

<b>MOD2</b>	4
<b>MODELLI</b>	5
Tracciabilità	5
Paradigma di modellazione	5
<b>Linguaggi di modellazione</b>	5
Uml	5
<b>Modelli di processo</b>	5
<b>Modello a cascata</b>	6
<b>Modelli evolutivi</b>	7
Extreme programming	7
<b>Modelli ibridi</b>	7
Sviluppo incrementale	7
Sviluppo iterativo	7
<b>Rational Unified Process</b>	8
Prospettiva Dinamica	8
Prospettiva Statica	8
Prospettiva Pratica	9
<b>Processi Software</b>	9
<b>SICUREZZA E PRIVACY</b>	10
<b>Nuova Normativa</b>	10
Pseudonimizzazione	10
<b>Concetti Base</b>	10
Analizzare e Progettare la Sicurezza	11
Sistemi Critici	11
<b>Security Engineering</b>	12
<b>Analisi del Rischio</b>	13
Valutazione Preliminare del Rischio	13
Ciclo di Vita della Valutazione del Rischio	14
<b>Specifiche dei Requisiti di Sicurezza</b>	14
Security Use Case e Misuse Case	15
<b>PROGETTAZIONE PER LA SICUREZZA</b>	16
<b>Progettazione Architetturale</b>	16
Linee Guida di Progettazione	16
<b>Security Testing</b>	17

Black Box Testing	17
White Box Testing	18
<b>Capacita di Sopravvivenza del Sistema</b>	<b>18</b>
<b>PROGETTAZIONE</b>	<b>19</b>
<b>Progettazione Architetturale</b>	<b>19</b>
Blackboard	20
MVC	20
Layer	20
Client/Server	20
Broker	21
Pipe & Filters	21
<b>Progettazione di Dettaglio</b>	<b>21</b>
<b>MOD1</b>	<b>22</b>
<b>Fattori di Qualità del Software</b>	<b>23</b>
<b>Principi OO</b>	<b>24</b>
Ereditarietà	24
Polimorfismo	24
Regole di Naming in .NET	24
<b>Struttura di una Applicazione</b>	<b>25</b>
<b>Introduzione al Framework .NET</b>	<b>25</b>
Tecnologia COM	25
Framework .NET	26
Common Language Runtime	27
Common Type System	28
Common Language Specification	28
<b>Tipi in .NET</b>	<b>28</b>
<b>Delegati ed Eventi</b>	<b>28</b>
Delegati	28
Eventi	29
<b>Interfaccia Utente</b>	<b>29</b>
<b>Metadati e Introspezione</b>	<b>30</b>
Metadati	30
Reflection	30
Meta Programming in .NET	31
<b>Design Principles</b>	<b>31</b>

Premessa	31
The Single Responsibility Principle	32
The Dependency Inversion Principle	32
The Interface Segregation Principle	32
The Open/Closed Principle	32
The Liskov Substitution Principle	32
Principles of Package Architecture	33
Release/Reuse Equivalency Principle	33
The Common Closure Principle	33
The Common Reuse Principle	33
Relationships between Packages	33
Acyclic Dependencies Principle	33
Stable Abstractions Principle	33
Stratification Principle	33
Interface Segregation Principle	34
Narrow Inheritance Interface	34
The Law of Demeter	34
The Common Reuse Principle	34
Design Pattern	35
Classificazione dei Design Pattern	35
Pattern SINGLETON	36
Pattern OBSERVER	36
Pattern MVC	36
Pattern FLYWEIGHT	37
Pattern STRATEGY	38
Pattern ADAPTER	39
Pattern DECORATOR	40
Ereditarietà Dinamica	40
Pattern STATE	41
Pattern COMPOSITE	42
Pattern VISITOR	43

Questo PDF contiene appunti del corso di Ingegneria del software tenuto dal prof. Patella e dalla prof.ssa Molesini nell'AA 2017/18. Contiene esclusivamente ciò che serve per affrontare bene lo scritto, non contiene le parti del modulo 2 che riguardano il progetto.

# MOD2

# MODELLI

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.

Quindi esso costituisce una visione semplificata di un sistema che lo rende più comprensibile e valutabile e facilita l'inserimento di informazione e la collaborazione fra persone.

Uno degli scopi dei processi model based è rendere esplicite alcune conoscenze di base utilizzando diagrammi che sono anche utili ad individuare rischi e scelte progettuali.

I modelli che descrivono un sistema devono darne una descrizione completa, consistente e non troppo ridondante.

## Tracciabilità

In qualsiasi direzione si percorra l'insieme dei modelli deve essere possibile mappare uno o più elementi in un modello in uno o più elementi di un altro, questo serve a garantire coerenza e consistenza tra i modelli, creare un percorso logico dai requisiti al codice (e viceversa) e tenere sotto controllo le modifiche

## Paradigma di modellazione

L'introduzione del paradigma ad oggetti promuove un rapporto fra struttura e funzionamento molto più articolato e offre uno spazio concettuale molto più ricco che caratterizza il modo con cui si analizzano, si progettano e si costruiscono oggi i sistemi SW.

Impostare un progetto e la costruzione di un sistema SW su questo paradigma permette di modularlo nel modo più conveniente il rapporto tra la parte algoritmica e la parte strutturale del sistema. Inoltre permette di superare la tradizionale concezione per la quale un programma è una sequenza di istruzioni su dei dati arrivando alla concezione di un sistema costituito da una rete di oggetti che interagiscono fra di loro attraverso interfacce accuratamente progettate.

## Linguaggi di modellazione

Un linguaggio di modellazione è un linguaggio semi-formale che si usa per modellare un sistema. Esempi: UML, XMI, OPM

Anche il codice è una rappresentazione del modello espressa in un particolare linguaggio, essa risulta più dettagliata e fornisce una visione flat, quindi non mette in evidenza punti salienti e non dà una visione d'insieme. Tanto che solitamente modelli e codice sono disallineati.

I diagrammi realizzati con i giusti linguaggi di modellazione servono sostanzialmente a risolvere i probabili problemi di comunicazione che si creerebbero all'interno del team di sviluppo.

## Uml

È un linguaggio che serve a visualizzare, specificare, costruire e documentare un sistema e gli elaborati prodotti durante il suo sviluppo ed è utilizzabile per la modellazione durante tutto il ciclo di vita del SW.

## Modelli di processo

Un processo di sviluppo è un insieme ordinato di passi che coinvolge tutte quelle attività, vincoli e risorse per produrre il desiderato output a partire dall'insieme dei requisiti.

Fasi generiche:

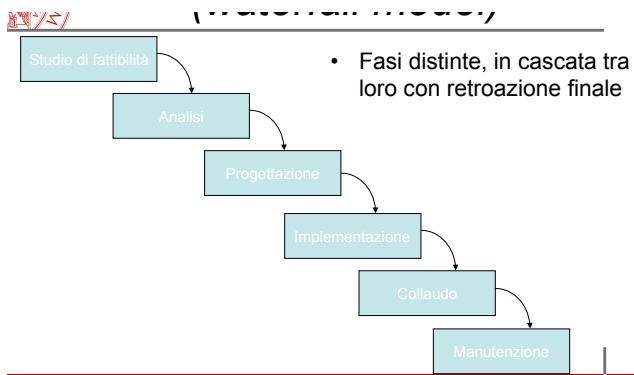
- **Specifico**: cosa il sistema dovrebbe fare e vincoli di sviluppo
- **Sviluppo**: produzione del sistema SW
- **Validazione**: testare che il sistema sviluppato sia quello che il committente voleva
- **Evoluzione**: cambiamenti nel prodotto in accordo a modifiche dei requisiti o incremento delle funzionalità

Un modello di processo SW è una rappresentazione semplificata di un processo presentato da una specifica prospettiva, sostanzialmente definisce un template attorno al quale organizzare un vero processo di sviluppo.

Modelli:

- Modello a cascata
- Modelli evolutivi
- Sviluppo incrementale
- Modello a spirale
- Modelli specializzati

## Modello a cascata



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. Ogni fase produce dei semilavorati del processo di sviluppo:

- Documentazione
- Codice
- Sistema complessivo

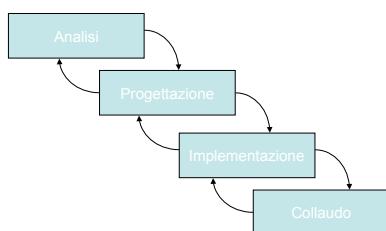
I limiti sono dati dalla rigidità che impone, in particolare da due assunti:

- **Immutabilità dell'analisi**
- **Immutabilità del processo**

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.

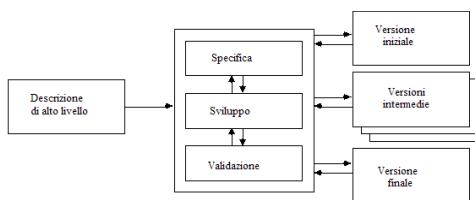
Il prototipo deve essere svilupparlo in tempi brevi e con costi minimi poiché sarà “usa e getta” in quanto ha l'obiettivo unico di comprendere meglio le richieste del cliente trasformandosi via via nel prodotto finale

- Evoluzioni successive al modello originale ammettono forme limitate di retroazione a un livello



# Modelli evolutivi

- Partendo da specifiche molto astratte, si sviluppa un primo prototipo
  - da sottoporre al cliente
  - da raffinare successivamente



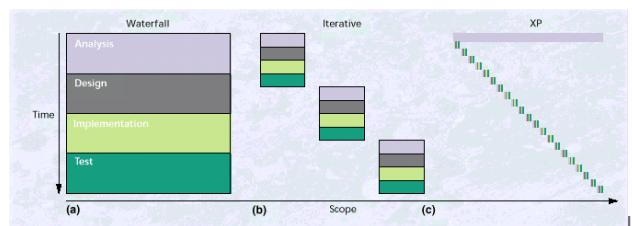
Questi modelli risultano applicabili su sistemi di piccole dimensioni e di breve durata che possono essere parti di sistemi più grandi. I maggiori problemi legati a questo modello sono la mancanza di visibilità del processo di sviluppo, la mancanza di strutturazione e richiedono una particolare abilità nella programmazione

Esistono vari 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 interazioni.

Il vantaggio fondamentale è che ad ogni interazione è possibile confrontarsi con gli utenti sulle specifiche e le funzionalità e rivedere le scelte di progetto.

I modelli evolutivi si sono orientati su cicli sempre più brevi e iterazioni sempre più veloci fino ad arrivare al modello più radicale detto *extreme programming*

- L'illustrazione, tratta da un articolo di Kent Beck, mostra l'evoluzione dal modello a cascata, all'*extreme programming*



## Extreme programming

Il modello più estremo fra quelli evolutivi è, come già detto, quello chiamato *extreme programming*.

Si basa su:

- Comunicazione fra gli sviluppatori
- Testing continuo e ben fatto
- Semplicità nel codice
- Coraggio: non si deve aver paura di modificare il sistema, sarà modificato in continuazione

## Modelli ibridi

Sono modelli composti di sotto sistemi per ognuno dei quali è possibile adottare un diverso modello di sviluppo.

Soltanamente conviene creare e raffinare prototipi funzionanti dell'intero sistema secondo l'approccio incrementale-iterativo

## Sviluppo incrementale

Si costruisce il sistema sviluppandone sistematicamente e in sequenza parti ben definite. Una volta costruita una parte esse non viene più modificata

## Sviluppo iterativo

Si effettuano molti passi dell'intero ciclo di sviluppo del SW per costruire iterativamente tutto il sistema aumentandone ogni volta il livello di dettaglio

### SVILUPPO INCREMENTALE-ITERATIVO

- Si individuano sottoparti relativamente autonome

- Si realizza il prototipo di una di esse
- Si continua con le altre parti
- Si aumenta progressivamente l'estensione e il dettaglio dei prototipi
- E così via ...

## Rational Unified Process

È un modello di processo SW iterativo ibrido, non definisce un singolo, specifico processo ma un framework adattabile che può dar luogo a diversi processi in diversi contesti (è pensato per progetti di grandi dimensioni)

Individua tre diverse visioni del processo di sviluppo:

- Dinamica
- Statica
- Pratica

### Prospettiva Dinamica

#### Inception

Generalizzazione dell'analisi di fattibilità. Lo scopo principale è quello di delineare nel modo più accurato il possibile business case, ovvero:

- comprendere il tipo di mercato
- Identificare tutte le entità esterne

#### Elaboration

Definisce la struttura complessiva del sistema. Comprende l'analisi di dominio e una prima fase di progettazione dell'architettura

#### Construction

Progettare, programmare e testare il sistema. Le diverse parti del sistema vengono sviluppate parallelamente e poi integrate, al termine della fase si dovrebbe avere un sistema funzionante e la relativa documentazione pronta.

#### Transition

Il sistema passa all'ambiente dello sviluppo a quello del cliente finale

### Prospettiva Statica

La prospettiva statica si concentra sulle attività di produzione del software chiamate workflow.

Tipi di workflow:

- **Modellazione** delle attività aziendali
- **Requisiti**: vengono individuati gli attori e sviluppati i casi d'uso
- **Analisi e Progetto**: viene creato e documentato un modello di progetto
- **Implementazione**: i componenti del sistema sono implementati
- **Test**
- **Rilascio**
- **Gestione della configurazione e delle modifiche**: gestisce i cambiamenti di sistema
- **Gestione del progetto**: gestisce lo sviluppo di sistema
- **Ambiente**: rende disponibili al team di sviluppatori gli strumenti adeguati

## Prospettiva Pratica

Describe la buona prassi di ingegneria del software che si raccomanda di usare nello sviluppo di sistemi.

Le pratiche fondamentali sono 6 :

- 1) **Sviluppare software ciclicamente**: pianificare gli incrementi del sistema basati sulle priorità del cliente
- 2) **Gestire i requisiti**: documentare esplicitamente i requisiti del cliente e i cambiamenti effettuati
- 3) **Usare architetture basate sui componenti**: strutturare l'architettura del sistema con un approccio a componenti
- 4) **Creare modelli visivi del software**: usare modelli grafici UML
- 5) **Verificare la qualità del software**: assicurarsi che il software raggiunga gli standard di qualità dell'organizzazione
- 6) **Controllare le modifiche del software**: gestire i cambiamenti del software usando un sistema per la gestione delle modifiche

## Processi Software

La definizione di un processo non può prescindere dalle caratteristiche dell'organizzazione in cui esso avviene e dalla competenza ed esperienze di chi è coinvolto.

***“Any piece of software reflects the organizational structure that produced it”***

L'uso di un processo inappropriato riduce la qualità e l'utilità del prodotto software che deve essere sviluppato o migliorato.

# SICUREZZA E PRIVACY

## Nuova Normativa

Impone l'obbligo di aderenze di un prodotto software che tratti dati personali ai principi della GDPR:

- privacy by design & by default
- minimaliste
- pseudonimizzazione
- Trasferimento dati fuori EU
- Adeguatezza delle misure di sicurezza

Non è qualcosa che si possa aggiungere dopo, a sistema già progettato: va considerata fin dall'inizio.

## Pseudonimizzazione

Processo di trattamento dei dati personali in modo tale che i dati non possano più essere attribuiti ad un interessa specifico senza l'utilizzo di informazioni aggiuntive.

## PRINCIPI

I dati personali devono:

- essere tratti in modo lecito e trasparente.
- essere raccolti per determinate finalità (esplicite e legittime)
- essere adeguati, pertinenti e limitati a quanto necessario per la finalità
- essere esatti e aggiornati
- essere 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
- essere trattati in modo da garantire un'adeguata sicurezza

## Concetti Base

### SICUREZZA INFORMATICA

Salvaguardia dei sistemi informatici da potenziali rischi e/o violazione dei dati.  
Per garantirla bisogna preoccuparsi di :

- **Impedire** l'accesso ad utenti non autorizzati
- **Regolamentare** l'accesso ai diversi soggetti

È volta a proteggere l'informazione garantendone riservatezza, integrità, autenticità e disponibilità.

***“La forza di un sistema è pari alla forza dell’anello più debole che lo compone”***

### PROTEZIONE FISICA

Risulta essenziale che la vulnerabilità fisica non sia la più facilmente attaccabile.

### CRITTOGRAFIA

Serve a garantire riservatezza e, nel caso della asimmetrica, a dare identificazione, autenticazione e paternità.

## SICUREZZA DELLE PASSWORD

Aspetto sempre fondamentale per impedire l'accesso a utenti non autorizzati e nascondere/vincolare l'accesso a documenti riservati

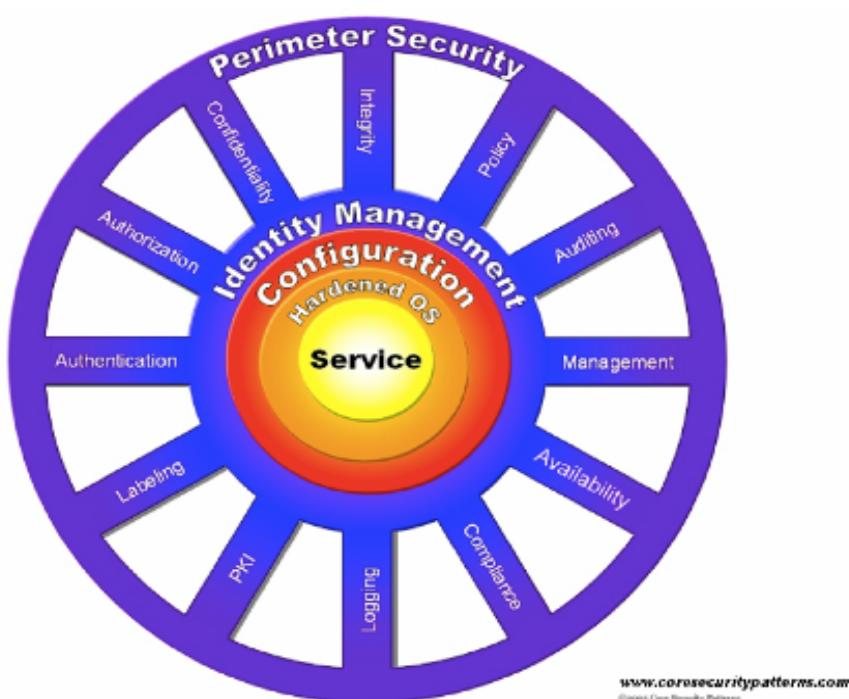
## PEC

Sistema di posta nel quale è fornita al mittente documentazione elettronica, con valenza legale, attestante l'invio e la consegna di documenti informatici.

## Analizzare e Progettare la Sicurezza

La definizione di una politica di sicurezza deve tenere conto di vincoli tecnici, logistici, amministrativi, politi ed economici, imposti dalla struttura organizzativa in cui il sistema opera. Proprio per questo la sicurezza deve essere introdotta fin dalle prime fasi di analisi dei requisiti. Bisogna definire un piano per la sicurezza.

Ogni componente funzionale del sistema deve essere reso sicuro.



La sicurezza di un sistema deve essere visto come una ruota di cui dobbiamo andare a garantire l'integrità in ogni sua parte. Il mozzo rappresenta l'applicazione da sviluppare mentre i 12 raggi sono i servizi di sicurezza 'core' applicabili a qualsiasi applicazione. Il bordo è il perimetro di sicurezza.

## Sistemi Critici

I sistemi critici sono sistemi tecnici da cui dipendono persone o aziende, se questi non forniscono i loro servizi nel modo giusto possono verificarsi seri problemi.

- Sistemi **safety-central**: I fallimenti possono provare incidenti, perdita di vite umane o seri danni ambientali
- Sistemi **mission-critical**: I malfunzionamenti posso causare il fallimento di alcune attività
- Sistemi **business-critical**: I fallimenti possono portare a costi molto alti

La proprietà più importante di un sistema critico è la:

$$\text{fidatezza} = \text{disponibilità} + \text{affidabilità} + \text{sicurezza}$$

I componenti che rendono un sistema critico sono sia SW che HW

## ASPETTI ORGANIZZATIVI

Diventa necessaria un'appropriata politica di formazione e sensibilizzazione degli utenti allo scopo di non rendere vani gli sforzi affrontati per creare un'adeguata infrastruttura.

## DOCUMENTAZIONE

**Business Impact Analysis:** tramite l'analisi dei rischi si cerca di stimare i danni di un eventuale attacco

**Security Policy:** definisce le politiche di sicurezza

**Security Plan:** implementa le regole della security policy

**Disaster Recovery Plan:** norme da assumere in caso di attacco

**Security Audits:** livello corrente di sicurezza

# Security Engineering

L'ingegneria della sicurezza è 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 delle loro contromisure. Gli ingegneri devono garantire che il sistema sia progettato per resistere agli attacchi.

## GESTIONE DELLA SICUREZZA

- Gestione degli utenti e dei permessi
- Deployment e mantenimento del sistema
- Monitoring degli attacchi, rilevazione e ripristino

## GLOSSARIO

**Bene:** risorsa del sistema da proteggere

**Esposizione:** possibile perdita o danneggiamento

**Vulnerabilità:** debolezza nel sistema SW

**Attacco:** sfruttamento di una vulnerabilità

**Minaccia:** circostanza che potrebbe causare danni

**Controllo:** misura protettiva che riduce una vulnerabilità

## TIPI DI MINACCE

- Minacce alla riservatezza del sistema o dei suoi dati
- Minacce all'integrità del sistema o dei suoi dati
- Minacce alla disponibilità del sistema o dei suoi dati

## TIPI DI CONTROLLO

- Controlli per garantire che gli attacchi non abbiano successo
- Controlli per identificare e respingere attacchi
- Controlli per il ripristino

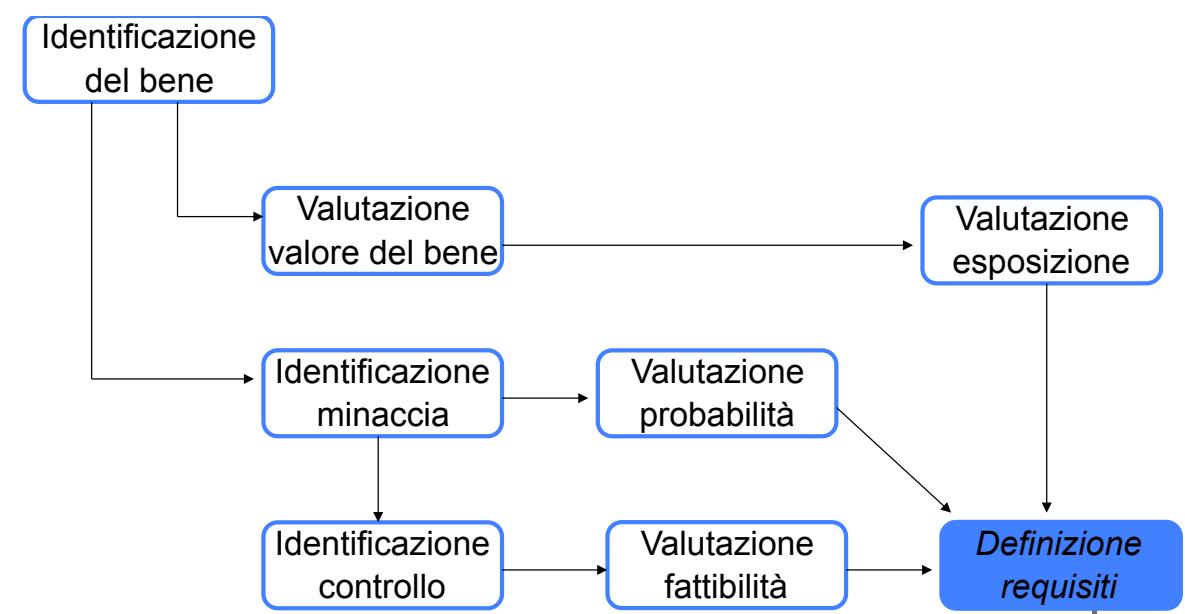
# Analisi del Rischio

L'analisi del rischio si occupa di valutare le perdite che un attacco può causare e bilanciare quest'ultime con i costi richiesti per la protezione dei beni stessi

La valutazione del rischio è un processo che si divide in due fasi:

- Valutazione preliminare del rischio
- Ciclo di vita della valutazione

## Valutazione Preliminare del Rischio



### IDENTIFICAZIONE DEL BENE

Analisi delle risorse logiche, fisiche classificandole e definendo le loro dipendenze critiche

### IDENTIFICAZIONE DELLE MINACCIE

In questa fase si cerca di definire quello che non deve poter accedere al sistema come attacchi intenzionali, attacchi a livello logico ed eventi accidentali

### VALUTAZIONE DELL'ESPOSIZIONE

Ad ogni minaccia occorre associare un rischio così da indirizzare l'attività di individuazione delle contromisure verso le aree più critiche

### VALUTAZIONE DELLA PROBABILITÀ

Si cerca di stimare la probabilità di occorrenza di attacchi intenzionali

### IDENTIFICAZIONE DEL CONTROLLO

Occorre scegliere il controllo da adottare per neutralizzare gli attacchi individuati

### VALUTAZIONE DI FATTIBILITÀ

Valuta il grado di adeguatezza di un controllo

## Ciclo di Vita della Valutazione del Rischio

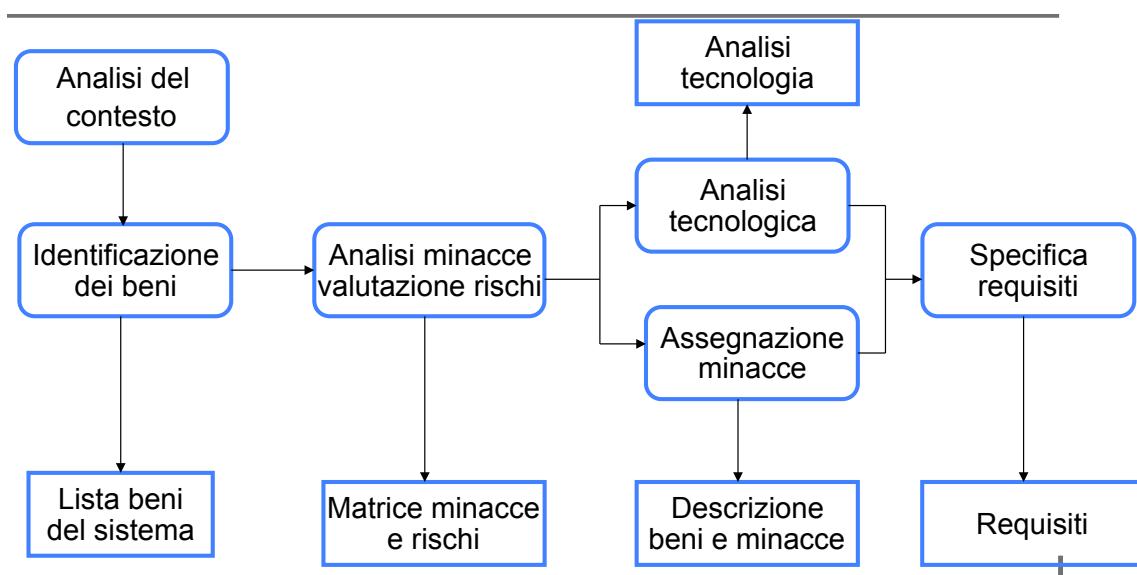
È necessaria la conoscenza dell'architettura del sistema e dell'organizzazione dei dati.  
La valutazione del rischio dovrebbe essere parte di tutto il ciclo di vita del software:  
dall'ingegnerizzazione dei requisiti al deployment del sistema.

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.

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

## Specifiche dei Requisiti di Sicurezza

Non è possibile specificare i requisiti associati alla sicurezza in modo quantitativo, quasi sempre questa tipologia di requisiti è espressa nella forma “non deve”.



### ANALISI DEL CONTESTO

In questa fase si studiano le finalità della struttura dell'organizzazione

### CATEGORIA REQUISITI DI SICUREZZA

- Requisiti di identificazione
- Requisiti di autenticazione
- Requisiti di autorizzazione
- Requisiti di immunità
- Requisiti di integrità
- Requisiti di scoperta delle intrusioni
- Requisiti di non-ripudiazione
- Requisiti di riservatezza
- Requisiti di controllo della protezione
- Requisiti di protezione della manutenzione del sistema

## **Security Use Case e Misuse Case**

I misuse case si concentrano sulle interazione tra l'applicazione e gli attaccanti che cercano di violarla.

La condizione di successo del misuse case è l'attacco andato a buon fine, questo li rende particolarmente adatti ad analizzare le minacce ma non molto utili per i determinare i 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.

I casi d'uso non dovrebbero mai specificare meccanismi di sicurezza che devono essere lasciate alla progettazione. Bisogna quindi pensarli evitando di specificare vincoli progettuali non necessari.

# PROGETTAZIONE PER LA SICUREZZA

La sicurezza non può essere aggiunta al sistema ma deve essere progettata insieme ad esso. Anche se sicurezza è anche un problema implementativo oltre che strutturale poiché spesso le vulnerabilità sono introdotte durante la fase di implementazione: è possibile ottenere un'implementazione non sicura da una progettazione sicura.

## Progettazione Architetturale

La scelta dell'architettura del sistema influenza profondamente la sicurezza: un'architettura inappropriata non garantisce riservatezza, integrità ed il livello di disponibilità necessario.

Due problemi fondamentali sono:

- Protezione
- Distribuzione

I due sono potenzialmente in conflitto.

Tipicamente la migliore architettura per fornire un alto grado di protezione è quella a layer dove i beni critici sono posizionati al livello più basso.

Per migliorare la protezione, inoltre, sarebbe bene che le credenziali di accesso ai diversi livelli fossero diverse fra loro.

Se la protezione dei dati è un requisito critico la soluzione migliore sarebbe usare un'architettura client/server con i meccanismi di sicurezza sulla macchina server.

Questa architettura porta con sé delle vulnerabilità che possono essere risolte con un'architettura distribuita con i server replicati in diversi punti della rete.

Tipico problema: lo stile architettura più appropriato per la sicurezza potrebbe essere in conflitto con altri requisiti, spesso soddisfare tutti i requisiti nella stessa architettura presenta molti problemi.

## Linee Guida di Progettazione

Non ci sono regole rigide per ottenere un sistema sicuro.

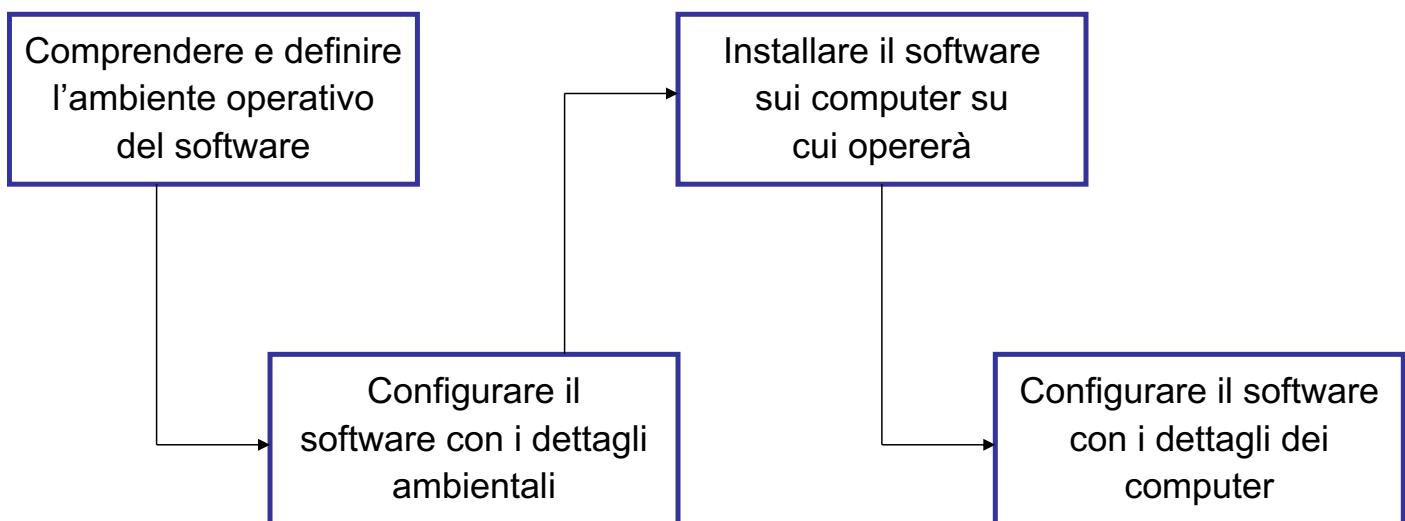
Differenti tipi di sistema richiedono differenti misure tecniche per ottenere un livello di sicurezza accettabile.

Esistono comunque linee guida generali di ampia applicabilità per la progettazione di sistemi sicuri.

1. **Basare le decisioni della sicurezza su una esplicita politica**: la Security Policy è un documento di alto livello che definisce cosa è la sicurezza e non come ottenerla
2. **Evitare un singolo punto di fallimento**: nei sistemi critici è buona norma di progettazione quella di cercare di evitare un singolo punto di fallimento, questo significa che non ci si dovrebbe affidare a un singolo meccanismo per assicurarla.
3. **Fallire in modo certo**: qualche tipo di fallimento è inevitabile ma si dovrebbe sempre fallire in modo certo per controllare tutti i comportamenti del sistema
4. **Bilanciare sicurezza e usabilità**: ogni volta che si aggiunge una caratteristica di sicurezza al sistema questo inevitabilmente diventa meno usabile
5. **Essere consapevoli dell'esistenza dell'ingegneria sociale**: ovvero che c'è chi studia come convincere con l'inganno utenti accreditati al sistema a rilevare informazioni riservate

6. **Usare ridondanza e diversità riduce i rischi** : mantenere più di una versione del SW e dei dati che non dovrebbero usare la stessa piattaforma o tecnologia
7. **Validare tutti gli input** : prevenire SQL injection e simili
8. **Dividere in compartimenti i beni** : organizzare le informazioni nel sistema in modo che gli utenti abbiano accesso solo alle informazioni necessarie piuttosto che a tutte le informazioni del sistema.
9. **Progettare per il deployment** : bisogna progettare il sistema in modo che siano inclusi programmi di utilità per semplificare il deployment e verificare potenziali errori di configurazione
10. **Progettare per il ripristino** : bisogna sempre progettare il sistema con l'assunzione che gli errori di sicurezza possano accadere

## DEPLOYMENT DEL SOFTWARE



## MINIMIZZARE I PRIVILEGI DI DEFAULT

Il software deve essere progettato in modo tale che la configurazione di default fornisce i minimi privilegi essenziali, in questo modo vengono limitati i danni di un possibile attacco

## RIMEDIARE A VULNERABILITÀ

Bisogna includere meccanismi diretti per aggiornare il sistema e riparare le vulnerabilità di sicurezza che vengono scoperte

# Security Testing

Il test di un sistema gioca un ruolo chiave nel processo di sviluppo software e dovrebbe essere eseguito con molta attenzione.

Il test della sicurezza è un lavoro molto lungo, spesso più complesso dei test funzionali che vengono svolti normalmente.

## 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 applicando un insieme di tentativo di violazione del sistema basati sulle informazioni ottenute.

Esistono vari tool per scagionare e indagare un'applicazione.  
Questo test non prende in esame solo debolezze del codice ma vengono svolti test mirati anche a livello infrastrutturale.

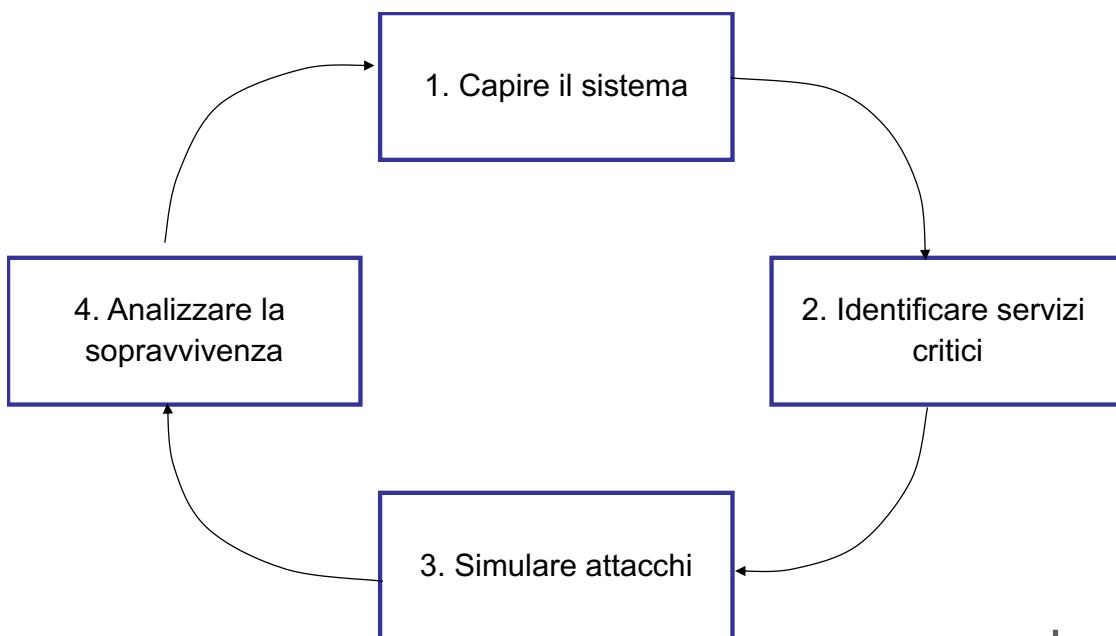
## White Box Testing

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 ed operano una revisione cercando possibili debolezze.  
I bachi tipici riguardano problemi di corsa critica e la mancanza di verifica dei parametri di input e sono specifici di ogni applicazione

## Capacita di Sopravvivenza del Sistema

Si intende la capacità del sistema di continuare a fornire i servizi essenziali agli utenti legittimi mentre è sotto attacco e dopo che parti del sistema sono state danneggiate come conseguenza di un attacco o di un fallimento.

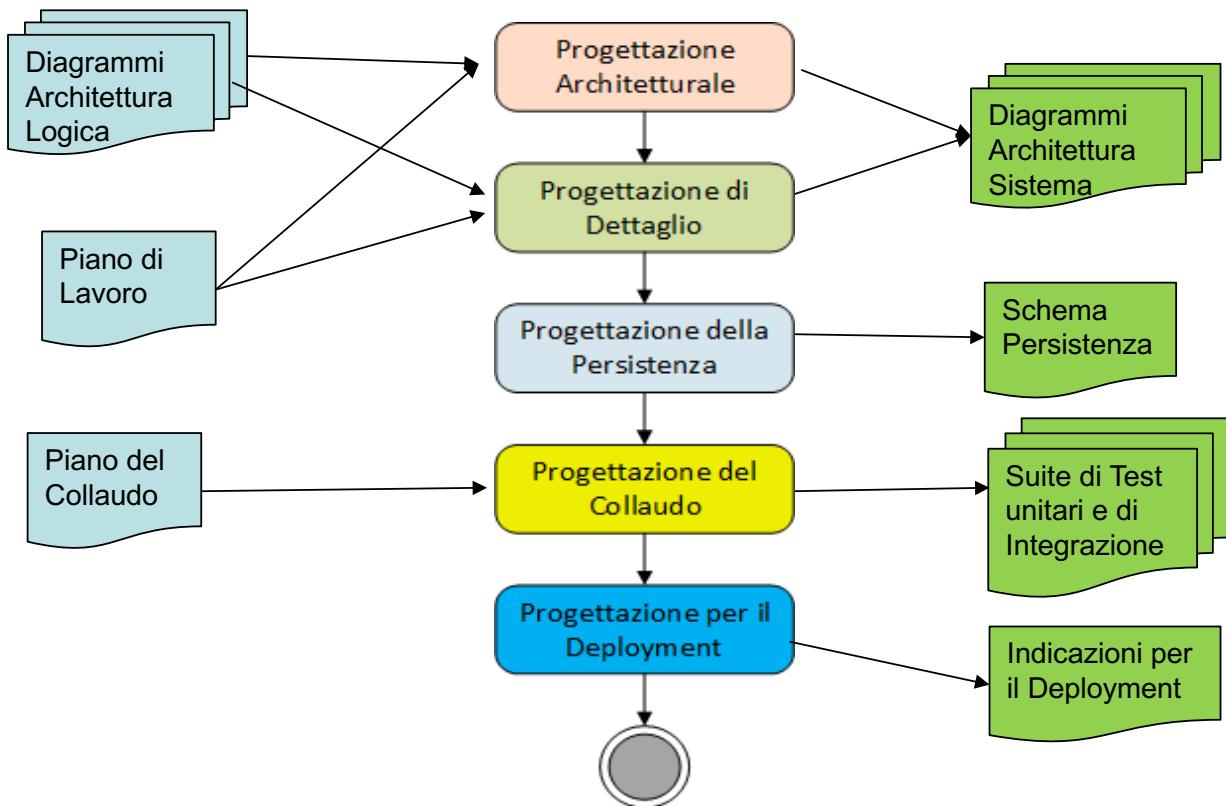
L'analisi e la progettazione delle capacità di sopravvivenza dovrebbero essere parte del processo di ingegnerizzazione dei sistemi critici.



Aggiungere le tecniche di sopravvivenza costa molto e spesso le aziende sono molto riluttanti ad investire sulla sopravvivenza, soprattutto se non sono mai state vittime di attacchi e perdite.

# PROGETTAZIONE

L'obiettivo della progettazione è quello di arrivare ad ottenere l'Architettura del Sistema attraverso una serie di raffinamenti successivi dell'architettura Logica. Vanno considerati anche utili tutti gli aspetti vincolanti che sono stati trascurati nelle fasi precedenti. La fase di progettazione deve mirare non solo ad individuare e descrivere una soluzione al problema, ma soprattutto a descrivere i motivi che hanno determinato questa soluzione.



## Progettazione Architetturale

Nella progettazione architetturale gli ingegneri devono prendere delle decisioni che influenzano profondamente il sistema.

Basandosi sulle proprie esperienze e conoscenze devono rispondere ad alcune domande fondamentali che li porteranno a scegliere la migliore architettura.

L'architettura del sistema influenza le prestazioni, la robustezza, la distribuibilità e la manutenibilità del sistema .

La struttura è tipicamente condizionata sia dalla tipologia di applicazione che si vuole realizzare sia dai requisiti non funzionali.

Ovviamente ci possono conflitti fra i requisiti non funzionali che rendono più complicato scegliere la giusta architettura.

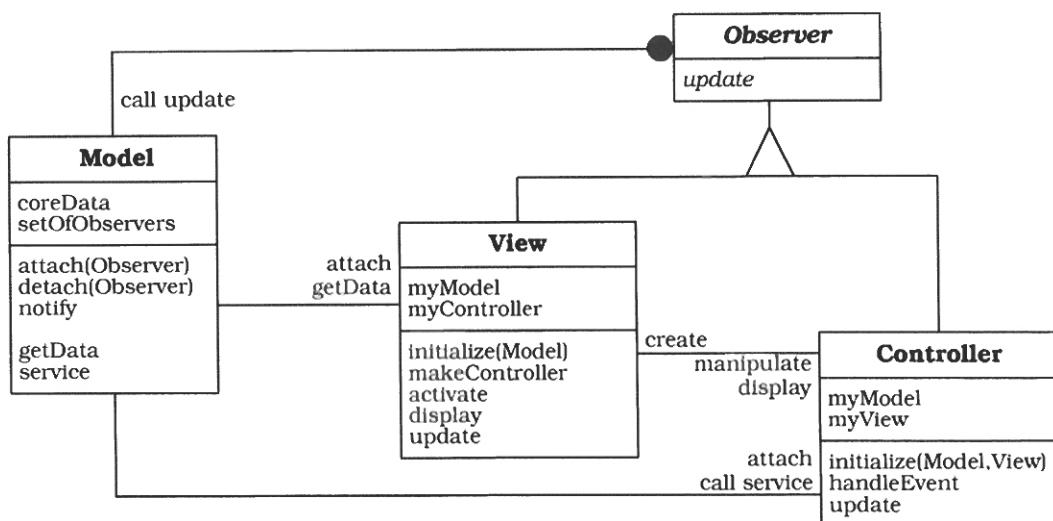
Seguono alcuni esempi di architettura fra i più utilizzati

## Blackboard

Il pattern Blackboard aiuta a strutturare quelle applicazioni in cui vengono applicate strategie di soluzione non deterministiche (IA).

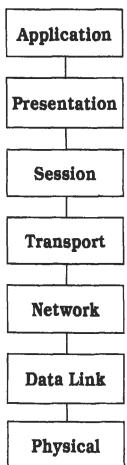
I diversi sotto-insieme condividono le stesse conoscenze attraverso la Blackboard al fine di costruire una soluzione.

## MVC



## Layer

Aiuta a strutturare quelle applicazioni che possono essere scomposte in gruppi di sotto-attività in cui ciascun gruppo si trova ad un ben definito livello di astrazione



## Client/Server

Questo pattern aiuta a strutturare un'applicazione come un insieme di servizi forniti da uno o più server ed un insieme di client che li utilizza

## **Broker**

Il pattern Broker può essere usato per strutturare sistemi distribuiti con disaccoppiamento tra i diversi sotto-insieme che comunicano tra loro attraverso remote server invocation.  
Il broker è responsabile della coordinazione delle comunicazioni fra le varie parti del sistema.

## **Pipe & Filters**

Questo pattern aiuta a strutturare quelle applicazioni che processano stream di dati. Ogni passo del processo è incapsulato in un apposito filtro ed i dati attraversano una pipe di filtri.

# **Progettazione di Dettaglio**

La progettazione di dettaglio va a definire il dettaglio dell'architettura del sistema nelle sue tre viste:

- Struttura
- Interazione
- Comportamento

Per realizzare un sistema funzionante occorre considerare GUI, DB, Framework, librerie, .... per avere un software estensibile e modulare.

È compito della progettazione di dettaglio identificare e definire classi in accordo alla specifica architettura scelta.

Bisogna mantenere massi indipendenza possibile dal linguaggio e dall'ambiente di programmazione , DMBS, S.O. e Hardware.

Queste specifiche devo essere tenute in conto solo se vincolanti.

# **MOD1**

# Fattori di Qualità del Software

## CORRETTEZZA

Data una definizione dei requisiti che il software deve soddisfare, il software si dice corretto se rispetta tali requisiti.

## ROBUSTEZZA

Il software si dice robusto se si comporta in maniera accettabile anche in corrispondenza di situazioni anomale e comunque non specificate nei requisiti.

Nel caso di situazioni anomale il software non deve causare disastri.

## AFFIDABILITÀ

Il software si dice affidabile se le funzionalità offerte corrispondono ai requisiti e in caso di guasto non produce né danni fisici né economici.

## FACILITÀ D'USO

Il software si dice facilmente utilizzabile in base alla facilità con cui l'utilizzatore è in grado di utilizzare il software. Tale caratteristica deve essere calibrata per chi è il destinatario ultimo del software.

## EFFICIENZA

L'efficienza di un software si calcola con lo studio della complessità che si effettua controllando parametri come il tempo di calcolo e l'occupazione della memoria principale.

## ESTENSIBILITÀ

Si definisce estensibile un software facilmente modificabile. Viene valutata in base alla semplicità architettura e alla modularità del sistema.

## RIUSABILITÀ

Il software è riusabile se può essere riutilizzato completamente, o in parte, in nuove applicazioni.

## VERIFICABILITÀ

Facilità con il software può essere sottoposto a test.

## PORATABILITÀ

Facilità con cui il prodotto software può essere trasferito su altre architetture HW o SW.

# Principi OO

ADT (Abstract Data Type): dati + codice che opera sui dati = interfaccia (visibile) + implementazione (nascosta)

Due principi fondamentali :

- Information Hiding
- Incapsulamento

Un ADT nasconde ai suoi utilizzatori tutti i dettagli della sua struttura interna e del suo funzionamento cosicché chi lo utilizza lo possa fare senza conoscere necessariamente i dettagli ma gli bastano gli input e gli output di cui l'ADT ha bisogno.

Questo permette di modificare un componente senza ripercuotere i cambiamenti del codice di un ADT su tutto il software sviluppato.

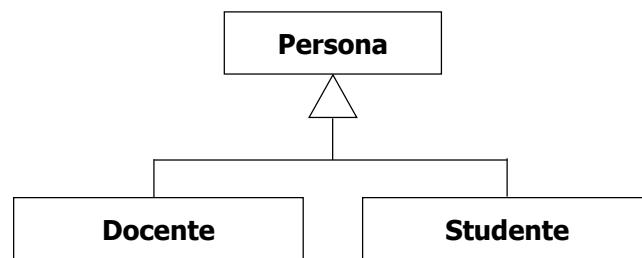
## Ereditarietà

Attributi e operazioni comuni vengono specificati una volta sola, l'obiettivo è quello di semplificare la definizione e la realizzazione di tipi di dato simili.

- Model Inheritance: il modello di qualcosa è modello anche di altro
- Software Inheritance: il codice di una classe è usato da un'altra, non riguarda obbligatoriamente il modello

### EREDITARIETÀ DI INTERFACCIA E DI ESTENSIONE

Implementa meccanismi di compatibilità fra tipi: una sottoclass è un sottotipo compatibile con tutti i tipi definiti lungo la sua catena ereditaria.



### EREDITARIETÀ DI REALIZZAZIONE

Consiste praticamente nel copiare il codice di una classe in un'altra anche per scopi diversi e rendendolo privato nella classe che eredita.

## Polimorfismo

È la capacità della stessa cosa di apparire in forma diverse e in contesti diversi e di cose diverse di apparire sotto la stessa forma in un determinato contesto

- Overriding
- Binding dinamico
- Virtual Method Table

## Regole di Naming in .NET

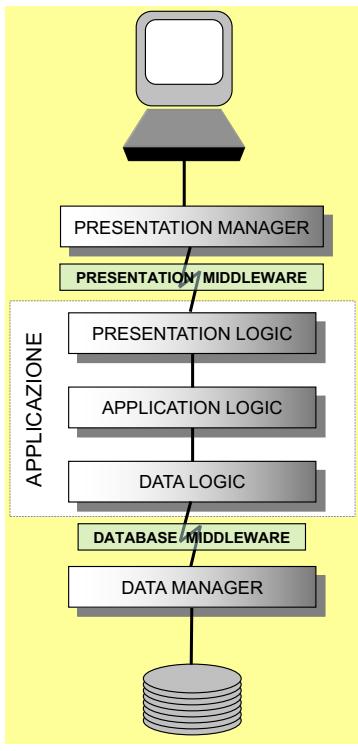
I nomi delle classi devono iniziare con una lettera maiuscola, indicare al singolare un oggetto della classe oppure indicare al plurale gli oggetti contenuti nella classe.

## RELAZIONI

- Generalizzazione/Ereditarietà
- Realizzazione
- Associazione (aggregazione e composizione)
- Dipendenza (istanza-classe e classe-metaclasse)

A ha bisogno di B per lo svolgimento di alcune funzionalità che A non è in grado di effettuare quindi A non funziona correttamente se B non è corretto.

## Struttura di una Applicazione



**Presentation Manager** gestisce l'interazione con l'utente (schermo, tastiera, mouse, ...) comprendete le interfacce grafiche (GUI)

**Data Manager** gestisce la persistenza di qualsiasi tipo essa sia

**Presentation Logic** gestisce l'interazione con l'utente a livello logico, crea gli output, prende gli input e li valida e gestisce gli errori

**Application Logic** è la logica di controllo dei componenti

**Data Logic** gestisce la persistenza a livello logico: consistenza dei dati, istruzione SQL, r/w su file, gestione errori

**Middleware** è il software che permette la comunicazione tra processi e API che isolano il codice dell'applicazione dai sottostanti formati e protocolli.

**Presentation Middleware** emula terminali alfanumerici

**Database Middleware** si occupa di trasferire sulla rete le richieste SQL e i dati del DBMS all'applicazione

**Application Middleware** gestisce la comunicazione tra due componenti della stessa applicazione dando meno vincoli progettuali

## Introduzione al Framework .NET

### Tecnologia COM

COM è una piattaforma indipendente, distribuita ed OO per creare componenti software. Definisce uno standard per far interagire oggetti COM con altri oggetti.

La specifica COM necessita del reference counting per assicurarsi che oggetti singoli rimangano "in vita" finché hanno qualcuno che ha accesso ad una o più delle sue interfacce e che vengano eliminati quando non c'è nessuno che ha più bisogno di loro.

L'ereditarietà è solo con composizione e delega.

La posizione di ogni componente è salvata nel Windows Registry e può esserci solo una versione di un certo componente installato.

## Framework .NET

È composto da un ambiente di esecuzione e da una libreria di classi.

Si occupa di semplificare lo sviluppo e il deployment, di aumentare l'affidabilità del codice ed unifica il modello di programmazione.

È completamente indipendente da COM ma è fortemente integrato con esso.

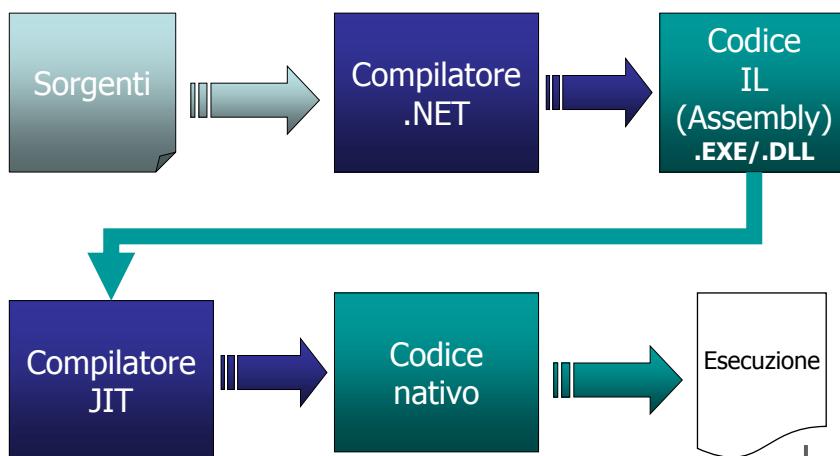
È un ambiente OO in cui qualsiasi entità è un oggetto

Le funzionalità del framework sono disponibili a tutti i linguaggi .NET e i componenti della stessa applicazione possono essere scritti in linguaggi diversi, in questo caso è comunque supportata l'ereditarietà.

.NET è un'implementazione di CLI, questo permette che applicazioni scritte in linguaggi di alto livello possano essere eseguite in diversi sistemi senza la necessità di riscrivere l'applicazione.

Concetti chiave:

- Intermediate Language IL
- Common Language Runtime
- Common Type System
- Common Language Specification



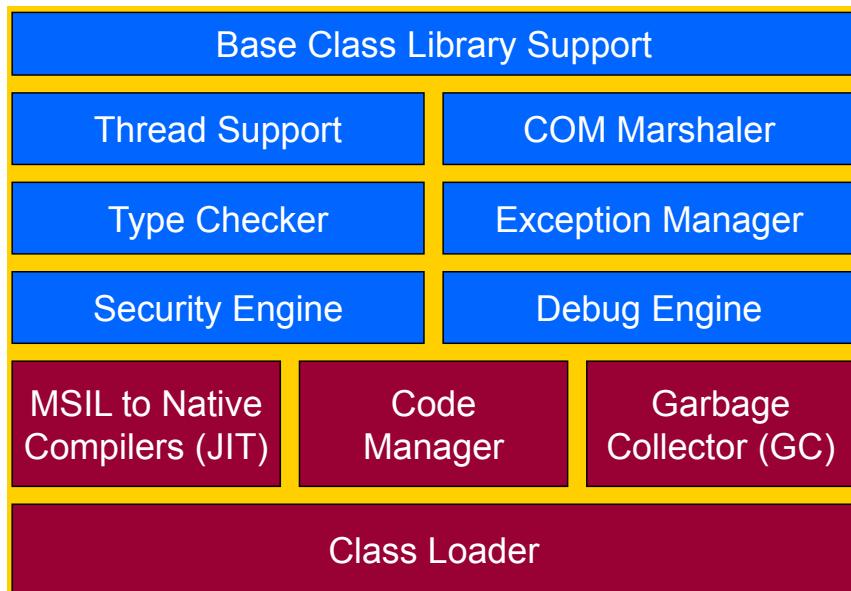
## ASSEMBLY



È l'unità minima per la distribuzione ed il versioning.  
Normalmente è composto da un solo file ma può essere diviso in moduli

## Common Language Runtime

Offre vari servizi:



## GARBAGE COLLECTOR

Gestisce il ciclo di vita di tutti gli oggetti .NET, si occupa di distruggere automaticamente gli oggetti quando non sono più referenziati.

A differenza di COM non si basa sul Reference Counting, queste da maggiore velocità di allocazione e consente riferimenti circolari.

In fase di inizializzazione di un processo il CLR riserva una reigione contigua di spazio di indirizzamento managed heap e memorizza un puntatore all'indirizzo di partenza della regione.

Il garbage collector verifica se nell'heap esistono oggetti non più utilizzati dall'applicazione.  
Gli oggetti vivi sono quelli raggiungibili direttamente o indirettamente dalle radici  
Gli oggetti garbage sono quelli non più raggiungibili in nessun modo.

Quando parte ipotizza che tutti gli oggetti siano garbage quindi scansiona le radice e per ognuna marca l'eventuale oggetto referenziato e tutti gli oggetti a loro volta raggiungibili da esso.

Finita la scansione tutti gli oggetti non marcati sono veramente garbage.

Quindi rilascia la memoria utilizzata da essi, compatta la memoria ancora in uso e aggiorna i puntatori.

Se un oggetto contiene almeno un riferimento ad un oggetto unmanaged (una risorsa del S.O. generalmente) è necessario eseguire del codice per rilasciare la risorsa prima della deallocazione dell'oggetto, quindi ciò deve essere fatto dal programmatore.

Il metodo Finalize viene invocato dal garbage collector in modo non deterministico quindi è fondamentale utilizzare dispose o close in modo deterministico.

Il pattern Dispose ci permette di far rilasciare la risorsa in modo deterministico dal compilatore tramite la clausola using e l'utilizzo dell'interfaccia IDisposable.

## GESTIONE DELLE ECCEZIONI

In CLR un'eccezione è un oggetto che eredita dalla classe System.Exception  
Può essere generata dal programma in esecuzione e dall'ambiente di runtime

## Common Type System

Alla base di tutto ci sono classi, strutture, interfacce, enumerativi e delegati.  
.NET è fortemente tipizzato ed OO

## Common Language Specification

Definisce le regole di compatibilità tra linguaggi dando regole per gli identificatori, la denominazione di proprietà ed eventi, per i costruttori degli oggetti, ...

Tutto è un oggetto e viene diviso in due tipi:

- Tipi riferimento: riferimenti ad oggetti allocati sull'heap
- Tipi valore: allacciati sullo stack o parte di altri oggetti

## Tipi in .NET

Dal punto di vista del modo in cui le istanze vengono gestite in memoria i tipi posso essere distinti in Reference Type e Value Type

### REGOLE

- Information hiding a livello di assembly
- Information hiding a livello di classe
- Information hiding a livello di field

Costanti: il nome dovrebbe iniziare con una lettera maiuscola e solitamente deve essere pubblica

Field: il nome deve iniziare con ‘\_’ seguito da lettera minuscola e deve essere privato

Field read-only: une delle due convenzioni precedenti

### PASSAGGIO DEGLI ARGOMENTI

Value type viene passata una copia dell'oggetto ed eventuali modifica vengono effettuate su copia senza aver effetto sull'originale

Reference Type viene passata una copia del riferimento all'oggetto ed eventuali modificheranno effetto sulla copia e non sul riferimento originale.

## Delegati ed Eventi

### Delegati

I delegati sono oggetti che possono contenere il riferimento a un metodo tramite il quale il metodo stesso può essere invocato.

Gli oggetti funzione (funtori) sono oggetti che si comportano come una funzione, sono simili ai puntatori a funzione del C/C++

Esempio di dichiarazione di un delegato: delegate int Azione(int param);

Esempio di definizione di un delegato: public Azione azione;

Esempio di inizializzazione di un delegato: azione = obj.nomeMetodo;

Esempio di invocazione di un metodo dal delegato: int y = azione(10);

È possibile assegnare al delegato una lista di metodi. All'atto della chiamata del delegato vengono invocati in sequenza e in modo sincrono tutti i metodi assegnati al delegato.

```
azione = fun1; azione += fun2; azione += fun3;
```

```
Int x = azione(10); //vengono chiamati fun1, fun2 e fun3
```

In questo caso i metodi assegnati ad azione avranno tutti 10 in input ed x conterrà il risultato dell'ultimo metodo applicato, in questo caso fun3.

```
azione -= fun3; //azione non contiene più il riferimento a fun3
```

Un'istanza di un delegato contiene, quindi, uno o più metodi referenziati come callable entity. Per un metodo statico consiste semplicemente in un metodo, per un'istanza di un metodo consiste in un'istanza e un metodo su quell'istanza.

I delegati sono perfetti per l'invocazione anonima.

## Eventi

Gli eventi automatizzano il supporto per la registrazione dei metodi ai delegati e l'implementazione privata.

Gli eventi facilitano la scrittura del codice rispetto ai delegates e risultano più "puliti"

Un evento può essere scatenato dall'interazione con l'utente o dalla logica del programma.

**Event sender:** l'oggetto o la classe che scatena l'evento

**Event receiver:** l'oggetto o la classe che è notificato quando l'evento di verifica

**Event handler:** il metodo 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.

Un evento, sostanzialmente, incapsula un delegato.

Per convenzione gli event delegate in .NET hanno due parametri: source e data.

Una volta dichiarato l'evento può essere trattato come un delegato speciale.

## Interfaccia Utente

System.Windows.Forms contiene le classi per creare applicazioni basate su finestre.

La classe Application (statica) contiene metodi che sono fondamentali per gestire l'infrastruttura dell'applicazione a finestre.

System.Drawing contiene oggetti grafici base.

La classe Control è la classe base dei controller dei form, contiene metodi e attributi che definiscono l'aspetto e il comportamento della finestra.

# Metadati e Introspezione

## Metadati

***“Metadata is data that describes other data. For example, the definition of a class is metadata”***

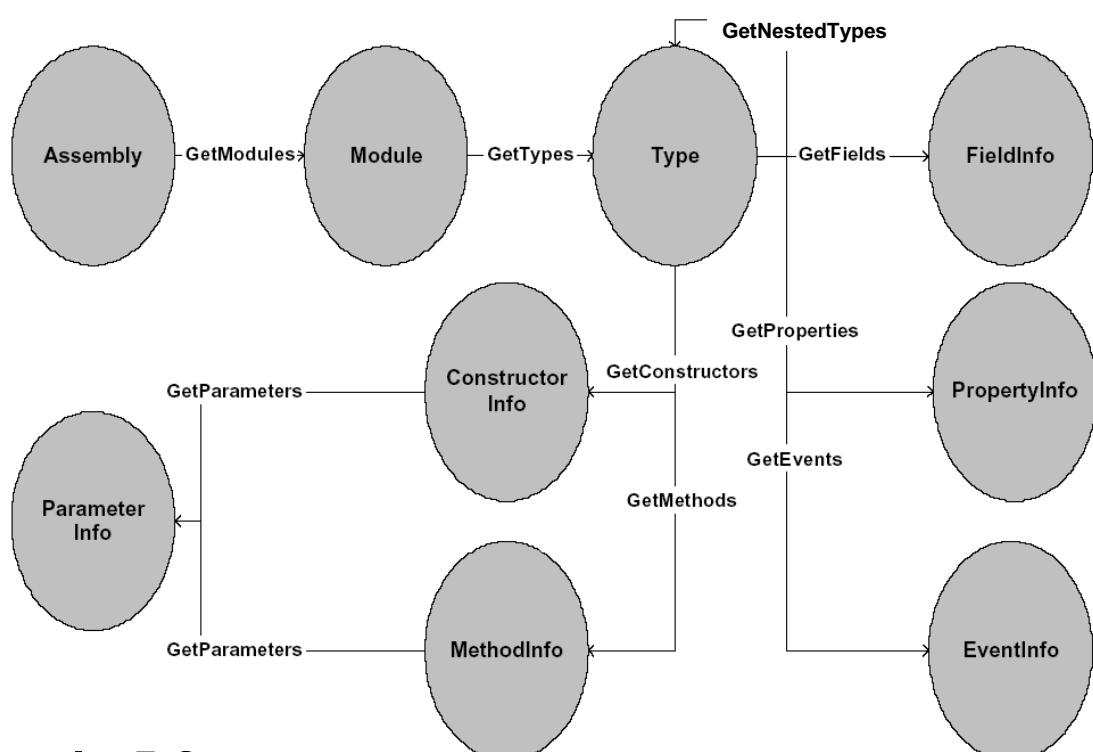
Si occupano di far sì che un componente abbia abbastanza informazioni per essere auto descritto.

I metadati in .NET sono generati dalla definizione del tipo, salvati con essa e utilizzabili a run-time.

## Reflection

La Reflection può essere usata per esaminare i dettagli di un’assembla, instanziare oggetti e chiamare metodi scoperti a run time, per creare, compilare ed eseguire al volo.

System.Type è il punto focale della Reflection. Tutti gli oggetti e i valori sono istanze di tipo. Type è in grado di scoprire il tipo di un oggetto (o di un valore) e referenziare i tipi con nomi simbolici, gli stessi tipo sono istanze di Type.



## CUSTOM ATTRIBUTES

Sono il modo semplice per aggiungere informazioni ai metadati per ogni elemento dell'applicazione.

Possono essere usati in modo che i client possano automaticamente usare certe funzionalità visibili tramite reflection.

## Meta Programming in .NET

Può essere usato per creare dinamicamente nuove classi, inserirle in una struttura già esistente e istanziarle.

System.Reflection permette di creare assembly al volo.

# Design Principles

La Design Quality dipende dalle specifiche priorità organizzative.

Un buon design dovrebbe essere:

- Il più affidabile
- Il più efficiente
- Il più manutenibile
- Il più economico
- ...

Discuteremo soprattutto della mantenibilità del design.

Cosa rende un software non buono?

- Rigidità del software: difficoltà di modifica del SW
- Fragilità del software: tendenza del SW a rompersi in diverse parti ogni volta che cambia
- Immobilità del software: difficoltà di riusare il software per altri progetti
- Viscosità: tendenza ad incoraggiare cambiamenti del SW che cambiano l'intento iniziale

## Premessa

### IL PRINCIPIO ZERO

È un principe di logica noto come rasoio di Occam: non bisogna introdurre concetti che non siano strettamente necessari, in poche parole: "Quello che non c'è non si rompe"

### SEMPLICITÀ E SEMPLICISMO

La semplicità è un fattore importantissimo : il SW deve fare i conti con una notevole componente di complessità generata dal contesto quindi è importante non aggiungerne altra.

"Keep it as simple as possible but not simpler" bisogna essere semplici ma non semplicistici

### DIVIDE ET IMPERA

La decomposizione è una tecnica fondamentale per il controllo e la gestione di complessità: la qualità della progettazione dipende direttamente dalla qualità delle scelte di decomposizione adottate.

## **MAKE ALL OBJECT DATA PRIVATE**

Usare dati pubblici è sempre un grosso rischio di aprire i moduli.

## **The Single Responsibility Principle**

Non ci deve mai essere più di un motivo per una classe di cambiare. Una classe ha responsabilità singola: fa quello, lo fa bene e fa solo quello.

## **The Dependency Inversion Principle**

Ogni dipendenza dovrebbe avvenire tramite interfaccia o una classe astratta, nessuna dipendenza dovrebbe essere legata ad una classe concreta.

I moduli di basso livello contengono la maggior parte del codice e della logica di implementativa e quindi sono più soggetti a cambiamenti.

Se i moduli di alto livello dipendono dai dettagli dei moduli di basso livello i cambiamenti si propagano e le conseguenze sono: rigidità, fragilità e immobilità.

## **The Interface Segregation Principle**

I client non dovrebbero essere forzati a dipendere da interfacce che non utilizzano.  
Più interfacce specifiche sono meglio di una generale.

L'utilizzo di fat interface crea un'inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori.

## **The Open/Closed Principle**

Si dovrebbero definire moduli in modo tale che possano essere estesi senza modificarli.

È possibile creare astrazioni che rendono un modo immutabile ma rappresentino un gruppo illimitato di comportamenti.

Il segreto sta nell'utilizzo di interfacce: ad un'interfaccia immutabile possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi.

## **The Liskov Substitution Principle**

Le sottoclassi dovrebbero essere sostituibili per le loro classi base.

Quindi un client di una classe base dovrebbe continuare a funzionare bene se gli viene passata un'istanza di una classe derivata dalla base.

OCP si basa sull'uso di classi concrete derivate da astrazione, invece LSP costituisce una guida per creare queste classi concrete tramite ereditarietà.

La chiave per evitare le violazioni del principio di Liskov risiede nel Design by Contract

## **DESIGN BY CONTRACT**

Ogni metodo ha un insieme di pre-condizioni, requisiti minimi che devono essere soddisfatti dal chiamante perché il metodo possa essere eseguito correttamente, ed un insieme di post-condizioni, requisiti che devono essere soddisfatti dal metodo nel caso di esecuzione corretta.

Questi due insieme costituiscono un contratto tra il chiamante e il metodo.  
Quando un metodo viene ridefinito in una sottoclasse:

- le pre-condizioni devono essere identiche o meno stringenti
- Le post-condizioni devono essere identiche o più stringenti

## Principles of Package Architecture

### Release/Reuse Equivalency Principle

Un elemento riutilizzabile non può essere riutilizzato almeno che non sia gestito da una release o qualcosa di simile.

Un Cliente non avrebbe riusare un elemento almeno che non ci sia tracciabilità di version number e il mantenimento delle vecchie versioni.

### The Common Closure Principle

Classi che si modificano insieme appartengono allo stesso package.

### The Common Reuse Principle

Le classi che non vengono riusate insieme non dovrebbero stare nello stesso package.

## Relationships between Packages

### Acyclic Dependencies Principle

Le dipendenze fra package non devono creare cicli, altrimenti la modifica di un package crea un ciclo “infinito” di necessità di modifica fra i vari package.

### Stable Abstractions Principle

I pacchetti stabili dovrebbero essere astratti.

### Stratification Principle

In un sistema organizzato a livelli il diagramma dei package è aciclico: non esistono riferimenti circolari.

L'effetto delle dipendenze circolari è quello di ridurre la riusabilità del codice.

### LE DIPENDENZE DI CREAZIONE

La tecnica più semplice per eliminare la dipendenza fra classi concrete consiste nell'interposizione di interfacce, questo basa nel caso delle relazione di uso ma non se abbiamo una relazione di composizione in cui una classe deve anche provvedere alla creazione di una di livello inferiore.

Una possibile regola è : se la classe A crea oggetti di classe B, la classe B dovrebbe essere final oppure il metodo in cui viene creato dovrebbe essere ridefinibile.

## Interface Segregation Principle

Più interfacce specifiche sono meglio di una generale.

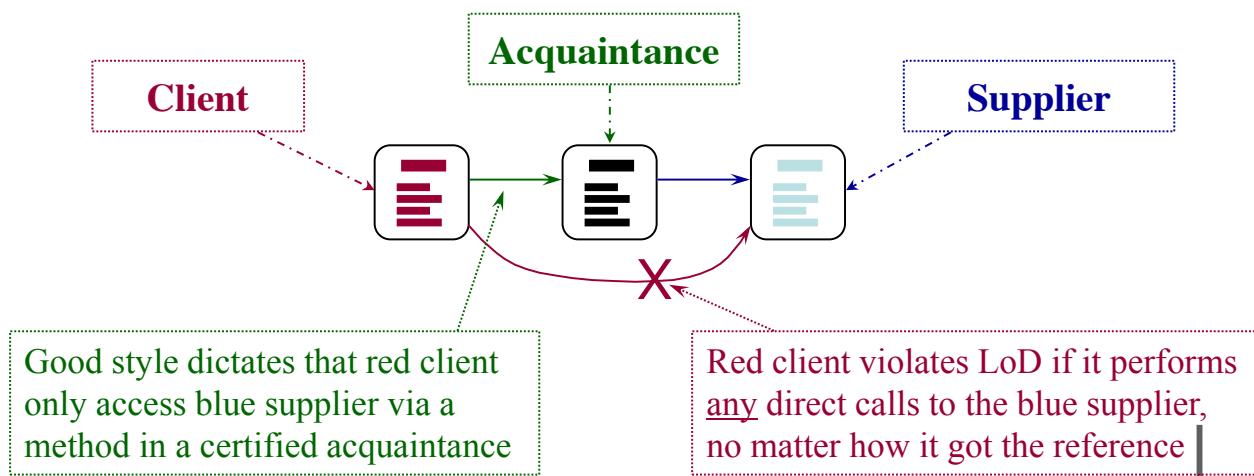
I Client non dovrebbero essere forzati a dipendere da interfacce che non usano.

## Narrow Inheritance Interface

Comportamenti suddivisi su più metodi dovrebbero essere basati su un set minimo di metodi che devono essere sovrascritti (override).

## The Law of Demeter

Qualsiasi oggetto che riceve un messaggio in un metodo deve essere uno di un ristretto set di oggetti.



La legge può essere violata solo per ottimizzare il codice.

## The Common Reuse Principle

Tutte le classi in un package devo essere riusate insieme. Se ne riusi solo una, le riusi tutte.

# Design Pattern

Ogni pattern descrive un problema che si ripete spesso e descrive il core della soluzione al problema.

L'obiettivo dei design pattern è, quindi, quello di risolvere problemi progettuali specifici e rendere i progetti OO più flessibili e riutilizzabili.

Ogni design pattern cattura e formalizza l'esperienza acquisita nell'affrontare e risolvere i problemi permettendo di riutilizzare tale esperienza in altri casi simili.

I design pattern sono composti da quattro elementi fondamentali:

- **Nome:** identifica il pattern
- **Problema:** descrive quando applicarlo
- **Soluzione:** descrive il pattern, ovvero gli elementi che lo compongono e le loro relazioni, responsabilità e collaborazioni
- **Conseguenze:** descrive vantaggi e svantaggi dell'applicazione

## Classificazione dei Design Pattern

- **Pattern di creazione:** risolvono problemi inerenti al processo di creazione di oggetti
- **Pattern strutturali:** risolvono problemi inerenti la composizione di classi o di oggetti
- **Pattern comportamentali:** risolvono problemi inerenti le modalità di interazione e di distruzione delle responsabilità tra classi o tra oggetti

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

## Pattern SINGLETON

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

La classe deve tenere traccia della sua sola istanza, intercettare tutte le richieste di creazione e fornire un modo per accedere all'istanza unica.

**Alternativa:** classe non instanziabile con soli membri statici.

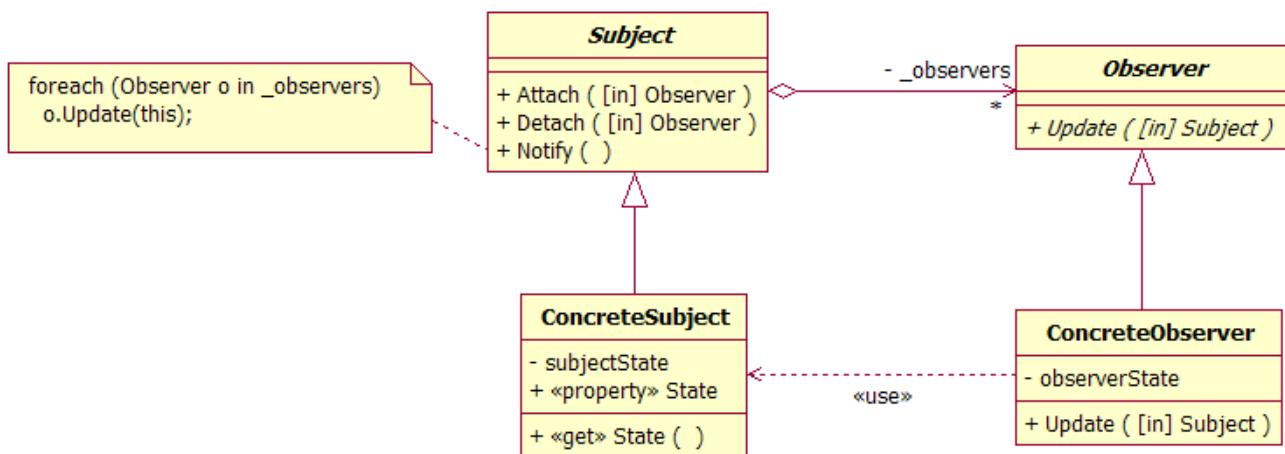
Perché un singleton? Il singleton può implementare 1 o più interfacce e può essere specializzato ed è possibile creare nella GetInstance un'istanza specializzata che dipende dal contesto corrente.

## Pattern OBSERVER

Serve nei contesti in cui la modifica in un oggetto (soggetto) richiede che altri oggetti (observers) siano aggiornati.

Questa relazione può essere codificata esplicitamente nel soggetto, ma richiede conoscenze specifiche su come gli observers debbano essere modificati, rendendo difficilmente riutilizzabili.

La soluzione è creare una relazione 1:N tra il soggetto e gli observers in modo che la modifica del soggetto sia notificata agli observers che si modificano da soli.



Vedi esempio Boss-Worker su slide 14 in poi del pacco Design Pattern

## Pattern MVC

### Modello

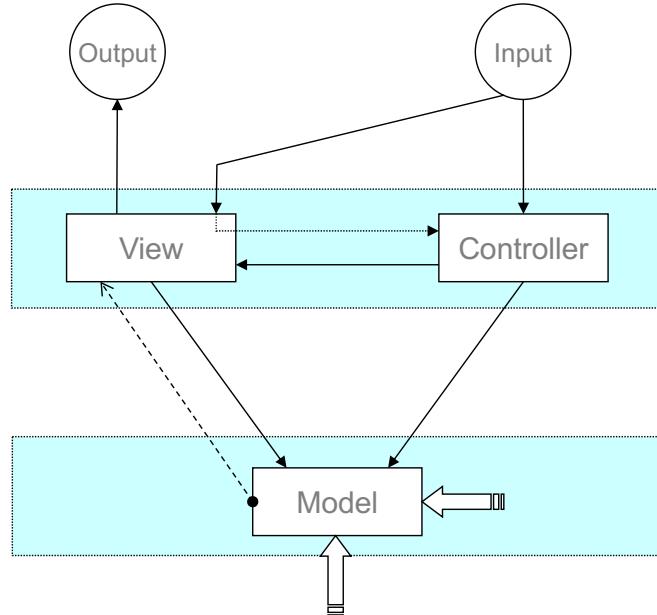
Gestisce un insieme di dati logicamente correlati. Risponde alle interrogazioni sui dati, risponde alle istruzioni di modifica dello stato, genera un evento quando lo stato cambi, registra gli oggetti interessati alla notifica dell'evento.

### View

Gestisce un'area di visualizzazione, nella quale presenta all'utente una vista dei dati gestiti dal modello, si registra presso il modello per ricevere l'evento di cambiamento di stato.

## Controller

Gestisce gli input dell'utente, mappa le azioni dell'utente in comandi e invia tali comandi al modello e/o alle view che effettuano le operazioni appropriate



## Pattern FLYWEIGHT

Descrivere come condividere oggetti leggeri in modo tale che il loro uso non sia troppo costoso.

Un flyweight è un oggetto condiviso che può essere utilizzato simultaneamente ed efficientemente da più clienti, non deve essere distinguibile da un oggetto non condiviso e non deve fare ipotesi sul contesto nel quale opera.

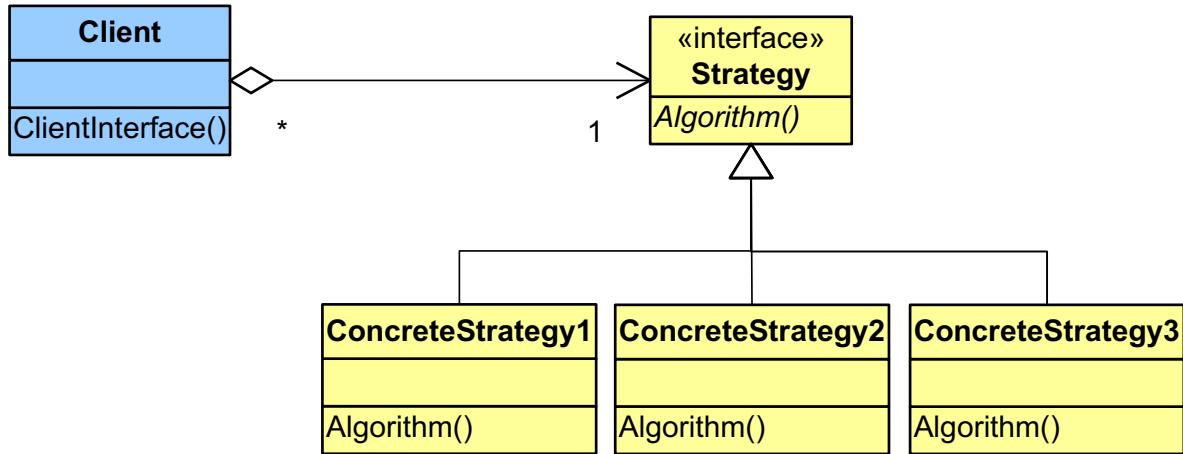
Per assicurare un corretta condivisione i clienti non devono stanziare direttamente i flyweight ma devono ottenerli tramite una factory.

Distinzione tra stato intrinseco e stato estrinseco

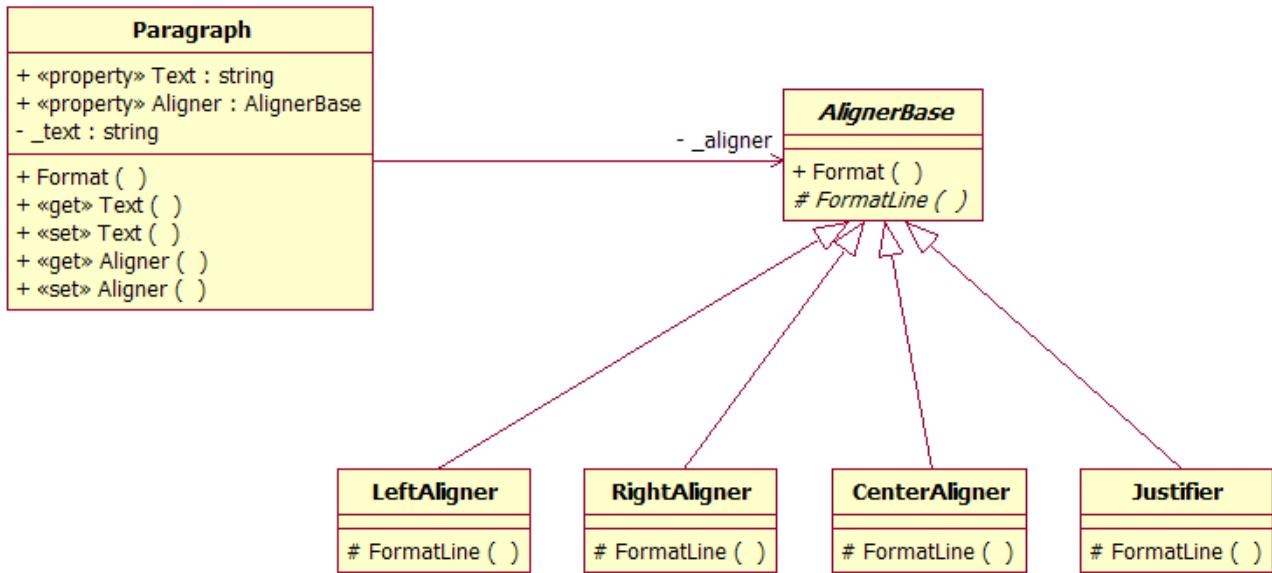
- **Stato intrinseco**: non dipende dal contesto di utilizzo e quindi può essere condiviso da tutti i clienti
- **Stato estrinseco**: dipende dal contesto di utilizzo e quindi non può essere condiviso dai clienti

## Pattern STRATEGY

Permette di definire un insieme di algoritmi tra loro correlati, ncapsulare tali algoritmi i una gerarchia di classi e renderli intercambiabili

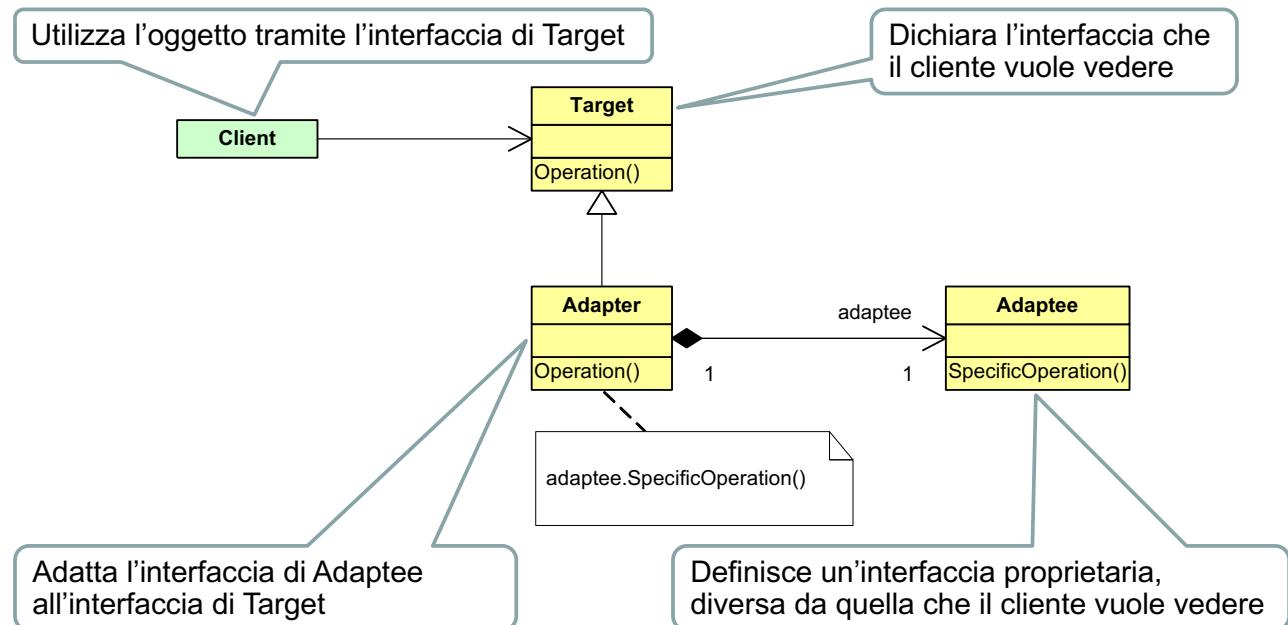


Esempio:



## Pattern ADAPTER

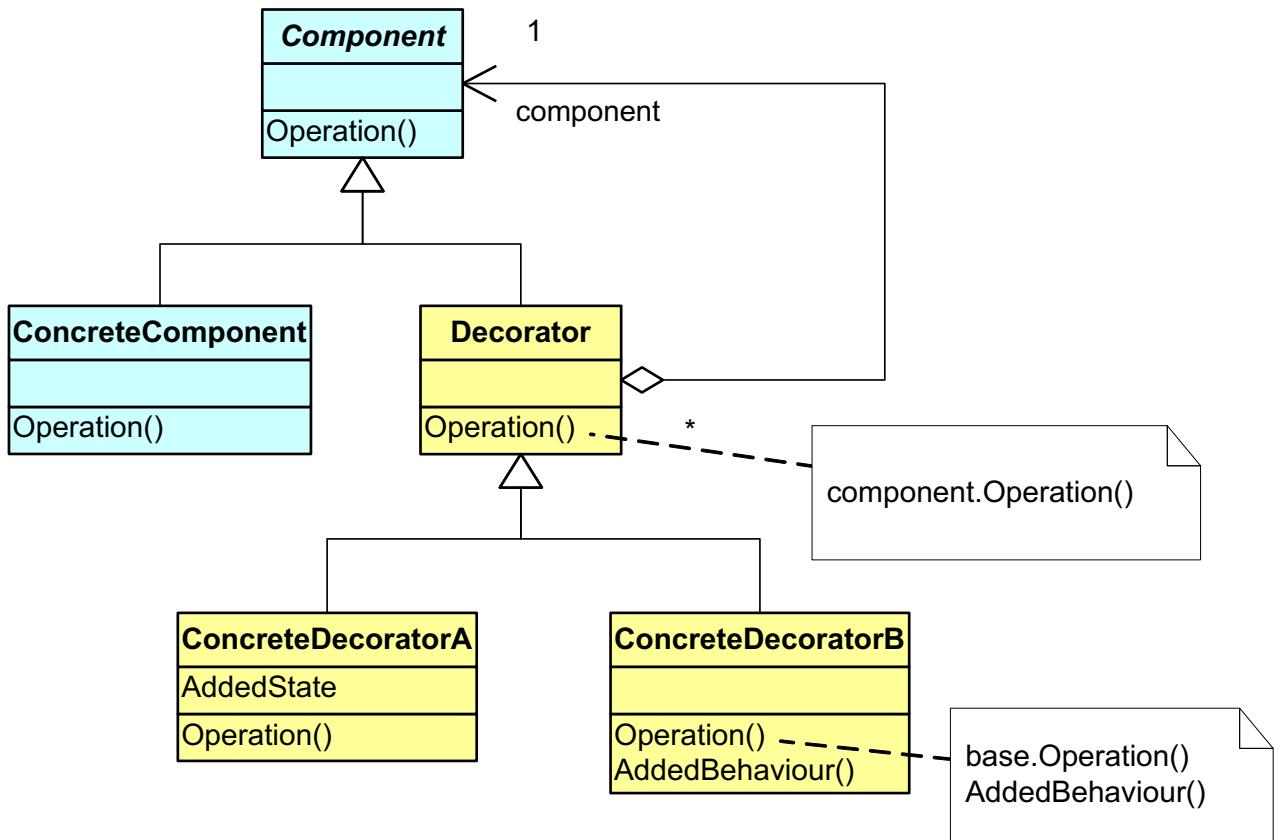
Converte l'interfaccia originale di una classe nell'interfaccia che si aspetta il cliente.  
Permette a classi che hanno interfacce incompatibili di lavorare insieme e si usa quando si vuole riutilizzare una classe esistente la cui interfaccia non è conforme a quella desiderata



## Pattern DECORATOR

Permette di aggiungere responsabilità ad un oggetto dinamicamente.

Fornisce un'alternativa flessibile alla specializzazione, 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



**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 componenti e definisce un'interfaccia di Component

**ConcreteDecorator** aggiunge responsabilità al componente referenziato

## Ereditarietà Dinamica

Una sottoclasse deve sempre essere una versione più specializzata della superclasse.

Un buon test sul corretto utilizzo dell'ereditarietà è che sia valido il principio di sostituibilità di Liskov.

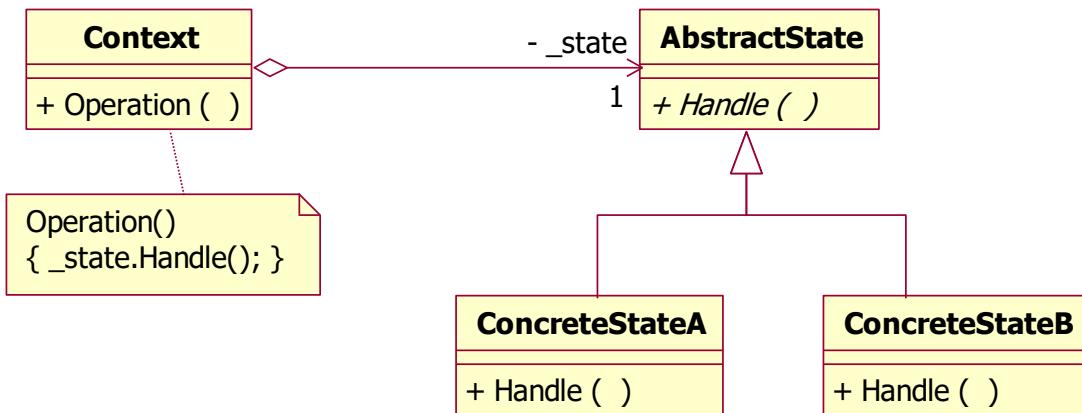
Come può un oggetto cambiare comportamento al cambiare del suo stato?

- **Prima possibilità** si cambia la classe dell'oggetto runtime (quasi mai possibile)
- **Seconda possibilità** si utilizza il pattern State che usa un meccanismo di delega, grazie al quale l'oggetto è in grado di comportarsi come se avesse cambiato stato

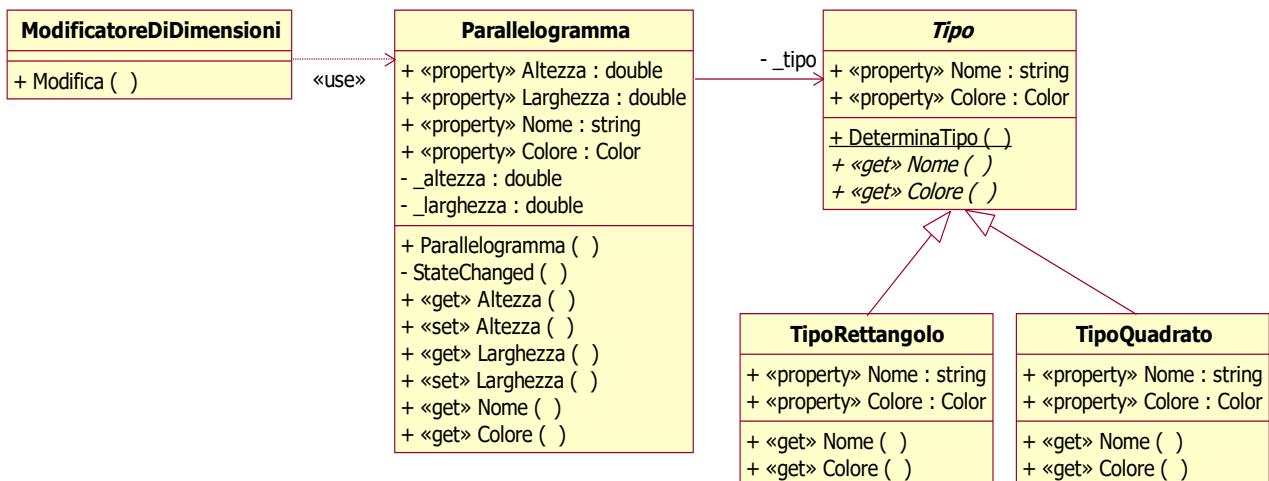
## Pattern STATE

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.  
Permette anche di emulare l'ereditarietà multipla.



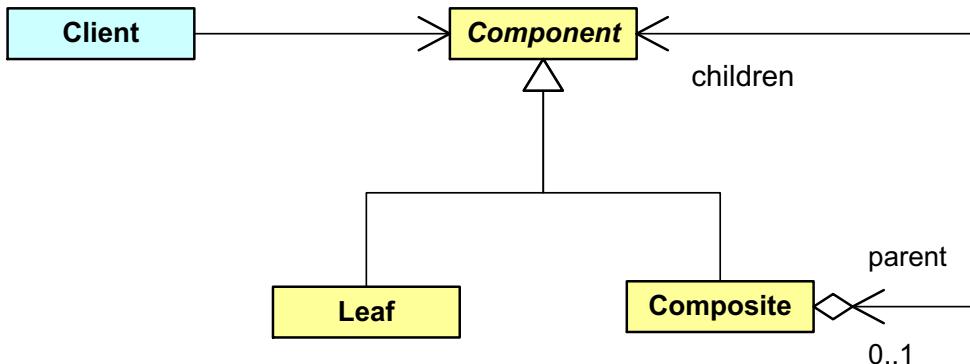
Esempio



## Pattern COMPOSITE

Quando i programmati trattano dati strutturati ad albero devono spesso discriminare se stanno visitando un nodo o una foglia. Questa differenza è una possibile fonte di complessità per il codice e, se non trattata a dovere, rende il programma facilmente soggetto a errori. La soluzione è adottare un'interfaccia che permetta di trattare oggetti complessi e primitivi in modo uniforme. Permette di comporre oggetti di 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 composti



[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 e definisce il comportamento di tali oggetti

[Composite](#) descrive oggetti che possono avere figli 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, ...)

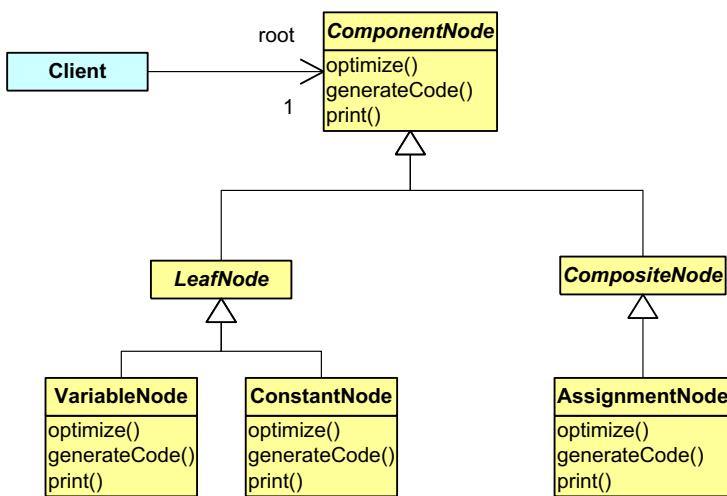
- [Riferimento esplicito al genitore](#): semplifica l'attraversamento e la gestione della struttura, l'attributo che contiene il riferimento al genitore e la relativa gestione devono essere posti nella classe Component
- [Massimizzazione dell'interfaccia Component](#): un obiettivo è quello di fare in modo che il client veda solo l'interfaccia Component in cui devono essere inserite tutte le operazioni che devono essere utilizzate dai clienti
- [Trasparenza](#): dichiaro tutto al livello più alto, in modo tale che il cliente possa trattare gli oggetti in modo uniforme ma così il cliente potrebbe cercare di fare cose senza senso come aggiungere figli ai figli, per evitare ciò dovremmo disporre di un modo per verificare se è possibile aggiungere figli all'oggetto su cui si vuole agire ed i metodi Add e Remove devono generare eccezioni
- [Sicurezza](#): tutte le operazioni sui figli vengono messe in Composite, a questo punto qualsiasi invocazione sui figli genera un errore in fase di compilazione ma il cliente deve conoscere e gestire due interfacce diverse, dobbiamo quindi disporre di 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 su gli elementi di una struttura senza dover modificare le classi degli elementi coinvolti.

Ad esempio, si condirei la rappresentazione di un programma come un albero i cui nodi descrivono elementi sintattici del programma, su tale albero devono poter essere effettuate molte operazioni di tipo diverso:

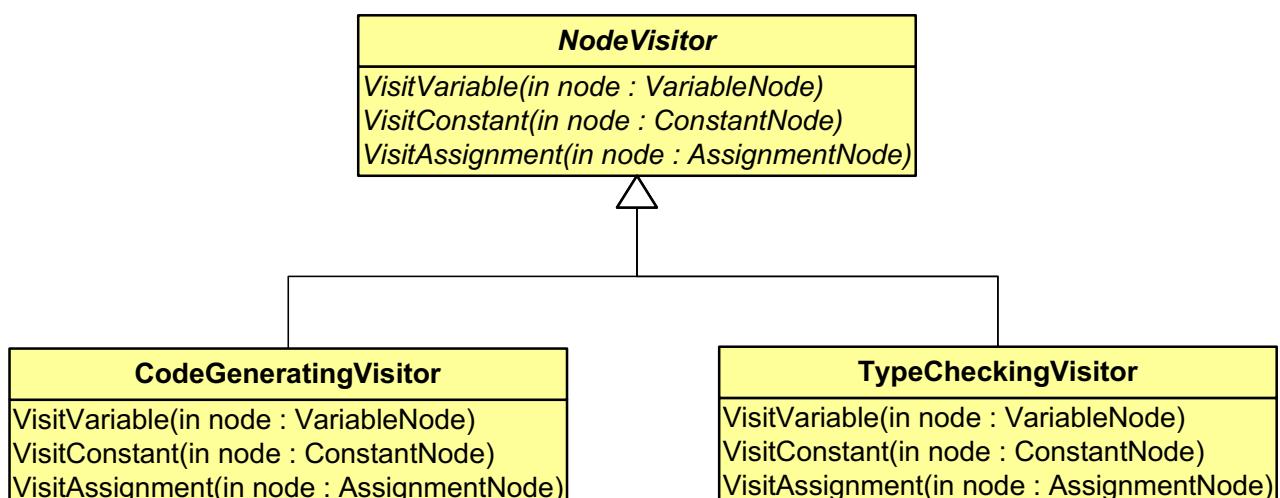
- Controllare che tutte le variabili siano definite
- Eseguire delle ottimizzazioni
- Generare il doccia macchina
- Stampare l'albero in formato leggibile
- ...



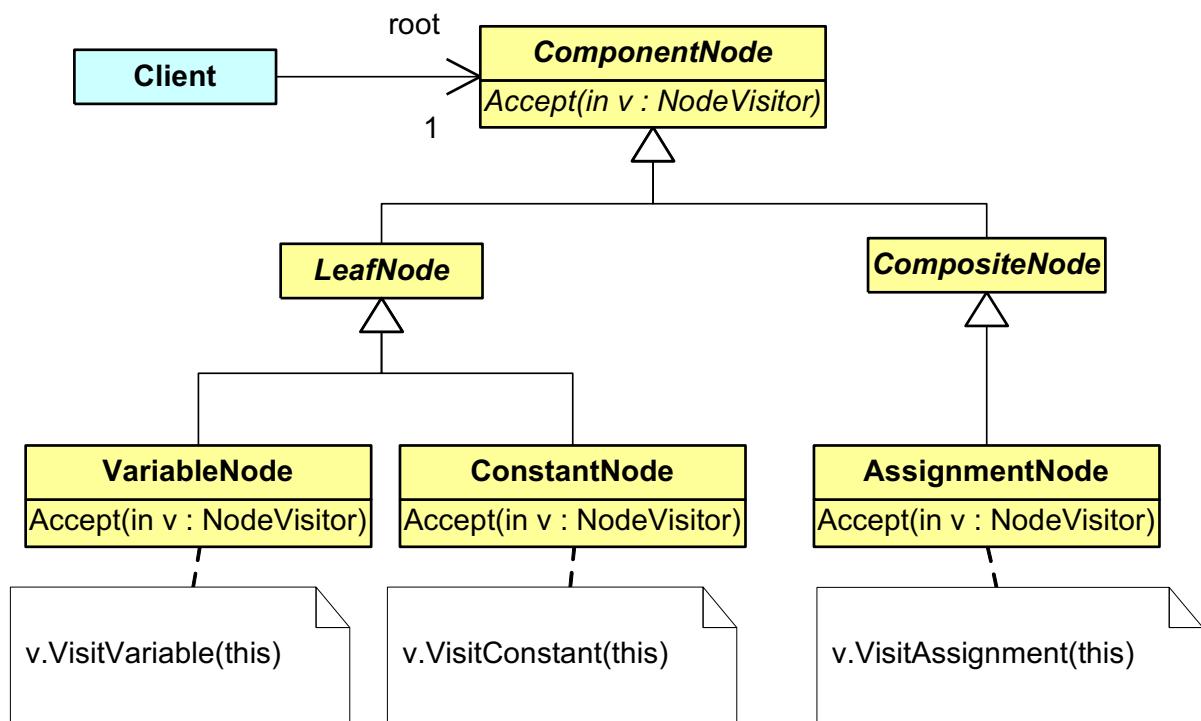
La soluzione è quella di eliminare le singole operazioni all'albero, la cui responsabilità principale è quella di rappresentare un programma sotto forma di albero.

Tutto il codice relativo un singolo tipo di operazione viene raccolto in una singola classe, i nodi dell'albero devono accettare la visita delle istanze di queste nuove classi (visitor) per aggiungere un nuovo tipo di operazione e sufficiente progettare una nuova classe.

Il visitor deve dichiarare un'operazione per ogni tipo di nodo concreto

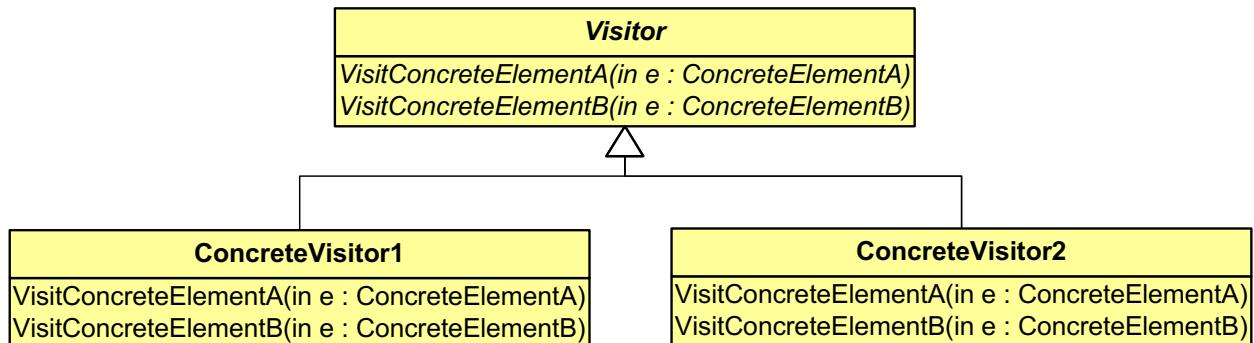


Ogni nodo deve dichiarare un'operazione per accettare un generico visitor



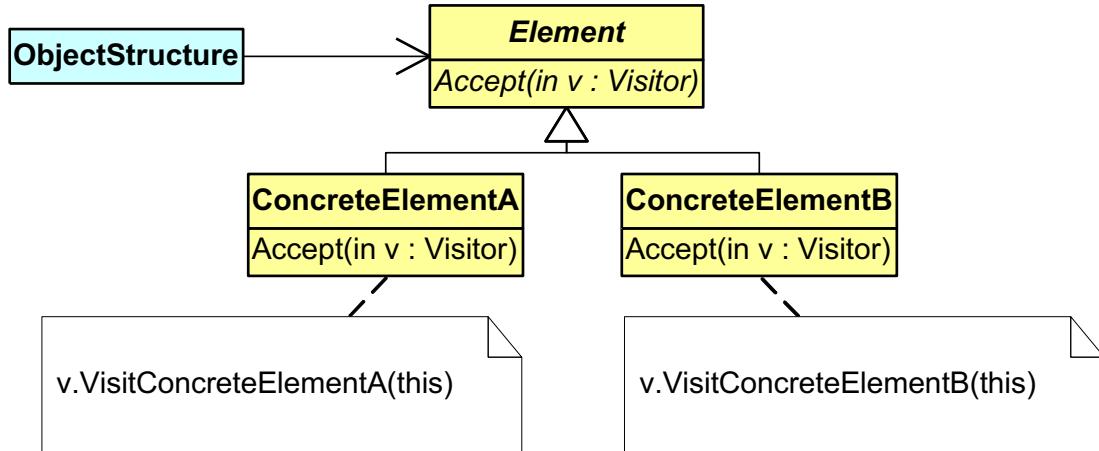
**Visitor** (interfaccia o classe astratta) dichiara un metodo Visit per ogni classe di elementi concreti

**ConcreteVisitor** definisce tutti i metodi Visit, globalmente definisce l'operazione da effettuare sulla struttura e, se necessario, ha un proprio stato



[Element](#) (interfaccia o classe astratta) dichiara un metodo Accept che accetta un Visitor come argomento

[ConcreteElement](#) definisce il metodo Accept



La `ObjectStructure` può essere definita come `Composite` o come normale collection, deve poter enumerare i suoi elementi e deve dichiarare un'interfaccia che permetta a un cliente di fare visitare la struttura a un Visitor

Questo pattern 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 deve essere in grado di accedere allo stato degli elementi cui deve operare.

È difficile aggiungere una nuova classe `ConcreteElement` perché per ogni nuova classe è necessario inserire un nuovo metodo `Visit` in tutti i Visitor esistenti

Non è necessario che tutti gli elementi da visitare derivino da una classe comune

Ogni Visitor durante operazione può modificare il proprio stato.

L'operazione che deve essere effettuata dipende dal tipo di due oggetti: il visitor e l'elemento.