischio ed a seguire le categorie ed i rischi associati (con l'indicazione se noto o probabile); e così via. Anche la descrizione dei rischi è qui lasciata al team che potrebbe decidere di elencare i rischi come tratti peculiari, come risposte ad alcune domande od altro ancora.

Una volta avuto il catalogo dei rischi, ogni rischio va stimato: cioè bisogna conoscere la probabilità che il rischio sia reale e le conseguenze dei problemi che sorgono qualora il rischio si realizzi.

Tabella dei rischi

La tecnica che viene presentata si basa, oltre che sul catalogo dei rischi, anche su una strategia suggerita dalla U.S. Air Force. Questa strategia vuole che il capo progetto individui i fattori di rischio che incidono sui componenti di rischio del software: prestazioni, costi, supporto e tempi.

Categoria \ Componenti		Prestazioni	Supporto	Costi	Tempi
Catastrofico	1	La non soddisfazione dei requisiti provoca il fallimento della missione		Ritardi e aumento dei costi di valore superiore a mezzo miliardo	
	2	Da degrado significativo a non raggiungimento delle prestazioni tecniche	Software inadeguato o non sostenibile	Deficit finanziario significativo, probabile superamento del budget	Data di consegna impossibile da rispettare
Critico	1	La non soddisfazione dei requisiti provoca un degrado delle prestazioni tale da rendere dubbia la riuscita della missione		Ritardi operativi e aumento dei costi nell'ordine di 500 milioni	
	2	Perdita di prestazioni tecniche	Ritardi secondari nelle modifiche	Deficit finanziario, possibile superamento del budget	Possibile slittamento della data di consegna
Marginale	1	La non soddisfazione dei requisiti provoca degrado della missione secondaria		Ritardi recuperabili e aumento dei costi nell'ordine di 1-100 milioni	
	2	Perdita, da minima a piccola, di prestazioni tecniche	Necessità di supporto rapido	Risorse finanziarie sufficienti	Tabella dei tempi realistica
Trascurabile	1	La non soddisfazione dei requisiti provoca inconvenienti di poco conto o effetti non operativi		Un errore provoca effetti secondari su tempi e costi; valore inferiore a 1 milione	
	2	Nessuna perdita di prestazioni tecniche	Facilità di supporto	Budget non utilizzato in pieno (possibile)	Possibilità di consegna anticipata

Note: (1) Conseguenze potenziali di errori o guasti non rilevati del software.
(2) Conseguenze potenziali del mancato raggiungimento dei risultati attesi.

Illustrazione 1 Componenti/Categoria

Ai fini della presente trattazione, i componenti di rischio sono definiti nel modo seguente:

- **Rischi sulle prestazioni**: il grado di incertezza sul fatto che il prodotto soddisfi i requisiti e sia adatto all'uso previsto.
- **Rischi sui costi**: il grado di incertezza sul fatto che il budget venga rispettato.
- **Rischi sul supporto**: il grado di incertezza sul fatto che il software risultante sia facile da correggere, adattare, potenziare.
- **Rischi sui tempi**: il grado di incertezza sul fatto che i tempi previsti siano rispettati e che il prodotto sia consegnato puntualmente.

Gli effetti di ogni fattore di rischio sono classificati in quattro categorie: trascurabile, marginale, critico e catastrofico. La figura denominata Componenti/Categoria indica le conseguenze potenziali di errori (righe contrassegnate da 1) o della mancanza realizzazione di un risultato desiderato (righe contrassegnate da 2). La categoria dell'effetto va scelta sulla base della descrizione che meglio corrisponde al caso in esame. Nella figura la valuta è in lire.

Vediamo a questo punto questa semplice tecnica di stima: la **tabella dei rischi**.

Si comincia stilando un elenco di tutti i rischi, non importa quanto remoti, nella prima colonna della tabella (i rischi vengono, solitamente, presi dal catalogo dei rischi). Ogni rischio viene suddiviso per categoria nella seconda colonna (qui per categoria intendiamo quelle del catalogo dei rischi). Nella successiva colonna della tabella si indica la probabilità che si verifichi il rischio. Quindi viene considerato l'impatto di ciascun rischio. Ogni componente del rischio viene definito utilizzando la categorizzazione presentata nella figura precedente e viene determinata una categoria d'impatto. Viene calcolato il livello medio di ognuno dei quattro componenti del rischio (prestazioni, supporto, costi, tempi) per determinare il valore globale d'impatto (si può usare il valore globale d'impatto quando uno dei componenti di rischio è il più significativo per il programma).

Dopo aver compilato le prime quattro colonne, si ordina la tabella secondo la probabilità e le conseguenze, ponendo in alto i rischi con alta probabilità ed alto livello di conseguenze ed in basso quelli con bassa probabilità. Il capo progetto esamina la tabella ordinata e traccia una linea di demarcazione orizzontale: solo i rischi sopra la linea riceveranno ulteriore attenzione. I rischi sotto la linea sono nuovamente valutati al fine di produrre una scala di precedenza secondaria.

Si devono trattare tutti i rischi sopra la linea di demarcazione. Nella tabella c'è una quinta colonna intestata con RMMM o RIS. Questa colonna contiene un puntatore ad un piano di riduzione, sorveglianza e gestione dei rischi (Risk Mitigation, Monitoring and Management Plan o RMMM) od alternativamente ad una raccolta di note informative sui rischi (Risk Information Sheet o RIS). Vedremo come è composto un RMMM o un RIS più avanti.

La probabilità di un rischio la si calcola in maniera empirica od attraverso esperienze precedenti. Una mano può darla classificando i rischi inzialmente come impossibile, improbabile, normale, probabile, frequente e poi dando ad essi un valore numerico (impossibile=minore del 10%; improbabile=10-25%; normale=25-50%; probabile=50-75%; frequente=maggiore del 75%).

Tre fattori incidono sulle conseguenze del realizzarsi di un rischio: la sua natura, la sua portata ed il tempo. La natura di un rischio indica quali problemi possono sorgere se il rischio si avvera. La portata di un rischio combina la gravità (quanto è serio) con la sua estensione (quanta parte del progetto è interessata o quanti clienti possono essere danneggiati). Il fattore temporale comprende il momento e lo spazio di tempo in cui gli effetti sono rilevabili.Quando si compila la tabella dei rischi bisogna tener conto di questi tre fattori, quando si decide la categoria d'impatto. Bisogna inoltre determinare, per ogni rischio, l'**esposizione al rischio**, **RE**, nel seguente modo:

RE=P x C

Dove P è la probabilità che si verifichi un rischio e C è il costo del progetto nel caso in cui si verifichi il rischio. L'esposizione totale calcolata per tutti i rischi più elevati della tabella dei rischi può fornire un metodo per modificare la stima dei costi finali del progetto. Può essere anche utilizzata per prevedere la probabilità di dover aumentare le risorse umane necessarie nello sviluppo del progetto.

Questa tecnica deve essere applicata nel corso di tutto il progetto al fine di stabilire se le mutate circostanze incidono sulla sua probabilità e sugli effetti. In seguito a questa operazione può essere ne-

cessario inserire nuovi rischi in tabella, eliminarne altri non più rilevanti e modificare la collocazione dei restanti.

A questo punto abbiamo una famiglia di terne $[r_i;l_i;x_i]$ che ci indicano rispettivamente i rischi, la probabilità che si verifichino ed il relativo impatto. Da questi bisogna ricavare i *livelli di riferimento* cioè un insieme di valori che indichino quando una delle componenti del rischio supera un livello limite da noi stabilito. Da questi ricavare il **punto di rottura** cioè il punto in cui uno o più livelli di riferimento ha superato la soglia di attenzione troppo marcatamente: proseguire il progetto porterà al fallimento!

Una piccola nota: come detto la descrizione dei rischi può essere fatta come più aggrada. Tuttavia, arrivati a questo punto è meglio essere più precisi e riformulare i rischi in formato **CTC** (Condizione – Transizione – Conseguenza). In pratica il rischio viene definito nella seguente forma:

Dato che <condizone> allora vi è una (probabile) <conseguenza>

Gestione del Rischio

Una volta che abbiamo stabilito i rischi, le loro conseguenze, i livelli di rferimento, il punto di rottura, ecc. bisogna decidersi a gestirli. Per far questo si usano tre documenti: il piano di riduzione dei rischi, la vigilanza sui rischi, la gestione dei rischi e la pianificazione delle urgenze.

Il **piano di riduzione dei rischi** indica quali sono le strategie da applicare per ridurre la probabilità che il rischio si verifichi. Ad esempio se il rischio è un alto turn-over del team, con probabilità del 70% di accadere con conseguenze critiche, un piano di riduzione dei rischi potrebbe comprendere:

- riunioni con lo staff tese a determinare le cause del ricambio (ad esempio condizioni di lavoro insoddisfacenti, salari bassi, mercato del lavoro competitivo);
- azioni volte ad alleviare queste cause su cui la direzione può intervenire prima dell'avvio del progetto;
- dopo l'avvio del progetto, supporre che ci sarà alto ricambio e predisporre le misure atte a garantire ugualmente la continuità del lavoro;
- organizzare il team in modo che le informazioni sull'attività di sviluppo siano ampiamente distribuite;
- definizioni di standard per la documentazione e di meccanismi che garantiscano la puntuale produzione dei documenti;
- revisioni collettive di ogni aspetto del progetto, così che più persone siano al corrente della situazione;
- predisposizione di membri di riserva per le posizioni tecnicamente cruciali.

La **vigilanza sui rischi** è un'attività volta a tener sotto controllo i fattori che possono indicare l'avvicinarsi di un rischio, stabilirne la soglia di attenzione e quando attuare le manovre correttive. Nell'esempio, i fattori da sorvegliare sono i seguenti:

- atteggiamento generale dei membri del team e risposta alle pressioni;
- grado di coesione del team;
- relazioni personali fra i membri del team;
- problemi potenziali relativi ai compensi ed ai benefici;
- disponibilità di posti di lavoro alternativi, interni all'azienda od esterni;

Inoltre bisogna controllare che tutte le misure prese per ridurre il rischio siano efficaci. Ad esempio le “definizioni di standard per la documentazione e di meccanismi che garantiscano la puntuale produzione dei documenti” sono un modo di garantire la continuità anche nel caso in cui un membro importante lasci il progetto. Il capo progetto deve verificare meticolosamente che ogni documento prodotto sia coerente e che offra informazioni necessarie ad un eventuale nuovo memebro costretto ad unirsi al team del progetto.

La vigilanza sui rischi contiene anche un punto: quello che a me piace definire come “Massa Critica”. Quando gli “indicatori” raggiungono questo punto significa che il rischio è certo e deve passare la palla alla **gestione dei rischi e pianificazione delle urgenze**. Questo documento presume che i tentativi di prevenzione siano falliti e che il rischio si sia avverato. Indica come comportarsi in tal caso. Nell’esempio: si dispone di personale di riserva, le informazioni sono ampiamente documentate e distribuite tra i membri del team; il capo progetto può temporaneamente concentrare le risorse (e ritoccare la tabella dei tempi) sulle funzioni senza problemi di personale, dando modo ai nuovi venuti di inserirsi gradualmente. Tutti coloro che stanno lasciando il team sono invitati a sospendere il lavoro produttivo ed a dedicare il tempo rimasto a trasmettere le informazioni utili.

RMMM e RIS

I documenti sopracitati vengono formalmente messi all’interno del piano di riduzione, sorveglianza e gestione dei rischi (Risk Mitigation, Monitoring and Management Plan o RMMM) od alternativamente in una raccolta di note informative sui rischi (Risk Information Sheet o RIS).

Nota: il RIS può anche essere una specifica del RMMM.

La struttura di un documento RMMM è così composta:

1.0 Introduzione

Questa sezione fornisce un vista d’insieme sul piano di riduzione, sorveglianza e gestione dei rischi.

1.1 Scopo ed intenzioni dell’attività del RMMM

Una descrizione generale del fuoco del RMMM includendo obbiettivi e responsabilità organizzative.

1.2 Ruolo organizzativo della gestione del rischio

Descrizione di chi ha la responsabilità per la gestione del rischio.

2.0 Rischi del progetto

Questa sezione descrive tutti i rischi del progetto conosciuti.

2.1 Tabella del rischio

Una presentazione di tutti i rischi, probabilità ed impatti.

2.1.1 Descrizione del rischio *m*

Il rischio m è descritto con le sottocondizioni rilevanti. La sezione 2.1.1 è ripetuta per ogni rischio m.

2.1.2 Probabilità ed impatto del rischio *m*

La probabilità e l’impatto del rischio m descritta nella sezione 2.1.2 è ripetuta per ogni rischio m.

2.2 Raffinamento del rischio

I rischi con alta probabilità / alto impatto sono raffinati usando l'approccio CTC.

3.0 Piano di riduzione, sorveglianza e gestione dei rischi

Questa sezione discute l'RMMM per ogni rischio. Questa sezione può avere riferimenti a fogli RIS anziché avere tutte le descrizioni al suo interno.

3.1 Riduzione del Rischio per il rischio *m*

Come evitiamo il rischio m? La sezione 3.1 è ripetuta per ogni rischio m.

3.2 Monitoraggio del Rischio per il rischio *m*

Quali condizioni dobbiamo monitorare per determinare se il rischio m sta divenendo più o meno probabile? La sezione 3.2 è ripetuta per ogni rischio m.

3.3 Gestione del Rischio per il rischio *m*

Che piani di emergenza mettiamo nel programma nel presupposto che il rischio m accadrà? La sezione 3.3 è ripetuta per ogni rischio m.

4.0 Condizioni speciali

Una discussione sulle circostanze speciali che possono innescare i rischi critici di progetto e le azioni richieste che dovrebbero essere intrapese quando queste condizioni accadono.

Una nota informativa sui rischi (RIS) è invece così strutturata:

ID del Rischio:

Ogni nota informativa del rischio è creata con un unico ID. Questo ID serve per collegamenti ad altre RIS o per collegarsi al RMMM

Titolo del Rischio:

Il titolo del rischio dovrebbe essere il nome del rischio che è stato indentificato del progetto software. Di conseguenza, il titolo del rischio deve essere breve ma sufficientemente descrittivo che i membri del team del progetto software siano in grado di identificarlo velocemente.

Data Identificazione:

La data con la quale il rischio è stato inizialmente identificato. Utile per catalogazione.

Probabilità:

Questo valore indica la relativa probabilità che il rischio del progetto si verifichi.

Priorità:

La priorità è il tasso di urgenza che è dato ad un rischio comparato con altri rischi nello stesso progetto. Il valore può essere: Basso, Medio, Alto, Urgente.

Impatto:

L'impatto è il tasso che indica un possibile effetto sul progetto software nel caso in cui il rischio avvenga. Il valore può essere:

Trascurabile
Nessun effetto sul progetto.

Marginale
Ha impatto sul tempo necessario al completamento del progetto ma non nel lungo termine.
Critico
Il tempo necessario al completamento del progetto richiede pesanti modifiche per recuperare dal rischio.
Catastrofico
Il progetto è definitivamente compromesso e non può essere completato.

Descrizione del rischio:

La descrizione del rischio descrive brevemente il rischio in modo più dettagliato rispetto al titolo. Di conseguenza il rischio dovrebbe essere descritto in termini che ogni membro del team del progetto software sia in grado di capirlo.

Raffinamento/Contesto:

Questa sezione del RIS dovrebbe essere usata per ricapitolare le circostanze, le risorse ed altro interessati dal rischio, mentre la descrizione raffinata del rischio è divisa in sotto-condizioni.

Sotto-condizioni:

Ogni sotto-condizione nella sezione Raffinamento/Contesto dovrebbe descrivere una condizione che contribuisce alla causa del rischio.

Riduzione/Monitoraggio:

La sezione Riduzione/Monitoraggio del RIS presenta una lista di proposte di azioni per eliminare facilmente o prevenire il dato rischio. Ogni azione è riferita come un Passo.

Passi:

Un passo individuale della Riduzione/Monitoraggio dovrebbe essere presa dal team di sviluppo per eliminare o prevenire un rischio, prima che esso avvenga.

Gestione:

La sezione di gestione dà i dettagli su come occuparsi di un rischio se si trasforma in una realtà. La sezione della gestione include due oggetti: il piano di Contingenza e l'attivatore.

Piano di contingenza:

Il piano di contingenza è un piano dettagliato per il team del progetto che agisce nel caso il rischio descritto avvenga.

Attivatore:

L'attivatore del piano di contingenza identifica l'evento che indica che il rischio si è materializzato. L'attivatore può servire da prova del tornasole quando si decide se promulgare il piano di emergenza.

Stato corrente:

La stato corrente è definito da una lista di aggiornamenti che descrivono i cambiamenti di ogni stato del rischio in termini di cambio di probabilità, impatto, conseguenze, ecc. Ogni aggiornamento di stato deve contenere una data ed una descrizione.

Data stato:

La data di quando lo stato è cambiato.

Descrizione dello stato:

Descrizione dello stato corrente o dell'azione presa in riferimento al rischio identificato.

Responsabile:

Il nome del membro del progetto che ha compilato il RIS.

Assegnato:

Il nome del membro del progetto che è responsabile di esaminare il rischio e raccomandare le riduzioni.

Attività di ombrello: Pianificazione

Un'altra attività che viene svolta nella fase di Inception, ma che prosegue per tutto l'arco del progetto, è la pianificazione. La pianificazione consiste nel trovare un quadro di stima per l'assegnazione delle varie risorse in un calendario, cioè stabilire settimana per settimana, mese per mese o quanto serve, quali sono le risorse assegnate ad una specifica parte del progetto. Normalmente la pianificazione è aggiornata periodicamente per riflettere cambiamenti e/o ritardi vari. La pianificazione deve stabilire anche un "Miglior Caso" (quando non si presenta nessun problema) ed un "Peggior Caso" (quando si presentano tutti i problemi). Da questi due è possibile stabilire un "Caso Realistico" (praticamente una media tra i due).

Prima di procedere nel dettaglio vediamo perché il software è in ritardo. I motivi possono essere:

- Scadenze irrealistiche (esterne)
- Requisiti mutati non riflessi nel calendario
- Sottostima in buona fede
- Rischi non considerati...
  - → Difficoltà tecniche impreviste
  - → Difficoltà umane
  - → Cattiva comunicazione
  - → Difetto nel management
- e così via...

Gli ultimi due punti (Sottostima in buona fede, Rischi non considerati) sono stati trattati precedentemente. Il secondo punto (Requisiti mutati non riflessi nel calendario) è il motivo per cui questa è un'attività di ombrello. Il primo punto è tra i più difficili da risolvere, in quanto la gente non sempre ha la pazienza di aspettare e non conosce le difficoltà inerenti al progetto. D'altronde senza scadenze il progetto non finerebbe mai. Per gestire le scadenze irrealistiche si può: eseguire stime dettagliate basate su dati storici in modo da tirar fuori una stima dei tempi il più realistica possibile; se non è possibile dilatare la data di scadenza, sviluppare un modello incrementale che fornisce le funzionalità critiche entro la scadenza originale. Avuti questi due bisogna spiegare la stima al cliente ed offrire l'alternativa incrementale "sviluppata".

Principi fondamentali

Come per le altre aree dell'ingegneria del software, anche per la pianificazione temporale si possono descrivere alcuni principi fondamentali:

- **Ripartizione**: Un progetto dev'essere ripartito in attività e compiti di dimensioni ragionevoli. A tale scopo si scompongono sia il prodotto che il processo.
- **Interdipendenza**: Occorre determinare le dipendenze reciproche tra le attività ed i compiti in cui è stato ripartito il progetto, alcuni compiti possono essere svolti in sequenza, altri in parallelo. Alcune attività non possono nemmeno cominciare finché non è disponibile un "semilavorato" prodotto da un'altra attività, altre sono fra di loro indipendenti.
- **Assegnazione del tempo**: Ad ogni compito si deve assegnare un certo numero di "unità di lavoro" (ad esempio giorni-uomo). Inoltre ad ogni compito si devono attribuire la data di inizio e quella di fine, in funzione delle interdipendenze e della modalità, a tempo pieno o parziale, con cui è svolto
- **Validazione dell'assegnazione**: Ad ogni progetto è assegnato un numero definito di membri dello staff. Al momento dell'assegnazione dei tempi, il capo progetto deve accertarsi di non attribuire più persone di quelle disponibili in ogni istante.
- **Responsabilità definite**: Ogni compito deve essere affidato ad un membro del team.
- **Risultati definiti**: Ogni compito deve produrre un risultato predefinito.
- **Punti di controllo (o pietre miliari)**: Ad ogni compito o gruppo di compiti si deve associare un punto di controllo del progetto. Un punto di controllo è compiuto quando la qualità di uno o più semilavorati è stata esaminata ed approvata.

Ciascuno dei principi appena citati deve essere applicato ad ogni aggiornamento della tabella dei tempi.

Relazione tra personale e lavoro

La relazione tra personale e lavoro è non lineare: il tempo supplementare di comunicazione, cresce con l'aumentare del personale:

E = LOC^3/(P^3*t^4)

Distribuzione del carico di lavoro

Le tecniche di stima descritte precedentemente forniscono una previsione delle unità di lavoro (mesi-uomo) necessari a completare lo sviluppo di un prodotto software. Nello stabilire la distribuzione del carico di lavoro tra le fasi di definizione e sviluppo, si raccomanda in genere di seguire la regola detta del "40-20-40": il 40% dell'impegno è assegnato all'analisi ed alla progettazione e la stessa percentuale è attribuita ai collaudi finali, mentre il 20% dell'impegno è dedicato alla scrittura del codice.

Definizione dei compiti di un progetto software

Più volte abbiamo detto che un problema deve essere scomposto in un insieme di compiti necessario a risolverlo. Purtroppo non esiste un insieme di compiti valido in assoluto: ciò che può essere proficuo per un progetto grande, potrebbe rivelarsi un disastro in un progetto piccolo e viceversa. Perciò un progetto software per essere efficace deve definire una collezione di insiemi di compiti, ciascuno destinato a soddisfare le necessità dei vari tipi di progetto.

1. Un *insieme di compiti* è una collezione di compiti, lavori e punti di controllo necessari a portare a termine un progetto. Esso deve essere selezionato in modo da imporre una disciplina sufficiente a garantire la qualità del software, senza allo stesso tempo sovraccaricare il team di compiti super-

flui.

Gli insiemi di compiti devono soddisfare tipi diversi di progetto e gradi diversi di rigore. Non è facile compilare una classificazione completa, ma la maggior parte dei produttori di software affronta progetti delle categorie seguenti.

1. *Progetti di innovazione*, intrapresi allo scopo di esplorare nuove idee o di applicare una nuova tecnologia
2. *Progetti di sviluppo di una nuova applicazione*, intrapresi a seguito di richieste specifiche da parte di un cliente
3. *Progetti migliorativi*, quando un prodotto esistente è sottoposto a modifiche sostanziali che possono riguardare le funzioni, le prestazioni o le interfacce osservabili dagli utenti
4. *Progetti di manutenzione*, che correggono, adattano o estendono un prodotto esistente in modi non direttamente percepibili da un utente.
5. *Progetti di ristrutturazione*, intrapresi allo scopo di ristrutturare, in tutto od in parte, un sistema esistente.

Anche all'interno di un tipo assegnato, vari fattori influenzano la scelta dell'insieme di compiti. Combinati fra loro, tali fattori offrono un'indicazione del grado di rigore con il quale applicare il processo software. In genere, ad esempio, i progetti piccoli e non critici per l'azienda si possono affrontare con rigore inferiore a quelli vasti, complessi e critici. Si noti peraltro che ogni progetto dev'essere condotto in modo da ottenere un prodotto puntuale e di alta qualità. Si possono definire quattro gradi di rigore:

- **Informale**: si applicano tutte le attività portanti del processo, ma si richiede solo un insieme minimo di compiti. In generale i compiti ausiliari sono ridotti al minimo e le esigenze di documentazione meno pesanti. Si applicano ugualmente tutti i principi fondamentali dell'ingegneria del software.
- **Strutturato**: la struttura portante del processo è pienamente rispettata. Si applicano le attività portanti ed i relativi compiti appropriati al tipo di progetto, oltre alle attività ausiliarie necessarie a garantire la qualità del prodotto. I compiti di assicurazione della qualità, gestione della configurazione, documentazione e misurazione sono condotti in maniera integrata.
- **Rigoroso**: lo schema del processo è applicato con un livello di disciplina sufficiente a garantire l'alta qualità del prodotto. Si applicano tutte le attività ausiliarie e si produce una ricca documentazione.
- **"Di intervento rapido"**: si applica la struttura portante del processo, ma, a causa di una situazione di emergenza, si applicano solo i compiti essenziali per mantenere un buon livello qualitativo. Dopo la consegna del prodotto al cliente si procede a ristabilire la situazione normale (cioè a compilare la documentazione completa ed a svolgere ulteriori revisioni). **Nota**: Se tutto viene eseguito in emergenza, vi è qualcosa di sbagliato nel processo software o nelle persone che gestiscono l'azienda od in entrambe le cose.

Il capo progetto deve mettere a punto una strategia sistematica per selezionare il grado di rigore opportuno per un dato progetto. A tale scopo, sono stati definiti alcuni criteri di adattamento, a partire dai quali si calcola un valore per la selezione dell'insieme dei compiti.

I **criteri di adattamento** servono a determinare il grado di rigore consigliato con cui applicare ad un progetto il modello di processo. Per i progetti software sono stati definiti undici criteri:

- Dimensioni del progetto
- Numero di utenti potenziali
- Criticità rispetto alla missione aziendale
- Longevità dell'applicazione
- Stabilità dei requisiti
- Facilità di comunicazione fra cliente e sviluppatore
- Maturità della tecnologia applicabile
- Vincoli prestazionali
- Caratteristiche "embedded / non embedded"
- Composizione dello staff del progetto
- Fattori di ristrutturazione

Ad ogni criterio si assegna un voto da 1 a 5, dove 1 rappresenta un progetto per il quale si richiede solo un piccolo sottoinsieme di compiti di processo ed i requisiti metodologici e di documentazione sono minimi, mentre 5 rappresenta un progetto al quale si deve applicare il corredo completo di compiti e per il quale i requisiti metodologici e di documentazione sono rilevanti.

Per selezionare l'insieme di compiti adatto ad un progetto, si devono compiere i passi seguenti:

1. Esaminare i criteri indicati qui sopra ed assegnate i voti relativi (da 1 a 5) sulla base delle caratteristiche del progetto. I voti andranno inseriti nella tabella più in basso
2. Esaminare i coefficienti (pesi) assegnati ad ogni criterio. I valori dei coefficienti variano da 0,8 ad 1,2 e danno un'indicazione dell'importanza relativa di un dato criterio per il tipo di software sviluppato nel contesto locale. Se necessario, si possono apportare modifiche tese a rispecchiare le condizioni locali.
3. Moltiplicare i valori inseriti nella tabella per i rispettivi coefficienti e per il moltiplicatore del tipo di progetto che si intraprende. Il moltiplicatore vale 0 od 1 ed indica la rilevanza del criterio nel tipo di progetto. Il risultato del prodotto voto x coefficiente x moltiplicatore va inserito nella colonna Prodotto della tabella, separatamente per ogni criterio.
4. Calcolare la media delle voci e porre il risultato nella casella TSS (Task Set Selector – Selettore dell'insieme dei compiti). Questo valore guida la scelta dell'insieme dei compiti più indicato per il progetto in esame.

Criterio di adattamento	***Voto***	***Peso***	***Moltiplicatore***					***Prodotto***
			Innovazione	*Nuova applicazione*	*Migliorativo*	*Manutenzione*	*Ristrutturazione*	
Dimensioni progetto		1,2	0	1	1	1	1	
Numero utenti		1,1	0	1	1	1	1	
Criticità aziendale		1,1	0	1	1	1	1	
Longevità		0,9	0	1	1	0	0	
Stabilità requisiti		1,2	0	1	1	1	1	
Facilità comunicazioni		0,9	1	1	1	1	1	
Maturità tecnologica		0,9	1	1	0	0	1	
Vincoli su prestazioni		0,8	0	1	0	0	1	
Embedded /non embedded		1,2	1	1	0	0	1	
Composizione staff		1	1	1	1	1	1	
Interoperabilità		1,1	0	1	1	1	1	
Fattori ristrutturazione		1,2	0	0	0	0	1	
Selettore (TSS)								

Dopo aver calcolato il TSS, si può consultare la tabella seguente e servirsene come guida per la selezione dell'insieme di compiti

TSS	*Grado di rigore*
TSS < 1,2	Informale
1,0 < TSS < 3,0	Strutturato
TSS > 2,4	Rigoroso

La sovrapposizione dei valori di TSS nelle categorie contigue è voluta ed intende sottolineare come sia impossibile definire contorni netti. Nell'analisi conclusiva, il valore di TSS, l'esperienza ed il buon senso devono contribuire alla scelta dei compiti. Normalmente se il selettore dell'insieme dei compiti è in un'area di sovrapposizione, si deve scegliere il livello di rigore meno formale a meno che i rischi non siano elevati.

Scelta dei compiti

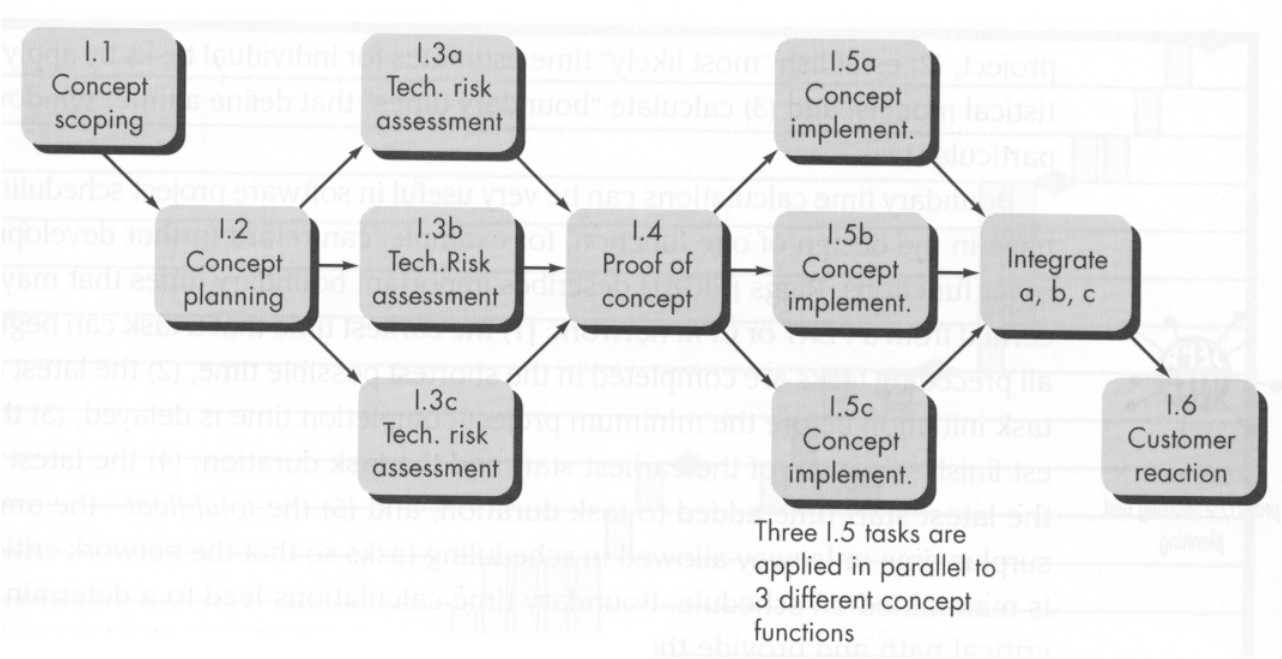
Una volta stabilita la categoria di progetto ed il rigore necessario è giunto il momento di stabilire quali sono i compiti necessari e come vanno distribuiti nel processo. Stabilire quali sono i compiti è una cosa che varia da progetto a progetto, tuttavia lo Unified Process ci dice in cosa consistono le varie fasi, di conseguenza sappiamo che la fase di Incenption contiene tutte le procedure iniziali atte a vedere se il problema è fattibile, quella di Elaboration si occupa dell'analisi del problema, ecc. In base a questo è possibile stabilire, almeno a grandi linee, quali sono i compiti necessari nelle varie fasi (estrazione dei requisiti, calcolo stime, analisi di requisiti, ecc.)

Una volta visto a grandi linee quali compiti sono necessari per la realizzazione del progetto, essi vanno raffinati, divisi in compiti più semplici che possono venir assegnati ai membri del team. Per una maggiore esplicazione possiamo seguire la tecnica per la scomposizione del lavoro e la divisione dei compiti: "Work Breakdown Structure (WBS)". Tale tecnica inizia dall'elemento di massimo livello (corrispondente al progetto nella sua interezza), si scompone quindi nei suoi componenti (sistemi, facilities, oggetti da consegnare, ecc.). Ciascuno di questi componenti viene a sua volta suddiviso nei suoi componenti costitutivi. Questa suddivisione livello per livello prosegue riducendo l'entità, la complessità ed il costo di ciascun elemento, fin quando non si raggiunge il livello di identificazione di un oggetto da consegnare. Questo viene poi scomposto nei principali compiti che debbono essere eseguiti dalle singole funzioni. L'obiettivo è di identificare elementi e compiti chiaramente gestibili ed attribuibili alla responsabilità di un membro del team che possano essere pianificati, valutati, schedulati e controllati. Ogni compito deve risultare realizzabile da un membro del team ed integrarsi coi compiti svolti dagli altri membri.

Definizione di un reticolo dei compiti

Fra i singoli compiti e sottocompiti ci sono dipendenze reciproche, basate sulla sequenza di svolgimento. Inoltre, quando il progetto coinvolge più persone, è frequente il caso in cui due o più compiti siano svolti in parallelo. In questo caso occorre coordinare i compiti svolti simultaneamente, affinché siano completati nel momento in cui i compiti successivi ne richiedono i risultati.

Un **reticolo di compiti** o **rete delle attività** è una rappresentazione grafica della successione di compiti di un progetto. Nella sua forma più semplice il reticolo rappresenta i compiti più importanti. Nella figura seguente è illustrato un reticolo per un progetto di sviluppo di un'idea innovativa



Notare che il reticolo dei compiti è basato sui compiti stabili a grandi linee, eventualmente dividendo (come nel caso del compito 1.3 ed 1.5) qualcuno di questi se già si sa che possono essere divisi in parti che possano procedere in parallelo.

La natura concorrente delle attività di ingegneria del software impone alcuni importanti requisiti riguardo la pianificazione temporale. Poiché i compiti svolti in parallelo non sono sincronizzati, il pianificatore deve determinare le relazioni di dipendenza per coordinare l'avanzamento del progetto. Inoltre il capo progetto deve sapere quali compiti si trovano sul cammino critico, cioè quei compiti che devono rispettare i tempi previsti affinché il progetto in generale possa rispettare la propria scadenza.

Pianificazione temporale

Stabiliti il reticolo dei compiti e suddivisi ulteriormente i compiti, è necessario affidare i compiti al team e vedere come essi possano distribuirsi nel corso del tempo.

Il PERT (Program Evaluation and Review Technique, tecnica di valutazione e revisione dei piani) ed il CPM (Critical Path Method, metodo del cammino critico) sono due metodi applicati allo sviluppo del software. Entrambi sono pilotati da informazioni elaborate nelle prime fasi di pianificazione di un progetto:

- stime del carico di lavoro;
- scomposizione delle funzioni del prodotto;
- selezione del modello di processo e dell'insieme dei compiti appropriati;
- Decomposizione dei compiti.

Le dipendenze reciproche fra i compiti si possono esprimere per mezzo di un reticolo. È possibile definire i compiti, tecnicamente denominati anche WBS (Work Breakdown Structure, struttura di ripartizione del lavoro), sia per il prodotto come unità, sia per le singole funzioni.

Sia PERT che CPM forniscono strumenti quantitativi che consentono al pianificatore di determinare il **cammino critico**, ossia la catena di compiti che determina la durata di un progetto, stabilire le sti-

me più probabili dei tempi necessari al completamento di ogni singolo compito, applicando modelli statistici, calcolare i limiti temporali che definiscono la “finestra” temporale entro cui ciascun compito deve essere iniziato e terminato.

Il calcolo di tali limiti può essere molto utile per la pianificazione temporale di un progetto. Lo slittamento dei tempi di progettazione di una funzione, ad esempio, può ritardarne altre.

Ci possono essere alcuni importanti limiti temporali che possono essere dedotti dalle reti PERT o CPM: il maggior anticipo possibile dell’inizio di un compito, cioè l’istante in cui può iniziare un compito, supponendo di aver svolto nel minor tempo possibile tutti i compiti precedenti; il maggior ritardo possibile nell’iniziare un compito, in modo da non dover ritardare il minimo tempo previsto per il completamento del progetto; la prima data possibile di completamento (la somma della prima data possibile e della durata del compito); l’ultima data possibile di completamento (l’ultima partenza possibile più la durata del compito); la flessibilità totale, ossia la quantità di ritardi o di perdite di tempo consentita nella pianificazione dei compiti, in modo da mantenere nei tempi previsti il cammino critico della rete.

Il calcolo dei limiti temporali porta alla determinazione del cammino critico e fornisce al responsabile del progetto un metodo quantitativo per la valutazione dell’avanzamento del progetto, a mano a mano che i compiti vengono completati.

Un altro strumento utile per la pianificazione temporale è il diagramma dei tempi, detto **diagramma di Gantt**. Questo diagramma mostra, per i vari compiti, la loro durata temporale, i punti di inizio e fine, quali compiti possono essere svolti in parallelo ed i punti di controllo.

Sia PERT, sia Gantt sono compilati a partire da strumenti software.

Esempio: SurfReport

Il cliente vuole un servizio di informazioni su tempo e maree; dalle richieste preliminari ricevute dal cliente si è proceduto alla seguente decomposizione funzionale:

- **A, B, C**: screen scrapers per 3 URLs
- **D**: modulo per formattare l’email
- **E**: modulo per spedire l’ email
- **F**: modulo temporizzatore

Si è stabilito quanto tempo ci voglia per ogni compito:

- **Moduli A, B, C**: Perl; 1 ora ciascuno
- **Modulo D**: Perl; 30 minuti
- **Modulo E**: Perl/sendmail; 1 ora
- **Modulo F**: Shell script/cron; 1 ora

Stime di tempi: 5.5 ore (una persona), 3.5 ore (3 persone)

Diagramma di Gantt

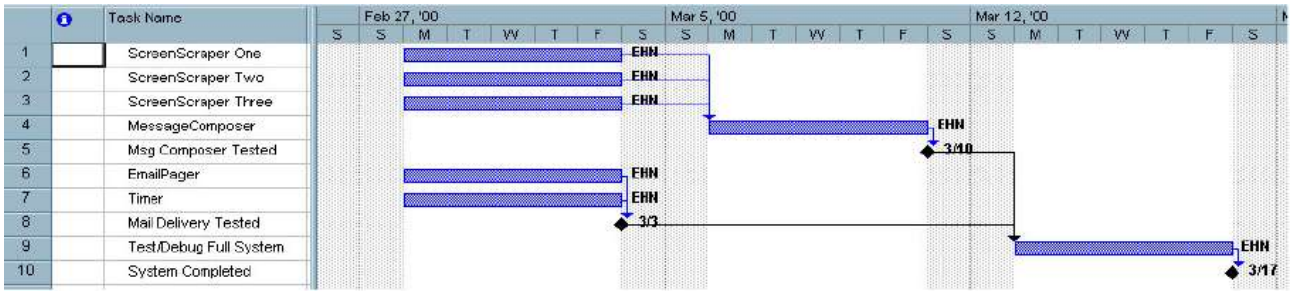


Diagramma PERT

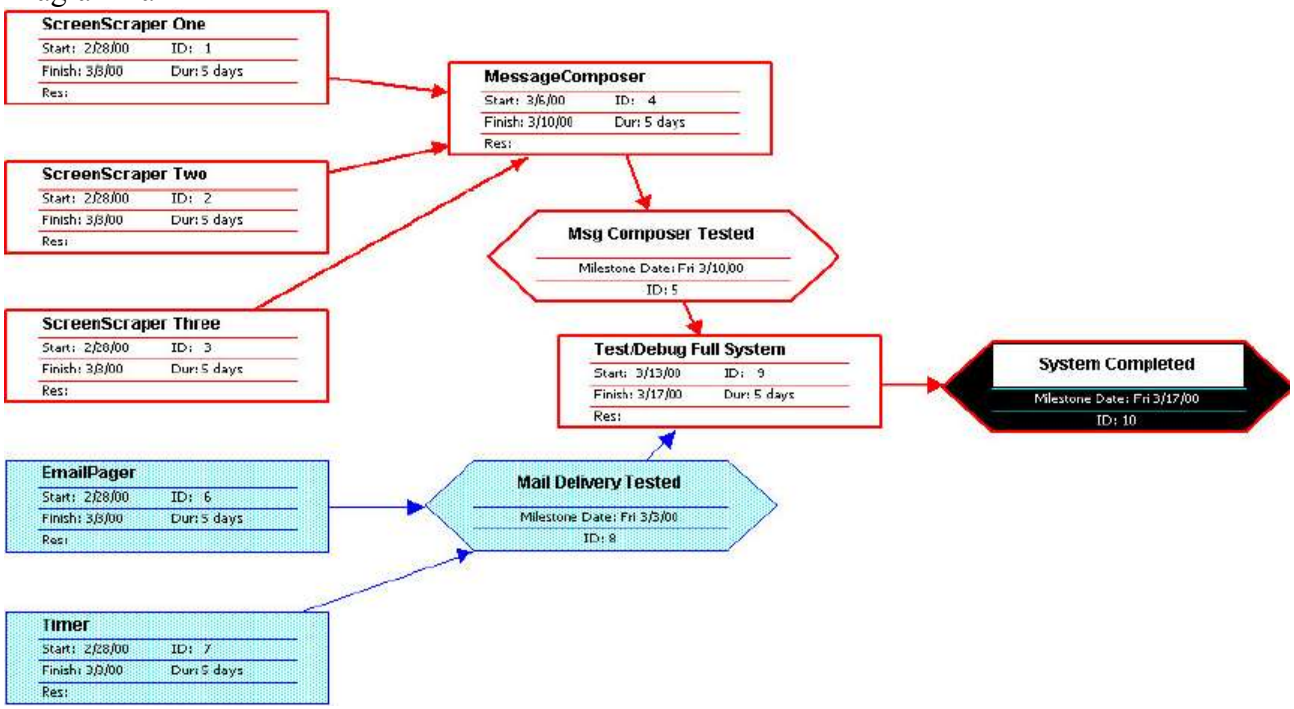
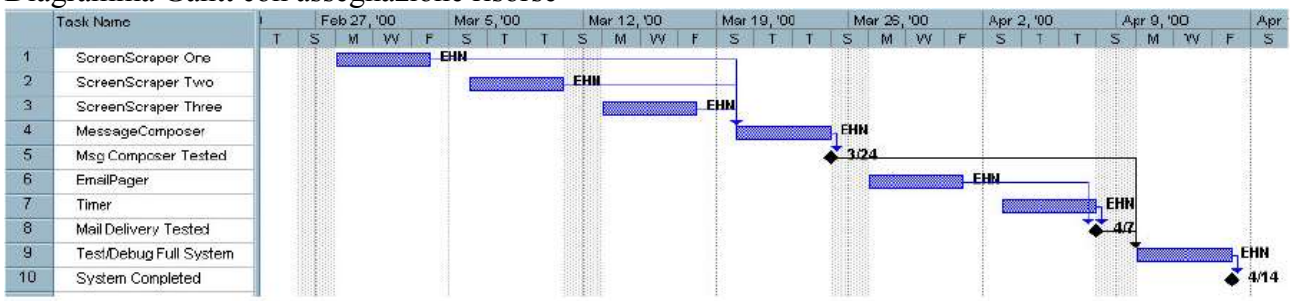
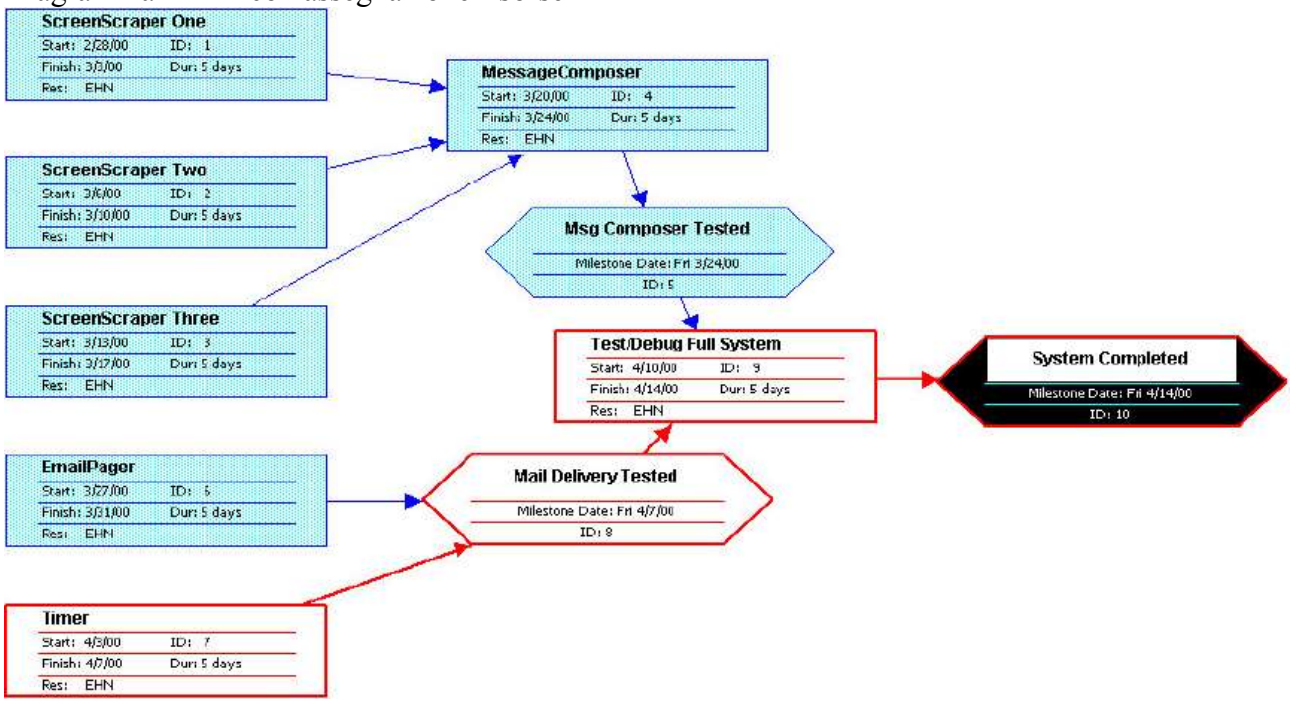


Diagramma Gantt con assegnazione risorse



Durata estesa a7 settimane

Diagramma PERT con assegnazione risorse



Sorveglianza della tabella dei tempi

La tabella dei tempi costituisce una guida per il capo progetto. Se opportunamente sviluppata, essa definisce i compiti ed i punti di controllo che devono essere tenuti sotto controllo a mano a mano che il progetto avanza. Esistono vari modi per ottenere questo controllo:

- tenere periodiche riunioni dedicate allo stato del progetto, nel corso delle quali ogni membro del team riferisce sui progressi e sui problemi;
- valutare i risultati delle revisioni svolte nel corso del progetto;
- determinare se i punti di controllo formali del progetto (i rombi nel diagramma di Gantt) sono portati a termine nei tempi previsti;
- confrontare, per ogni compito elencato nella tabella del progetto (che riunisce tutti i compiti con le date di inizio e fine), la data effettiva di inizio con quella prevista;
- consultare informalmente i tecnici per avere le loro valutazioni soggettive sulla situazione e sui problemi potenziali;
- utilizzare l'analisi del valore acquisito (vedi avanti) per stabilire quantitativamente i progressi.

Nella realtà un capo progetto esperto impiega tutte queste tecniche.

Il capo progetto si serve del controllo per amministrare le risorse disponibili, gestire i problemi e dirigere lo staff. Quando il lavoro procede bene (tempi e costi sono rispettati, le revisioni indicano un effettivo progresso ed i punti di controllo sono rispettati) il controllo non è pesante.

Tuttavia, qualora vi siano problemi, il capo progetto deve esercitare un controllo per riportare la situazione nei binari il più presto possibile. Quando si rileva un problema (ed uno slittamento dei tempi, è un problema), si possono dirottare le risorse nell'area interessata, ad esempio ridistribuendo lo staff o ritoccando la tabella dei tempi.

Analisi del valore acquisito

Il sistema del valore acquisito fornisce una scala di valutazione di ogni compito di un progetto software, indipendentemente dal tipo di lavoro da svolgere. Si stimano le ore totali per l'intero progetto ed ad ogni compito viene assegnato un valore acquisito sulla base di una stima percentuale del totale.

Per determinare il valore acquisito devono essere svolte le seguenti operazioni:

1. Determinare il costo previsto del lavoro pianificato BCWS (Budgeted Cost of Work Scheduled) per ogni compito rappresentato nel piano. Durante l'attività di stima, viene pianificato il lavoro (in ore-uomo o giorni-uomo) di ogni compito. Pertanto, $BCWS_i$ è il costo pianificato per il compito i. Per determinare i progressi in un determinato punto della pianificazione del progetto, si considera che il valore di BCWS è la somma dei valori $BCWS_i$ di tutti i compiti che devono essere completati in un determinato momento della pianificazione del progetto.
2. Sommare i valori BCWS di tutti i compiti per calcolare il costo globale di completamento BAC (Budget At Completion)
   $$BAC = \sum BCWS_k \ \ per\ tutti\ i\ compiti\ k$$
3. Poi viene calcolato il valore del costo del lavoro eseguito BCWP (Budgeted Cost of Work Performed). Il valore BCWP è la somma dei BCWS di tutti i compiti che sono stati effettivamente completati in un determinato momento della pianificazione del progetto.

Si noti che la distinzione fra BCWS e BCWP sta nel fatto che il primo rappresenta il budget per le attività che si prevedeva di completare ed il secondo rappresenta il budget delle attività effettivamente completate.

Dati i valori BCWS, BAC e BCWP possono essere calcolati alcuni importanti indicatori del progresso del lavoro:

Schedule Performance Index (indice delle prestazioni della pianificazione) SPI = BCWP / BCWS

Schedule Variance (varianza della pianificazione) SV=BCWP – BCWS

SPI è un indicatore dell'efficienza con cui il progetto sta utilizzando le risorse pianificate. Un valore di SPI vicino ad 1 indica un'efficiente esecuzione della pianificazione del progetto. SV è semplicemente un'indicazione assoluta della varianza rispetto alla pianificazione prevista.

Percentuale pianificata per il completamento = BCWS / BAC

fornisce un'indicazione della percentuale di lavoro che deve essere completata nel tempo t

Percentuale di completamento = BCWP / BAC

fornisce un'indicazione quantitativa della percentuale di completamento del progetto in un determinato punto del tempo t.

È anche possibile calcolare il costo effettivo del lavoro eseguito, ACWP (Actual Cost of Work Performed). Il valore di ACWP è la somma degli sforzi attualmente spesi sui compiti che sono stati completati in un determinato punto della pianificazione temporale del progetto. È pertanto possibile calcolare i due valori seguenti:

Cost Performance Index (indice prestazionale dei costi) CPI=BCWP / ACWP

Cost Variance (varianza dei costi) CV=BCWP – ACWP

Un valore CPI vicino ad 1 è un buon indicatore del fatto che il progetto rientra nella somma defini-

ta. CV è un'indicazione assoluta dei risparmi in termini di costo (rispetto ai costi pianificati) in una determinata fase del progetto.

Come un radar, l'analisi del valore acquisito evidenzia le difficoltà di pianificazione prima che divengano evidenti. Questo consente al manager del progetto software di intraprendere azioni correttive prima che si sviluppi una crisi del progetto.

Controllo degli errori

Una delle cose da considerare quando si assegnano le risorse è l'efficienza della rimozione dei difetti (vedi il capitolo Metriche). Le valutazioni effettuate in tal ambito possono essere utilizzate anche per indirizzare meglio le risorse di revisione e/o collaudo.

Piano di progetto

Ogni passo del processo di sviluppo produce un lavorato, che deve essere rivisto e che forma la base per i passi che seguono. Il piano di progetto è il coronamento delle attività di pianificazione. Esso contiene una base di informazioni sui costi ed i tempi, da utilizzare lungo tutto il processo software.

Il piano di progetto è un documento relativamente breve, indirizzato ad un pubblico eterogeneo. Esso deve comunicare la portata e le risorse al responsabile del progetto, al personale tecnico ed al cliente; definire i rischi e suggerire le tecniche di governo dei rischi; definire costi e tempi, per consentire alla direzione di esaminarli; fornire una visione globale a tutte le persone in qualche modo coinvolte nel progetto; delineare le strategie per la qualità e per il governo dei cambiamenti.

La presentazione dei costi e dei tempi varia a seconda del pubblico a cui è indirizzata. Se il piano è utilizzato soltanto come documento interno, si possono presentare anche i risultati di ciascuna tecnica impiegata per le stime dei costi. Se il piano è distribuito esternamente all'organizzazione, viene fornito solo il risultato globale delle stime dei costi (si combinano i risultati di tutte le tecniche impiegate a tal scopo). Analogamente, il livello di dettaglio del paragrafo dedicato alla pianificazione dei tempi può variare a seconda del pubblico e del grado di formalità del piano.

È importante notare che la pianificazione del progetto software non è un documento statico. Infatti il team del progetto lo rivede ripetutamente, aggiornando i rischi, le stime, le pianificazioni e le altre informazioni correlate a mano a mano che si acquisiscano nuove conoscenze sul progetto.

Riepilogo

Riepilogando, l'attività di Requisiti (o Requirements) ha lo scopo di estrarre ed analizzare i requisiti di sistema. Essa si svolge in più iterazioni: la prima serve a controllare che il progetto sia fattibile ed abbia un costo accettabile per il cliente, le altre servono ad estrarre e classificare via via i requisiti per il sistema.

Durante l'inizio di questa attività partono anche le attività di ombrello che calcolano le stime, valutano i rischi e pianificano risorse e tempi per portare a termine il progetto.

Riepilogo documenti prodotti

L'attività di Requirements produce tutta una serie di documenti, il cui scopo è definire bene cosa deve fare il sistema.

Normalmente il documento dei requisiti comprende:

Il **documento di visione**, contenente:

- **Premesse del progetto** (Obiettivi e scopo, Contesto, Stakeholders, ecc.)
- **Servizi del Sistema** (Scopo del sistema, Requisiti Funzionali, Requisiti informativi, ecc.)
- **Vincoli** (Requisiti di interfaccia, di prestazione, di sicurezza, operativi, ecc.)

Gli **aspetti procedurali** (problemi aperti, previsione costi, ecc.)

Le **"Appendici"**. E qui includiamo:

- Modello di business
- Glossario
- Modello dei casi d'uso (diagramma e documentazione aggiuntiva)
- Diagramma delle attività e di stato
- Valutazione del rischio (RMMM, RIS, tabella dei rischi, ecc.)
- Stima del costo (per ogni requisito)
- SRS
- Piano di progetto iniziale (assegnazione dei compiti, tempi di esecuzione, diagrammi PERT, ecc.)
- Prototipi
- Documento di architettura iniziale

Ci può essere della Documentazione aggiuntiva, come analisi di attività precedenti, ma non è presente in tutti i progetti.

Analisys (Analisi)

Scopo dell'attività di analisi è quello di creare modelli che catturano il comportamento desiderato del sistema. L'attività di Analisi inizia non appena sono pronti i primi requisiti e costruisce le classi di analisi e le realizzazioni dei casi d'uso (il fuoco è ancora sul cosa).

Il modello di analisi descrive la storia del sistema, cattura l'insieme e contiene i modelli appartenenti al dominio del problema (quindi con esclusione di classi di comunicazione o di database). Il modello deve essere il più utile possibile agli stakeholders. Esso deve essere il più semplice possibile, non prendere decisioni sull'implementazione ed includere solamente classi del dominio del problema.

È proprio durante questa attività che si stabilisce l'architettura di base. L'architettura di base del software di un programma o di un sistema di calcolo è costituita dalle strutture del sistema, che comprendono i componenti software, le loro proprietà visibili e le relazioni fra di essi.

Tutta l'attività, dunque, verte su:

Individuazione delle classi di analisi

Analisi architetturale

Analisi dei casi d'uso

Classi di Analisi

Le classi di analisi fanno una fotografia in grande del sistema, lasciando qualsiasi dettaglio di imple-

mentazione all'attività di design.

Esse comprendono

- Nome – Richiesto
- Attributi – solamente un insieme limitato affinché si possa capire cosa memorizza la classe. Solo il nome; tipi, valori di default e visibilità non sono richiesti
- Metodi – solo i metodi di alto livello per far capire le responsabilità della classe. Solo il nome; parametri, valore di ritorno e visibilità non sono richiesti.
- Eventuali valori di default, parametri, tipi, ecc. vengono specificati se sono necessari a far capire il modello.

Alcune regole da seguire quando si cercano le classi di analisi:

- Da tre a cinque responsabilità per classe – le classi devono essere mantenute semplici.
- Nessuna classe deve essere isolata – le classi vengono create per cooperare tra loro.
- Diffidare di molte piccole classi – Avere molte classi può portare a difficoltà di manutenzione. Raggrupate delle funzionalità.
- Diffidate di poche grandi classi – Problema inverso. Dividete le funzionalità.
- Diffidate dei funtoidi – i funtoidi sono delle funzioni o delle procedure che vengono aggiunte ad una classe (la classe ha solo tale metodo o metodi simili).
- Diffidate delle classi onnipotenti – classi come Sistema o Controller indicano di aver aggiunto troppe responsabilità a classi non ben definite.
- Diffidate di alberi di ereditarietà profondi – questo è un cattivo uso dell'ereditarietà.

Diagramma delle classi

Per visualizzare le classi, l'UML ci viene incontro con il diagramma delle classi. Grazie a questo strumento si visualizzano graficamente le classi e le loro associazioni.

Una classe viene rappresentata con un rettangolo contenente un nome; questo rettangolo può essere diviso in tre comparti: Il primo riporta il nome, il secondo gli attributi ed il terzo le operazioni.

Ogni attributo è definito da un nome, opzionalmente si può indicare la visibilità, il tipo ed un eventuale valore di default. La visibilità è indicata attraverso un simbolo prima del nome; tale simbolo può essere:

- + → indica che l'attributo è pubblico
- - → indica che l'attributo è privato
- # → indica che l'attributo è protetto

Il tipo è indicato con un nome, separato dal nome dell'attributo attraverso i due punti. I tipi possibili in UML si classificano in tipi *primitivi* (cioè tipi intuitivamente conosciuti come interi, stringhe, ecc., oppure definiti di base nel linguaggio di programmazione sottostante) e tipi *classe* (indica che l'attributo contiene il riferimento ad un oggetto istanza della classe specificata).

Il valore di default è il valore, separato dal segno di uguale dal resto, che l'attributo prende automaticamente quando un oggetto viene istanziato.

In più è possibile indicare se un attributo è statico (cioè un attributo comune a tutte le istanze) ante-

ponendogli il simbolo $ oppure sottolineandolo. Se un attributo è derivato (cioè esso non esiste fisicamente, ma è ricavato attraverso il calcolo) allora gli si antepone il simbolo /

Per quanto riguarda le operazioni anch'esse sono indicate da un nome, normalmente seguito da una coppia di parentesi. È possibile definirne la visibilità (indicata come per gli attributi), i parametri che prende (mettendoli dentro le parentesi che seguono il nome, e separando ognuno di essi tramite la virgola) ed il tipo di ritorno (separandolo con due punti).

Se si specificano i parametri essi devono avere nome e tipo (separati attraverso i due punti).

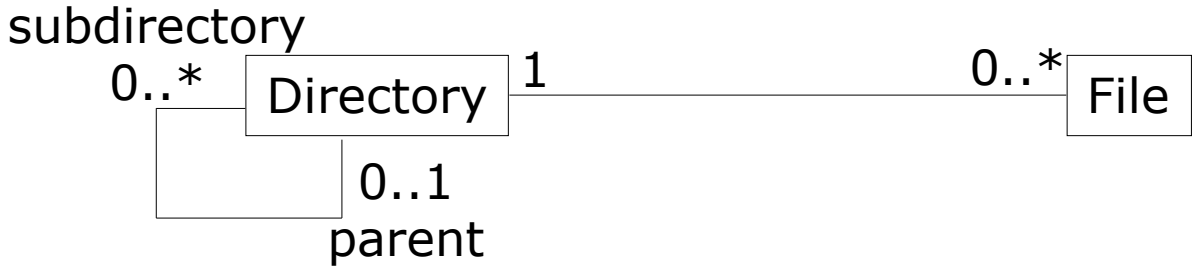
Anche i metodi possono essere statici (cioè sono globali) e come tali vengono indicati anteponendo il simbolo $ o sottolineandoli. Non possono esistere metodi derivati.

Per quanto riguarda le associazioni, si indicano utilizzando una linea che collega due classi. Ogni linea rappresenta un'associazione; le associazioni possono avere molteplicità (segnata agli estremi), nomi (segato al centro), nomi di ruoli (segnati agli estremi). Se la linea è in realtà una freccia, significa che la navigabilità è solo in quella direzione.

In questo esempio possiamo notare un triangolo vicino al nome; questo indica solamente la direzione in cui va letto il nome: la compagnia assume delle persone, non sono le persone che assumono la compagnia. Serve solo come nota all'utente per fargli capire come si era letta l'associazione per dargli quel nome. Ed in effetti qui possiamo dalla compagnia risalire alle persone assunte e dalle persone possiamo risalire alla compagnia che le ha assunte. L'associazione ha inoltre una molteplicità uno a molti, cioè una compagnia può assumere più persone, mentre una persona può essere assunta da una sola compagnia.

In questo esempio l'associazione ha una navigabilità: questo significa che dalla compagnia possiamo risalire alle persone ma non viceversa. Anche qui l'associazione ha una molteplicità uno a molti. Possiamo notare che in questo caso siano stati specificati i ruoli della compagnia e delle persone, mentre è stato omesso il nome dell'associazione (che rischiava di creare confusione)

Le associazioni possono anche essere cicliche:



Per rappresentare le associazioni di aggregazione si usa mettere nel lato del contenitore, un rombo vuoto, mentre per quelle di composizione si usa mettere nel lato del contenitore, un rombo pieno.

Le generalizzazioni sono rappresentate da una freccia, la cui punta è vuota.

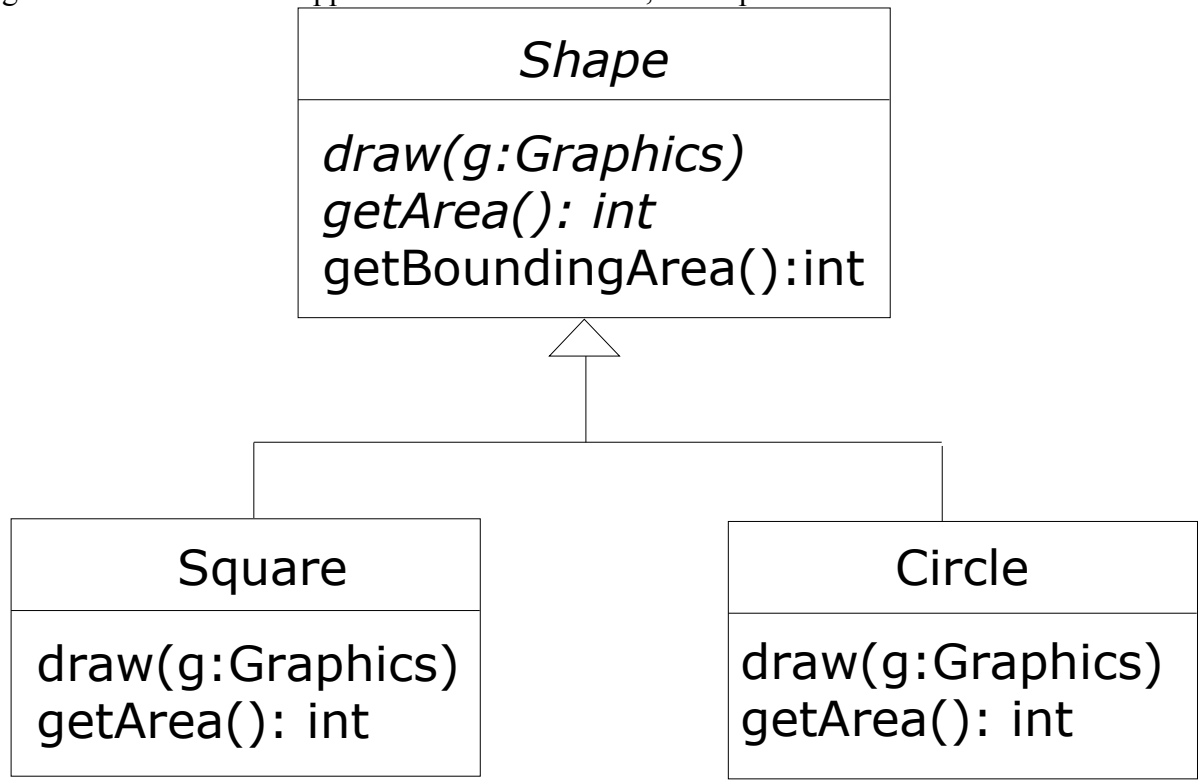
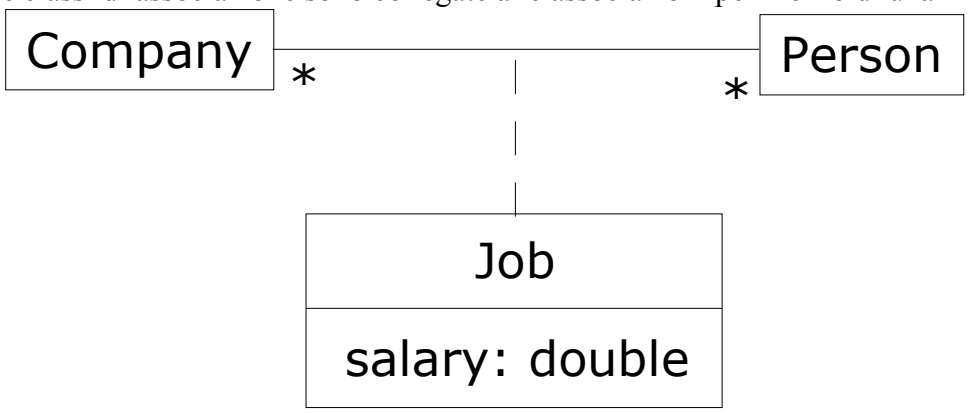
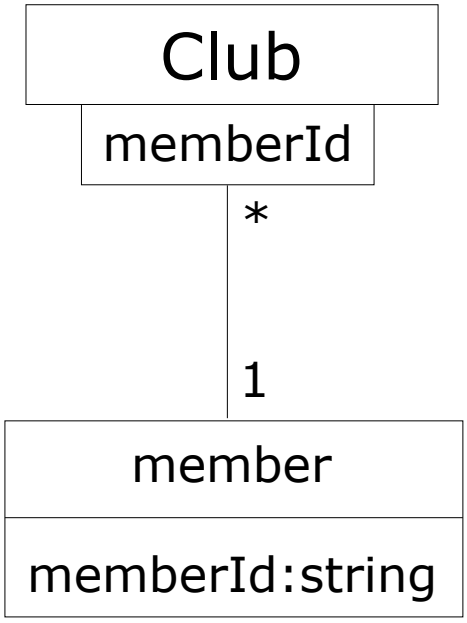


Illustrazione 2 Qui vediamo che sia Square che Circle estendono Shape

Le classi di associazione sono collegate alle associazioni per mezzo di una linea tratteggiata.



Infine ci possono essere le associazioni qualificate, cioè delle associazioni che indicano che da una classe è possibile estrarre elementi di un'altra classe attraverso un classificatore; nell'esempio sotto notiamo che memberId è il classificatore con il quale possiamo estrarre da un qualsiasi oggetto Club, tutti gli oggetti Member che ne fanno parte.



Per concludere, in questo diagramma è possibile rappresentare anche le dipendenze delle diverse classi. La rappresentazione avviene per mezzo di una freccia tratteggiata su cui viene posto un nome:

<<use>> → questo indica che una classe usa l'altra (il tipo d'uso non è specificato).

<<call>> → questo indica che una classe chiama un metodo dell'altra

<<send>> → questo indica che una classe invia un segnale all'altra

<<instantiate>> → questo indica che una classe istanzia un oggetto dell'altra

Ovviamente possono esistere altri tipi di dipendenza, persino definiti dall'utente, purché siano definiti da qualche parte (leggi glossario).

Analisi Architetturale

Il compito dell'analisi architetturale è di dividere le classi di analisi in package, in modo da diminuire l'accoppiamento tra le classi (cioé minimizzare le dipendenze tra i package), minimizzare il numero di elementi pubblici e protetti in ogni package e massimizzare il numero di elementi privati in ogni package.

I package mettono insieme classi che si occupano della stessa semantica; vengono solitamente trovati osservando i casi d'uso e le relazioni tra le classi.

I casi d'uso sono spesso realizzati attraverso classi provenienti da diversi package; tuttavia uno o più casi d'uso che supportano un particolare processo od attore, oppure un insieme di casi d'uso relazionati, possono indicare un package potenziale.

Le relazioni che potrebbero indicare package sono ereditarietà, aggregazione, composizione o dipendenza. Una volta individuato un package si minimizzano i membri pubblici e protetti, infatti lo scopo del package è quello di avere alta coesione all'interno, ma basso accoppiamento con altri package. Le dipendenze tra i vari package non possono essere cicliche; quando questo avviene significa che i due package sono in realtà lo stesso package duplicato od esistono elementi in comune che in realtà devono far parte di un nuovo package.

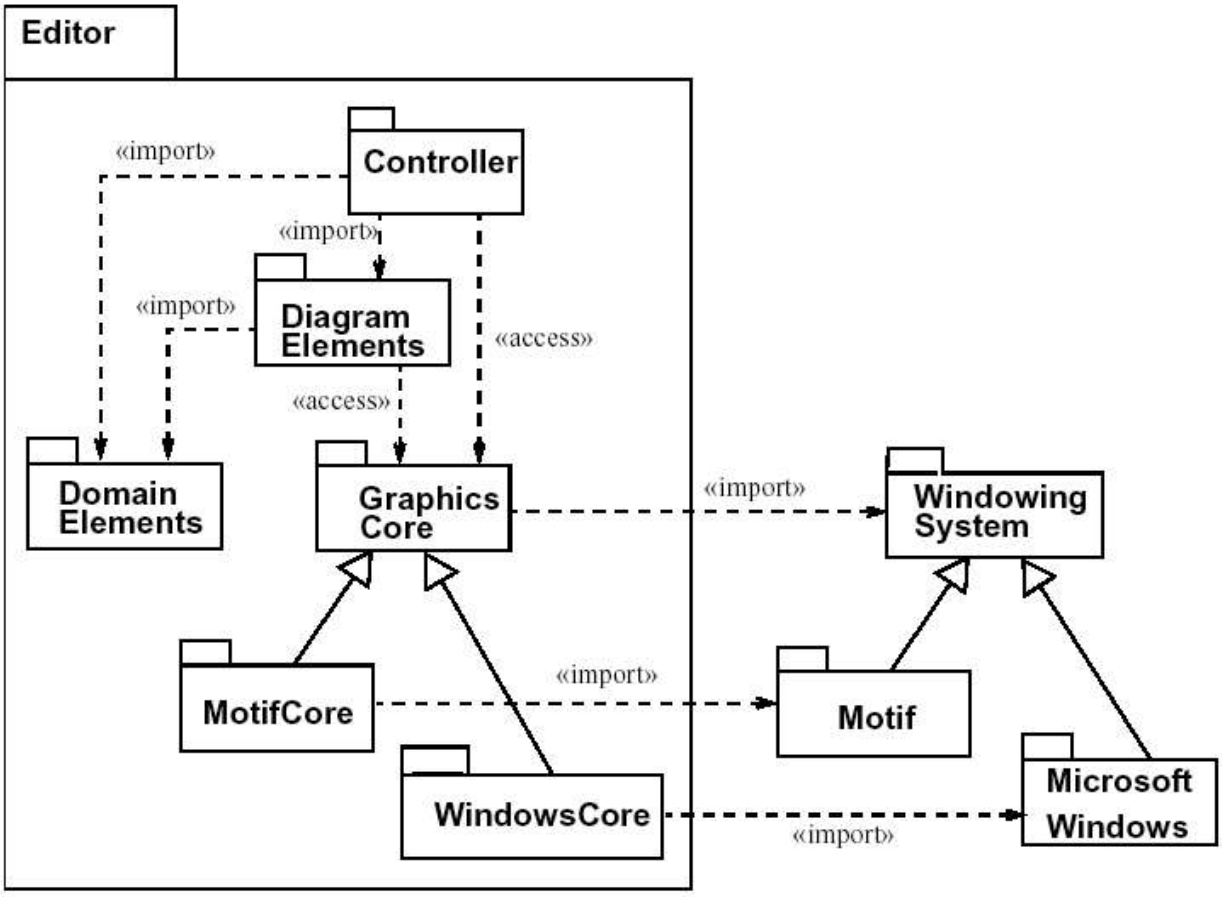
In UML questo viene rappresentato con il diagramma dei Package

Diagramma dei Package

Il diagramma dei package è formato da un rettangolo che contiene classi ed altri package; in cima al rettangolo vi è un secondo rettangolo (in modo da formare una scheda) contenente il nome del package; è possibile che vi siano dipendenze tra questi package, rappresentate con freccie tratteggiate con sopra un nome. Oltre alle dipendenze menzionate per le classi, esistono due dipendenze importanti per i package:

<<access>> → con il quale si indica che un package accede agli elementi pubblici dell'altro package, facendovi riferimento esplicito

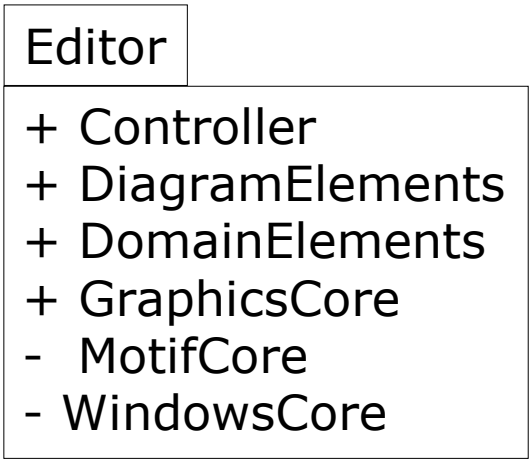
<<import>> → con il quale si indica che un package accede agli elementi pubblici dell'altro package, senza farvi riferimento in maniera esplicita, ma importando al proprio interno il namespace, cioè ogni volta che si fa riferimento ad un oggetto, e questo non appartiene al package corrente, viene ricercato tra gli elementi pubblici del package su cui vi è la dipendenza import.



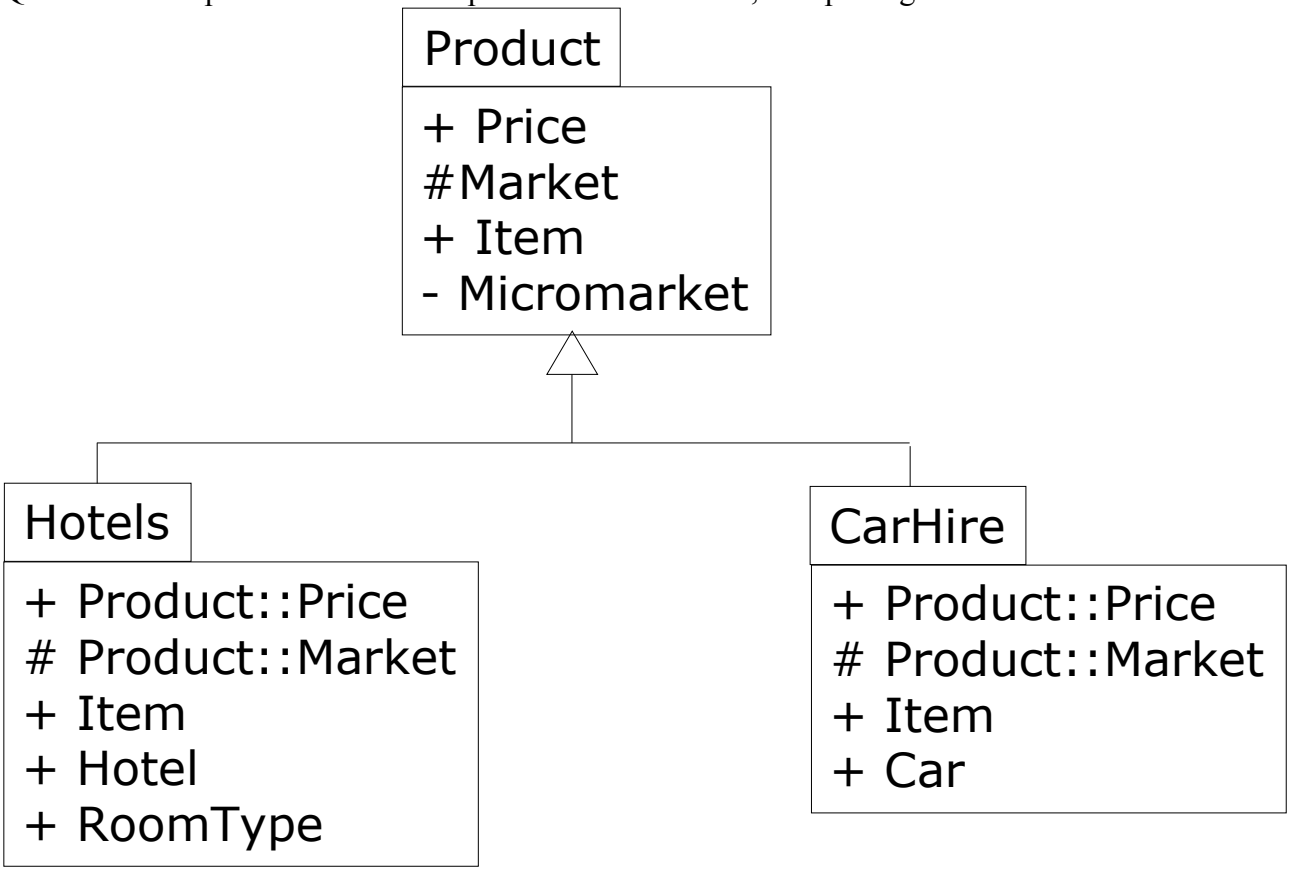
Nel diagramma qui sopra, possiamo notare anche un altro modo di visualizzare il package, quando non ci interessa il loro contenuto interno: il nome viene messo all'interno del rettangolo più grande.

Oltre a questo notiamo delle freccie con la punta vuota: queste indicano una generalizzazione tra package. Che significa? Una generalizzazione tra package indica semplicemente che alcune classi generalizzano classi appartenenti ad un altro package (che può essere utile come si vede nell'esempio sopra).

Esiste un altro modo per visualizzare i package e le classi interni ed è quello di mettere i nomi dei package preceduti da un segno meno, più o cancelletto a seconda della loro visibilità. Questo sistema fa perdere le dipendenze tra i package, ma permette di vedere a colpo d'occhio quali sono gli elementi pubblici e privati



Questo sistema permette anche di capire cosa estende cosa, tra i package



Nell'esempio sopra possiamo notare che Micromarket è privato e quindi non può essere esteso, Item viene riscritto dalle due classi, mentre Price e Market, vengono ripresi esattamente dalle superclassi dentro il package Product.

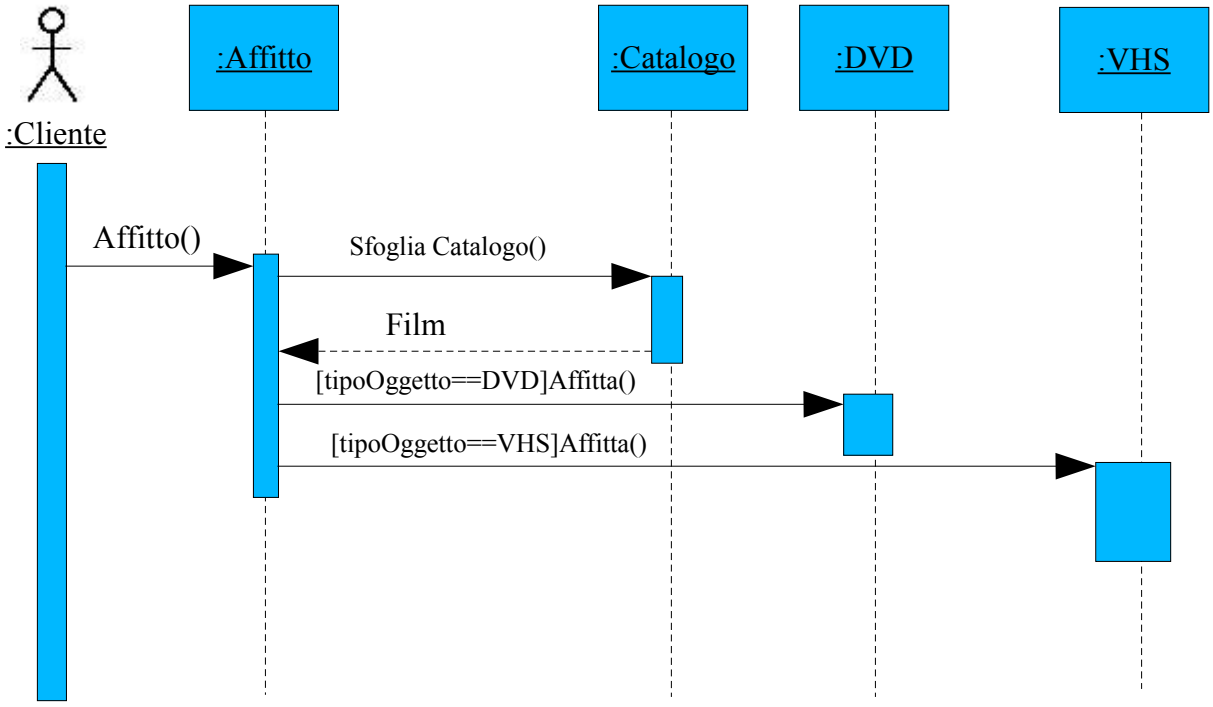
Realizzazione dei casi d'uso

A mano a mano che le classi di analisi vengono completate, si vede come possono interagire per realizzare i vari casi d'uso. A supporto di questo esistono i diagrammi di collaborazione e i diagrammi di sequenza. Questi due diagrammi hanno la stessa semantica ed esprimono lo stesso concetto: “come interagiscono le classi al fine di realizzare il caso d'uso preso in esame?”. La differenza sta nella visione di insieme che forniscono: i primi centrano l'attenzione su quali classi lavorano per realizzare il caso d'uso, mentre la sequenza delle operazioni è secondaria; i secondi focalizzano l'attenzione sulla sequenza temporale dei messaggi scambiati, su quando gli oggetti vengono creati e distrutti, mettendo in secondo piano l'insieme delle classi che interagisce. Il più delle volte gli strumenti CASE sono in grado, dato uno di questi diagrammi, di generare l'altro.

Diagrammi di sequenza

Nel diagramma di sequenza gli oggetti e gli attori sono disposti in orizzontale; la sequenza dei messaggi scambiati è mostrata dall'alto al basso nella dimensione verticale. Ogni linea verticale è una linea temporale chiamata linea di vita di un oggetto. Una linea di vita è rappresentata da una linea tratteggiate, se l'oggetto non è attivo (in attesa di istruzioni), o con un rettangolo, se l'oggetto sta lavorando (gli oggetti sempre attivi, come un attore od il tempo, hanno sempre e solo un rettangolo). Una freccia rappresenta un messaggio da un oggetto chiamante (**sender**) ad uno chiamato (**target**) per dirgli di eseguire un'operazione; alla fine dell'elaborazione si può indicare il ritorno dalla chiamata con una linea tratteggiata. Il messaggio ha un nome e può avere una lista di parametri attuali; se il metodo deve restituire un valore esso può essere fatto in due modi: nella chiamata si specifica il valore di ritorno nella forma val:=fun (param), oppure nella freccia di ritorno si indica un nome che identifica cosa sta venendo restituito. La freccia di ritorno può essere del tutto omessa quando si capisce in che momento la funzione ritorna (spesso, quindi, con gli oggetti che si risvegliano con una chiamata e tornano in letargo una volta finito l'esecuzione del metodo). Un messaggio può essere inviato ad una collezione di oggetti (un insieme, una lista, ecc.); per indicare questo si usa anteporre un asterisco al nome del messaggio. Prima di addentrarci ulteriormente nella sintassi di questi diagrammi, vediamo un esempio.

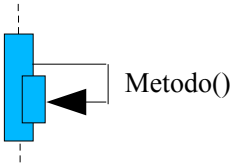
Vediamo il funzionamento di un diagramma di sequenza nel nostro esempio dell'affitto video.



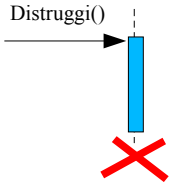
Già possiamo notare alcune cose non dette: Dato che sono gli oggetti a collaborare, si usa la notazione :nome, questa notazione indica che sto parlando di un oggetto del tipo nome, di quale degli n oggetti che possono venir creati, non importa (se fosse stato importante, ci sarebbe stato anche un nome prima dei due punti). Anche l'utente viene considerato un oggetto sempre attivo, ai fini del diagramma (e così lo sarebbe stato pure un eventuale tempo).

Un'altra cosa che si nota è l'uso delle parentesi quadre: in questo caso si indica di seguire la freccia, se la condizione specificata dentro le parentesi è verificata (in questo caso, quindi, si chiama il metodo Affitta sull'oggetto DVD, se il film che vogliamo affittare è un DVD, altrimenti chiamiamo il metodo Affitta sull'oggetto VHS).

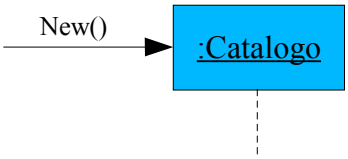
Nei diagrammi di sequenza si possono invocare anche metodi propri, la creazione di nuovi oggetti, la loro distruzione, ed i segnali; con segnale intendiamo una chiamata asincrona, cioè il chiamante dice ad un altro oggetto (od anche se stesso) di invocare un suo metodo, ma non aspetta la fine dell'esecuzione per continuare il suo lavoro.



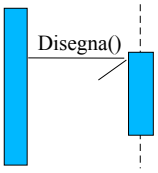
Esempio di chiamata di un proprio metodo



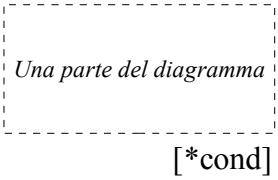
Distruzione di un oggetto



Creazione di un oggetto di tipo Catalogo



Esempio di invio di un segnale



La parte di diagramma rappresentata all'interno del rettangolo, viene eseguita sulla condizione (che può essere while, for, until, for each, ecc.)

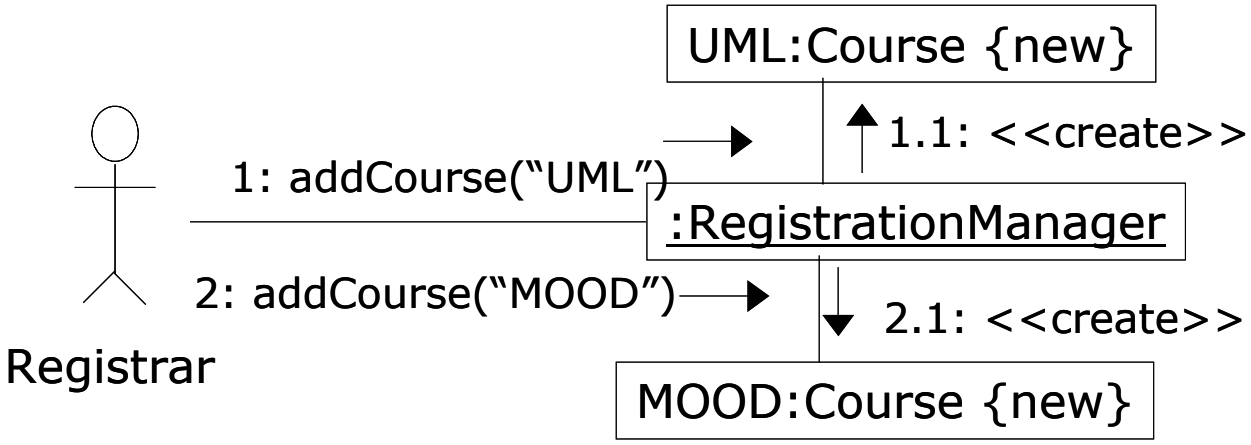
A volte i diagrammi di sequenza vengono realizzati a partire anche dai diagrammi di attività (che aiutano ad individuare altre operazioni che le classi potrebbero avere).

Diagrammi di collaborazione

Nei diagrammi di collaborazione si enfatizza il ruolo che hanno gli oggetti tra di loro. Escludendo l'utente, che è rappresentato con un omino, ogni oggetto è rappresentato con un rettangolo con den-

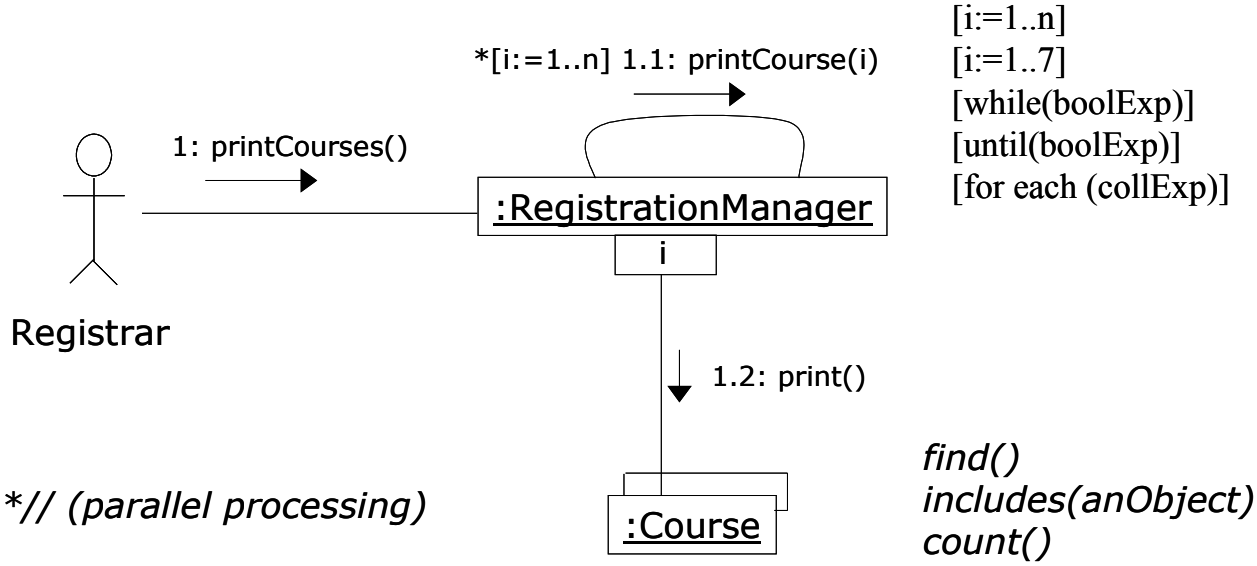
tro il nome del ruolo ed il tipo di oggetto, separati dai due punti. Tutta la scritta è sottolineata.Le invocazioni tra gli oggetti sono rappresentate da freccie il cui descrittore è così formato: eventuale condizione tra parentesi quadre, numero di sequenza, il carattere due punti, nome del metodo da chiamare (e relativi parametri) oppure nome di quello che viene restituito. Se si vuole omettere il messaggio di ritorno, si può mettere il valore di ritorno nel nome del metodo chiamato, separandolo con :=

Esempio:



Qui vediamo anche la creazione di oggetti: il messaggio speciale <<create>> e la specifica che l'oggetto è nuovo ({new})

Il linguaggio del diagramma è in realtà più potente, infatti permette anche di inviare messaggi a gruppi di oggetti, specificare processi paralleli, e specificare operazioni sull'insieme degli oggetti:



Qui vediamo l'interazione specificata attraverso il classificatore i; con questo indichiamo che il messaggio print successivo deve essere eseguito per ogni oggetto identificabile con i, che come si vede accanto può essere specificato in diversi modi:

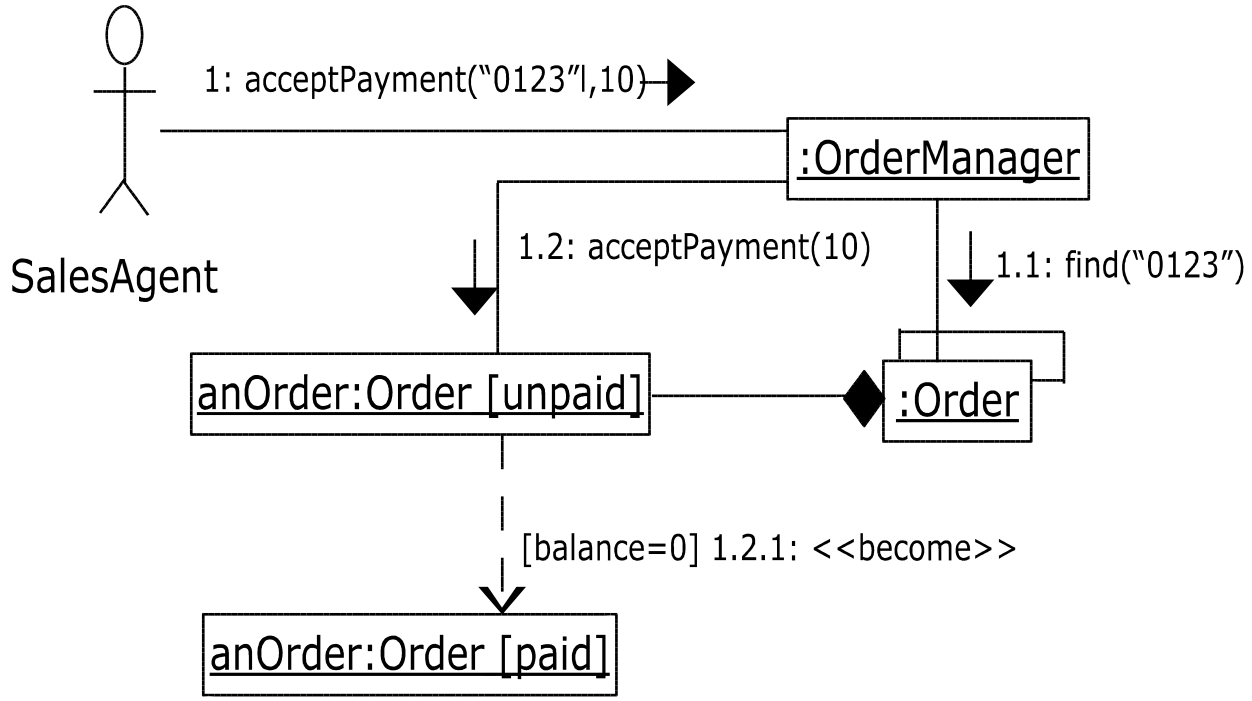
- Con [i:=1..n] indichiamo un ciclo per ogni elemento della collezione
- Con [i:=1..7] indichiamo un ciclo for da 1 a 7
- Con [while(boolExp)] indichiamo un ciclo while
- Con [until(boolExp)] indichiamo un ciclo until
- Con [for each (collExp)] indichiamo di ciclare su ogni oggetto preso dalla collezione risultante dall'esecuzione dell'espressione collExp

L'asterisco indica un chiamata dello stesso metodo a più elementi e, come dice la nota in basso, questo può avvenire in maniera parallela, oltre che sequenziale.

I metodi scritti in basso a destra, indicano il tipo di operazioni effettuabili sull'insieme stesso:

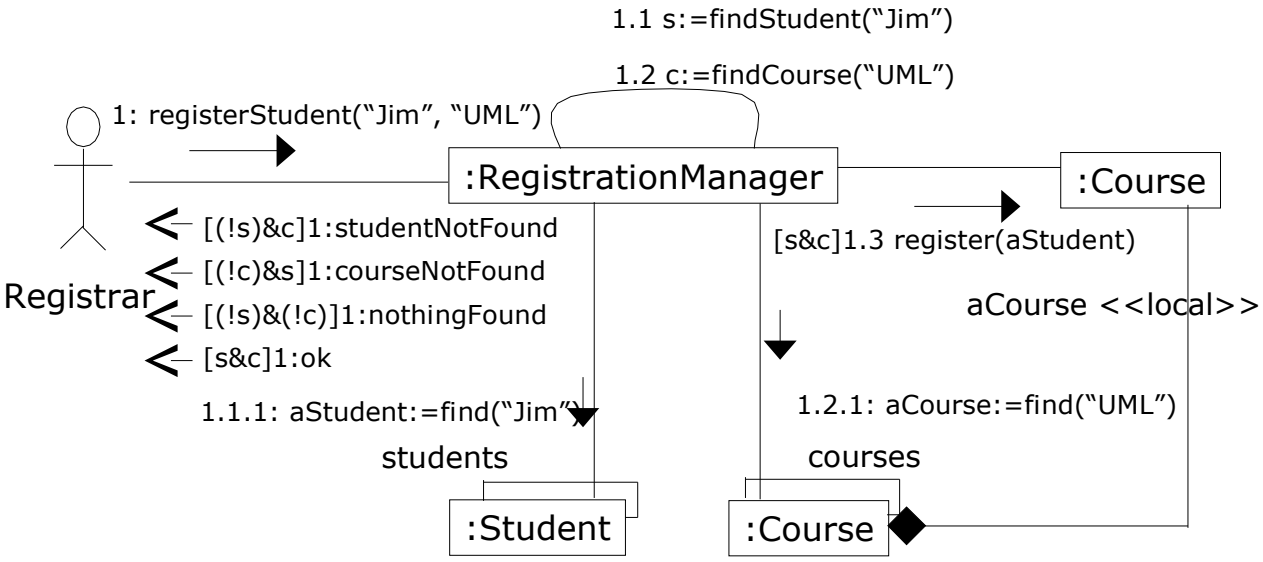
- find() - con il quale si cerca un elemento dentro l'insieme e viene restituito tale oggetto
- includes(anObject) – con il quale si controlla se un oggetto è dentro un insieme
- count() - con il quale si conta il numero di elementi.
- Ovviamente metodi di aggiunta e rimozione, i cui nomi, però, cambiano a seconda della semantica dell'insieme (pila, coda, insieme, tabella, ecc.)

Oltre a quello che abbiamo appena visto, nei diagrammi di collaborazione si possono indicare gli oggetti sempre attivi (che vengono indicati con la parola chiave Thread dentro il rettangolo che ospita il nome), messaggi asincroni, detti anche segnali (usando una freccia con metà punta) ed infine si possono indicare anche gli stati di un oggetto (necessario per capire alcune collaborazioni):



In questo caso vediamo l'oggetto anOrder passare dallo stato unpaid, allo stato paid attraverso una parola chiave become.

Ecco un ulteriore esempio di un diagramma di collaborazione:



Qui possiamo notare un uso massiccio delle cose dette precedentemente. Notare l'uso della parola <<local>> con la quale indichiamo che l'oggetto Course, appartenente all'insieme Course, è di fatto l'oggetto trovato attraverso il find().

Ultima cosa da dire sono i messaggi inviato da un oggetto a se stesso: essi, come abbiamo potuto vedere negli esempi precedenti, sono rappresentati come un cappio sopra il rettangolo che contiene il nome. Gli insiemi sono rappresentati, come abbiamo visto, da un doppio rettangolo, quasi ad indicare una pila di rettangoli.

Diagrammi di sequenza o collaborazione?

Quale dei due diagrammi usare, in realtà dipende da quale focus si vuole dare: se è più importante il tempo, si usa un diagramma di sequenza, altrimenti di collaborazione.

Trovare Classi ed Associazioni

N.d.A. Questo paragrafo non fa propriamente parte dello Unified Process, ma si ritiene utile per una migliore chiarificazione :)

Durante l'attività di analisi è importante riuscire a stabilire le classi e le associazioni e, di quest'ultime, il relativo tipo. Non esistono tecniche sempre valide, ma qualcosa possiamo dire.

Trovare le classi

Esistono alcune tecniche per trovare le classi:

Approccio basato su frasi nominali: Questo approccio consiglia all'analista di cercare nel documento i sostantivi o le frasi in cui il nome ha una prevalenza sulla parte verbale (sono in genere frasi di tipo assertivo). Ogni sostantivo è considerato una classe candidata. L'elenco viene quindi suddiviso in tre gruppi: classi rilevanti, classi incerte, classi irrilevanti.

Le classi irrilevanti sono quelle esterne al dominio del problema: i sostantivi corrispondenti appaiono nel discorso, ma non è possibile formulare un'asserzione che descriva il loro scopo nel contesto del sistema di analisi.

Le classi rilevanti sono quelle che manifestamente appartengono al dominio del problema: i sostantivi rappresentanti i nomi di tali classi appaiono con frequenza nel documento dei requisiti ed è possibile confermare l'importanza e la finalità di tali classi con la conoscenza generale del dominio applicativo e con l'analisi di sistemi simili.

Le classi incerte sono quelle che non è possibile classificare con sicurezza. Tali classi costituiscono la sfida maggiore: è necessario analizzarle ulteriormente per vedere se includerle od escluderle nella lista delle classi rilevanti. Proprio l'analisi di questo gruppo permette di ottenere un modello delle classi migliore di un altro.

Il difetto di questo sistema è che presuppone un documento dei requisiti completo ed accurato; in più l'analisi di un lungo documento testuale diventa ben presto pesante e non conduce ad un risultato completo ed accurato.

Approccio basato sulle strutture comuni: Questo approccio tenta di ricavare le classi da un'analisi generale degli oggetti del mondo reale. Normalmente si dividono gli oggetti in gruppi e si cerca di riempire questi gruppi. Due esempi:

Lista 1

Concetto: nozioni condivise, su cui esiste un accordo da una parte di un ampia comunità di persone.

Evento: fatti che accadono in una certo istante di tempo oppure ha una durata trascurabile rispetto ad una scala di riferimento.

Organizzazione: qualunque tipologia di raggruppamento o collezione di cose che ha una precisa finalità nel contesto in esame.

Persona: ruolo che una persona riveste nel sistema.

Posto: locazioni fisiche rilevanti per il sistema.

Lista 2

Fisica: è qualcosa che esiste veramente

Business: è rilevante per il sistema.

Logica: informazioni

Applicativa: che fa parte dell'applicazione

Computer: che riguarda la macchina

Comportamentale: che riguarda i comportamenti del sistema.

Come si vede dai due esempi non è sempre facile riempirli; inoltre i raggruppamenti non valgono per tutti i tipi di problemi; infine può dare adito a diversi dubbi. Tuttavia può essere utile per controllare delle classi trovate con altri sistemi.

Approccio guidato dai casi d'uso: Questo approccio sembrerebbe quello migliore, in quanto tutto l'Unified Process si fa guidare dai casi d'uso; tuttavia a questo stadio, esso assomiglia molto all'approccio basato sulle frasi nominali e ne condivide pregi e difetti. La differenza sta nel fatto che mentre con quell'altro si analizzava il documento dei requisiti, qui si analizza i caso d'uso stessi e ciò può portare ad un'analisi più attenta e meno soggetta ad errori (se i casi d'uso sono stati ben costruiti...).

Approccio CRC: Questo approccio prevede lo svolgimento di riunioni di gruppo, semplificate dal-

l'uso di schede costituite da tre sezioni: il *nome della classe*, scritto nella sezione più in alto, le *responsabilità*, elencate nella sezione di sinistra, i *collaboratori*, elencati nella sezione di destra. Le responsabilità sono i servizi (operazioni) che la classe è in grado di fornire alle altre classi. Tali responsabilità possono richiedere la collaborazione (servizi) di altre classi, le quali sono elencate nella sezione “collaboratori”.

L'approccio CRC è un processo dinamico durante il quale gli sviluppatori si scambiano le schede, compilandole col nome delle classi ed assegnando le responsabilità e le collaborazioni durante “l'esecuzione” di uno scenario di elaborazione (un caso d'uso). Allorquando siano necessari servizi non coperti da classi esistenti, è creata una nuova classe cui sono assegnate le appropriate responsabilità e collaborazioni. Se una classe diventa troppo appesantita, è suddivisa in un numero di classi più piccole.

A differenza di altri approcci, l'approccio CRC identifica le classi analizzando lo scambio di messaggi tra gli oggetti per realizzare le funzionalità del sistema. L'enfasi è posta su una distribuzione uniforme dell'”intelligenza” del sistema ed alcune classi potrebbero essere derivate da tale necessità “tecnica”, piuttosto che emergere come “oggetti di business”. In tal senso l'approccio CRC potrebbe essere adatto alla verifica di classi già scoperte per mezzo di altri metodi ed alla determinazione delle loro proprietà (implicate da responsabilità e collaborazioni).

Approccio misto: Nella pratica si usa un misto dei vari approcci. Questi partono dalla conoscenza del problema ed usano le tecniche sopra esposte per estrarre le classi.

Linee guida

1. Ciascuna classe deve avere una chiara funzionalità del sistema
2. Ogni classe descrive uno stampo per un insieme di oggetti. Classi singolari, per le quali è possibile immaginare solo un oggetto singolo, non sono propriamente buone classi. Ad esempio se il sistema è progettato per una singola organizzazione, l'esistenza di una classe Organizzazione non ha molto senso.
3. Ogni classe deve essere specificata tramite un insieme di attributi. In questo stadio si definiscono gli attributi essenziali per definire lo stato di una classe ed in suo compito. Altri attributi possono essere ignorati (ma l'analista deve preoccuparsi che l'informazione non vada perduta). È improbabile che tutti gli attributi siano definiti dai requisiti, ma in questo stadio è importante che essi corrispondano ai requisiti (gli altri saranno aggiunti più avanti).
4. Classi ed attributi sono elementi distinti di modellazione. Se un certo concetto debba essere modellato come classe o come attributo, dipende dal dominio applicativo. Ad esempio Colore di un automobile è normalmente percepito come un attributo della classe Automobile. Tuttavia in una carrozzeria, Colore è una classe con i propri attributi (lucentezza, densità, trasparenza, ecc.)
5. Ogni classe comprende un insieme di operazioni. Comunque a tale stadio non è necessario definire le operazioni.

Le associazioni

Le associazioni tra le classi vengono definite mentre si definiscono le classi; in generale se un attributo non è un tipo primitivo, allora esso indica un'associazione; se due classi collaborano fra di loro, allora vi è un'associazione. Particolare attenzione va posta sulle associazioni che rappresentano l'aggregazione, la composizione e la generalizzazione, in quanto semanticamente forti.

Per quanto riguarda aggregazione e composizione possiamo dire che (le parti in corsivo indicano la “parola chiave” che permette di decidere):

- Se una classe *è parte di* un'altra classe, allora questa classe è il contenuto di un'aggregazione o di una composizione.
- Se una classe *ha* o *raggruppa* altre classi, allora questa classe è il contenitore di un'aggregazione.
- Se una classe *possiede* o *possiede in modo esclusivo* altre classi, allora questa classe è il contenitore di una composizione.

Per quanto riguarda la generalizzazione possiamo dire che:

- Le caratteristiche comuni a più classi possono essere astratte in una classe più generica.
- Se una classe *è un tipo di* un'altra classe, allora questa è una sottoclasse della prima.
- Se una classe *può essere* un'altra classe, allora questa è una superclasse della prima.

Operazioni e stati

A questo stadio normalmente non è necessario definire operazioni e stati; tuttavia, in certi casi, quando è necessario alla comprensione del sistema, è possibile specificare le operazioni o gli stati disponibili.

Riepilogo

L'attività di Analisi si preoccupa ancora del cosa il sistema deve fare, ma si inizia a vederlo dal punto di vista dello sviluppatore. Anch'essa si svolge in più iterazioni a mano a mano che vengono scoperti i requisiti. L'unica cosa "scolpita sulla roccia" che si presuppone è l'architettura di base, definita alle prime iterazioni (se non adirittura alla prima) e portata avanti per tutto il resto del progetto.

Riepilogo dei documenti prodotti

Una serie di documenti è prodotta durante questa attività, tutti volti a capire cosa fa il sistema dal punto di vista dello sviluppatore.

Essi sono:

- **Architettura di base**: questa è l'architettura su cui si baserà tutto il progetto. Una volta che questa architettura è stabilita, essa non potrà più essere variata
- **Diagramma delle classi**: modella le classi e le associazioni del sistema.
- **Diagramma di sequenza e/o collaborazione**: modella il comportamento delle classi per la risoluzione dei casi d'uso
- **Diagramma dei package d'analisi**: modella come sono accoppiate le classi.
- **Diagramma delle attività e degli stati**: caso mai siano necessari alla comprensione.
- **Documentazione aggiuntiva**: Ad esempio i fogli CRC, se si usa questo approccio.

Design (Progettazione)

L'attività di Design verte sul come deve funzionare il sistema, piuttosto che sul cosa deve fare. Questa attività ha maggior fuoco nella parte finale della fase di Elaboration e nella prima parte della fase di Construction, ma avendo aspetti comuni all'attività di analisi, spesso è svolta in parallelo.

L'attività di Design riprende, dunque, i documenti prodotti dalla fase di analisi e vi aggiunge elementi appartenenti al dominio della soluzione. Lo scopo è aggiungere al cosa il come: trovato cosa deve fare il sistema, bisogna trovare come farglielo fare. Vengono perciò aggiunte anche tutte quelle

classi di comunicazione o di database, interfacce, utilità, ecc. In questa fase è necessario analizzare anche fattori come la performance, il throughtput, la disponibilità, la possibile distribuzione tra nodi diversi e la comunicazione, il deadlock, il liveness, la riconfigurazione, la persistenza, la sicurezza, la distribuzione trasparente degli oggetti, l'individuazione ed il recupero degli errori, la gestione delle transizioni, ecc.

Il modello di Design va a guardare l'implementazione tecnica, osservando i requisiti non funzionali e l'architettura sottostante, mette maggiore risalto sulle interfacce ed in generale approfondisce il modello di analisi andando a raffinarlo ed a scomporlo ulteriormente. È durante questa attività (anche se è tra le prime cose fatte, perché è su di essa che si appoggia il sistema) che l'architettura di base si evolve in maniera definitiva andando a decidere l'esatto tipo di sistema (client-server, distribuito, ecc.).

Prima di addentrarci in questa attività, ci poniamo una domanda: per quale motivo bisogna mantenere due modelli, visto che uno è un raffinamento dell'altro?

Beh, il modello di analisi è più semplice da comprendere (contiene dal 1% al 10% di tutte le classi del sistema) ed è quindi utile per: spiegare il progetto a nuove persone, comprendere il sistema successivamente, verificare la soddisfazione dei requisiti e tracciabilità, pianificare il mantenimento ed i miglioramenti, comprendere l'architettura logica, affidare all'esterno la costruzione, avere una visione sul problema e quindi individuare più facilmente le fasi critiche. Purtroppo mantenere due modelli è piuttosto oneroso e dispendioso.

Esistono 4 soluzioni al mantenimento dei due modelli:

1. Raffinare il modello di analisi nel modello di design
   - Unico modello: si perde quello di analisi
2. Raffinare il modello di analisi in design, usare uno strumento CASE per ricostruire il modello di analisi
   - Modello di analisi ricostruito non garantito essere coerente con quello originale
3. Congelare il modello di analisi, copiarlo e raffinarlo
   - Due modelli, spesso non coerenti
4. Due modelli separati
   - Due modelli coerenti, ma con mantenimento della coerenza costoso

Se il progetto è grosso, complesso o strategico, normalmente richiede anche il modello di analisi. In questo caso scegliere di mantenere il modello di analisi (e sceglierlo di congelarlo o di aggiornarlo).

Se il progetto è piccolo (meno di 200 classi di Design) ancora comprensibili, non è strategico, od ha vita breve (ma molti sistemi vivono più di quanto atteso!) allora è meglio lasciar perdere di mantenere il modello di analisi. In questo caso si usa il modello di analisi come base per quello di design (l'eventuale riscostruzione viene lasciata ad uno strumento CASE).

L'attività di Design si basa su:

- classi di progetto
- interfacce
- sottosistemi di progetto
- realizzazioni di casi di uso – progetto

Classi di progetto

Le classi di progetto (o di design) partono da quelle di analisi e le completano con tutto quello che serve (visibilità, valori di default, parametri, metodi di supporto, costruttori, ecc.). Si aggiungono le informazioni di implementazione e sulle classi di supporto, come classi per l'accesso ai database, la realizzazione di GUI, di accesso al sistema, ecc.

Una classe di design deve essere completa e sufficiente, primitiva, avere alta coesione e basso accoppiamento.

Completezza e sufficienza

I metodi pubblici definiscono il contratto tra le classi ed i clienti delle classi; devono dunque avere nomi significativi, in quanto essi creano attese che devono essere soddisfatte; i metodi della classe devono essere focalizzati sullo scopo della classe.

La completezza assicura che la classe soddisfi tutto quello che l'utente si aspetta da essa.

La sufficienza mantiene la classe il più semplice e focalizzata possibile.

La regola d'oro per la completezza e la sufficienza è che la classe deve fare quello che l'utente si aspetta dalla classe, né tanto di più, né tanto di meno.

Primitività

La classe deve offrire servizi primitivi (cioè non ulteriormente scomponibili) ed atomici (cioè tali che l'operazione non può essere interrotta), non deve avere metodi diversi per fare la stessa cosa.

La primitività è rilassabile solo quando vi è un vantaggio in termini di prestazioni significativo.

Alta coesione

Le classi devono realizzare un singolo concetto, avere metodi focalizzati sullo scopo della classe.

Se la classe ha diverse responsabilità, creare classi "helper" a cui delegare.

Basso accoppiamento

Le classi devono essere associate con altre classi solo se permettono di realizzare responsabilità, oppure solo se c'è un vero collegamento semantico fra loro.

Alcune associazioni vengono dal modello di analisi, altre da vincoli di implementazione. Quest'ultime devono essere valutate molto attentamente.

L'alto accoppiamento all'interno del package è ammissibile.

Progettazione delle associazioni

Durante l'attività di analisi si sono create delle associazioni. Durante la fase di design queste associazioni vengono completate e trasformate in implementazione.

Dato che normalmente i linguaggi di programmazione non hanno un supporto per le associazioni, esse dovranno essere trasformate in aggregazioni e composizioni. L'unico caso in cui vengono lasciate come associazioni è quando sono cicliche (un'aggregazione od una composizione non possono essere cicliche).

Prima di iniziare a trasformare le associazioni, esse devono tutte avere:

- Navigabilità
- Molteplicità da entrambi i lati

Tutte le associazioni possono avere un nome di ruolo alla fine del lato bersaglio.

Se si seguono questi accorgimenti è possibile effettuare una migliore implementazione delle associazioni.

Dunque i passi da seguire sono:

- Aggiungere le molteplicità ed il nome di ruolo se sono assenti
- Decidere quale lato dell'associazione sarà il tutto e quale la parte
- Guardare la molteplicità dalla parte del tutto: se è esattamente uno allora può essere una composizione, altrimenti deve essere un'aggregazione
- Aggiungere la navigabilità dal tutto alla parte.

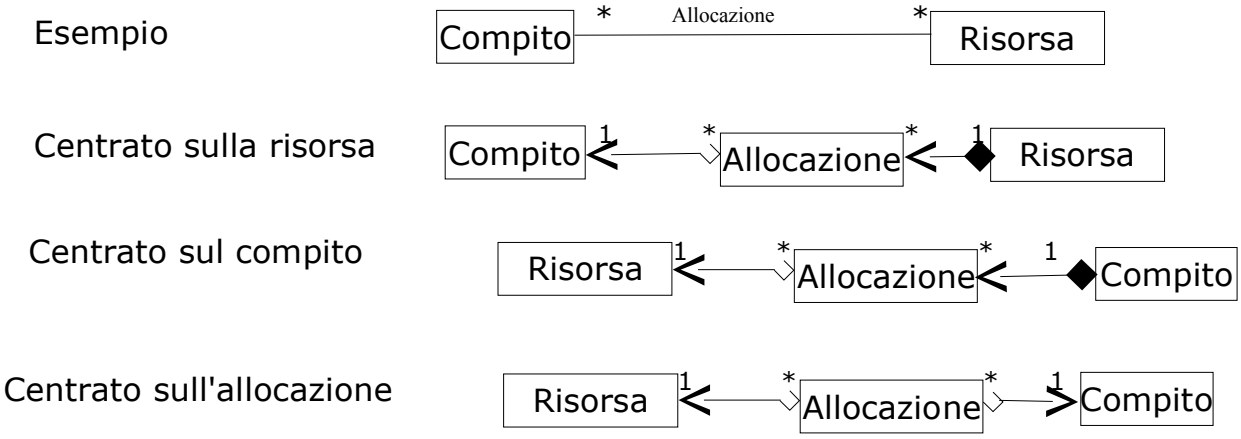
Vediamo una ad una le varie associazioni.

Associazioni uno ad uno: Le associazioni uno ad uno sono le più facili da implementare, in quanto implicano una forte relazione tra le due classi: questa dunque sarà una composizione oppure un oggetto sarà attributo di un altro (se non è una classe particolarmente importante).

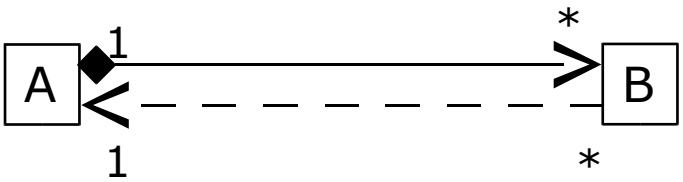
Associazioni molti ad uno: Se non hanno cicli le associazioni molti ad uno sono rappresentate come aggregazioni; in caso contrario uno degli archi rimane come associazione (quest'ultimo caso probabilmente indica una callback).

Associazioni uno a molti: Queste associazioni rappresentano degli insiemi nel lato molti. Per implementarla si usa il supporto nativo del linguaggio e/o si creano delle classi collezione (vedi classi collezione, sotto).

Associazioni molti a molti: Queste associazioni non hanno alcun riferimento in nessun linguaggio; di conseguenza bisogna trasformarle in complicate relazioni molti ad uno.



Associazioni bidirezionali: Anche queste associazioni non hanno alcun riferimento nei linguaggi di programmazione e come tali vanno modellate in altro modo. Dato che normalmente rappresentano callback la rappresentazione che viene usata è la seguente:



Con la freccia tratteggiata indichiamo che l'intero si passa come parametro ad un metodo della parte, o parte crea istanza intero in uno dei suoi metodi.

Classi di associazione: Le classi di associazione non hanno una trasformazione immediata in composizioni, aggregazioni, od altro. Una classe di associazione dovrà, dunque, essere una classe a parte ed avere un qualche tipo di legame con le classi che avevano l'associazione in cui questa classe era classe di associazione. Una volta deciso come implementare la classe di associazione, bisognerà inserire dei vincoli nel modello per ripristinare il significato semantico.

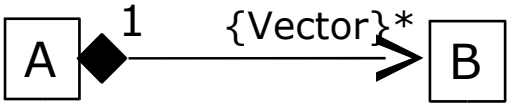
Classi collezione

Le classi collezione sono specializzate per maneggiare collezioni di altri oggetti. hanno metodi per aggiungere oggetti alla collezione, rimuovere oggetti dalla collezione, recuperare oggetti dalla collezione, muoversi attraverso gli oggetti della collezione.

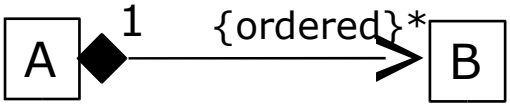
Esistono tanti tipi di classi collezione ognuno con una propria prerogativa. UML supporta degli stereotipi per indicare che semantica usa la classe collezione. Esistono tre modi per rappresentare una classe collezione in UML:

Modellazione esplicita classe collezione

Specificare un'implementazione il cui prodotto CASE provvederà a rendere una classe colezione esplicita



Specificare la semantica della collezione, ma non l'implementazione



Gli stereotipi (cioè quei valori messi tra parentesi graffe) che l'UML supporta per le classi collezione sono:

- {ordered} – è una collezione ordinata
- {unordered} – è una collezione non ordinata
- {set} – è un insieme
- {lifo} – è una pila
- {queue} – è una coda

- {bag} – come {set} ma ammette duplicati
- {sequence} – è una sequenza ordinata, indicizzata e che ammette duplicati (l'array è un buon esempio).

Notare che solo i primi due sono standard UML, gli altri sono usati ma non standard, quindi dovranno essere specificati nel documento che accompagna il modello (o tramite l'OCL, come vedremo più avanti).

L'ultimo tipo che viene molto usato è la mappa o dizionario. Essa non ha né una semantica UML né OCL, dunque dovrà essere specificata nei dettagli e sarà, probabilmente, dipendente dal linguaggio usato per l'implementazione. La mappa lavora mantenendo un insieme di nodi, dove ogni nodo punta a due oggetti: una chiave ed un oggetto valore. È ottimizzata per trovare gli oggetti valore in base alla chiave.

Note e vincoli

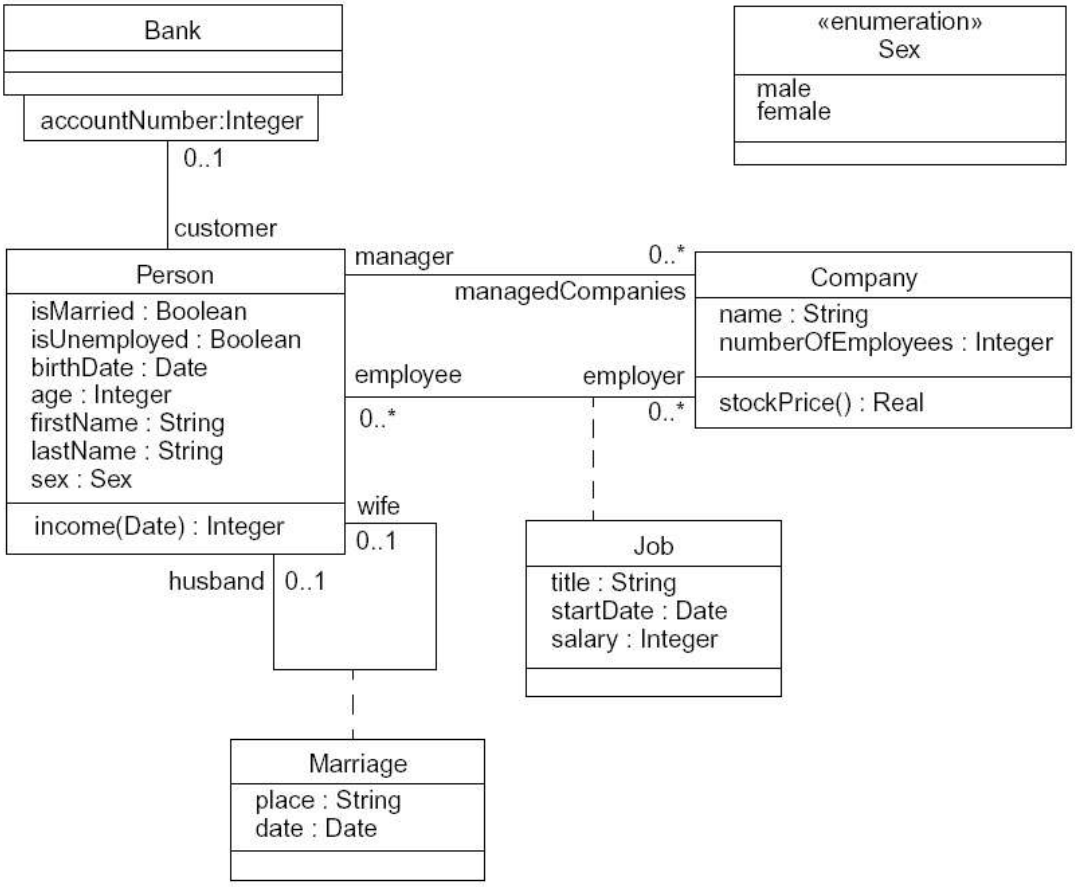
Non sempre è possibile riuscire ad esprimere tutta la modellazione attraverso solo i diagrammi; a volte per specificare la semantica del diagramma è necessario introdurci note o vincoli. UML permette di fare questo collegando l'elemento su cui si vuole mettere una nota, attraverso una linea tratteggiata, ad un rettangolo con un angolo piegato. All'interno di questo rettangolo vi sarà scritta la nota. Usare lo stereotipo <<constraint>> implica che quella nota non è solo un qualcosa per indicare il significato dell'elemento, ma un vero e proprio vincolo. Spesso risulta necessario utilizzare i vincoli là dove il diagramma delle classi di design perde parte della sua semantica (si pensi alle classi di associazione o alle relazioni molti a molti, come visto prima).

Le note ed i vincoli vengono solitamente descritti in linguaggio naturale. Tuttavia a volte il linguaggio naturale può essere fuorviante o lasciare adito a dubbi, quando esprime vincoli. Per questo è stato sviluppato un apposito linguaggio formale per esprimerli: esso prende il nome di OCL.

OCL

L'Object Contrastaint Language (OCL) è il linguaggio formale utilizzato per esprimere vincoli (detti anche invaranti), pre e post condizioni, guardie. Esso è usato anche per esprimere vincoli globali al sistema e non solo ad elementi appartenenti ad un diagramma. Proprio per questo motivo esso è spesso realizzato come documento separato e le note UML inserite nel diagramma rimandano a questo documento. Per far ciò, ovviamente, ogni vincolo dovrà avere un nome od un altro tipo di identificatore univoco.

Nel corso della trattazione ci riferiremo a questo esempio per esplicitare il linguaggio OCL(visualizzato nella pagina successiva).



Ogni espressione OCL inizia con una dichiarazione di contesto: esso indica a quale elemento i vincoli fanno riferimento. Esso è indicato con la parola chiave *context*. La parola *self* è un sistema per riferirsi al contesto senza dirlo esplicitamente; ad esempio:

context Company

inv: self.numberOfEmployees>50

indica che dobbiamo prendere l'attributo NumberOfEmployees nel contesto di Company.

La parola chiave self può essere sottointesa se il contesto rimane chiaro.

Se l'attributo fosse statico si potrebbe scrivere:

context Company

inv: Company.numberOfEmployees>50

Il contesto può essere riferito ad un nome esplicito:

context c: Company

La dichiarazione di un invariante è presentata con la parola chiave *Invariant* (a volte abbreviata con *inv*). Tutte le invarianti devono "restituire" un valore booleano. Anche un'invariante può essere riferita con un nome:

context c: Company

inv enoughEmployees: numberOfEmployees>50

Le pre e post condizioni sono presentate con la parola chiave *precondition* e *postcondition* (a volte abbreviate con *pre* e *post*). Le pre e post condizioni devono essere associate ad un metodo. Per far questo si usa la seguente sintassi di contesto:

context TypeName::operationName(param1: Type, ...): ReturnType

pre: param1>...

post: result=...

Ad esempio:

context Person::income(d: Date): Integer

post: result=5000

Il nome *self* può essere utilizzato per riferirsi all'oggetto su cui l'operazione viene chiamata; la parola chiave *result* si riferisce al risultato dell'operazione, se c'è ne uno. Anche le pre e post condizioni possono avere un nome.

context Typename::operationName(param1 : Type1, ...): ReturnType
pre parameterOk: param1 > ...
post resultOk: result = ...

A volte il contesto deve essere riferito ad uno specifico package; la parola chiave usata è, per l'appunto, *package*.

Ecco un esempio:

package Companies

context Company

inv: self.manager.isUnemployed = false

Il contesto è la classe Company del package Companies; l'invariante è: la proprietà isUnemployed dell'oggetto di tipo Manager referenziato da Company (self) deve essere false sempre.

La potenza dell'OCL sta nel fatto che è possibile definire vincoli molto complessi; infatti prevede di utilizzare variabili e tipi per esprimere condizioni, nonché l'esecuzione su operazioni riferite ad insiemi.Oltre a ciò è possibile eseguire operazioni e richiamare metodi, purché essi non abbiamo alcun side-effect sull'oggetto e non ne cambino lo stato.

I tipi disponibili (con le loro operazioni) sono

- booleani su cui possiamo eseguire operazioni di
  - uguaglianza (b1=b2) – serve a vedere se due booleani sono uguali
  - and (b1 and b2) – restituisce vero se entrambi sono veri (nota: usa la short-circuit evaluation)
  - or (b1 or b2) – restituisce vero se uno dei due è vero (nota: usa la short-circuit evaluation)
  - xor (b1 xor b2) – restituisce vero se uno dei due è vero, ma non entrambi
  - not (b1) – restituisce vero se b1 è falso
  - implicazione (b1 implies b2) – restituisce vero se b1 è falso o se entrambi sono veri

  - if then else (if b1 then oclExpression1 else oclExpression2 endif) – restituisce il risultato della valutazione di oclExpression1 se b1 è vero, altrimenti restituisce il risultato della valutazione di oclExpression2.
- Interi su cui possiamo eseguire operazioni di
  - uguaglianza (i1=i2) – restituisce vero se i due numeri sono uguali
  - diversità (i1<>i2) – restituisce vero se i due numeri sono diversi
  - minoranza (i1<=i2) – restituisce vero se i1 è più piccolo od uguale ad i2
  - minoranza stretta (i1<i2) – restituisce vero se i1 è più piccolo di i2
  - maggioranza (i1>=i2) – restituisce vero se i1 è più grande od uguale ad i2
  - maggioranza stretta (i1>i2) – restituisce vero se i1 è più grande di i2
  - meno unario (-i1) – rende un numero positivo, negativo e viceversa
  - somma (i1+i2) – restituisce la somma
  - sottrazione (i1-i2) – restituisce la sottrazione
  - moltiplicazione (i1*i2) – restituisce la moltiplicazione
  - divisione (i1/i2) – restituisce la divisione (come numero reale)
  - valore assoluto (i.abs()) – restituisce il valore assoluto di i
  - divisione intera (i1.div(i2)) – restituisce il quoziente della divisione
  - modulo (i1.mod(i2)) – restituisce il resto della divisione
  - massimo (i1.max(i2)) – restituisce il più grande tra i1 ed i2
  - minimo (i1.min(i2)) – restituisce il più piccolo tra i1 ed i2
- reali su cui possiamo eseguire operazioni di
  - ugualianza (r1=r2) – restituisce vero se i due numeri sono uguali
  - diversità (r1<>r2) – restituisce vero se i due numeri sono diversi
  - minoranza (r1<=r2) – restituisce vero se r1 è più piccolo od uguale ad r2
  - minoranza stretta (r1<r2) – restituisce vero se r1 è più piccolo di r2
  - maggioranza (r1>=r2) – restituisce vero se r1 è più grande od uguale ad r2
  - maggioranza stretta (r1>r2) – restituisce vero se r1 è più grande di r2
  - meno unario (-r1) – rende un numero positivo, negativo e viceversa
  - somma (r1+r2) – restituisce la somma
  - sottrazione (r1-r2) – restituisce la sottrazione
  - moltiplicazione (r1*r2) – restituisce la moltiplicazione
  - divisione (r1/r2) – restituisce la divisione
  - valore assoluto (r.abs()) – restituisce il valore assoluto di r
  - floor (r1.floor()) – restituisce il più grande intero che è più piccolo od uguale ad r1

  - arrotondamento (r1.round()) – effettua l'arrotondamento matematico restituendo di conseguenza l'intero più vicino ad r1.
  - massimo (r1.max(r2)) – restituisce il più grande tra r1 ed r2
  - minimo (r1.min(r2)) – restituisce il più piccolo tra r1 ed r2
- stringhe su cui possiamo eseguire operazioni di
  - ugualianza (s1=s2) – restituisce vero se s1 è uguale ad s2
  - lunghezza della stringa (s.size()) – restituisce il numero di caratteri della stringa
  - concatenazione (s1.concat(s2)) – restituisce una stringa che è la concatenazione di s1 ed s2
  - tutto maiuscole (s.toUpper()) – restituisce una stringa in cui tutti i caratteri sono in maiuscolo
  - tutto minuscole (s.toLower()) – restituisce una stringa in cui tutti i caratteri sono in minuscolo
  - sottostringhe (s.substring(i1,i2)) – restituisce una stringa che è la sottostringa compresa tra il carattere numero i1 ed il carattere numero i2 (compresi)
- enumerazione – questo rappresenta una classe utilizzata per enumerare; nell'esempio la classe Sex rappresenta un'enumerazione, cioè sappiamo che l'attributo sex di tipo Sex della classe Person, potrà assumere solo i valori Male e Female.
- Oggetti – dato che OCL serve ad UML, qualsiasi oggetto può essere usato come tipo.

Per usare un'enumerazione in OCL si usa il seguente modo:

```
context Person
inv: sex = Sex::male
```

Con oclExpression si intende qualsiasi espressione in OCL, la valutazione di questa espressione ritornerà sempre un tipo (d'altronde l'OCL serve a verificare condizioni, non ad eseguire operazioni).

Variabile ed operazioni in OCL

In OCL è possibile definirsi variabili ed operazioni, in modo da poter rendere il modello più leggibile; per far questo su usa il costrutto *let*. Ad esempio:

```
context Person inv:
let
	income : Integer = self.job.salary->sum()
let
	hasTitle(t : String) : Boolean = self.job->exists(title = t)
in
	if isUnemployed then
		self.income < 100
	else
		self.income >= 100 and self.hasTitle('manager')
	endif
```

è decisamente più leggibile di

```
context Person inv:
if isUnemployed then
```

```
        self.job.salary->sum() < 100
else
        self.job.salary->sum() >= 100 and self.job->exist(title = ‘manager’)
endif
```

Le espressioni let possono essere usate sia dentro invarianti che dentro pre e post condizioni. Tuttavia esse esistono solo all'interno del vincolo; per farle esistere all'interno di tutto il contesto bisogna usare il costrutto *definition* (abbreviato con *def*)

```
context Person def:
let
        income : Integer = self.job.salary->sum()
let
        hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

In nessun caso i nomi devono essere in conflitto.

Tipi e commenti

Prima di passare a vedere come si comporta OCL con gli oggetti (che è la parte più potente e complicata) è meglio sapere un paio di cose sui tipi.

OCL è un linguaggio fortemente tipato; ogni espressione ha un tipo e normalmente non vi è modo di confrontare od assegnare tipi diversi. Tuttavia esiste un'eccezione a questa regola ed è la cosidetta conformazione dei tipi: ogni tipo si trova all'interno di una gerarchia, quelli che si trovano in una gerarchia più bassa possono essere usati in luogo a quelli nella gerarchia più alta. Questo si uniforma al concetto di generalizzazione. Proprio grazie a questo possiamo usare

- interi al posto dei reali
- insiemi al posto di collezioni (vedremo più avanti cosa intendiamo esattamente)
- bag al posto di collezioni (vedremo più avanti cosa intendiamo esattamente)
- sequenze al posto di collezioni (vedremo più avanti cosa intendiamo esattamente)
- sottoclassi al posto di superclassi

Nessun altro tipo di sostituzione è ammesso!

Come per i tipi interi e reali è possibile effettuare confronti, così è possibile controllare se due oggetti sono lo stesso oppure sono due oggetti diversi, attraverso l'uguaglianza (=) o la disuguaglianza (<>).

I commenti in OCL vengono fatti precedere dal doppio trattino (--) e valgono fino alla fine della linea.

Oggetti

Bene, è giunto il momento di addentrarci nell'uso degli oggetti, associazioni e proprietà.

In OCL possiamo accedere alle classi, come tipi, ed ai relativi oggetti per prendere i valori che ci servono per controllare invarianti, pre e post condizioni. Proprio per questo, dato un oggetto, possiamo accedere alle sue proprietà (od attributi), operazioni (o metodi) senza side-effect, e recuperare altri oggetti collegati a quello di contesto attraverso le sue associazioni.

Per accedere ad un attributo si usa il suo nome, eventualmente separandolo con un punto dal contesto di riferimento. Ad esempio:

```
context Person
inv: self.age>0
```

L'accesso alle operazioni avviene allo stesso modo; tuttavia, siccome possono avere parametri, avranno sempre una coppia di parentesi. Ad esempio

```
context Company inv:
self.stockPrice() > 0
```

Se il metodo ha dei parametri essi saranno descritti all'interno delle parentesi eventuamente separandoli con virgole.

Navigazione tra le associazioni

Per quanto riguarda le associazioni, noi possiamo navigare tra le associazioni del diagramma delle classi, per recuperare oggetti collegati al contesto di riferimento ed accedere alle loro proprietà e metodi. Per far questo noi navighiamo sull'associazione usando il nome del ruolo dell'oggetto collegato. Il risultato può essere un oggetto od un insieme a seconda della molteplicità (se è 0..1 oppure 1 allora è un oggetto, altrimenti è un insieme). Ad esempio

```
context Company
inv: self.manager.isUnemployed = false
inv: self.employee->notEmpty()
```

Nella prima invariante self.manager è un oggetto in quanto la sua molteplicità è 1, mentre self.employee è un insieme di oggetti Person, in quanto la sua molteplicità è 0..*

Faccio notare che nel caso in cui il tipo di associazione sia classificato come {ordered} e la classe alla fine dell'associazione ha molteplicità maggiore di uno, allora il risultato sarà una sequenza (Per capire meglio i concetti di insieme e sequenza e come possano essere utilizzati, vedere oltre).

Nel caso in cui il nome di ruolo manchi, è possibile utilizzare il nome della classe (scritto in lettere minuscole) se questo non crea ambiguità (ad esempio, nelle associazioni riflessive, il nome del ruolo diventa obbligatorio).

Nelle associazioni 0..1 è possibile considerare l'oggetto collegato sia come insieme che come singolo elemento. In altre parole è possibile usare sia

```
context Company inv:
self.manager->size() = 1
```

che

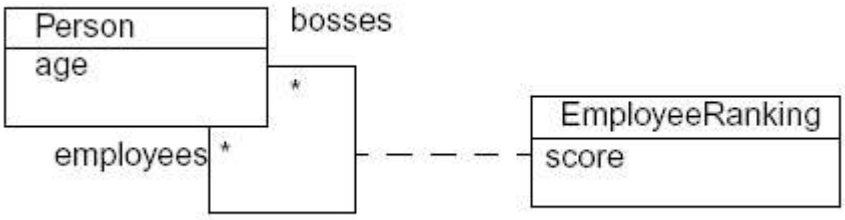
```
context Company inv:
self.manager.age> 40
```

Questo permette di fare controlli sull'esistenza o meno dell'oggetto collegato:

```
context Person inv:
self.wife->notEmpty() implies self.wife.sex = Sex::female
```

Classi di associazione

Per permettere la navigazione verso classi di associazioni, essi si identificano con il loro nome scritto a caratteri minuscoli (come per quando manca il nome di ruolo nell'associazione tra classi normali). Tuttavia è possibile che vi siano ambiguità dovute alle associazioni ricorsive; ad esempio



noi possiamo navigare verso la classe EmployeeRanking da due direzioni, quella col ruolo bosses e quella col ruolo employees. Per rimuovere l'ambiguità si mette dopo il nome della classe di associazione, il nome del ruolo da cui stiamo partendo, racchiuso tra parentesi quadre. Quindi possiamo usare:

```
context Person inv:
self.employeeRanking[bosses]->sum() > 0
```

Per intendere che lo vediamo dal lato dei bosses, oppure

```
context Person inv:
self.employeeRanking[employees]->sum() > 0
```

Per intendere che lo vediamo dal lato degli employee.

Due note: è possibile usare la sintassi delle parentesi quadre anche senza che vi sia ambiguità; inoltre questo tipo di navigazione restituisce l'insieme di tutti gli oggetti della classe di associazione che sono collegati agli oggetti che hanno il ruolo specificato.

La navigazione dalle classi di associazione avviene come per le classi normali.

Associazioni qualificate

Per navigare attraverso associazioni qualificate abbiamo due modi: il primo restituisce l'insieme di tutti gli oggetti collegati al contesto ed è effettuato semplicemente indicando semplicemente il nome del ruolo:

```
context Bank inv:
self.customer ...
```

Il secondo sistema permette di riferirsi ad un oggetto specifico ed è effettuato indicando oltre al nome del ruolo l'identificatore specifico tra parentesi quadre:

```
context Bank inv:
self.customer[8764423]...
```

Ovviamente se c'è più di un qualificatore bisogna specificare tutti i valori per tutti i classificatori, separandoli con virgole. Non è possibile specificare solo una parte dei qualificatori.

Package

A volte può capitare che due classi abbiano lo stesso nome, ma appertengano a due package diversi. Per evitare ambiguità, in OCL si usa la sintassi PakageName::ClassName, ovunque ce ne sia bisogno; in caso di package annidati è possibile usare PackageName1::PackageName2:: ... ::PackageNameN::ClassName

Sottoclassi e superclassi

È possibile usare le proprietà ed i metodi di una superclasse:

```
context B inv:
self.oclAsType(A).p1 -- accede alla proprietà p1 definita in A
self.p1 -- accede alla proprietà p1 definita in B
```

Ovviamente supponendo che B sia sottoclasse di A

Metodi predefiniti

In OCL esistono alcuni speciali metodi che hanno tutti gli oggetti

oclIsTypeOf(t : OclType) : Boolean
oclIsKindOf(t : OclType) : Boolean
oclInState(s : OclState) : Boolean
oclIsNew() : Boolean
oclAsType(t : OclType) : instance of OclType
supertypes(): insieme di tipi
allSupertypes(): insieme di tipi

Il primo permette di controllare se il tipo dell'oggetto e t siano lo stesso:

```
context Person
inv: self.oclIsTypeOf( Person ) --  true
inv: self.oclIsTypeOf( Company) --  false
```

Il secondo determina se t è sottotipo (sottoclasse) dell'oggetto

Il terzo determina se l'oggetto si trova nello stato s, dove s è il nome dato allo stato nel diagramma degli stati. In caso di stati composti, è possibile richiedere se l'oggetto si trovi in un sottostato separando i nomi con doppi due punti (::).

Il quarto può essere usato solo nelle postcondizioni e determina se l'oggetto è stato creato durante l'operazione.

Il quinto traforma un tipo in un altro, o per meglio dire, considera un oggetto come se appartenesse ad un'altra classe purché compatibile (cioè deve essere sottoclasse).

Il sesto restituisce l'insieme di tutti le superclassi dell'oggetto (se non ci sono superclassi, restituisce l'insieme vuoto, se l'oggetto è sottoclasse di una sola classe, l'insieme avrà un elemento solo).

Il settimo ripercorre tutto l'albero di derivazione fino alla radice e ritorna l'insieme di tutti gli stati che incontra nei cammini.

Valori precedenti nelle postcondizioni

In certi casi è necessario riferirsi ad un valore precedente all'elaborazione. Ad esempio vogliamo che un metodo birthdayHappens sommi 1 all'età (age) precedente; questo fa si che la postcondizione sia age=age+1, ma mettendola così sembrerebbe che l'età corrente deve essere uguale all'età corrente più uno (un po' difficile). Per eliminare l'ambiguità OCL usa la parola chiave @pre da posporre al nome:

```
context Person::birthdayHappens()
post: age = age@pre + 1
```

così l'ambiguità è tolta: l'età corrente deve essere uguale all'età precedente più uno.

La parola chiave @pre può essere anche messa dopo un metodo ad indicare che interessa il risultato dell'operazione prima dell'elelaborazione corrente.

```
context Company::hireEmployee(p : Person)
pre : not employee->includes(p)
post: employees->includes(p) and
      stockprice() = stockprice@pre() + 10
```

Nel caso in cui la proprietà sia un oggetto possiamo avere:

a.b@pre.c -- prende il vecchio valore delle proprietà b di a; sia esso x
-- e poi prende il nuovo valore della proprietà c di x
a.b@pre.c@pre --prende il vecchio valore della proprietà b di a; sia esso x
-- e poi prende il vecchio valore della proprietà c di x

Collezioni

È giunto il momento di parlare di insiemi (Set), sequenze (Sequence) e "valigie" (Bag).

Il risultato della navigazione attraverso associazioni è un insieme (Set), se l'associazione è identificata dalla parola {ordered}, allora il risultato è una sequenza (Sequence). La combinazione di diverse navigazioni ottiene una valigia (Bag). Proprio per questo motivo questi tre oggetti sono predefiniti (anche come tipo) in OCL. Tutti e tre vengono accumunati col nome (ed il tipo) di Collezione, in quanto il loro compito è contenere oggetti.

In OCL abbiamo dunque tre tipi di collezione: Set, Sequence e Bag. Il primo è l'insieme a cui tutti siamo abituati, il secondo rappresenta una sequenza di oggetti, mentre il terzo è un insieme che ammette duplicati. È possibile creare questi oggetti al volo:

Per l'insieme:

Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }

Per la sequenza:

Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }

Per la valigia:

Bag {1 , 3 , 4, 3, 5 }

Dato che una sequenza consecutiva di interi è un po' tedioso crearla con questa sintassi, ne è stata inventata un'apposita: si indica il primo numero si mette una sequenza di due punti (..) e si mette l'ultimo numero. Quindi le seguenti due creazioni

Sequence{ 1..(6 + 4) }

Sequence{ 1..10 }

sono identiche a

Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Esistono dunque tre modi per ottenere una collezione (che sia un Set, un Bag od una Sequence):

1. Creandolo, come abbiamo visto sopra
2. Navigando tra le associazioni, come abbiamo visto precedentemente

3. Attraverso operazioni che restituiscono collezioni, come quella di unione di insiemi.

N.B.: In OCL non esistono collezioni di collezioni, quindi

Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }

è uguale a

Set{ 1, 2, 3, 4, 5, 6 }

Per quanto riguarda la conformità di tipo abbiamo che ognuno dei tre oggetti si conforma al tipo collezione (Collection), ma non tra di loro.

Ognuno di questi oggetti ha le proprie operazioni; per accedere alle operazioni di un insieme, in OCL si usa la sintassi della freccia (->) come abbiamo visto in tutti gli esempi sopra.

Operazioni comuni a tutte le collezioni

Iniziamo, dunque, dalle operazioni comuni a tutte le collezioni:

Select

Con questa operazione restituiamo una collezione dello stesso tipo di quella di partenza in cui sono verificate tutte le proprietà indicate. La sintassi è

collection->select(boolean-expression)

Quindi il risultato sarà una collezione in cui tutti gli elementi rendono vera la boolean-expression.

Nota: la boolean-expression può essere anche data da proprietà di tipo bool o metodi che restituiscono un tipo bool.

Ad esempio:

```
context Company inv:
self.employee->select(age > 50)->notEmpty()
```

Abbiamo che verranno selezionati tutti gli employee che hanno la proprietà age più grande di 50 e poi si controlla che l'insieme non sia vuoto.

Un'altra sintassi che permette di riferirsi direttamente agli oggetti che stiamo scorrendo è

collection->select(v | boolean-expression-with-v)

v è la variabile di iterazione.

Ad esempio

```
context Company inv:
self.employee->select(p | p.age > 50)->notEmpty()
```

Indica di prendere tutti gli elementi p dell'insieme employee tali che la proprietà age di p sia maggiore di 50. Si controlla poi che l'insieme risultante non sia vuoto.

L'ultima sintassi permette di tipare la variabile d'iterazione:

collection->select(v : Type | boolean-expression-with-v)

Sempre nell'esempio

```
context Company inv:
self.employee.select(p : Person | p.age > 50)->notEmpty()
```

Indica di prendere tutti gli elementi p di tipo Person dall'insieme employee tali che la proprietà age di p sia maggiore di 50. Si controlla poi che l'insieme risultante non sia vuoto.

Reject

Questo metodo è l'opposto della Select: restituisce tutti gli elementi che non soddisfano l'espressione booleana. Ha la stessa sintassi di Select ed è equivalente ad una Select in cui si è negata l'espressione booleana.

Collect

A differenza della Select e della Reject, che restituiscono un sottocollezione, la collect restituisce una Bag di differenti oggetti; ad esempio se vogliamo sapere tutte le date di nascita di tutti gli impiegati di una compagnia possiamo usare

self.employee->collect(birthDate)

oppure

self.employee->collect(person | person.birthDate)

oppure

self.employee->collect(person : Person | person.birthDate)

Dall'esempio si deduce che usa la stessa sintassi di Select e Reject.

Notare due cose: Il tipo di oggetto restituito è un Bag in quanto è possibile che vi siano oggetti duplicati (le date di nascita, nell'esempio, saranno molto facilmente duplicate); la seconda cosa importante è che la Collect è un'operazione comune e di conseguenza può essere abbreviata con la sintassi del punto:

self.employee.birthdate

è equivalente a

self.employee->collect(birthdate)

forAll

Quando dobbiamo verificare un vincolo per tutti gli elementi di una collezione, possiamo usare l'operazione forAll:

collection->forAll(v : Type | boolean-expression-with-v)

collection->forAll(v | boolean-expression-with-v)

collection->forAll(boolean-expression)

Questa operazione restituisce un bool: ritorna vero se tutti gli elementi soddisfano la boolean-expression, ritorna falso se ce ne almeno uno che non la soddisfa.

Ad esempio

context Company
inv: self.employee->forAll(forename = 'Jack')
inv: self.employee->forAll(p | p.forename = 'Jack')
inv: self.employee->forAll(p : Person | p.forename = 'Jack')

Queste tre restituiscono vero se la proprietà forename di ogni employee è uguale a Jack.

Esiste una variante di questa operazione che permette di usare più variabili d'iterazione: semplicemente vengono elencate le variabili separandole con le virgole. Ad esempio

context Company **inv**:

```
self.employee->forAll( e1, e2 |
                        e1 <> e2 implies e1.forename <> e2.forename)
```

Questo controlla che non vi siano due impiegati che hanno lo stesso nome. È semanticamente uguale a

```
context Company inv:
self.employee->forAll(e1 | self.employee->forAll (e2 |
                                                    e1 <> e2 implies e1.forename <> e2.forename)))
```

Exist

A volte siamo interessati a sapere se esiste un oggetto che soddisfa una certa proprietà all'interno di una collezione, anziché sapere se tutti gli elementi la soddisfano. L'operatore Exist serve proprio a questo. Ha lo stesso funzionamento di forAll, ma è un operatore esistenziale anziché universale.

Iterate

Questa operazione è la più complicata, ma la più potente tra quelle che iterano sugli elementi delle collezioni, tant'è vero che tutte le operazioni precedenti possono essere specificate con questo operatore.

La sua sintassi è:

collection->iterate(elem : Type; acc : Type = <expression> |

expression-with-elem-and-acc)

dove elem è la variabile d'iterazione, mentre acc è l'accumulatore. Ad acc viene inizialmente assegnato il valore expression, poi elem inizia a iterare tra gli elementi della collezione e viene valutato expression-with-elem-and-acc per elem; il risultato della valutazione di ogni iterazione viene assegnato ad acc.

Vediamo come possiamo implementare la collect con la iterate:

collection->iterate(x : T; acc : T2 = Bag{} |

acc->including(x.property))

Ovviamente con including intendiamo l'operazione, descritta più avanti, che permette di aggiungere elementi ad un insieme.

Risolvere le ambiguità in un operazione

Prima di addentrarci su altre operazioni disponibili, vediamo come mai gli operatori precedenti hanno tre sintassi diverse. Il motivo è dato dall'ambiguità. Procediamo con un esempio:

```
context Person inv:
employer->forAll( employee->exists( lastName = name) )
```

Qui vengono introdotte tre variabili implicitamente. La prima è self, la seconda è l'iteratore del forAll, la terza è l'iteratore del exist. Le varie proprietà (employer, employee, lastName e name) devono essere assegnate a queste variabili.E qui sorgono i problemi, infatti employer viene asseganto alla variabile self di tipo Person e come tale deve essere una sua proprietà, employee può essere assegnato sia a self che all'iteratore del forAll; ma employee non appartiene a Person e quindi non ci sono ambiguità. Ma lastName e Name possono essere assegnate a tutte e tre le variabili, ed entrambi appartengono sia a self sia al terzo iteratore (in quanto entrambi di tipo Person). Questo è ambiguo e

non può essere risolto.

Usando un altro tipo di sintassi il problema è risolto:

```
context Person
inv: employer->forAll( employee->exists( p | p.lastName = name) )
```

oppure

```
context Person
inv: employer->forAll( employee->exists( self.lastName = name) )
```

Size

Restituisce il numero di elementi della collezione

Include

Restituisce vero se la collezione contiene l'elemento specificato:

collection -> include(object)

Exclude

Restituisce vero se la collezione NON contiene l'elemento specificato:

collection -> exclude(object)

Count

Restituisce il numero di volte che un oggetto compare all'interno della collezione:

collection -> count(object)

IncludesAll

Restituisce vero se la collezione contiene tutti gli elementi della collezione specificata:

collection -> includesAll(collection2)

ExcludesAll

Restituisce vero se la collezione NON contiene nessun elemento della collezione specificata:

collection -> excludesAll(collection2)

IsEmpty

Restituisce vero se la collezione è vuota

NotEmpty

Restituisce vero se la collezione è piena

Sum

Restituisce la somma di tutti gli elementi. Gli elementi devono supportare l'operazione di somma (interi e reali)

IsUnique

Restituisce vero se la valutazione dell'espressione data risulta diversa per ogni elemento della collezione:

collection -> isUnique(expr)

SortedBy

Restituisce una sequenza in cui gli elementi sono ordinati per l'espressione. Il primo elemento è quell'elemento per cui expr risulta il più piccolo. Expr deve supportare l'operatore <

Any

Restituisce l'elemento della collezione che rende vera l'espressione. Se più di un elemento la soddisfa ne viene restituito uno a caso. Se nessun elemento la soddisfa il risultato è indefinito.

Collection -> any(expr)

Operazioni disponibili solo per gli insiemi

Quelle qui di seguito valgono solo per gli insiemi

Union

Questa operazione restituisce un insieme che è l'unione dell'insieme di partenza e quello specificato

Set -> union(Set2)

Alternativamente restituisce un Bag che è l'unione tra l'insieme di partenza ed il Bag specificato

Set -> union(Bag)

Uguaglianza

Confronta due insiemi e restituisce vero se e solo se tutti gli elementi dell'insieme specificato stanno in quello di partenza.

Set = Set2

Intersection

Questa operazione restituisce un insieme che è l'intersezione dell'insieme di partenza e quello specificato

Set -> intersection(Set2)

Alternativamente restituisce un Bag che è l'intersezione tra l'insieme di partenza ed il Bag specificato

Set -> intersection(Bag)

Differenza

Restituisce un insieme contenente tutti gli elementi che stanno nell'insieme di partenza, ma non in quello specificato

Set – Set2

Including

Aggiunge l'elemento specificato all'insieme di partenza

Set -> including(object)

Excluding

Restituisce l'insieme di partenza meno l'elemento specificato

Set -> excluding(object)

SymmetricDifference

Restituisce un insieme che contiene tutti gli elementi dell'insieme di partenza e dell'insieme specificato, purché non si trovino in entrambi

Set -> symmetricDifference(Set2)

AsSequence

Restituisce una sequenza che contiene tutti gli elementi dell'insieme di partenza con un ordine non specificato.

AsBag

Restituisce un Bag che contiene tutti gli elementi dell'insieme di partenza

Operazioni disponibili solo per i Bag

Quelle qui di seguito valgono solo per i Bag

Union

Questa operazione restituisce un Bag che è l'unione del Bag di partenza e quello specificato

Bag -> union(Bag2)

Può anche restituire un Bag che è l'unione tra il Bag di partenza e l'insieme specificato

Bag -> union(Set)

Ugualianza

Confronta due Bag e restituisce vero se tutti gli elementi del Bag specificato stanno in quello di partenza e si ripetono lo stesso numero di volte.

Bag = Bag2

Intersection

Questa operazione restituisce un Bag che è l'intersezione del Bag di partenza e quello specificato

Bag -> intersection(Bag2)

Alternativamente restituisce un insieme che è l'intersezione tra il Bag di partenza ed l'insieme specificato

Bag -> intersection(Set)

Including

Aggiunge l'elemento specificato al Bag di partenza

Set -> including(object)

Excluding

Restituisce il Bag di partenza meno l'elemento specificato

Set -> excluding(object)

AsSequence

Restituisce una sequenza che contiene tutti gli elementi del Bag di partenza con un ordine non specificato.

AsSet

Restituisce un insieme che contiene tutti gli elementi del Bag di partenza con i duplicati rimossi.

Operazioni disponibili solo per le sequenze

Quelle qui di seguito valgono solo per le sequenze

Uguaglianza

Confronta due sequenze e restituisce vero se e solo se tutti gli elementi della sequenza specificata stanno in quella di partenza e si trovano nello stesso ordine

Sequence = Sequence2

Union

Restituisce una sequenza che contiene tutti gli elementi della sequenza di partenza seguiti da quella specificata

Sequence -> union(Sequence2)

Append

Aggiunge l'oggetto specificato in coda alla sequenza iniziale

Sequence -> append(Sequence2)

Prepend

Aggiunge l'oggetto specificato in testa alla sequenza iniziale

Sequence -> prepend(Sequence2)

SubSequence

Restituisce una sequenza che contiene tutti gli elementi della sequenza di partenza a partire dal numero specificato fino al secondo numero specificato compresi

Sequence -> subSequence(lower,higher)

At

Restituisce l'i-esimo elemento

Sequence -> at(i)

First

Restituisce il primo elemento

Last

Restituisce l'ultimo elemento

Including

Stesso funzionamento di Append

Excluding

Restituisce una sequenza in cui sono state rimosse tutte le occorrenze dell'oggetto specificato.

Sequence -> excluding(object)

asBag

Restituisce un Bag contenente tutti gli elementi della sequenza inclusi i duplicati

asSet

Restituisce un insieme contenente tutti gli elementi della sequenza con i duplicati rimossi

E questo è tutto per l'OCL :)

Approccio BCE

N.d.A: Questo approccio non appartiene strettamente all'Unified Process, ma l'UML lo supporta completamente :)

Una tecnica considerata buona nella programmazione ad oggetti è quella di dividere la visualizzazione, dall'implementazione, dai dati associati. L'approccio BCE (Boundary-Control-Entity – frontiera, controllo, entità) consiste in questo.

UML supporta questo approccio usando gli stereotipi Boundary, Control ed Entity. La tecnica consiste, brevemente, in questo: tutte le classi vengono divisi in tre macro categorie (possono anche essere package):

- Classe Boundary: descrive gli oggetti che rappresentano l'interfaccia tra un attore ed il sistema. Rappresenta una parte dello stato del sistema che è presentata all'utente in forma visiva oppure sonora.
- Classe Control: descrive gli oggetti che percepiscono gli eventi generati dall'utente e controllano l'esecuzione del processo di business. Rappresenta un supporto per le operazioni e le attività di un caso d'uso.
- Classe Entity: descrive gli oggetti che rappresentano la semantica delle entità nel dominio applicativo. Corrisponde ad una struttura dati nel database del sistema.

Le classi Boundary corrispondono dunque alla GUI, le classi Control corrispondono alle funzionalità del sistema, mentre quelle Entity agli oggetti che rappresentano i dati nella memoria del sistema (i dati normalmente risiedono su file di testo, database, file in formato XML, HTML, PDF, proprietario, ecc. e non sono oggetti che il sistema manipola direttamente). Un esempio di classe Boundary può essere la finiestra che si presenta all'utente, uno della classe Control può essere la funzionalità di ricerca, uno della classe Entity l'oggetto che permette l'accesso al database.

Le classi Boundary dialogano con l'utente e con le classi Control (non sanno neanche dell'esistenza delle classi Entity), le classi Entity dialogano con le classi Control e con i dati che gestiscono (non sanno neanche dell'esistenza delle classi Boundary) e le classi Control dialogano con le classi Boundary ed Entity (non sanno niente su come presentare all'utente i risultati od il formato di memorizzazione dei dati).

L'approccio BCE è un “modo di pensare” che riflette una buona pratica di ingegneria del software e come tale dovrebbe essere parte di qualsiasi metodo di sviluppo delle applicazione, indipendentemente dalla sottostante piattaforma d'implementazione. Il principale vantaggio di questo approccio è nel raggruppamento di classi in strati gerarchizzati; ciò migliora la comprensione del modello e ne riduce la complessità. Riduce, inoltre, il rischio di creare *classi mostruose* che hanno il controllo

sull'intera funzionalità del sistema.

Nessun linguaggio di programmazione permette di realizzare appieno questo approccio, in quanto le classi Boundary non hanno nessun senso di esistere senza una classe Control che gestisca i loro eventi; di conseguenza spesso le classi Control e Boundary vengono fuse dai linguaggi di programmazione. D'altronde, si pensi alla chiusura di una finestra: quando si preme sul tasto di chiusura, la finestra si chiude, in quanto ha codificato al suo interno il codice di chiusura (come è giusto che sia, visto che manovra la finestra), ma quel codice appartiene al dominio delle classi Control (agisce sulla finestra e non presenta niente all'utente – è la finestra che si chiude, non il codice).

Generalizzazione, ereditarietà, polimorfismo e loro problemi

N.d.A.: Questo non fa parte dello Unified Process, ma mi sembrava doveroso trattarlo per una migliore comprensione :)

La generalizzazione è un costrutto estremamente potente perché si porta appresso concetti di sostituibilità, ereditarietà e polimorfismo. Prima di procedere nella trattazione di tali potenti, e pericolose, caratteristiche è importante capire la sottile differenza tra generalizzazione ed ereditarietà.

La generalizzazione è una relazione semantica tra classi per cui l'interfaccia della sottoclasse deve comprendere tutte le proprietà (pubbliche e protette) della superclasse.

L'ereditarietà è il meccanismo per mezzo del quale elementi più specifici incorporano la struttura ed il comportamento definito da elementi più generali.

La generalizzazione è un modo di definire classi più specifiche (sottoclassi) che sono dello stesso tipo delle classi più generiche (superclassi). Questo rappresenta una grande opportunità perché permette di mettere una sottoclasse ogni volta che vi è bisogno di una superclasse ed eseguire di conseguenza compiti più specifici. Permette di ridurre il numero di freccie nei diagrammi, in quanto, proprio in virtù della sostituibilità, le relazioni che coinvolgono la superclasse valgono anche per le sottoclassi. Il meccanismo della sostituibilità è generato grazie all'ereditarietà ed al polimorfismo, ma proprio per questo vi possono essere vari problemi, se non bene usato.

Quando parlavamo di come trovare le associazioni, abbiamo detto che se una classe è un tipo di un'altra classe allora abbiamo una generalizzazione. Questo di tipo generalizzazione è costruito con l'ereditarietà d'interfaccia. In altre parole la sottoclasse eredita gli attributi e le operazioni della superclasse ed eventualmente può cambiarne l'implementazione. Ma l'ereditarietà può non essere (e spesso non è) solo ereditarietà d'interfaccia, ma essa si porta appresso implementazioni per permettere il riuso del codice (anzi è usata soprattutto per questo scopo, per l'altro tipo si usa il costrutto delle interfacce, come descritto in un paragrafo successivo). Questa caratteristica, unita al polimorfismo, permette di realizzare classi in grado di estendere funzionalità, variare funzionalità ed adirittura restringere funzionalità della superclasse. Ma questo può portare a grossi problemi.

Un uso corretto dell'ereditarietà d'implementazione è quella per estensione, cioè una sottoclasse ha più proprietà o metodi della superclasse, cioè è un tipo della superclasse. Attenzione che, benché sia chiamata per estensione, normalmente il polimorfismo è usato per introdurre ulteriori vincoli sulle proprietà.

Un uso problematico dell'ereditarietà d'implementazione è quello per restrizione, cioè una sottoclasse elimina metodi e/o proprietà di una superclasse od ancora rimuove dei vincoli dalle proprietà. Questo mina il riuso, in quanto può darsi che una classe chiami un metodo della sottoclasse che è stato in realtà abolito o presupporre delle post condizioni che la sottoclasse non rispetta. Il risultato si può immaginare...

A questo si può aggiungere un uso totalmente improprio dell'ereditarietà d'implementazione: quella cosidetta per convenienza. In questo modo si costruisce una sottoclasse solo per poter utilizzare alcuni metodi già pronti (ad esempio estendere una classe Moto con una classe Macchina), aggiungere i metodi non supportati dalla superclasse e rimuovere quelli incompatibili. Questo causa la totale non sostituibilità, grossi problemi di manutenzione e di comprensione. D'altronde quando una macchina può essere una moto?

Altri problemi possono derivare dalle modifiche delle superclassi. Infatti immaginiamoci che ad una superclasse vengano cambiati proprietà e metodi (fondendone alcuni, dividendone altri, stravolgendone altri ancora); cosa accade alle sottoclassi? Esse ereditano la nuova interfaccia e la nuova implementazione e può darsi che non riesca più a svolgere il compito datogli (ed adirittura la funzionalità impazzisca completamente).

Un ultimo problema dato dalla generalizzazione è dovuto al binding statico che viene fatto; se una classe è sottoclasse di un'altra, essa non può cambiare superclasse nel corso dell'esecuzione. Da questo segue una certa rigidità del codice. Ad esempio supponiamo che esistano una classe Impiegato e due sottoclassi di questa: Manager e Programmatore. Con questo schema un Impiegato che è un Programmatore non potrà mai divenire Manager in quanto sono due classi diverse ed un oggetto non può cambiare classe.

In quest'ultimo caso la soluzione è banale: Programmatore e Manager non sono un tipo di impiegato, ma un tipo di Lavoro; un Impiegato svolge un Lavoro (cioè possiede un oggetto Lavoro); quando l'Impiegato da Programmatore diventa Manager, viene distrutto l'oggetto Programmatore e creato un oggetto Manager; la relazione con l'oggetto Impiegato non cambia di una virgola, in quanto sia Programmatore che Manager sono tipi di Lavoro.

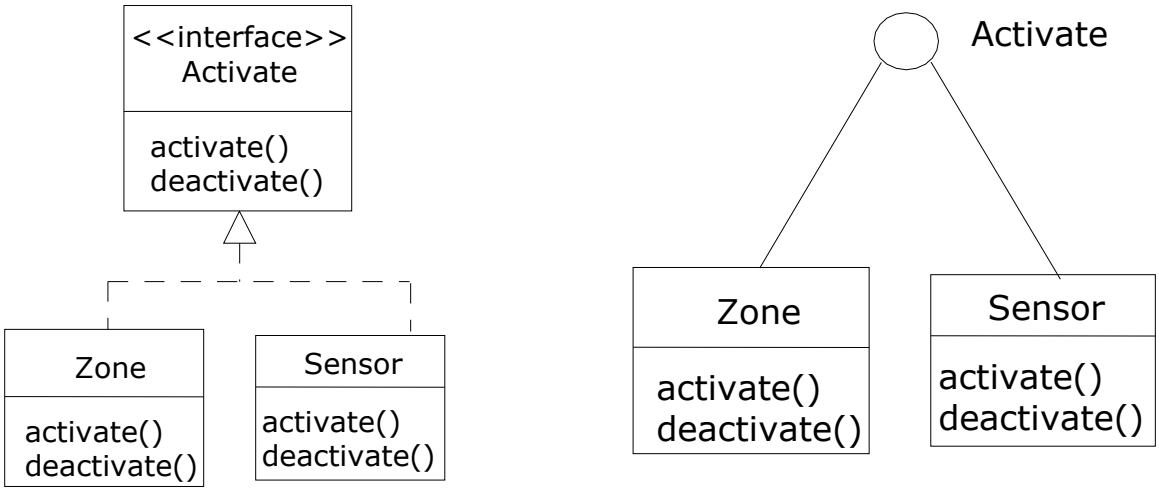
Il problema delle modifiche può essere aggirato con l'aggregazione/composizione: praticamente la classe in realtà non ne estende nessun’altra, ma possiede oggetti/raggruppa oggetti delle classi che avrebbe dovuto estendere, e ridichiara tutte le proprietà ed i metodi di cui ha bisogno, delegando agli oggetti da essa posseduti tutte le funzioni che non è in grado di soddisfare da solo. Il problema è nei diagrammi (le relazioni si moltiplicano) e nella lunghezza e complessità nella realizzazione della nuova classe.

Come si può vedere i problemi rimangono aperti. Dunque prestare attenzione nell'uso della generalizzazione.

Interfacce

Un'interfaccia specifica un insieme di nomi di operazioni. L'idea chiave dietro le interfacce è quella di separare la specifica delle funzionalità dalla sua implementazione. Di conseguenza l'interfaccia definisce un contratto implementato da un classificatore (classe, sottosistema, componente). L'interfaccia deve avere firma completa dei suoi metodi, semantica dell'operazione che deve svolgere (testo o pseudocodice), opzionalmente vincoli, tag, stereotipi. Non deve in nessun caso avere attributi, implementazioni, relazioni navigabili da sé a qualcos'altro.

In UML un'interfaccia viene rappresentata in due modi:



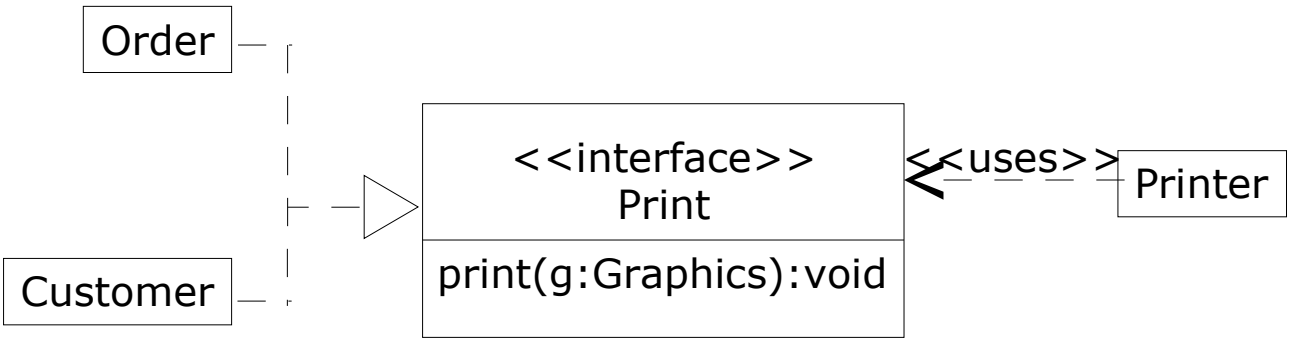
Con il primo sistema (rappresentato a sinistra) si indica esplicitamente quali sono i metodi che le classi devono implementare. Con il secondo sistema (rappresentato a destra) si indica che le due classi implementano un interfaccia di nome Activate, le operazioni da implementare dovranno essere ricavate dai nomi dei metodi delle classi.

Come usare le interfacce?

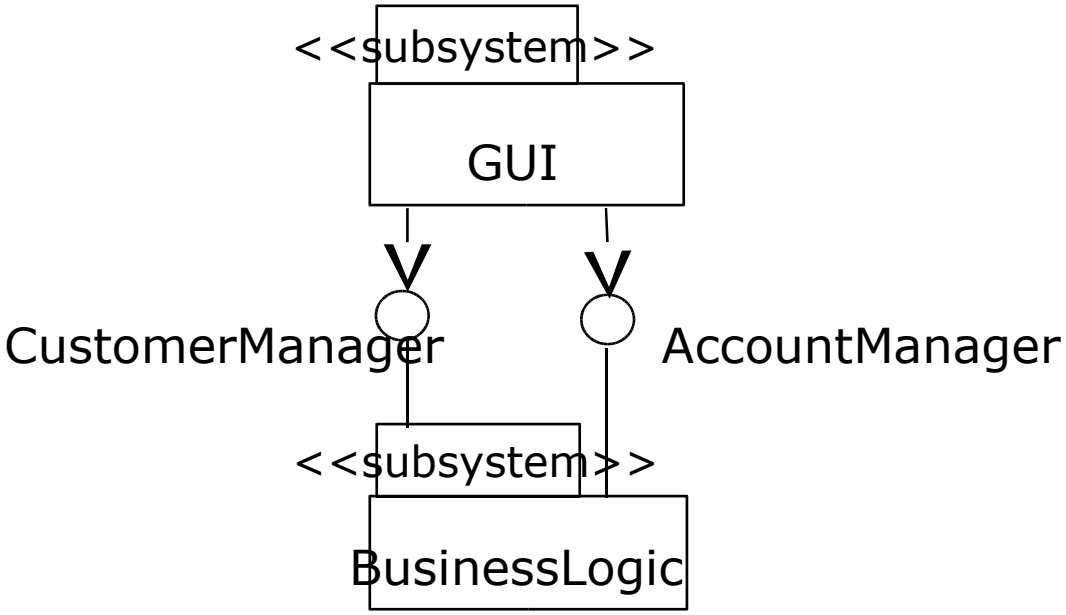
Le interfacce vengono usate per poter permettere ad un classificatore di poter adoperare altre classi, senza sapere come vengono implementati i metodi.

Ad esempio: La classe Printer deve stampare un documento, esso può essere un ordine (classe Order) o una scheda cliente (classe Customer). Anziché dover mettere delle specifiche all'interno di Printer per sapere come stampare le relative classi, si lascia il compito ad esse facendo loro implementare un metodo print, definito dall'interfaccia, che Printer provvederà a chiamare (e Printer non ha bisogno di sapere a quale classe si sta rivolgendo per la stampa: uno potrebbe mettere una classe Invoice che implementa tale interfaccia e Printer sarà in grado di stamparla).

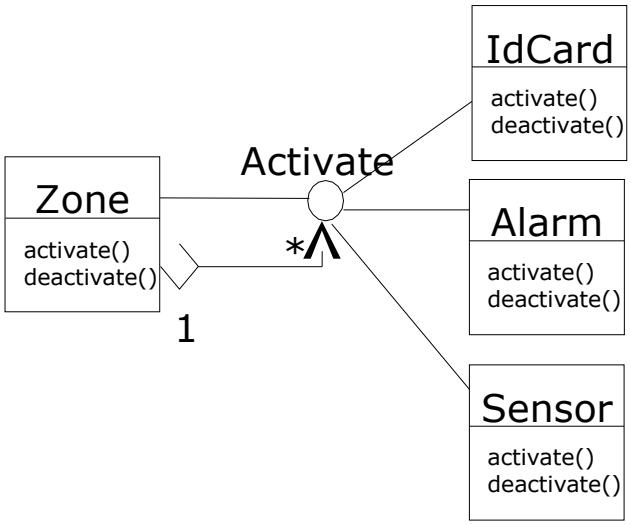
In UML si rappresenterebbe la cosa, così:



Le interfacce vengono inoltre usate per collegare sottosistemi (per sapere cosa sono i sottosistemi, vedi l'apposito paragrafo più sotto).



Un ultimo esempio potrebbe riguardare l'aggregazione di interfacce: Un ufficio è diviso in Zone, che comprende un numero di zone sussidiarie, ogni zona ha un lettore di Card che attiva o disattiva l'ID Card per quella zona. Una Card attivata per un aggregato di zone è valida per tutte le parti di quella zona. In questo modello ogni zona è una collezione che contiene degli oggetti che realizzano l'interfaccia Activate. Questo rende molto facile inserire nuovi device, come un allarme antifumo ed il suo lettore di ID Card.



Come si può notare dalla figura la relazione di aggregazione da Zone ad Activate è unidirezionale. Questo è necessario perché altrimenti altri oggetti possono avere associazioni verso l'intefaccia, ma un'interfaccia stessa non può avere associazioni con nient'altro. Si nota anche che l'aggregazione asimmetrica impedisce di dare se stesso come parte all'oggetto Zone.

Individuare le interfacce

- Discutere le associazioni: vanno ad una classe o devono essere più flessibili? Se devono essere più flessibili, probabilmente hanno bisogno di un'interfaccia.
- Discutere l'invio dei messaggi: vanno ad oggetti di una sola classe o devono essere più flessibili? Se devono essere più flessibili, probabilmente hanno bisogno di un'interfaccia.
- Fattorizzare gruppi di operazioni riusabili. Per esempio molte classi del sistema devono essere in grado di stampare loro stessi su un dispositivo di output, probabilmente ci sarà bisogno di un'interfaccia Print.
- Fattorizzare insiemi di operazioni comuni a più classi.
- Identificare classi che giocano lo stesso ruolo. Il ruolo indica una possibile interfaccia.
- Identificare possibili estensioni. A volte c'è bisogno di progettare un sistema che si espanderà in futuro. La domanda è "Nel futuro, altre classi avranno bisogno di essere aggiunte al sistema?" Se la risposta è sì, provare a definire una o più interfaccie che definiscono il protocollo per aggiungere queste nuove classi.

Sottosistemi

Un sottosistema è un "package" che nasconde la propria fase di design. È utilizzata per separare questioni relative al progetto, rappresentare componenti a grana grossa, racchiudere sistemi legacy.

Permettono, inoltre, di dividere la progettazione del sistema in più parti ed eventualmente affidarli a persone diverse.

Un sottosistema è rappresentato come un package con lo stereotipo Subsystem. tale package può essere diviso in tre parti (nessuna delle quali obbligatoria):

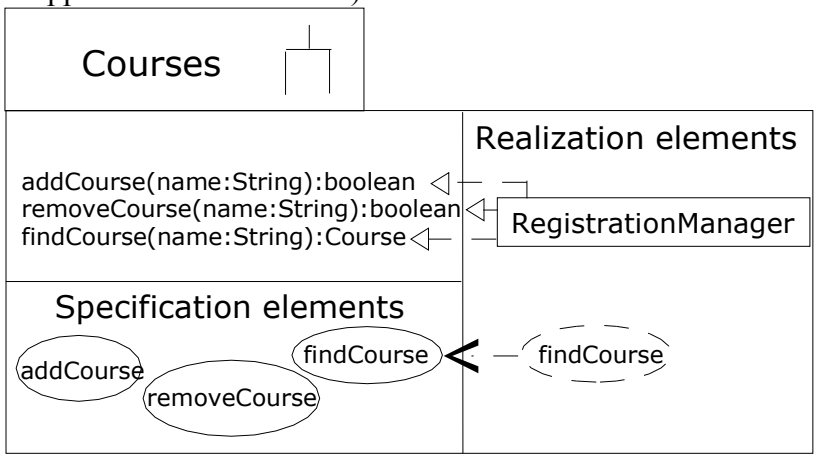
Operazioni (Senza etichetta): Questo comparto contiene le operazioni che il sottosistema rende pubbliche (ovviamente saranno realizzate da qualche classe pubblica).

Elementi di specifica: Questo comparto è per quegli elementi come casi d'uso od interfacce che specificano alcuni aspetti del sistema.

Elementi di realizzazione: Questo comparto è per quegli elementi che realizzano il sistema.

Si possono visualizzare esplicitamente le associazioni tra i vari elementi dei compartimenti.

Esempio (e relativa rappresentazione in UML):

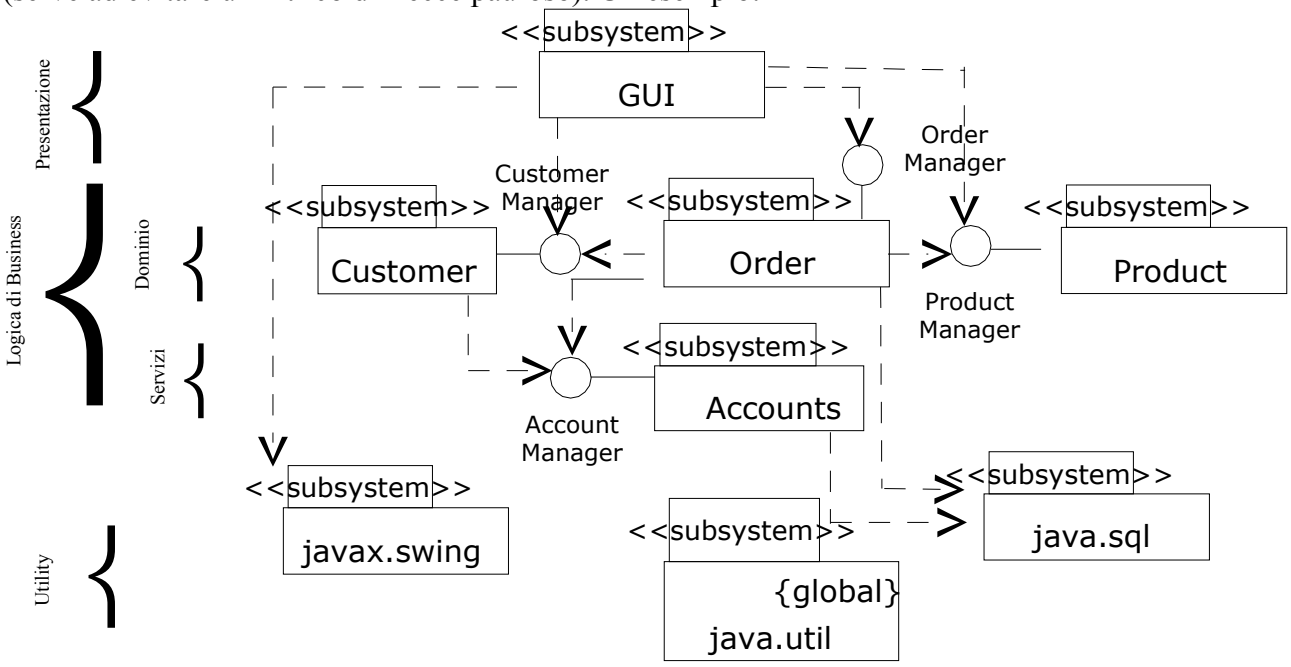


Il simbolo che notiamo vicino al nome del sottosistema Courses è una rappresentazione grafica del-

lo stereotipo <<subsystem>>.

Per evitare di esibire classi pubbliche, ogni sottosistema dovrebbe essere corredato da un'interfaccia.

Gli insiemi di sottosistemi possono essere divisi in strati ad indicare la funzionalità di certo insieme di sottosistemi. I sottosistemi marcati con la parola chiave {global} sono sottosistemi usati da tutti (serve ad evitare un intrico di frecce pauroso). Un esempio:



La costruzione di sottosistemi avviene attraverso l'analisi dei package di analisi, dall'introduzione di sistemi relativi al dominio della soluzione (come un sistema di accesso al database o di comunicazione), l'introduzione di sistemi cosidetti Legacy (cioè in cui non si ha alcun controllo).

Realizzazione dei casi d'uso – progetto

Allo stesso modo dell'attività di analisi, durante la fase di design si riprendono i diagrammi di interazione per entrare nelle specifiche dei casi d'uso. Il focus in questo caso, non è il cosa, ma il come: nei diagrammi d'interazione (Design), entrano in scena tutte le classi di utilità e di contorno e specificano, per ogni caso d'uso, tutti i passi che deve effettuare il sistema per risolverlo. Eventuali sottosistemi possono essere trattati come scatole nere (rimandando al sottosistema i dettagli).

Nei diagrammi di sequenza e collaborazione i sottosistemi vengono visti come oggetti con lo stereotipo <<subsystem>>; ovviamente i metodi che si possono invocare ed i risultati che può dare sono dati dalle operazioni che il sottosistema rende pubbliche.

Diagrammi degli stati

Benché possano essere usati in altre attività per chiarificare alcune parti del progetto, i diagrammi di attività vengono usati principalmente durante l'attività di Design, in quanto è importante ora conoscere l'evolversi della situazione delle classi. Si prenda ad esempio una classe Parser: nell'attività di analisi bastava sapere che esisteva (il suo compito è riconoscere le stringhe), ma nell'attività di design è indispensabile sapere come fa e di conseguenza tutti i suoi mutamenti interni.

L'UML viene incontro alla rappresentazione degli stati di una classe attraverso il diagramma degli stati, che è un diagramma estremamente potente (nel senso che ci si possono specificare molte

cose).

Tale diagramma contiene uno stato iniziale ed uno finale rappresentati rispettivamente con un cerchio pieno e con un cerchio pieno inscritto in uno vuoto. Lo stato iniziale è in realtà uno pseudostato e passa immediatamente al vero stato iniziale della macchina; lo stato finale è invece uno stato vero e proprio. Ogni stato interno è rappresentato da un cerchio vuoto con dentro il nome dello stato.

In UML uno Stato è una condizione o situazione durante la vita di un oggetto, durante la quale esso soddisfa qualche condizione, esegue qualche attività o è in attesa di qualche evento. Lo Stato dipende dai valori degli attributi dell'oggetto, i suoi collegamenti con altri oggetti, o da quello che sta facendo.

Uno stato può avere associato attività ed azioni: le azioni sono istantanee e non interrompibili, mentre le attività hanno una durata e sono interrompibili.

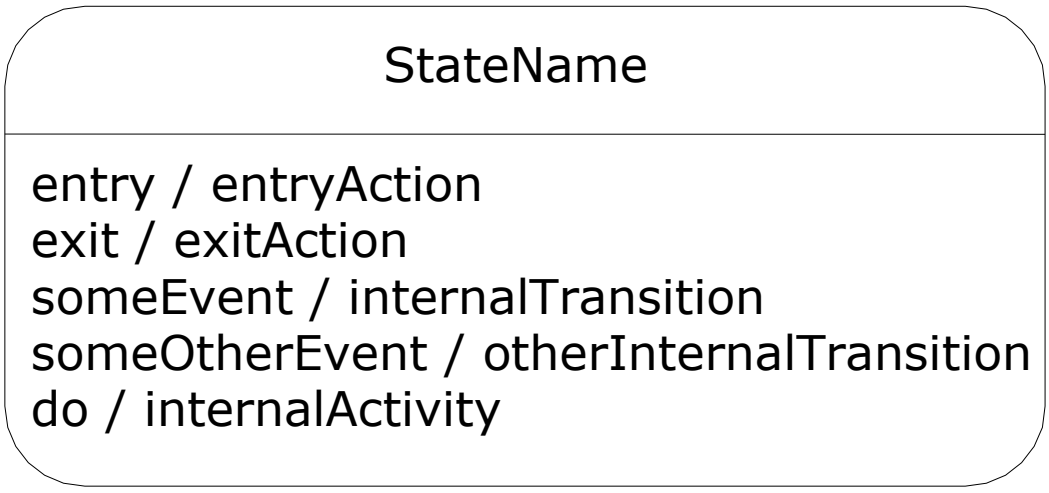
Le azioni sono di tre diversi tipi:

- entry – eseguita ad ogni ingresso nello stato, su qualsiasi transizione
- exit – eseguita ad ogni uscita dallo stato, su qualsiasi transizione
- internal – azione associata a qualche evento, ma che non fa lasciare lo stato (entry e exit non vengono eseguite)

Le attività sono di un tipo solo:

- do – eseguita fino a che si permane nello stato, può terminare prima di lasciare lo stato, indipendentemente. Produce evento di completamento.

Ecco un esempio di stato completo di azioni ed eventi

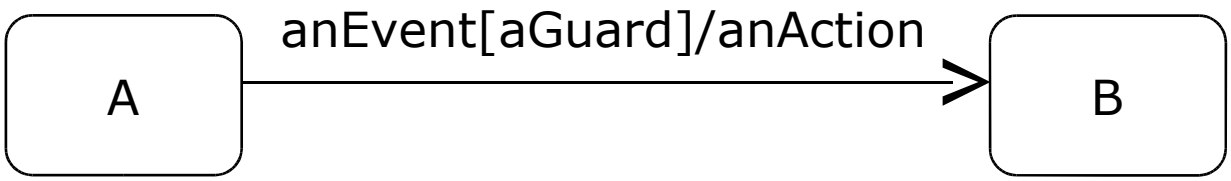


Le transazioni tra uno stato e l'altro sono rappresentate da frecce su cui vi possono essere scritte:

Eventi – occorrenza interna o esterna che fa scattare la transizione.

Guardie – condizione booleana che deve verificarsi prima che la transizione possa accadere.

Azione – ciò che viene eseguito quando la transizione scatta.

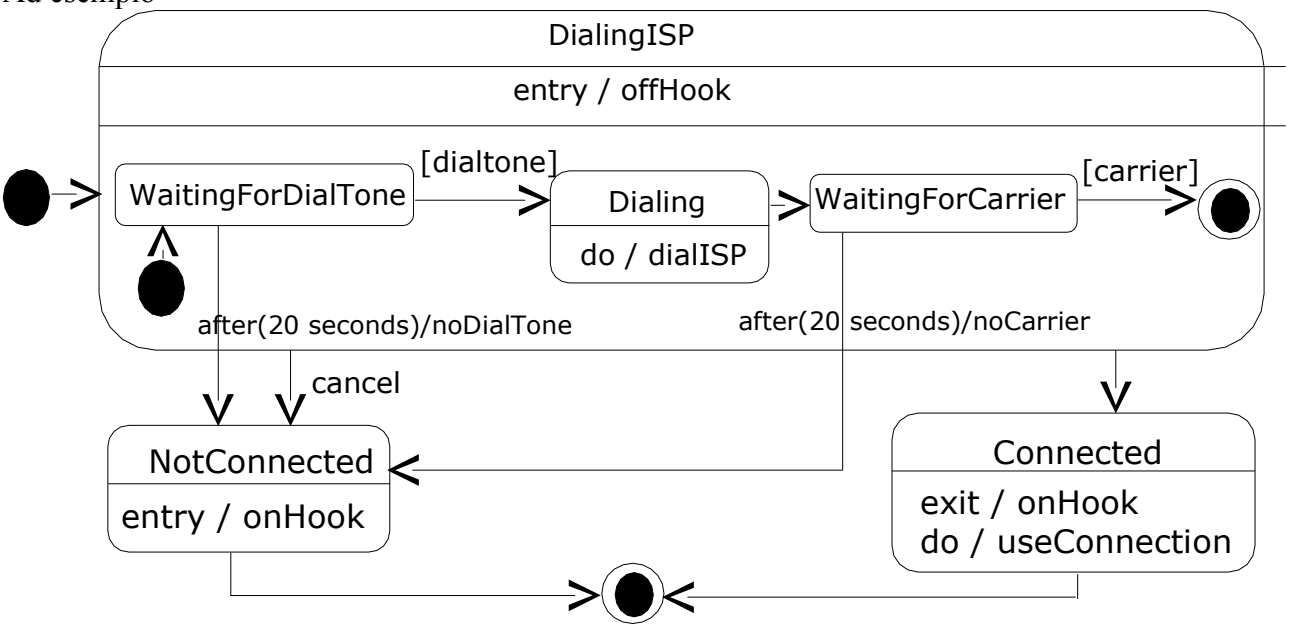


Gli eventi possono essere:

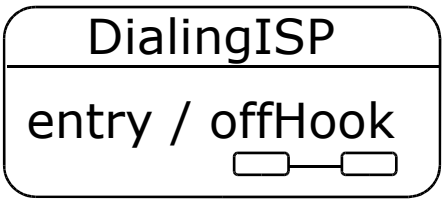
- CallEvent – Richiesta di esecuzione di un'azione (o sequenza di azioni), deve avere nomi di metodi della classe contesto. Possono avere parametri e valore di ritorno.
- SignalEvent – Ricezione di un pacchetto di informazioni inviate asincronamente. Signal è argomento dell'evento.
- ChangeEvent – Usa la parola chiave **when** ed un'espressione booleana. Gli attributi usati devono appartenere alla classe contesto. Avvengono nel passare da *false* a *true* .
- TimeEvent – Usa le parole chiavi **when** od **after** . Gli attributi usati devono appartenere alla classe contesto.
- A volte il tipo di evento da usare cambia a seconda di come si vuole vedere il modello: ad esempio una CallEvent può divenire una ChangeEvent se passiamo dalla vista esterna (osserviamo la temperatura di un paziente) a quella interna (quando la temperatura del paziente cambia, allora...).

Ma il diagramma degli stati non si ferma qui: è possibile rappresentare stati composti, cioè uno stato contiene una o più macchine a stati annidate. Ogni sottostato eredita tutte le transizioni dello stato genitore e l'annidamento è transitivo.

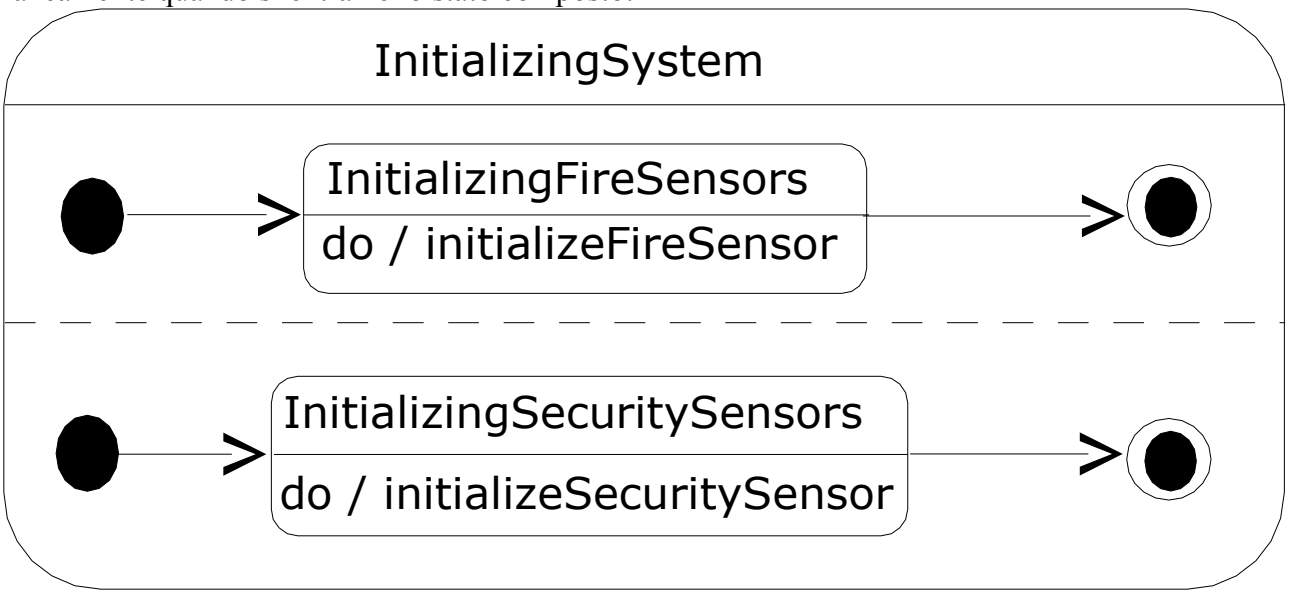
Ad esempio



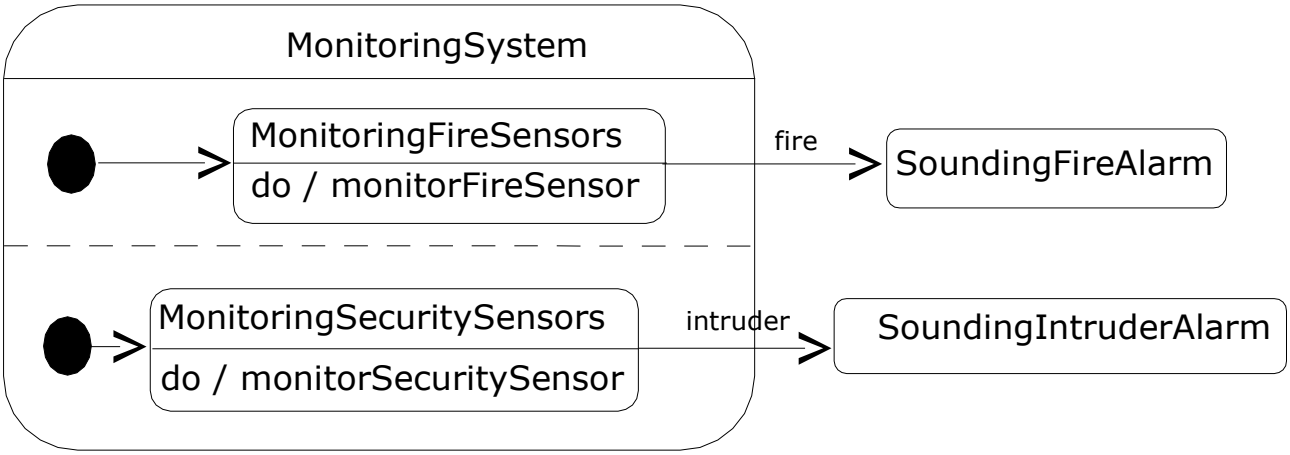
È anche possibile rappresentare gli stati composti senza far vedere i loro stati interni (rimandando ad un altro diagramma la rappresentazione):



Uno stato composto può anche essere concorrente, cioè due macchine a stati si muovono contemporaneamente quando si entra nello stato composto:

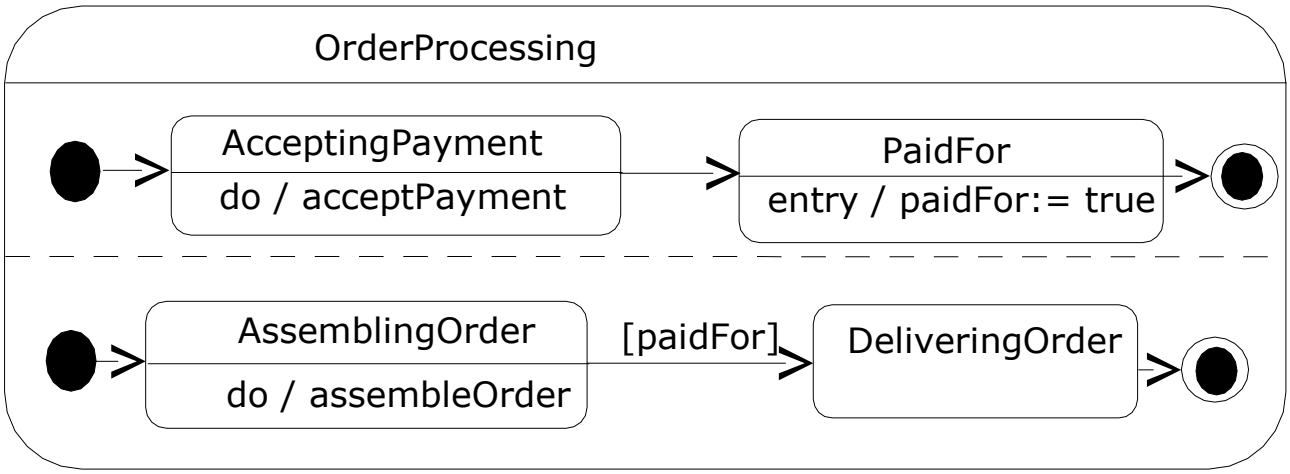


Ovviamente le macchine potrebbero indicare di uscire dallo stato composto in maniera indipendente:

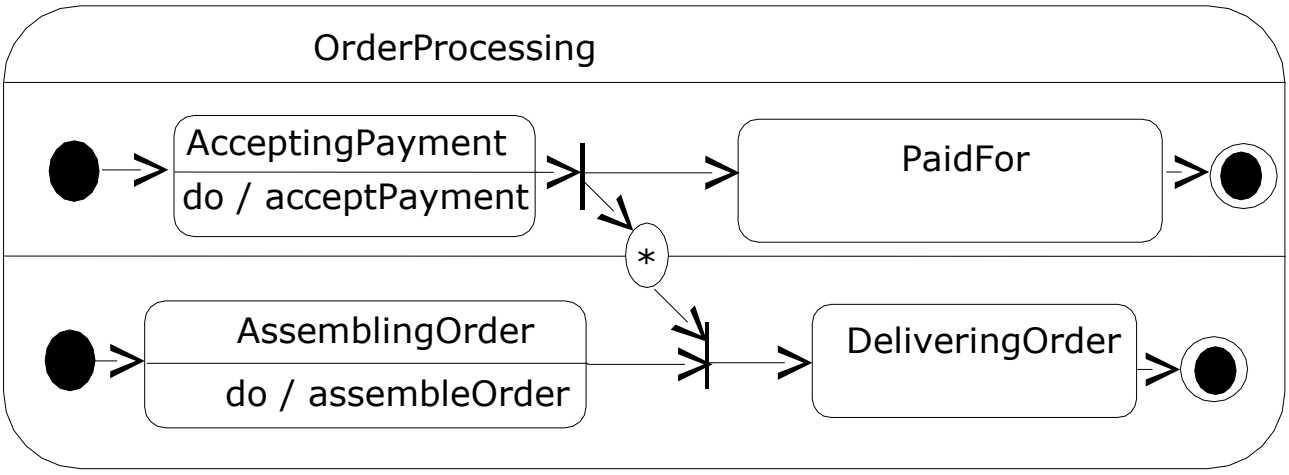


Qualsiasi sia la transazione eseguita, viene conclusa l'attività di entrambe le macchine e lasciato lo stato composto.

Inoltre le macchine interne potrebbero voler comunicare tra loro; ciò è possibile o attraverso gli attributi:



O attraverso i cosi detti stati di sincronizzazione, cioè dei particolari stati dove parte dell'esecuzione resta in attesa che l'altra macchina sia pronta per poi proseguire insieme:



In realtà nell'esempio la prima macchina prosegue comunicando all'altra la possibilità di proseguire.

Un'altra potenzialià del diagramma degli stati è la possibilità di usare gli stati History che permettono di ricordare lo stato in cui si è lasciata una macchina e rientrare nella macchina in quello stato anziché nello stato iniziale. Esistono due tipi di stati History:

- Shallow history – Ci si ricorda solo dello stato al livello indicato.
- Deep history – Ci si ricorda degli stati delle sottomacchine.

I primi stati sono indicati con una lettera H, mentre i secondi con H*. La differenza tra le due riguarda solo se si è in uno stato composto: uno stato di tipo H farà ricominciare tutte le sottomacchine dall'inizio, mentre uno di tipo H* si ricorderà anche dove erano arrivate anche le sottomacchine e le farà ricominciare da quel punto.

Riepilogo

L'attività di Design si occupa di come realizzare il sistema, basandosi sulle analisi effettuate e sull'architettura di base definita, inizia a sviluppare in modo dettagliato le varie operazioni necessarie alla costruzione del sistema.

Riepilogo dei documenti prodotti

Tutta una serie di documenti viene prodotta durante questa attività, tutti volti a capire come realizzare il sistema. Essi sono:

- **Diagramma delle classi**: completa tutte le classi e le associazioni dal punto di vista implementativo. Trasforma le classi di analisi in quelle di design e traduce i modelli non compatibili (come le classi di associazione). Comprende anche le interfacce
- **Diagramma di sequenza e/o collaborazione**: modella il comportamento delle classi per la risoluzione dei casi d'uso, dal punto di vista implementativo (comprese le classi di accesso)
- **Diagramma dei package di design**: modella come sono accoppiate le classi.
- **Diagramma degli stati**: Modella i comportamenti interni delle classi.
- **Sottosistemi**: visualizza i sottosistemi utilizzati, dove sono utilizzati e, per quelli non legacy, vi sono i rimandi alle loro specifiche implementative, con i rispettivi diagrammi.
- **Documentazione aggiuntiva**: Ad esempio i documenti OCL, l'analisi delle scelte implementative, ecc.

Implementation (Implementazione)

L'attività di implementazione segue quella di Design e trasforma il modello di Design in un sistema eseguibile. Non esiste un vero e proprio modello d'implementazione e spesso esso è lasciato alla capacità del programmatore. Tuttavia un modello di implementazione trasforma elementi del modello di design in componenti e organizza le componenti secondo meccanismi di strutturazione e modularizzazione.

Durante questa attività il sistema prende vita: da semplici diagrammi si passa a programmi eseguibili, librerie, database, ecc.

Nonostante che il modello di implementazione esatto sia lasciato al programmatore, ci si aspetta che esso comprenda:

- Piano d'integrazione
- Componenti
- Diagramma di Deployment

Piano d'integrazione

Il piano d'integrazione consiste in una sequenza di build (versione eseguibile del sistema), richiesta per un’iterazione, comprendente funzionalità da costruire per ogni build e le parti del modello di implementazione influenzate dal build.

Un build dovrebbe aggiungere funzionalità a quello precedente implementando casi d’uso e/o scenari completi, non dovrebbe includere troppe componenti nuove o raffinate, dovrebbe essere basato sul build precedente e partire dagli strati bassi ed espandersi verso strati generali e specifici dell'applicazione.

Componenti

Un componente è una parte fisica e rimpiazzabile di un sistema, che racchiude l’implementazione, è conforme a, e fornisce la realizzazione di un insieme di interfacce.

Esempi: File sorgente, Sottosistemi di implementazione, Controlli ActiveX, JavaBeans, Enterprise JavaBeans, Java servlets, Java Server Page, ecc.

Diagramma dei componenti

Un componente è rappresentato graficamente come un rettangolo sul cui lato sinistro vi sono due rettangoli più piccoli. Il nome identificatore del componente è scritto all'interno del rettangolo più grande. Normalmente vi è uno stereotipo ad indicare il tipo di componente:

- <<executable>>: programma che può eseguire su un nodo
- <<file>>: contiene codice sorgente o dati
- <<library>>: statica o dinamica
- <<table>>: tabella di una base di dati
- <<document>>: documento

Il diagramma mostra i componenti e le loro relazioni, che possono essere di dipendenza o composizione. Nel primo caso indica la necessità di un servizio, nel secondo che quel componente è in realtà formato da diversi componenti. Le relazioni solitamente non hanno nome. Benché non sia necessario, normalmente i componenti comunicano attraverso interfacce, in modo tale che possano essere sostituite senza cambiare altri componenti.

Ad esempio:

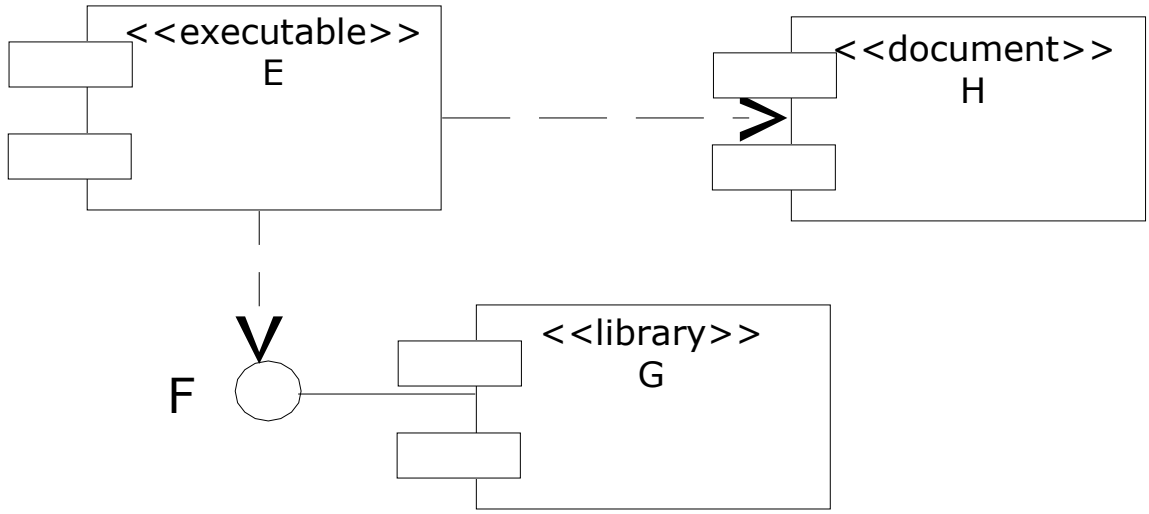


Diagramma di Deployment

Il diagramma di Deployment mostra l'hardware fisico su cui il sistema software verrà eseguito e come questo è distribuito sull'hardware.

Ci sono due forme di diagramma di Deployment:

Forma di descrittore – Contiene nodi e relazioni tra nodi e componenti. Un nodo rappresenta un tipo d'hardware (come un PC), mentre un componente rappresenta un tipo di software (come Word)

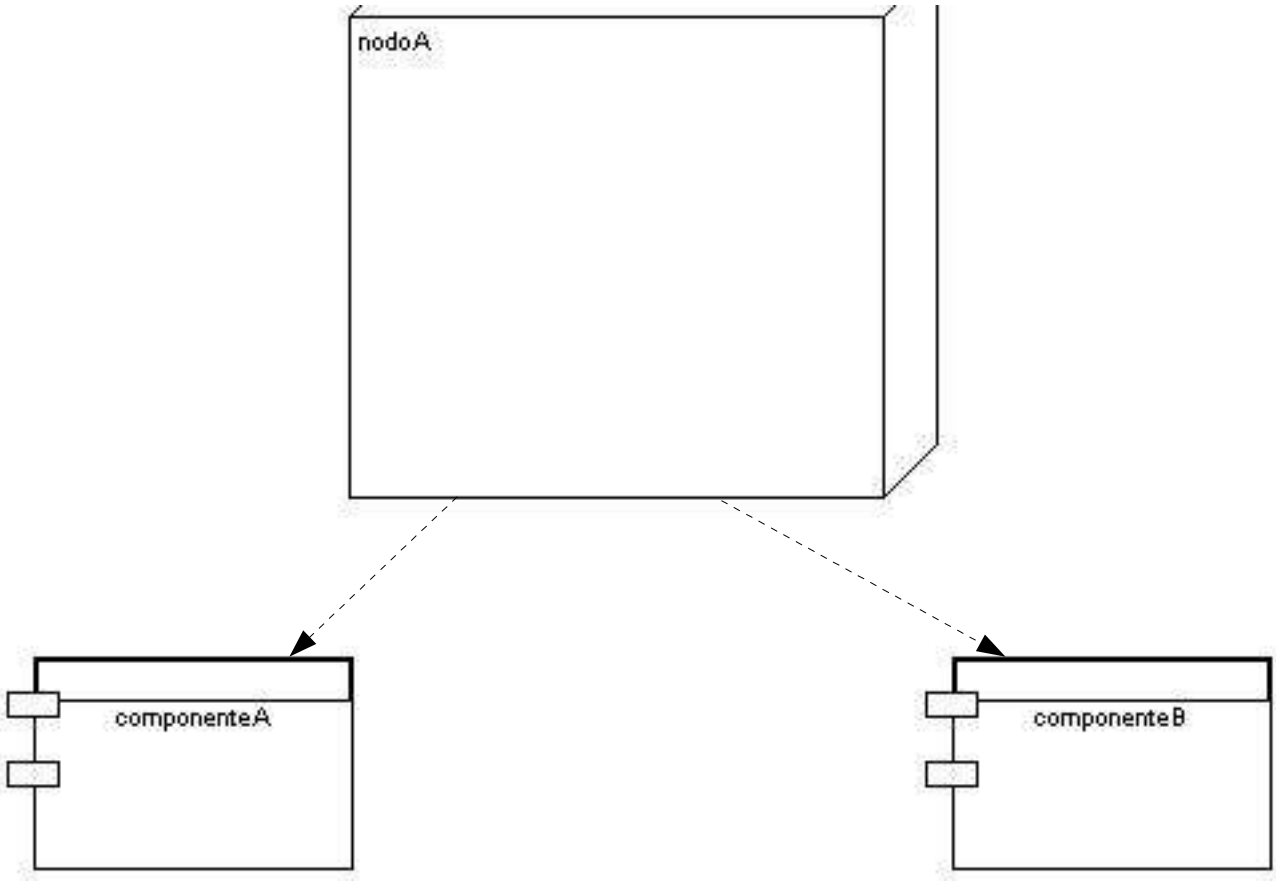
Forma d'istanza – contiene istanze di nodi e relazioni tra istanze di nodi ed istanze di componenti. Un'istanza di nodo rappresenta uno specifico identificabile pezzo hardware (come il computer di Andrea), mentre un'istanza di componente rappresenta una specifica istanza di un tipo di software (come una particolare copia di Word usata per scrivere). Se non si conoscono i dettagli di una specifica istanza , si può usare un'istanza anonima.

Una prima versione può essere stata creata durante l'attività di design (visto che in quella si fanno le scelte implementative), ma in genere il vero lavoro viene fatto durante questa attività. Per costruire il diagramma di deployment, normalmente, si inizia dalla forma di descrittore, che verrà via via raffinata man mano che si conoscono i dettagli sul sito di installazione (arrivando alla forma d'istanza).

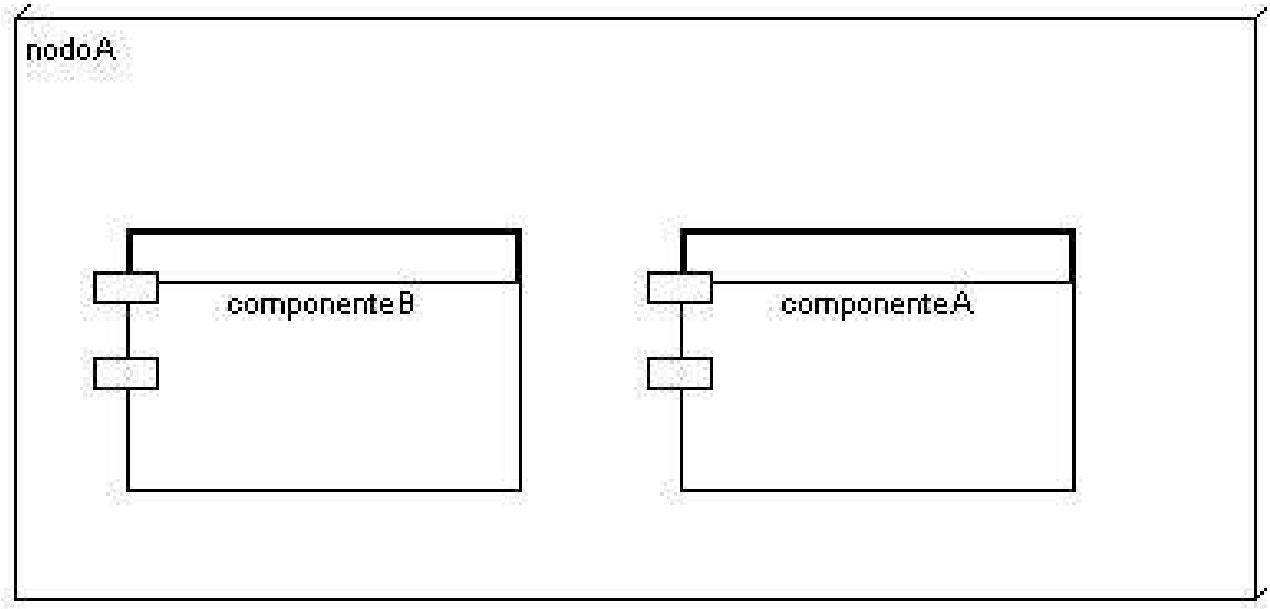
I nodi vengono rappresentati da cubi contenenti un nome ed eventualmente uno stereotipo che ne indichi il tipo (<<printer>>, <<SQL Server>>, ecc.). Gli stereotipi qui usati non sono standard e le loro definizioni devono essere specificati chiaramente (nel glossario). Le relazioni possono indicare connessioni di rete o remote ed in questo caso è obbligatorio il nome che indichi il tipo di protocollo impiegato.

I nodi sono locazioni fisiche, in cui vengono inseriti i vari componenti (ed è lì che vengono eseguiti). Esistono due modi per rappresentare il contenimento dei componenti in un nodo: uno è usare una relazione di dipendenza da un nodo ai componenti che lì risiedono; la seconda è di mettere i componenti all'interno del nodo.

Ad esempio



Oppure



Il Programma (o sistema)

C'è poco da dire ed è pure ovvio, ma è di estrema importanza: è durante questa attività che i programmatori si mettono all'opera, si decidono gli algoritmi, e si costruisce il programma (od il sistema) vero e proprio.

Riepilogo

Durante questa attività viene generato il sistema eseguibile, si stabilisce dove devono andare le varie parti e cosa e come implementare materialmente (le righe di codice).

Riepilogo dei documenti prodotti

Questa attività produce i documenti che specificano la parte implementativa di tutto il progetto: essi comprendono:

- Piano d'integrazione
- Diagramma dei componenti
- Diagramma di Deployment
- Il programma
- Qualsiasi altro documento che il programmatore ritiene importante (documentazione tecnica, scelte implementative, listati di codice commentato, documentazione interna, ecc.)
- Manuale dell'utente e documentazione aggiuntiva necessaria per il cliente.

Test

L'attività di test è una parte molto importante durante lo Unified Process, infatti è compresa in quasi tutte le fasi. Lo Scopo è quello di pianificare i test per ogni iterazione. In particolare bisogna realiz-

zare:

- Test di integrazione per ogni build
- Test di sistema alla fine dell'iterazione

Bisogna dunque progettare e implementare i test, decidere i casi di test che specificano cosa provare, le procedure che specificano come eseguire i test, i componenti eseguibili di test per l'automazione, eseguire i test e gestirne i risultati.

La pianificazione dei test avviene già durante la fase di Inception, quando si definiscono ambiti del sistema. Durante la fase di Elaboration si fanno i test sull'architettura di base. Durante la fase di Construction, per ogni step di implementazione, si usano principalmente test d'integrazione e di sistema per ogni build. Durante la fase di Transition si identificano i difetti da primo uso e si utilizzano i regression test.

Prima di perderci definitivamente, cerchiamo di mettere ordine.

Ogni tipo di test viene pianificato, cioè viene stabilita la strategia generale e le risorse necessarie; la strategia generale indica quanti test devono essere automatici e quanti no, cosa provare ed in che modo. Dalla strategia generale discende il modello del test: cioè non il cosa ma il come provare i singoli aspetti del progetto (componenti, GUI, Manuale Utente, ecc.); il modello descrive per linee generali i casi di test, le procedure ed i componenti. È importante in questa fase (ad ogni iterazione) far evolvere il modello rimuovendo i casi obsoleti, raffinandone altri ed aggiungendone nuovi.

Una volta stabilito il modello si passa alla fase di progettazione dei test, cioè si entra nello specifico per definire i casi di test per ogni build, strutturare le procedure e definire i test d'integrazione (interazione fra i componenti) e di sistema (provano il sistema sotto diverse configurazioni, carico, numero di attori, grandezza del database, ecc.).

Un caso di test specifica un modo per provare il sistema, cioè cosa provare (insieme di requisiti), con quali input e con quali risultati attesi, sotto quali condizioni provare. Il test deve essere possibile e non troppo costoso. I casi di test possono verificare casi d'uso o scenari (comportamento osservabile, black box) e realizzazioni di casi d'uso (interazioni fra componenti, white box). Per ogni caso di test bisogna stabilire i suoi scenari (cioè le variazioni di input ed output). Una cosa semplice è usare una matrice: sulle righe si mettono i casi di test e sulle colonne si mettono i valori di input ed output aspettati per ogni scenario.

Una procedura di test specifica come eseguire i casi di test o loro parti, se devono essere svolti automaticamente o manualmente. Ogni procedura dovrebbe testare un solo sottosistema.

Dalle procedure discendono le implementazioni dei test, cioè la definizione dei componenti da realizzare o mediante programmazione specifica o tramite registrazioni delle azioni.

I componenti di test non sono altro che le procedure di test automatizzate, solitamente scritte con un linguaggio di programmazione, o registrate mediante altri strumenti automatici.

Una volta che i test sono stati preparati, sono state assegnate le risorse e i tool automatici sono funzionanti è giunto il momento di eseguirli: seguendo le linee guida del modello e delle procedure, si provano le varie funzionalità del sistema.

I singoli componenti vengono testati secondo i metodi e le procedure stabilite; dato che i componenti devono lavorare tra di loro si può cercare di vedere se a fronte di certi input danno i dati corretti; per far questo si costruiscono classi fittizie che danno al componente l'input e prendono dal componente l'output.

Quando i vari componenti sono pronti si eseguono i test d'integrazione, cioè si vede se i componenti lavorano davvero tra di loro (se i componenti sono corretti, ma nell'insieme non funziona, significa che qualche componente non dà i risultati che si aspetta un altro componente).

Quando una build è pronta e tutti i test (compresi quelli d'integrazione) sono passati, è giunto il momento dei test di sistema che prevedono test di installazione presso il cliente, test di configurazione (ad esempio configurazioni di rete), test negativi (che mirano a fare andare in errore il sistema, per vederne le debolezze), test di stress (che fanno lavorare il sistema con risorse insufficienti).

I vari risultati vengono poi confrontati, si cercano gli errori, si riportano gli errori al team di sviluppo che dovrà correggerli (e questo può andare ad influire su tutte le attività).

Normalmente a fine di ogni ciclo si tirano le somme e si cercano di vedere quanti e quali sono stati gli errori più comuni in modo da evitare di ripeterli successivamente.

Nota molto importante (anche se già detta): l'attività di test deve essere svolta in parallelo alle altre attività; tuttavia tutte le attività devono dipendere da essa: se, ad esempio, l'architettura non passa il test, allora è inutile proseguire con le altre attività che dipendono da essa, ma bisognerà prima correggere e ritestare l'architettura.

Difetti dello Unified Process

Abbiamo visto lo Unified Process e come esso lavora, osservandone i vantaggi. Ma lo Unified Process presenta anche dei difetti.

Un paio di difetti che si possono notare sono dovuti alle attività: esso presuppone che i requisiti siano espressi in linguaggio naturale ed è compito del progettista tradurli in casi d'uso; questo non è sempre vero: esistono tutta una serie di problemi che hanno una descrizione dei requisiti estremamente formale ed è difficile passare da essi a casi d'uso. Inoltre lo Unified Process di standard prevede che lo stesso team si occupi di analisi, progetto ed implementazione; diversificare il team può portare a rallentamenti.

Oltre a ciò alcuni dei grandi vantaggi dello Unified Process sono anche suoi grandi svantaggi:

- iterativo ed incrementale: questo fa sì che il prodotto completo potrebbe richiedere lunghi tempi per essere pronto od adirittura, se non si fa attenzione, non finire mai (tuttavia i singoli incrementi vengono pianificati). Certo, questo sistema permette di costruire applicazioni “su misura” e di rilasciare il prodotto con minori difetti (in quanto le funzionalità vengono integrate man mano).
- centrato sull'architettura: benché questo permetta di creare un sistema che lavori in maniera egregia sull'architettura, se si ci accorge che l'architettura ideata non era corretta (ad esempio si è pensato ad un'architettura client-server mentre si vede più tardi che era meglio una distribuita), allora si è costretti a gettare via completamente tutto il lavoro di progettazione ed implementazione ed in buona parte anche quello di analisi.

Per ovviare a questi difetti è necessaria una forte dimestichezza e preparazione nella costruzione di software con questo approccio e preparare buone procedure di test che coinvolgono anche l'architettura, i manuali ed i documenti da produrre, ecc.

L'ultimo “difetto”, se così si può dire, è che lo Unified Process si appoggia ad UML, questo fa si che i sistemi grandi sono estremamente difficili da realizzare e mantenere ed i diagrammi UML non aiutano di certo alla comprensione (vengono ad occupare pagine e pagine con riferimenti incrociati veramente terrificanti).

APPENDICI

Refactoring

Il Refactoring è un concetto “nuovo” con il quale si prende un codice e vi si apportano particolari modifiche. Esso cerca di:

- Individuare le parti di codice comuni a diversi metodi o classi
- Manipolare il codice per minimizzare duplicazioni e massimizzare riuso
- Mantenere la semantica delle computazioni
- Defininire le operazioni mediante precondizioni e postcondizioni
- Lasciare la possibilità di comporre refactoring complessi partendo da quelli semplici

I Refactoring semplici si basano su:

- Creare entità di programma
- Cancellare entità di programma
- Cambiare entità di programma
- Muovere variabili membro

Creare entità di programma

Creare nuove entità di programma significa:

- Creare una classe vuota
- Creare una nuova variabile membro
- Creare una funzione membro

Cancellare entità di programma

Cancellare entità di programma siginifca:

- Cancellare classi non referenziate
- Cancellare variabili non referenziate
- Cancellare insiemi di funzioni membro

Cambiare entità di programma

Cambiare entità di programma significa:

- Cambiare il nome di una classe
- Cambiare il nome di una variabile
- Cambiare il nome di una funzione membro
- Cambiare il tipo di un insieme di variabili o funzioni
- Cambiare il metodo di accesso
- Aggiungere argomenti alle funzioni
- Togliere argomenti alle funzioni
- Riordinare gli argomenti delle funzioni

- Aggiungere il corpo di una funzione
- Cancellare il corpo di una funzione
- Convertire un riferimento ad una variabile in una chiamata di funzione
- Sostituire una sequenza di istruzioni con una chiamata di funzione
- Espandere una funzione “in linea” in una chiamata di funzione
- Cambiare la superclasse di una classe

Muovere variabili membro

Muovere variabili membro significa:

- Spostare una variabile membro da una sottoclasse in una superclasse
- Spostare una variabile membro da una superclasse in una sottoclasse

Composizioni primitive

Combinando le “tecniche” sopra riportate si possono creare Refactoring molto complessi. Alcune composizioni sono:

- Astrarre un accesso ad una variabile membro
- Convertire un segmento di codice in una funzione
- Spostare una classe da un package ad un altro

Proprietà da non violare

- Superclassi uniche
- Nomi di classe distinti
- Nomi dei membri distinti
- Le variabili membro ereditate (Inherited) non devono essere ridefinite
- La ridefinizione delle funzioni membre devono essere compatibili con la firma originaria
- Gli assegnamenti devono essere Type-Safe (cioè non ci possono essere assegnamenti di tipo errati)
- Equivalenza della semantica delle operazioni e dei riferimenti

Esempio di refactoring complesso

Prendiamo queste quattro classi:

```
class Client1 {
    SpecServer1 serv;
    SpecServer1 getServer1() { ... }
    void exploitService() {
         serv = getServer1();
```

```
        Object result = serv.service1(this);
        // code using result
    }
}

class Client2 {
    SpecServer2 serv;
    SpecServer2 getServer2() { ... }
    void exploitService() {
          serv = getServer2();
          Object result = serv.service2(this);
          // code using result
    }
}

class SpecServer1 {
   void service1(Object requester) { ... }
}

class SpecServer2 {
   void service2(Object requester) { ... }
}

E trasformiamole:
abstract class AbstractClient {
    Server serv;
    abstract Server getServer();
    void exploitService() {
        serv = getServer();
        Object result = serv.service(this);
        specificAction(result);
    }
    abstract void specificAction(Object arg);
}
```

```
class Client1 extends AbstractClient {
    Server getServer() {
       //code from getServer1();
    }
    void specificAction(Object arg) {
      // previous code using result
    }
}
class Client2 extends AbstractClient {
    Server getServer() {
       // code from getServer2();
    }
    void specificAction(Object arg) {
      // previous code using result
    }
}

interface Server {
    void service(Object requester());
}

class SpecServer1 implements Server {
   void service(Object requester) {
      service1(requester);
   }
  void service1(Object requester) { ... }
}

class SpecServer2 implements Server {
   void service(Object requester) {
      service2(requester);
   }
  void service2(Object requester) { ... }
}
```

Abbiamo applicato i seguenti refactoring:

- Trasformato del codice in un metodo (codice estratto specificAction(object arg))
- Spostato un metodo (exploitService() spostato alla superclasse)
- rinominato un metodo (getServer1() e getServer2() rinominati in getServer())
- Rinominate variabili d'istanza e spostate (serv1 e serv2 spostati in AbstractClient come serv)

Specifica degli effetti del refactoring

Dato che le regole di riscrittura definiscono gli effetti definiti direttamente sul codice, c'è bisogno di mantenere coerente il modello di design con le trasformazioni del codice. Per far ciò bisogna andare a ritroso su trasformare tutti i diagrammi UML.

Ecco un piccolo schema che indica per alcuni refactoring alcuni diagrammi che vanno ritoccati

Refactoring	Affected Diagrams	Creation/Deletion
Add Class	Class	State
Remove Class	Class, Sequence	State
Rename Class	Class Sequence	
Remove Method	Class, Sequence, State	
Add Parameter to Method	Class	
Extract Code As Method	Class, Sequence, State	
Push Up/Down Method	Class, Sequence, State	

Ecco come nell'esempio sopra cambiano i diagrammi (pagina successiva).

Diagramma delle classi:

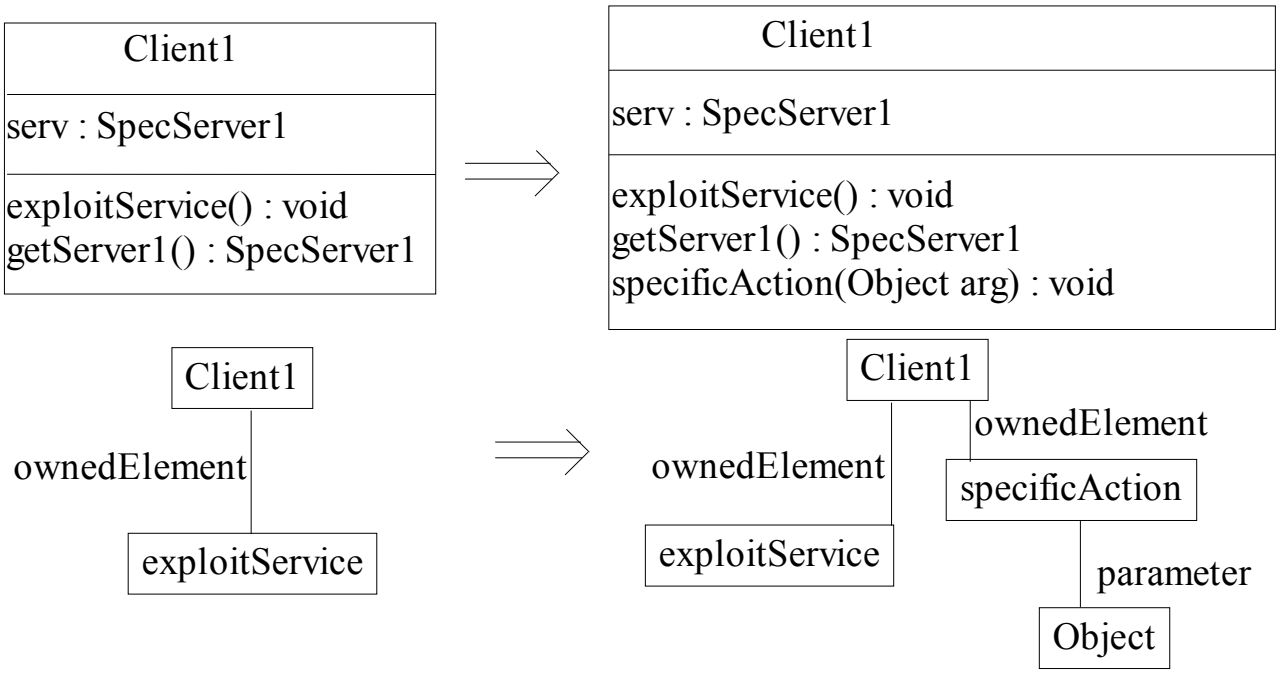


Diagramma di sequenza:

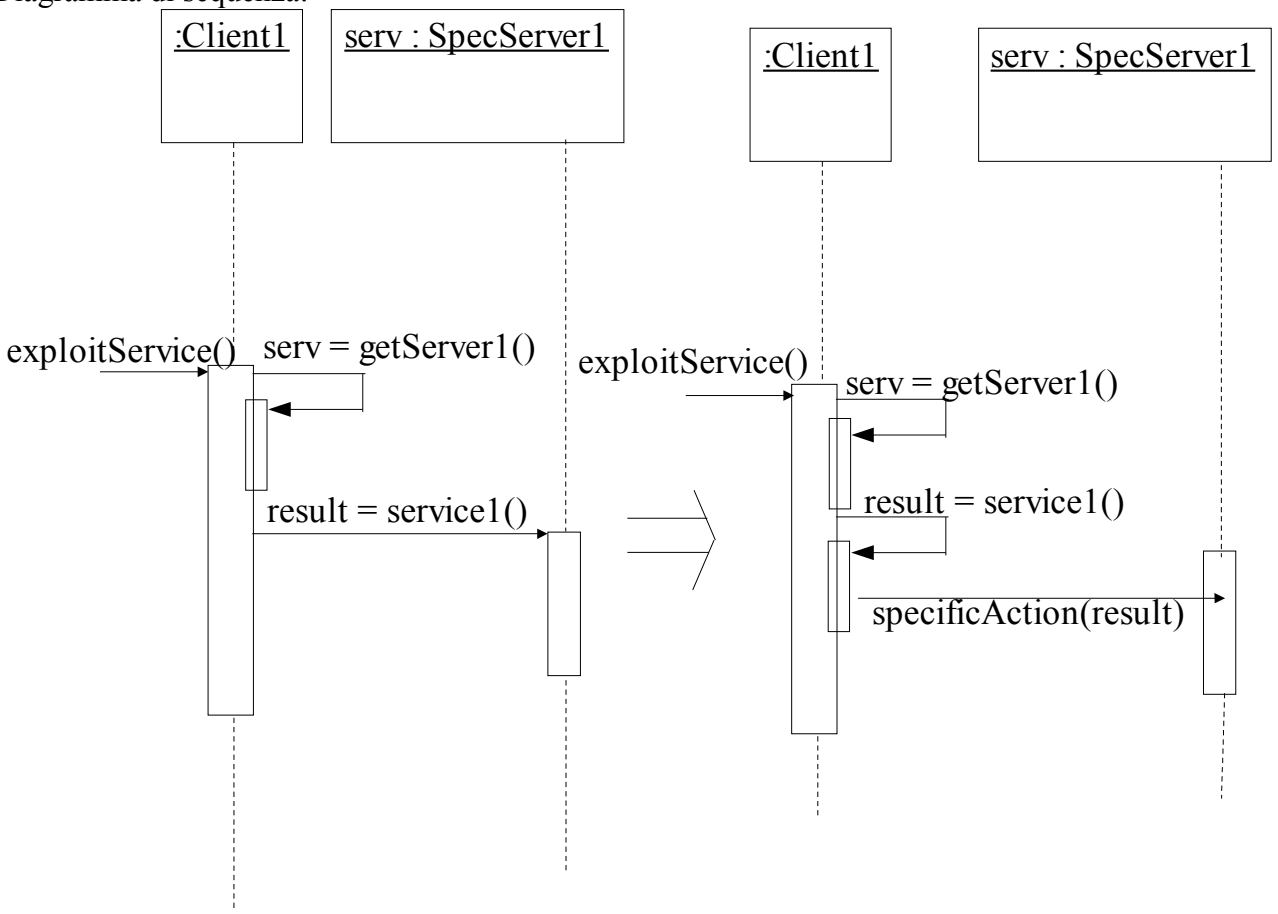
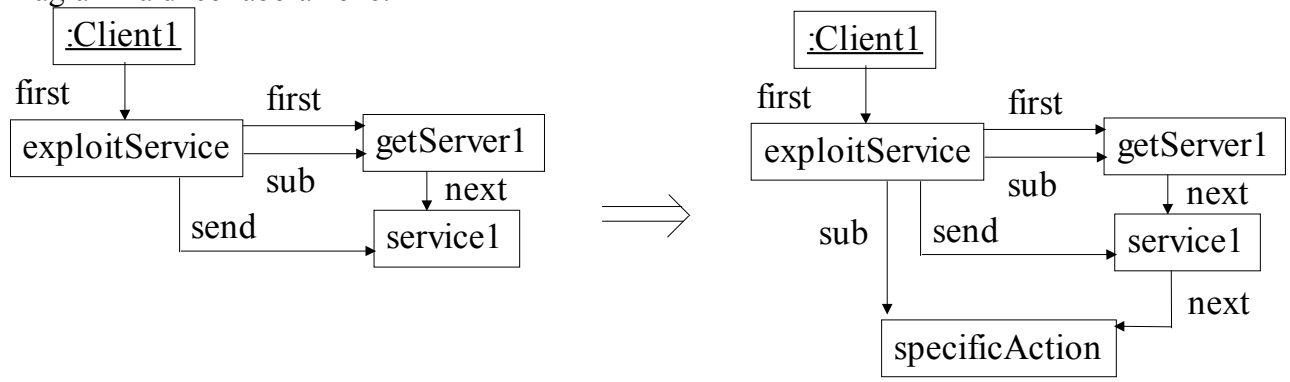


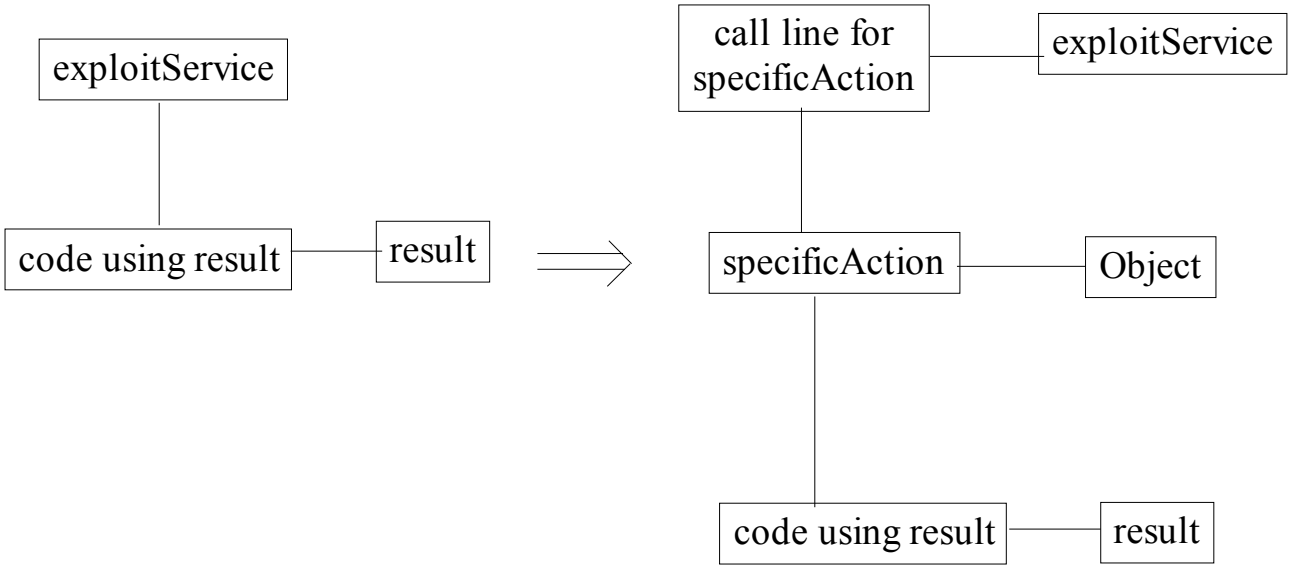
Diagramma di collaborazione:



Per concludere ecco alcune note sull'estrazione di codice come metodo:

Il codice estratto deve costituire un blocco, deve fare riferimento a campi della classe o a variabili dichiarate localmente al blocco. Nel caso in cui faccia uso di variabili esterne al blocco, queste non devono essere modificate nel codice, e diventano parametri passati al metodo.

Brevemente con lo schema:



Bibliografia

Leszek A. Maciaszeck – Sviluppo di Sistemi Informativi con UML (Analisi dei requisiti e progetto di sistema) – Assison-Wesley

Roger S. Pressman – Principi di Ingegneria del Software (Terza Edizione) – Mc Graw Hill

Jim Arlow, Ila Neustadt – UML and Unified Process (Practical Object-Oriented Analisys and Design) – Booch Jacobson Rumbaugh (Edizione italiana a cura della Mc Graw Hill)

Lucidi del corso di Ingegneria del Software 1 tenuto dal professor Bottoni

Vari siti Internet.