Versione 1.3 – 12/06/2020

Ingegneria del Software T – Ingegneria Informatica

Corso tenuto da Marco Patella

APPUNTI DI TEORIA

PER L'ESAME SCRITTO

Appunti di Andrei Daniel Ivan, andreidaniel.ivan@studio.unibo.it

per l'anno accademico 2019/2020

(nel caso troviate errori grammaticali, errori concettuali, cose poco chiare o aggiornamento delle slide, mandatemi una mail e aggiornerò il file)

Indice

Prefazione

Il seguente documento, scritto per facilitare lo studio per l'esame di teoria di Ingegneria del Software T, non ha lo scopo di sostituire le slide e gli strumenti di studio forniti dal docente; chiunque lo usi per studiare si assume quindi la responsabilità di non star studiando dal materiale di studio fornito del professore ma da un documento scritto sulla base di esso, che può quindi presentare qualche errore. Durante la scrittura di questo documento ho in larghissima parte semplicemente trascritto il contenuto delle slide adattandolo in forma testuale, in modo tale che la lettura sia chiara e scorrevole; il resto del lavoro è stato quello di tradurre tutto il contenuto presente in inglese nelle slide e di togliere le parti di codice che non sono strettamente necessarie al fine dello studio (e inoltre, almeno nell'anno 2019/2020, non sono richieste all'esame). Sempre nell'anno 2019/2020 faccio presente che non è stato svolto il modulo **1.10 – Interfaccia utente in .NET** e quindi non sarà oggetto di esame.

1.2. Introduzione

L'**ingegneria del software** è una disciplina tecnologica e gestionale che permette di affrontare in modo sistematico e quantificabile, in termini di tempi e costi, lo sviluppo e la manutenzione dei prodotti software. L'ingegneria del software si occupa quindi del nuovo sistema che deve essere costruito, ovvero il **prodotto** che si sta realizzando, e del suo **processo di sviluppo**. Si ha la necessità di ingegnerizzare entrambi questi elementi, infatti un prodotto mal ingegnerizzato è poco usabile, estendibile e manutenibile; allo stesso modo, un processo di sviluppo sbagliato o mal concepito porta alla creazione di un prodotto di scarsa qualità: tutto questo porta alla necessità di creare modelli sia per il prodotto che per il processo di sviluppo. Lo sviluppatore deve essere in grado di affrontare la complessità dei grossi progetti:

- sviluppare prodotti con le caratteristiche di qualità desiderate;
- gestire le risorse, specie quelle umane, in modo produttivo;
- rispettare i vincoli economici e di tempi specificati.

Il problema dello sviluppo del software è un problema di gestione della complessità; in altri settori, l'uomo ha imparato a progettare e costruire sistemi complessi usando i principi dell'ingegneria tradizionale per affrontare la complessità:

- suddivisione del progetto in sotto-progetti (moduli) e così via sino ad avere moduli facilmente progettabili;
- utilizzo di parti e sottosistemi già pronti, che possono essere sviluppati in casa o comprati da fornitori esterni;
- standardizzazione dei componenti in modo che possano essere facilmente collegabili e intercambiabili;
- utilizzo del calcolatore come ausilio per la progettazione, l'esecuzione di compiti ripetitivi e i calcoli.

Lo sviluppo di un progetto non è lasciato al caso o all'estro di pochi progettisti, ma è un'attività in larga misura sistematica: è un insieme strutturato di attività che regolano lo sviluppo di un prodotto software. La struttura del processo di sviluppo può avere una forte influenza sulla qualità, sul costo e sui tempi di realizzazione del prodotto. Il processo di sviluppo:

- stabilisce un ordine di esecuzione delle attività;
- specifica quali elaborati devono essere forniti e quando;
- assegna le attività agli sviluppatori;
- fornisce criteri per monitorare il progresso dello sviluppo, misurarne i risultati e pianificare i progetti futuri;
- copre l'intero ciclo di vita del software.

Le fasi del processo di sviluppo del software sono:

1. studio di fattibilità;
2. analisi dei requisiti;
3. analisi del problema;
4. progettazione;
5. realizzazione e collaudo dei moduli;
6. integrazione e collaudo del sistema;
7. installazione e training;
8. utilizzo e manutenzione.

Costruire un software con caratteristiche di qualità non significa solo produrre codice che funziona, ma anche dare risposta a due domande fondamentali: è stata costruita la cosa giusta? Il prodotto è stato costruito nel modo giusto?

Il processo di revisione può avvenire a prodotto costruito o in itinere e secondo approcci a black box o white box: l'**approccio a black box** prevede che un osservatore esterno esamini il prodotto software come se fosse una scatola nera e percepisca le qualità esterne che devono essere garantite (affidabilità, facilità d'uso, velocità, etc.); al contrario, nell'**approccio a white box** devono essere osservabili le qualità interne esaminando la struttura interna del prodotto software come se fosse una scatola trasparente: il software deve essere modulare,

leggibile, etc. Le qualità interne influenzano le qualità esterne e sono un modo per realizzare le qualità esterne. I **fattori di qualità** di un software sono:

- **correttezza**: un software si dice corretto se data una definizione dei requisiti che deve soddisfare, esso rispetta tali requisiti;
- **robustezza**: il software si dice robusto se si comporta in maniera accettabile anche in corrispondenza di situazioni anomale o comunque non specificate nei requisiti: nel caso di situazioni anomale, il software non deve causare disastri (perdita di dati o peggio) ma deve o terminare l'esecuzione in modo pulito o entrare in una modalità particolare in cui non sono più attive alcune funzionalità;
- **affidabilità**: il software si dice affidabile se le funzionalità offerte corrispondono ai requisiti e se in caso di guasto non produce danni fisici o economici: il software è quindi affidabile se è corretto e robusto;
- **facilità d'uso**: è la facilità con cui l'utilizzatore del software è in grado di imparare ad usare il sistema, utilizzare il sistema, fornire i dati da elaborare, interpretare i risultati e gestire le condizioni di errore;
- **efficienza**: è fare un buon utilizzo delle risorse hardware, ovvero dei processori, della memoria principale, delle memorie secondarie, dei canali di comunicazione, etc.;
- **estensibilità**: è la facilità con cui il software può essere modificato per soddisfare nuovi requisiti. La modifica di programmi di piccole dimensioni non è un problema serio, ma i sistemi software di medie e grandi dimensioni possono soffrire di fragilità strutturale: modificando un singolo elemento della struttura si rischia di far collassare l'intera struttura. Ci sono due principi essenziali: semplicità architetturale e modularità e decentralizzazione. Secondo il primo principio più l'architettura del sistema è semplice e più è facile da modificare; secondo il secondo principio invece più il sistema è suddiviso in moduli autonomi e più è facile che una modifica coinvolga un numero limitato di moduli;
- **riusabilità**: il software è riusabile se può essere riutilizzato completamente, o in parte, in nuove applicazioni. Permette di non dover reinventare soluzioni a problemi già affrontati e risolti. Influenza tutte le altre caratteristiche dei prodotti software;
- **verificabilità**: è la facilità con cui il prodotto software può essere sottoposto a test;
- **portabilità**: è la facilità con cui il prodotto software può essere trasferito su altre architetture hardware e/o software.

Molte di queste caratteristiche sono definibili soltanto in maniera intuitiva e poco rigorosa; alcune di loro sono anche in contrapposizione, infatti, ad esempio, non si può avere sia efficienza che portabilità: l'importanza dell'una o dell'altra caratteristica varia a seconda del settore applicativo. Il costo del software aumenta esponenzialmente se è richiesto un livello molto alto di una qualunque di queste caratteristiche.

1.3. Principi e concetti object-oriented

ADT

Nella programmazione basata sugli oggetti per definire un ADT, *Abstract Data Type*, occorre definire un'interfaccia, ovvero un insieme di operazioni pubbliche applicabili ai singoli oggetti di quel tipo. Per implementare un ADT, invece, occorre definire una classe che implementa l'interfaccia dell'ADT, formata da un insieme di attributi privati e un insieme di metodi pubblici e privati che accedono in esclusiva agli attributi privati.

Incapsulamento

Un ADT nasconde ai suoi utilizzatori (clienti) tutti i dettagli della sua struttura interna e del suo funzionamento interno. L'obiettivo dell'incapsulamento è nascondere le scelte progettuali, spesso soggette a cambiamenti, proteggendo le altre parti del programma (ovvero i clienti dell'ADT) da eventuali cambiamenti di tali scelte. I vantaggi sono la minimizzazione delle modifiche da fare durante le fasi di sviluppo e di manutenzione e l'aumento della possibilità di riutilizzo. L'incapsulamento è applicabile a tutti i livelli: ai singoli attributi membro di una classe, ai singoli componenti del sistema, etc.

Oggetti e classi

Ogni oggetto:

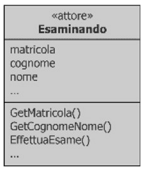
- è identificabile in modo univoco, ovvero ha una sua identità;
- ha un insieme di attributi;
- ha uno stato, ovvero l'insieme dei valori associati ai suoi attributi;
- ha un insieme di operazioni che operano sul suo stato e che forniscono servizi ad altri oggetti;
- ha un comportamento;
- interagisce con altri oggetti.

Gli oggetti sono raggruppabili in classi; ogni classe descrive oggetti con caratteristiche comuni, cioè con gli stessi attributi e con le stesse operazioni (lo stesso comportamento).

A tempo di compilazione, ogni classe definisce l'implementazione di un tipo di dato astratto; a tempo di esecuzione, ogni oggetto è un'istanza di una classe (traduzione comune, anche se impropria, del termine *instance*). Un'istanza è un particolare oggetto di una determinata classe e quindi di un particolare tipo; ogni istanza è separata dalle altre, ma condivide le sue caratteristiche generali con gli altri oggetti della stessa classe.

Notazione UML

Una classe si rappresenta come un rettangolo diviso in 1 o 3 sezioni. La prima sezione contiene il nome della classe (in grassetto se normale, in grassetto corsivo se astratta) e può contenere lo stereotipo della classe (ad esempio controllore, attore, evento, etc.) e il nome del pacchetto (package, namespace, etc.). La seconda sezione contiene gli attributi e la terza sezione contiene le operazioni (in corsivo se astratte).



Programmazione orientata agli oggetti

Nella programmazione orientata agli oggetti le classi possono essere organizzate in una gerarchia di generalizzazione o di ereditarietà che mostra la relazione tra classi e oggetti generiche e classi di oggetti più specifiche; gli oggetti della sottoclasse devono essere in grado di esibire tutti i comportamenti e le proprietà esibiti dagli oggetti appartenenti alla superclasse, in modo tale da poter essere "sostituiti" liberamente a questi ultimi. La sottoclasse può esibire caratteristiche aggiuntive rispetto alla superclasse ed eseguire in maniera differente alcune delle funzionalità della superclasse, a patto che questa differenza non sia osservabile dall'esterno.

Ereditarietà
Nell'ereditarietà gli attributi e le operazioni comuni devono essere specificati una volta sola, mentre gli attributi e le operazioni specifiche vengono aggiunti e/o ridefiniti. L'obiettivo è quello di semplificare la definizione e la realizzazione di tipi di dato simili. L'ereditarietà permette di esprimere in modo esplicito le caratteristiche comuni fin dalle prime attività dell'analisi. Un'alternativa molto valida all'ereditarietà è la composizione-delega, una modalità in cui si inserisce un'entità all'interno di una struttura dati di un'altra entità: in questo modo l'associazione tra le due entità può avvenire dinamicamente, a run-time, e si ha maggiore flessibilità ed estendibilità; l'ereditarietà si usa quindi se e solo se vale la relazione IsA, altrimenti si preferisce la composizione-delega.
L'ereditarietà può essere semplice, ovvero ogni classe della gerarchia deriva:

- da una e una sola superclasse (Java, .NET);
- al più da una sua superclasse (C++).

Nel caso dell'ereditarietà semplice la struttura che si ottiene è sempre un albero. Nel caso invece in cui l'ereditarietà è multipla, almeno una classe della gerarchia deriva da due o più superclassi (possibile in C++). Se esistono antenati comuni:

- la struttura che si ottiene è un reticolo;
- si hanno conflitti di nome;
- la gestione può diventare molto complessa.

Tra due o più classi di una gerarchia possono esistere dei vincoli {overlapping} o {disjoin}.

Polimorfismo
È la capacità della stessa cosa di apparire in forme diverse in contesti diversi e di cose diverse di apparire sotto la stessa forma in un determinato contesto.
La classificazione di Cardelli-Wegner prevede due macro-tipologie di polimorfismo: quello **universale**, che a sua volta può essere *per inclusione* e *parametrico*, e **ad-hoc**, che a sua volta può essere *di overloading* o si può rappresentare tramite la *coercion* (coercizione).

Programmazione generica rispetto ai tipi
Nella programmazione generica rispetto ai tipi esiste il concetto di classe generica: una classe in cui uno o più tipi sono parametrici. Ogni classe generata da una classe generica costituisce una classe indipendente: non esiste un legame di ereditarietà.

Regole di naming (.NET)
I nomi delle classi devono:

- iniziare con una lettera maiuscola;
- indicare al singolare un oggetto della classe, oppure indicare al plurale gli oggetti contenuti nella classe (se la classe è una classe contenitore).

Relazioni
La maggior parte delle classi (degli oggetti) interagisce con altre classi (altri oggetti) in vari modi. L'interazione tra entità diverse è possibile solo se tra loro esiste un qualche tipo di relazione; è quindi necessario modellare non solo le entità coinvolte ma anche le relazioni tra tali entità. Nella modellazione object-oriented le relazioni sono:

- generalizzazione/ereditarietà (IsA);
- realizzazione (implements);
- associazione: associazione generica, aggregazione (has) e composizione (has subpart);
- dipendenza: collaborazione (usa), relazione istanza-classe e relazione classe-metaclasse.

In ogni tipo di relazione, esiste un cliente C che dipende da un fornitore di servizi F. C ha bisogno di F per lo svolgimento di alcune funzionalità che C non è in grado di effettuare autonomamente, di conseguenza per il corretto funzionamento di C è indispensabile il corretto funzionamento di F.

Processo di sviluppo orientato agli oggetti

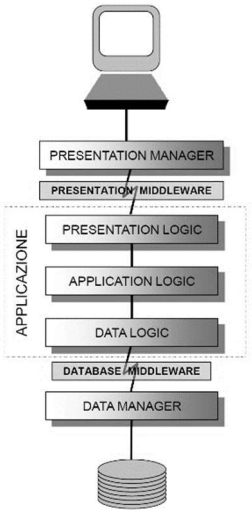
Il mondo reale è fatto di oggetti. Un sistema software OO rappresenta una porzione del mondo reale, o virtuale, ed è composto di oggetti software che interagiscono tra di loro. L'interazione tra gli oggetti software avviene mediante l'invio di messaggi: questo è concettualmente differente dall'invocazione di una funzione in quanto è l'oggetto che riceve il messaggio che decide quale funzione invocare per servire la richiesta.

L'**analisi orientata agli oggetti** (OOA) ha l'obiettivo di modellare la porzione del mondo reale (o virtuale) di interesse; gli oggetti di analisi modellano entità reali o concettuali della porzione del mondo reale d'interesse e le operazioni a esse associate. In questo contesto, ogni classe descrive una categoria di oggetti.

La **progettazione orientata agli oggetti** (OOD) ha l'obiettivo di modellare la soluzione: gli oggetti di analisi possono subire trasformazioni e sono introdotti gli oggetti di progettazione; in questo contesto, ogni classe descrive un tipo di dato.

La **programmazione orientata agli oggetti** (OOP) ha l'obiettivo di realizzare la soluzione. Vengono utilizzati linguaggi di programmazione OO (C++, C#, Java) che permettono di definire le classi di oggetti e sistemi run-time che permettono di creare, utilizzare e distruggere istanze di tali classi durante l'esecuzione. In questo contesto, ogni classe descrive l'implementazione di un tipo di dato.

1.4. Struttura a livelli di un'applicazione



Un'applicazione è formata da più livelli. Il livello più vicino all'utente è il **presentation manager**, ovvero il livello che si occupa della gestione dell'interazione con l'utente, come la gestione dello schermo, della tastiera, del mouse. Principalmente avviene tramite interfacce grafiche (API di sistema o browser HTML) ma può anche avvenire tramite interfacce a caratteri (emulatori).
Il livello **presentation logic** è il livello che si occupa della gestione dell'interazione con l'utente a livello logico:

- formattazione e visualizzazione delle informazioni (output);
- accettazione dei dati (input);
- prima validazione dei dati;
- gestione dei messaggi di errore.

L'**application logic** è la logica dell'applicazione e il controllo dei componenti, mentre la **data logic** è la gestione della persistenza a livello logico:

- consistenza dei dati;
- apertura e chiusura dei file;
- istruzioni SQL e transazioni;
- gestione degli errori.

L'ultimo livello, il **data manager**, si occupa della gestione della persistenza tramite file system, usando API di sistema, o tramite RDBMS, usando ad esempio DB2, Oracle, MS SQL Server, etc.
Sopra il data manager e sotto il presentation manager sono presenti, rispettivamente, il **database middleware** e il **presentation middleware**. Un middleware è un software che permette la comunicazione tra processi. Il database middleware è un software proprietario che trasferisce sulla rete le richieste SQL dall'applicazione al DBMS e i dati dal DBMS all'applicazione. Il presentation middleware, invece, è ad esempio un software di emulazione di terminali alfanumerici o una combinazione di web browser (lato client), web server, protocollo HTTP, etc.
Nel livello dell'application logic è presente un **application middleware**, che si occupa di gestire la comunicazione tra due componenti della stessa applicazione e può anche interconnettere tipi diversi di middleware.

1.5. Introduzione al Framework .NET

Il Framework .NET nasce come ambiente di esecuzione ed include anche una libreria di classi (standard ed estensioni MS); semplifica lo sviluppo e il deployment, aumenta l'affidabilità del codice e unifica il modello di programmazione. È inoltre completamente indipendente da COM, *Component Object Model*, e fortemente integrato con esso.
È possibile scegliere liberamente il linguaggio, infatti le funzionalità del framework .NET sono disponibili in tutti i linguaggi che sono compatibili; i componenti della stessa applicazione possono essere scritti in linguaggi diversi ed è supportata l'ereditarietà anche tra linguaggi diversi. È possibile estendere una qualsiasi classe .NET mediante l'ereditarietà, infatti, diversamente da COM, si usa e si estende la classe stessa e non si deve utilizzare la composizione-delega.
.NET è un'implementazione di CLI, *Common Language Infrastructure*, che, insieme al linguaggio C#, sono standard ECMA. Esistono anche altre implementazioni di CLI, come SSCLI, ROTOR, Mono, DotGNU, Intel OCL, etc. In CLI applicazioni scritte in differenti linguaggi di alto livello possono essere eseguite in ambienti diversi senza la necessità di riscriverle per considerare le caratteristiche individuali di ciascun ambiente.
Alcuni concetti chiave di .NET sono:

- **Microsoft IL**, *Intermediate Language*;
- **CLR**, *Common Language Runtime*: è l'ambiente di esecuzione run-time per le applicazioni .NET. Il codice che viene eseguito sotto il suo controllo si dice **codice gestito**;
- **CTS**, *Common Type System*: sono i tipi di dato supportati da .NET. Consente di fornire un modello di programmazione unificato;
- **CLS**, *Common Language Specification*: sono regole che i linguaggi di programmazione devono seguire per essere interoperabili all'interno di .NET; è un sottoinsieme di CTS.

Il codice può essere:

- **interpretato**: passa dalla sorgente all'interprete e infine all'esecuzione;
- **nativo**: passa dalle sorgenti al compilatore, che lo trasforma in codice nativo (.exe), e infine all'esecuzione;
- **IL**: passa dalle sorgenti al compilatore .NET, che lo trasforma in codice IL (assembly) .exe o .dll. Da qui passa al compilatore JIT in cui diventa codice nativo e può essere eseguito.

L'assembly è l'unità minima per la distribuzione e il versioning. Normalmente è composto da un solo file, ma può essere anche più di uno. È formato da metadati, suddivisi in:

- **manifest**: è la descrizione dell'assembly. Ha un'identità (nome, versione, cultura), una lista dei file che compongono l'assembly, i riferimenti ad altri assembly da cui dipende, i permessi necessari per l'esecuzione, etc.
- **tipi di dati**: sono metadati che descrivono tutti i tipi contenuti nell'assembly.

Oltre ai metadati sono presenti anche il codice IL, ottenuto da un qualsiasi linguaggio di programmazione, e le risorse (immagini, icone, etc.). I metadati vengono usati dai compilatori, dagli ambienti RAD, dai tool di analisi dei tipi e del codice e dagli sviluppatori.
Gli assembly possono essere:

- **privati**: sono utilizzati da un'applicazione specifica e si trovano nella sua directory;
- **condivisi**: sono utilizzati da più applicazioni e si trovano in C:\Windows\assembly;
- **scaricati** da URL: fanno parte della cache e si trovano in C:\Windows\assembly\download.

Il CLR offre vari servizi alle applicazioni, come funzionalità specifiche (ad esempio il garbage collector) o funzionalità esistenti (ad esempio I/O su file) mediate da CLR. Il garbage collector gestisce il ciclo di vita di tutti gli oggetti .NET; questi vengono distrutti automaticamente quando non sono più referenziati. A differenza di COM, non si basa sul *reference counting*, quindi si ha maggiore velocità di allocazione e sono consentiti i riferimenti circolari ma si perde la distruzione deterministica.

Per quanto riguarda le eccezioni in CLR, sono oggetti che ereditano dalla classe System.Exception. Esistono i concetti universali di lancio *(throw)*, cattura (*catch*) ed esecuzione di codice di uscita da un blocco controllato (*finally*); sono disponibili in tutti i linguaggi .NET con sintassi diverse.
I CTS, ovvero i tipi di dato supportati dal framework .NET, sono alla base di tutti i linguaggi .NET. CTS fornisce un modello di programmazione unificato e progettato per linguaggi object-oriented, procedurali e funzionali: sono esaminate le caratteristiche di 20 linguaggi, tutte le funzionalità sono disponibili con IL e ogni linguaggio utilizza alcune caratteristiche.
Alla base di CTS ci sono i tipi: classi, strutture, interfacce, enumerativi, delegati. È fortemente tipizzato, l'invocazione dei metodi virtuali viene risolta a run-time e si ha ereditarietà singola di estensione ed ereditarietà multipla di interfaccia.
In CTS tutto è un oggetto, infatti la classe radice è System.Object; esistono due categorie di tipi: quelli riferimento e quelli valore. I primi sono riferimenti a oggetti allocati sull'heap gestito e sono indirizzi di memoria, i secondi sono allocati sullo stack o sono parte di altri oggetti e sono una sequenza di byte.
Un qualsiasi tipo valore può essere automaticamente convertito in un tipo riferimento (boxing) mediante un up cast implicito a System.Object; un tipo valore "boxed" può tornare ad essere un tipo valore standard (unboxing) mediante un down cast esplicito. Un tipo valore "boxed" è un clone indipendente.
CLS definisce le regole di compatibilità tra linguaggi: per gli identificatori, per la denominazione di proprietà ed eventi, per costruttori degli oggetti, regole di overload più restrittive; sono ammesse interfacce multiple con metodi con lo stesso nome ma non sono ammessi puntatori unmanaged.

1.6. Garbage Collector in .NET

In un ambiente object-oriented, ogni oggetto che deve essere utilizzato dal programma è descritto da un tipo e ha bisogno di un'area di memoria dove memorizzare il suo stato. I passi per utilizzare un oggetto di tipo riferimento sono:

1. allocare la memoria per l'oggetto;
2. inizializzare la memoria per rendere utilizzabile l'oggetto;
3. usare l'oggetto;
4. eseguire un clean up dello stato dell'oggetto, se necessario;
5. liberare la memoria.

Per allocare la memoria ogni linguaggio ha le sue modalità: ad esempio in C si usano le funzioni *malloc*, *calloc* e *realloc*, in Java si usa *new* e in C# si usa *newobj*.
Quando si inizializza la memoria, a ogni variabile deve essere sempre assegnato un valore prima che essa venga assegnata (*definite assignment*): il compilatore deve assicurarsi che ciò sia sempre verificato tramite data-flow analysis del codice. I valori di default sono usati in generale per i tipi valore; ad esempio, in Java le variabili di classe, locali e i componenti di un array sono inizializzati al valore di default, ma non le variabili di istanza. Il costruttore viene utilizzato per i tipi classe in Java, C++ e C#.
Per il clean up dello stato si usa un distruttore (o più propriamente finalizzatore) in C++ e C#; questo è unico, non ereditabile, senza overload, parametri e modificatori. È invocato automaticamente alla distruzione dell'oggetto. In Java invece si usa il metodo di Object *finalize()*; viene invocato automaticamente alla distruzione dell'oggetto e il momento in cui viene invocato dipende dalla JVM.
Per liberare la memoria in C si usa la funzione *free()*, in C++ si usano le funzioni *free()* e *delete* mentre in Java e C# si usa il garbage collector.
La garbage collection è una modalità automatica di rilascio delle risorse utilizzate da un oggetto. Migliora la stabilità dei programmi, infatti evita errori connessi alla necessità, da parte del programmatore, di manipolare direttamente i puntatori alle aree di memoria. I pro sono:

- dangling pointer;
- doppia de-allocazione;
- memory leak.

I contro sono:

- aumenta la richiesta di risorse di calcolo;
- non si sa con certezza il momento in cui viene effettuate la garbage collection;
- il rilascio della memoria è non deterministico.

Le strategie di garbage collection attuabili sono tre:

1. **tracing**: si determinano gli oggetti potenzialmente raggiungibili e si eliminano gli oggetti non raggiungibili;
2. **reference counting**: ogni oggetto contiene un contatore che indica il numero di riferimenti a esso, quindi la memoria può essere liberata quando il contatore raggiunge lo 0;
3. **escape analysis**: si spostano gli oggetti dallo heap allo stack. L'analisi viene effettuata a compile-time in modo da stabilire se un oggetto, allocato all'interno di una subroutine, non è accessibile al di fuori di essa; in questo modo si riduce il lavoro del garbage collector.

Per quanto riguarda il tracing, un oggetto è raggiungibile soltanto in due casi: se è un oggetto radice, ovvero se è creato all'avvio del programma o da una sub-routine, o se è referenziato da un oggetto raggiungibile (la raggiungibilità è una chiusura transitiva). La definizione di garbage tramite la raggiungibilità non è ottimale, infatti può accadere che un programma utilizzi per la prima volta un oggetto molto prima che questo diventi irraggiungibile; occorre quindi fare una distinzione tra garbage sintattico, ovvero oggetti che il programma non può raggiungere, e garbage semantico, ovvero oggetti che il programma non vuole più usare.
Per quanto riguarda il reference counting, invece, ci sono molteplici svantaggi:

- cicli di riferimenti: se due oggetti si referenziano a vicenda, il loro contatore non raggiungerà mai lo 0;

- aumenta l'occupazione di memoria;
- si riduce la velocità delle operazioni sui riferimenti: ogni operazione su un riferimento deve anche incrementare e decrementare i contatori;
- atomicità dell'operazione: ogni modifica a un contatore deve essere resa operazione atomica in ambienti multi-threaded;
- assenza di comportamento real-time: ogni operazione sui riferimenti può potenzialmente causare la deallocazione di diversi oggetti; il numero di tali oggetti è limitato solamente dalla memoria allocata.

In fase di inizializzazione di un processo, il CLR riserva una regione contigua di spazio di indirizzamento **managed heap** e memorizza in un puntatore (netxtObjPtr) l'indirizzo di partenza della regione. Quando deve eseguire una *newobj*, il CLR:

1. calcola la dimensione in byte dell'oggetto e aggiunge all'oggetto due campi di 32 o 64 bit: un puntatore alla tabella dei metodi e un campo SyncBlockIndex;
2. controlla che ci sia spazio sufficiente a partire da NextObjPtr; in caso di spazio insufficiente si usa la garbage collection o si lancia una OutOfMemoryException;
3. thisObjPtr = NextObjPtr;
4. NextObjPtr += sizeof(oggetto);
5. invoca il costruttore dell'oggetto (this = thisObjPtr);
6. restituisce il riferimento all'oggetto.

Il garbage collector verifica se nell'heap esistono oggetti non più utilizzati dall'applicazione. Ogni applicazione ha un insieme di radici (root) e ogni radice è un puntatore che contiene l'indirizzo di un oggetto di tipo riferimento oppure vale *null*; le radici sono:

- variabili globali e field statici di tipo riferimento;
- variabili locali o argomenti attuali di tipo riferimento sugli stack dei vari thread;
- registri della CPU che contengono l'indirizzo di un oggetto di tipo riferimento.

Gli oggetti vivi sono quelli raggiungibili direttamente o indirettamente dalle radici, gli oggetti garbage sono quelli non raggiungibili direttamente o indirettamente dalle radici.
Quando si avvia, il garbage collector ipotizza che tutti gli oggetti siano garbage, quindi scandisce le radici e per ogni radice marca l'eventuale oggetto referenziato e tutti gli oggetti a loro volta raggiungibili a partire da tale oggetto. Se durante la scansione incontra un oggetto già marcato in precedenza, lo salta, sia per motivi di prestazione che per gestire correttamente riferimenti ciclici. Una volta terminata la scansione delle radici, tutti gli oggetti non marcati sono non raggiungibili e quindi garbage.
Nella seconda fase, rilascia la memoria usata dagli oggetti non raggiungibili, compatta la memoria ancora in uso modificando nello stesso tempo tutti i riferimenti agli oggetti spostati e unifica la memoria disponibile, aggiornando il valore di NextObjPtr. Tutte le operazioni che effettua sono possibili in quanto il tipo di un oggetto è sempre noto ed è possibile utilizzare i metadati per determinare quali field dell'oggetto fanno riferimento ad altri oggetti.
La fase di finalizzazione non è responsabilità del garbage collector ma del programmatore; se un oggetto contiene esclusivamente tipi valore e/o riferimenti a oggetti *managed*, non è necessario eseguire alcun codice particolare, mentre se un oggetto contiene almeno un riferimento a un oggetto *unmanaged* (in genere una risorsa del sistema operativo, come file, connessione a database, socket, etc.) è necessario eseguire del codice per rilasciare la risorsa, prima della deallocazione dell'oggetto. È necessario rilasciare (*dispose*) o chiudere (*close*) le risorse in modo deterministico, usando ad esempio le eccezioni, oppure se un tipo T vuole offrire ai suoi utilizzatori un servizio di clean up esplicito, deve implementare l'interfaccia IDisposable e i clienti di T possono utilizzare l'istruzione *using*; all'uscita del blocco *using*, viene sempre invocato automaticamente il metodo *dispose*.

1.7. Tipi in .NET

Dal punto di vista del modo in cui le istanze vengono gestite in memoria (rappresentazione, tempo di vita, etc.), i tipi possono essere distinti in **valore** e **riferimento**. Dal punto di vista sintattico (sintassi del linguaggio C#), i tipi possono essere distinti in:

- **classi**: *class*;
- **interfacce**: *interface*;
- **strutture**: *struct*;
- **enumerativi**: *enum*;
- **delegati**: *delegate*;
- **array**: [ ].

Questi in .NET si concretizzano sempre in una classe, anche nel caso di tipi built-in e di interfacce. In generale, un tipo può contenere la definizione di 0 o più:

- **costanti** – sempre implicitamente associate al tipo;
- **campi** – read-only o read-write, associati alle istanze o al tipo;
- **metodi** – associate alle istanze o al tipo;
- **costruttori** – di istanza o di tipo;
- **operatori** – sempre associati al tipo;
- **operatori di conversione** – sempre associati al tipo;
- **proprietà** – associate alle istanze o al tipo;
- **indexer** – sempre associati alle istanze;
- **eventi** – associati alle istanze o al tipo;
- **tipi** – annidati.

Modificatori di visibilità

Modificatore	Tipi a livello base	Tipi annidati	Operazioni - eventi	Dati
private	Non applicabile	Visibile nel tipo contenitore (default)		default
protected	Non applicabile	Visibile nel tipo contenitore e nei suoi sottotipi		Applicare esclusivamente a costanti ed eventualmente a campi read-only (è comunque preferibile l'accesso mediante una proprietà)
internal	Visibile nell'assembly contenitore (default)	Visibile nell'assembly contenitore		
protected internal	Non applicabile	Visibile sia nel tipo contenitore e nei suoi sottotipi, sia nell'assembly contenitore		
public	Visibilità completa	Visibilità completa		

I modificatori di visibilità non sono applicabili nei seguenti casi:

- **costruttori di tipo** (statici): sono sempre inaccessibili, infatti sono invocati direttamente dal CLR;
- **distruttori** (finalizer): sono sempre inaccessibili, anche loro sono invocati direttamente dal CLR;
- **membri di interfacce**: sono sempre pubblici;
- **membri di enum**: sono sempre pubblici;
- **implementazione esplicita di membri di interfacce**: hanno una visibilità particolare (pubblici/privati) e non è modificabile;
- **namespace**: sono sempre pubblici.

Le regole sono:

- massimizzare l'incapsulamento minimizzando la visibilità;
- nascondere le informazioni a livello di assembly: dichiarare *public* solo i tipi significati dal punto di vista concettuale;

- nascondere le informazioni a livello di classe: dichiarare *public* solo metodi, proprietà ed eventi significativi dal punto di vista concettuale e dichiarare *protected* solo le funzionalità che devono essere visibili nelle classi derivate, ma non esternamente;
- nascondere le informazioni a livello di field: field *private* e proprietà *public*, field *private* e proprietà *protected*.

Costanti e field

Una **costante** è un simbolo che identifica un valore che non può cambiare. Il tipo della costante può essere solo un tipo considerato primitivo dal CLR (compreso *string*); il valore deve essere determinabile a tempo di compilazione. I nomi delle costanti devono avere la maiuscola iniziale ed è possibile applicare *const* anche alle variabili locali.

Un **field** è un data member che può contenere un valore (un'istanza di tipo valore) oppure un riferimento (a un'istanza di un tipo riferimento), ovvero, in genere, la realizzazione di un'associazione. Un field può essere di **istanza** (default) oppure di **tipo** (static). Può essere **read-write** (default) oppure **read-only** (readonly) e viene inizializzato nella definizione o nel costruttore. Esiste sempre un valore di default (0, 0.0, false, null).

La regola base è che vanno definite come *const* solo le costanti vere, cioè i valori veramente immutabili nel tempo (nelle versioni del programma); negli altri casi vanno utilizzati field statici read-only. Per le costanti il nome dovrebbe iniziare con una lettera maiuscola e, di solito, dovrebbe essere pubblica, anche se non è sempre così. Il nome di un field deve iniziare con _ seguito da una lettera minuscola e deve essere privato (si dà l'accesso sempre mediante proprietà). Per i field read-only occorre scegliere, a seconda delle situazioni, una delle due convenzioni precedenti.

Modificatori di metodi

I modificatori di metodi *virtual*, *abstract*, *override*, *override sealed* e *sealed override* sono applicabili a metodi, proprietà (metodi *get* e *set*), indexer (metodi *get* e *set*) ed eventi (metodi *add* e *remove*).

L'implementazione di un metodo virtuale può essere cambiata dall'override di un membro in una classe derivata. Quando un metodo virtuale è invocato, a run-time il tipo di un oggetto è controllato per vedere se è stato effettuato un override. A default i metodi non sono virtuali.

Il modificatore *abstract* si usa per indicare che il metodo che lo usa non contiene un'implementazione. I metodi astratti sono implicitamente dei metodi virtuali e la dichiarazione di metodi astratti è permessa solo in classi astratte; l'implementazione è prevista da un metodo override.

Un metodo override provvede una nuova implementazione di un membro derivato da una classe base; il metodo sovrascritto da una dichiarazione override è definito come il metodo base sovrascritto. Il metodo base sovrascritto deve essere virtual, abstract o override e deve avere la stessa firma del metodo override.

Una dichiarazione di override non può cambiare l'accessibilità del metodo base sovrascritto e l'uso del modificatore *sealed* impedisce a una classe derivata di sovrascrivere ulteriormente il metodo.

Passaggio degli argomenti

Il passaggio degli argomenti prevede tre tipologie di argomenti:

- **in** (*default* in C#): l'argomento deve essere inizializzato, viene passato per valore (per copia) ed eventuali modifiche del valore dell'argomento non hanno effetto sul chiamante;
- **in/out** (*ref* in C#): l'argomento deve essere inizializzato, viene passato per riferimento ed eventuali modifiche del valore dell'argomento hanno effetto sul chiamante;
- **out** (*out* in C#): l'argomento può non essere inizializzato, viene passato per riferimento e le modifiche del valore dell'argomento (l'inizializzazione è obbligatoria) hanno effetto sul chiamante.

Le regole da rispettare sono l'utilizzare prevalentemente il passaggio standard per valore e utilizzare il passaggio per riferimento (*ref* o *out*) solo se strettamente necessario, ovvero se ci sono oltre due valori da restituire al chiamante ed è presente oltre un valore da utilizzare e modificare nel metodo. In questo caso occorre scegliere

ref se l'oggetto passato come argomento deve essere già stato inizializzato e *out* se è responsabilità del metodo inizializzare completamente l'oggetto passato come argomento.

Costruttori di istanza
La responsabilità dei costruttori di istanza è quella di inizializzare correttamente lo stato dell'oggetto appena creato. In mancanza di altri costruttori, esiste sempre un costruttore di default senza argomenti che, semplicemente, invoca il costruttore senza argomenti della classe base. Nel caso delle classi, il costruttore senza argomenti può essere definito dall'utente, mentre, nel caso delle strutture, il costruttore senza argomenti non può essere definito dall'utente per motivi di efficienza; in entrambi i casi è comunque possibile definire altri costruttori con differente signature e differente visibilità. Se il costruttore lancia un'eccezione e questa non viene gestita all'interno del costruttore stesso (quindi arriva al chiamante), questo non è un problema anche se in C++ è una situazione non facilmente gestibile.

Costruttori di tipo
La responsabilità dei costruttori di tipo è quella di inizializzare correttamente lo stato comune a tutte le istanze della classe. È dichiarato come *static* e implicitamente *private*; è sempre senza argomenti, quindi non si ha overloading. Può accedere esclusivamente ai membri (field, metodi, etc.) statici della classe. Se esiste, viene invocato automaticamente dal CLR prima della creazione della prima istanza della classe e prima dell'invocazione di un qualsiasi metodo statico della classe. Occorre non basare il proprio codice sull'ordine di invocazione di costruttori di tipo. Un costruttore di tipo va definito solo se strettamente necessario, ovvero se i campi statici della classe non possono essere inizializzati in linea e devono essere inizializzati solo se la classe viene effettivamente utilizzata. Se il costruttore lancia un'eccezione e questa non viene gestita all'interno del costruttore stesso (quindi arriva al chiamante), la classe non è più utilizzabile (TypeInitializationException).

Interfacce
In C#, un'interfaccia può contenere esclusivamente:
- **metodi**: considerati pubblici e astratti;
- **proprietà**: considerate pubbliche e astratte;
- **indexer**: considerati pubblici e astratti;
- **eventi**: considerati pubblici e astratti.

In CLR, un'interfaccia è considerata una particolare classe astratta di sistema che, ovviamente, non deriva da System.Object, però le classi che la implementano derivano per forza da System.Object.
Un'interfaccia può essere implementata sia dai tipi riferimento che dai tipi valore ed è considerata sempre un tipo riferimento.

Nella seguente tabella si mostrano le differenze tra un'interfaccia e una classe astratta:

Interfaccia	Classe astratta
Deve descrivere una funzionalità semplice, implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra di loro).	Può descrivere una funzionalità anche complessa, comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro).
Può "ereditare" da 0 a più interfacce.	Può "ereditare" da 0 a più interfacce e da 0 a più classi (astratte e/o concrete): in questo caso deve essere almeno una classe se esiste una classe radice di sistema e al massimo una classe se non è ammessa l'ereditarietà multipla.
Non può essere istanziata.	Non può essere istanziata.
Non può contenere uno stato.	Può contenere uno stato (comune a tutte le sottoclassi).
Non può contenere attributi membro e metodi (e proprietà ed eventi) statici (a parte eventuali costanti comuni)	Può contenere attributi membro e metodi (e proprietà ed eventi) statici.
Non contiene alcuna implementazione.	Può essere implementata completamente, parzialmente o per niente.
Le classi concrete che la implementano devono realizzare tutte le funzionalità.	Le classi concrete che la estendono devono realizzare tutte le funzionalità non implementate e possono fornire una realizzazione alternativa a quelle implementate.
Deve essere stabile: se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate.	Può essere modificata. Quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di default, in modo tale da non dover modificare le sottoclassi.
Non può gestire la creazione delle istanze delle classi che la implementano.	Può gestire la creazione delle istanze delle sue sottoclassi.
La creazione deve essere effettuata: • dai costruttori delle suddette classi; • da un'istanza di una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (classe factory).	La creazione può essere effettuata come per l'interfaccia, ma anche da un metodo statico della classe astratta (metodo factory).

1.8. Classi e interfacce base in .NET

In .NET tutte le classi derivano da System.Object e ogni metodo definito nella classe Object è disponibile per ogni oggetto nel sistema. Le classi derivate possono fare l'override di alcuni di questi metodi, tra cui:

- *Equals*: supporta il confronto tra oggetti;
- *ToString*: produce una stringa di testo leggibile dall'essere umano che descrive l'istanza di una classe;
- *GetHashCode*: genera un numero corrispondente al valore di un oggetto per supportare l'uso delle hash table;
- *Finalize*: esegue le operazioni di pulizia prima che un oggetto venga recuperato automaticamente.

Object.Equals
Il metodo *Equals* è definito come *public virtual bool Equals(object obj)* e restituisce *true* se l'oggetto è uguale a obj, *false* altrimenti.
L'implementazione di *Equals* non deve lanciare eccezioni e quando si fa l'override di *Equals* va fatto anche l'override di *GetHashCode*, altrimenti le *Hashtable* potrebbero non funzionare correttamente.
Se il linguaggio di programmazione supporta l'overloading dell'operatore e se si sceglie di fare l'overloading dell'operatore di uguaglianza per un determinato tipo, quel tipo dovrebbe fare l'override del metodo *Equals*. Tali implementazioni del metodo *Equals* dovrebbero restituire gli stessi risultati dell'operatore di uguaglianza.

System.IComparable
L'interfaccia IComparable ha un metodo pubblico definito come *CompareTo([in] obj : System.Object) : System.Int32*. Confronta l'istanza corrente con un altro oggetto dello stesso tipo. Restituisce un intero con segno a 32 bit che indica l'ordine relativo dei confronti; il valore restituito è minore di zero se l'istanza è inferiore a *obj*, zero se l'istanza è uguale a *obj* e maggiore di zero se l'istanza è maggiore di *obj*. Per definizione, qualsiasi oggetto è maggiore di un riferimento a *null*. Il parametro *obj* deve essere dello stesso tipo della classe o del tipo valore che implementa questa interfaccia; in caso contrario, viene generata una ArgumentException.
Esiste anche l'interfaccia IComparer che mette a disposizione il metodo pubblico *Compare([in] x : SystemObject, [in] y : System.Object) : System.Int32*. Questa interfaccia si usa insieme ai metodi di Array.Sort e di Array.BinarySearch e offre un modo per personalizzare l'ordinamento di una collezione di elementi.

System.IConvertible
Questa interfaccia fornisce i metodi per convertire il valore di un'istanza di un tipo di implementazione in un tipo supportato da CLR che ha un valore equivalente. Se non vi è alcuna conversione significativa in un tipo di CLR, allora l'interfaccia deve avere un metodo che permetta il lancio di InvalidCastException.

Conversione di tipo
Si ha **conversione ampliata** quando il valore di un tipo viene convertito in un altro tipo di dimensioni uguali o maggiori (ad esempio da Int32 a Int64 o da Int32 a Double). Si ha invece una **conversione ristretta** quando il valore di un tipo viene convertito nel valore di un altro tipo di dimensioni inferiori (ad esempio da Int32 a Int16 o da Int32 a Byte).
Le **conversioni implicite**, come ad esempio certe conversioni numeriche o l'up cast, non generano eccezioni. Nelle conversioni numeriche il tipo di destinazione dovrebbe essere in grado di contenere, senza perdita di informazione, tutti i valori ammessi dal tipo di partenza; nell'up cast, invece, vale il principio di sostituibilità, ovvero deve essere sempre possibile utilizzare una classe derivata al posto della classe base.
Le **conversioni esplicite**, invece, come ad esempio alcune conversioni numeriche o il down cast possono generare eccezioni. In questo caso nelle conversioni numeriche il tipo di destinazione non sempre è in grado di contenere il valore del tipo di partenza.
Altre conversioni di tipo sono il boxing (up cast), che è una conversione implicita e l'unboxing (down cast), che è una conversione esplicita.

L'utente ha a disposizione dei metodi per definire delle conversioni di tipo: *public static implicit operator typeOut(typeIn obj)* e *public static explicit operator typeOut(typeIn obj)*. Questi metodi sono metodi statici di una classe o di una struttura; la keyword *implicit* indica l'utilizzo automatico (cast implicito) e quindi il metodo non deve generare eccezioni mentre la keyword *explicit* indica la necessità di un cast esplicito e quindi il metodo può generare eccezioni. *typeOut* è il tipo del risultato del cast e *typeIn* è il tipo del valore da convertire. *typeIn* o *typeOut* deve essere il tipo che contiene il metodo.
Tramite *ToString()* si può convertire un Int32 in una stringa e tramite *ToString(string formatString)* si può convertire con anche una formattazione particolare. È possibile anche convertire una stringa in un Int32 tramite il metodo *Int32.Parse(string str)*. Per effettuare la conversione di numeri formattati in un modo particolare esiste il metodo *Int32.Parse(string str, System.Globalization.NumberStyles style)*, in cui l'enumerativo *NumberStyles* contiene tutte le opzioni possibili.

System.Double
In questa classe è presente un metodo *TryParse(…)*, simile al metodo *Parse*, che però non lancia eccezioni nel caso in cui la conversione fallisce: se la conversione va a buon fine, restituisce *true* e il risultato viene settato al valore restituito dalla conversione; se invece la conversione non va a buon fine, restituisce *false* e il valore restituito è 0.

System.Enum
Questa classe offre metodi per:
- confrontare le istanze della classe Enum;
- convertire il valore di un'istanza in una stringa;
- convertire una stringa che contiene un numero in un'istanza della classe Enum;
- creare un'istanza di un enumerativo e un valore specifico.

System.DateTime
Rappresenta un istante nel tempo, generalmente espresso come data e ora del giorno. Il tipo di valore *DateTime* rappresenta date e orari con valori che vanno dalle 12:00:00 di mezzanotte, 1° gennaio 0001 alle 11:59:59, 31 dicembre 9999. I valori del tempo sono misurati in unità di 100 ns chiamate tick. *DateTime* rappresenta un istante nel tempo, mentre un *TimeSpan* rappresenta un intervallo di tempo.

System.String
Rappresenta una stringa immutabile a lunghezza fissa di caratteri Unicode. Una stringa è definita immutabile perché il suo valore non può essere modificato una volta che è stata creata; i metodi che modificano una stringa in realtà restituiscono una nuova stringa contenente la modifica. Se è necessario modificare il contenuto effettivo di un oggetto simile a una stringa, occorre utilizzare la classe System.Text.StringBuilder.

System.ICloneable
Questa interfaccia supporta la clonazione di un oggetto: si crea una nuova istanza di una classe con lo stesso valore di un'istanza esistente. Il metodo *Clone()* crea un nuovo oggetto che è una copia dell'istanza corrente. Il clone può essere implementato come una **copia superficiale**, in cui vengono duplicati solo gli oggetti di livello superiore e non vengono create istanze dei campi, o come una **copia profonda**, in cui tutti gli oggetti sono duplicati. Il metodo *Clone()* restituisce una nuova istanza dello stesso tipo dell'oggetto.

System.Collections.IEnumerable
Questa interfaccia espone un enumeratore che supporta una semplice iterazione su una collezione. Il metodo *GetEnumerator()* restituisce un enumeratore che può essere usato per iterare su una collezione; un enumeratore può soltanto leggere i dati in una collezione ma non li può modificare.

System.Collections.IEnumerator
Questa interfaccia offre a disposizione il metodo *Reset()*, che restituisce un enumeratore al suo stato iniziale, il metodo *MoveNext()*, che permette di muoversi sul prossimo elemento della collezione e restituisce *true* se l'operazione ha avuto successo o *false* altrimenti, e il metodo *Current()*, che restituisce l'oggetto a cui l'enumeratore si sta riferendo.
Questa interfaccia non deve essere implementata direttamente da una classe contenitore ma deve essere implementata da una classe separata (eventualmente annidata nella classe contenitore) che fornisce la funzionalità di iterare sulla classe contenitore. Tale suddivisione di responsabilità permette di utilizzare contemporaneamente più enumeratori sulla stessa classe contenitore. La classe contenitore deve implementare l'interfaccia IEnumerable. Se una classe contenitore viene modificata, tutti gli enumeratori ad essa associati vengono invalidati e non possono più essere utilizzati.

System.Array
Questa classe mette a disposizione array mono-dimensionali e multi-dimensionali (rettangolari o anche di altre dimensioni non standard).

1.9. Delegati ed eventi in .NET

I **delegati** sono oggetti che possono contenere il riferimento a un metodo tramite il quale il metodo può essere invocato. Sono oggetti funzione (functor), ovvero oggetti che si comportano come una funzione. Sono simili ai puntatori a funzione del C/C++, ma object-oriented e molto più potenti. L'utilizzo standard è per le **funzionalità di callback**:

- elaborazione asincrona;
- elaborazione cooperativa: il chiamato fornisce una parte del servizio, il chiamante fornisce la parte rimanente (ad esempio il quicksort in C);
- gestione degli eventi: chi è interessato a un certo evento si registra presso il generatore dell'evento, specificando il metodo che gestirà l'evento.

L'istanza di un delegato incapsula uno o più metodi (con un particolare set di argomenti e il tipo restituito), ciascuno dei quali è riferito a un'entità richiamabile: per i metodi statici l'entità richiamabile è semplicemente un metodo, per le istanze di metodi invece un'entità richiamabile è un'istanza e un metodo su quell'istanza. Un delegato applica solo una firma del metodo e non sa e non ha interesse nel sapere a che classe o oggetto sta riferendo; questo rende i delegati adatti per l'invocazione anonima.
L'invocazione di un'istanza delegata, il cui elenco di invocazione contiene più voci, procede invocando ciascuno dei metodi nell'elenco di invocazione in modo sincrono e in ordine. Ad ogni metodo chiamato viene trasmessa la stessa serie di argomenti fornita all'istanza del delegato; se tale invocazione di un delegato include parametri di riferimento ogni invocazione del metodo avverrà con un riferimento alla stessa variabile e i cambiamenti a quella variabile, eseguiti con un metodo nell'elenco di invocazione, saranno visibili ai metodi più in basso nell'elenco di invocazione. Se l'invocazione del delegato include parametri di output o un valore di ritorno, il loro valore finale verrà dall'invocazione dell'ultimo delegato nell'elenco.
Un esempio di delegato è il modello boss-worker: è necessario modellare un worker che effettua un'attività, o un lavoro, e un boss, che controlla l'attività dei suoi worker; ogni worker deve notificare al proprio boss quando il lavoro inizia, quando è in esecuzione e quando finisce. Le soluzioni possibili sono varie e includono: soluzioni class-based, interface-based, delegate-based, event-based e usando il pattern Observer.
Un delegato è un'entità type-safe che si pone tra un caller e 0 o più call target e che agisce come un'interfaccia con un solo metodo.
Usare campi pubblici per la registrazione offre troppo accesso: un cliente può sovrascrivere dei target precedentemente registrati e li può invocare; un'alternativa migliore sarebbe avere metodi di registrazione pubblici insieme a campi delegati privati, ma può risultare noioso da fare manualmente: per questo si usa il modificatore ***event***, che automatizza il supporto per la pubblica registrazione e l'implementazione privata.
Possono essere forniti gestori di registrazione eventi definiti dall'utente: un vantaggio di scrivere i propri metodi di registrazione è il controllo; la sintassi alternativa, simile alle proprietà, supporta i gestori di registrazione definiti dall'utente e consente di subordinare la registrazione o di personalizzarla in altro modo. la sintassi dell'accesso lato client non è interessata ma è necessario fornire spazio di archiviazione per i clienti registrati.
Un evento, in programmazione, può essere scatenato dall'interazione con l'utente o dalla logica del programma. Ci sono tre entità relative agli eventi:

- **event sender**: l'oggetto o la classe che scatena l'evento, ovvero la sorgente dell'evento;
- **event receiver**: l'oggetto o la classe per il quale l'evento è determinante e che quindi desidera essere notificato quando l'evento si verifica;
- **event handler**: il metodo dell'event receiver che viene eseguito all'atto della notifica.

Quando si verifica l'evento, il sender invia un messaggio di notifica a tutti i receiver; in genere, il sender non conosce né i receiver né gli handler e il meccanismo che viene utilizzato per collegare sender e receiver/handler è il delegato (che permette invocazioni anonime).
Un evento incapsula un delegato, quindi è necessario dichiarare un tipo di delegato prima di poter dichiarare un evento. Per convenzione, i delegati degli eventi in .NET hanno due parametri: la fonte che ha scatenato l'evento e i dati per l'evento. Molti eventi, come ad esempio molti eventi legati all'interfaccia utente, non hanno

dati: in queste situazioni basta utilizzare il delegato presente nella classe System.EventHandler in quanto è sufficiente; occorre definire nuovi delegati per gestire gli eventi solo quando gli eventi hanno dei dati.
Quando un evento non deve passare informazioni aggiuntive ai propri gestori si usa la classe System.EventArgs; quando invece sono necessarie informazioni aggiuntive è necessario derivare una classe dalla classe EventArgs, aggiungere i dati necessari e usare il delegato EventHandler<TEventArgs>.
Quando si dichiara un evento, ad esempio *public event EventHandler Changed*, la keyword *event* ne limita la visibilità e le possibilità di utilizzo; una volta dichiarato, l'evento può essere trattato come un delegato di tipo speciale e in particolare può essere *null* se nessun cliente si è registrato e può essere associato a uno o più metodi da invocare.
Per scatenare un evento è opportuno definire un metodo protetto virtuale *onNomeEvento* (ad esempio *onChanged*) e invocare sempre quello; l'invocazione dell'evento può avvenire solo all'interno della classe nella quale l'evento è stato dichiarato, benché l'evento sia stato dichiarato come pubblico.
Al di fuori della classe in cui l'evento è stato dichiarato, un evento viene visto come un delegato con accessi molto limitati; le sole operazioni effettuabili dal cliente sono agganciarsi a un evento (aggiungere un nuovo delegato all'evento mediante l'operatore +=) e sganciarsi da un evento (rimuovere un delegato dall'evento mediante l'operatore -=).
Per iniziare a ricevere le notifiche di un evento, il cliente deve definire il metodo (event handler) che dovrà essere invocato all'atto della notifica dell'evento (con la stessa signature dell'evento) e creare un delegato dello stesso tipo dell'evento, farlo riferire al metodo e aggiungerlo alla lista dei delegati associati all'evento.
Per smettere di ricevere le notifiche di un evento, il cliente deve rimuovere il delegato dalla lista dei delegati associati all'evento.
Poiché += e -= sono le uniche operazioni consentite su un evento al di fuori del tipo che dichiara l'evento, un codice esterno può aggiungere e rimuovere handler per un evento ma non può ottenere o modificare in nessun altro modo l'elenco sottostante dei gestori di eventi.
Gli eventi forniscono un modo generalmente utile per gli oggetti di segnalare cambiamenti di stato che potrebbero essere utili ai client di quell'oggetto e sono un elemento fondamentale per la creazione di classi che possono essere riutilizzate in un gran numero di programmi diversi.

1.10. Interfaccia utente in .NET

Il design e lo sviluppo tipici dei programmi Windows comprendono:

- 1 o più classi derivate da System.Windows.Forms.Form;
- un design dell'interfaccia utente con VisualStudio .NET;
- la possibilità di fare qualsiasi cosa direttamente tramite codice.

La gestione dei thread dei programmi Windows prevede:

- un singolo thread dedicato all'interfaccia utente che esegue il pump dei messaggi e può fare anche altre cose, ma si interrompe solo per un breve periodo (o mai);
- altri thread che lavorano in background per svolgere altre funzionalità non legate all'interfaccia.

System.Windows.Forms.Application

È una classe non instanziabile con metodi, proprietà ed eventi statici pubblici; si usa per gestire l'infrastruttura dell'applicazione Windows e offre:

- metodi per effettuare il pump dei messaggi, come ad esempio *Run(Form form)* ed *Exit()*, che informa tutti i messaggi che devono terminare e successivamente chiude tutte le richieste una volta che il messaggio è stato processato;
- Eventi a livello di applicazione: *Idle*, *ApplicationExit*.

System.Windows.Forms

È un namespace che contiene classi per la creazione di applicazioni su Windows. Le classi possono essere raggruppate nelle seguenti categorie: Components, Common Dialog Boxes e Controls (Form e UserControl).

System.Drawing

Contiene oggetti grafici di base:

- classi: Graphics, Font, Brush, Pen, Icon, Bitmap, etc.;
- creatori di istanze: Brushes, Pens, SystemBrushes, SystemColors, SystemIcons, Cursor;
- Strutture: Point, Size, Rectangle, Color, etc.

Al suo interno è contenuta la classe System.Drawing.Graphics, che rappresenta una superficie in cui si può disegnare: questa può essere in memoria, basata su form o su HDC. Si usa per "disegnare" e "dipingere" i controlli: DrawString(), DrawImage(), FillEllipse(), FillRectangle(), etc.

Components

Il termine componente viene generalmente utilizzato per un oggetto riutilizzabile e in grado di interagire con altri oggetti. Un componente del framework .NET soddisfa tali requisiti generali e fornisce inoltre supporto in fase di progettazione. Un componente può essere utilizzato in un ambiente di sviluppo rapido di applicazioni:

- può essere aggiunto alla casella degli strumenti di Visual Studio .NET;
- può essere trascinato e rilasciato su un modulo;
- può essere manipolato su una superficie di progettazione.

Il supporto della fase di progettazione di base è integrato in .NET: uno sviluppatore di componenti non deve svolgere nessun compito aggiuntivo per sfruttare queste funzionalità.

Common Dialog Boxes

Le finestre di dialogo comuni possono essere utilizzate per fornire ad un'applicazione un'interfaccia utente coerente quando si eseguono attività come l'apertura e il salvataggio di file, la manipolazione dei caratteri o del colore del testo o la stampa. Alcune di queste finestre sono: OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog, PageSetupDialog, PrintPreviewDialog e PrintDialog.
Inoltre, il namespace System.Windows.Forms fornisce la classe MessageBox per la visualizzazione di una finestra di messaggio che può visualizzare e recuperare i dati dall'utente.

Controls

Un controllo è un componente che fornisce (o abilita) le funzionalità dell'interfaccia utente.
Ci sono vari tipi di controlli:

- quelli progettati per l'immissione di dati, come ad esempio TextBox, ComboBox, etc.;
- quelli che permettono di visualizzare i dati dell'applicazione, come ad esempio Label, ListView, TreeView, DataGridView, etc.;
- quelli per richiamare i comandi all'interno dell'applicazione, come Button, LinkLabel, etc.

Esistono anche i contenitori per i controlli dei figli, come Panel, SplitContainer, TableLayoutPanel, etc. e i contenitori di componenti come ToolStrip, MenuStrip, ContextMenuStrip, etc.
La classe System.Windows.Forms.Controls è la classe base per tutti i controlli/moduli; deriva da Component, fornisce le funzionalità di base per tutti i controlli e avvolge il tutto in una finestra di gestione del sistema operativo sottostante. Implementa:

- proprietà per la modifica delle impostazioni di un'istanza: Size, BackColor, CntextMenu, etc.;
- metodi per eseguire azioni su un'istanza: Show(), Hide(), Invalidate(), etc.;
- eventi per la registrazione "esterna" per la notifica di eventi: Click, DragDrop, ControlAdded, etc.

Le classi derivate sovrascrivono e specializzano le funzionalità, come ad esempio metodi, proprietà ed eventi: TextBox – PasswordChar, Undo(), Copy(), Button – Image, PerformClick().
Per usare i controlli dal punto di vista del design occorre:

1. aggiungere il controllo al contenitore;
2. modificare l'aspetto e il comportamento del controllo impostando proprietà;
3. definire e registrare i metodi per gestire gli eventi della GUI: pulsanti clic, selezioni di menu, movimenti del mouse, eventi timer, etc.; il comportamento di default è implementato dalle classi base.

Dal lato del codice invece, occorre:

1. creare e aggiungere il controllo;
2. impostare le proprietà;
3. definire un event handler;
4. registrare il controllo per la notifica di un evento.

Forms

È un namespace contenuto in System.Windows.Forms.Form. È una derivazione specializzata di Control utilizzata per implementare una finestra o una finestra di dialogo di livello superiore. Ottiene gran parte delle sue funzionalità dalle classi base e le specializzazioni includono:

- una barra per il titolo, un menu di sistema, le funzioni per minimizzare e massimizzare la finestra;
- un menu principale;
- la gestione dei pulsanti della finestra di dialogo;
- l'implementazione di MDI, *Multiple Document Interface*.

Il namespace System.Windows.Forms offre classi che non derivano dalla classe Control ma che possono comunque offrire funzionalità grafiche a un'applicazione; le classi ToolTip e ErrorProvider offrono informazioni all'utente e le classi Help e HelpProvider offrono le funzionalità per mostrare all'utente informazioni di aiuto.

1.11. Metadati e introspezione in .NET

I **metadati** sono dati che permettono di descrivere altri dati: ad esempio, la definizione di una classe sono metadati. A condizione che un componente sia dotato di informazioni sufficienti per essere auto-descrittivo, le interfacce supportate da un componente possono essere esplorate dinamicamente.
In C++ possono essere considerati come metadati i file header: i clienti li includono a tempo di compilazione per usare i tipi che vengono dichiarati. I file header in C++ sono specifici del linguaggio e offrire informazioni tra linguaggi diversi è un problema comune.
COM e CORBA, *Common Object Request Broker Architecture* usano IDL per offrire metadati. Sono una cosa in più da capire per gli sviluppatori e sono suddivisi in file diversi a seconda di quale tipo descrivono.
In .NET e in Java i metadati sono generati dalla definizione di un tipo. Sono salvati insieme alla definizione del tipo e sono disponibili a run-time: questo è il concetto di **riflessione**. Si può usare la riflessione per:

- esaminare i dettagli di un assembly;
- istanziare oggetti e chiamare metodi scoperti in fase di esecuzione;
- creare, compilare ed eseguire assembly al volo.

Il punto centrale della riflessione è System.Type. Tutti gli oggetti e tutti i valori sono istanze di tipi; i tipi stessi sono istanze del tipo System.Type e c'è un singolo oggetto Type per ogni tipo definito nel sistema.
È possibile creare istanze di tipi e/o accedere ai membri con un late binding:

- si possono istanziare tipi in memoria, scegliendo il costruttore da chiamare;
- si possono invocare metodi;
- si possono invocare i metodi getter e setter.

I membri pubblici sono sempre accessibili e i membri non pubblici sono accessibili solo se i chiamanti hanno i permessi necessari. La classe System.Activator permette di creare istanze dinamicamente: è l'equivalente dell'operatore *new* che permette il late binding. Alloca spazio per le istanze dei tipi, chiama il costruttore specifico e restituisce un riferimento a un oggetto generico.
Gli **attributi personalizzati** sono un semplice modo per aggiungere informazioni ai metadati per ogni tipo di applicazione. Possono essere aggiunti ad un assembly usando una sintassi speciale e possono essere usati in modo tale che i clienti possano raccogliere automaticamente certe funzionalità. Sono supportati in qualsiasi linguaggio .NET e sono semplicemente classi che derivano da System.Attribute. Possono contenere metodi e proprietà. Per creare attributi personalizzati occorre:

- dichiarare la classe dell'attributo;
- dichiarare i costruttori;
- dichiarare le proprietà;
- applicare l'AttributeUsageAttribute (facoltativo): specifica alcune delle caratteristiche della classe, come ad esempio a quali elementi l'attributo è applicabile, quando l'attributo può essere ereditato e quando no, quando possono esistere molteplici istanze di un attributo per un elemento e quando no.

Una volta che gli attributi personalizzati sono stati creati, si usa la riflessione per leggerli. Si può ottenere una lista degli attributi personalizzati tramite il metodo *GetCustomAttributes*.
Numerose classi funzionano insieme per offrire la programmazione tramite metadati in .NET. Utilizzando gli oggetti precedenti e altri, è possibile creare un assembly al volo:

- System.Reflection.Emit consente di scrivere l'IL necessario per creare e compilare l'assembly;
- è quindi possibile chiamare questo assembly dal programma che lo ha creato;
- l'assembly può essere archiviato sul disco in modo tale che altri programmi possano utilizzarlo.

La classe System.Reflection permette di definire l'identità unica di un assembly mentre la classe System.Reflection.Emit permette di crearlo e compilarlo.

1.12. Principi di design

La qualità del design è un concetto sfuggente. La qualità dipende da specifiche priorità organizzative, infatti un buon design può essere, ad esempio, il rendere un programma:

- il più affidabile;
- il più efficiente;
- il più manutenibile;
- il più economico.

Ciò che rende un design un cattivo design può essere:

- una cattiva interpretazione dei requisiti;
- rigidità del software: una singola modifica influisce su molte altre parti del sistema;
- fragilità del software: una singola modifica influisce su parti non previste del sistema;
- immobilità del software: è difficile riutilizzarlo in un'altra applicazione;
- viscosità: è difficile fare la cosa giusta ma è facile fare la cosa sbagliata.

La **rigidità del software** è la tendenza del software a non essere facile da cambiare. Ogni modifica causa una serie di modifiche a cascata in moduli dipendenti tra di loro e quindi per una piccola modifica è necessario modificare gran parte del programma; l'effetto di questo problema è che si ha paura nel correggere problemi non critici in quanto ciò potrebbe richiedere molto tempo.

La **fragilità del software** è la tendenza del software a rompersi in molti punti ogni volta che viene cambiato; i cambiamenti tendono a causare comportamenti imprevisti in altre parti del sistema (spesso in aree che non hanno relazioni concettuali con l'area che è stata modificata). Ogni correzione peggiora le cose, introducendo più problemi di quanti ne vengano risolti, quindi tale software è impossibile da mantenere; come effetto si ha che ogni volta che si fa una correzione, si teme che il software si rompa in modo imprevisto.

L'**immobilità del software** è l'incapacità di riutilizzare il software da altri progetti o da parti dello stesso progetto. Ad esempio, uno sviluppatore scopre di aver bisogno di un modulo simile a quello scritto da un altro sviluppatore, ma il modulo in questione si collega troppo ad altre parti del software. Dopo molto lavoro, lo sviluppatore scopre che il lavoro e il rischio necessari per separare le parti desiderabili del software da quelle non desiderate sono troppo grandi da tollerare, quindi non ne vale la pena. Questo si tramuta nel riscrivere da capo il software al posto di riutilizzarlo.

Per quanto riguarda la **viscosità**, gli sviluppatori di solito trovano più di un modo per apportare una modifica: alcuni preservano il design, altri no (ovvero sono hack). Si ha quindi la tendenza a incoraggiare i cambiamenti del software che sono hack piuttosto che cambiamenti del software che preservano l'intento di progettazione originale. Esistono due tipi di viscosità:

- **viscosità del design**: implementare metodi efficaci che mantengano il design è più difficile che usare hack;
- **viscosità dell'ambiente**: l'ambiente di sviluppo è lento e inefficiente (i tempi di compilazione sono molto lunghi, il sistema di controllo del codice sorgente richiede ore per registrare pochi file, etc.).

Si ha quindi che è facile fare la cosa sbagliata ed è difficile fare la cosa giusta: questo comporta una degenerazione della manutenibilità del software a causa di hack, scorciatoie, fix temporanei, etc.

Il motivo per cui si possono fare scelte cattive sono ovvi, come ad esempio una scarsa conoscenza di come si fa un buon design, tecnologie che cambiano, limiti di tempo o risorse, etc. Motivi meno ovvi sono, ad esempio:

- la decomposizione del software: questo è un processo lento ed anche un design originariamente pulito ed elegante può degenerare nel corso dei mesi/anni;
- i requisiti cambiano spesso in un modo in cui il design originale o il designer non prevedevano;
- si vengono a creare dipendenze non pianificate e improprie che poi non vengono gestite.

Nell'ingegneria del software i requisiti cambiano. In effetti, il documento dei requisiti è il documento più volatile del progetto: se il nostro design fallisce a causa dei costanti requisiti in evoluzione, sono i nostri progetti a essere colpiti e bisogna, in qualche modo, trovare un modo per rendere i nostri progetti resistenti a tali cambiamenti e proteggerli dalla decomposizione. Tutti i sintomi sopra menzionati sono direttamente o

indirettamente causati da dipendenze improprie tra i moduli del software. È l'architettura delle dipendenze che sta degradando e con essa la capacità di mantenere il software; per prevenire il degrado dell'architettura delle dipendenze, è necessario gestire le dipendenze tra i moduli di un'applicazione: per questo la progettazione orientata agli oggetti è piena di principi e tecniche per la gestione delle dipendenze dei moduli.
Alcuni principi di design sono:

- il principio di singola responsabilità;
- il principio di inversione delle dipendenze;
- il principio di segregazione delle interfacce;
- il principio aperto/chiuso;
- il principio di sostituzione di Liskov.

Esiste un ulteriore principio, detto **il principio zero**: non bisogna introdurre concetti che non siano strettamente necessari. Significa che tra due soluzioni bisogna preferire quella che introduce il minor numero di ipotesi e che fa uso del minor numero di concetti.
La **semplicità** è un fattore importantissimo, infatti il software deve fare i conti con una notevole componente di complessità, generata dal contesto in cui deve essere utilizzato: è quindi estremamente importante non aggiungere altra complessità arbitraria. Il problema è che la semplicità richiede uno sforzo non indifferente e in generale le soluzioni più semplici vengono in mente per ultime. Bisogna fare poi molta attenzione ad essere semplici ma non semplicistici.
Un approccio valido per iniziare è quello del **divide et impera**: consiste nel dividere un problema complesso in tanti sotto-problemi fin quando questi non diventano semplici e facilmente risolvibili. La decomposizione è una tecnica fondamentale per il controllo e la gestione della complessità: non esiste un solo modo per decomporre il sistema, ma la qualità della progettazione dipende direttamente dalla qualità delle scelte di decomposizione adottate. In questo contesto il principio fondamentale è quello di minimizzare il grado di accoppiamento tra i moduli del sistema; alcune regole sono, infatti: minimizzare la quantità di interazione fra i moduli, eliminare tutti i riferimenti circolari fra moduli, etc.
Il **principio di singola responsabilità** afferma che ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso. Tutti i servizi offerti dall'elemento dovrebbero essere strettamente allineati a tale responsabilità.
Il **principio di inversione delle dipendenze** afferma che i moduli di alto livello, ovvero i clienti, non dovrebbero dipendere dai moduli di basso livello, ovvero i fornitori di servizi, ma entrambi dovrebbero dipendere da astrazioni. I moduli di basso livello contengono la maggior parte del codice e della logica implementativa e quindi sono i più soggetti a cambiamenti; se i moduli di alto livello dipendono dai dettagli dei moduli di basso livello (ovvero sono accoppiati in modo troppo stretto), i cambiamenti si propagano e le conseguenze sono:

- rigidità: bisogna intervenire su un numero elevato di moduli;
- fragilità: si introducono errori in altre parti del sistema;
- immobilità: i moduli di alto livello non si possono riutilizzare perché non si riescono a separare da quelli di basso livello.

Questo principio funziona perché:

- le astrazioni contengono pochissimo codice (in teoria nulla) e quindi sono poche soggette a cambiamenti;
- i moduli non astratti sono soggetti a cambiamenti ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli.

I dettagli del sistema sono stati isolati, separati da un muro di astrazioni stabili, e questo impedisce ai cambiamenti di propagarsi (***design for change***); allo stesso tempo i singoli moduli sono maggiormente riusabili perché sono disaccoppiati fra di loro (***design for reuse***).

A causa di questo principio, i sistemi software dovrebbero essere stratificati, cioè organizzati a livello; le dipendenze transitive e quelle cicliche vanno eliminate, mentre le astrazioni in uso non devono essere modificate ma devono essere estese.
Il **principio di segregazione delle interfacce** afferma che un cliente non dovrebbe dipendere da metodi che non usa, e che pertanto è preferibile che le interfacce siano molte, specifiche e piccole (composte da pochi metodi) piuttosto che poche, generali e grandi. Le *fat interfaces* creano una forma indiretta di accoppiamento (inutile) fra i clienti e se un cliente richiede l'aggiunta di una nuova funzionalità all'interfaccia, ogni altro cliente è costretto a cambiare anche se non è interessato alla nuova funzionalità: questo crea un inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori. Se i servizi di una classe possono essere suddivisi in gruppi e ogni gruppo viene utilizzato da un diverso insieme di clienti, occorre creare interfacce specifiche per ogni tipo di cliente e implementare tutte le interfacce nella classe.
Il **principio aperto/chiuso** è il principio più importante per progettare entità riutilizzabili: afferma che le entità (classi, moduli, funzioni, etc.) software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche. I moduli andrebbero scritti in modo tale da essere estesi senza però la necessità di modificarli: per fare questo occorre usare l'astrazione, infatti è possibile creare astrazioni che rendono un modulo immutabile rappresentando un gruppo illimitato di componenti. Si possono utilizzare interfacce (o classi astratte) dato che a un'interfaccia immutabile possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi. Un modulo che utilizza astrazioni non dovrà mai essere modificato, dal momento che le astrazioni sono immutabili, ma potrà cambiare comportamento se si utilizzano nuove classi che implementano le astrazioni. Una volta che la maggior parte dei moduli di un programma rispetta questo principio, le nuove funzionalità sono implementabili semplicemente aggiungendo nuovo codice piuttosto che cambiare il codice già fatto.
Il **principio di Liskov** afferma che un cliente che usa istanze di una certa classe A deve poter usare istanze di una qualsiasi sottoclasse di A senza accorgersi della differenza. Questo principio costituisce una guida per creare queste classi concrete mediante l'ereditarietà. La principale causa di violazioni al principio di Liskov è data dalla ridefinizione di metodi virtuali nelle classi virtuali; la chiave per evitare queste violazioni risiede nel ***design by contract***. Nel *design by contract* ogni metodo ha:

- un insieme di pre-condizioni: requisiti minimi che devono essere soddisfatti dal chiamante affinché il metodo possa essere eseguito correttamente;
- un insieme di post-condizioni: requisiti che devono essere soddisfatti dal metodo nel caso di esecuzione corretta.

Questi due insiemi di condizioni costituiscono un contratto tra chi invoca il metodo (cliente) e il metodo stesso (e quindi la classe a cui appartiene). Le pre-condizioni vincolano il chiamante, le post-condizioni vincolano il metodo; se il chiamante garantisce il verificarsi delle pre-condizioni, il metodo garantisce il verificarsi delle post-condizioni. Quando un metodo viene ridefinito in una sottoclasse, le pre-condizioni devono essere identiche o meno stringenti e le post-condizioni devono essere identiche o più stringenti: questo perché un cliente che invoca il metodo conosce il contratto definito a livello della classe base, quindi non è in grado di soddisfare pre-condizioni più stringenti o di accettare post-condizioni meno stringenti. In caso contrario, il cliente dovrebbe conoscere informazioni sulla classe derivata e questo porterebbe a una violazione del principio di Liskov.
Esistono anche tre principi per l'architettura dei package:

- il **principio di equivalenza di riuso/rilascio**: un elemento riutilizzabile, sia esso un componente, una classe o un cluster di classi, non può essere riutilizzato se non è gestito da un sistema di rilascio di qualche tipo. I clienti rifiuteranno o comunque dovrebbero rifiutare di riutilizzare un elemento a meno che l'autore non prometta di tenere traccia dei numeri di versione e mantenere le versioni precedenti per un po'. Poiché i package sono l'unità di rilascio in Java, sono anche l'unità di riutilizzo. Pertanto, chi progetta il sistema farebbe bene a raggruppare le classi riutilizzabili in package;
- il **principio di chiusura comune**: il lavoro necessario per gestire, testare e rilasciare un package non è banale in un sistema di grandi dimensioni. Maggiore è il numero di package che cambiano in una determinata versione, maggiore è il lavoro di rebuild, test e distribuzione della versione. Dato che si

vuole ridurre al minimo il numero di package che vengono modificati in ogni ciclo di rilascio del prodotto, si raggruppano le classi che si pensa possano cambiare insieme;

- il **principio di riutilizzo comune**: una dipendenza da un package è una dipendenza da tutto ciò all'interno del package. Quando un package cambia e il suo numero di versione cambia, tutti i client di quel package devono verificare che funzionino con il nuovo package, anche se nulla di ciò che hanno usato all'interno del package è effettivamente cambiato. Pertanto, le classi che non vengono riutilizzate insieme non devono essere raggruppate in un package.

Questi tre principi non possono essere soddisfatti contemporaneamente. Il principio di equivalenza di riutilizzo/rilascio e il principio di riutilizzo comune rendono il riutilizzo dei package più semplice, mentre il principio di chiusura comune semplifica la vita dei manutentori. Il principio di chiusura comune si impegna a rendere i package il più grandi possibile (dopo tutto, se tutte le classi vivono in un solo package, solo un package cambierà mai). Il principio di riutilizzo comune, tuttavia, cerca di rendere i package molto piccoli. All'inizio di un progetto, quando si progetta l'architettura, si può impostare la struttura dei package in modo che il principio di chiusura comune domini, per facilitare lo sviluppo e la manutenzione. Successivamente, man mano che l'architettura si stabilizza, si può modificare la struttura dei package per massimizzare gli altri due principi in modo tale da permettere il massimo riutilizzo.
Esistono ulteriori tre principi per le relazioni tra i package:

- il **principio delle dipendenze acicliche**: afferma che le dipendenze tra i package non devono formare dei cicli. Dopo aver apportato modifiche a un package, gli sviluppatori possono rilasciare i package nel resto del progetto. Prima di poter eseguire questo rilascio, tuttavia, devono verificare che il package funzioni. Per farlo, devono compilarlo e fare una build con tutti i package da cui dipende. Una singola dipendenza ciclica che sfugge al controllo può rendere l'elenco delle dipendenze molto lungo. Quindi, qualcuno deve guardare la struttura di dipendenza dei package con regolarità e interrompere i cicli ovunque appaiano: per rompere un ciclo si può aggiungere un package in mezzo o aggiungere una nuova interfaccia;
- il **principio delle dipendenze stabili**: le dipendenze tra i package, in un progetto, dovrebbero andare nella direzione della stabilità dei package. Un package dovrebbe dipendere solo da package più stabili di lui;
- il **principio delle astrazioni stabili**: afferma che i package stabili dovrebbero essere package astratti. La stabilità è correlata alla quantità di lavoro richiesta per apportare una modifica; un package con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare qualsiasi modifica con tutti i package dipendenti.

Nel complesso, si può dire che i package nella parte superiore sono instabili e flessibili mentre i package in fondo sono molto difficili da cambiare. I package altamente stabili nella parte inferiore della rete di dipendenza possono essere molto difficili da cambiare, ma secondo il principio aperto/chiuso non devono essere difficili da estendere. Se i package stabili nella parte inferiore sono anche molto astratti, possono essere facilmente estesi. È possibile comporre un'applicazione da package instabili facili da modificare e package stabili facili da estendere.

1.13. Pattern di design

I pattern di design catturano e formalizzano l'esperienza acquisita nell'affrontare e risolvere uno specifico problema progettuale. Permettono di riutilizzare l'esperienza in altri casi simili, risolvendo problemi progettuali e rendendo i progetti object-oriented più flessibili e riutilizzabili. Ogni pattern di design ha quattro elementi essenziali:

- **un nome significativo**: identifica il pattern;
- **il problema**: descrive quando applicare il pattern;
- **la soluzione**: descrive il pattern, cioè gli elementi che lo compongono (classi e istanze) e le loro relazioni, responsabilità e collaborazioni;
- **le conseguenze**: descrivono vantaggi e svantaggi dell'applicazione del pattern. Permettono di valutare le alternative progettuali.

I pattern di design possono essere:

- **di creazione**: risolvono problemi inerenti al processo di creazione di oggetti;
- **strutturali**: risolvono problemi inerenti alla composizione di classi o di oggetti;
- **comportamentali**: risolvono problemi inerenti alle modalità di interazione e di distribuzione delle responsabilità tra classi o tra oggetti.

Una prima tabella riassuntiva dei pattern è:

Pattern di creazione	Pattern strutturali	Pattern comportamentali
• Abstract Factory • Builder • Factory Method • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsability • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template Method • Visitor

Pattern Singleton

Assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. La classe deve:

- tenere traccia della sua sola istanza;
- intercettare tutte le richieste di creazione, al fine di garantire che nessun'altra istanza venga creata;
- fornire un modo per accedere all'istanza unica.

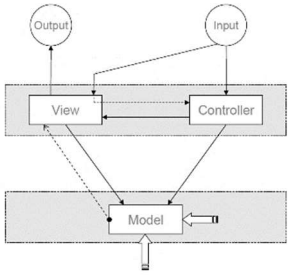
Un'alternativa è una classe non istanziabile (*static class*) con soli membri statici. I vantaggi del pattern singleton sono che può implementare una o più interfacce e che può essere specializzato e si può creare nella *GetInstance* un'istanza specializzata che dipende dal contesto corrente.

Pattern Observer

Il pattern observer si usa in un contesto in cui un cambiamento in un oggetto richiede, a volte, un aggiornamento di altri oggetti (gli observer). Questa relazione può essere programmata nell'oggetto che cambia, ma ciò richiede la conoscenza di come gli observer dovrebbero essere aggiornati: gli oggetti si intrecciano e non possono essere facilmente riutilizzati. La soluzione è creare una relazione uno-a-molti tra un oggetto e gli altri

che dipendono da esso: la modifica di un oggetto causerà la ricezione di una notifica degli altri, facendo sì che si aggiornino da soli.

Pattern MVC: Model – View – Controller



Il pattern MVC permette di suddividere un'applicazione, o anche la sola interfaccia dell'applicazione, in tre parti: il model, che si occupa dell'elaborazione e dello stato, la view, che si occupa dell'output e il controller, che si occupa dell'input.
Il model:

- gestisce un insieme di dati logicamente correlati;
- risponde alle interrogazioni sui dati;
- risponde alle istruzioni di modifica dello stato;
- genera un evento quando lo stato cambia;
- registra, in forma anonima, gli oggetti interessati alla notifica dell'evento;
- in Java, deve estendere la classe java.util.Observable.

La view:

- gestisce un'area di visualizzazione, nella quale presenta all'utente una vista dei dati gestiti dal model: mappa i dati del model, o una parte, in oggetti visuali e visualizza tali oggetti su un particolare dispositivo di output;
- si registra presso il model per ricevere l'evento di cambiamento di stato;
- in Java, deve implementare l'interfaccia java.util.Observer.

Il controller:

- gestisce gli input dell'utente (mouse, tastiera, etc.);
- mappa le azioni dell'utente in comandi;
- invia tali comandi al model e/o alla view, che effettuano le operazioni appropriate;
- in Java, è un *listener*.

Esiste anche il pattern MVP, Model – View – Presenter, molto simile all'MVC ma con la differenza che la view è passiva.

Pattern Flyweight

Descrive come condividere oggetti "leggeri", cioè a granularità molto fine, in modo tale che il loro uso non sia mai troppo costoso. Un *flyweight* è un oggetto condiviso che può essere utilizzato simultaneamente ed efficientemente da più clienti del tutto indipendenti tra loro. Benché condiviso, non deve essere distinguibile da un oggetto non condiviso e non deve fare ipotesi sul contesto nel quale opera. Per assicurare una corretta condivisione, i clienti non devono istanziare direttamente i flyweight ma devono ottenerli esclusivamente tramite una FlyweightFactory. È presente una distinzione tra stato intrinseco e stato estrinseco; lo stato intrinseco:

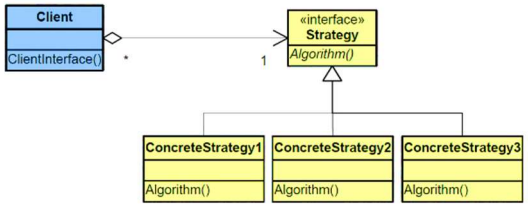
- non dipende dal contesto di utilizzo e quindi può essere condiviso da tutti i clienti;
- è memorizzato nel flyweight.

Lo stato estrinseco:

- dipende dal contesto di utilizzo e quindi non può essere condiviso dai clienti;
- è memorizzato nel cliente o calcolato da esso;
- viene passato al flyweight quando viene invocata una sua operazione.

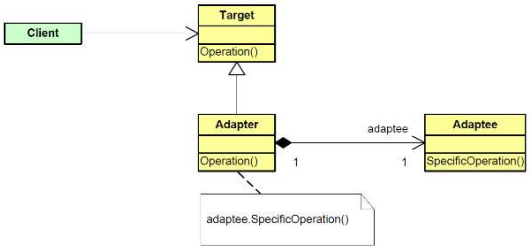
Pattern Strategy

Questo pattern permette di definire un insieme di algoritmi tra loro correlati, incapsulare tali algoritmi in una gerarchia di classi e rendere gli algoritmi intercambiabili. In questo modo è facilmente possibile cambiare gli algoritmi di un'applicazione.



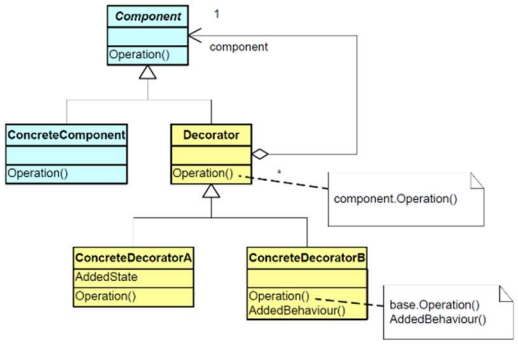
Pattern Adapter

Questo pattern converte l'interfaccia originale di una classe nell'interfaccia (diversa) che si aspetta il cliente e permette a classi che hanno interfacce compatibili di lavorare insieme. Si usa quando si vuole riutilizzare una classe esistente e la sua interfaccia non è conforme a quella desiderata. Questo pattern è noto anche come *wrapper*.



Pattern Decorator

Questo pattern permette di aggiungere responsabilità a un oggetto in modo dinamico. Fornisce un'alternativa flessibile alla specializzazione, infatti, in alcuni casi, le estensioni possibili sono talmente tante che per poter supportare ogni possibile combinazione si dovrebbe definire un numero troppo elevato di sottoclassi. Sulla base del diagramma a destra:



- **Component** (interfaccia o classe astratta) dichiara l'interfaccia di tutti gli oggetti ai quali deve essere possibile aggiungere dinamicamente responsabilità;
- **ConcreteComponent** definisce un tipo di oggetto al quale deve essere possibile aggiungere dinamicamente responsabilità;
- **Decorator** (classe astratta) mantiene un riferimento a un oggetto di tipo Component e definisce un'interfaccia conforme all'interfaccia di Component;
- **ConcreteDecorator** aggiunge responsabilità al componente referenziato.

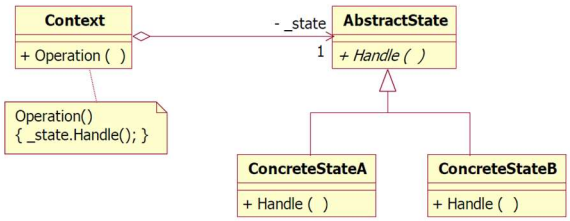
Ereditarietà dinamica

Una sottoclasse deve sempre essere una versione più specializzata della sua superclasse o classe base. Un buon test sul corretto utilizzo dell'ereditarietà è che sia valido il principio di sostituibilità di Liskov. Affinché questo principio sia valido, è necessario che le pre-condizioni di tutti i metodi della sotto-classe siano uguali o più deboli, che le post-condizioni di tutti i metodi della sotto-classe siano uguali o più forti e che ogni metodo ridefinito nella sotto-classe mantenga la semantica del metodo originale.
Un oggetto può cambiare comportamento al cambiare del suo stato in due modi:

- cambiando la classe dell'oggetto run-time: nella maggior parte dei linguaggi di programmazione a oggetti questo non è possibile e inoltre è meglio che un oggetto non possa cambiare la sua classe durante la sua esistenza, infatti la classe di un oggetto deve basarsi sulla sua essenza e non sul suo stato;
- utilizzando il pattern State che usa un meccanismo di delega, grazie al quale l'oggetto è in grado di comportarsi come se avesse cambiato classe.

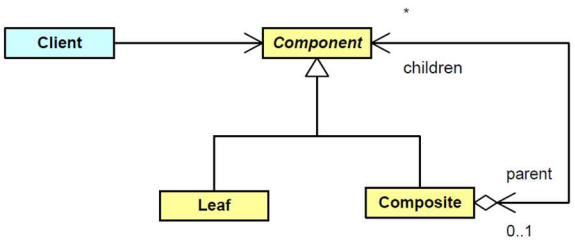
Pattern State

Questo pattern localizza il comportamento specifico di uno stato e suddivide il comportamento in funzione dello stato. Le classi concrete contengono la logica di transizione da uno stato all'altro e il pattern permette anche di emulare l'ereditarietà multipla.



Pattern Composite

Questo pattern permette di comporre oggetti in una struttura ad albero, al fine di rappresentare una gerarchia di oggetti contenitori - oggetti contenuti. Permette ai clienti di trattare in modo uniforme oggetti singoli e oggetti compositi. Sulla base del diagramma a destra:



- **Component** (classe astratta) dichiara l'interfaccia e realizza il comportamento di default;
- **Client** accede e manipola gli oggetti della composizione attraverso l'interfaccia di Component;
- **Leaf** descrive oggetti che non possono avere figli-foglie e definisce il comportamento di tali oggetti;
- **Composite** descrive oggetti che possono avere figli-contenitori e definisce il comportamento di tali oggetti.

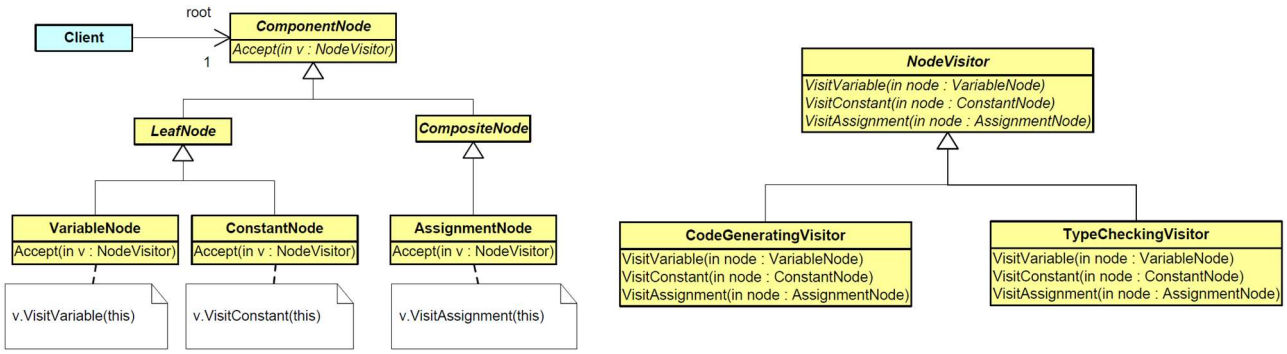
Il contenitore dei figli deve essere un attributo di Composite e può essere di qualsiasi tipo (array, lista, albero, tabella hash, etc.). È presente un riferimento esplicito al genitore (parent): questo semplifica l'attraversamento e la gestione della struttura, ma l'attributo che contiene il riferimento al genitore e la relativa gestione devono essere posti nella classe Component. Tutti gli elementi che hanno come genitore lo stesso componente devono essere gli unici figli di quel componente: è necessario incapsulare l'assegnamento di parent nei metodi *Add* e *Remove* della classe Composite oppure incapsulare le operazioni di *Add* e *Remove* nella *set* dell'attributo parent della classe Component.

Gli obiettivi del pattern composite sono:

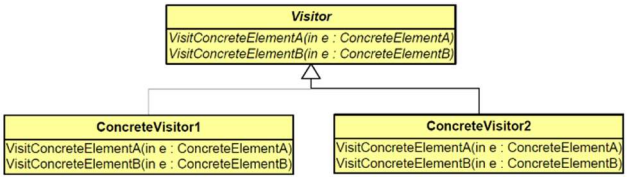
- **massimizzazione dell'interfaccia Component**: si fa in modo che il cliente veda solo l'interfaccia di Component. In Component devono essere inserite tutte le operazioni che devono essere utilizzate dai clienti e nella maggior parte dei casi, Component definisce una realizzazione di default che le sottoclassi devono ridefinire (alcune di queste operazioni possono essere prive di significato per gli oggetti foglia, come *Add*, *Remove*, etc.);
- **trasparenza**: dichiarando tutto al livello più alto, il cliente può trattare gli oggetti in modo uniforme. Il cliente potrebbe anche cercare di fare cose senza senso però, come aggiungere figli alle foglie: per ovviare a ciò *Add* e *Remove* devono avere una realizzazione di default che genera un'eccezione e bisogna disporre di un modo per verificare se è possibile aggiungere figli all'oggetto su cui si vuole agire;
- **sicurezza**: tutte le operazioni sui figli vengono messe in Composite. A questo punto, qualsiasi invocazione sulle foglie genera un errore in fase di compilazione ma il cliente deve conoscere e gestire due interfacce differenti. Nel caso si scelga come obiettivo principale la sicurezza, occorre disporre un modo per verificare se l'oggetto su cui si vuole agire è un Composite.

Pattern Visitor

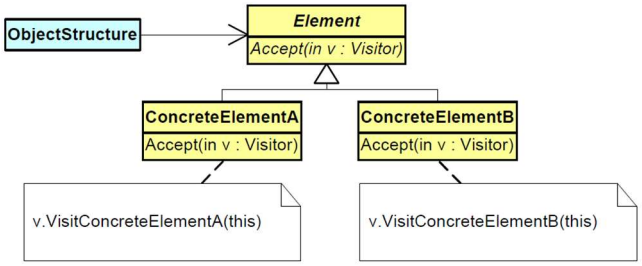
Permette di definire una nuova operazione da effettuare sugli elementi di una struttura senza dover modificare le classi degli elementi coinvolti. Tutto il codice relativo ad un singolo tipo di operazione viene raccolto in una singola classe e per aggiungere un nuovo tipo di operazione è sufficiente progettare una nuova classe. Il visitor deve dichiarare un'operazione per ogni tipo di nodo concreto e ogni nodo deve dichiarare un'operazione per accettare un generico visitor.



Il **Visitor**, una classe astratta o un'interfaccia, dichiara un metodo *Visit* per ogni classe di elementi concreti mentre ConcreteVisitor definisce tutti i metodi *Visit* e globalmente definisce l'operazione da effettuare sulla struttura e, se ecessario, ha un proprio stato.



Element, una classe astratta o un'interfaccia, dichiara un metodo *Accept* che accetta un Visitor come argomento e **ConcreteElement** definisce il metodo *Accept*. **ObjectStructure** può essere realizzata come Composite o come una normale collezione (array, lista, etc.); deve poter enumerare i suoi elementi e deve dichiarare un'interfaccia che permetta a un cliente di far visitare la struttura a un Visitor.



Nel complesso, il pattern visitor facilita l'aggiunta di nuove operazioni. È possibile aggiungere nuove operazioni su una struttura esistente semplicemente aggiungendo un nuovo visitor concreto; senza il pattern visitor sarebbe necessario aggiungere un metodo ad ogni classe degli elementi della struttura. Ogni Visitor concreto raggruppa i metodi necessari per eseguire una data operazione e nasconde i dettagli di come tale operazione debba essere eseguita.

Si fa uso dell'incapsulamento, infatti ogni Visitor deve essere in grado di accedere allo stato degli elementi su cui deve operare; è difficile aggiungere una nuova classe ConcreteElement in quanto per ogni nuova classe ConcreteElement è necessario inserire un nuovo metodo *Visit* in tutti i Visitor esistenti: la gerarchia Element deve essere poco o per nulla modificabile, ovvero deve essere stabile.

Non è necessario che tutti gli elementi da visitare derivino da una classe comune e durante l'operazione ogni Visitor può modificare il proprio stato (ad esempio per accumulare dei valori o altro). *Accept* è un'operazione di tipo double dispatch, ovvero dipende dal tipo di due oggetti: il visitor e l'elemento.

1.14. Dalla progettazione all'implementazione

Quando si arriva alla fase di implementazione, apparentemente occorre soltanto implementare tutte le classi che sono state individuate nelle varie fasi (analisi, progettazione, etc.). Tuttavia, anche a questo livello occorre effettuare delle scelte progettuali che hanno un'impatto sulle caratteristiche del software, come ad esempio efficienza, riusabilità, etc. Il progetto di dettaglio rappresenta una descrizione del sistema molto vicina alla codifica, ovvero che la vincola in maniera sostanziale (ad esempio descrivendo non solo le classi in astratto ma anche i loro attributi e metodi, con relativi tipi e firma). Durante la progettazione di dettaglio è necessario definire:

- i tipi di dato che non sono stati definiti nel modello OOA;
- la navigabilità delle associazioni tra classi e le relative implementazioni;
- le strutture dati necessarie per l'implementazione del sistema;
- le operazioni necessarie per l'implementazione del sistema;
- gli algoritmi che implementano le operazioni;
- la visiblità di classi, attributi, operazioni, etc.

La navigabilità di un'associazione specifica la possibilità di spostarsi da un qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione (a seconda della molteplicità). I messaggi possono essere inviati solo nella direzione della freccia. A livello di analisi del problema, le associazioni di composizione e di aggregazione hanno una direzione precisa, ovvero detti A il contenitore e B l'oggetto contenuto, è A che contiene B e non viceversa; a livello implementativo, un'associazione può essere mono-direzionale quando da A si deve poter accedere a B, ma non viceversa, e bi-direzionale quando da A si deve poter accedere a B e da B si deve poter accedere velocemente ad A. Dal punto di vista implementativo, la bi-direzionalità è molto efficiente ma occorre tenere sotto controllo la consistenza delle strutture dati utilizzate per la sua implementazione.

Per implementare le associazioni con molteplicità 0..1 o 1..1 occorre aggiungere alla classe cliente un attributo membro che rappresenta il riferimento all'oggetto della classe fornitore e/o l'identificatore univoco dell'oggetto della classe fornitore, solo se persistente, o il valore dell'oggetto della classe fornitore (solo nel caso di composizione e molteplicità 1..1).

Per implementare le associazioni con molteplicità 0..* o 1..* occorre aggiungere alla classe cliente un attributo membro che referenzia un'istanza di una classe contenitore, ovvero una classe le cui istanze sono collezioni di riferimenti a oggetti della classe fornitore. La classe contenitore può essere realizzata oppure presa da una libreria.

Una **classe contenitore** (o semplicemente un contenitore) è una classe le cui istanze contengono oggetti di altre classi. Se gli oggetti contenuti sono in numero fisso, è sufficiente un vettore predefinito del linguaggio, se invece gli oggetti contenuti sono in numero variabile, un vettore predefinito non basta e occorre una classe contenitore. Le funzionalità minime di una classe contenitore devono essere inserire, rimuovere, trovare un oggetto in una collezione ed enumerare (iterare su) gli oggetti della collezione. I contenitori possono essere classificati in funzione del modo in cui contengono gli oggetti (contenimento per riferimento, quindi gli oggetti sono reference type, oppure contenimento per valore, in cui gli oggetti sono value type) o dell'omogeneità o eterogeneità di tali oggetti (quindi gli oggetti sono omogenei, ovvero tutti dello stesso tipo, o eterogenei, ovvero gli oggetti possono essere di tipo diverso).

Nel **contenimento per riferimento**:

- l'oggetto contenuto esiste per conto proprio;
- l'oggetto contenuto può essere in più contenitori contemporaneamente;
- quando un oggetto viene inserito in un contenitore, non viene duplicato ma ne viene memorizzato solo il riferimento;
- la distruzione del contenitore non comporta la distruzione degli oggetti contenuti.

Nel **contenimento per valore**:

- l'oggetto contenuto viene memorizzato nella struttura dati del contenitore ed esiste solo in quanto contenuto fisicamente in un altro oggetto;
- quando un oggetto deve essere inserito in un contenitore, viene duplicato;
- la distruzione del contenitore comporta la distruzione degli oggetti contenuti.

Per il **contenimento di oggetti omogenei**:

- i contenitori ideali sono le classi generiche;
- il tipo degli oggetti contenuti viene lasciato generico e ci si concentra sugli algoritmi di gestione della collezione di oggetti;
- quando serve una classe contenitore di oggetti appartenenti a una classe specifica, è sufficiente istanziare la classe generica, specificando il tipo desiderato.

Per il **contenimento di oggetti eterogenei**:

- per implementare i contenitori (solo per riferimento) è necessario usare l'ereditarietà e sfruttare la proprietà che un puntatore alla superclasse radice della gerarchia può puntare a un'istanza di una qualunque sottoclasse;
- la classe contenitore può essere generica, ma il tipo deve essere la superclasse radice della gerarchia (nei peggiori dei casi, *object*).

Un modo alternativo per implementare un'associazione tra due oggetti è tramite un **dizionario**, ovvero un particolare tipo di contenitore che associa due oggetti: la chiave e il rispettivo valore. La chiave può essere un oggetto qualsiasi, non necessariamente una stringa o un intero, ma deve essere unica. Il dizionario, data una chiave, ritrova in modo efficiente il valore ad essa associato.

Un oggetto, contenitore o meno, può contenere un riferimento univoco ad un altro oggetto. Nel caso di strutture dati interamente contenute nello spazio di indirizzamento dell'applicazione, un oggetto può essere identificato univocamente mediante il suo indirizzo logico di memoria. Nel caso di database o di sistemi distribuiti, ad ogni oggetto deve essere associato un identificatore univoco persistente tramite il quale deve essere possibile risalire all'oggetto stesso, sia che esso risieda in memoria, sia che risieda in disco o in rete. L'identificatore univoco è un attributo che al momento della creazione dell'oggetto viene inizializzato con un valore generato automaticamente dal sistema, il valore della chiave primaria di una tabella relazionale, etc.

Se esistono strutture con ereditarietà multipla e se il linguaggio di programmazione non la ammette, è necessario convertire le strutture in strutture con solo ereditarietà semplice. Una prima possibilità è usare la composizione-delega: si sceglie la più significativa tra le superclassi e si eredita esclusivamente da questa; tutte le altre superclassi diventano possibili "ruoli" e vengono connesse mediante composizione: le caratteristiche delle superclassi escluse vengono incorporate nella classe specializzata tramite composizione-delega e non tramite ereditarietà. La seconda possibilità è appiattire tutto in una gerarchia semplice e implementare un'interfaccia: in questo modo, una o più relazioni di ereditarietà si perdono e gli attributi e le operazioni corrispondenti devono essere ripetuti nelle classi specializzate.

Per quanto riguarda la velocità del software, il software con le prestazioni migliori fa la cosa giusta abbastanza velocemente, ovvero soddisfa i requisiti e/o le attese del cliente, pur rimanendo entro costi e tempi preventivati. Per migliorare la velocità percepita può bastare memorizzare risultati intermedi e fare un'accurata progettazione dell'interazione con l'utente (ad esempio usando multi-threading). Un trafico di messaggi molto elevato tra oggetti può invece richiedere dei cambiamenti per aumentare la velocità. Di norma, la soluzione è che un oggetto possa accedere direttamente ai valori di un altro oggetto (aggirando l'incapsulamento). Questo tipo di modifica deve essere presa in considerazione solo dopo che tutti gli altri aspetti del progetto sono stati soggetti a misure e modifiche; l'unico modo per sapere se una modifica contribuirà in modo significativo a rendere il software abbastanza veloce è tramite le misure e l'osservazione.

1.15. Sistemi di controllo delle versioni

Il controllo delle versioni serve per gestire le modifiche a documenti, programmi, grandi siti web e altre collezioni di informazioni. La necessità di un modo logico per organizzare e controllare le revisioni è esistita quasi da quando esiste la scrittura, ma il controllo delle versioni è diventato molto più importante e complicato quando è iniziata l'era dell'informatica. Tutti i software hanno molteplici versioni: diverse release di un prodotto, variazioni in base a differenti piattaforme hardware e software, versioni alpha, beta e ogni volta che si modifica un programma. Il controllo della versioni permette di tracciare le versioni in modo tale da permettere di ripristinare vecchie versioni e far coesistere molteplici versioni. Oltre a questi, ulteriori benefici sono l'avere vecchie e molteplici versioni di un software, il poter tracciare la sua storia e avere dei backup.
Una prima soluzione possibile è una soluzione casalinga: gli sviluppatori salvano molteplici copie delle differenti versioni di un programma e le etichetta in modo appropriato: questo è un modo inefficiente, in quanto si dovrebbero mantenere molte copie quasi uguali tra di loro e inoltre richiede una opportuna gestione dei permessi di lettura, scrittura ed esecuzione: questo aggiunge pressione e sforzo a chi deve gestire i permessi in quanto deve garantire che il codice non sia compromesso. Un sistema di controllo delle versioni:

- supporta la memorizzazione del codice sorgente;
- fornisce una cronologia delle modifiche;
- fornisce un modo per lavorare in parallelo su aspetti differenti del software;
- fornisce un modo per lavorare in parallelo senza interferenze;
- fornisce un modello di sviluppo;
- incrementa la produttività.

Nei sistemi di controllo delle versioni sono presenti alcuni concetti fondamentali. Il primo è quello dei **progetti**, ovvero un insieme di file nel controllo delle versioni. Nel controllo delle versioni è indifferente il tipo di file, questo infatti può essere sia un file di compilazione che un test, semplicemente si gestiscono le versioni di una collezione di file. Un altro concetto è quello di **repository**, ovvero il posto in cui vengono salvati i file correnti e la cronologia. Di solito si trova su un server remoto, su una macchina affidabile e sicura; tutti gli utenti condividono la stessa repository e a volte questa si chiama anche *depot*. Un ulteriore concetto fondamentale è quello di **copia di lavoro**, ovvero la copia locale di una repository in un momento specifico. Il nome deriva dal fatto che tutto il lavoro svolto sui file di una repository si fa inizialmente su una copia di lavoro. Ogni sviluppatore ne ha una sulla sua macchina. Concettualmente, è una sandbox: un ambiente che isola le modifiche al codice non testato e la sperimentazione diretta dall'ambiente di produzione. L'atto di copiare il contenuto di una repository in una cartella di lavoro è detto *check out*; al termine, si può fare una commit delle modifiche, ovvero un *check in*. Il workflow da seguire è: copiare i file dalla repository alla propria cartella di lavoro, aggiornare il codice nella propria cartella di lavoro, aggiornare la repository dalla propria cartella di lavoro (controllando prima che il codice sia corretto) e ripetere.
Fare un check out, modificare un file e fare il check in comporta la creazione di una nuova versione di un file: questo procedimento è detto **revisione**. La maggior parte delle modifiche sono piccole e, di solito, per motivi di efficienza, non si salva un nuovo file per intero ma si salvano le differenze rispetto alla versione precedente. In questo modo si minimizza lo spazio necessario e si evita di rendere i procedimenti di check out e check in più lenti. Di ciascuna revisione, il sistema memorizza le differenze che ci sono in quella revisione, il numero della versione (ad esempio si è passati dalla 1.5 alla 1.6) e altri metadati, come l'autore, l'orario a cui è stato fatto il check in, etc.
L'ultimo concetto fondamentale è quello di **branch**, ovvero una diramazione. Un branch sono semplicemente due revisioni di un file. Il check in normale non crea una diramazione, è necessaria una richiesta esplicita per crearne una.

Modello LMU, *Lock-Modify-Unlock*

In questo modello la repository consente a una sola persona per volta di modificare un file. Ogni volta che qualcuno vuole modificare un file, lo deve prima bloccare; in questo modo, se un file è bloccato, nessun altro lo può modificare e una volta terminata la modifica l'utente che lo stava modificando lo sblocca.
Il blocco può causare problemi amministrativi, infatti ogni tanto può capitare che un utente blocchi un file ma si dimentichi di sbloccarlo: se un altro utente vuole modificare quel file, non potrà farlo e dovrà aspettare inutilmente; se ad esempio l'utente che ha bloccato il file si è dimenticato di sbloccarlo e non è più reperibile, l'altro utente che vuole lavorarci sopra dovrà contattare un amministratore per farsi sbloccare il file e questo causa molti ritardi e perdite di tempo. Il blocco può anche causare una serializzazione non necessaria, ad esempio un utente A potrebbe voler modificare l'inizio di un file e un utente B potrebbe voler modificare la fine dello stesso file: queste modifiche non si sovrappongono affatto e si potrebbero tranquillamente fare in contemporanea senza la necessità di alternarsi. Un ulteriore problema che si può causare è quello di avere un falso senso di sicurezza: si suppong che un primo utente modifichi il file A e un secondo utente modifichi contemporaneamente un file B; se i file A e B hanno delle dipendenze tra di loro, le modifiche fatte a questi due file potrebbero creare incompatibilità e all'improvviso i file A e B non sono più adatti per lavorare insieme: il meccanismo di blocco non può prevenire questo problema. L'ultimo ovvio problema è che il modello LMU non permette di lavorare offline.

Modello CMM, *Copy-Modify-Merge*

In questo modello non esiste il concetto di blocco: ogni volta che qualcuno fa il check in di una copia di lavoro, le sue modifiche vengono unite (*merge*) a quelle fatte da altri utenti. Tramite questo sistema di merging, ci possono essere due possibili risultati in seguito alla modifica di una versione: un successo o un conflitto. Si ha conflitto quando due programmatori modificano lo stesso pezzo di codice e questi si sovrappongono: il sistema non sa come gestire la situazione e quindi dà come risultato un conflitto. Nel caso di conflitti, il sistema non può apportanre modifiche in quanto il risultato finale non è unico e dipende dall'ordine in cui vengono fatte le modifiche. Di solito il controllo delle versioni mostra la presenza di un conflitto che va però risolto manualmente. Il rilevamento dei conflitti si basa sulla distanza delle modifiche: modifiche alla stessa riga daranno un conflitto, modifiche a linee diverse probabilmente non creeranno conflitti. Tuttavia, la mancanza di conflitti non indica per forza che due modifiche funzionino insieme.

Confronto tra LMU e CMM

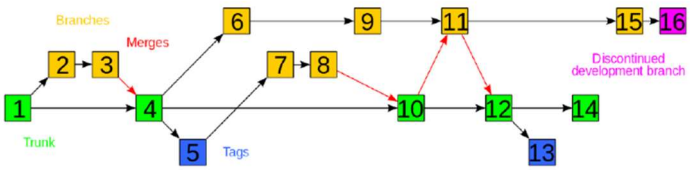
Nel modello LMU il principale vantaggio è quello che si può lavorare in modo comodo quando i file non sono unibili, come ad esempio con le immagini grafiche; nel caso del modello CMM i vantaggi, invece, sono:

- funziona in modo estremamente fluido anche se può sembrare un modello caotico;
- gli utenti possono lavorare in parallelo senza mai aspettarsi l'un l'altro;
- la maggior parte delle modifiche fatte in contemporanea di solito non si sovrappone, infatti i conflitti sono poco frequenti;
- il tempo necessario per risolvere i conflitti è molto inferiore al tempo perso causato da un sistema con blocco.

DVCS e CVCS

A differenza del CVCS, *Centralized Version Control* System, il DVCS, *Distributed Version Control System*, permette la sincronizzazione scambiato patch, ovvero set di modifiche, da peer a peer. Non esiste una copia canonica, di riferimento, ma solo copie di lavoro (anche se è possibile creare una repository "ufficiale" centrale). Le operazioni comuni (come commit, visualizzazione della cronologia e ripristino delle modifiche) sono veloci in quanto non è necessario comunicare con un server centrale. Ogni copia funzionante funziona efficicacemente come backup remoto del codice e della sua cronologia delle modifiche.

Le revisioni si possono generalmente vedere come una linea di sviluppo, un “albero”, con rami che fuoriscono, formando una o più linee dritte che proseguono in parallelo. In realtà la struttura è più complicata e si forma un grafico diretto aciclico, ma per molti scopi un albero con tanti merge è un’approssimazione adeguata. Alcuni concetti di DVCS sono:



- **trunk**: è la linea principale di sviluppo;
- **branch**: è un “ramo”, ovvero una diramazione. Un set di file di una certa versione possono essere diramati in un determinato momento in modo che, da quel momento in poi, due copie di tali file possano svilupparsi a velocità differenti o in modi diversi indipendentemente l’una dall’altra. È una copia completa del codice base, inclusa la sua cronologia, e quando gli aggiornamenti sono consolidati si può riunire al trunk principale;
- **tag**: anche detto etichetta, si riferisce a un’istantanea importante nel tempo, coerente con molti file. Questi file in quel preciso istante possono essere tutti etichettati con un nome o un numero di versione intuitivi e significativi.
- **push/pull**: si tratta di trasferire copie revisionate da una repository all’altra. Il push è avviato dalla sorgente, il pull è avviato dalla repository. In genere, un utente invia una richiesta pull e un manutentore deve fare il merge della richiesta.

Affinché gli altri possano vedere le modifiche, devono accadere quattro cose:

- una commit;
- un push;
- un pull eseguito da chi vuole vedere le modifiche;
- un aggiornamento da parte di chi vuole vedere le modifiche.

I comandi di commit e aggiornamento spostano soltanto le modifiche dalla copia di lavoro alla repository locale; al contrario, i comandi di push e pull effettuano modifiche tra la repository locale e le altre.
Le DVCS si concentrano in particolare sul condividere modifiche: ogni modifica ha un GUID per permettere un facile tracciamento di tutto ciò si fa. Delle buone pratiche sono:

- quando si fa una commit, scrivere in modo chiaro cosa si è fatto;
- rendere ogni commit un’unità logica;
- evitare di fare commit indiscriminate;
- incorporare frequentemente modifiche fatte da altri in modo da rendere più rari i conflitti;
- condividere spesso le modifiche, sempre per evitare conflitti;
- coordinarsi con i colleghi;
- non fare commit di file generati.

I pro dei DVCS sono:

- tutti hanno una sandbox locale;
- funziona offline;
- è rapido: quasi tutto avviene localmente;
- creare nuovi branch e fare merge è facile, in quanto i GUID rendono tutto più facile;
- è necessaria meno gestione.

I contro sono:

- c’è ancora bisogno di un backup;
- non esiste realmente un’ultima versione;
- non ci sono realmente numeri di revisione, infatti i GUID non sono numeri di release, ma si possono sempre usare i tag.

Git

Un esempio di DVCS è Git. I concetti fondamentali sono quelli di **directory**, ovvero la repository principale e centrale; **directory di lavoro**, ovvero una copia locale di una repository, e l'**area di sosta**, ovvero un file (un indice) che specifica quali file modificati devono essere salvati (committed) nella repository. In Git un file può essere **committed**, ovvero è salvato in una repository; **modificato**, ovvero è stato modificato ma non ancora committed, e **in sosta**, ovvero è stato segnato per essere committed presto.

Il workflow in Git è:

- fare il check out del progetto dalla directory di Git alla propria directory di lavoro;
- modificare i file: le modifiche restano all'interno della directory di lavoro;
- mettere i file in sosta, ovvero selezionare soltanto i file di cui si vuole fare presto una commit;
- fare una commit: salvare permanentemente dei file.

2.2. Modelli, linguaggi e modelli di processo di sviluppo

Per **modello** si intende genericamente una rappresentazione di un oggetto o di un fenomeno reale che riproduce caratteristiche o comportamenti ritenuti fondamentali per il tipo di ricerca che si sta svolgendo. Nei processi di costruzione del software, il termine "modello" va inteso come un insieme di concetti e proprietà volti a catturare aspetti essenziali di un sistema, collocandosi in un preciso spazio concettuale. Per l'ingegnere del software quindi, un modello costituisce una visione semplificata di un sistema che rende il sistema stesso più accessibile alla comprensione e alla valutazione e, inoltre, facilita il trasferimento di informazione e la collaborazione tra persone. Nel processo di produzione del software il lavoro dei diversi attori (analisti, progettisti, sviluppatori, etc.) si basa su un insieme di conoscenze che spesso rimangono implicite all'interno del processo. Uno degli scopi dei processi model-based è rendere esplicite queste conoscenze attraverso la costruzione di diagrammi espressi con notazioni formali, ovvero attraverso l'uso di un linguaggio con sintassi e semantica ben precise. Lo scopo di questi diagrammi è rappresentare modelli del sistema per descrivere in modo conciso e preciso conoscenze sul problema; i modelli del sistema sono inoltre utili per individuare rischi e scelte progettuali. L'uso dei modelli è motivato dalla difficoltà della mente umana di impostare ragionamenti efficaci in presenza di dettagli troppo minuti. I linguaggi per la descrizione dei modelli rappresentano il culmine della continua evoluzione verso livelli di astrazione più elevati rispetto alle macchine che hanno caratterizzato la storia dei linguaggi di programmazione.
L'insieme dei modelli che descrivono un sistema dovrebbe formare una descrizione completa, consistente e non troppo ridondante: la transizione tra un modello e l'altro deve essere continua e i modelli devono essere connessi tra loro in modo sistematico: un elemento in un modello deve avere il suo o i suoi corrispettivi in un altro.
Un concetto fondamentale è quello di **tracciabilità**: in qualsiasi direzione si percorra la sequenza di modelli generati, deve essere possibile mappare uno o più elementi in un modello in uno o più elementi in un altro. Nei modelli software, la tracciabilità serve per:

- garantire coerenza e consistenza tra i modelli;
- creare idealmente un percorso logico che parte dai requisiti e arriva al relativo codice (e viceversa);
- tenere sotto controllo le modifiche.

È spesso molto arduo poter garantire la tracciabilità.

Linguaggi di modellazione
Un linguaggio di modellazione è un linguaggio (semi-)formale che può essere utilizzato per descrivere, ovvero modellare, un sistema di qualche natura. Nell'ingegneria del software un modello di un sistema software, o di qualche suo aspetto, prende il nome di **modello software**. Quello che si esprime attraverso i diagrammi è quindi una rappresentazione del modello creata attraverso l'uso di un linguaggio; lo stesso modello può essere rappresentato da linguaggi diversi, il cui potere espressivo potrebbe essere differente.
Anche il codice è una rappresentazione del modello espressa in un particolare linguaggio. Il codice:

- è il modello più dettagliato;
- fornisce una visione "piatta";
- non mette in evidenza i punti salienti;
- non aiuta ad avere una visione d'insieme del sistema con un solo colpo d'occhio.

Con le attuali tecnologie a disposizione si ha necessità di creare sia modelli ad alto livello di astrazione, sia il codice. Questo però crea un problema di fondo: i modelli ed il codice sono tipicamente disallineati: il disallineamento, infatti, si crea già durante la fase di implementazione. Alcune modifiche fatte nel codice non vengono quasi mai riflesse nei modelli di progettazione del sistema; tali modelli non offrono più una view coerente e viene meno il requisito di tracciabilità: a questo punto, operare con tecniche di reverse engineering causa solo più mali.
Generare modelli del sistema partendo dal codice produce modelli che presentano tutti i difetti già riscontrati nel codice e fanno perdere l'utilitià dei modelli di progettazione.

L'approccio migliore per mantenere l'allineamento e la tracciabilità è:

- apportare le modifiche direttamente al modello di progettazione;
- ove possibile, generare codice da questo attraverso opportuni generatori di codice;
- negli altri casi, modificare direttamente a mano il codice in modo coerente con il modello.

Dato che il linguaggio naturale è troppo impreciso e il codice è preciso ma troppo dettagliato, la soluzione migliore è usare un linguaggio di modellazione:

- è sufficientemente preciso;
- è flessibile dal punto di vista descrittivo per poter arrivare a un qualunque livello di dettaglio;
- deve essere possibilmente standard.

Esiste ormai una convergenza su un unico linguaggio di modellazione di tipo grafico: UML, *Unified Modeling Language*.

Modelli di processo

Un processo di sviluppo è un insieme ordinato di passi che coinvolge tutte quelle attività, vincoli e risorse necessari per produrre il desiderato output a partire dall'insieme dei requisiti in ingresso. Tipicamente un processo è composto da differenti fasi messe in relazione una con l'altra. Ogni fase identifica una porzione di lavoro che deve essere svolto nel contesto del processo, le risorse che devono essere utilizzate e vincoli a cui si deve obbedire. Ogni fase inoltre può essere composta da più attività che devono essere messe in relazione tra loro.

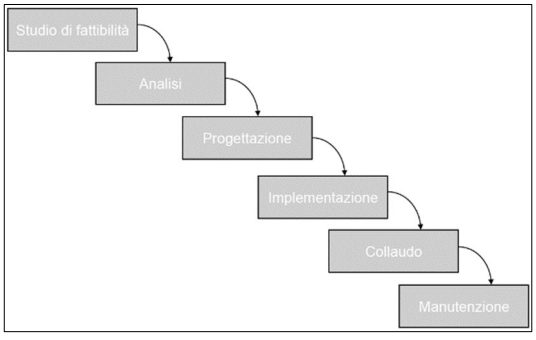
Il processo di sviluppo software è un isieme coerente di politiche, strutture organizzazionali, tecnologie, procedure e deliverable che sono necessari per concepire, sviluppare, installare e manutenere un prodotto software. Le fasi generiche sono:

- **specifica**: cosa il sistema dovrebbe fare e quali sono i vincoli di sviluppo;
- **sviluppo**: produzione del sistema software;
- **validazione**: testare che il sistema sviluppato sia quello che il committente voleva;
- **evoluzione**: cambiamenti nel prodotto in accordo a modifiche dei requisiti o incremento delle funzionalità del sistema.

Un modello di processo software è una rappresentazione semplificata di un processo presentato da una specifica prospettiva. Un modello di processo prescrive le fasi attorno alle quali il processo dovrebbe essere organizzato, l'ordine di tali fasi e l'interazione e la coordinazione del lavoro tra le diverse fasi. In altre parole, un modello di processo definisce un template attorno al quale organizzare e dettagliare un vero processo di sviluppo.

Modello a cascata

Questo modello presenta fasi distinte, in cascata tra loro con retroazione finale. Questo modello si fonda sul presupposto che introdurre cambiamenti sostanziali nel software in fasi avanzate dello sviluppo ha costi troppo elevati pertanto, ogni fase deve essere svolta in maniera esaustiva prima di passare alla successiva, in modo da non generare retroazioni. Le uscite che una fase produce come ingresso per la fase successiva sono i cosiddetti **semilavorati** del processo di sviluppo: documentazione di tipo cartaceo, codice dei singoli moduli, il sistema complessivo.



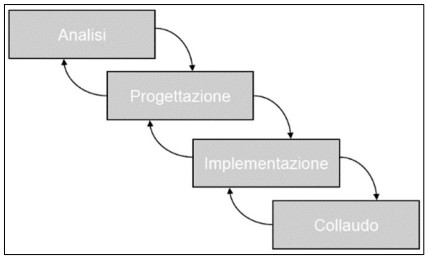
Oltre alle fasi, è fondamentale definire:

- **semilavorati**, al fine di garantire che ci possa essere un'attività di controllo della qualità dei semilavorati;
- **date** entro le quali devono essere prodotti i semilavorati, al fine di certificare l'avanzamento del processo secondo il piano stabilito.

I limiti sono dati dalla sua rigidità, in particolare da due assunti di fondo molto discutibili:

- **immutabilità dell'analisi**: i clienti sono in grado di esprimere esattamente le loro esigenze e, di conseguenza, in fase di analisi iniziale è possibile definire tutte le funzionalità che il software deve realizzare;
- **immutabilità del progetto**: è possibile progettare l'intero sistema prima di aver scritto una sola riga di codice.

Nella realtà, la visione che i clienti hanno del sistema evolve man mano che il sistema prende forma e quindi le specifiche cambiano in continuazione; nel campo della progettazione, le idee migliori vengono in mente per ultime, quando si comincia a vedere qualcosa di concreto e inoltre i problemi di prestazioni costringono spesso a rivedere le scelte di progetto. Evoluzioni successive al modello originale ammettono forme limitate di retroazione a un livello (figura a destra).



Per evitare problemi, prima di iniziare a lavorare sul sistema vero e proprio è meglio realizzare un prototipo in modo da fornire agli utenti una base concreta per meglio definire le specifiche. Una volta esaurito il compito, il prototipo viene abbandonato e si procede a costruirsi il sistema reale secondo i canoni del modello a cascata. Questo approccio risulta però quasi sempre così dispendioso da annullare i vantaggi economici che il modello a cascata dovrebbe garantire.

Prototipo

È un modello approssimato dell'applicazione. L'obiettivo è quello di essere mostrato al cliente, o farlo usare a questi, al fine di ottenere un'indicazione su quanto il prototipo colga i reali fabbisogni. Deve essere sviluppabile in tempi brevi con costi minimi. È un'alternativa interessante in tutti i casi in cui lo sviluppo dell'applicazione parta inizialmente con requisiti non perfettamente noti o instabili. L'obiettivo è comprendere meglio le richieste del cliente e quindi migliorare la definizione dei requisiti del sistema; il prototipo si concentra sulle parti che sono mal comprese con l'obiettivo di contribuire a chiarire i requisiti.

Si parla di **programmazione esplorativa**: il prototipo si trasforma progressivamente nel prodotto finale con l'obiettivo di lavorare a stretto contatto con il cliente per chiarire completamente i requisiti. Prima si sviluppano le parti del sistema che sono ben chiare (requisiti ben compresi) e poi si aggiungono nuove parti/funzionalità come proposto dal cliente.

Modelli evolutivi

Partendo da specifiche molto astratte, si sviluppa un primo prototipo da sottoporre al cliente e da raffinare successivamente. Esistono diversi modelli di tipo evolutivo, ma tutti in sostanza propongono un ciclo di sviluppo in cui un prototipo iniziale evolve gradualmente verso il prodotto finito attraverso un certo numero di iterazioni. Il vantaggio fondamentale è che ad ogni iterazione è possibile confrontarsi con gli utenti per quanto riguarda le specifiche e le funzionalità e inoltre si possono rivedere le scelte di progetto.

I modelli evolutivi si sono orientati verso cicli sempre più brevi e iterazioni sempre più veloci, fino ad arrivare al modello più radicale che prende il nome di *Extreme Programming*.

Nell'Extreme Programming, si ha:

- **comunicazione** tra sviluppatori e tra sviluppatori e clienti;
- **testing**: più codice per il test che per il programma vero e proprio;
- **semplicità**: il codice deve essere il più semplice possibile. Complicare il codice tentando di prevedere futuri riutilizzi è controproducente sia come qualità di codice prodotto, sia come tempo necessario. D'altra parte, il codice semplice e comprensibile è il più riutilizzabile;
- **coraggio**: non si deve aver paura di modificare il sistema che deve essere ristrutturato in continuazione, ogni volta che si intravede un possibile miglioramento.

Questo modello inoltre si può applicare soltanto a sistemi di piccole dimensioni, sistemi che avranno breve durata e a parti di sistemi più grandi.

I problemi dei modelli evolutivi sono:

- il processo di sviluppo non è visibile (ad esempio non è disponibile la documentazione);
- il sistema sviluppato è poco strutturato (si fanno modifiche frequenti);
- è richiesta una particolare abilità nella programmazione.

Modelli ibridi

Sono sistemi composti da sotto-sistemi. Per ogni sotto-sistema è possibile adottare un diverso modello di sviluppo: il modello evolutivo per sotto-sistemi con specifiche ad alto rischio, il modello a cascata per sotto-sistemi con specifiche ben definite. Di norma conviene creare e raffinare prototipi funzionanti dell'intero sistema o di sue parti secondo l'approccio incrementale-iterativo.

Sviluppo incrementale

In questo sviluppo si costruisce il sistema sviluppandone sistematicamente e in sequenza parti ben definite. Una volta costruita una parte, essa non viene più modificata; è di fondamentale importanza essere in grado di specificare perfettamente i requisiti della parte da costruire prima della sua implementazione.

Sviluppo iterativo

In questo sviluppo si effettuano molti passi dell'intero ciclo di sviluppo del software per costruire iterativamente tutto il sistema aumentandone ogni volta il livello di dettaglio. Questo tipo di sviluppo non funziona bene per progetti significativi.

Sviluppo incrementale-iterativo

In questo modello ibrido:

- si individuano sotto-parti relativamente autonome;
- si realizza il prototipo di una di esse;
- si continua con altre parti;
- si aumenta progressivamente l'estensione e il dettaglio dei prototipi, tenendo conto delle altre parti interagenti;
- così via.

RUP – *Rational Unified Process*

RUP, un'estensione dello *Unified Process*, è un modello di processo software iterativo. È un modello ibrido: contiene elementi di tutti i modelli di processo generici. Non definisce un singolo specifico processo, bensì un framework adattabile che può dar luogo a diversi processi in diversi contesti (per esempio, in diverse organizzazioni o nel contesto di progetti con diverse caratteristiche). Questo modello è pensato soprattutto per progetti di grandi dimensioni.

RUP individua tre diverse visioni del processo di sviluppo:

- una prospettiva dinamica che mostra le fasi del modello nel tempo;
- una prospettiva statica che mostra le attività del processo coinvolte;
- una prospettiva pratica che suggerisce le buone prassi da seguire durante il processo.

La **prospettiva dinamica** è composta da più fasi. La prima, quella di **avvio**, ha lo scopo di delineare nel modo più accurato possibile il business case, ovvero:

- comprendere il tipo di mercato al quale il progetto afferisce e identificare gli elementi importanti affinché esso conduca a un successo commerciale;
- identificare tutte le entità esterne (persone e sistemi) che interagiranno con il sistema e definire tali interazioni;
- fra gli strumenti utilizzati ci sono un modello dei casi d'uso, la pianificazione iniziale del progetto, la valutazione dei rischi, una definizione grossolana dei requisiti e così via.

La seconda fase della prospettiva dinamica è l'**elaborazione**: si definisce la struttura complessiva del sistema. Questa fase comprende l'analisi del dominio e una prima fase di progettazione dell'architettura. Devono essere soddisfatti i seguenti criteri:

- modello dei casi d'uso completo all'80%;
- descrizione dell'architettura del sistema;
- sviluppo di un'architettura eseguibile che dimostri il completamento degli use case significativi;
- revisione del business case e dei rischi;
- pianificazione del progetto complessivo.

Se il progetto non soddisfacesse i criteri, potrebbe essere abbandonato oppure rivisitato. Al termine, si transita in una situazione di rischio più elevato, in cui le modifiche del progetto saranno più difficili e dannose.
La fase successiva è quella di **costruzione**: si progetta, programma e testa il sistema. Le diverse parti del sistema vengono sviluppate parallelamente e poi integrate; al termine della fase si dovrebbe avere un sistema software funzionante e la relativa documentazione pronta.
L'ultima fase è quella di **transizione**: il sistema passa dall'ambiente di sviluppo a quello del cliente finale. Vengono condotte le attività di training degli utenti e il beta testing del sistema a scopo di verifica e validazione; si deve, in particolare, verificare che il prodotto sia conforme alle aspettative descritte nella fase di avvio e se questo non è vero si procede a ripetere l'intero ciclo.
La **prospettiva statica** di RUP si concentra sulle attività di produzione del software chiamate workflow. RUP identifica sei workflow principali e tre di supporto; dato che RUP è stato progettato insieme a UML, la descrizione dei workflow è orientata ai modelli associati a UML. Il vantaggio di presentare sia la visione statica che quella dinamica è che le fasi dello sviluppo non sono associate a specifici workflow: tutti i workflow possono essere attivi in ogni stadio del processo. I sei workflow principali sono:

- **modellazione delle attività aziendali**: i processi aziendali sono modellati utilizzando il business case;
- **requisiti**: vengono identificati gli attori che interagiscono con il sistema e sviluppati i casi d'uso per modellare i requisiti;
- **analisi e progetto**: viene creato e documentato un modello di progetto utilizzando modelli architetturali e sequenziali dei componenti e degli oggetti;
- **implementazione**: i componenti del sistema sono implementati e strutturati nell'implementazione dei sotto-sistemi. La generazione automatica del codice a partire dai modelli velocizza questa fase;
- **test**: il test dei sotto-sistemi è un processo iterativo eseguito in parallelo all'implementazione. Il test del sistema finale segue il completamento di tutti gli altri test;
- **rilascio**: viene creata una release del prodotto, viene distribuita agli utenti ed installata nelle loro postazioni di lavoro.

I tre workflow di supporto, invece, sono:

- **gestione della configurazione e delle modifiche**: gestisce i cambiamenti nel sistema;
- **gestione del progetto**: gestisce lo sviluppo del sistema;
- **ambiente**: rende disponibili gli strumenti adeguati al team di sviluppatori.

La **prospettiva pratica** di RUP descrive la buona prassi di ingegneria del software che si raccomanda si usare nello sviluppo dei sistemi. Le pratiche fondamentali sono sei:

- **sviluppare il software ciclicamente**: pianificare gli incrementi del sistema basati sulle proprietà del cliente e sviluppare e consegnare le funzioni con la priorità più alta all'inizio del processo di sviluppo;
- **gestire i requisiti**: documentare esplicitamente i requisiti del cliente e i cambiamenti effettuati. Inoltre, analizzare l'impatto dei cambiamenti sul sistema prima di accettarli;
- **usare architetture basate sui componenti**: strutturare l'architettura del sistema con un approccio a componenti;
- **creare modelli visivi del software**: usare modelli grafici UML per rappresentare le visioni statiche e dinamiche del software;

- **verificare la qualità del software**: assicurarsi che il software raggiunga gli standard di qualità dell'organizzazione;
- **controllare le modifiche al software**: gestire i cambiamenti del software usando un sistema per la gestione delle modifiche, procedure e strumenti di gestione della configurazione.

Scelta del processo di sviluppo ideale

Non esiste un processo di sviluppo ideale. La definizione di un processo non può prescindere dalle caratteristiche dell'organizzazione in cui esso avviene e dalla competenza ed esperienza degli attori coinvolti. Le fasi e le attività del processo di sviluppo potrebbero essere organizzate in modi differenti e descritte a differenti livelli di dettaglio per differenti tipi di sistemi; l'uso di un processo inappropriato riduce la qualità e l'utilità del prodotto software che deve essere sviluppato o migliorato.

2.4. Analisi dei requisiti: sicurezza e privacy

Dal 25 maggio 2018 la GDPR, *General Data Protection Regulation*, sostituisce la DPD, *Data Protection Directive*. Un prodotto software che tratta dati personali è obbligato ad aderire ai principi della GDPR:

- privacy by design e by default (misure tecniche e organizzative);
- minimalità e proporzionalità;
- anonimizzazione e pseudonimizzazione;
- prestare attenzione al trasferimento dei dati al di fuori dell'EU;
- adeguatezza delle misure di sicurezza.

Queste misure non possono essere "aggiunte dopo", a sistema già progettato, ma vanno considerate fin dall'inizio.

La **pseudonimizzazione** è il processo di trattamento dei dati personali in modo tale che i dati non possano più essere attribuiti a un interessato specifico senza l'utilizzo di informazioni aggiuntive, sempre che tali informazioni aggiuntive siano conservate separatamente e soggette a misure tecniche e organizzative intese a garantire la non attribuzione a una persona identificata o identificabile.

I dati personali devono essere:

- trattati in modo lecito, equo e trasparente nei confronti dell'interessato;
- raccolti per finalità determinate, esplicite e legittime, e successivamente trattati in modo non incompatibile con tali finalità;
- adeguati, pertinenti e limitati a quanto necessario rispetto alle finalità per le quali sono trattati;
- esatti e, se necessario, aggiornati; devono essere prese tutte le misure ragionevoli per cancellare o rettificare tempestivamente i dati inesatti rispetto alle finalità per le quali sono trattati;
- conservati in una forma che consenta l'identificazione degli interessati per un arco di tempo non superiore al conseguimento delle finalità per le quali sono trattati; i dati personali possono essere conservati per periodi più lunghi a condizione che siano trattati esclusivamente per finalità di archiviazione nel pubblico interesse o per finalità di ricerca scientifica e storica o per finalità statistiche, conformemente all'articolo 83.

Sicurezza informatica

La sicurezza informatica è la salvaguardia dei sistemi informatici da potenziali rischi e/o violazioni dei dati. L'obiettivo di un attacco è il contenuto informativo, quindi la sicurezza informatica deve preoccuparsi di:

- impedire l'accesso a utenti non autorizzati;
- regolamentare l'accesso ai diversi soggetti che potrebbero avere autorizzazioni diverse per evitare che i dati appartenenti al sistema informatico vengano copiati, modificati o cancellati.

Ciò che va protetto è l'**informazione**, in particolare la riservatezza, l'integrità, l'autenticità e la disponibilità. L'informazione va trattata per mezzo di calcolatori e reti, quindi è necessario avere un accesso controllato a tali mezzi tramite identificazione, autenticazione e autorizzazione.

Le violazioni possono essere molteplici:

- tentativi non autorizzati di accesso a zone riservate;
- furto di identità digitale o di file riservati;
- utilizzo di risorse che l'utente non dovrebbe poter utilizzare;
- etc.

La sicurezza informatica si occupa anche di prevenire eventuali DoS, *Denial of Service*: sono attacchi sferrati al sistema con l'obiettivo di rendere inutilizzabili alcune risorse onde danneggiare gli utenti.

Nella scelta delle misure di sicurezza incidono diverse caratteristiche dell'informazione e del contesto: dinamicità, dimensione e tipo di accesso, tempo di vita, costo di generazione, costo in caso di violazione, valore percepito e tipologia di attaccante. Nel garantire la protezione fisica è essenziale che la vulnerabilità fisica non sia la più facilmente attaccabile.

Crittografia e biometria

La crittografia può essere di due tipi: simmetrica, che garantisce riservatezza ma non identifica né autentica, e asimmetrica, che garantisce riservatezza con l'obiettivo di identificare e quindi autenticare. Sono necessarie infrastrutture per la certificazione della chiave pubblica, ovvero terze parti fidate che possano certificare l'autenticità di una chiave pubblica.

La **crittografia simmetrica**, classica e moderna, è implementata con dispositivi segreti, algoritmi segreti o chiavi segrete. Tipicamente le tecniche derivano dalla teoria dell'informazione (confusione e diffusione) e si usa una singola chiave per cifrare e decifrare.

La **crittografia asimmetrica** invece è moderna, infatti nasce ufficialmente nel 1976. È basata sulla teoria della complessità computazionale: sono presenti due chiavi correlate ma non facilmente calcolabili l'una dall'altra. La chiave privata è strettamente personale e quindi identifica il possessore; l'uso di una determinata chiave privata può essere verificato da chiunque per mezzo della corrispondente chiave pubblica.

La **biometria** è un componente cardine per la terna di fattori per l'autenticazione forte. Un problema non ancora risolto è il fatto che viene usata per l'identificazione (lunghe operazioni di confronto e cattivo bilanciamento tra falsi positivi e falsi negativi). È meglio usare la biometria per l'autenticazione, dove si fa un solo confronto con minore probabilità di errori e le prestazioni sono più sfumate rispetto ad altre tecniche.

Un aspetto fondamentale è la sicurezza delle password: è necessario impedire l'accesso a utenti non autorizzati e nascondere e/o vincolare l'accesso a documenti.

Definizione di una politica di sicurezza

La definizione di una politica di sicurezza deve tenere conto di vincoli tecnici, logistici, amministrativi, politici ed economici imposti dalla struttura organizzativa in cui il sistema opera. Per questo serve introdurre la sicurezza sin dalle prime fasi di analisi dei requisiti di un nuovo sistema: le vigenti leggi, le politiche e i vincoli aziendali sono la base di partenza per la definizione di un piano per la sicurezza. Un'applicazione o un servizio possono consistere di uno o più componenti funzionali allocati localmente o distribuiti sulla rete. La sicurezza viene vista come un processo complesso, come una catena di caratteristiche dalla computer system security, network security, application-level security sino alle problematiche di protezione dei dati sensibili. La sfida maggiore lanciata ai progettisti è quella di progettare applicazioni sicure e di qualità che tengano conto in modo strutturato di tutti gli aspetti della sicurezza sin dalle prime fasi di analisi del sistema.

Sistemi critici

I sistemi critici sono sistemi tecnici o socio-tecnici da cui dipendono persone o aziende. Se questi sistemi non forniscono i loro servizi come ci si aspetta, possono verificarsi seri problemi e importanti perdite. Ci sono tre tipi di sistemi critici:

- **sistemi safety-critical**: i fallimenti possono provocare incidenti, perdita di vite umane o seri danni ambientali;
- **sistemi mission-critical**: i malfunzionamenti possono causare il fallimento di alcune attività e obiettivi diretti;
- **sistemi business-critical**: i fallimenti possono portare a costi molto alti per le aziende che li usano.

La proprietà più importante di un sistema critico è la sua **fidatezza**, ovvero un insieme di disponibilità, affidabilità, sicurezza e protezione. Ci sono diverse ragioni per le quali la fidatezza è importante:

- i sistemi non affidabili, non sicuri e non protetti sono rifiutati dagli utenti;
- i costi di un fallimento del sistema potrebbero essere enormi;
- i sistemi inaffidabili possono causare perdita di informazioni.

I componenti del sistema che possono causare fallimenti sono hardware, software e anche umani. Nel caso dell'hardware, si può fallire a causa di errori nella progettazione, di un guasto a un componente o perché i componenti hanno terminato la loro vita naturale; nel caso software, si può fallire a causa di errori nelle sue specifiche, nella sua progettazione o nella sua implementazione; per quanto riguarda gli operatori umani, questi

possono sbagliare a interagire con il sistema. Con l'aumentare dell'affidabilità di software e hardware gli errori umani sono diventati la più probabile causa di difetto di un sistema.

Attacchi

La sicurezza e la protezione dei sistemi critici sono diventate sempre più importanti con l'aumentare delle connessioni di rete. Da una parte le connessioni di rete espongono il sistema ad attacchi da parte di malintenzionati, dall'altra parte fa in modo che i dettagli delle vulnerabilità siano facilmente divulgati e facilita la distribuzione di patch. Esempi di attacchi sono:

- **exploit**: metodo che sfrutta un bug o una vulnerabilità per l'acquisizione di privilegi;
- **buffer overflow**: fornire al programma più dati di quanto esso si aspetti di ricevere, in modo che una parte di questi vadano scritti in zone di memoria dove sono, o dovrebbero essere, altri dati o lo stack del programma stesso;
- **shell code**: sequenza di caratteri che rappresenta un codice binario in grado di lanciare una shell, che può essere utilizzato per acquisire un accesso alla linea di comando;
- **sniffing**: attività di intercettazione passiva dei dati che transitano in una rete;
- **cracking**: modifica di un software per rimuovere la protezione dalla copia, oppure per ottenere accesso ad un'area riservata;
- **spoofing**: tecnica con la quale si simula un indirizzo IP privato da una rete pubblica facendo credere agli host che l'IP della macchina server da contattare sia il suo;
- **trojan**: programma che contiene funzionalità maliziose. La vittima è indotta a eseguire il programma poiché viene spesso inserito nei videogiochi pirati;
- **DoS**: il sistema viene forzatamente messo in uno stato in cui i suoi servizi non sono disponibili, influenzando così la disponibilità del sistema.

Ingegneria della sicurezza

L'ingegneria della sicurezza fa parte del più vasto campo della sicurezza informatica. Nell'ingegnerizzazione di un sistema software non si può prescindere dalla consapevolezza delle minacce che il sistema dovrà affrontare e dei modi in cui tali minacce possono essere neutralizzate. Quando si considerano le problematiche di sicurezza nell'ingegnerizzazione di un sistema, vanno presi in considerazione due aspetti diversi: la sicurezza dell'applicazione e la sicurezza dell'infrastruttura su cui il sistema è costruito.

La sicurezza di un'applicazione è un problema di ingegnerizzazione del software dove gli ingegneri devono garantire che il sistema sia progettato per resistere agli attacchi. La sicurezza dell'infrastruttura è invece un problema manageriale nel quale gli amministratori dovrebbero garantire che l'infrastruttura sia configurata per resistere agli attacchi; gli amministratori dei sistemi devono inizializzare l'infrastruttura in modo tale che tutti i servizi di sicurezza siano disponibili e devono monitorare e riparare eventuali falle di sicurezza che emergono durante l'uso del software.

Minacce e controlli

Ci sono diversi tipi di minacce:

- **minacce alla riservatezza del sistema o dei suoi dati**: le informazioni possono essere svelate a persone o programmi non autorizzati;
- **minacce all'integrità del sistema o dei suoi dati**: i dati o il software possono essere danneggiati o corrotti;
- **minacce alla disponibilità del sistema o dei suoi dati**: può essere negato l'accesso agli utenti autorizzati al software o ai dati.

Queste minacce sono interdipendenti; se un attacco rende il sistema non disponibile, la modifica sulle informazioni potrebbe non avvenire, rendendo così il sistema non integro.

Esistono anche diversi tipi di controlli:

- **controlli per garantire che gli attacchi non abbiano successo**: la strategia è quella di progettare il sistema in modo da evitare i problemi di sicurezza (ad esempio i sistemi militari sensibili non sono connessi alla rete pubblica);
- **controlli per identificare e respingere attacchi**: la strategia è quella di monitorare le operazioni del sistema e identificare pattern di attività atipici, nel caso agire di conseguenza (spegnere parti di sistema, restringere l'accesso agli utenti, etc.);
- **controlli per il ripristino**: backup, replicazione, polizze assicurative, etc.

Analisi del rischio

L'analisi del rischio si occupa di valutare le possibili perdite che un attacco può causare ai beni di un sistema e di bilanciare queste perdite con i costi richiesti per la protezione dei beni stessi. Si noti che i costi di protezione sono molto inferiori ai costi causati dalle perdite.

L'analisi del rischio è una problematica più manageriale che tecnica. Il ruolo degli ingegneri della sicurezza è quindi quello di fornire una guida tecnica e giuridica sui problemi di sicurezza del sistema, sarà poi compito dei manager decidere se accettare i costi della sicurezza o i rischi che derivano dalla mancanza di procedure di sicurezza. L'analisi del rischio inizia dalla valutazione delle politiche di sicurezza organizzazionali che spiegano cosa dovrebbe e cosa non dovrebbe essere consentito fare. Le politiche di sicurezza propongono le condizioni che dovrebbero sempre essere mantenute dal sistema di sicurezza, quindi aiutano a identificare le minacce che potrebbero sorgere. La valutazione del rischio è un processo in più fasi:

- **valutazione preliminare del rischio**: determina i requisiti di sicurezza dell'intero sistema;
- **ciclo di vita della valutazione del rischio**: avviene contestualmente e segue il ciclo di vita dello sviluppo del software.

Analisi dei beni

Durante questa fase si può stabilire la seguente agenda di attività: analisi delle risorse fisiche, analisi delle risorse logiche e infine analisi delle dipendenze fra risorse.

Nell'attività di **analisi delle risorse fisiche**, il sistema informatico viene visto come insiemi di dispositivi che per funzionare hanno bisogno di spazio, alimentazione, condizioni ambientali adeguate, protezioni da furti e danni materiali. In particolare, occorre:

- un'individuazione sistematica di tutte le risorse fisiche;
- ispezionare e valutare tutti i locali che ospiteranno le risorse fisiche;
- verificare la cablatura dei locali.

Nell'attività di **analisi delle risorse logiche** il sistema viene visto come insieme di informazioni, flussi e processi; in particolare, occorre:

- classificare le informazioni in base al valore che hanno per l'organizzazione, il grado di riservatezza e il contesto di afferenza;
- classificare i servizi offerti dal sistema informatico affinché non presentino effetti collaterali pericolosi per la sicurezza del sistema.

Infine, nell'**analisi delle dipendenze tra risorse**, per ciascuna risorsa (fisica o logica) occorre individuare di quali altre risorse essa ha bisogno per funzionare correttamente. Questa analisi tende a evidenziare le risorse potenzialmente critiche, ovvero quelle da cui dipende il funzionamento di un numero elevato di altre cose. I risultati di questa analisi sono usati anche nella fase di valutazione del rischio, in particolare sono di supporto allo studio della progettazione dei malfunzionamenti a seguito dell'occorrenza di eventi indesiderati.

Identificazione delle minacce

In questa fase si cerca di definire quello che non deve poter accadere nel sistema. Si parte dal considerare come evento indesiderato qualsiasi accesso che non sia esplicitamente permesso; a tal fine è possibile in generale distinguere tra attacchi intenzionali ed eventi accidentali.

Gli **attacchi intenzionali** vengono caratterizzati in funzione della risorsa (sia fisica che logica) che viene attaccata e delle possibili tecniche usate per l'attacco. Le tecniche di attacco possono essere classificate in funzione del livello al quale operano: si distingue tra tecniche a livello fisico e a livello logico. Gli **attacchi a livello fisico** sono principalmente tesi a sottrarre o danneggiare le risorse critiche, si tratta quindi di furto (un attacco alla disponibilità e alla riservatezza) e danneggiamento (un attacco alla disponibilità e all'integrità). Gli **attacchi a livello logico**, invece, sono principalmente tesi a sottrarre informazione o degradare l'operatività del sistema. Dal punto di vista dei risultati che è indirizzato a conseguire, un attacco logico può essere classificato come: intercettazione e deduzione, intrusione, disturbo.
Negli **eventi accidentali** invece, una loro possibile casistica in base alla frequenza è:

- a livello fisico:
    - guasti ai dispositivi che compongono il sistema;
    - guasti ai dispositivi di supporto.
- A livello logico:
    - perdita di password o chiave hardware;
    - cancellazione di file;
    - corruzione del software di sistema.

Valutazioni

A ogni minaccia occorre associare un rischio, così da indirizzare l'attività di individuazione delle contromisure verso le aree più critiche. Per **rischio** si intende una combinazione della probabilità che un evento accada con il danno che l'evento può arrecare al sistema. Nel valutare il danno si tiene conto delle dipendenze tra le risorse e dell'eventuale propagazione del malfunzionamento.
La probabilità di occorrenza di attacchi intenzionali dipende principalmente dalla facilità di attuazione e dai vantaggi che potrebbe trarne l'intruso. Il danno si misura come grado di perdita dei tre requisiti fondamentali (riservatezza, integrità, disponibilità) ma l'attaccante applicherà sempre tutte le tecniche di cui dispone, su tutte le risorse attaccabili: è quindi necessario valutare anche il rischio di un attacco composto, ovvero un insieme di attacchi elementari concepiti con un medesimo obiettivo e condotti in sequenza.
Occorre scegliere il controllo da adottare per neutralizzare gli attacchi individuati:

- valutazione del rapporto costo/efficacia;
- analisi di standard e modelli di riferimento;
- controllo di carattere organizzativo;
- controllo di carattere tecnico.

La **valutazione del rapporto costo/efficacia** permette di valutare il grado di adeguatezza di un controllo; mira ad evitare che i controlli presentino un costo ingiustificato rispetto al rischio dal quale proteggono. L'efficacia del controllo è definita come funzione del rischio rispetto agli eventi indesiderati che neutralizza. Il costo di un controllo deve essere calcolato senza dimenticare i costi nascosti. Occorre tenere presenti le limitazioni che i controlli impongono e le operazioni di controllo che introducono nel flusso di lavoro del sistema informatico e dell'organizzazione. Le principali voci di costo sono: costo di messa in opera del controllo, peggioramento dell'ergonomia dell'interfaccia utente, decadimento delle prestazioni del sistema nell'erogazione dei servizi e l'aumento della burocrazia.
I **controlli di carattere organizzativo** vengono eseguiti con la condizione essenziale che affinché la tecnologia a protezione del sistema informatico risulti efficace, questa venga utilizzata nel modo corretto da personale pienamente consapevole. Devono quindi essere definiti con precisione ruoli e

responsabilità nella gestione sicura di tale sistema; per ciascun ruolo, dall'amministratore al semplice utente, devono essere definite norme comportamentali e procedure precise da rispettare.
I **controlli di carattere tecnico** sono o controlli di base, a livello del sistema operativo e dei servizi di rete, o controlli specifici del particolare sistema (questi si attestano normalmente a livello applicativo). I controlli tecnici più frequenti sono la configurazione sicura del sistema operativo di server e postazioni di lavoro e il confinamento logico delle applicazioni server su server dedicati.
Alcuni controlli di carattere tecnico sono:

- l'etichettatura delle informazioni allo scopo di avere un controllo più fine sui diritti di accesso;
- moduli software di cifratura integrati con le applicazioni;
- apparecchiature di telecomunicazione in grado di cifrare il traffico dati in modo trasparente alle applicazioni;
- firewall e server proxy in corrispondenza di eventuali collegamenti con reti TCP/IP;
- chiavi hardware e/o dispositivi di riconoscimento degli utenti basati su rilevamenti biofisici.

Un insieme di controlli non deve presentarsi come una "collezione di espedienti" non correlati tra loro, ma è importante integrare i vari controlli in una politica di sicurezza organica. Occorre operare una selezione dei controlli adottando un sottoinsieme di costo minimo che rispetti alcuni vincoli:

- **completezza**: il sottoinsieme deve fare fronte a tutti gli eventi indesiderati;
- **omogeneità**: le contromisure devono essere compatibili e integrabili tra loro;
- **ridondanza controllata**: la ridondanza delle contromisure ha un costo e deve essere rilevata e vagliata accuratamente;
- **effettiva attuabilità**: l'insieme delle contromisure deve rispettare tutti i vincoli imposti dall'organizzazione nella quale andrà ad operare.

Nel ciclo di vita della valutazione del rischio, è necessaria la conoscenza dell'architettura del sistema e dell'organizzazione dei dati. La piattaforma e il middleware sono già stati scelti, così come la strategia di sviluppo del sistema; questo significa che si hanno molti più dettagli riguardo a che cosa è necessario proteggere e sulle possibili vulnerabilità del sistema. Le vulnerabilità possono essere "ereditate" dalle scelte di progettazione, ma non solo. La valutazione del rischio dovrebbe essere parte di tutto il ciclo di vita del software: dall'ingegnerizzazione dei requisiti al deployment del sistema. Il processo seguito è simile a quello della valutazione preliminare dei rischi, con l'aggiunta di attività riguardanti l'identificazione e la valutazione delle vulnerabilità. La valutazione delle vulnerabilità identifica i beni che hanno più probabilità di essere colpiti da tali vulnerabilità; vengono messe in relazione le vulnerabilità con i possibili attacchi di sistema. Il risultato della valutazione del rischio è un insieme di decisioni ingegneristiche che influenzano la progettazione o l'implementazione del sistema o limitano il modo in cui esso è usato.

Security use case e misuse case

I **misuse case** si concentrano sulle interazioni tra l'applicazione e gli attaccanti che cercano di violarla. La condizione di un successo di un misuse case è l'attacco andato a buon fine: questo li rende particolarmente adatti per analizzare le minacce, ma non molto utili per la determinazione dei requisiti di sicurezza; è invece compito dei **security use case** specificare i requisiti tramite i quali l'applicazione dovrebbe essere in grado di proteggersi dalle minacce.
Alcune linee guida per i security use case sono:

- i casi d'uso non dovrebbero mai specificare meccanismi di sicurezza, infatti le decisioni relative ai meccanismi devono essere lasciate alla progettazione;

- vanno attentamente differenziati dalle informazioni secondarie: interazioni del sistema, azioni del sistema e post-condizioni sono i soli requisiti;
- evitare di specificare vincoli progettuali non necessari;
- documentare esplicitamente i percorsi individuali attraverso i casi d'uso al fine di specificare i reali requisiti di sicurezza;
- basare i security use case su differenti tipi di requisiti di sicurezza fornisce una naturale organizzazione dei casi d'uso;
- documentare le minacce alla sicurezza che giustificano i percorsi individuali attraverso i casi d'uso;
- distinguere chiaramente tra interazioni degli utenti e degli attaccanti;
- distinguere chiaramente tra le interazioni che sono visibili esternamente e le azioni nascoste del sistema;
- documentare sia le precondizioni che le post-condizioni che catturano l'essenza dei percorsi individuali.

Requisiti di sicurezza

Non è sempre possibile specificare i requisiti associati alla sicurezza in modo quantitativo; quasi sempre questa tipologia di requisiti è espressa nella forma "non deve": definisce comportamenti inaccettabili per il sistema ma non definisce funzionalità richieste al sistema. L'approccio convenzionale della specifica dei requisiti è basato sul contesto, sui beni da proteggere e sul loro valore per l'organizzazione. Le categorie dei requisiti di sicurezza sono:

- **requisiti di identificazione**: specificano se un sistema deve identificare gli utenti prima di interagire con loro;
- **requisiti di autenticazione**: specificano come identificare gli utenti;
- **requisiti di autorizzazione**: specificano i privilegi e i permessi di accesso degli utenti identificati;
- **requisiti di immunità**: specificano come il sistema deve proteggersi da virus, worm e minacce simili;
- **requisiti di integrità**: specificano come evitare la corruzione dei dati;
- **requisiti di scoperta delle intrusioni**: specificano quali meccanismi utilizzare per scoprire gli attacchi al sistema;
- **requisiti di non-ripudiazione**: specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento;
- **requisiti di riservatezza**: specificano come deve essere mantenuta la riservatezza delle informazioni;
- **requisiti di controllo della protezione**: specificano come può essere controllato e verificato l'uso del sistema;
- **requisiti di protezione della manutenzione del sistema**: specificano come un'applicazione può evitare modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione.

2.7. Progettazione per la sicurezza

La sicurezza non è qualcosa che può essere aggiunto al sistema ma deve essere progettata insieme al sistema prima dell'implementazione. La "sicurezza" è anche un problema implementativo, spesso infatti le vulnerabilità sono introdotte durante la fase di implementazione: è possibile ottenere un'implementazione non sicura da una progettazione sicura ma non è possibile ottenere un'implementazione sicura partendo da una progettazione non sicura.

Progettazione architetturale
La scelta dell'architettura del sistema influenza profondamente la sicurezza; un'architettura inappropriata non garantisce riservatezza e integrità delle informazioni e il livello di disponibilità richiesto. Vanno considerati due problemi fondamentali quando si progetta l'architettura del sistema:

- **protezione**: come dovrebbe essere organizzato il sistema in modo che i beni critici possano essere protetti dagli attacchi esterni;
- **Distribuzione**: come dovrebbero essere distribuiti i beni in modo da minimizzare gli effetti di un attacco andato a buon fine.

I due problemi sono potenzialmente in conflitto: se si mettono tutti i beni in un unico posto si può costruire un buon livello di protezione ad un costo non eccessivo; se però la protezione fallisce, tutti i beni sono compromessi. Distribuire i beni porta a un maggiore costo per la protezione e ci sono più possibilità che la protezione possa fallire se i beni sono distribuiti, ma se questo avviene vi è la perdita solo parziale dei beni.
Tipicamente la miglior architettura per fornire un alto grado di protezione è quella a layer, in cui si posiziona i beni critici da proteggere nel livello più basso. Il numero di layer necessari varia da applicazione ad applicazione e dipende dalla criticità dei beni che devono essere protetti. Per migliorare la protezione inoltre sarebbe bene che le credenziali di accesso ai diversi livelli fossero diverse tra loro.
Se la protezione dei dati fosse un requisito critico si potrebbe anche usare un'architettura client-server con i meccanismi di protezione locati nella macchina server. La versione tradizionale client-server ha svariate limitazioni: ad esempio, se la sicurezza viene compromessa, le perdite associate ad un attacco saranno alte e i costi di recupero saranno anch'essi elevati. Il sistema è inoltre maggiormente soggetto ad attacchi di DoS che sovraccaricano il server. Una possibile soluzione può essere quella di adottare un'architettura distribuita in cui il server viene replicato in punti diversi della rete.
Un tipico problema è che lo stile architetturale più appropriato per la sicurezza potrebbe essere in conflitto con gli altri requisiti dell'applicazione e quindi soddisfare entrambi i requisiti nella stessa architettura presenta molti problemi.

Linee guida di progettazione
Non ci sono regole rigide per ottenere un sistema sicuro. Differenti tipi di sistema richiedono differenti misure tecniche per ottenere un livello di sicurezza accettabile. La posizione e i requisiti di diversi gruppi di utenti influenzano pesantemente cosa è e cosa non è accettabile; ci sono comunque linee guida generali di ampia applicabilità per la progettazione di sistemi sicuri che possono fungere da:

- mezzo per migliorare la consapevolezza dei problemi di sicurezza in un team di progettisti software;
- base per una lista di controlli da fare durante il processo di validazione del sistema.

Alcune linee guida di progettazione sono:

- basare le decisioni della sicurezza su un'esplicita politica;
- evitare un singolo punto di fallimento;
- fallire in un certo modo;
- bilanciare sicurezza e usabilità;
- essere consapevoli dell'esistenza dell'ingegneria sociale;
- usare ridondanza e diversità riduce i rischi;
- validare tutti gli input;

- dividere in compartimenti i beni;
- progettare per il deployment;
- progettare per il ripristino.

La **security policy** è un documento di alto livello che definisce cosa è la sicurezza ma non come ottenerla. La policy non dovrebbe definire i meccanismi usati per fornire e far rispettare la sicurezza. Le politiche di sicurezza devono essere incorporate nella progettazione al fine di:

- specificare come le informazioni possono essere accedute;
- quali precondizioni devono essere testate per l'accesso;
- a chi concedere l'accesso.

Tipicamente le politiche vengono rappresentate come un insieme di regole e condizioni; tali regole devono essere incorporate in uno specifico componente del sistema chiamato *Security Authority* che avrà il compito di far rispettare le politiche all'interno dell'applicazione. A livello progettuale le politiche di sicurezza sono suddivise in sei specifiche categorie:

- **identity policies**: regole per la verifica delle credenziali degli utenti;
- **access control policies**: regole da applicare sia alle richieste di accesso alle risorse sia all'esecuzione di specifiche operazioni messe a disposizione dall'applicazione;
- **content-specific policies**: regole da applicare a specifiche informazioni durante la memorizzazione e la comunicazione;
- **network and infrastructure policies**: regole per controllare il flusso dei dati e il deployment sia delle reti che dei servizi infrastrutturali di hosting pubblici e privati;
- **regulatory policies**: regole a cui l'applicazione deve sottostare per soddisfare i requisiti legali e le leggi in vigore nel Paese/Stato in cui il sistema opera;
- **advisor and information policies**: queste regole non sono imposte, ma sono caldamente consigliate in riferimento alle regole dell'organizzazione e al ruolo delle attività di business. Per esempio, queste regole possono essere applicate per informare il personale sull'accesso ai dati sensibili o per stabilire comunicazioni commerciali con partner esterni.

Nei sistemi critici è buona norma di progettazione quella di cercare di **evitare un singolo punto di fallimento**. Questo perché un singolo fallimento in una parte del sistema non si trasformi nel fallimento di tutto il sistema; per quanto riguarda la sicurezza questo significa che non ci si dovrebbe affidare a un singolo meccanismo per assicurarla, ma si dovrebbero impiegare differenti tecniche: questo viene spesso chiamato "difesa in profondità". Ad esempio, se si usa la password per autenticare, si dovrebbe anche includere un meccanismo di sfida e risposta. Qualche tipo di fallimento è inevitabile in tutti i sistemi, ma i sistemi critici per la sicurezza dovrebbero sempre fallire in modo sicuro. Non si dovrebbero avere procedure di fall-back meno sicure del sistema stesso e anche se il sistema fallisce non deve essere consentito ad un attaccante di accedere ai dati riservati.

Sicurezza e usabilità sono spesso in contrasto: per avere sicurezza bisogna introdurre un numero di controlli che garantiscano che gli utenti siano autorizzati ad usare il sistema e che nello stesso tempo agiscano in accordo alle politiche di sicurezza; questo inevitabilmente ricade sull'utente che ha bisogno di più tempo per imparare ad utilizzare il sistema. Ogni volta che si aggiunge una caratteristica di sicurezza al sistema questo inevitabilmente diventa meno usabile, quindi a volte può diventare contro produttivo introdurre nuove caratteristiche di sicurezza a spese dell'usabilità.

L'**ingegneria sociale** si occupa di trovare modi per convincere con l'inganno utenti accreditati al sistema a rivelare informazioni riservate. Questi approcci si avvantaggiano della "volontà di aiutare" delle persone e della loro fiducia nell'organizzazione. Dal punto di vista della progettazione contrastare l'ingegneria sociale è quasi impossibile. Se la sicurezza fosse molto critica non ci si dovrebbe affidare solo a meccanismi di autenticazione basati su password ma bisognerebbe utilizzare meccanismi di autenticazione forte: meccanismi di log che tracciano sia la locazione che l'identità dell'utente e programmi di analisi del log che potrebbero essere utili ad identificare brecce nella sicurezza.

Ridondanza significa mantenere più di una versione del software e dei dati nel sistema. **Diversità** significa che le diverse versioni del sistema non dovrebbero usare la stessa piattaforma o essere basati sulle stesse tecnologie; in questo modo, una vulnerabilità della piattaforma o della tecnologia non influirà su tutte le versioni e non condurrà a un comune punto di fallimento.
Un comune attacco al sistema consiste nel fornire input inaspettati che causano un comportamento imprevisto: crash, perdita della disponibilità del servizio, esecuzione di codice malizioso. Tipi esempi sono buffer overflow e SQL injection. Si possono evitare molti di questi problemi progettando la **validazione dell'input** in tutto il sistema: nei requisiti dovrebbero essere definiti tutti i controlli che devono essere applicati e bisogna usare la conoscenza dell'input per definire questi controlli.
Compartimentalizzare significa organizzare le informazioni nel sistema in modo che gli utenti abbiano accesso solo alle informazioni necessarie piuttosto che a tutte le informazioni nel sistema. Gli effetti di un attacco in questo modo sono più contenuti: qualche informazione sarà persa o danneggiata, ma è poco probabile che tutte le informazioni del sistema siano coinvolte.
Molti problemi di sicurezza sorgono perché il sistema non viene configurato correttamente al momento del deployment. Bisogna sempre progettare il sistema in modo che siano inclusi programmi di utilità per semplificare il deployment e in modo che si possano verificare potenziali errori di configurazione ed omissioni nel sistema di deployment.
Bisogna sempre progettare il sistema con l'assunzione che gli errori di sicurezza possano accadere; si deve quindi pensare a come ripristinare il sistema dopo possibili errori e riportarlo ad uno stato operazionale sicuro.
Il deployment di un sistema coinvolge:

- configurazione del sistema per operare nell'ambiente: dalla semplice impostazione di parametri delle preferenze degli utenti alla definizione di regole e modelli di business che governano l'esecuzione del software;
- installazione del sistema sui computer dell'ambiente;
- configurazione del sistema installato.

Nella fase di deployment vengono spesso introdotte in modo accidentale delle vulnerabilità. La configurazione e il deployment sono spesso visti solo come problemi di amministrazione e quindi al di fuori del processo di ingegnerizzazione mentre in realtà i progettisti software hanno la responsabilità di progettare per il deployment.
Bisogna sempre fornire supporti per il deployment che riducano la probabilità che gli amministratori compiano degli errori quando configurano il software. Esistono delle linee guida per la progettazione per il deployment:

- includere supporto per visionare ed analizzare le configurazioni;
- minimizzare i privilegi di default;
- localizzare le impostazioni di configurazione;
- fornire modi per rimediare a vulnerabilità di sicurezza.

Si devono sempre **includere programmi di utilità** che consentano agli amministratori di esaminare la configurazione corrente del sistema; sorprendentemente questi programmi mancano nella maggior parte dei sistemi software e gli utenti sono spesso frustrati dalla difficoltà di trovare i dettagli della configurazione: per un quadro completo della configurazione spesso occorre visionare diversi menu e questo porta ad errori ed omissioni. Idealmente in fase di visualizzazione delle configurazioni si dovrebbero evidenziare impostazioni critiche per la sicurezza.
Il software deve essere progettato in modo tale che la **configurazione di default fornisca i minimi privilegi essenziali**; in questo modo vengono limitati i danni di un possibile attacco.
Quando si progetta il supporto per le configurazioni del sistema bisognerebbe assicurarsi che ogni risorsa che appartiene alla stessa parte del sistema venga configurata nella stessa posizione. Se le informazioni di configurazione non sono localizzate:

- è facile dimenticarsi di farlo;
- può capitare di non essere a conoscenza dell'esistenza di meccanismi per la sicurezza già inclusi nel sistema;
- se tali meccanismi presentano configurazioni di default si potrebbe essere esposti ad attacchi.

Bisogna **includere meccanismi diretti per aggiornare il sistema** e riparare le vulnerabilità di sicurezza che vengono scoperte. Questi potrebbero includere verifiche automatiche per aggiornamenti di sicurezza e il download di tali aggiornamenti non appena sono disponibili. Va comunque considerato che gli aggiornamenti devono coinvolgere contemporaneamente centinaia di PC su cui tipicamente il software è installato.

Testare la sicurezza

Il test di un sistema gioca un ruolo chiave nel processo di sviluppo software e dovrebbe essere eseguito con molta attenzione. È quindi sorprendente che l'area dei test della sicurezza sia quella più trascurata durante lo sviluppo del sistema; questo può essere attribuito a diversi fattori:

- mancanza di comprensione dell'importanza dei test relativi alla sicurezza;
- mancanza di tempo;
- mancanza di conoscenza su come svolgere un test di sicurezza;
- mancanza di tool integrati per compiere test.

Il test della sicurezza è un lavoro molto lungo e tedioso, spesso molto più complesso dei test funzionali che vengono svolti normalmente. Inoltre, esso coinvolge diverse discipline: ci sono tradizionali test per accertare la sicurezza dei requisiti applicativi che possono essere svolti normalmente dal team di testing ma esistono dei test non funzionali di "rottura" del sistema che devono essere condotti da esperti di sicurezza, ovvero il black box testing e il white box testing.

Il **black box testing** ha come assunzione di base la non conoscenza dell'applicazione. I tester affrontano l'applicazione come farebbe un attaccante: indagano sulle informazioni riguardanti la struttura interna e successivamente applicano un insieme di tentativi di violazione del sistema basati sulle informazioni ottenute. I tester possono impiegare una varietà di tool per scansionare e indagare l'applicazione: ci sono centinaia di tool in rete per l'hacking di applicazioni che permettono di "scandagliare" le porte dei sistemi perpetuando attacchi sfruttando le debolezze ben conosciute di svariati linguaggi di programmazione. Questo test non prende in esame solo debolezze del codice, ma vengono svolti test mirati anche al livello infrastrutturale:

- errori di configurazione di reti e host;
- falle di sicurezza nelle macchine virtuali;
- problemi legati ai linguaggi di implementazione.

Il **white box testing**, invece, ha come assunzione di base la completa conoscenza dell'applicazione. I tester hanno accesso a tutte le informazioni di configurazione e anche al codice sorgente; essi operano una revisione del codice cercando possibili debolezze e inoltre scrivono test per stabilire come trarre vantaggio dalle debolezze scoperte. Tipicamente questi tester sono ex-sviluppatori o persone che conoscono molto bene l'ambiente di sviluppo. I tool a disposizione differiscono molto da quelli usati nel black box test: tipicamente sono tool di debugging che consentono di trovare bachi e vulnerabilità specifici del sistema. I bachi tipici riguardano problemi di corsa critici e la mancanza di verifica dei parametri di input e sono specifici di ogni applicazione. Questi test portano a scoprire anche altri problemi, come il memory leak e problemi di prestazione che contribuiscono al danneggiamento della disponibilità e dell'affidabilità dell'intero sistema.

Capacità di sopravvivenza del sistema

Con il termine "capacità di sopravvivenza" si intende la capacità del sistema di continuare a fornire i servizi essenziali agli utenti legittimi mentre è sotto attacco o dopo che parti del sistema sono state danneggiate come conseguenza di un attacco o di un fallimento. La capacità di sopravvivenza è una proprietà dell'intero sistema, non dei singoli componenti di questo. Il lavoro sulla capacità di sopravvivenza è molto critico poiché l'economia e la vita sociale dipendono da infrastrutture controllate da computer. L'analisi e la progettazione della capacità di sopravvivenza dovrebbero essere parte del processo di ingegnerizzazione dei sistemi sicuri; la disponibilità dei servizi critici è l'essenza della sopravvivenza: questo significa conoscere:

- quali sono i servizi maggiormente critici;
- come questi servizi possono essere compromessi;
- qual è la qualità minima dei servizi che deve essere mantenuta;

- come proteggere questi servizi;
- come ripristinare velocemente il sistema se i servizi diventano non disponibili.

Il *Survivable Analysis Systems* è un metodo di analisi ideato per valutare le vulnerabilità nel sistema e supportare la progettazione di architetture e caratteristiche che promuovono la sopravvivenza del sistema. In questo metodo la sopravvivenza del sistema è un processo a 4 fasi e dipende da strategie complementari. Le strategie sono:

- **resistenza**: evitare problemi costruendo all'interno del sistema le capacità di respingere attacchi;
- **identificazione**: individuare problemi costruendo all'interno del sistema le capacità di riconoscere attacchi e fallimenti e valutare il danno risultante;
- **ripristino**: tollerare problemi costruendo all'interno del sistema le capacità di fornire servizi essenziali durante un attacco e ripristinare le complete funzionalità dopo l'attacco.

Le 4 fasi invece sono:

- **capire il sistema**: riesaminare gli obiettivi del sistema, i requisiti e l'architettura;
- **identificare servizi critici**: identificare i servizi che devono essere mantenuti e i componenti che devono svolgere tale compito;
- **simulare attacchi**: identificare gli scenari o i casi d'uso dei possibili attacchi insieme ai componenti influenzati da questi attacchi;
- **analizzare la sopravvivenza**: identificare i componenti che sono sia essenziali che a rischio e identificare le strategie di sopravvivenza basate su resistenza, identificazione e ripristino.

Aggiungere le tecniche di sopravvivenza ha un costo e spesso le aziende sono molto riluttanti nell'investire sulla sopravvivenza, specie se non hanno mai subito attacchi e perdite. È comunque sempre buona norma investire nella sopravvivenza prima piuttosto che dopo aver subito un attacco. L'analisi della sopravvivenza non è ancora inclusa nella maggior parte dei processi di ingegnerizzazione del software ma con la crescita dei sistemi critici sembra probabile che questo tipo di analisi sarà sempre più utilizzato.