

# Riassunti di Ingegneria del Software

Simone Montali  
monta.li

3 gennaio 2020

## Prefazione

Questo progetto nasce dalla necessità di trovare un metodo di studio per questa materia che, ai più sembra banale. Il problema di fondo è proprio in questa apparente banalità: si finisce per studiarla di fretta pensando di conoscerla, e ci si rende conto troppo tardi di non essere pronti. Mi scuso, anzitutto, per il vocabolario misto italiano-inglese che utilizzerò in queste pagine. Tanti di voi sanno quanto sia complicato esprimere certi concetti in italiano. Ho inserito un piccolo glossario a fine documento. Questo documento ha lo scopo di essere il giusto mezzo tra completezza e sinteticità. Mi scuso, in secundis, per i toni a volte scurrili. Un vero informatico è arrabbiato */per il codice che non compila/LATEX che fa ciò che vuole/il computer che si impalla/quel piccolo bugfix di Linux che diventa un bagno di sangue/*, e non c'è modo migliore di sfogare le incazzature informatiche che imprecare in riassunti che leggeranno le generazioni a venire. Non so come tu ti sia procurato questo documento, ma se hai 5 minuti da buttare, dai una letta alle cose che ho scritto qui. Ci troverai anche un'altra valangata di appunti. Buona studiata ed in bocca al lupo per tutto.

## 1 T1 - Software Development Process

Anche la signora seduta all'angolo di via Cavour, sa che oggi più che mai i software sono parte di tutti i processi che fanno girare il mondo. L'Ingegneria del software è però molto di più che scrivere codice. Infatti, è piuttosto un concetto di risoluzione dei problemi del mondo reale, sfruttando software. I requirements sono sempre più stringenti: tempi brevi, sistemi complessi, molte funzionalità richieste. Un buon software deve avere ottime **maintainability, dependability, efficiency, acceptability**. Problemi e soluzioni sono complessi, ma il software offre estrema flessibilità. Esso è un sistema discreto. Alcuni problemi tipici possono essere le scadenze, i budget, le performance, la manutenzione. Le sfide principali sono rappresentate da **eterogeneità, delivery, trust**. L'attività di problem solving è composta da due fasi: l'analisi e la sintesi.

### 1.1 Quindi? Cos'è l'ingegneria del software?

Siamo giunti al nocciolo della questione: **di cosa stiamo parlando?** L'ingegneria del software è un insieme di **tecniche, metodologie, strumenti** che aiutano nella produzione di software di alta qualità dati un budget, una scadenza, e delle modifiche continue. La sfida principale è quindi quella di avere a che fare con complessità elevate. Siamo di fronte anche a un aumento delle responsabilità: un ingegnere del software non deve solo scrivere codice, ma piuttosto lavorare con competenze e confidenzialità, applicando un'etica.

### 1.1.1 Processo del software

In seguito ad una rappresentazione astratta, si procede con un set di attività strutturato: specifica dei requirements, design, implementazione, validazione, evoluzione.

## 1.2 Modelli di sviluppo del software

Distinguiamo tra **plan-driven** e **agile** development. Nel primo, prima si pianificano i requirement, e solo in seguito si sviluppa il software. Nel secondo si sviluppa il software un pezzo alla volta, a stretto contatto col cliente.

### 1.2.1 Modello a cascata

In questo modello, **plan-driven**, le specifiche e lo sviluppo sono separati. I pro sono, ad esempio, un'ottima documentazione e manutenzione semplice. D'altro canto, però, le specifiche vengono *"congelate"* dopo la prima fase, il cliente viene poco coinvolto, ed i tempi sono più lunghi.

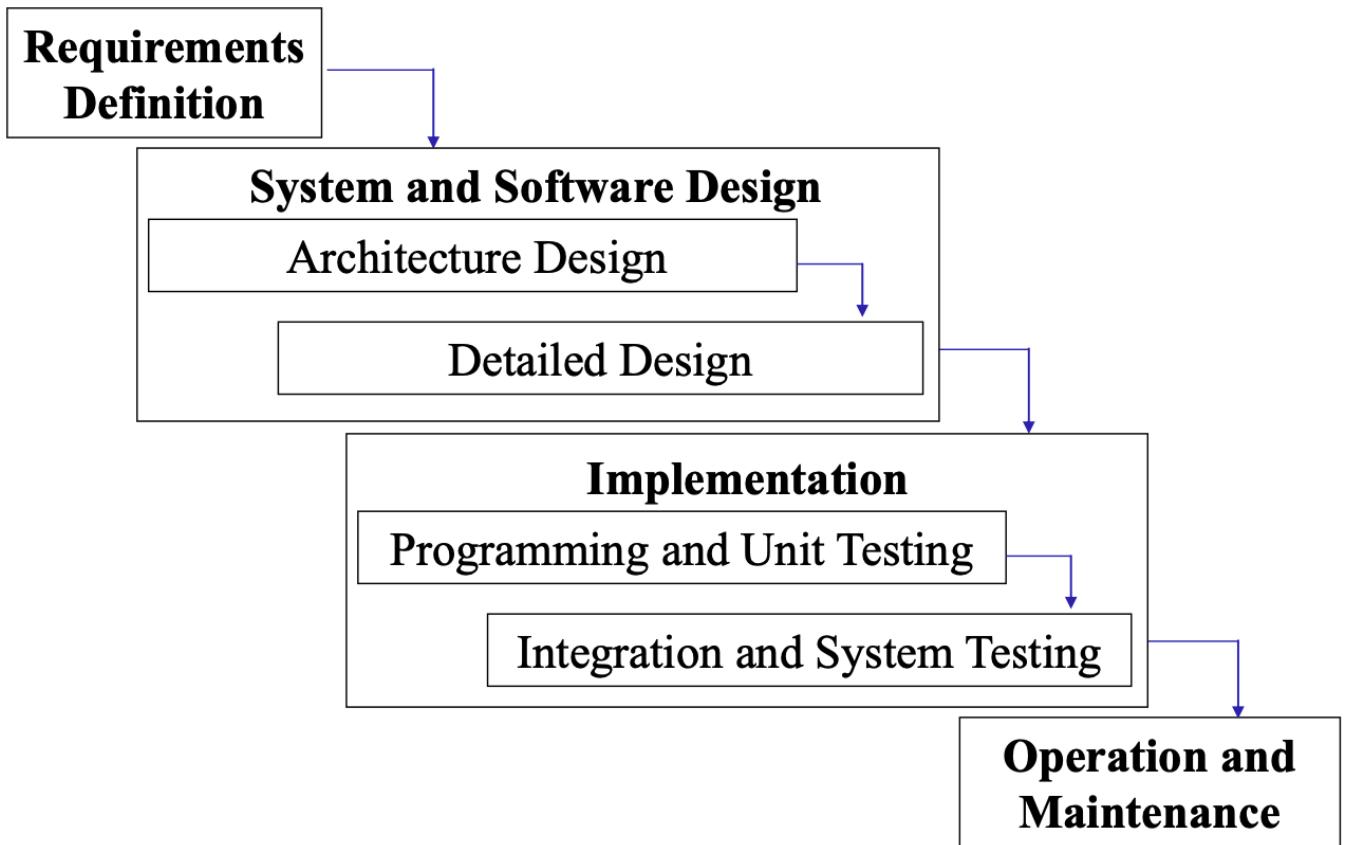


Figura 1: Waterfall Model

### 1.2.2 Modello a spirale

Nel modello a spirale, abbiamo diverse fasi che si susseguono a spirale; il *risk handling* viene gestito tramite prototipazione, che permette di testare i prodotti contro i requirements. Alcuni pro possono essere l'elevata prevenzione dei rischi, la completezza della documentazione, la flessibilità. È, però, un modello costoso, riservato ad esperti ed a progetti costosi e richiedenti molta sicurezza. La spirale può allargarsi all'infinito.

Il **prototipo** è un'implementazione limitata del sistema, rappresentando solo alcuni aspetti. È utilizzato in varie fasi dello sviluppo. Porta però vari vantaggi, come un'elevata usabilità, un buon design, una grande facilità di manutenzione, ed un ridotto costo di sviluppo.

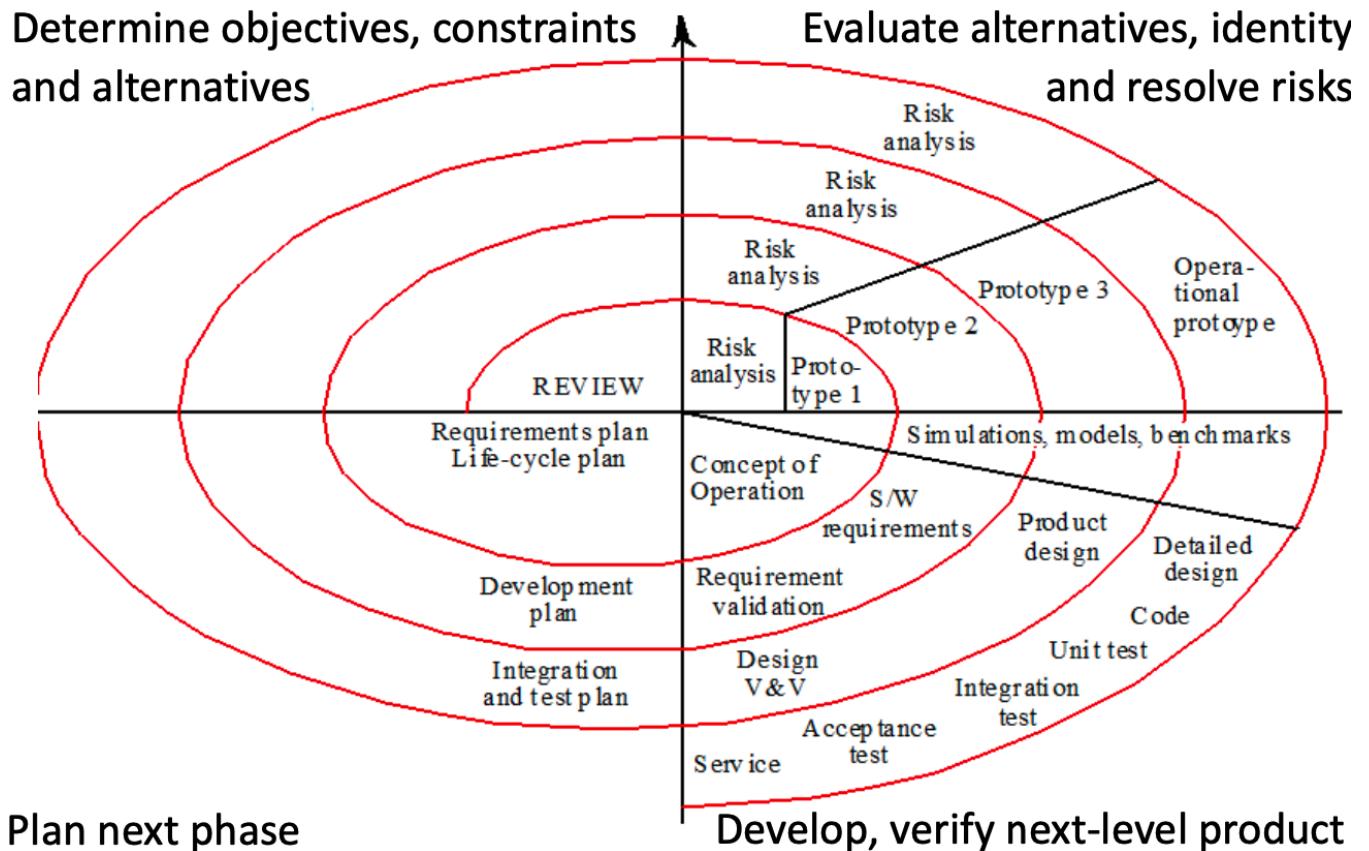


Figura 2: Spiral Model

### 1.2.3 Incremental development

L'incremental development consiste in una prima fase di raccolta dei requirement, da cui nasce la versione iniziale, una fase di design, ed una fase di implementazione, che produce la versione finale. Alcuni pro possono essere la naturale presenza di prototipi ad ogni aggiunta di feature, un basso rischio di fallimento progettuale, una quantità di testing variabile in base alla priorità. Alcuni contro: bassa *process visibility*, sistemi mal strutturati, skill speciali necessarie. Esso è adatto per progetti piccoli, o parti di progetti grandi.

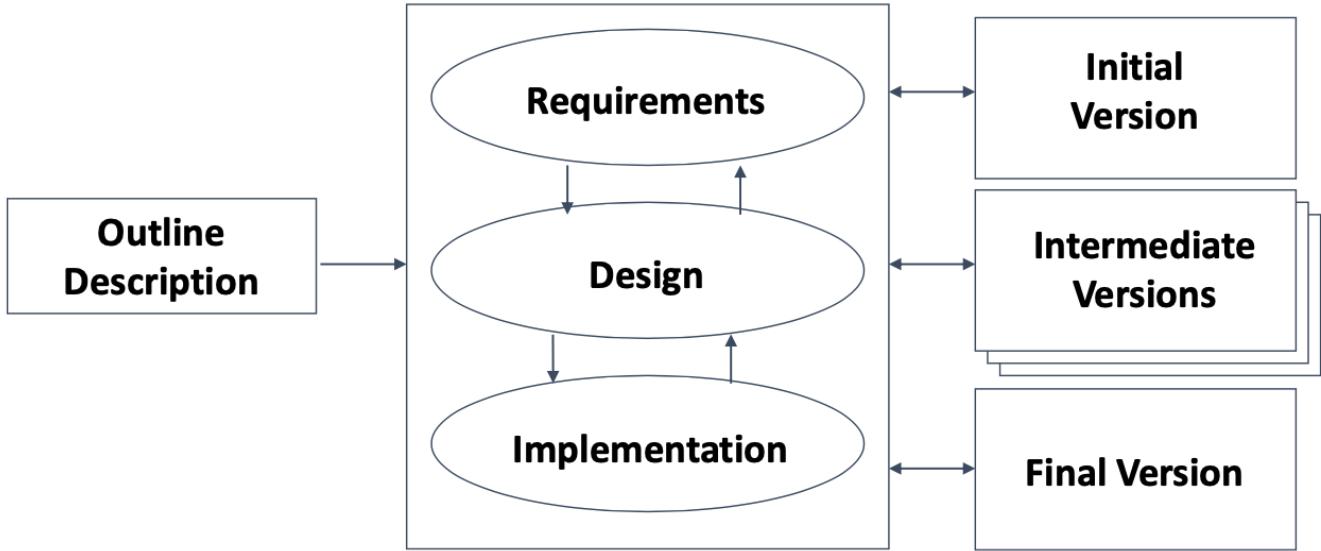


Figura 3: Incremental Development

#### 1.2.4 Test driven development

Qui, i test vengono scritti prima dell'implementazione, rendendo note le difficoltà da subito. Rende il debug più semplice. Si aggiunge un test, si prova il codice vecchio con il test nuovo, si aggiunge la feature e si verifica che il test sia ancora positivo.

#### 1.2.5 Agile development

*"Tutti fanno Agile, nessuno fa Agile."* L'agile è basato sulla **continuous delivery**, con dei requirements in continuo cambiamento. Il cliente è direttamente coinvolto, aggiungendo requirements man mano che il progetto va avanti. È una metodologia semplice nella quale il team si auto organizza, ma ha svariati rischi, come la mancanza di planning, la necessità di team esperti, la documentazione scarna o spesso errata.

#### 1.2.6 Extreme Programming

*Suona più badass di quanto sia realmente/s* L'XP è utilizzato in situazioni in cui i requirements variano velocemente, i team sono ridotti e "affiatati" (spesso si ricorre al **pair programming**). È un tipo di programmazione agile, basato su design semplice, release minori, refactoring continuo, alta semplicità.

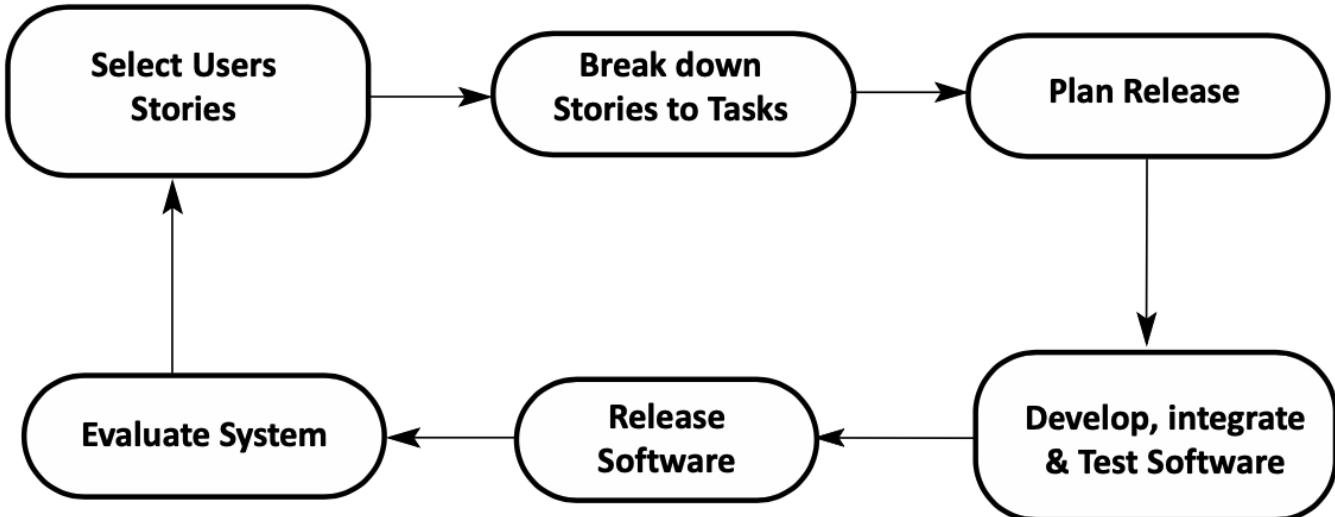


Figura 4: Extreme Programming

### 1.3 Reusable software

È spesso comodo lavorare per **microservizi** atomici, in modo da poterli riutilizzare. Parliamo, ad esempio, di API. Questo riduce i costi e i tempi di sviluppo, al costo di leggeri sacrifici sul lato dei requirements, e un mancato controllo sull’evoluzione del software → funzionalità che variano il loro comportamento. *Parlo a lei, signor React (ah, già, Mark), che cambia tutto ad ogni versione.*

## 2 T2 - Coding, Debugging, Testing

Un buono stile di coding è fondamentale perché i programmi vengono scritti una volta, ma letti molte. Alcuni aspetti importanti sono il layout, i nomi, i commenti.

### 2.1 Legge di Ambler per gli standard

Più uno standard è utilizzato, più è facile comunicare tra membri del team. Si possono inventare standard quando necessario, ma attenzione a non perdere tempo in qualcosa che non verrà riutilizzato. Tutti i linguaggi hanno standard reperibili. *Consiglio per la vita: quando vi servono, esistono gli standard di Google, gente più affidabile di me.* È consigliabile avere standard aziendali. Uno standard consiste di nomenclatura, formattazione, formato e contenuto dei commenti. È necessario documentare tutte le volte in cui si infrangono gli standard. (*È davvero necessario infrangerlo?*)

### 2.2 Coding practices

Alcune pratiche sono rappresentate da:

- **indentazione**, che oltre ad essere importante per la leggibilità, a volte fa parte delle regole di compilazione. *e.g. Python.*
- **whitespaces**, migliorano la leggibilità. *Però, non fate le bestie di satana mettendo uno spazio dopo l’apertura della parentesi. Non fatelo.*
- **Naming, commenting**, rendono la comprensione del codice molto più facile.

I commenti sono uno strumento fondamentale da tenere sempre vicino al codice. Non pensiate però di essere autorizzati a scrivere *spaghetti code*, se commentato. Refactorate. Fondamentale è spiegare i compiti di classi, funzioni, variabili o blocchi di codice complessi. *In generale, se avreste bisogno di spiegarlo a Guido Soncini, commentatelo.* Utilizzare uno standard permette ai vostri colleghi di non dover riscrivere il vostro codice perché non gli piace. Rendete più semplice aggiungere funzioni, o creare la documentazione. È, insomma, tempo ben speso.

## 2.3 Dealing with errors

Distinguiamo tra prima, durante e dopo: **prevention, detection, recovery**. Alcune fonti di errori possono essere un design errato, una mancanza di isolazione, o typos. Per esempio, errori di "confini" negli array, errori di *off-by-one*, errori di input errati. Per debuggare, bisogna riconoscere l'esistenza di un bug, isolarne la fonte, identificarne la causa, trovare un fix, applicarlo, e **testarlo**. *Lo scrivo in grassetto perché mi capita spesso di rompere più di quello che metto a posto.* Riconoscere un bug spesso è complicato, soprattutto quando accade solo in determinate situazioni, o se il software è difficile da testare. Per trovare i bug, potete usare dei print statement, molto veloci da usare ma spesso incompleti e poco pratici. Per questo, esistono gli strumenti di debug, che permettono di bloccare il codice in determinati punti, analizzare le variabili, e *bestemmiare con calma*. Spesso gli errori sono dovuti al design piuttosto che all'implementazione.

## 2.4 Testing

Il testing permette di scovare errori e bug, ma non la loro assenza. È purtroppo impossibile testare ogni caso. Il tester deve conoscere il sistema e le tecniche di testing. **Il tester non dovrebbe essere il programmatore.** Spesso il programmatore ha in mente il modo corretto di far funzionare il programma, e quindi difficilmente trova casi in cui il suddetto si rompe.

### 2.4.1 Unit testing

Lo unit testing permette di testare singole unità di codice, trovando fallo negli algoritmi, i dati, la sintassi. Un set di test cases viene creato e poi utilizzato.

### 2.4.2 Integration testing

L'integration testing prova un gruppo di sottosistemi, o anche l'intero software. Viene eseguito dai programmatori, il goal è testare le interfacce oltre ai sottosistemi. L'intero sistema è visto come un insieme di sottosistemi, l'obiettivo è quello di testare tutte le interfacce e l'interazione tra sottosistemi. La strategia determina il modo in cui i sottosistemi vengono testati. Molte fallo sono date da problemi nell'interazione tra sottosistemi. Le fallo non intercettate in questa fase diventeranno molto più costose.

### 2.4.3 System Testing

Il system testing testa l'intero sistema, per verificare che rispetti i requirements funzionali e non, oltre alle prestazioni.

### 2.4.4 Functional Testing

Questo tipo di testing viene svolto per verificare la funzionalità del sistema. I test cases vengono ideati a partire dai requirement del progetto, ed il sistema è trattato come una black box.

#### 2.4.5 Performance Testing

Questo tipo di testing tenta di provare il sistema in situazioni estreme, come alti carichi, input errati, grandi volumi di dati. Alcuni esempi sono stress testing, security testing, volume testing, recovery testing.

#### 2.4.6 Acceptance Testing

Questo tipo di testing prova che il sistema sia effettivamente pronto per la fase di production. I test vengono scelti ed effettuati dal cliente. Questi sono i famosi **alpha e beta tests**. Nel primo, il software è ancora nell'environment di sviluppo. Nella beta, l'environment è quello del cliente e l'utilizzo effettuato è realistico.

### 3 T3 - System Modeling and UML

Il system modeling fornisce rappresentazioni astratte a problemi reali, tramite notazione grafica. Un modello funzionale dovrebbe introdurre i componenti essenziali, utilizzare una notazione *consistente*, ed utilizzare tool al supporto della creazione. Il modello esprime quindi la realtà, adattata a dei modelli standard. Il system modeling deve essere **predictive**, in quanto deve essere svolto prima del development. Dev'essere **extracted** da un sistema esistente, tramite analisi delle proprietà del software. Deve essere **prescriptive**, ossia definire un set di regole e limiti per l'evoluzione del software.

#### 3.1 UML

Unified Modeling Language nasce per unire diversi standard/linguaggi di modellazione, con diagrammi multipli e interoperabilità. UML è **semplice, espressivo, utile, consistent, estensibile**. Alcuni esempi di views sono:

- Use Case view
- Structural view
- Behavioral view
- Implementation view
- Environment view

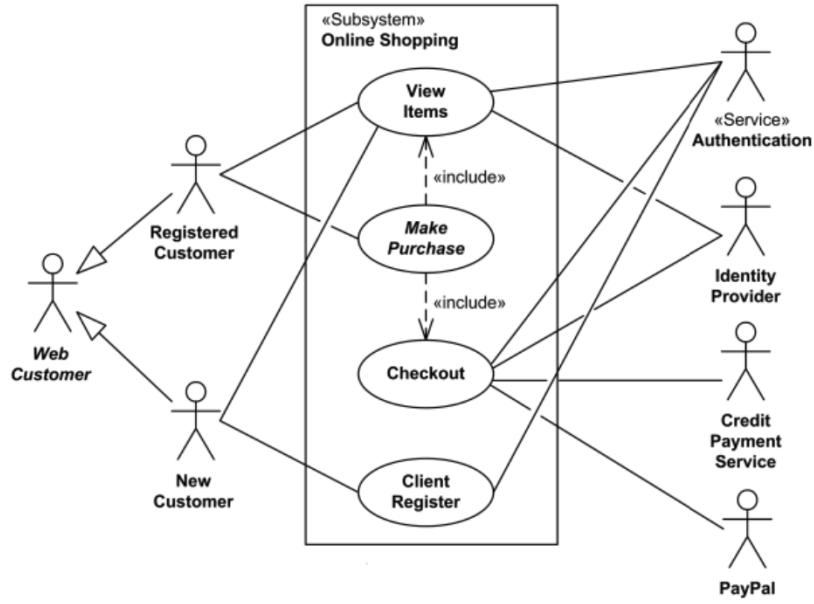


Figura 5: Use case diagram

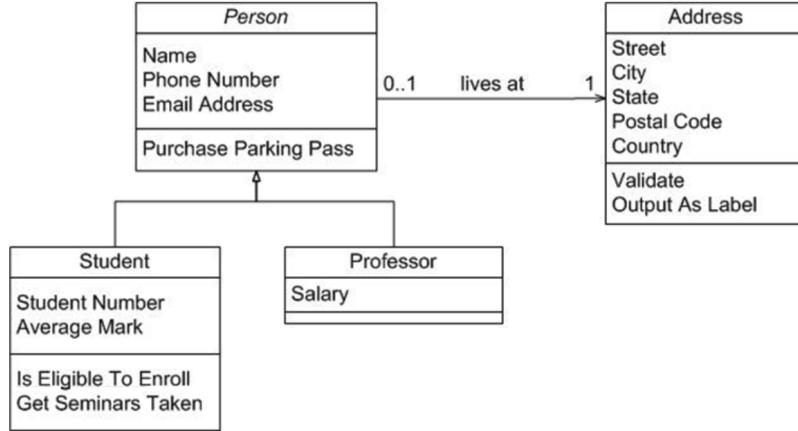


Figura 6: Class diagram

## 4 T4 - Requirements Engineering

Gli scopi del requirements engineering sono **identificare** i servizi necessari ed i constraint, **definire** offerta e contratto, **ottenere** tutte le informazioni necessarie al design. I desiderata sono:

- **Validi**, esprimendo le reali necessità
- **Non ambigui**, leggibili in un solo modo
- **Completi**
- **Comprensibili** da tutte le persone coinvolte
- **Consistenti**, non possono contraddirsi

- **Prioritizzati**, a volte bisogna scegliere
- **Verificabili**, con test
- **Modificabili** senza difficoltà
- **Traceable**, la loro origine è chiara

## 4.1 Some requirements classifications

### 4.1.1 Functional requirements (System Feature)

Descrivono funzionalità di sistema o di servizi, come l'input di dati, operazioni svolte, workflow, dati in output, autorizzazioni. Ad esempio, in una biblioteca, un functional requirement può essere la ricerca di libri da parte di un socio.

### 4.1.2 Non-functional requirements (System Feature)

Descrivono constraints di parti del sistema e del suo sviluppo. Specificano criteri per giudicare l'operato del sistema. Con l'esempio di prima, i libri devono avere un codice che rispetti lo standard ISBN, e il sistema non deve rilasciare informazioni sensibili sui soci a determinati autorizzati. Alcune metriche per questi requirements possono essere velocità, dimensione, facilità d'uso, affidabilità, robustezza, portabilità.

### 4.1.3 Domain requirements (System Feature)

I domain requirements derivano dal dominio dell'applicazione, ossia l'ambito in cui si lavora.

### 4.1.4 Volatile Requirements (Static/Dynamic Nature)

I **mutable requirements** sono requirements destinati a cambiare, come normative o tasse. Gli **emergent requirements** cambiano quando il cliente capisce di più sul sistema. I **consequential requirements** emergono con l'informatizzazione di un sistema che non lo era. I **compatibility requirements** emergono dal doversi interfacciare con altri sistemi appartenenti all'organizzazione.

## 4.2 Rischi

Alcuni rischi nella scrittura dei requirements possono essere:

- Imprecisioni
- Conflitti tra più requirements

## 4.3 Documento di specifica dei requirements

Il **documento di specifica dei requirements** specifica i requirement di sistema, includendone una definizione e una specifica. È detto **System Specification** se include direttive su hardware e software, **Software Requirements Specification** (SRS) se include il solo software. Dovrebbe seguire lo standard IEEE 830. Un SRS deve avere un'**introduzione**, una **descrizione generale** ed infine **le feature e i requirement**. Dovrebbe avere un formato stratificato, notazioni grafiche e termini consistenti, acronimi chiari, indice, glossario, ed uno stile non ambiguo. A tal proposito, il linguaggio naturale spesso nasconde delle insidie: mancanza di chiarezza, ambiguità, troppa flessibilità... Bisogna quindi inventare uno standard di utilizzo del linguaggio naturale, con sintassi fissa, termini chiari. Alcune keyword sono: **shall**, **should**,

**can, must, may, will, might, expected to, could.** Alcune alternative al linguaggio naturale possono essere un linguaggio naturale strutturato, linguaggi di descrizione del design, notazioni grafiche, notazioni formali. I rischi del processo di specifica sono una mancanza di comprensione, requirements che cambiano rapidamente, imprecisione nella stesura del documento, difficoltà nel conciliare conflitti.

## 5 T5 - Requirement engineering and UML

### 5.1 Use case diagram

Nello use case diagram includiamo tutti i casi d'uso del sistema, da parte di diversi **attori**, rappresentanti utenti, ma anche servizi o sistemi. Il **system boundary** divide l'esterno e l'interno del sistema, gli use case dagli attori. Un attore può generalizzare un altro attore. Idem per gli use case. Uno use case può contenere la funzionalità di un altro use case. Uno use case può essere usato per estendere il comportamento di un altro use case, anche con condizioni. Opzionalmente, possiamo includere le molteplicità delle relazioni.

### 5.2 Class diagram

Una classe è rappresentata da un rettangolo che mostra il nome della classe, e opzionalmente il nome degli attributi e delle operazioni. Il nome della classe, gli attributi e le operazioni sono separati in compartmenti. Il simbolo che precede attributi e operazioni ne indica la visibilità: + se pubblico, - se privato, # se protetto, ~ se package. Un'**interfaccia** è una specifica di comportamento che deve essere implementato o, più semplicemente, un **contract**. Un **template** definisce un pattern i cui parametri rappresentano tipi, e può essere applicato a classi, packages, operazioni.

#### 5.2.1 Associazione

Un'associazione è una relazione tra elementi, che implica che uno dei due sia una variabile dell'altro. Essa è rappresentata da un connettore che può includere ruoli, cardinalità, direzione e constraints. Per più elementi, si può usare un **diamond**.

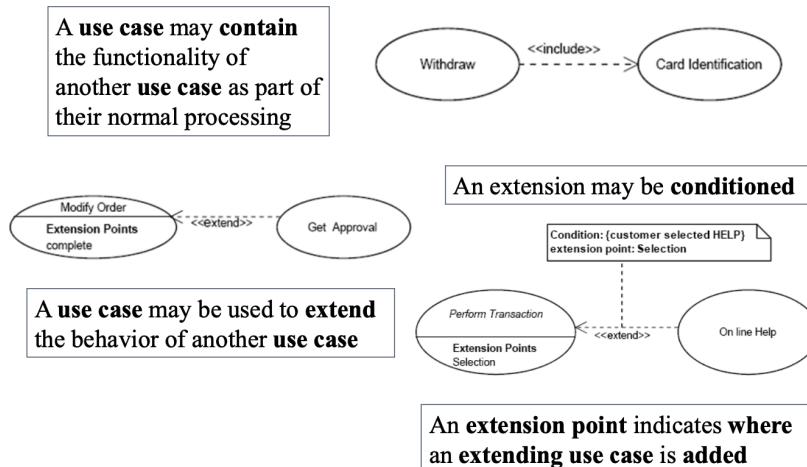


Figura 7: Extension

### 5.2.2 Generalization e nesting

La generalizzazione indica **ereditarietà**, disegnata come una freccia che parte dal figlio e arriva al padre. Il connettore di nesting indica che una classe è nested nella classe dove arriva l'operatore. Con ciò, intendiamo che essa è definita all'interno del target.

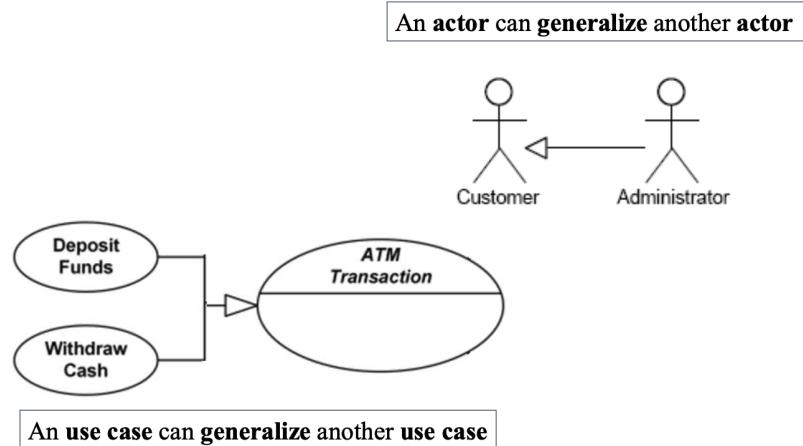


Figura 8: Generalization

### 5.2.3 Dipendenza e realizzazione

La **dipendenza** è una forma debole di relazione, e mostra un'interazione tra un client ed un supplier. La **realizzazione** è una relazione tra una specifica e la sua implementazione.

### 5.2.4 Aggregazione e composizione

L'**aggregazione** rappresenta elementi composti da elementi minori. È indicata da un diamante bianco che punta verso il contenitore. I componenti possono essere condivisi da più contenitori. La **composizione** è rappresentata da un diamante nero. Una classe di associazione rappresenta invece un'associazione più complessa, che ha operazioni e attributi.

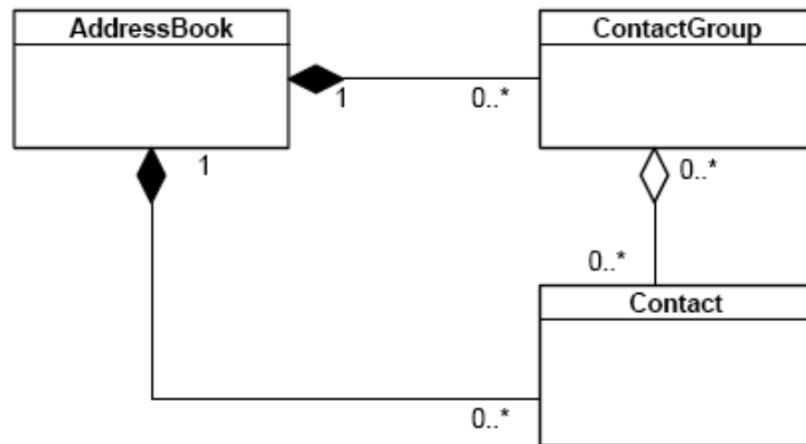


Figura 9: Aggregazione e composizione

### 5.3 Sequence diagram

Il sequence diagram indica, su una *lifeline* (*aka timeline*) verticale, l'interazione tra le classi. Se il nome della lifeline è *self*, la suddetta rappresenta il classifier a cui appartiene il diagramma. Una **lifeline** rappresenta un *partecipante* del sequence diagram. Possiamo avere diversi tipi di messaggi tra lifeline: sincroni, asincroni, risposte, persi, trovati, self (ricorsività). Per rappresentare logica procedurale come if, cicli, thread, usiamo i **fragment**.

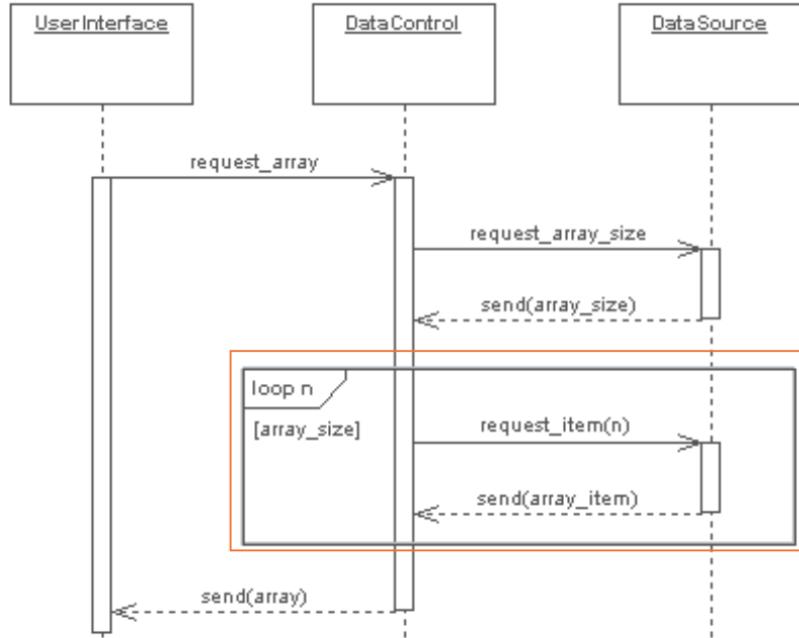


Figura 10: Sequence Diagram con fragment

### 5.4 Activity diagram

Un **activity diagram** è un diagramma di flusso che rappresenta un'operazione eseguita sul sistema, con decisioni, I/O, fork/join, timeout/segnali, concorrenza. Possiamo raggruppare alcune attività *related*. Alcuni esempi di activity diagrams strani:

- **State machine diagram**, che dà indicazioni a livello hardware
- **Choice and Junction pseudo state**
- **Compound state** tramite il quale possiamo separare alcune parti del diagramma e "includerle"
- **History state and concurrent regions**, con il primo indichiamo un salvataggio dello stato, col secondo due frazioni di diagramma che si svolgono contemporaneamente

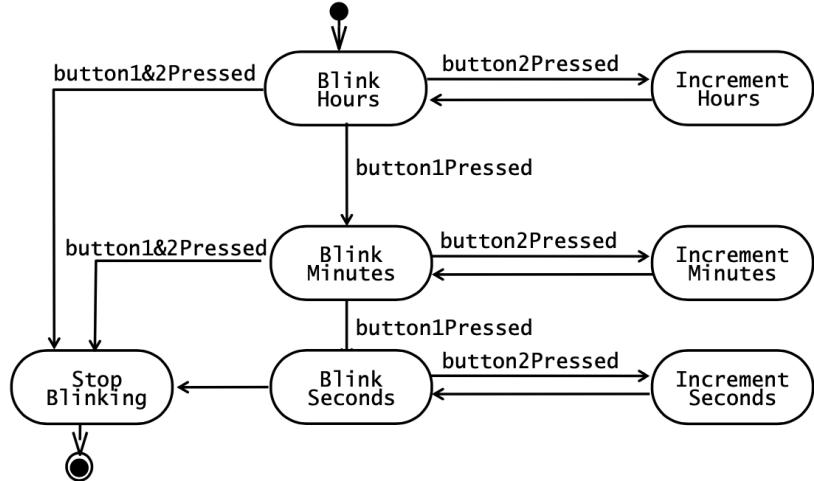


Figura 11: State machine diagram

## 5.5 Robustness diagram

Il **robustness diagram** è un UML semplificato che ha lo scopo di raffinare gli use case, verificandone correttezza, completezza e requisiti. Presenta tre **object nodes**:

- **Boundary**, che permette la comunicazione tra attori e sistema
- **Control**, intermediario tra boundary ed entity, implementa la logica che gestisce i vari elementi e le loro interazioni
- **Entity**, rappresenta un'unità informativa del sistema

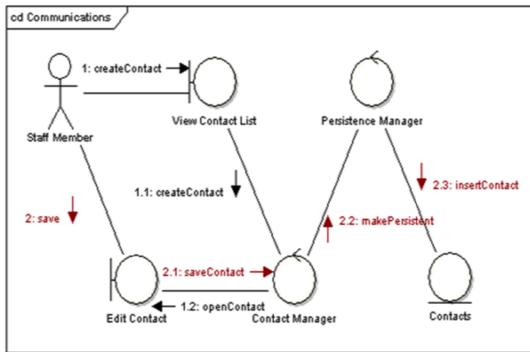


Figura 12: Robustness communication diagram

## 6 T6 - Feasability and requirements elicitation

Per ottenere le informazioni necessarie, dobbiamo identificare le fonti, acquisire le informazioni, analizzarle e verificarle. Infine, vanno sintetizzate. I vari stakeholder *aka le persone interessate* possono essere così categorizzati con i loro interessi;

- **Stakeholder**, interessato nel progetto

- **Developer**, alta produttività, mancanza di errori, minore sforzo possibile
- **Marketing sales**, soddisfazione del cliente e vendite
- **Project management**, budget, scadenze
- **Investor**, velocizzazione del processo
- **Customer and user**, usabilità e workflow

## 6.1 Tecniche di elicitation

*(elicitation = tirare fuori le informazioni)*

### 6.1.1 Document analysis

Bisogna preparare i documenti che possono essere adatti e rilevanti, studiarli, annotare informazioni ed elencare le domande al riguardo. Infine, assieme agli stakeholder, verificare le note, organizzare i requirement e rispondere alle suddette domande. Questa tecnica può essere sfruttata quando non c'è presenza degli stakeholder, o per il cross-checking.

### 6.1.2 Observation of the work environment

L'obiettivo è determinare **chi, cosa, dove, quando, perché e come**. Bisogna ottenere il permesso dai supervisori, informare gli osservati, prendere appunti, evitare di disturbare ma al tempo stesso non dare nulla per scontato. I dati ottenuti sono molto affidabili.

### 6.1.3 Questionario

Serve determinare i fatti e le opinioni necessarie, in secundis, da chi reperirli. Determinare quindi le domande, aperte o chiuse, da fare. Può essere conveniente "testare" il questionario su un gruppo piccolo, per poi metterlo a posto e presentarlo a tutti. Questa tecnica restituisce molti dati, è facile da attuare ed affidabile. Non c'è però interazione, e si rischiano risposte incomplete.

### 6.1.4 Interviste

Bisogna, prima di tutto, decidere chi intervistare. In seguito, preparare le domande e porle all'intervistato, con un linguaggio chiaro. Conviene prendere appunti e memo. Questo è un metodo ottimo perché permette di verificare i fatti, coinvolgendo gli end user. Non è però adatto a comprendere i domain requirements.

### 6.1.5 Scenarios and use cases

Essi sono esempi IRL di come il sistema verrà usato, basati su situazioni reali su cui gli stakeholder hanno senz'altro qualcosa da dire. Gli **scenari** sono semplicemente forme strutturate di user stories. Per gli use cases usiamo UML. Queste tecniche sono semplici ed utili, anche per sistemi complessi. È però difficile capire quando fermarsi, e soprattutto non sono utili a capire i **non-functional requirements**.

## 6.2 Attività di supporto all'elicitation

### 6.2.1 Brainstorming

Composto di due fasi, la prima di **storm** in cui si generano le idee, la seconda **calm** in cui vengono filtrate. Necessari due ruoli chiave: uno **scribe** ed un **moderatore**. Nel filtraggio delle idee è fondamentale: unire idee simili, applicare i criteri di accettabilità, votare con una soglia o votare con dei *campaign speeches*.

### 6.2.2 Focus group

Concetto simile al brainstorming ma più strutturato, esplora pro e contro di determinate opzioni. Fondamentale un moderatore.

### 6.2.3 Prototypes

Si mostra l'esecuzione di una task, si identificano le alternative, si cerca di estrapolare i possibili problemi.

## 7 T7 - Use Cases

Gli **use case** sono scenari che sfruttano diagrammi UML, descrivono le task del sistema e l'interazione con gli attori. Non sono mappati one-to-one coi requirements, ma ogni requirement deve essere coperto da almeno uno use case. Gli use case sono quindi composti da use case diagrams, descrizioni testuali, ed interaction diagrams (opzionali). Per identificare gli actor bisogna definire prima i boundary, poi gli utenti, l'hardware, i ruoli. Per identificare gli use case bisogna prima identificare il dominio. Vanno poi annotati come verbi rappresentanti le azioni. Per l'identificazione degli scenari, bisogna comprendere la situazione iniziale e il flusso di eventi. È utile capire cosa può andare male, e tutti i flussi alternativi. Infine, consideriamo la situazione finale. Uno use case deve avere un singolo attore iniziatore, pochi step, non deve includere scelte implementative, può includere UML. Bisogna poi dare una priorità ai vari use case, in base all'impatto, la difficoltà, la necessità.

### 7.1 Componenti principali

Uno use case deve definire lo stato iniziale e le precondizioni. Deve definire l'ordine degli eventi, le alternative, le situazioni eccezionali, e i risultati. Deve inoltre menzionare gli attori coinvolti, i diagrammi related, i problemi di design. Alcune guidelines:

- Non pensare al lato implementativo
- Essere narrativi
- Elencare gli scenari funzionanti
- Elencare tutti i possibili use case
- Utilizzare un formato standard
- Utilizzare i verbi appropriati
- Documentare le situazioni eccezionali
- Non rappresentare singoli step come use cases

<b>Name</b>	The Use Case name. Typically the name is of the format <action> + <object>.
<b>ID</b>	An identifier that is unique to each Use Case.
<b>Description</b>	A brief sentence that states what the user wants to be able to do and what benefit he will derive.
<b>Actors</b>	The type of user who interacts with the system to accomplish the task. Actors are identified by role name.
<b>Organizational Benefits</b>	The value the organization expects to receive from having the functionality described. Ideally this is a link directly to a Business Objective.
<b>Frequency of Use</b>	How often the Use Case is executed.
<b>Triggers</b>	Concrete actions made by the user within the system to start the Use Case.
<b>Preconditions</b>	Any states that the system must be in or conditions that must be met before the Use Case is started.
<b>Postconditions</b>	Any states that the system must be in or conditions that must be met after the Use Case is completed successfully. These will be met if the Main Course or any Alternate Courses are followed. Some Exceptions may result in failure to meet the Postconditions.
<b>Main Course</b>	The most common path of interactions between the user and the system. 1. Step 1 2. Step 2
<b>Alternate Courses</b>	Alternate paths through the system. AC1: <condition for the alternate to be called> 1. Step 1 2. Step 2  AC2: <condition for the alternate to be called> 1. Step 1
<b>Exceptions</b>	Exception handling by the system. EX1: <condition for the exception to be called> 1. Step 1 2. Step 2  EX2 <condition for the exception to be called> 1. Step 1

Figura 13: Use case template

## 8 T8 - Requirements Analysis

L'analisi dei requirements raffina e struttura i requisiti in modo da renderli più chiari, precisi e formali. Esso definisce l'*analysis model*, che raffina la descrizione informale dei requirement e la converte in diagrammi di flusso. Il suddetto non è un modello di design, ed è indipendente dalla piattaforma. Descrive le funzionalità che il sistema deve realizzare.

### 8.1 Analysis Classes

Il concetto è quello di astrarre le entità del problema. Bisogna gestire i functional requirements (i non-functional sono gestiti nel **design architetturale**). Incorpora un set minimale di **responsabilità**. Questa analisi è rappresentata con diagrammi di classe.

### 8.2 Classes discovering techniques

Esistono diverse tecniche atte a scovare le classi coinvolte. Procediamo ad elencarne le più importanti.

#### 8.2.1 Noun verb analysis

Questa analisi sfrutta i documenti contenenti le specifiche di progetto, cercandovi nomi e verbi. Pare ovvio che la completezza dei documenti è fondamentale. La qualità dipende anche dallo stile di scrittura dei requisiti.

### 8.2.2 Use case driven approach

Questo approccio sfrutta gli scenari di use case, e dipende quindi dalla correttezza dei suddetti.

### 8.2.3 Common Class Patterns

Quest'analisi si basa sulla teoria della **classificazione generica degli oggetti**. Essa fornisce delle linee guida, ma non un processo sistematico atto ad ottenere le classi. Porta diversi rischi, tra cui quello di possibili interpretazioni errate dei nomi.

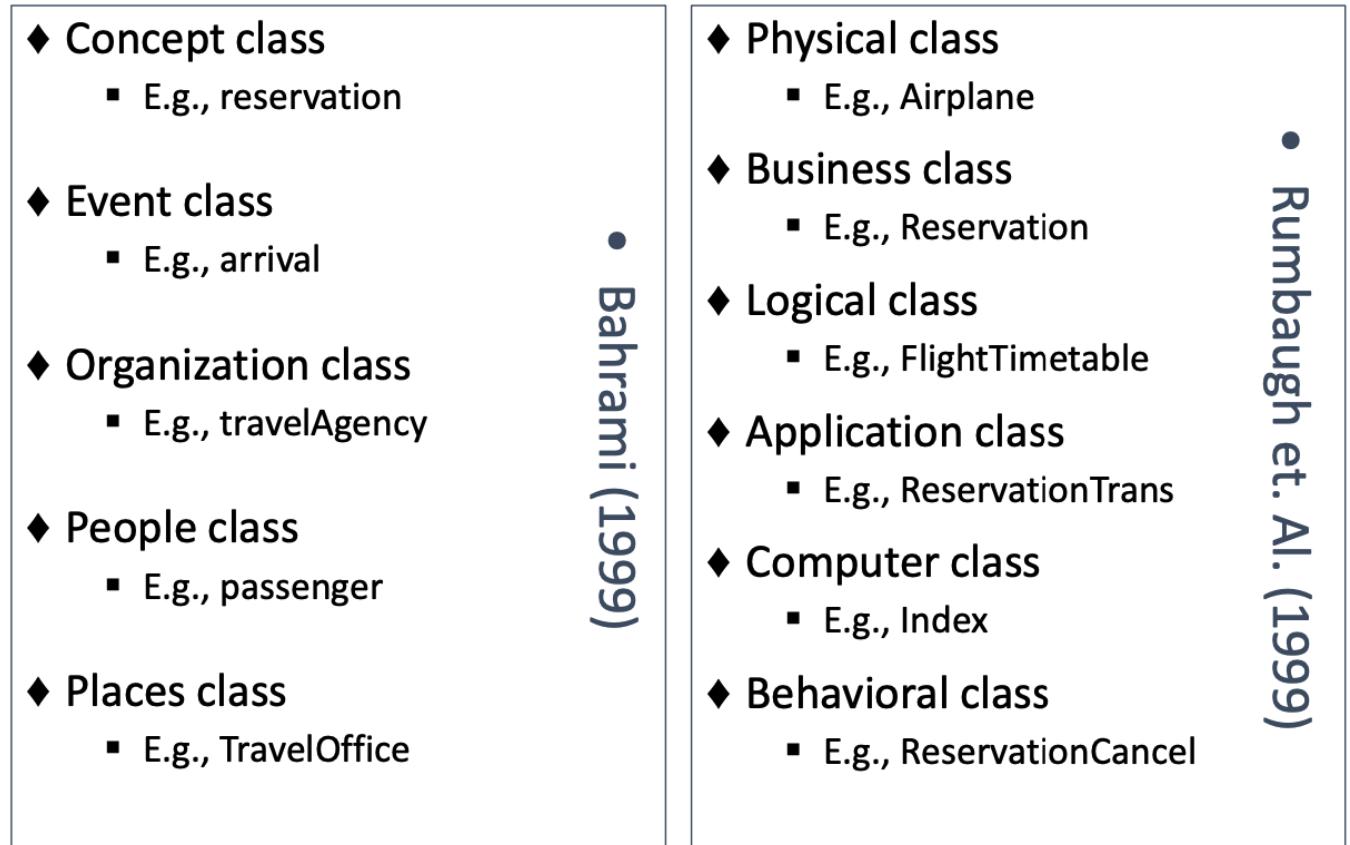


Figura 14: Common Class Patterns

### 8.2.4 CRC Cards

Le **CRC Cards** sono utilizzate in specifiche sessioni di brainstorming. Generalmente si parte dagli use cases, e si creano delle card con:

- Class name
- Responsibilities
- Collaborators

Non viene fornito un metodo sistematico, e per questo le CRC cards sono piuttosto un mezzo di validazione dei requirements.

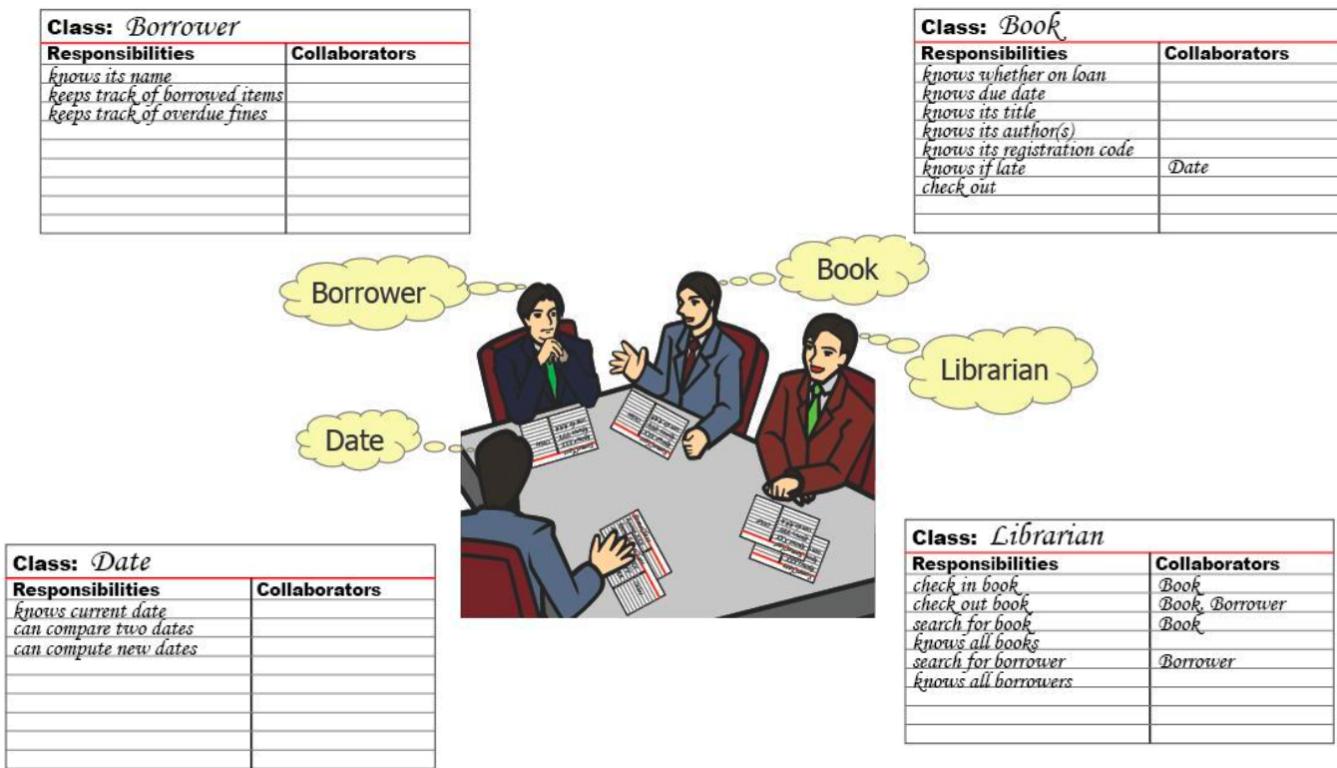


Figura 15: CRC Cards

Gli achievement di questo metodo sono:

- Verificare la correttezza dello use case
- Verificare la correttezza delle associazioni
- Verificare la correttezza delle generalizzazioni
- Trovare le classi omesse
- Scovare opportunità di refactor

### 8.3 Mixed approach

Come sempre, la via di mezzo è quella vincente:

1. Le classi iniziali provengono dalla conoscenza del dominio
2. Si sfrutta, come guida, il common class pattern
3. Per aggiungere altre classi, noun verb analysis
4. Per verificare il lavoro, Use Case approach
5. Per il brainstorming, CRC

## 9 T9 - Requirements Validation and Management

### 9.1 Validation

La validazione consiste nel verificare la correttezza dei requirement, con due obiettivi principali: **complettezza** e **costi**. Inoltre:

- **Consistency**: assenza di conflitti
- **Realism**: possono essere effettivamente implementati?
- **Verifiability**: possono essere verificati?

La review dei requirement dovrebbe essere svolta continuamente durante la loro stesura. Essa deve essere svolta sia dal progettista che dal cliente. Alcuni issue da verificare sono:

- Verifiability: il requirement è testabile?
- Comprehensibility: il requirement è stato compreso correttamente?
- Traceability: l'origine del requirement è chiara?
- Adaptability: i requirement possono essere cambiati senza impatti sul resto?

#### ◆ Requirement

**Does D1 and D2 and D3 and S  $\Rightarrow$  R?**

- (R) Reverse thrust shall only be enabled when the aircraft is moving on runway

**Are the domain assumptions (D) right?**

#### ◆ Domain Properties

**Are (R) or (S) what is really needed?**

- (D1) Deploying reverse thrust in mid-flight has catastrophic effects
- (D2) Wheel pulses are on if and only if wheels are turning
- (D3) Wheels are turning if and only if the plane is moving on the runway

#### ◆ System specification

**D3 is false because the plane may hydroplane on wet runway**

- (S) The system shall allow reverse thrust to be enabled if and only if wheel pulses are on

Figura 16: Esempio di requirement review

## 9.2 Requirements management

Si verificano errori, conflitti ed inconsistenze. Vanno tenuti in conto i lati tecnici, la schedule, i costi, seguendo le priorità del cliente. Va quindi stimato il costo del progetto, controllata la requirements volatility, negoziati i cambiamenti, ri-stimati i costi. L'interesse è soddisfare il cliente, rimanendo nel budget. Le necessità del requirement management sono:

- Identificazione dei requirement
- Un processo di modifica
- Policy di tracciabilità
- Supporto del CASE tool

### 9.2.1 CASE tool support

I requirement devono essere salvati in una zona sicura. Il change management deve essere un processo di workflow con passaggi definiti, e, se possibile, automatizzati. *Immaginatelo come un Github Actions, ma nel '95.* Deve inoltre esserci un'automatizzazione dei link tra requirement, rinforzando la tracciabilità.

### 9.2.2 Identificazione dei requirement

I requirement devono essere identificati univocamente. Potremmo quindi numerare capitoli/sezioni, ma *fa un po' schifo come approccio.* Per questo, sfruttiamo una numerazione dinamica, identificante i record nel DB. Potremmo sfruttare un'identificazione simbolica, con, ad esempio, sigle. *Se usi degli issue tracker, è lo stesso concetto dei codici tra quadre.*

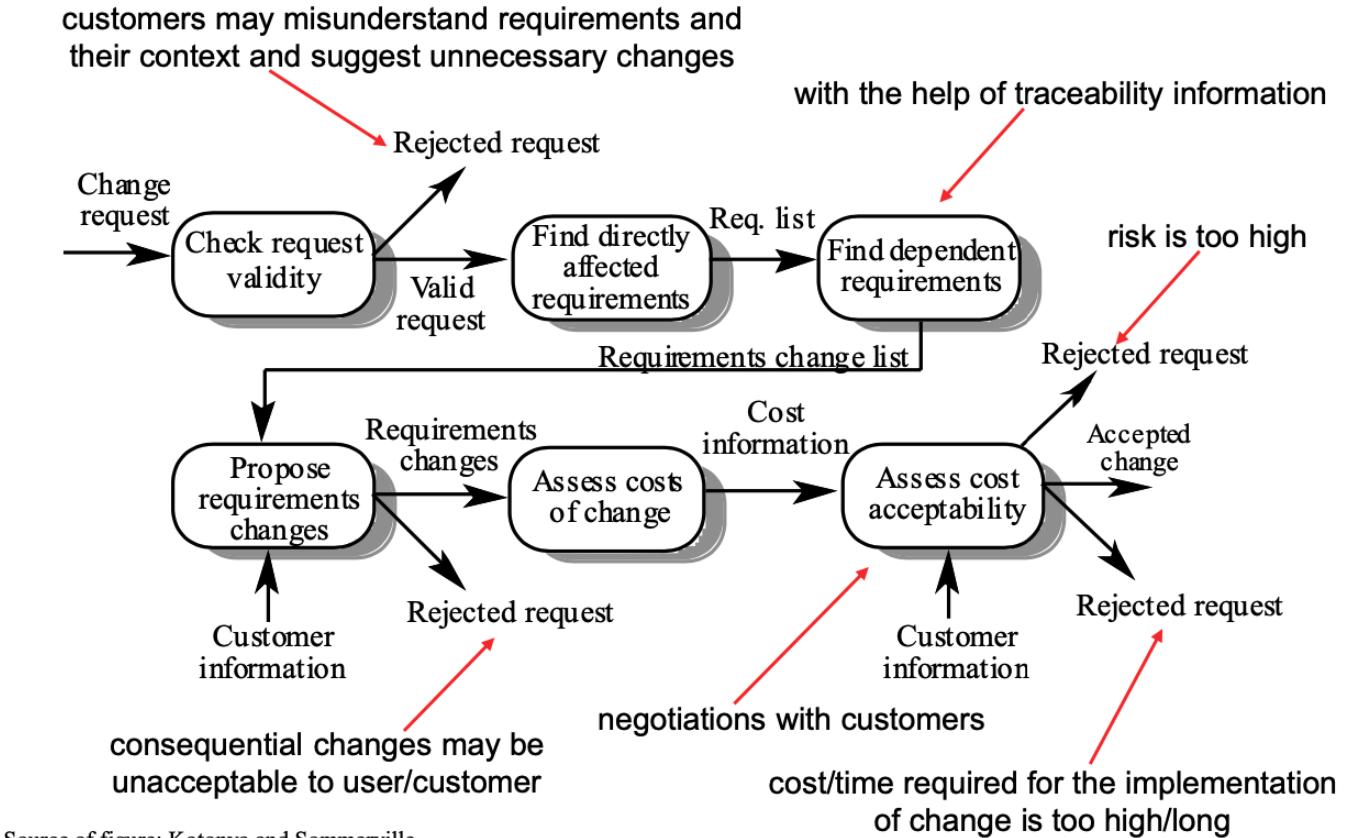
### 9.2.3 Stati dei requirement

I requirement possono avere degli stati (*Anche qui, aver usato issue tracker può chiarire il concetto*):

- Proposed
- Approved
- Rejected
- Implemented
- Verified
- Deleted

### 9.2.4 Change management

È una buona idea implementare un version control dei requirement, per poter tracciare i cambiamenti. Prima di modificare un requirement, è importante analizzare il problema, i costi, e le specifiche.



Source of figure: Kotonya and Sommerville

Figura 17: Change management diagram

### 9.3 Traceability

Lo sviluppo produce relazioni: predecessore-successore, master-subordinate. È fondamentale identificare queste relazioni ed il loro grado, allo scopo di tenere traccia dei *path* nelle gerarchie. Alcuni esempi di traceability:

- User needs → Product features
- Requirements → Implementation
- Use case → Test case

Distinguiamo tra vari tipi di traceability:

- **Source:** la persona o il documento che hanno richiesto il requirement
- **Rationale:** fonte delle informazioni che descrivono le motivazioni del requirement
- **Requirements:** alcuni requirement sono collegati ad altri, da cui dipendono
- **Architecture:** collega i requirement ai sottosistemi in cui sono implementati
- **Design:** collega i requirement con specifici software/hardware necessari all'implementazione
- **Interface:** collega l'interfaccia di sistemi esterni che hanno reso necessario il requirement

- **Feature:** collega il requirement ai suoi componenti
- **Tests:** collega il requirement ai test cases che lo verificano
- **Code:** generalmente non stabilito direttamente, ma può essere implicato

Nella tracciabilità, si può incappare in diversi issue:

- Gli stakeholder richiedono informazioni diverse
- Elevata quantità di informazioni richieste
- Problemi nel management manuale
- Necessità di tool specializzati
- Problemi di integrazione

### 9.3.1 Traceability Planning

Elementi fondamentali al planning sono:

- Tipi di stakeholder
- Informazioni richieste
- Dove e da chi raccogliere informazioni
- Dove e come mantenere le informazioni
- Dove e come cercare le informazioni

Abbiamo inoltre dei constraints al planning:

- **Numero di requirements:** maggiore è il numero di requirement, più sono necessarie delle policy formali di tracciabilità
- **Expected system lifetime:** in caso di lunghi tempi di vita, sono necessarie policy di tracciabilità più comprensive
- **Livello di maturità dell'organizzazione:** in organizzazioni con un'elevata maturità di processo vengono usate policy più dettagliate
- **Dimensioni di progetto e team:** maggiori le dimensioni, maggiore la formalità richiesta
- **Tipo di sistema:** sistemi critici come quelli real-time, o richiedenti sicurezza, hanno bisogno di policy di tracciabilità migliori
- **Constraint addizionali del cliente:** il cliente può necessitare constraint vari, ad esempio standard militari

*I più svegli di voi si chiederanno perché passo dalla lezione 9 alla 12. La risposta è semplice, la 10 e la 11 sono dei doppioni di 8 e 9.*

## 12 T12 - Object Constraint Language

I diagrammi UML spesso non sono abbastanza raffinati per descrivere tutti i constraint sulle relazioni tra entità. Per questo, spesso, essi sono espressi tramite il linguaggio naturale, coi soliti problemi di ambiguità, o con un linguaggio formale, non comprensibile ai più. L'**Object Constraint Language** fa parte delle specifiche UML, ed è un linguaggio formale con sintassi e semantiche definite. Permette di descrivere espressioni e constraint tramite modelli object-oriented. È semplice da leggere e fu sviluppato da niente meno che **IBM**. Può essere utilizzato in vari modi:

- Come query language
- Per specificare constraint su classi e tipi
- Per specificare constraint su operazioni
- Per specificare target su messaggi e azioni
- Per definire espressioni utili alla navigazione tra modelli UML
- Per testare requirement e specifiche

Definiamo tra 3 data types: quelli basic, quelli user-defined e le collection. È possibile eseguire operazioni booleane, operazioni reali/intere. Sono disponibili vari tipi di collection, tra cui nominiamo i set, gli ordered set, le bag (*no, non quelle bag*), in cui gli elementi possono essere ripetuti, e le sequenze, che sono bag ordinate. Le collection hanno delle operazioni (le classiche operazioni da collection), come **size**, **isEmpty**, **notEmpty**, **sum**, **count**, **includes**, **includesAll**.

### 12.1 Model types

I **model types** sono classi, subclasses, classi di associazione, interfacce, enumerazioni. Hanno proprietà, come attributi, metodi, navigazioni (derivate dalle associazioni), enumerazioni. Possono essere inoltre *referenziate* in espressioni OCL.

### 12.2 Operazioni, espressioni, constraints

Abbiamo alcune operazioni di base, come **oclIsTypeOf(type: OclType)**, o **oclIsKindOf**. Ogni espressione, contenente operazioni query, restituisce un risultato. Il tipo dell'espressione è il tipo del risultato. Un constraint è semplicemente un'espressione booleana. I constraint sono quindi restrizioni su uno o più valori di un modello/sistema object-oriented. Sono dichiarativi, non hanno *side effects*, ed hanno sintesi formale e semantiche. Perché si usano?

- Migliore documentazione
- Più precisione
- Comunicazione senza ambiguità

Abbiamo svariati tipi di constraint:

- **Invariant:** deve essere vero quando l'istanza è a riposo
- **Pre-condition:** deve essere vero quando un'operazione è invocata
- **Post-condition:** deve essere vero dopo il completamento di un'operazione

- **Guard:** deve essere vero prima che una transazione possa accadere, quindi è una specie di precondition

La sintassi è semplice: [termine del constraint] [nome del constraint]:[espressione booleana]. I termini per i constraint suddetti sono inv, pre, post, pre/post...implies oclInState.

## 13 T13 - Design Process

Il **design process** è il processo che permette di trovare come implementare il sistema, partendo dai requirements e tenendo conto dei **design general principles**. L'obiettivo è un **modello** ragionevole e realizzabile.

### 13.1 Linee guida

- Esibire un'organizzazione modulare che fa un uso intelligente di controllo tra i componenti
- Partizionare logicamente in componenti che realizzano compiti
- Descrivere sia dati, che procedure
- Arrivare ad interfacce che riducono la complessità
- Derivare utilizzando un metodo ripetibile che fa uso delle informazioni dei requirements

Un buon design deve implementare tutti i requirement esplicativi del modello di analisi, e tutti quelli impliciti desiderati dal cliente. Dev'essere leggibile e comprensibile, sia per chi scrive codice, sia per chi lo testa e supporta. In generale, dovrebbe dare un'immagine generale del software, indicandone i dati, i domini funzionali e *behavioral* dal punto di vista dell'implementazione.

### 13.2 Stadi del design process

Il primo passo è la **comprensione del problema**, che richiede di osservare il problema da punti di vista differenti, per riconoscere i design requirements. Il secondo è **identificare una o più soluzioni**, valutando quelle possibili e scegliendo la più appropriata, in base all'esperienza del designer ed alle risorse disponibili. Il design process è un **processo iterativo ed incrementale**.

### 13.3 Design as series of decisions

Il design può essere visto come una serie di decisioni, scomponendolo in più sottoproblemi, ognuno dei quali presentante diverse soluzioni, tra cui bisogna decidere. Alcuni tradeoff possono essere:

- Funzionalità vs. Usabilità
- Costo vs. Robustezza
- Efficienza vs. Portabilità
- Velocità di sviluppo vs. Funzionalità
- Costo vs. Riutilizzabilità
- Compatibilità col vecchio vs. Leggibilità

Gli step per le priority based decision sono:

1. Elencare le alternative possibili
2. Elencare i pro e i contro di ogni alternativa, rispetto agli obiettivi e alle priorità
3. Determinare se alcune alternative impediscono il raggiungimento degli obiettivi
4. Scegliere l'alternativa che permette di coprire più obiettivi
5. Aggiustare le priorità per le decisioni future

Alcuni strumenti utilizzabili durante questo processo sono la *concept table*, la *concept fan*, la *decision matrix*.

### 13.3.1 Approcci

Abbiamo tre possibili tipi di approccio:

- Approccio **top-down**: prima, si *designa* la struttura di alto livello del sistema, poi si prendono le decisioni man mano più specifiche
- Approccio **bottom-up**: qui, prima si decidono le utilities di basso livello riutilizzabili, poi il modo di integrarle insieme
- Approccio **mixed**: si utilizza il *top-down* per decidere la struttura, ed il *bottom-up* per i componenti riutilizzabili

### 13.3.2 Attività, rischi e obiettivi

Distinguiamo tre attività nel design:

- Design dell'architettura, dove si decidono i sottosistemi ed i componenti e la loro interazione
- User interface design
- Component/class design, dove si decidono le strutture dati ed i meccanismi computazionali

Il design è però un'attività che richiede notevole esperienza, e pertanto un solo ingegnere non dovrebbe mai tentare il design di sistemi grandi, ma piuttosto studiare quelli di altri sistemi. Infatti, un cattivo design potrebbe portare a manutenzione costosa, mentre un buon design dovrebbe essere tenuto in conto per tutta la vita del software, in modo flessibile, documentato, e con una buona gestione delle modifiche. In conclusione, un buon design è **flessibile, dettagliato al punto giusto, ben documentato, e con un buon change management**.

## 14 T14 - Design concepts

Un sistema software ha diversi, possibili, problemi:

- **Rigidità**: quando il codice è difficile da cambiare, il management è riluttante al cambiamento
- **Fragilità**: quando anche cambiamenti minimi creano effetti a cascata, rompendo il codice in punti inaspettati
- **Immobilità**: il codice è così intricato da rendere impossibile il riutilizzo di componenti, che sarebbe più costoso della realizzazione da zero
- **Viscosità**: molto più semplice l'hack del mantenimento del design originale

## 14.1 Software design concepts

### 14.1.1 Abstraction

L'**astrazione** permette di focalizzarsi su aspetti importanti di un problema ad un livello particolare, senza l'offuscamento dovuto a dettagli non importanti. Permette la descrizione di un sistema come struttura a livelli

### 14.1.2 Refinement

Il raffinamento (?) è un processo top-down dove, in ogni passo, una o più istruzioni sono decomposte in istruzioni più dettagliate, partendo da una specifica di alto livello e dividendola in sottoproblemi ricorsivamente, finché i sottoproblemi non hanno soluzioni immediate. Questo non è appropriato per sistemi di larga scala o distribuiti, ma piuttosto per il design di metodi.

### 14.1.3 Information hiding

Il design richiede una serie di decisioni, e per ognuna di queste, dobbiamo chiederci chi deve sapere e chi no. L'**information hiding** è strettamente legato a:

- **Abstraction**: se qualcosa è nascosto, l'utente può astrarre da quell'informazione *Scusate.*
- **Coupling**: un segreto diminuisce il coupling tra un modulo ed il suo environment
- **Cohesion**: il segreto è ciò che unisce le parti del modulo

### 14.1.4 Modularity

La modularità è basata sull'integrazione di componenti separati in moduli, per risolvere requirements di problemi. È basata sull'uso di unità linguistiche modulari, poche interfacce semplici ed esplicite, offuscamento delle informazioni. Un metodo di design può essere detto modulare solo se supporta:

- **Decomposability**: un problema software può essere diviso in un piccolo numero di piccoli sottoproblemi connessi da una struttura semplice, ed abbastanza indipendenti da permettere lavoro separato su ogni componente.
- **Composability**: alcuni elementi software possono essere combinati tra loro per produrre nuovi sistemi, possibilmente in ambienti diversi da quello di sviluppo
- **Understandability**: un lettore umano può comprendere ogni modulo senza dover conoscere gli altri, o conoscerne una minima parte
- **Continuity**: una piccola modifica nella specifica modificherà un solo modulo, o una minima parte
- **Protection**: l'effetto di una condizione anormale durante l'esecuzione coinvolgerà un solo modulo, o verrà propagata ad una minima parte

I principi per il design modulare sono: **linguistic modular units** (i moduli devono corrispondere alle unità del linguaggio, come pacchetti o moduli), **poche interfacce, poco scambio di informazioni tra moduli, interfacce esplicite** (se due moduli comunicano, dev'essere ovvio), **information hiding** (tutte le informazioni di un modulo dovrebbero essere private, se non specificatamente dichiarato il contrario, e per l'accesso bisogna utilizzare interfacce).

#### 14.1.5 Cohesion

La coesione misura la *chiusura* di una relazione tra elementi di un componente/classe. Una coesione forte è desiderabile perché semplifica le correzioni, le modifiche, le estensioni, riduce il testing e promuove il riutilizzo. Abbiamo più livelli di coesione (dal più basso al più alto):

- **Coincidental:** gli elementi non hanno relazioni ma sono uniti per convenienza
- **Logical:** elementi che svolgono funzioni simili (input simile, error handling)
- **Temporal:** elementi che sono attivati allo stesso tempo e hanno avvio e fine comuni
- **Procedural:** elementi che compongono una singola sequenza di controllo
- **Communicational:** elementi che operano sullo stesso input o producono lo stesso output
- **Sequential:** elementi che condividono o operano sugli stessi dati (l'output di uno è l'input dell'altro)
- **Functional:** elementi che sopperiscono a un singolo functional requirement
- **Object:** solo operazioni tra oggetti permettono modifiche o ispezioni agli oggetti

#### 14.1.6 Coupling

Misura l'interconnessione tra moduli. Quando è debole, i moduli sono fortemente indipendenti. Livelli di coupling dal più debole al più forte:

- **No direct:** nessuna dipendenza
- **Data:** solo i dati necessari sono passati come argomenti
- **Stamp:** le strutture dati sono passate per lista di argomenti e solo una parte è utilizzata
- **Control:** si interfaccia passando flags e altri parametri
- **External:** è legato a device o device drivers
- **Common:** entrambi i moduli utilizzano dati globali
- **Content:** modifica gli statement/dati dell'altro modulo o si ramifica nel mezzo di un modulo

Il tradeoff qui è:

- **Coupling elevato:** i componenti sono difficili da comprendere autonomamente, le modifiche causano problemi negli altri moduli, il riutilizzo è scoraggiato. In compenso, migliori performance
- **Coupling ridotto:** aumenta i costi in performance, ma è più veloce da sviluppare e mantenere.

## 15 T15 - Object oriented design principles

Illustriamo in questa lezione i principi **SOLID**.

## 15.1 Principi dell'Object-Oriented Design

### 15.1.1 SRP - Single Responsibility Principle

Il principio di singola responsabilità afferma che una classe deve avere una sola ragione di cambiare. Le modifiche ai requirements solitamente si mappano sulle responsabilità. Più responsabilità equivalgono a più probabilità di cambiare. Le responsabilità nella stessa classe sono coupled. Tante classi con responsabilità distinte equivalgono a un design più flessibile.

*In programming, the Single Responsibility Principle states that every module or class should have responsibility over a single part of the functionality provided by the software.*

### 15.1.2 OCP - Open Closed Principle

Un modulo o componente dev'essere **aperto verso l'estensione**, ma **chiuso verso la modifica**. In questo modo, si possono aggiungere nuovi comportamenti, ma le modifiche non sono richieste. Ogni modifica può introdurre bug e richiede lavoro addizionale. Al contrario, scrivere nuove classi più difficilmente genera problemi.

*In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification. If you have a general understanding of OOP, you probably already know about polymorphism. We can make sure that our code is compliant with the open/closed principle by utilizing inheritance and/or implementing interfaces that enable classes to polymorphically substitute for each other.*

### 15.1.3 Liskov Substitution Principle

Le subclasses dovrebbero essere sostituibili dalle loro base classes: le classi derivate devono onorare i contratti delle loro classi base, estendendone le funzionalità senza modificarle. Altrimenti, le classi derivate producono effetti indesiderati quando usate con moduli esistenti.

*More generally it states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

### 15.1.4 ISP - Interface Segregation Principle

Avere multiple interfacce client-specific è molto meglio di averne una generica. In questo modo, le interfacce sono semplici e focalizzate. Per questo è consigliabile un refactor delle interfacce grandi in più sottointerfacce, in modo da non forzare il client a dipendere da cose di cui non ha bisogno.

*In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. Put more simply: Do not add additional functionality to an existing interface by adding new methods. Instead, create a new interface and let your class implement multiple interfaces if needed.*

### 15.1.5 DIP - Dependency Inversion Principle

I moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Entrambi dovrebbero dipendere da astrazioni. Le astrazioni non devono dipendere da dettagli, ma i dettagli dovrebbero dipendere da astrazioni. Bisognerebbe inoltre evitare di derivare, associare o dipendere da classi concrete o componenti.

## 15.2 Principi di package cohesion

### 15.2.1 REP - Release/Reuse Equivalency Principle

L'unità di riuso è l'unità di release: il codice non dovrebbe essere riutilizzato ricopiandolo e incollandolo. In questo modo, una modifica alla libreria originale verrebbe persa. Invece, il codice dovrebbe essere inserito tramite una release della libreria, in modo da poterla mantenere aggiornata. Un pacchetto costituito da classi riutilizzabili è più utile e riutilizzabile a sua volta. Pacchetti che non c'entrano nulla non dovrebbero essere inclusi.

### 15.2.2 CCP - Common Closure Principle

Le classi che cambiano insieme, devono stare insieme: questo minimizza l'impatto delle modifiche. Le classi dovrebbero essere *packaged* con coesione, indirizzando la stessa area funzionale o comportamentale, ed essendo inseparabili ed interdipendenti.

### 15.2.3 CRP - Common Reuse Principle

Le classi che non sono riutilizzabili insieme non dovrebbero essere raggruppate, quelle raggruppate dovrebbero cambiare per gli stessi motivi.

## 15.3 Package Coupling Principles

### 15.3.1 ADP - Acyclic Dependencies Principle

Non bisogna permettere cicli nel grafico delle dipendenze. Non bisogna interferire con i programmati. *Sono gente cattiva.*

### 15.3.2 SDP - Stable Dependencies Principle

Le dipendenze tra componenti dovrebbero essere nella direzione della stabilità: un componente deve dipendere solo da componenti più stabili. Ogni volta che un pacchetto cambia, tutti i pacchetti che ne dipendono devono essere validati per assicurare che lavorino correttamente dopo la modifica. Più pacchetti dipendono da un pacchetto instabile, più grande è il danno quando cambia.

### 15.3.3 SAP - Stable Abstractions Principle

Un pacchetto dev'essere tanto più astratto quanto è stabile. I pacchetti astratti dovrebbero essere responsabili e indipendenti, quelli concreti irresponsabili e dipendenti. Un'architettura ideale ha la seguente struttura: pacchetti instabili in cima, pacchetti stabili alla base.

## 15.4 Attività per un buon design

- **Dividi e conquista:** avere a che fare con qualcosa di grande tutto in una volta è molto più difficile di avere a che fare coi sottoproblemi.
- **Aumentare la coesione** quando possibile: un sistema con alta coesione tiene insieme cose che devono stare insieme e tiene fuori il resto
- **Ridurre il coupling**, ossia le interdipendenze tra moduli
- **Aumentare l'astrazione** il più possibile, allo scopo di nascondere i dettagli e ridurre la complessità

- **Aumentare la riutilizzabilità** quando possibile
- **Riutilizzare design e codice esistenti**
- Pensare alla **flessibilità**: anticipare le modifiche future è un buon investimento a lungo termine
- **Anticipare l'obsolescenza**: pianificare modifiche nella tecnologia o nell'enivroment del futuro
- Pensare alla **portabilità**
- Pensare alla **testabilità**
- Rimanere sempre sulla **difensiva**: non sappiamo mai se l'utente farà davvero ciò che deve.

## 16 T16 - Design Patterns

*Si, hai letto bene. Sei fottuto. Cominciamo.* La modellazione richiede di superare i nostri limiti mentali; un buon modello lo fa: è facile da ricordare, poco complesso, astratto, strutturato. Per comunicare un modello utilizziamo la navigazione e la riduzione di complessità, partendo da un modello molto semplice decorato di mano in mano. Per ridurre la complessità, cerchiamo ereditarietà e patterns. Nel OOP abbiamo strutture ricorrenti che promuovono astrazione, flessibilità, modularità, eleganza. Queste strutture sono un'importante conoscenza per lo sviluppo di ogni tipo di sistema, il problema è trovare il modo giusto di esprimerele. Un **design pattern** è una soluzione ricorrente ad un problema comune, che può essere usata in modi diversi. È dipendente da linguaggio ed implementazione. I design patterns hanno vari obiettivi:

- Codificare un buon design
- Dare nomi esplicativi alle strutture
- Catturare e preservare le informazioni sul design
- Facilitare la ristrutturazione/refactoring

Abbiamo due forme possibili: **Alexandrian** e **GoF**.

### 16.1 Creational Patterns

I creational patterns permettono la creazione flessibile di oggetti per rendere il sistema indipendente dagli oggetti creati. Ogni famiglia e le classi istanziate possono essere conosciute solo a runtime.

#### 16.1.1 Factory Method

**Scopo** Il metodo Factory fornisce un'interfaccia per creare oggetti in una subclass, permettendole di modificare il tipo di oggetti creati.

**Problema** Immaginiamo di dover creare un'applicazione che gestisca un servizio di logistica. La prima versione utilizza solo trasporto su ruota, quindi la maggior parte del codice sta nella classe `Truck`. Dopo un po', l'app diventa popolare. Decidi di implementare anche il trasporto via mare. Aggiungere la classe `Ship` richiederebbe modifiche in tutto il codice. Aggiungerne altre, pure.

**Soluzione** Il metodo Factory permette di sostituire la creazione di nuovi oggetti (con `new`) con chiamate a un metodo factory. Gli oggetti generati da questo metodo sono detti **prodotti**. C'è un vincolo da rispettare: i prodotti devono implementare un'interfaccia comune.

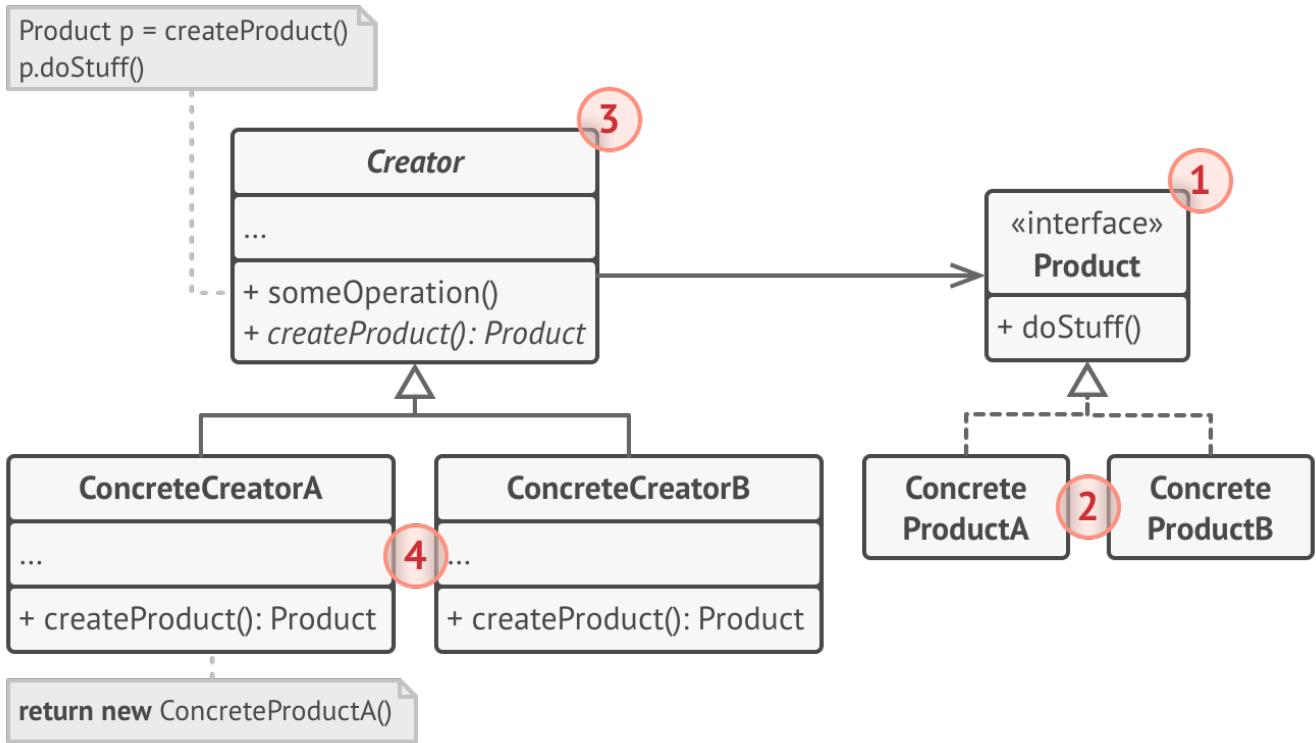


Figura 18: Esempio generico

## Applicabilità

- Una classe non può anticipare la classe degli oggetti da creare
- Una classe delega responsabilità a delle helper subclasses

### 16.1.2 Abstract Factory

**Scopo** L'**abstract factory** permette di creare famiglie di oggetti related senza specificarne le classi concrete.

**Problema** Immaginiamo di voler creare un simulatore di negozio di arredamenti. Il codice consiste di classi rappresentanti 1) una famiglia di prodotti (sedia, divano, tavolo) 2) varianti di questa famiglia. Serve un modo di creare oggetti in modo da matcharli con altri arredamenti della stessa famiglia. Non vogliamo inoltre modificare il codice esistente quando aggiungiamo nuovi prodotti o famiglie.

**Soluzione** Prima di tutto, dichiariamo le interfacce di ogni distinto prodotto, avendo come sottoclassi le varianti del suddetto. In secundis, dichiariamo l'**abstract factory**, un'interfaccia con una lista di metodi di creazione facenti parte di una famiglia di prodotti. Questi metodi ritornano prodotti astratti. A questo punto, estendiamo l'**abstract factory** con le **factory concrete** per ogni variante.

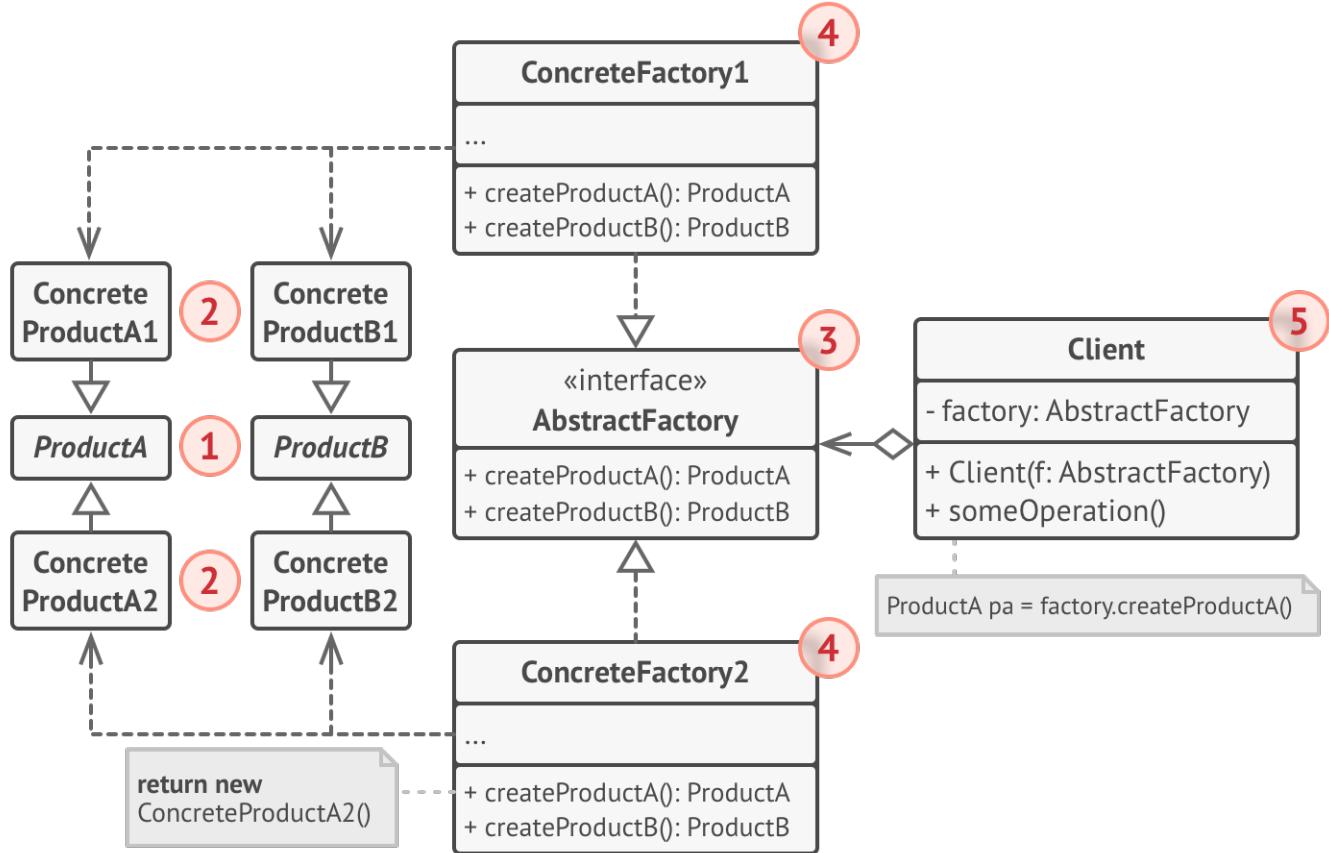


Figura 19: Struttura

**Applicabilità** Quando i client non possono anticipare gruppi di classi da istanziare.

### 16.1.3 Builder

**Scopo** Il **builder** permette di costruire oggetti un pezzo alla volta. Il pattern permette di produrre tipologie e rappresentazioni diverse di un oggetto, utilizzando lo stesso codice di costruzione.

**Problema** Immaginiamo un oggetto complesso che richiede inizializzazioni step-by-step di molti campi e oggetti annidati. Questo porterebbe ad un gigante costruttore con molti parametri. *Bella merda*. Per esempio, se avessimo un oggetto **House**, dovremmo costruire 4 muri, installare una porta, le finestre, un tetto. Se volessimo però qualche caratteristica in più, come un garage, dovremmo inserire un booleano tipo **hasGarage** nel costruttore.

**Soluzione** Il builder pattern suggerisce di estrarre il codice di costruzione dell'oggetto fuori dalla sua classe, ad oggetti detti **builders**. Possiamo andare oltre e creare una classe che chiama gli step dei builder, detta **director**, che definisce l'ordine delle chiamate, mentre i builder si occupano dell'implementazione. Il director non è strettamente necessario.

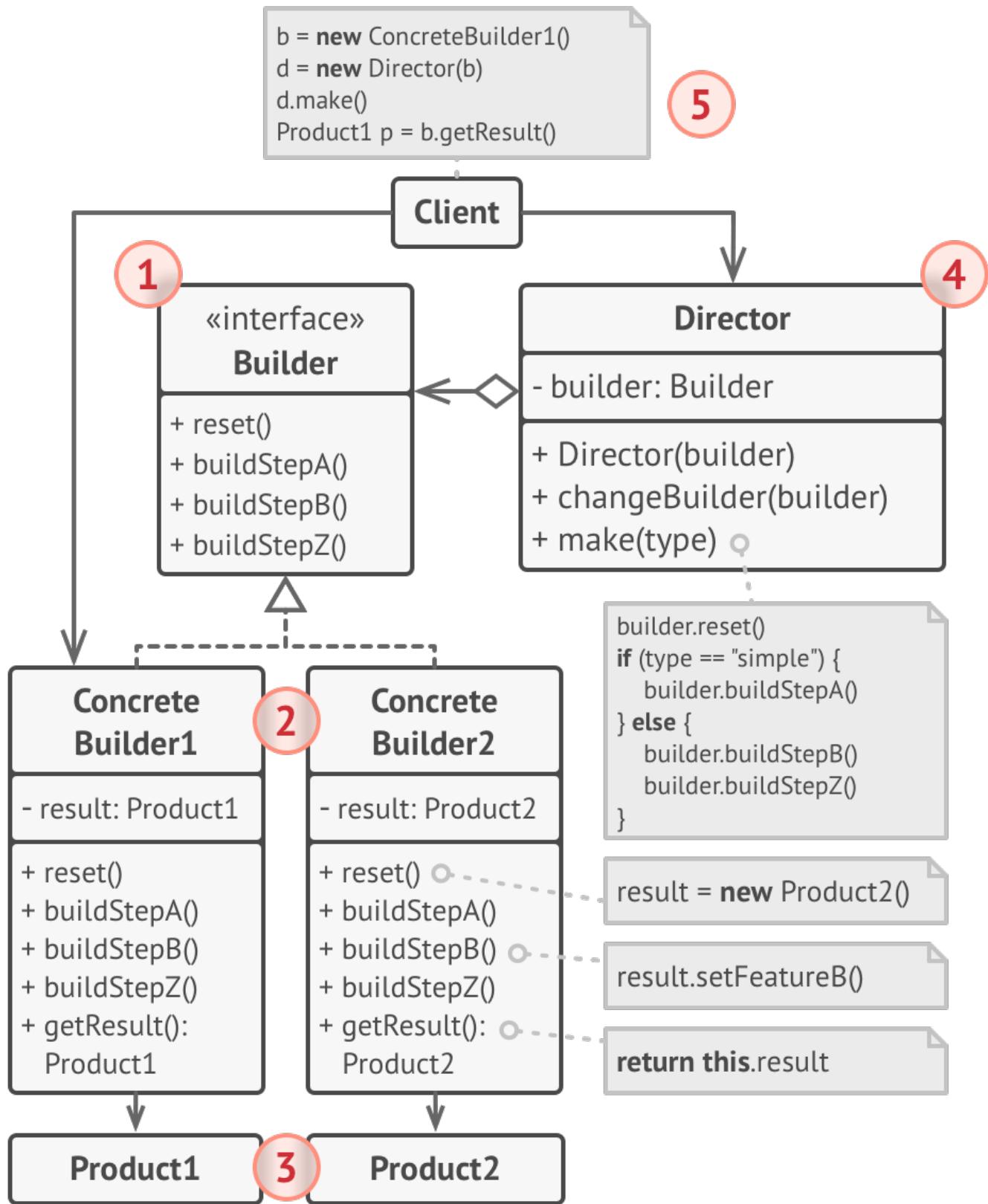


Figura 20: Struttura

**Applicabilità** Si applica quando l'algoritmo di creazione dell'oggetto deve essere indipendente dalle parti che lo compongono e come vengono assemblate. Insomma, quando il processo di costruzione deve permettere prodotti diversi.

#### 16.1.4 Prototype

**Scopo** Il **prototype** permette di copiare oggetti esistenti senza dover rendere il codice dipendente dalla classe effettiva.

**Problema** Immaginiamo di avere un oggetto, e volerne creare una copia esatta. Come fare? Farlo dall'esterno non è ideale: ci perderemmo tutti i dati privati! Inoltre, per fare il duplicato dobbiamo conoscere la classe concreta dell'oggetto. *Not stonks*.

**Soluzione** Il prototype pattern delega il processo di clonazione all'oggetto effettivo che stiamo copiando. Dichiariamo un'interfaccia comune, tipo **Clonable**, che contenga il metodo **clone()**. In questo modo, implementando il metodo **clone()** in tutte le classi (saranno metodi simili) abbiamo la possibilità di clonare oggetti. Un oggetto che supporta la clonazione è detto *prototype*.

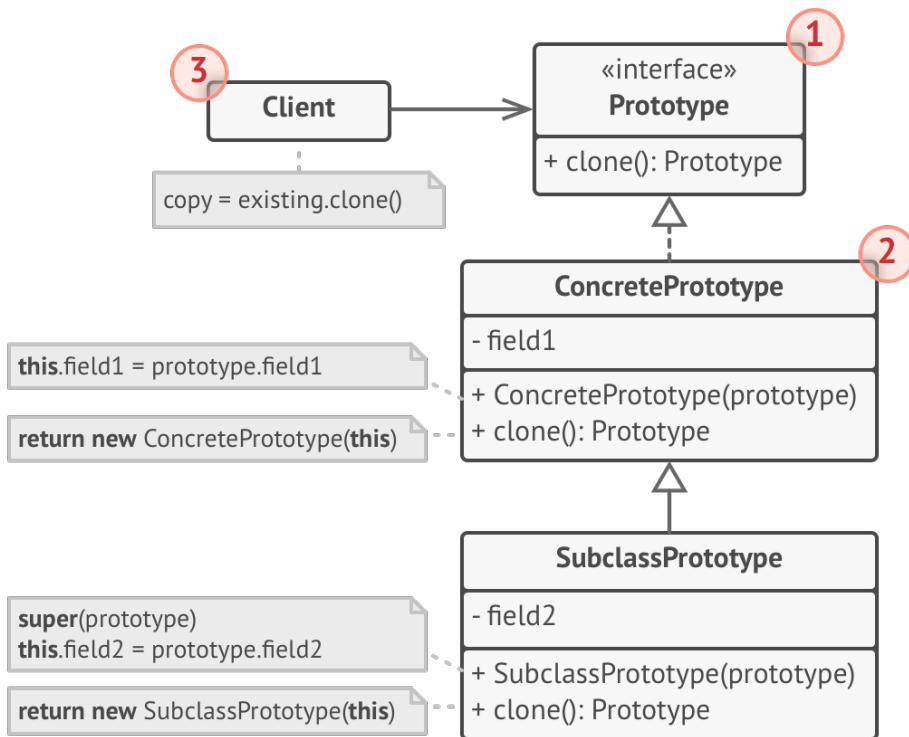


Figura 21: Struttura

**Applicabilità** Il pattern è applicabile quando le classi sono specificate a runtime, per evitare di costruire gerarchie di factories parallele alle gerarchie dei prodotti. Si può utilizzare quando le istanze di oggetti possono avere un numero finito di stati.

### 16.1.5 Singleton

**Scopo** Un singleton permette di assicurarsi che una classe abbia una sola istanza, fornendone un punto di accesso globale.

**Problema** Il singleton pattern risolve due problemi allo stesso tempo, ma **violando il Single Responsibility Principle**:

1. Si assicura che una classe abbia una sola istanza: questo torna utile quando, ad esempio, lavoriamo su dati di una risorsa condivisa, sia essa un DB, un file...
2. Fornisce un punto di accesso globale alla risorsa

**Soluzione** Tutte le implementazioni del singleton hanno questi due step in comune:

- Rendere il costruttore di default privato, in modo da non poter istanziare oggetti da fuori il singleton
- Creare un metodo di `getInstance` che faccia da costruttore. Se esiste già un'istanza la restituisce, altrimenti la crea e restituisce.

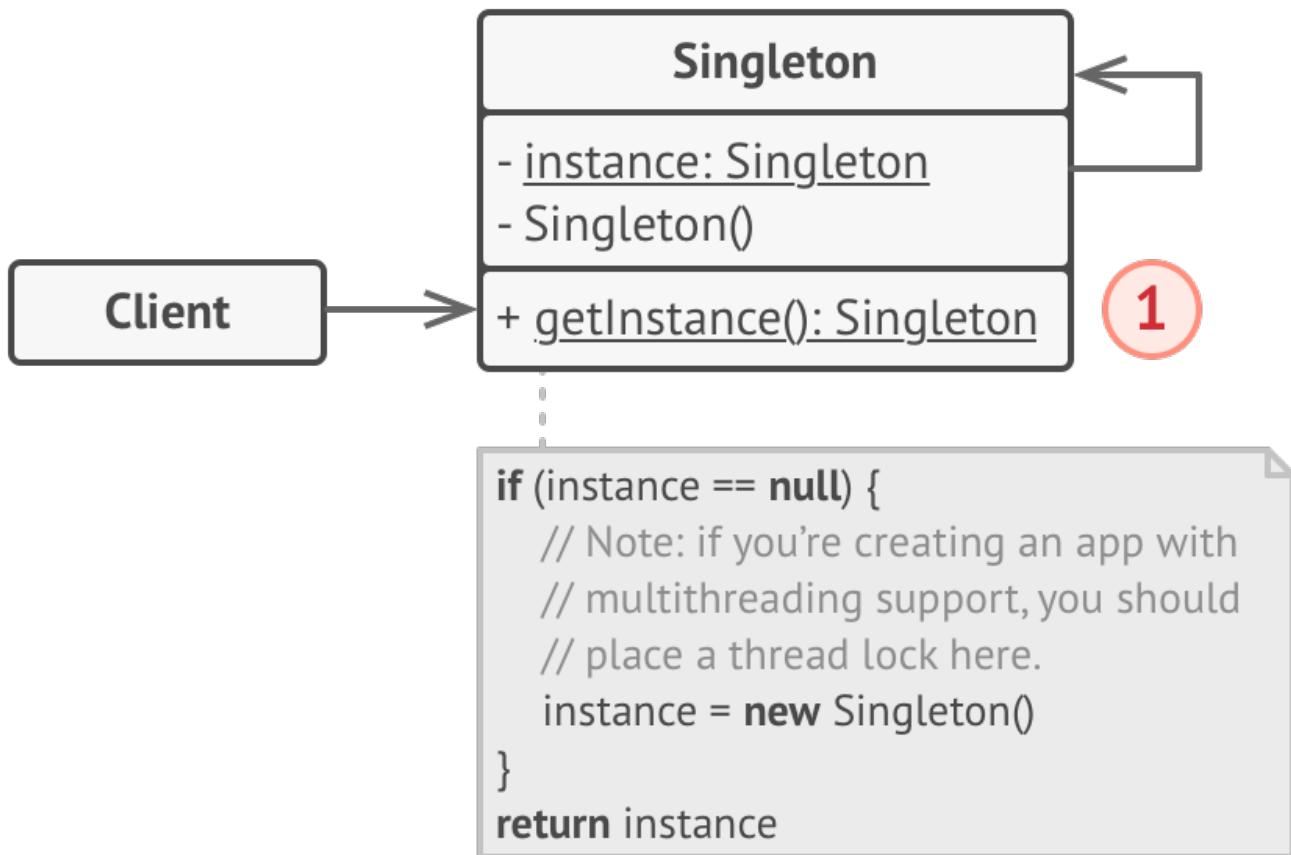


Figura 22: Struttura

**Applicabilità** Lo utilizziamo quando è necessaria una sola istanza di una classe, che deve essere globalmente accessibile.

## 16.2 Structural Patterns

### 16.2.1 Adapter

**Scopo** L'adapter permette ad oggetti con interfacce incompatibili di collaborare.

**Problema** Immaginiamo di star creando un'app di monitoraggio del mercato in borsa. L'app scarica i dati in XML da un'API. Ad un certo punto, vogliamo integrare un'altra API, che però restituisce dati in JSON. Potremmo cambiare questa libreria, ma sarebbe un casino.

**Soluzione** Creiamo un adattatore, che converta l'interfaccia di un oggetto in un'altra, in modo da farla comprendere all'oggetto. Un adapter wrappa l'oggetto in modo da nasconderne la complessità: chi lo utilizza non è neanche al corrente del fatto che sia un adapter.

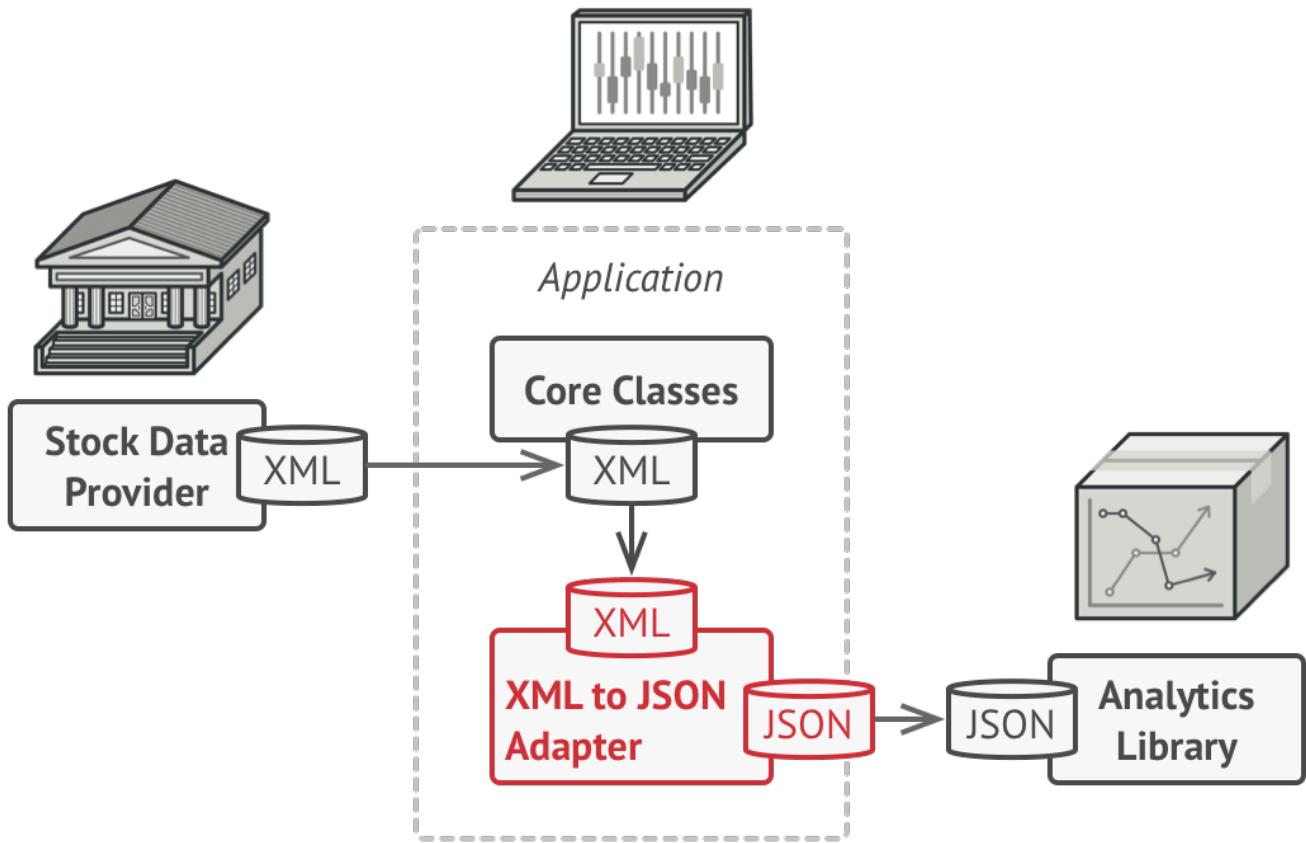


Figura 23: Struttura

**Applicabilità** Lo utilizziamo quando una classe esistente fornisce la funzionalità richiesta in parte, ma non implementa l'interfaccia necessaria.

### 16.2.2 Bridge

**Scopo** Il bridge permette di splittare un grande numero di classi in due gerarchie separate, astrazione e implementazione, sviluppabili indipendentemente.

**Problema** Ipotizziamo di avere una classe **Shape** con due subclasses: **Circle** e **Square**. Vogliamo estendere questa gerarchia per incorporare i colori, quindi inizialmente proponiamo di creare sottoclassi **RedCircle**, **BlueCircle**, **RedSquare**, **BlueSquare**. È palese che questo approccio non sia sostenibile per il futuro.

**Soluzione** Semplicemente, facciamo diventare il colore una proprietà dell'oggetto **Shape**. In questo modo, switchiamo da ereditarietà a composizione.

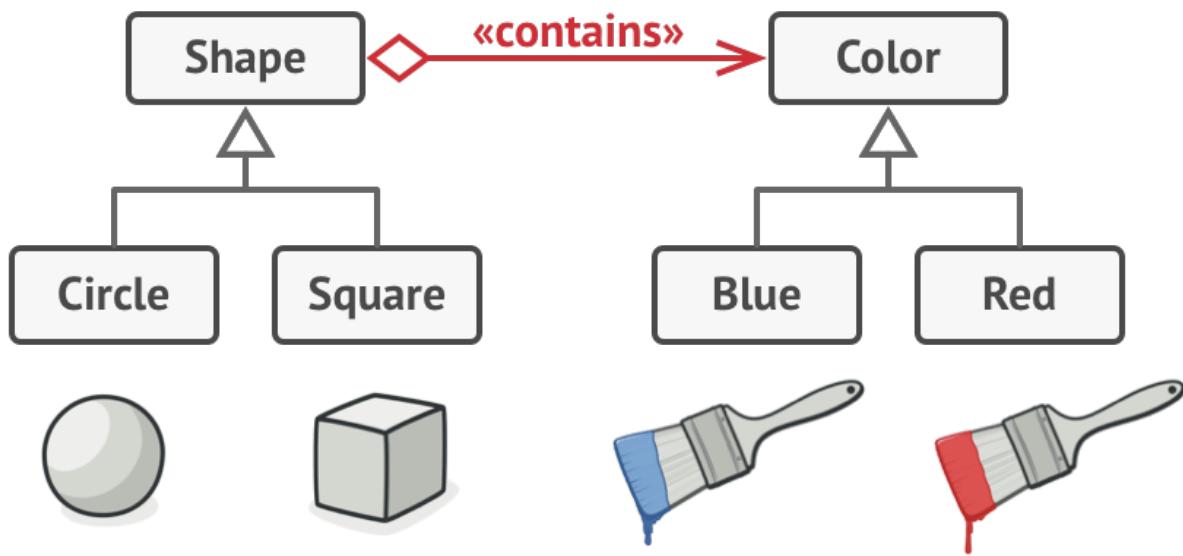


Figura 24: Esempio

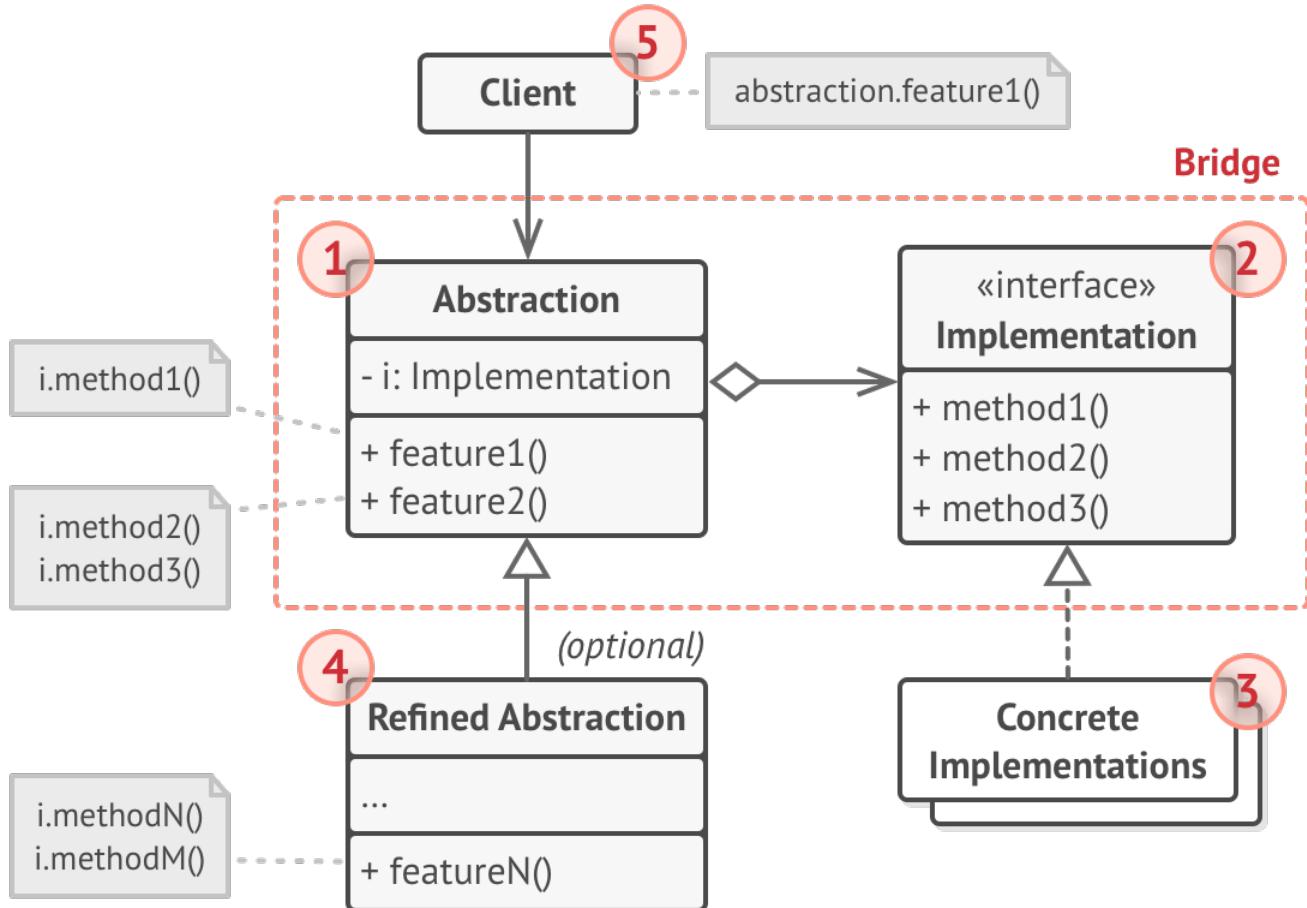


Figura 25: Struttura

**Applicabilità** Utilizziamo il bridge quando interfaccia ed implementazione devono variare indipendentemente. Richiede un’interfaccia comune tra gerarchie intercambiabili di classi.

### 16.2.3 Adapter vs. Bridge

Entrambi sono utilizzati per nascondere dettagli implementativi. L’adapter fa lavorare insieme componenti incompatibili, mentre il bridge permette ad astrazioni ed implementazioni di variare indipendentemente.

### 16.2.4 Composite

**Scopo** Il **composite pattern** permette di comporre oggetti in strutture ad albero e lavorare su queste strutture come oggetti individuali.

**Problema** Utilizzare il composite pattern ha senso solo quando il core del software è rappresentabile da un albero. Per esempio, immaginiamo di avere due tipi di oggetti: **Prodotto** e **Scatola**. Le scatole contengono, ovviamente, più oggetti, oppure altre scatole. Potremmo tentare un approccio diretto, ma sarebbe computazionalmente *una merda*.

**Soluzione** Il composite pattern suggerisce di lavorare su prodotti e scatole attraverso un'unica interfaccia, che dichiara un metodo per calcolare il costo totale. In un prodotto, restituirebbe il costo del prodotto. In una scatola, restituisce la chiamata ricorsiva a tutti i contenuti. *Avete già capito dove voglio arrivare.* In questo modo, basta chiamare il metodo di calcolo sulla scatola grande e verrà chiamato ricorsivamente su tutti i nodi. *Spaventosamente elegante.*

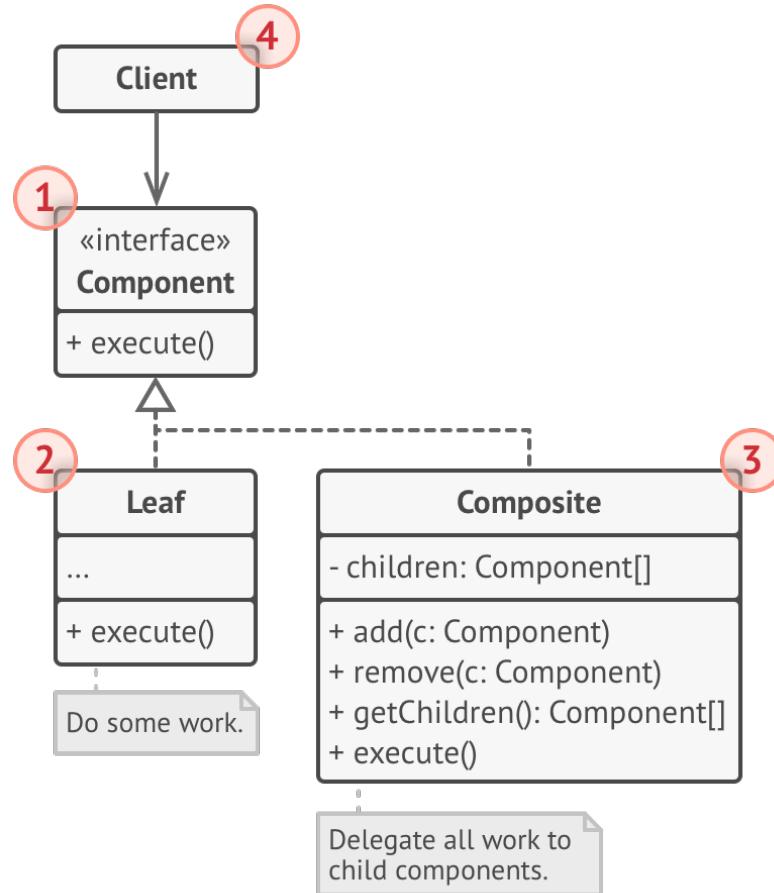


Figura 26: Struttura

**Applicabilità** È utile quando vogliamo rimuovere responsabilità a singoli oggetti dinamicamente, ove l'estensione per subclasses non è pratica.

### 16.2.5 Decorator

**Scopo** Il decorator permette di *agganciare* nuovi comportamenti ad oggetti, inserendoli in oggetti *wrapper* che contengono questi nuovi behaviors.

**Problema** Immaginiamo di stare lavorando ad una libreria per notifiche, che permette ad altri software di notificare gli utenti. La versione iniziale è basata sulla classe **Notifier** che ha pochi field, un costruttore, ed un singolo metodo `send()`. Questo metodo accetta un messaggio e lo invia agli utenti specificati nel costruttore. Ad un certo punto, realizzi di voler implementare altri tipi di notifiche, come messaggi Telegram o sms. Estendi quindi la classe **Notifier** con delle subclasses, come **FacebookNotifier** o **SMSNotifier**.

Ti chiedono, però, di poterne usare più di uno alla volta, e sarebbe da idioti cominciare a fare subclasses per ogni combinazione di notificatori.

**Soluzione** L'ereditarietà, in questo caso, non è la soluzione. Ci spostiamo sulla aggregazione/composizione. Il pattern Decorator è detto anche **Wrapper**. Questo esprime chiaramente l'idea di fondo: creiamo un oggetto wrapper che contiene i metodi del *target* e li chiama, però svolge operazioni che alterano il risultato, prima o dopo la chiamata. Generiamo così una struttura in cui **Notifier** possiede un oggetto di interfaccia **BaseDecorator**, che ha come subclasses i vari decorator. Generiamo così uno stack, che passiamo alla generazione di un nuovo decorator. Esso, tramite **super**, chiama i vari decorator della stack *in salita*. L'unico decorator che effettivamente viene chiamato è sempre l'ultimo.

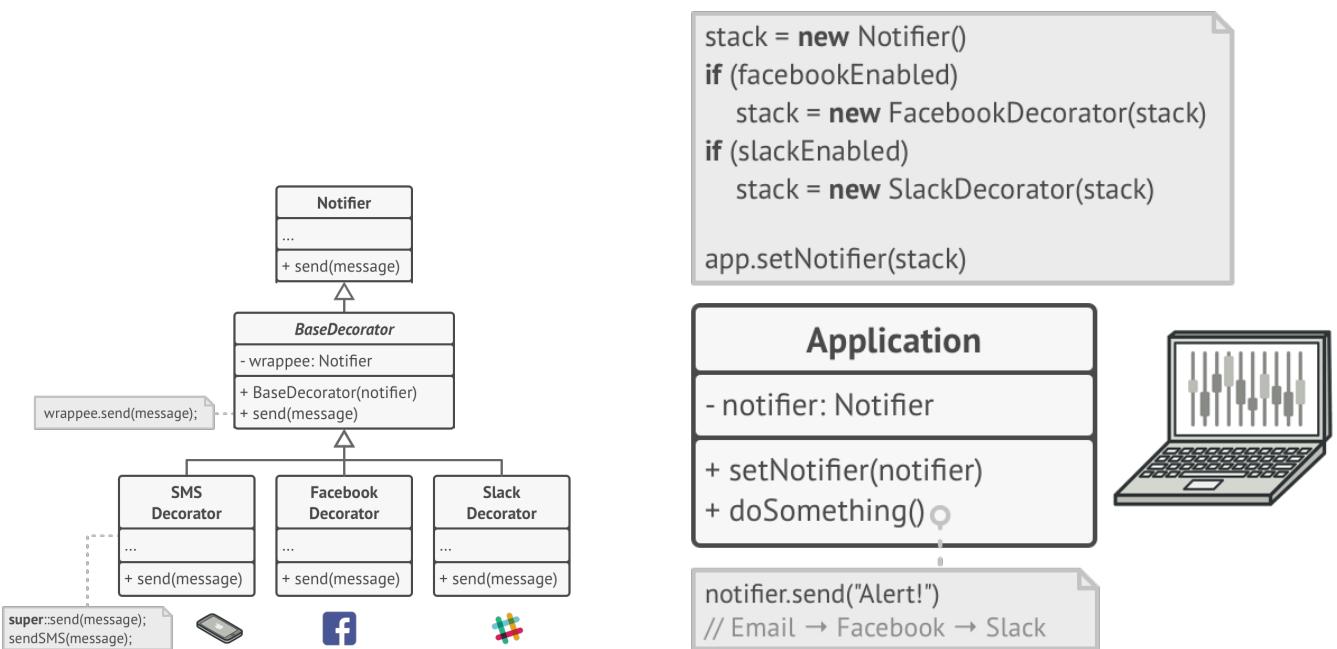


Figura 27: Struttura

**Applicabilità** Lo utilizziamo quando è utile aggiungere/rimuovere comportamenti agli oggetti **dinamicamente**, e l'estensione per subclasses non è conveniente.

### 16.2.6 Facade

**Scopo** Una **facade** è un pattern che fornisce un'interfaccia semplificata ad una libreria, un framework, o qualsiasi altro set complesso di classi.

**Problema** Immaginiamo di dover far funzionare il nostro codice con un set molto grande di oggetti che appartengono ad un sofisticato framework. Dovremmo, prima di tutto, inizializzare tutti gli oggetti, tenere traccia delle dipendenze, eseguire metodi nell'ordine corretto, e via dicendo. Il funzionamento del software diventerebbe quindi molto dipendente da questo framework.

**Soluzione** Una facade è una classe che fornisce una semplice interfaccia a un sottosistema complesso che contiene molti componenti. Essa fornisce funzionalità limitate, ma sufficienti all'utilizzo specifico che serve.

**Applicabilità** La utilizziamo per fornire un’interfaccia semplice a un sottosistema complesso, e *decouplare* le classi del subsystem dal nostro software.

**Struttura ideale di un subsystem** Un buon subsystem è basato su:

- Un’interfaccia
- Oggetti di controllo
- Oggetti di entità

E fa utilizzo di design pattern, come Adapter e Bridge.

### 16.2.7 Flyweight

**Scopo** Il **flyweight** permette di ridurre lo spazio di memoria necessario al salvataggio di oggetti, salvando le parti in comune.

**Problema** Immaginiamo di dover creare un videogioco sparacielo. Decidiamo di implementare un sistema di particelle complesso, in modo che proiettili, missili, detriti siano gestiti da oggetti. Avremmo quindi una struttura in cui ogni particella occupa molto spazio in memoria:

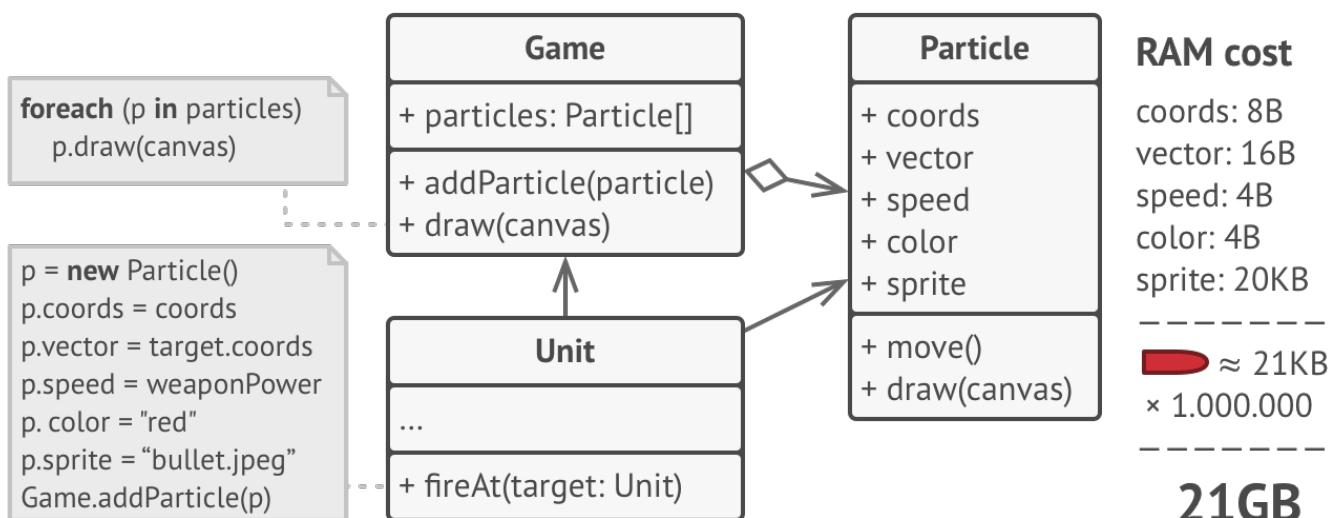


Figura 28: Struttura iniziale

**Soluzione** Notiamo però che molte informazioni sono ripetute! Infatti, ad esempio, colore e sprite, che occupano moltissima memoria, contengono quasi sempre gli stessi dati. Decidiamo, allora, di salvare le informazioni comuni in una classe a parte, in modo da salvare solo il minimo indispensabile per le singole particelle.

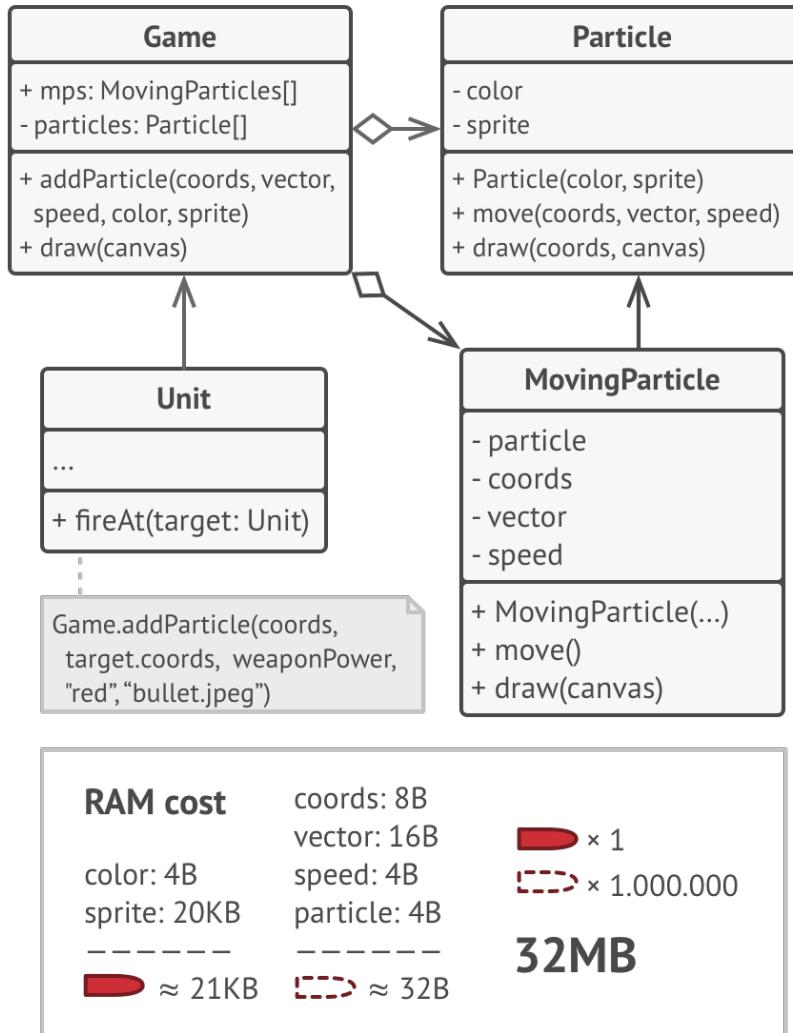


Figura 29: La soluzione usando il Flyweight pattern

**Applicabilità** Utilizziamo questo pattern quando abbiamo un **grande numero di oggetti**, con costi in memoria alti, stati che possono essere estratti dall'oggetto, ed un'applicazione che non dipende dall'identità degli oggetti.

#### 16.2.8 Proxy

**Scopo** Il **proxy** permette di avere un sostituto ad un oggetto, controllandone l'accesso da parte degli altri, e potendo eseguire operazioni prima e dopo.

**Problema** Perché dovremmo controllare l'accesso? Per esempio, se avessimo un oggetto che usa molte risorse di sistema ma che non utilizziamo sempre, sarebbe utile avere un *lazy loading*, ossia un'istanziazione solo quando necessario.

**Soluzione** Creiamo quindi una classe proxy che abbia la stessa interfaccia dell'oggetto desiderato. In questo modo, agli oggetti che lo richiedono, lo passiamo al posto dell'oggetto vero e proprio. Il proxy

crea quindi l'oggetto vero solo dopo la prima richiesta. Se, inoltre, volessimo filtrare i risultati ottenuti, potremmo farlo, un po' come accade in un Decorator.

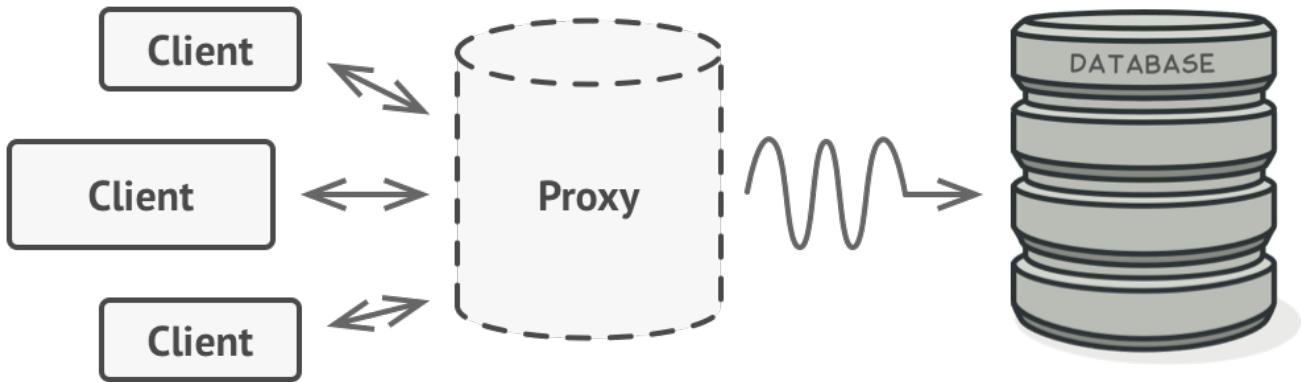


Figura 30: La soluzione usando il Flyweight pattern

**Applicabilità** Lo utilizziamo quando abbiamo necessità di **controllare l'accesso** ad un oggetto, e necessitiamo di una reference sofisticata al suddetto.

**Tipologie** Abbiamo più tipologie di proxy:

- Virtual proxy (expensive objects)
- Cache proxy (temporary objects)
- Remote proxy (remote objects)
- Protection proxy (shared objects)
- Smart reference (pointer extension)

## 16.3 Behavioral patterns

A volte è necessario eseguire richieste agli oggetti senza conoscenze sull'operazione richiesta o il destinatario della stessa. Un esempio possono essere le librerie grafiche per bottoni e menù, che non implementano l'azione ma passano la richiesta. I behavioral patterns si occupano di questo: gestiscono l'assegnamento di responsabilità tra oggetti.

### 16.3.1 Chain of responsibility

**Scopo** La **chain of responsibility** permette di passare richieste attraverso una catena di *handlers*. Quando riceve una richiesta, ogni handler decide se processarla o passarla al prossimo handler della catena.

**Problema** Immaginiamo di lavorare a un sistema di ordini online. Vogliamo restringere l'accesso al sistema, in modo che solo gli utenti autenticati possano ordinare. Vogliamo inoltre degli amministratori, che possano vedere tutti gli ordini. Decidiamo che questi controlli devono essere fatti in sequenza: se l'autenticazione fallisce, non è necessario fare altre verifiche. Col tempo, si aggiungono altre verifiche: un sanitizing dei dati, una verifica degli IP, un motore di caching. Ad ogni verifica aggiunta, il codice diventa un porcilaio.

**Soluzione** Trasformiamo quindi queste verifiche in *handlers*: ognuna avrà un metodo `check`, ed un campo per il prossimo handler della catena. Ad ogni richiesta, l'handler eseguirà le sue verifiche: se è tutto ok, passerà la richiesta al prossimo handler, altrimenti fermerà il tutto. È fondamentale che tutti gli handler implementino una stessa interfaccia.

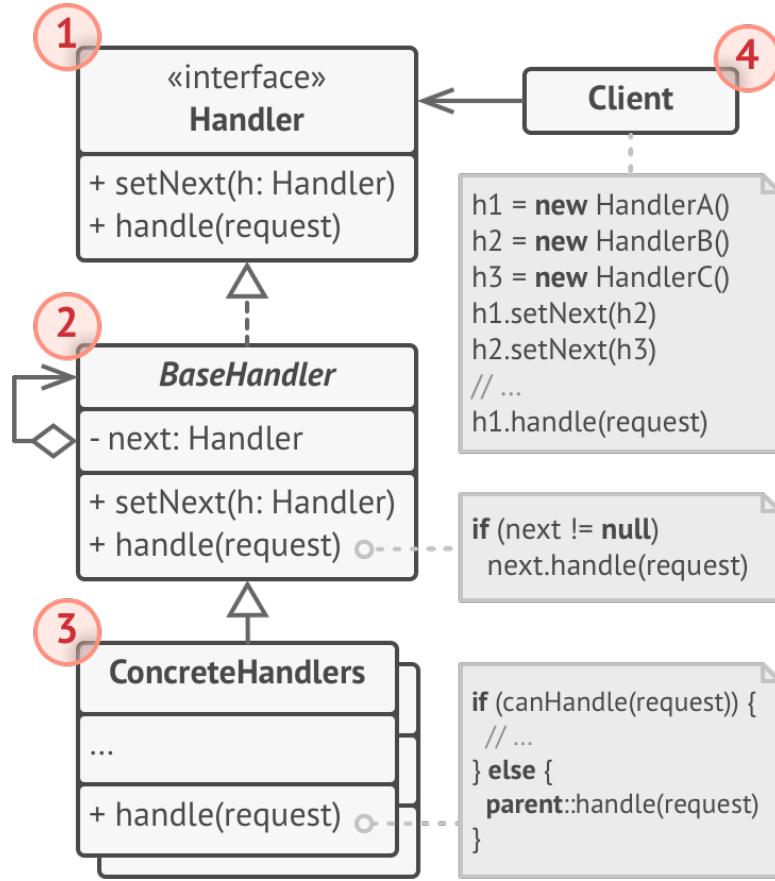


Figura 31: La soluzione

**Applicabilità** Utilizziamo questo pattern quando più di un oggetto deve eseguire un comando, l'handler non è conosciuto in anticipo e non può essere determinato automaticamente.

### 16.3.2 Command

**Scopo** Il **command** trasforma una richiesta in un oggetto *stand-alone* che contiene tutte le informazioni che la riguardano. Questo pattern può essere utilizzato per *decouplare* i boundary e control objects.

**Problema** Immaginiamo di lavorare ad un software di text editing. Vogliamo creare una toolbar che contenga tasti per le operazioni più importanti, con una classe `Button`. Cominciamo a creare subclasses per ogni bottone necessario: `OKButton`, `SaveButton`, `ApplyButton`, ed ogni tasto che si aggiunge, *la nostra voglia di vivere cala*. Dev'esserci un modo più furbo.

**Soluzione** Estraiamo i dati della richiesta tramite un oggetto dall'interfaccia `Command`, che avrà come implementazioni i vari tipi di comando possibili, aventi un solo metodo `execute`. In questo modo, ogni

bottone chiamerà l'`execute` del suo `Command`, ma non solo: i comandi potrebbero, ad esempio, anche essere chiamati da scorciatoie da tastiera!

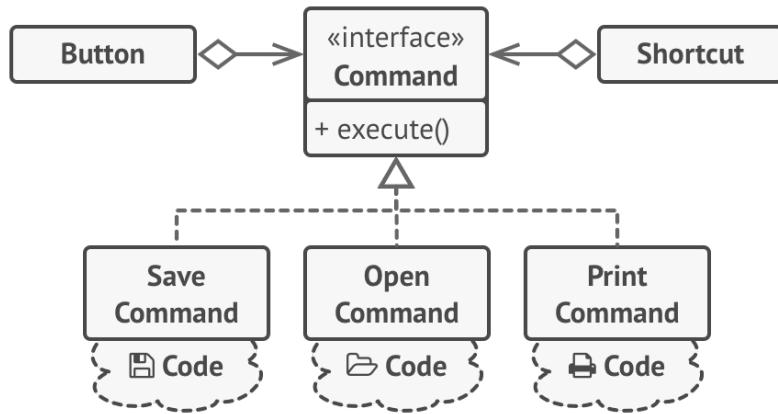


Figura 32: Soluzione usando un Command

**Applicabilità** Lo utilizziamo per parametrizzare oggetti con un’azione da eseguire, specificando le queue di azioni ed eseguire richieste in tempi diversi. Supportiamo così l’undo/redo.

### 16.3.3 Interpreter

**Scopo** L’**interpreter**, dato un linguaggio, definisce una rappresentazione per la sua grammatica ed un modo di interpretarne le frasi. Mappa un dominio ad un linguaggio, il linguaggio ad una grammatica, e la grammatica ad un design object-oriented gerarchico.

**Problema** Abbiamo una classe di problemi appartenenti ad un dominio ben definito. Se il dominio è caratterizzato da un linguaggio, il problema può essere risolto tramite un interprete.

**Soluzione** Modelliamo il dominio con una grammatica ricorsiva ad albero, in cui ogni regola è un nodo o una foglia.

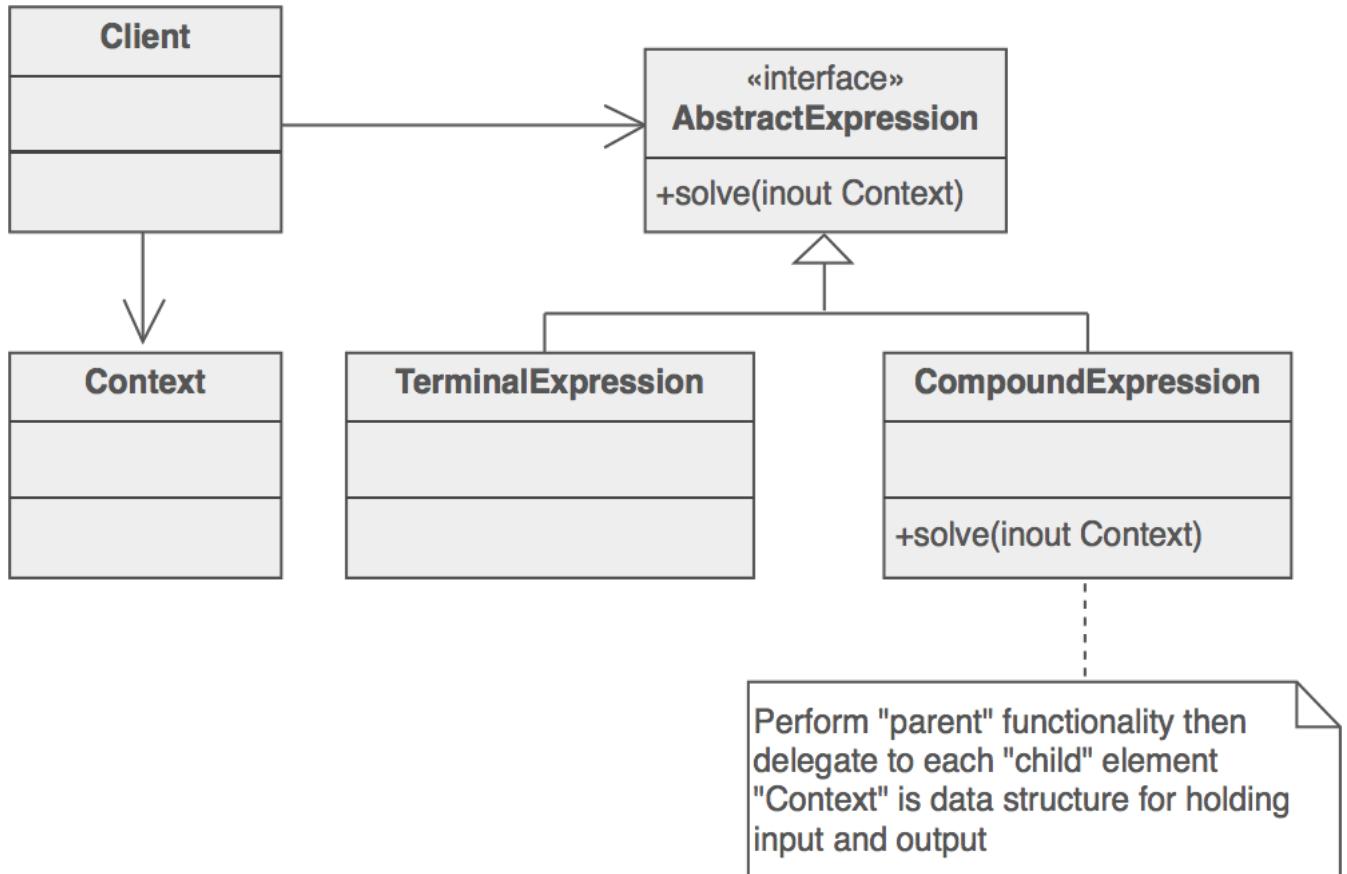


Figura 33: Struttura di un interpreter

**Applicabilità** Utilizzabile quando abbiamo un linguaggio facilmente mappabile con un albero, con grammatica semplice ed efficienza non importante.

#### 16.3.4 Iterator

**Scopo** L'**iterator** permette di attraversare gli elementi di una collection senza esporne le logiche di rappresentazione.

**Problema** Creiamo un nostro oggetto di collection per salvare dati, con una struttura *cazzutissima* ed ultra segreta, che ci permetterà di diventare milionari grazie all'intelligentissimo metodo di esplorazione dei dati che ho. Abbiamo ora un problema: non vogliamo esporre le logiche con cui esploriamo la collection.

**Soluzione** Estraiamo l'esplorazione della collection tramite un oggetto *iterator*. Potremmo addirittura avere più iterators, che implementino un'interfaccia costituita da: un metodo `getNext`, un metodo `hasNext`, ed un campo che salvi l'elemento corrente.

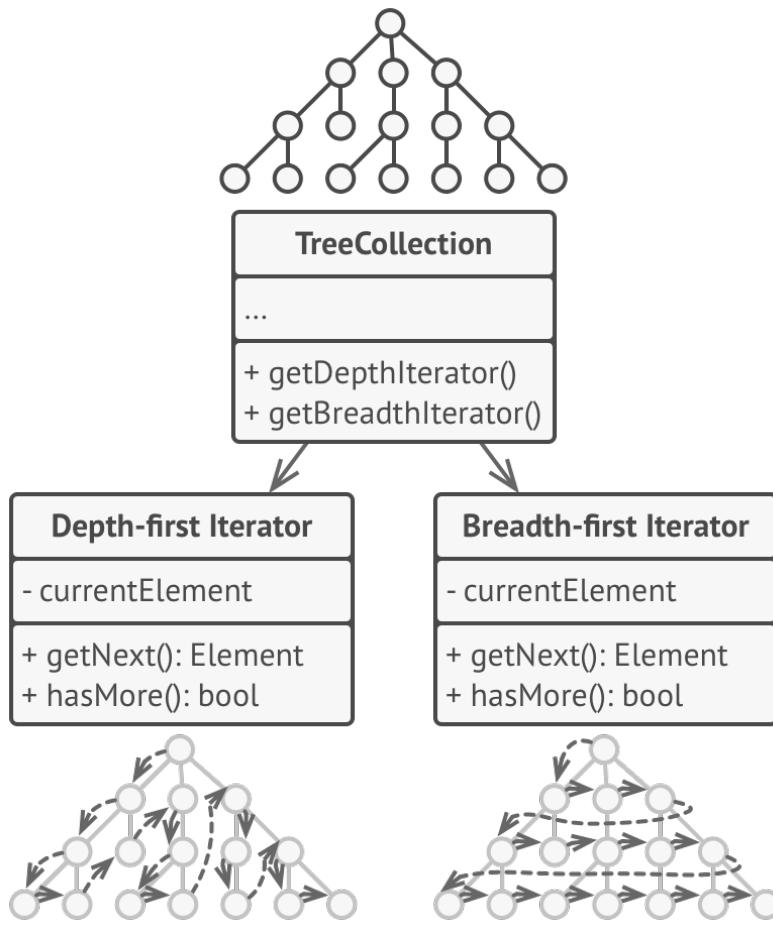


Figura 34: Struttura di un iterator multiplo

**Applicabilità** Lo utilizziamo per accedere ad una collection senza esporne l'implementazione, supportando esplorazioni simultanee, e fornendo un'unica interfaccia per l'esplorazione di collections diverse.

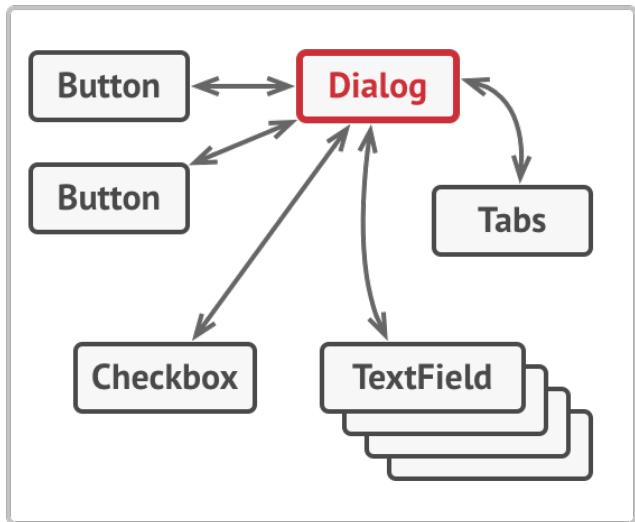
### 16.3.5 Mediator

**Scopo** Il **mediator** permette di ridurre le caotiche dipendenze tra oggetti. Esso restringe la comunicazione diretta tra gli oggetti e li forza a collaborare solo tramite un oggetto mediator.

**Problema** Immaginiamo di avere un dialogo di creazione e modifica dei profili utente, composto da textfields, checkboxes, bottoni. Alcuni di questi elementi devono collaborare: ad esempio, selezionando la checkbox "Sono Guido Soncini", il titolo diventerà "Reggio Merda". Applicare questa logica all'interno degli elementi è un bagno di sangue.

**Soluzione** Creiamo quindi un oggetto **Mediator** che regoli la comunicazione tra questi oggetti, chiamato dai suddetti a necessità. Nel nostro esempio, la classe **Dialog** potrebbe fungere da mediator. A questo punto, cambiamo la logica degli elementi: essi non avranno più verifiche e implementazioni al loro interno, ma semplicemente notificheranno **Dialog** delle loro modifiche di stato.

### Profile Dialog



### Login Dialog

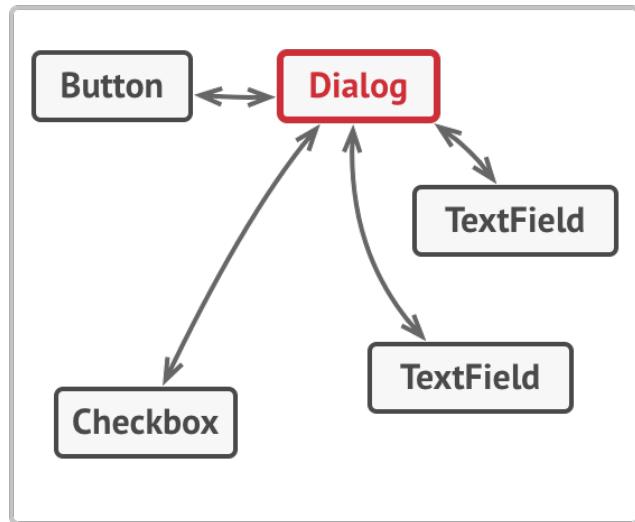


Figura 35: Struttura del dialog

**Applicabilità** Lo utilizziamo quando un set di oggetti deve comunicare in modo definito ma complesso, ed il riutilizzo degli oggetti è complesso perché sono pieni di logica. Un behavior distribuito tra diverse classi dovrebbe essere customizzabile senza troppo subclassing.

#### 16.3.6 Memento

**Scopo** Il **Memento** permette di salvare e ripristinare lo stato di oggetti senza rivelarne i dettagli implementativi.

**Problema** Immaginiamo di creare un software di text editing. Sarebbe molto comodo avere un tasto undo per annullare le operazioni. Per salvare però lo stato dell'editor, dovremmo salvare tutti i suoi fields. Il problema risulta ovvio: non possiamo accedere a field privati dall'esterno.

**Soluzione** Creiamo allora una classe **Snapshot**, che contenga i dati necessari al salvataggio, ed abbia oggetti generati dal protagonista del salvataggio, che avrà un metodo **makeSnapshot** ed un metodo **restore**. In questo modo, potremmo decidere di lasciare pubbliche alcune metadata dello snapshot, come la data o l'utente, senza pubblicarne il contenuto. Possiamo ora salvare gli snapshot all'interno di oggetti detti **caretakers**.

**Applicabilità** Utilizziamo il pattern quando vogliamo salvare lo stato di un oggetto per ripristinarlo, senza però rendere pubblici dati privati.

#### 16.3.7 Observer

**Scopo** L'**observer** è un pattern che permette di definire un meccanismo di *iscrizione* per notificare più oggetti di eventi che accadono.

**Problema** Immaginiamo due tipi di oggetto, *Customer* e *Store*. Il customer è interessato all'uscita di un nuovo modello di telefono: potrebbe visitare il negozio tutti i giorni per verificarne la disponibilità, ma questo, in termini prestazionali, fa cagare. Il negozio potrebbe, invece, inviare avvisi a tutti i clienti, anche quelli non interessati, ma questo farebbe arrabbiare i clienti.

**Soluzione** Il pattern Observer suggerisce di aggiungere un meccanismo di subscription in modo da poter iscrivere determinati clienti alle notifiche. In questo modo, il publisher avrà una lista di subscribers, aventi un metodo `update()` utile alla notifica di eventi.

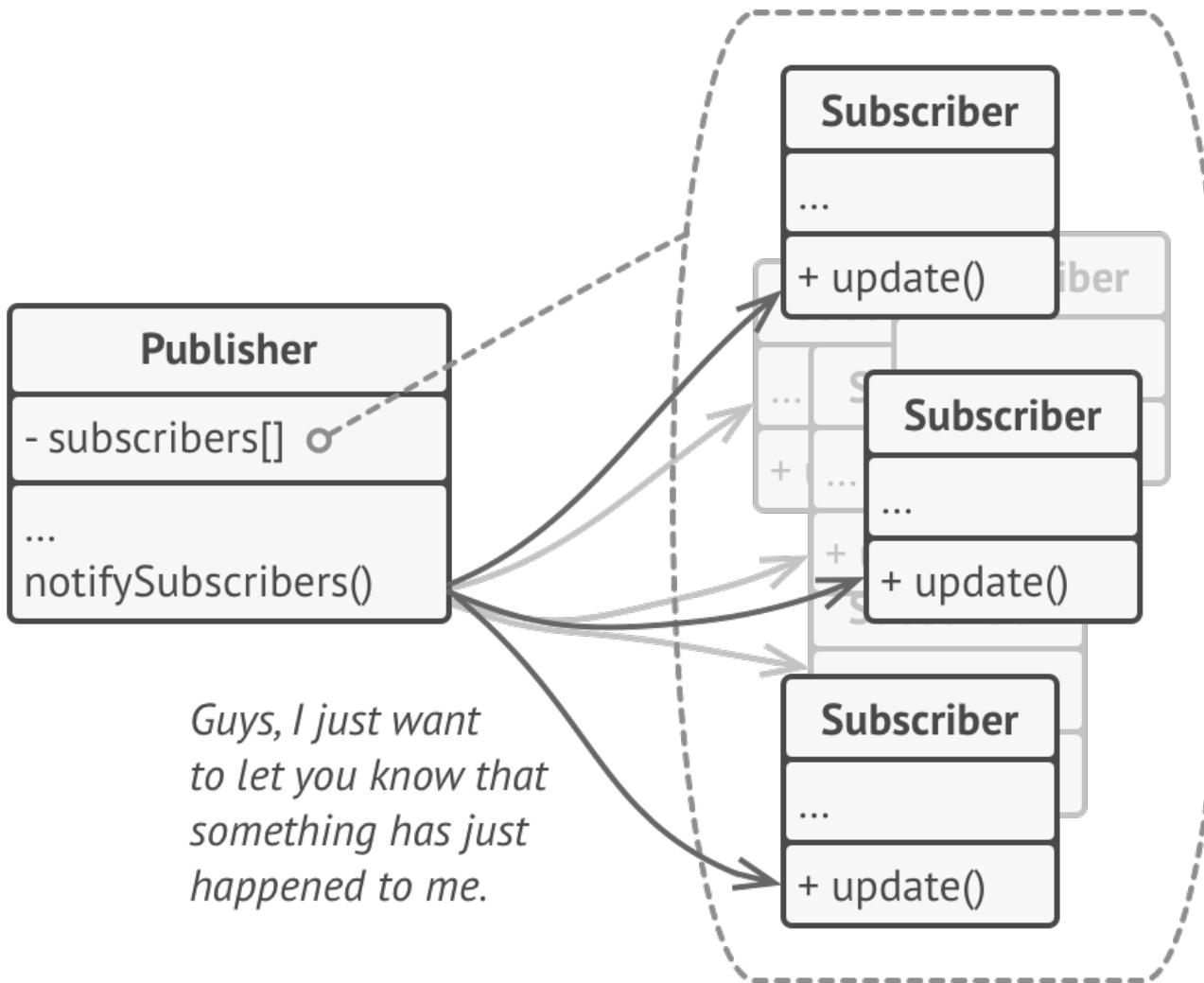


Figura 36: Struttura

**Applicabilità** Lo utilizziamo quando un'astrazione ha due aspetti, uno dipendente dall'altro, ed un cambiamento ad uno degli oggetti richiede cambiamenti all'altro.

#### 16.3.8 State

*Questo non è nelle slide.*

**Scopo** Lo **State** permette ad un oggetto di alterare il suo comportamento alla modifica del suo stato. All'apparenza, l'oggetto sembra aver cambiato classe.

**Problema** Lo state pattern è molto legato al concetto di macchina a stati finiti. L'idea di fondo è che in ogni momento esiste un numero **finito** di stati possibili per il programma. Immaginiamo di avere una classe **Document**, avente tre stati: **Draft**, **Moderation** e **Published**. Solitamente vengono utilizzati degli **switch..case** per modificare il comportamento in base alla proprietà **state**.

**Soluzione** L'alternativa è creare un interfaccia **State** implementata da tutti i possibili stati dell'oggetto, e dare all'oggetto una proprietà **state**. In questo modo, lo state potrà contenere i metodi variabili.

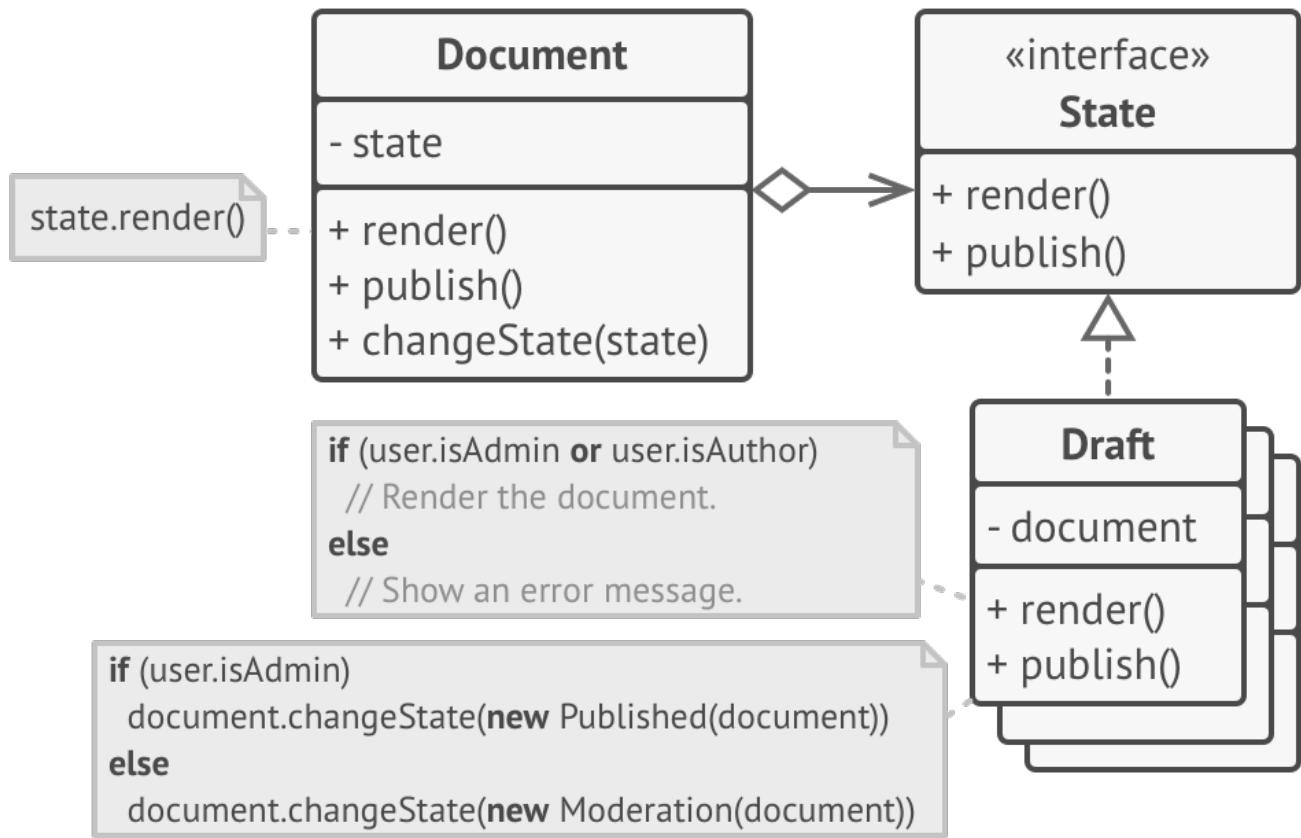


Figura 37: Struttura

### 16.3.9 Strategy

**Scopo** Lo **strategy** è un pattern che permette di definire una famiglia di algoritmi, inserirli in classi separate, e renderne gli oggetti intercambiabili.

**Problema** Stiamo creando un'app di navigazione per viaggiatori. Vogliamo implementare una funzione di calcolo itinerari, in cui l'utente possa trovare l'itinerario più veloce da un punto A ad un punto B. In una prima versione, implementiamo solo il calcolo per auto. Gli amanti della bicicletta, però, si incazzano. Cominciamo ad implementare vari metodi di calcolo di percorsi, ma a breve la nostra classe **Navigator** diventa uno schifo.

**Soluzione** Lo strategy pattern suggerisce di estrarre i vari modi di eseguire un’operazione in una classe separata. In questo modo, il **context**, ossia la classe originale, avrà un campo che salvi uno di questi oggetti. Nel nostro esempio di prima creeremmo un’interfaccia **RouteStrategy**, implementata, ad esempio, da **RoadStrategy**, **BikeStrategy** ed implementante un metodo **BuildRoute**.

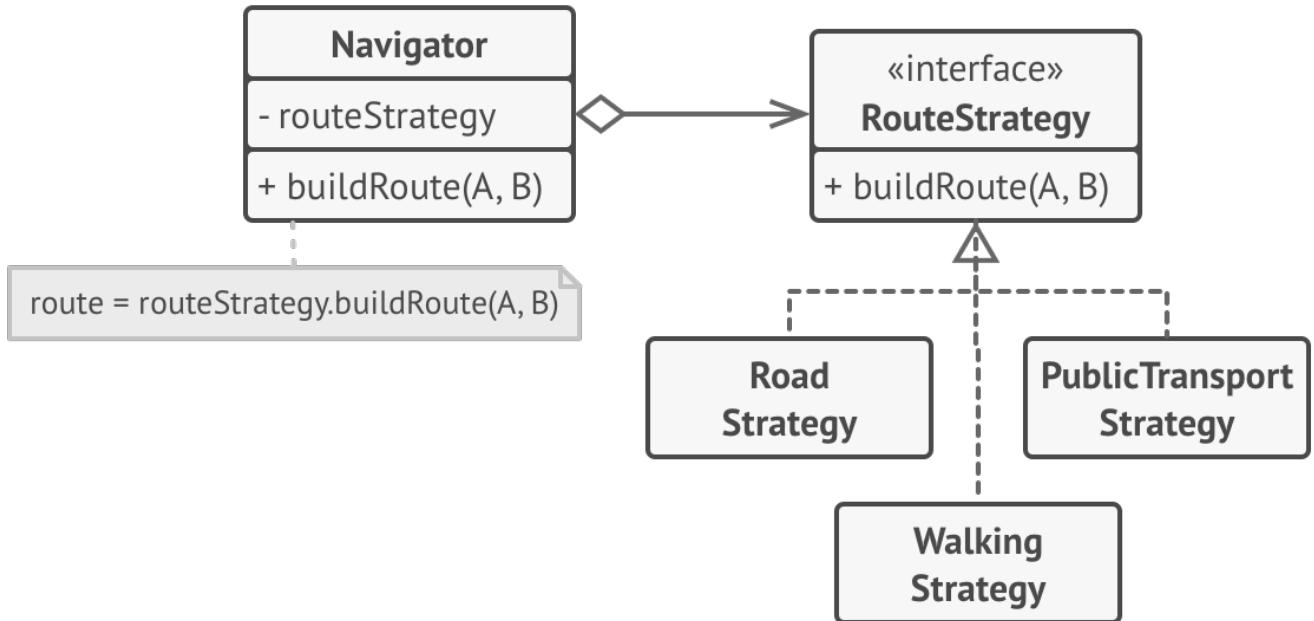


Figura 38: Struttura

**Applicabilità** Lo utilizziamo quando un oggetto può essere configurato per utilizzare algoritmi multipli encapsulabili e ricopribili da una sola interfaccia.

#### 16.3.10 Template

**Scopo** Il **template** è un pattern che permette di definire lo scheletro di un algoritmo nella sua superclasse, ma permette alle subclasses di overridare passaggi specifici dell’algoritmo senza modificarne la struttura.

**Problema** Stiamo creando un’app di data mining che analizza documenti di vario tipo: DOC, CSV, PDF. Ad un certo punto notiamo che queste classi hanno molto codice in comune.

**Soluzione** Il template pattern suggerisce di dividere un metodo in più step, in modo da poter overridare solo il necessario. Gli steps potrebbero essere implementati o abstract.

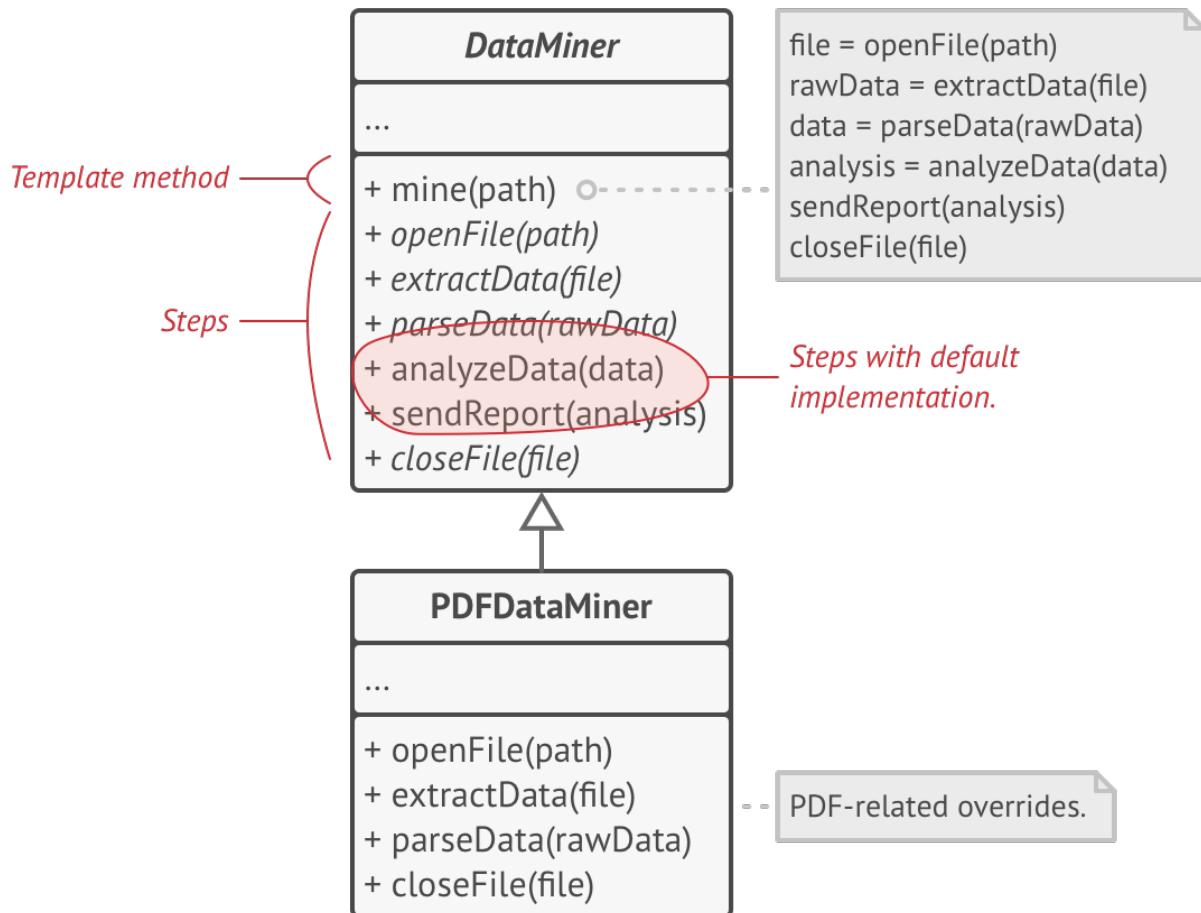


Figura 39: Struttura

**Applicabilità** Lo utilizziamo per implementare aspetti invarianti di un algoritmo quando le subclasses ne definiscono solo alcuni passaggi. In questo modo aumentiamo il riutilizzo del codice, e controlliamo le estensioni di subclass.

#### 16.3.11 Visitor

**Scopo** Il **visitor** permette di separare gli algoritmi dagli oggetti su cui operano.

**Problema** Immaginiamo di dover creare un'app che funziona con informazioni geografiche strutturate in un grande grafo. Ogni nodo può essere rappresentato da un'entità complessa, o atomica. I nodi sono connessi ad altri da strade. Vogliamo però esportare il grafo in XML, fattibile, ad esempio, aggiungendo un metodo **export** ai nodi. Il PM non vuole però farci modificare il codice dei nodi.

**Soluzione** Il pattern **visitor** suggerisce di inserire il nuovo comportamento in una classe detta *visitor*, che abbia più metodi per ogni classe dei tipi possibili. Questo però crea un problema: come fare a scegliere il metodo giusto? Implementiamo nei nodi un metodo **accept**, che prende in input il visitor e ne chiama il metodo adatto. Abbiamo sì modificato la classe nodo, ma in un modo poco distruttivo.

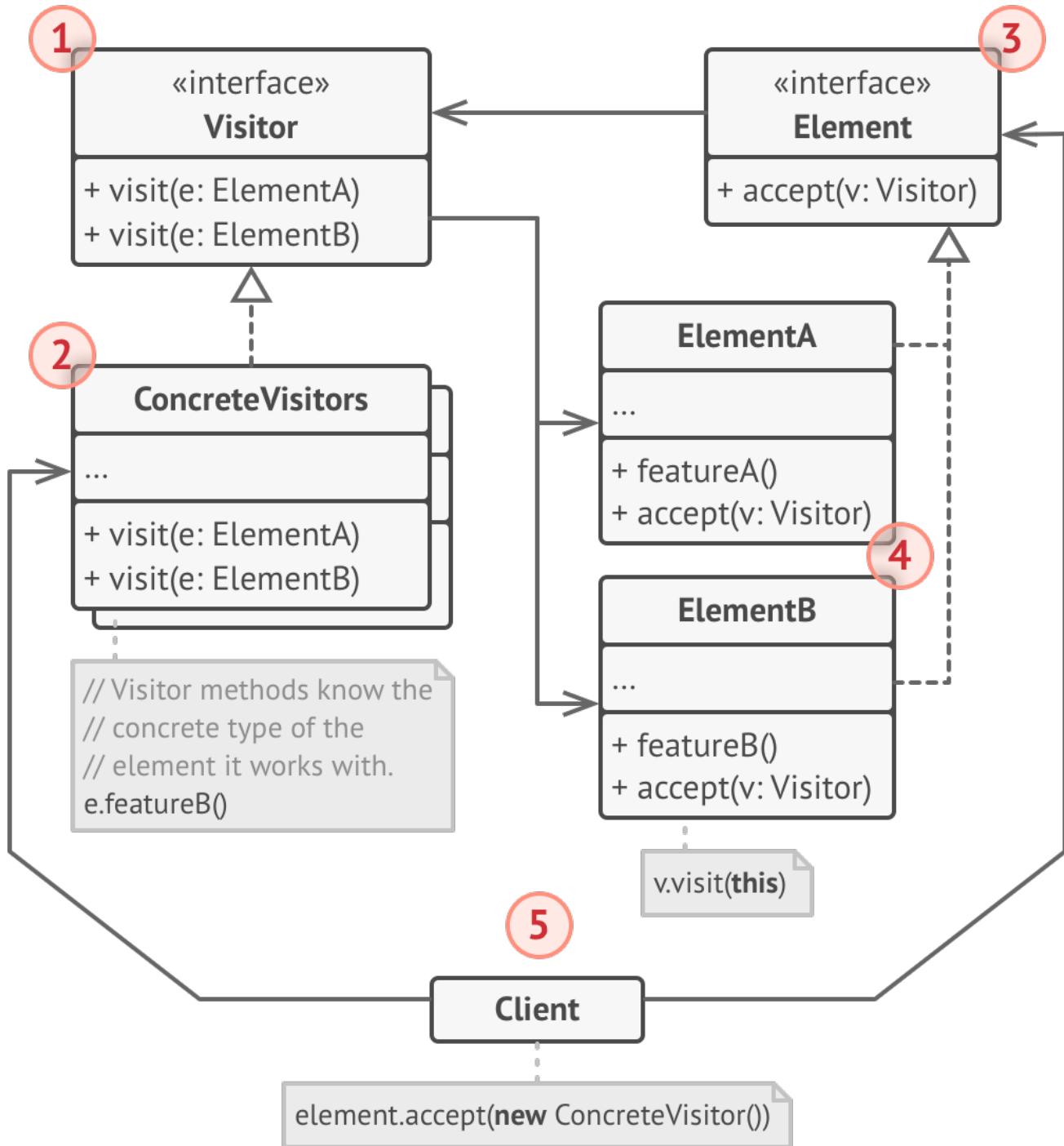


Figura 40: Struttura

**Applicabilità** Lo utilizziamo quando operazioni simili devono essere eseguite su un gruppo di oggetti di diversi tipi, ma salvati nella stessa struttura. La struttura degli oggetti è difficilmente modificata, ma è probabile avere nuove operazioni.

## 17 T17 - Architectural design

Il **design architetturale** permette di comprendere l'organizzazione ed il design della struttura generale di un sistema software. È il collegamento tra design e requirements engineering. Identifica i componenti strutturali e le loro relazioni in un processo creativo, quindi differente da sistema a sistema, ottenendo uno o più **modelli architetturali**. L'architettura è un modello di alto livello che descrive gli aspetti critici del sistema ed è comprensibile alla maggior parte degli stakeholders, permettendo la valutazione delle proprietà di sistema prima della sua esistenza. Fornisce strumenti e tecniche per costruire il sistema.

### 17.1 Software architecture

I risultati principali della **software architecture** sono gli aspetti rilevanti dell'architettura, una rappresentazione comprensibile, l'implementazione di strumenti e tecniche. Essa è una traccia che identifica componenti, interazioni, interconnessioni ed è definita in modo formale o informale. È fondamentale avere un vocabolario condiviso e ricco.

### 17.2 Elementi architetturali

Un subsystem è un sistema unico, le quali operazioni sono indipendenti dai servizi di altri subsystems. Un modulo è un componente di sistema che fornisce servizi ad altri componenti, ma non può essere considerato un subsystem.

**Componenti** Essi sono unità di computazione o data stores: clients, servers, databases, filtri, layers. Possono essere semplici o composti(subsystem). Un architettura consistente in più componenti composti è un sistema di sistemi.

**Connettori** Sono elementi architetturali che modellano le interazioni tra componenti e le regole che le governano. Le interazioni si dividono in semplici (chiamate procedurali, variabili condivise) e complesse/semantically rich (client-server, database access, async, piped data streams)

**Configurazioni** Sono grafi connessi di componenti e connettori che descrivono strutture architetturali identificanti connettività, proprietà concorrenti/distribuite, aderenza ad euristiche/stili. I componenti composti sono configurazioni.

### 17.3 Architectural design process

Tre fasi:

- **System structuring:** il sistema è decomposto in sottosistemi, e viene identificata la comunicazione tra sottosistemi
- **Control modeling:** stabilire un modello delle relazioni di controllo tra le parti
- **Modular decomposition:** i subsystem identificati sono decomposti in moduli

#### 17.3.1 Box and line diagrams

Sono diagrammi semplici ed informali che mostrano i subsystem e le loro relazioni. Mancano di semantica, non mostrando i tipi di relazioni o le proprietà. I requisiti per la semantica dei modelli dipendono da come il modello è usato.

### 17.3.2 Software architecture views

- **Logical view:** mostra le astrazioni chiave del sistema come oggetti o classi (class/state diagrams)
- **Process view:** mostra come i processi interagiscono a runtime, quindi gli aspetti dinamici e le attività, la concorrenza e sincronizzazione. Usa activity, sequence e communication diagrams.
- **Development view:** mostra come il software verrà decomposto per lo sviluppo, dalla prospettiva del programmatore. Utilizza component e package diagrams.
- **Physical view:** mostra l'hardware di sistema e come il software è distribuito tra processori/nodi. Utilizza deployment diagrams.

Gli step per questo design sono **capire i requirements, definire l'architettura, rappresentare e comunicare l'architettura, valutare l'architettura**. Si utilizzano implementazione, miglioramento e manutenzione.

### 17.3.3 Non functional requirements

- **Performance:** localizzare operazioni critiche e minimizzare la comunicazione
- **Security:** utilizzare un'architettura a strati
- **Safety:** localizzare feature con possibili falle di sicurezza, in un numero ridotto di subsystems
- **Availability:** includere componenti ridondanti e meccanismi di fault tolerance
- **Maintainability:** usare componenti *fine-grain* e sostituibili

### 17.3.4 Euristica dei subsystem

Bisogna assegnare oggetti identificati nello stesso use case, allo stesso subsystem. Creare un subsystem dedicato per oggetti atti al movimento di dati tra subsystems. Minimizzare il numero di associazioni tra subsystem boundaries. Tutti gli oggetti nello stesso subsystem devono essere funzionalmente collegati.

### 17.3.5 Layering and partitioning

Sono tecniche utili ad ottenere un basso coupling, che permettono di suddividere un sistema in subsystems. Il layering divide il sistema in strati che forniscono servizi allo strato più alto. Così, uno strato dipende solo da livelli più bassi e non conosce quelli più alti. Il partitioning divide un sistema in subsystems indipendenti.

### 17.3.6 Architecture reuse

I sistemi nello stesso dominio spesso hanno architetture simili: nascono quindi dei pattern/stili che catturano l'essenza di un'architettura.

## 17.4 Architectural patterns

Sono descrizioni di una buona pratica di design, testata in diversi ambienti, ed includono informazioni sul loro utilizzo. Sono rappresentati da tabelle e grafici.

#### 17.4.1 Model-View-Controller

Utilizziamo l'**MVC** quando abbiamo più modi di vedere ed interagire coi dati, e non abbiamo conoscenza dei futuri requirements dell'interazione coi dati. I dati e le rappresentazioni possono così cambiare indipendentemente, e abbiamo più views degli stessi dati. Aumenta però la complessità del codice. Abbiamo tre componenti:

- **Model**: mantiene i dati e li gestisce, riceve input dal controller.
- **View**: si occupa della presentazione dei dati all'utente
- **Controller**: risponde all'input dell'utente e interagisce coi modelli.

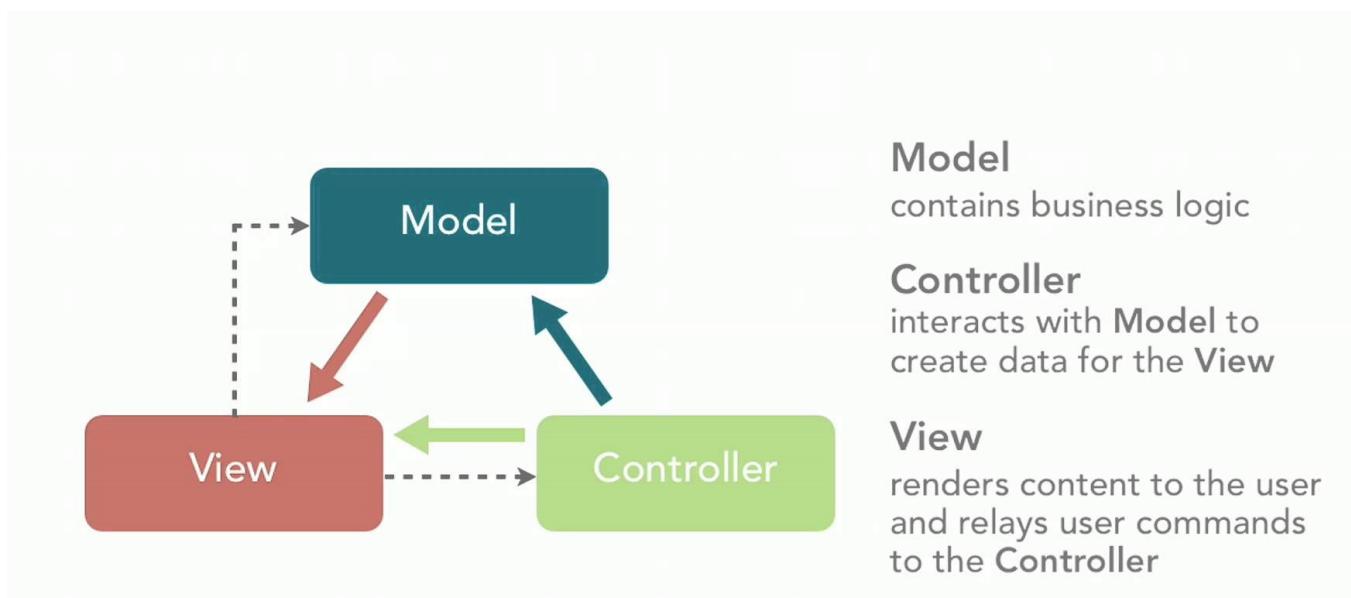
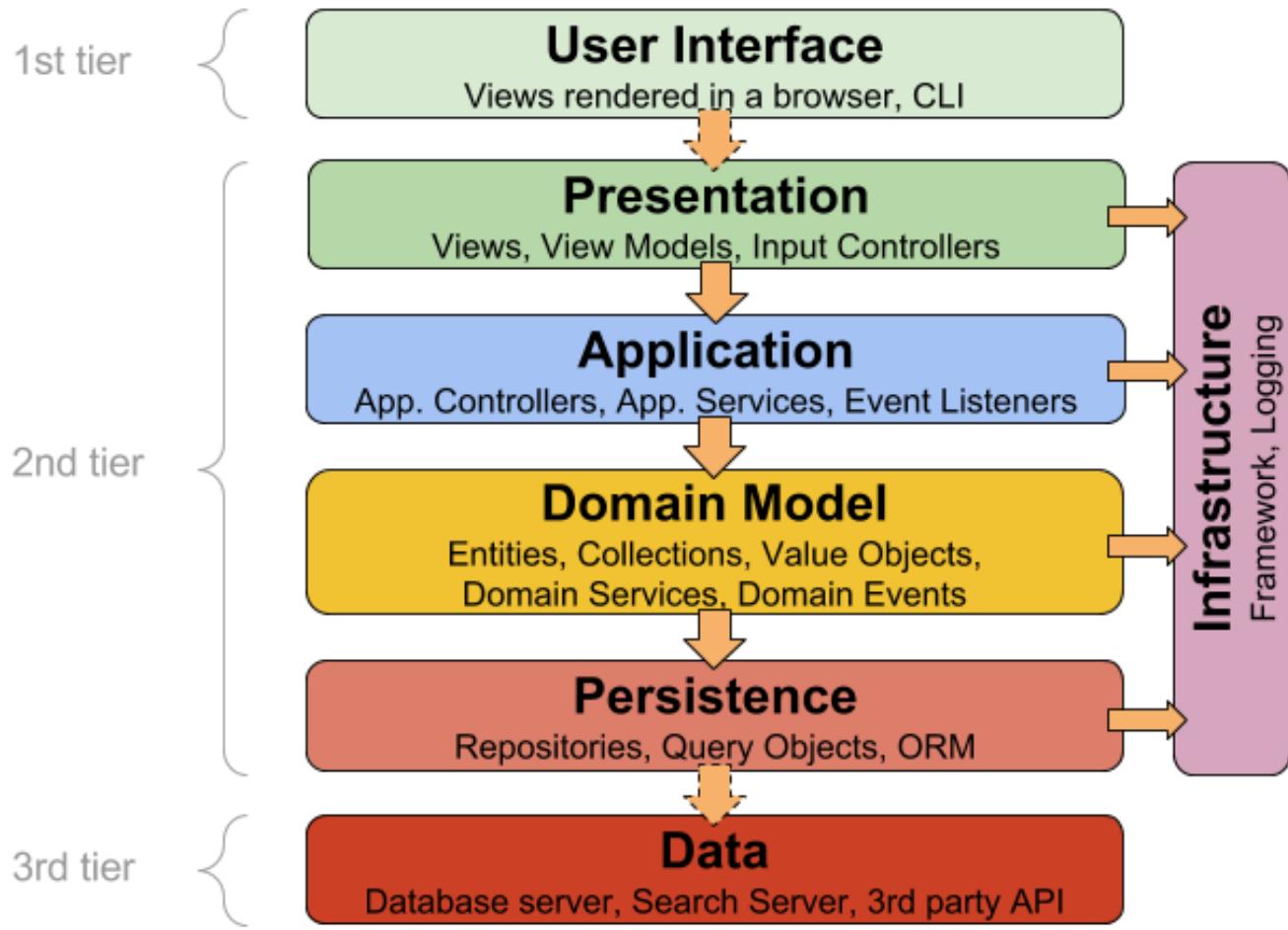


Figura 41: Struttura dell'MVC

#### 17.4.2 Layered Architecture

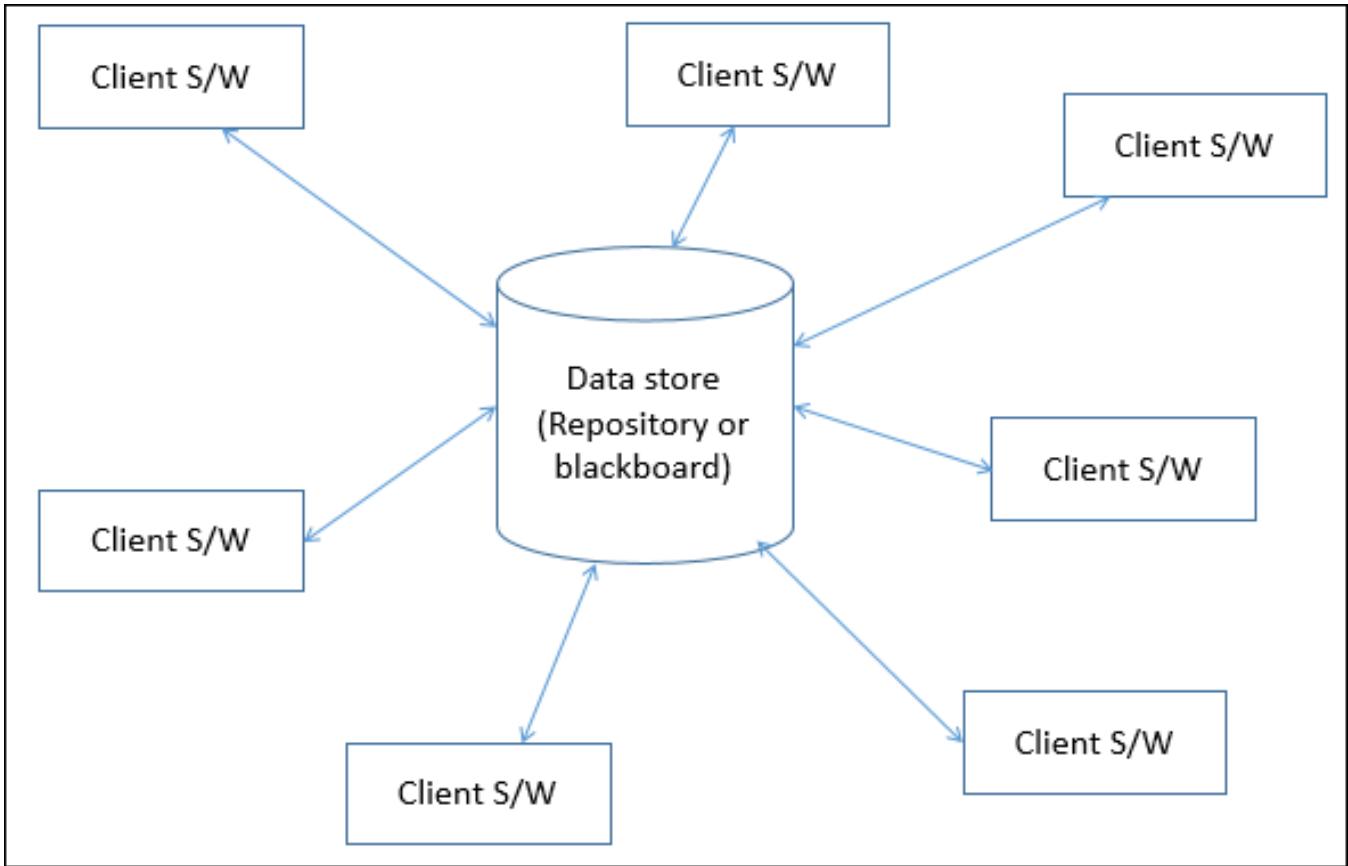
Qui, ogni layer fornisce dei servizi a quello sopra. Il più basso contiene i core services. Si utilizza quando dobbiamo creare nuove parti sopra sistemi esistenti, e lo sviluppo è basato su vari team che lavorano a funzionalità diverse. Risolve anche i requirements di sicurezza multi-level. È un ottimo sostituto dei layer di implementazione, supporta la ridondanza, ma ha problemi di performance e rende difficile separare nettamente i layers.



[www.herbertograca.com](http://www.herbertograca.com)

#### 17.4.3 Repository

La repository è accessibile a tutti i componenti, che non interagiscono tra loro ma solo con la repository, che gestisce tutti i dati. Viene utilizzato quando abbiamo grandi volumi di informazioni e l'inclusione di dati nella repo triggerà un'azione. Ha componenti indipendenti, propagazione dei cambiamenti, data management consistente, ma rischio di failure, comunicazione inefficiente e difficoltà di distribuzione della repo.



#### 17.4.4 Client-Server

Utilizziamo client-server quando i dati devono essere accessibili da più posizioni, ed il carico è variabile (distribuzione dei server). Facilita la distribuzione di server e funzionalità, ma ogni server è un punto di failure, abbiamo performance imprevedibili, e problemi di gestione quando i server hanno diversi proprietari.

**Thin e Fat client** Un thin client implementa l'interfaccia grafica, delegando al server ed alla rete le operazioni più pesanti. Un fat client, invece, esegue l'applicazione localmente, ed è quindi più complesso e potente. Gli aggiornamenti vanno installati su tutti i clients.

**Three tier architecture** Qui, abbiamo due strati di server, ad esempio un web server che si connette ad un database server.

#### 17.4.5 Pipe and filter

Il processo dei dati in un sistema è organizzato tramite una sequenza di componenti di processing, ognuno dei quali esegue una tipologia di trasformazione dei dati, i quali scorrono da un componente all'altro. Si utilizza in applicazioni di data processing nelle quali gli input sono processati in stadi separati per generare output. Un esempio sono i compilatori. Si prestano bene al riutilizzo ed ai business processes. Hanno evoluzione semplice e implementazione sequenziale/parallela. Devono però avere un formato di data transfer condiviso, e codificare/decodificare i dati.

## 17.5 Application architectures

I sistemi di applicazioni possono essere raggruppati per il tipo di business. Siccome i business hanno molto in comune, i loro sistemi applicativi tendono ad avere un'architettura comune che riflette i requirements. Un'architettura generica può essere configurata e adattata per creare un sistema che ha specifici requirements. Le application architectures sono un punto di partenza per il design, utilizzabili come checklist, come modalità organizzativa, come mezzo di riutilizzo, come vocabolario.

### 17.5.1 Centralized vs Decentralized

Il design centralizzato rende facili le modifiche nella control structure, penalizzando però le performance del singolo control object. Quello decentralizzato divide le responsabilità, funziona bene con l'Object-Oriented e permette di dividere il carico. Per scegliere, osserviamo i sequence diagrams ed i control objects, verificandone la partecipazione. Se il sequence diagram assomiglia a una forchetta, centralizzato, se assomiglia a una sedia, decentralizzato. *C'è scritto veramente?* I sequence diagrams sono derivati dagli use case.

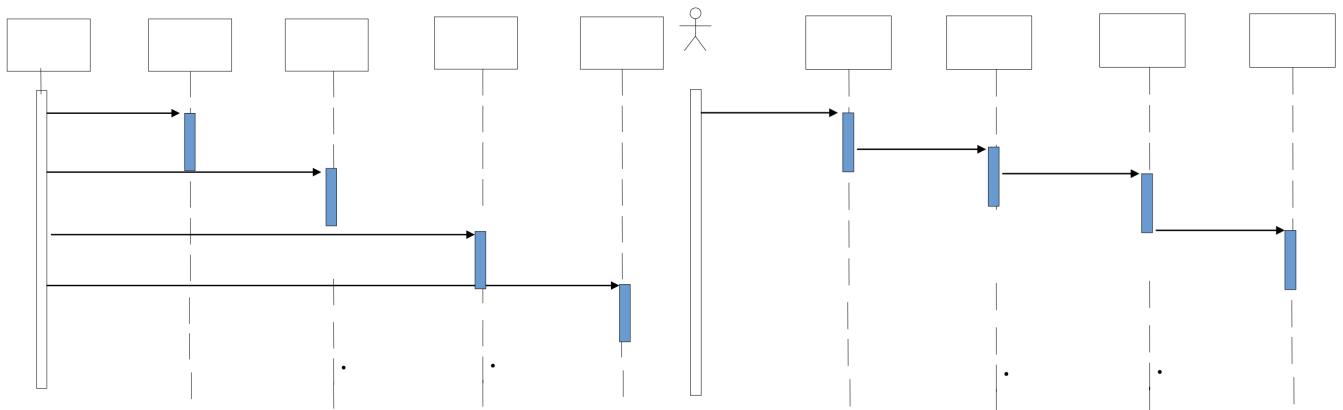


Figura 42: Fork and Stair

Nella decentralizzata, le operazioni hanno forte connessione e sono eseguite sempre nello stesso ordine. Nella centralizzata, le operazioni possono cambiare ordine o quantità.

### 17.5.2 Procedure vs Event Driven

Nel procedure driven, il controllo è nel codice del programma e l'utente ha poco controllo. Nell'event driven, il controllo risiede in un dispatcher che chiama funzioni via callbacks; l'utente ha qui molto controllo.

### 17.5.3 Vantaggi di una system architecture esplicita

- Comunicazione tra stakeholder: permette di discutere del sistema
- Analisi del sistema: permette di verificare i non functional requirements e fornire soluzioni
- Riutilizzo in larga scala: l'architettura è riutilizzabile su molti sistemi, la documentazione è di qualità, possiamo sviluppare product-line architectures
- Definisce una struttura di divisione del lavoro e permette le stime

Il design dell'architettura è uno dei primi passi della metodologia agile, perché il refactor futuro è molto costoso.

## 18 T18 - User Interface Design

Gli utenti di un sistema spesso giudicano più l'interfaccia delle funzionalità: un'interfaccia realizzata male può causare errori o far disinstallare il software. Nel design, bisogna considerare i fattori umani: memoria a breve termine limitata, errori, capacità differenti, preferenze di interazione diverse. Dobbiamo quindi matchare le skills, esperienze ed aspettative degli utenti, considerandone però anche i limiti e le possibilità di errore. È utile basarsi su un set di design principles:

- **User familiarity:** l'interfaccia dovrebbe essere basata su termini user-oriented e concetti, piuttosto che su concetti informatici.
- **Consistency:** il sistema dovrebbe avere un buon livello di consistenza, ad esempio comandi e menu nello stesso formato
- **Minimal surprise:** se un comando funziona in maniera conosciuta, l'utente dovrebbe poterne prevedere i risultati
- **Recoverability:** il sistema dovrebbe fornire resilienza agli errori umani
- **User guidance:** bisognerebbe fornire sistemi di aiuto, manuali online...
- **User diversity:** bisognerebbe permettere una personalizzazione dell'interfaccia, ad esempio scegliendo la dimensione del font
- **Real world mapping:** le informazioni dovrebbero essere presentate in un layout familiare
- **Consistency:** feature simili dovrebbero rimanere nello stesso posto e lavorare allo stesso modo
- **Less is more:** le feature meno importanti non devono essere in mezzo ai coglioni
- **Anticipation:** nascondere feature inaccessibili
- **Customization:** fornire agli utenti esperti feature avanzate
- **Transparency:** l'interfaccia non dovrebbe coprire altri contenuti
- **Contiguity:** inserire testi esplicativi vicino agli elementi grafici
- **Memory load:** ricordare all'utente i dettagli
- **User control:** identificare il responsabile delle azioni
- **Speak user's language:** istruzioni comprensibili, feedback, messaggi di errore

Abbiamo diversi issues: come passare informazioni dall'utente al computer? Come passare informazioni dal computer all'utente? Sfruttiamo metafore per il design: un set di componenti visuali, azioni e procedure che ricordano conoscenze già acquisite dall'utente in altri domini. Alcuni tipi di componenti:

- Windows: permettono di mostrare informazioni sullo schermo
- Icons: rappresentano diversi tipi di informazioni

- Menus e pulsanti: selezionano comandi
- Pointing: permette di selezionare in un menù
- Graphics elements: mixabili con testo sulla stessa finestra

Abbiamo più tipi di interazione: diretta, menu, form fill-in, comandi, linguaggio naturale.

**Direct manipulation** L'utente si sente in controllo del computer, ci mette poco ad imparare, ottiene feedback immediati. Però, la derivazione di un information space model è difficile, la navigazione può essere complessa e quindi richiedere molte risorse.

**Menu** I comandi sono presentati in una lista, lo sforzo di scrittura è minimale, gli errori improbabili. È possibile fornire aiuto contestuale. Però, le azioni complesse (and/or) sono difficili da rappresentare. I menù sono adatti a poche opzioni ed utenti non esperti, perché gli esperti preferiscono i comandi testuali.

**Form** Il form permette semplice data-entry, facilità di apprendimento, possibilità di verifica. Però richiede molto spazio a schermo e causa problemi quando l'utente non richiede esattamente ciò che è presente a schermo.

**Command language** I comandi permettono uno sviluppo rapido, di complessità arbitraria ed interfacce minimali. Però, gli utenti dovranno ricordare il linguaggio: non è adatto ad utenti saltuari. Faranno inoltre errori, e sarà richiesta la possibilità di scrittura.

**Linguaggio naturale** Esso è accessibile ad utenti inesperti ed estensibile facilmente. Però, il vocabolario è limitato e confinato a domini specifici. La tecnologia non è del tutto adatta a rendere queste interfacce accessibili ad utenti principianti (*questa slide probabilmente è stata scritta nel 2004, quando Siri e Google Assistant non esistevano. Un aggiornamento non sarebbe male eh*), ma gli utenti esperti odiano dover scrivere molto. È necessario poter scrivere.

## 18.1 Design process

Per il design è utile sviluppare un prototipo low-fidelity, che sia semplice ed economico anche nella modifica, in modo da far capire le caratteristiche principali senza concentrarsi sulle piccolezze estetiche. Bisogna spiegare le convenzioni agli utenti ed è difficile mostrare i comportamenti. Il processo di design consiste in:

1. Analisi e comprensione delle attività dell'utente
2. Produzione di un prototipo su carta e valutazione con gli utenti
3. Design del prototipo dinamico e valutazione
4. Design di un prototipo eseguibile, valutazione e implementazione finale

### 18.1.1 Information presentation

L'information presentation è costituita dallo studio di come presentare le informazioni all'utente, direttamente o trasformandole. L'MVC è una modalità che supporta più presentazioni di dati. Dobbiamo così porci più domande:

- L'utente è interessato alle informazioni o alle loro relazioni?

- Quando rapidamente variano?
- L'utente deve poter rispondere ai cambiamenti?
- È necessaria una manipolazione diretta?
- L'informazione è testuale o numerica?

Distinguiamo tra presentazione digitale ed analogica: la prima è compatta e precisa, la seconda più instantanea alla vista. È utile sfruttare i colori come una dimensione aggiuntiva, ad esempio per evidenziare eventi eccezionali. Bisogna però stare attenti a non usarli troppo, limitandone il numero. Utilizziamo il cambiamento di colore per evidenziare cambiamenti nel sistema. Sfruttiamo i colori per supportare le attività dell'utente in maniera intelligente e consistente, stando attenti anche agli accostamenti.

**Messaggi di errore** Una corretta rappresentazione degli errori è fondamentale. I messaggi devono essere educati, concisi, consistenti e costruttivi. Bisogna considerare anche altri fattori:

- Contesto: il messaggio deve riflettere il contesto attuale
- Esperienza: un utente esperto ha bisogno di errori più concisi, un principiante di errori più *verbose*
- Skill level: il messaggio deve riflettere le skill dell'utente
- Stile: i messaggi dovrebbero essere positivi
- Culture: i messaggi dovrebbero essere familiari con la cultura del paese dove viene utilizzato il software

**Sistema di aiuto** Il sistema di aiuto non dev'essere un semplice manuale online: dobbiamo sfruttare le caratteristiche dinamiche dello schermo.

## 18.2 Valutazione della user interface

Dopo il lavoro di design, bisogna valutarlo vs. le specifiche di usability, seguendo parametri come *learnability*, *speed of operation*, *robustness*, *recoverability*, *adaptability*. Utilizziamo più tecniche di valutazione:

- Expert reviews
- Questionari
- Registrazioni video dell'utilizzo
- Strumentazione di raccolta informazioni
- Feedback online dagli utenti
- Competitive usability testing

# 19 T19 - UML Diagrams for system design

## 19.1 Component Diagram

Il **component diagram** mostra gli elementi concettuali del business process (in UML1 erano componenti fisici), che forniscono o utilizzano interfacce per l'interazione con altri costrutti del sistema.

**Component** Unità logica del sistema, ha interfacce definite ed è rappresentato da un rettangolo.

**Interface** Descrive un gruppo di operazioni usate o create dai componenti. Un cerchio intero rappresenta un'interfaccia creata, mezzo ne rappresenta una required.

**Dependencies** Le dipendenze sono indicate da frecce tratteggiate.

**Ports** Le porte sono rappresentate da quadrati sul bordo. Sono usate per supportare l'esposizione di un'interfaccia required/created.

### 19.1.1 Internal View

Possiamo anche decidere di rappresentare l'interno di un componente come sottodiagramma

## 19.2 Package Diagram

Il **package diagram** rappresenta l'organizzazione degli elementi del modello in progetti di grandi dimensioni. I packages sono utilizzati per suddividere il sistema in entità logiche contenenti classi relazionate. Mostriamo i pacchetti come dei rettangoli con una sezione aggiunta in alto, e le dipendenze con frecce tratteggiate. Il nome viene inserito nella *tab* in alto. Tramite le frecce, definiamo anche *Merge*, *Import*, *Access*, *Use*.

## 19.3 Deployment Diagram

Il **deployment diagram** mostra l'architettura di esecuzione del software, quindi processori, nodi, devices, le loro connessioni e la distribuzione dei file. Utilizziamo questi diagrammi per mostrare l'hardware e software di esecuzione.

**Node** Il nodo rappresenta l'entità principale, che esegue componenti. Può essere hardware o software.

**Artifact** Gli artifacts sono elementi concreti che vengono causati dal processo di sviluppo. Possono essere librerie, configurazioni, archivi...

**Communication Path** Mostra la connessione tra due nodi.

**Manifest and deployment specification** Sono file che contengono configurazioni, ad esempio in XML.

## 20 T20 - Object Oriented Design

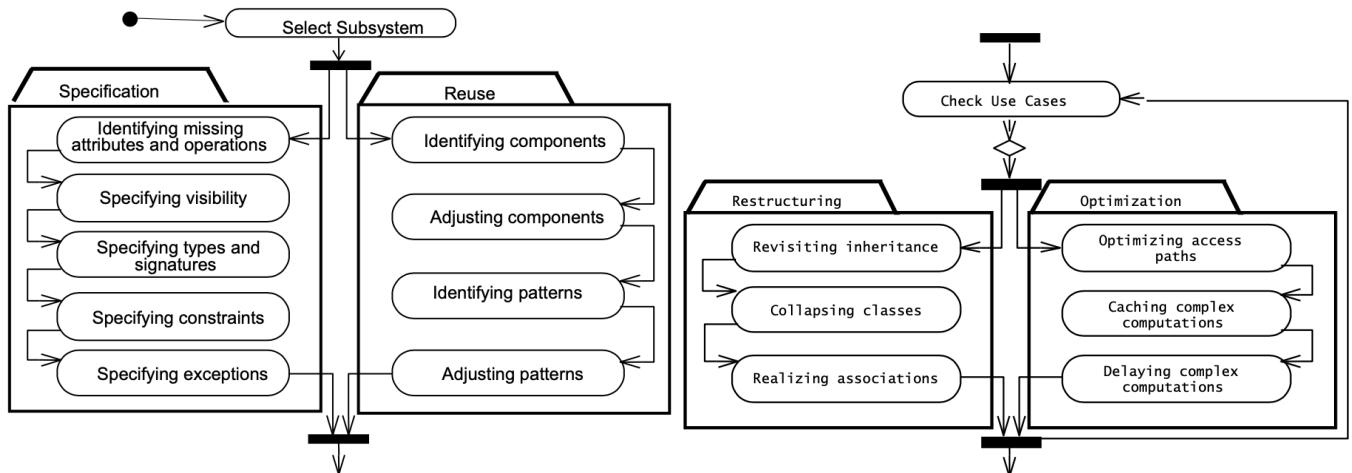
L'**Object Oriented Design** ha lo scopo di aggiungere dettagli alla requirements analysis e all'architecture model, prendendo decisioni riguardanti l'implementazione. Prepara l'implementazione attraverso decisioni sul design, ottimizza il system model, investiga alternative implementative. Gli obiettivi sono soprattutto non-functional. Esiste un gap tra il requirements gap ed il system gap, ed è proprio l'**object design gap**. Vogliamo costruire un Object Design Model che realizzi il modello Use Case, come base per l'implementazione. Vi sono però dei rischi di **incompletezza** e la presenza di attività **soggette ad errori**. Gli obiettivi sono:

- Riutilizzare le conoscenze del passato

- Riutilizzare funzionalità disponibili
- Supportare la definizione di robustezza e adattabilità
- Sviluppare nuove funzionalità
- Adattare un sistema esistente ad un nuovo ambiente/cliente

Notiamo 4 step fondamentali:

1. **Identificazione di soluzioni esistenti:** utilizzando l'ereditarietà, sfruttiamo componenti, soluzioni, design patterns.
2. **Interface specification:** descriviamo precisamente ogni class interface.
3. **Object Model Restructuring:** miglioriamo la comprensibilità ed estensibilità.
4. **Object Model Optimization:** miglioriamo le performance



## 20.1 Refactoring

Il **refactoring** è un processo che cambia la struttura interna di un software senza cambiarne il comportamento. L'obiettivo è quello di rendere il software più facile da capire, più economico da modificare, e migliorarne l'aderenza ai requirements. Si può applicare sia ai design models che al codice. Viene spesso detto **remodeling**, quando applicato ai design models. Perché lo facciamo? **Migliora il design del software**, velocizzandolo, **rende il design più comprensibile**, **rende il debugging più facile**, permette **di programmare più velocemente**. Teniamo in conto tre principi:

- Bisogna evitare di aggiungere funzionalità durante il refactoring
- Bisogna verificare l'esistenza di test ottimali prima del refactoring
- Procedere a **piccoli passi** localizzati

## 20.2 Tecniche di riutilizzo

### 20.2.1 Ereditarietà

Distinguiamo due utilizzi:

1. Descrizione delle tassonomie: utilizzata durante la requirements analysis, identifica oggetti del dominio che hanno una relazione, allo scopo di rendere l'analysis model più comprensibile.
2. Specifica delle interfacce: usata durante l'object design, identifica le signatures degli oggetti identificati, allo scopo di aumentare il riutilizzo, la modificabilità, l'estensione.

Distinguiamo, inoltre, tra **inheritance** (white-box reuse), e **composition** (black-box reuse). Nella prima i prodotti dello sviluppo (modelli, system/object design, codice) devono essere disponibili, e sfruttiamo implementation e specification. La seconda necessita solo di alcuni prodotti, al minimo gli eseguibili, e crea gli oggetti come aggregazione di quelli esistenti. Abbiamo più ragioni per le quali può essere necessaria la creazione di oggetti:

- I boundary objects necessitano di componenti della GUI
- Gli entity objects necessitano di componenti di data management specifici
- L'implementazione di algoritmi potrebbe necessitare di oggetti per salvare valori
- Le operazioni di alto livello potrebbero essere decomposte in operazioni di basso livello
- Gli use case actors dovrebbero essere definiti da interfacce
- Alcune classi di data collection potrebbero essere definite per salvare dati sugli attori
- Almeno una classe di avvio deve esistere per far partire il software
- Potrebbero essere necessarie classi controller e coordinator per regolare l'esecuzione

Vi è anche la possibilità di unire oggetti, per scelte di design: implementare un'entità come attributo, o come classe separata con associazioni ad altre classi. Le associazioni sono più flessibili degli attributi, ma spesso introducono indirection (mancanza di chiarezza) non necessaria. Troviamo ereditarietà per generalizzazione (ossia partendo dalle subclasses) o per specializzazione (partendo dalle superclasses). Per aumentare l'ereditarietà, riarrangiamo le classi per prepararle, cerchiamo di trasformare le superclasses in interfacce astratte (bridge pattern), verifichiamo se è possibile mappare le classi di un subsystem in una gerarchia di ereditarietà. Possiamo trovare dei problemi durante questo processo: non dobbiamo infatti esagerare implementando interfacce con metodi inutilizzati.

### 20.2.2 Delegation

La **delegation** è un modo di rendere la composizione potente quanto l'ereditarietà. Nella delegation, due oggetti sono coinvolti nella gestione di una richiesta del client: il **receiver** delega le operazioni al **delegate**, assicurandosi che il client non utilizzi il delegate in modo errato. I due metodi diversi hanno pro e contro: la delegation è più flessibile ma inefficiente, l'inheritance è facile da usare, supportata da più linguaggi, ma espone una subclass ai dettagli della classe parent, ed ogni modifica nel parent costringe le subclasses a cambiare.

**Contraction** Il goal della contraction è rendere le operazioni della superclass invisibili, implementando metodi nella superclasse ed overridandoli con metodi vuoti nella subclass. Questo processo andrebbe evitato, perché la superclass contiene operazioni che non hanno senso nella subclass, non *fitta* nella tassonomia della superclasse, e viola il principio di sostituzione di Liskov.

**Raffinare le associazioni con l'aggregazione** Aggiungiamo molteplicità e ruoli, decidendo *whole* e *part*, poi guardiamo alla molteplicità del whole: se è 1, usiamo la composizione, altrimenti aggregazione. Aggiungiamo infine la navigabilità dal whole alla part.

**Raffinare le associazioni con la reification** La **reification** riguarda il trattamento di una cosa astratta come se fosse concreta; nei termini object-oriented, intendiamo la caratterizzazione di qualcosa nei termini di **oggetto**. Nel contesto del design, il concetto è applicato alle relazioni tra oggetti o stati di un oggetto.

### 20.3 Packaging design

Il **packaging design** impacchetta il design in unità discrete che possono essere modificate, compilate, collegate, riutilizzate. Costruiscono moduli fisici, idealmente un pacchetto per subsystem. La system decomposition potrebbe non essere buona per l'implementazione. Due principi base: minimizzare il coupling, massimizzare la cohesion. Come fare?

1. Partire con un'interfaccia per ogni subsystem
2. Limitare il numero di operazioni dell'interfaccia ( $7 \mp 2$ )
3. Se l'interfaccia ha troppe operazioni, riconsideriamo il numero di interfacce
4. Se abbiamo troppe poche interfacce, riconsideriamo il numero di subsystems

Per definire le classi, aggiungiamo prima gli attributi, poi i metodi, e li inseriamo nell'object model. Infine aggiungiamo gli invariants. Per i metodi, invece, decidiamo le pre-conditions, le post-conditions, e lavoriamo su pseudocode, flowcharts, UML per specificare l'algoritmo.

### 20.4 Information hiding

Per ottenere un buon **information hiding**, definiamo interfacce pubbliche, applicando il "need to know" principle: meno dettagli una classe deve sapere, più facile è cambiarla/non rovinarla con cambiamenti. Il tradeoff qui è information hiding vs efficiency. Elenchiamo alcuni principi:

- Solo le operazioni di una classe ne possono manipolare gli attributi
- Nascondere gli oggetti esterni al subsystem boundary: definire le interfacce astratte che mediano tra il mondo esterno ed il sistema
- Non applicare ad un'operazione i risultati di un'altra operazione

## 21 T22 - Testing

Distinguiamo in 4 componenti:

- Test plan

- Test specification
- Test oracle
- Test cases

## 21.1 Componenti del testing

### 21.1.1 Test Plan

Il **test plan** è utilizzato per dimostrare che il software è privo di fallo e si comporta come richiesto dai requirements. Decomponi il processo in test specifici, utilizzando specifici data items e valori.

### 21.1.2 Test Specification

Documenta lo scopo di un test; in caso di test compositi, documenta la relazione tra le parti ed il whole test. Descrive le condizioni che indicano quando il test è completo. In generale, è un modo per valutare i risultati.

### 21.1.3 Test Oracle

È un set di risultati predetti per un set di test, e si usa per determinare il successo del testing. È estremamente difficile da creare, attraverso la requirements specification.

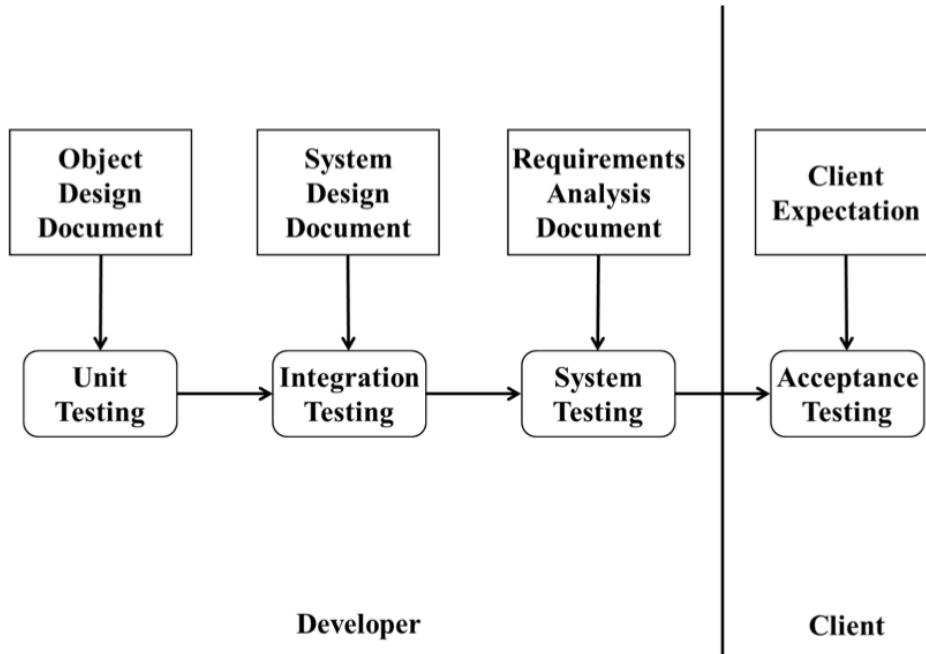
### 21.1.4 Test Cases

È un set di input al sistema. Un testing di successo è basato sulla scelta dei giusti test cases.

## 21.2 Test suite

Ovviamente bisogna testare il codice per farlo funzionare la prima volta, facendo *ad hoc testing* o costruendo una test suite. La seconda riduce il numero totale di bug, rende il codice più mantenibile e refactorabile, ma richiede più programmazione e tempo. Distinguiamo tra due tecniche di testing: **glass box testing**, in cui esaminiamo il codice, e **black box testing**, basato sulla conoscenza del risultato atteso.

## 21.3 Tipologie di testing



### 21.3.1 Unit testing

Lo **unit testing** testa le singole unità che compongono il sistema, allo scopo di trovare falle in algoritmi, dati, sintassi. Definiamo un set di test cases e ne valutiamo i risultati. Questo test è fatto dagli sviluppatori per assicurarsi che l'unità sia codificata correttamente e funzioni come ci si aspetta. Distinguiamo, anzitutto, tra **static testing** (analisi, code review) e **dynamic testing** (black-box, white-box).

**Black-box testing** Questo tipo di testing si concentra sul comportamento I/O, e richiede un test oracle. L'obiettivo è ridurre il numero di test cases tramite equivalence partitioning. Dividiamo input validi ed invalidi in classi di equivalenza, scegliamo i test cases per ogni classe di equivalenza. Per scegliere i test case, notiamo che l'input è valido in un range di valori, e scegliamo i test case da quelli sotto il range, nel range, oltre il range. L'input è valido solo se facente parte di un set discreto: prendiamo un test case valido, uno invalido.

**Glass-box testing** Non è possibile testare tutti i percorsi, cerchiamo di assicurarci che ogni linea di codice sia testata. Il glass-box testing ha dei limiti: un pezzo di codice definisce più execution paths, ed un buon test case deve provarli tutti, arrivando a quantità di test case enormi.

**Euristiche** Per ottenere gli unit tests:

1. Definiamo gli unit tests
2. Creiamo i test cases
3. Eliminiamo duplicati tra i test cases
4. Controlliamo staticamente il codice

5. Creiamo un test harness, i.e. una collezione di test cases
6. Descriviamo il test oracle
7. Eseguiamo i test cases
8. Compariamo i risultati con il test oracle

Per la selezione dei test cases, utilizziamo la conoscenza dei functional requirements, il design knowledge del sistema, delle data structures, degli algoritmi. Utilizziamo anche la conoscenza dell'implementazione di algoritmi e strutture dati.

### 21.3.2 Integration testing

L'integration testing testa un gruppo di subsystems, ed eventualmente l'intero sistema. È fatto dagli sviluppatori per testare le interfacce tra subsystems. L'intero sistema è visto come una collezione di subsystems, determinati durante il system e object design. L'obiettivo è testare tutte le interfacce tra subsystems e la loro interazione, strategizzando l'ordine con cui i subsystems sono testati. Perché lo facciamo? Gli unit tests funzionano solo in isolazione, ma spesso le falte sono nell'interazione. Spesso molti componenti riutilizzati non possono essere unit tested. Le falte non trovate durante l'integration testing verranno scovate in produzione, il che non va mai bene. Definiamo alcune componenti:

**Driver** È un componente che chiama l'unità testata e controlla i test cases

**Stub/Double** è un componente che simula la presenza di un altro, rispondendo alle chiamate con dati falsi. Non deve comportarsi esattamente come chi sostituisce, ma deve fornire circa la stessa API. Ne distinguiamo 4 tipi:

- **Dummy:** è utilizzato come segnaposto, passato come parametro ma mai utilizzato
- **Stub:** è un oggetto che forza il sistema verso il path che vogliamo testare
- **Mock:** ritorna valori hardcoded o precaricati
- **Fake:** rimpiazza l'originale con un'implementazione alternativa

**Integration testing strategies** Distinguiamo inoltre varie strategie di integration testing:

- **Big Bang Integration:** porta difficoltà nella fault isolation, ma non necessita di drivers o stubs, è una pratica comune dell'agile
- **Bottom-Up Integration:** testa i sistemi più importanti per ultimi, necessita di driver ma non stubs, è utile per i sistemi object-oriented e real time
- **Top-Down Integration:** i test cases possono essere definiti sulla base di funzionalità di sistema, non c'è bisogno di drivers ma gli stub sono difficili e in grande quantità. In secundis, le utilità di basso livello sono testate tardi.
- **Sandwich Testing:** top e bottom sono fatti insieme, senza testare i subsystem singoli e le loro interfacce prima dell'integrazione. È meno sistematico degli altri due e rende più complicata la failure isolation
- **Modified Sandwich:** Testa in parallelo il middle layer con driver e stubs, il top layer con stubs, il bottom layer con drivers. Testa in parallelo il top layer che accede al middle, ed il middle che accede al bottom.

### 21.3.3 System Testing

Il system testing testa l'intero sistema, ed è fatto dagli sviluppatori allo scopo di determinare se il sistema rispetta i requirements funzionali e di performance. I test cases sono creati partendo dal requirements analysis document o dallo user manual. I test cases sono centrati attorno ai requirements e alle funzioni chiave, ed il sistema è trattato come una black box. Gli unit test cases sono riutilizzati, ed i nuovi test cases vanno sviluppati.

**Performance testing** Il goal è tentare di violare i non-functional requirements, testando come il sistema si comporta quando è sovraccaricato, o in ordini non usuali di esecuzione, o con grandi volumi di dati. Citiamo, ad esempio: stress testing, security testing, volume testing, configuration testing, quality testing, recovery testing...

**Acceptance Testing** Il goal è dimostrare che il sistema è pronto per l'utilizzo, con test e svolti scelti dal cliente. Distinguiamo tra **alpha test**, in cui si testa nell'ambiente di sviluppo, e **beta test**, in cui si testa nell'ambiente del cliente.

## 21.4 Object-Oriented testing

I componenti testati sono classi, istanziate come oggetti. Meno granulare di test su funzioni individuali, quindi gli approcci al white-box testing vanno estesi. Non c'è inoltre un "top" ovvio per il top-down. Distinguiamo 4 livelli: test delle operazioni associate alle classi, test delle classi, test di cluster di oggetti cooperanti, test del sistema completo.

**Object Class Testing** Un test completo di una classe testa tutte le operazioni associate, gli attributi, tutti gli stati. L'ereditarietà rende questo test più complesso.

**Object Integration** I livelli di integrazione sono meno distinti nei sistemi object-oriented. Il cluster testing si occupa di integrare e testare cluster di oggetti, identificati tramite la conoscenza delle operazioni e le feature implementate nel cluster. Distinguiamo tre approcci: **use-case or scenario testing**, basato sulle interazioni utente, **thread testing**, basato sulle risposte del sistema, **object interaction testing**, testante le interazioni che si fermano quando un'istanza non chiama servizi di un altro oggetto.

### Ereditarietà, polimorfismo, dynamic binding

- Ereditarietà: i metodi ereditati devono essere ritestati nelle subclasses: il contesto delle superclasses potrebbe essere incompleto
- Polimorfismo: i parametri devono avere più di un set di valori e un'operazione deve essere implementata da più di un metodo
- Dynamic binding: i metodi che implementano un'operazione sono sconosciuti fino al runtime

**Continuous testing** Il continuous testing consiste in build e relativo testing ogni giorno, in modo che il sistema sia sempre eseguibile. Richiede dei tool di supporto, come un continuous build server, test automatizzati, tool supported refactoring, issue tracking...

## Glossario

**CASE tool** Il CASE tool è un software che supporta la progettazione di sistemi software, ad esempio con UML. 20

**constraint** un constraint è un limite che viene posto; restriction, limitation.. 9, 10, 22

**domain** Si intendono, qui, le conoscenze relative ad un determinato ambito. Ad esempio, sviluppando un sistema per Trenitalia(*qualcuno lo fa veramente? Non sono scritti da scimmiette?*), il dominio sarebbe quello della gestione dei treni. Il problema sta nel fatto che il programmatore non ha conoscenze al riguardo dei treni, e deve utilizzare quindi un linguaggio, implicazioni e conoscenze che non gli appartengono. 9

**volatility** Con RV intendiamo modifiche ai requirements che avvengono durante lo sviluppo del progetto. (*Il classico "Ah ma già che ci sei, mi aggiungi un ecommerce, una macchina del caffè, e la possibilità di accarezzare i clienti attraverso il computer? Maledetti*). 20