

# Orale APS

Giuseppe Facchi

19 maggio 2021

## Indice

<b>1</b>	<b>Processi per lo sviluppo software</b>	<b>3</b>
1.1	Introduzione . . . . .	3
1.2	Processo a cascata . . . . .	3
1.3	Sviluppo iterativo ed evolutivo . . . . .	3
1.3.1	Pianificazione iterativa, guidata dal rischio e dal cliente	4
1.4	UP . . . . .	4
1.4.1	Le fasi di UP . . . . .	4
1.4.2	Le discipline di UP . . . . .	4
1.5	Agile . . . . .	5
1.6	Fase di Ideazione . . . . .	5
<b>2</b>	<b>Requisiti evolutivi</b>	<b>6</b>
2.1	Modello dei Casi d'uso . . . . .	6
2.1.1	Attori . . . . .	6
2.1.2	Notazione: tre formati per i casi d'uso . . . . .	7
2.1.3	Come trovare i casi d'uso . . . . .	7
2.1.4	Verificare l'utilità dei casi d'uso . . . . .	7
<b>3</b>	<b>Modellazione di dominio</b>	<b>9</b>
3.1	Diagramma delle classi (modello di dominio) . . . . .	9
3.2	Diagramma degli oggetti . . . . .	9
<b>4</b>	<b>Diagrammi di interazione (parte di modello dei casi d'uso)</b>	<b>10</b>
4.1	Operazioni di sistema e diagrammi di sequenza di sistema . . .	10
4.1.1	Contratti . . . . .	10

4.2	Diagrammi di comunicazione . . . . .	10
4.3	Pro/Contro diagrammi Sequenza/Comunicazione . . . . .	11
<b>5</b>	<b>Diagramma delle classi di progetto</b>	<b>11</b>
5.1	Generalizzazione . . . . .	11
5.2	Aggregazione e composizione . . . . .	11
<b>6</b>	<b>Diagrammi di attività</b>	<b>12</b>
<b>7</b>	<b>Diagrammi di stato</b>	<b>13</b>
<b>8</b>	<b>RDD</b>	<b>14</b>
<b>9</b>	<b>GRASP: Progettazione di oggetti con responsabilità</b>	<b>14</b>
<b>10</b>	<b>Design Patterns</b>	<b>17</b>
10.1	Alcuni Design patterns . . . . .	18
10.1.1	Adapter . . . . .	18
10.1.2	Factory . . . . .	18
10.1.3	Singleton . . . . .	18
10.1.4	Strategy . . . . .	18
10.1.5	Composite . . . . .	19
10.1.6	Façade . . . . .	19
10.1.7	Observer . . . . .	19

# 1 Processi per lo sviluppo software

## 1.1 Introduzione

Un processo per lo sviluppo software definisce un approccio per la costruzione, il rilascio e la manutenzione del software. Esempi:

- **Processo a cascata**
- **UP**
- **Scrum**
- **Spirale**

## 1.2 Processo a cascata

Il processo software con ciclo di vita **a cascata** è basato su uno svolgimento **sequenziale** delle diverse attività.

Questo processo è **molto soggetto a fallimenti**, perché più si fa grande il software più è difficile implementare nuove features.

## 1.3 Sviluppo iterativo ed evolutivo

In questo approccio lo sviluppo è organizzato in una serie di mini-progetti brevi, di lunghezza fissa chiamati **iterazioni**. Il risultato di ciascuna iterazione è un **sistema eseguibile, testato e integrato**.

*Può essere chiamato anche sviluppo iterativo e incrementale o sviluppo iterativo ed evolutivo.*

### Vantaggi

- **Minore probabilità di fallimento**
- Riduzione **precoce** dei rischi maggiori
- **Progresso visibile** fin dall'inizio
- **Feedback precoce**, con coinvolgimento dell'utente

**Timeboxing:** Le iterazioni hanno una **lunghezza fissata**

### 1.3.1 Pianificazione iterativa, guidata dal rischio e dal cliente

In ciascuna **iterazione** viene stabilito il piano di lavoro dettagliato per una sola iterazione. In *UP* viene effettuata alla **fine** di ciascuna iterazione, per decidere il piano dell'iterazione successiva.

## 1.4 UP

**Unified Process** è un processo **iterativo** per la costruzione di sistemi orientati agli oggetti. In particolare viene ampiamente adottato **RUP (Rational Unified Process)**, un suo raffinamento.

- Processo **iterativo**
- Le pratiche di UP forniscono un esempio di struttura rispetto a come **eseguire** e dunque come spiegare l'**OOA/D** (*Object-Oriented Analysis/Design*)
- UP è **flessibile** e può essere applicato usando un **approccio leggero e agile** come ad esempio **Scrum**

### 1.4.1 Le fasi di UP

- **Ideazione:** *NON è la previsione dei requisiti del modello a cascata, ma una fase di fattibilità*
- **Elaborazione**
- **Costruzione**
- **Transizione**

### 1.4.2 Le discipline di UP

Una **disciplina** è un insieme di attività e dei relativi elaborati di una determinata area. In UP un elaborato è un qualsiasi prodotto di lavoro. Ci sono diverse discipline in UP:

- **Modellazione del business:** Modello di dominio
- **Requisiti:** Modello dei Casi d'uso per definire requisiti funzionali e non funzionali

- **Progettazione:** Modello di Progetto

## 1.5 Agile

*Non è possibile dare una definizione precisa di metodo agile perché le pratiche adottate variano notevolmente da metodo a metodo.*

Una pratica di base è quella che prevede **iterazioni brevi**, con raffinamenti evolutivi dei piani, dei requisiti e del progetto.

Lo **scopo della modellazione e dei modelli** è di **agevolare la comprensione e la comunicazione**, **NON** di documentare.

## 1.6 Fase di Ideazione

Lo scopo della fase di ideazione è stabilire una visione iniziale comune per gli obiettivi del progetto.

- Non si definiscono tutti i requisiti nella fase di ideazione
- La maggior parte dell'analisi dei requisiti avviene durante la fase di elaborazione
- Modello dei Casi d'Uso

## 2 Requisiti evolutivi

Un sistema deve fornire un certo numero di funzionalità, relative alla gestione di alcune tipologie di informazione e al possesso di determinate qualità (sicurezza e prestazioni). Un requisito è una capacità o una condizione a cui il sistema deve essere conforme.

**Requisiti funzionali** Descrivono il comportamento del sistema in termini di funzionalità fornite ai suoi utenti. Possono essere espressi in forma di casi d'uso.

**Requisiti non funzionali** Non riguardano le specifiche funzioni del sistema, ma sono relativi a proprietà del sistema, ad esempio sicurezza, prestazioni, scalabilità, ecc.

### 2.1 Modello dei Casi d'uso

In generale i casi d'uso sono storie scritte, testuali, di qualche attore che usa un sistema per raggiungere degli obiettivi. I casi d'uso non sono diagrammi, bensì testo.

**Attore** Qualcosa o qualcuno dotato di comportamento, come una persona o un'organizzazione o un sistema informatico.

**Scenario (istanza di caso d'uso)** E' una sequenza specifica di azioni e interazioni tra il sistema e alcuni attori.

Un caso d'uso è quindi una collezione di scenari correlati, sia di successo che di fallimento. I casi d'uso sono Requisiti Funzionali.

#### 2.1.1 Attori

- **Attore Primario:** Utilizza direttamente i servizi del SuD (system under discussion) affinché vengano raggiunti gli obiettivi utente.
- **Attore Finale:** Vuole che il SuD sia utilizzato affinché vengano raggiunti dei suoi obiettivi. Spesso attore primario e finale coincidono (es. Cliente commercio elettronico).

- **Attore di Supporto:** Offre un servizio al SuD (es. Sistema di autorizzazione al pagamento).
- **Attore fuori scena:** Non è un attore primario, finale, di supporto (es. Governo).

### 2.1.2 Notazione: tre formati per i casi d'uso

- Formato breve
- Formato informale
- Formato dettagliato

### 2.1.3 Come trovare i casi d'uso

1. Scegliere i confini del sistema
2. Identificare gli attori primari
3. Identificare gli obiettivi di ciascun attore primario
4. Definire i casi d'uso che soddisfino gli obiettivi utente

### 2.1.4 Verificare l'utilità dei casi d'uso

**Test del capo** Il capo pone una domanda per cui ci sarà una risposta, se la risposta non soddisfa il capo il caso d'uso non è mirato a ottenere i risultati il cui valore sia misurabile. Non è però sempre vero (es. Autenticazione utente, concetto semplice ma difficilmente implementabile)

**Test EBP (Elementary Business Process)** Simile al test del capo. Un processo di business elementare è un'attività svolta in risposta a un evento di business, che aggiunge valore e lascia i dati in uno stato consistente.

**Test della dimensione** Un caso d'uso deve essere costituito da più passi e corrisponde a 3 o più pagine nel formato dettagliato.

## Esempi

- *Negoziare un contratto con un fornitore*: Troppo ampio per essere un EBP
- *Gestire una restituzione*: Passa il test del capo, è un EBP, le dimensioni vanno bene
- *Effettuare il login*: Non passa il test del capo
- *Spostare una pedina*: Non passa il test della dimensione



## 3 Modellazione di dominio

Rappresentazione visuale di classi concettuali o di oggetti del mondo reale e delle relazioni tra di essi.

### 3.1 Diagramma delle classi (modello di dominio)

Applicando la notazione UML un modello di dominio può essere rappresentato da uno o più diagramma delle classi. Prevede:

- **Classi concettuali:** rappresentano cose o concetti del dominio di interesse
- **Associazioni tra classi:** rappresentano relazioni tra oggetti di due classi
- **Attributi di classi concettuali:** rappresentano proprietà elementari degli oggetti di una classe

#### Aggregazione e composizione

- **Aggregazione:** Tipo di associazione intero-parte (es. macchina-ruote)
- **Composizione:** Tipo di forte associazione intero-parte
  - Ciascuna istanza della parte appartiene a una sola istanza dell'intero alla volta
  - Ciascuna parte deve sempre appartenere ad un intero
  - La vita delle parti è limitata dall'intero: le parti possono essere create dopo l'intero, ma non prima e possono essere distrutte prima dell'intero, ma non dopo

### 3.2 Diagramma degli oggetti

Mostra un insieme di oggetti con i loro attributi e le loro relazioni in un dato momento.

## 4 Diagrammi di interazione (parte di modello dei casi d'uso)

- **Illustrazione dei partecipanti con le Lifelines:** rappresentano un'istanza di una classe
- **Definizione dei messaggi scambiati tra gli oggetti**

### 4.1 Operazioni di sistema e diagrammi di sequenza di sistema

Un diagramma di sequenza di sistema è un elaborato che illustra, per un particolare caso d'uso, eventi di input e di output relativi ai sistemi in discussione con un formato "a steccato" in cui gli oggetti che partecipano all'interazione sono mostrati in alto, uno a fianco dell'altro. Esso costituisce un input per i contratti delle operazioni e soprattutto per la progettazione degli oggetti.

I casi d'uso descrivono il modo in cui gli attori esterni interagiscono con il sistema software che interessa creare. Durante questa interazione, un attore genera **eventi di sistema**, che costituiscono un input per il sistema, di solito per richiedere l'esecuzione di alcune **operazioni di sistema**, che sono operazioni che il sistema deve definire proprio per gestire tali eventi.

#### 4.1.1 Contratti

Le sezioni di un contratto sono:

- **Operazione:** Nome e parametri dell'operazione
- **Riferimenti:** Casi d'uso in cui può verificarsi questa operazione
- **Pre-condizioni:** Ipotesi sullo stato del sistema prima dell'esecuzione
- **Post-condizioni:** Descrive i cambiamenti di stato degli oggetti nel modello di dominio dopo il completamento dell'operazione

### 4.2 Diagrammi di comunicazione

Mostrano le interazioni tra gli oggetti in un formato a grafo o a rete. in cui gli oggetti possono essere posizionati dovunque nel diagramma.

### 4.3 Pro/Contro diagrammi Sequenza/Comunicazione

I diagrammi di sequenza sono strumenti più potenti perché:

- UML è più incentrato sui diagrammi di sequenza
- Più facile vedere la sequenza "call-flow" poiché il tempo trascorre dall'alto verso il basso

I diagrammi di comunicazione sono strumenti più versatili perché:

- Effettuare modifiche è più semplice senza modificare l'intero call-flow
- Sono più comodi da disegnare

## 5 Diagramma delle classi di progetto

### 5.1 Generalizzazione

Una generalizzazione è una **relazione tassonomica** tra un classificatore più generale e un classificatore più specifico.

**NON equivale** all'ereditarietà nel **modello di dominio**

**Equivale** all'ereditarietà nel **modello di progettazione**, infatti qui la generalizzazione implica l'ereditarietà

### 5.2 Aggregazione e composizione

- **Aggregazione:** Tipo di associazione intero-parte (es. macchina-ruote)
- **Composizione:** Tipo di forte associazione intero-parte
  - Ciascuna istanza della parte appartiene a una sola istanza dell'intero alla volta
  - Ciascuna parte deve sempre appartenere ad un intero
  - La vita delle parti è limitata dall'intero: le parti possono essere create dopo l'intero, ma non prima e possono essere distrutte prima dell'intero, ma non dopo

## 6 Diagrammi di attività

Alcuni flussi di interazioni che devono essere supportati dal sistema possono essere **particolarmente elaborati e difficili da rappresentare con un caso d'uso**: è possibile utilizzare i diagrammi di attività in questo caso.

Nei diagrammi delle attività vengono rappresentati **azioni, flusso delle azioni, partecipanti, dati e flusso dei dati per diversi casi d'uso**.

Consentono di modellare un processo come un'attività costituita da un **insieme di nodi connessi da archi**.

Non è sempre chiaro quale parte dell'organizzazione esegue le varie attività, quindi i diagrammi possono essere in partizioni che mostrano le azioni intraprese da una singola organizzazione e classe.

## 7 Diagrammi di stato

Le macchine a stati possono essere utilizzate per **modellare il comportamento dinamico di classificatori** quali classi, casi d'uso, sottoinsiemi e interi sistemi.

Le macchine a stati esistono nel contesto di un particolare classificatore che risponde a eventi esterni, ha un **ciclo di vita definito** che può essere modellato come una successione di stati, transizioni ed eventi, e **può avere un comportamento corrente che dipende dai comportamenti precedenti**.

Solitamente vengono utilizzate per modellare il comportamento dinamico di classi.

Ogni macchina a stati ha uno **stato iniziale** che indica il primo stato della sequenza, e uno **stato finale** che termina la sequenza a meno che non esista un ciclo perpetuo di stati.

Uno **stato** è una **condizione o situazione della vita di un oggetto durante la quale tale oggetto soddisfa una condizione, esegue un'attività, o aspetta un evento**.

## 8 RDD

RDD (Responsibility-Driven Deployment) è un modello di progettazione oggetti basata su:

- Responsabilità, astrazione di ciò che si deve saper fare e responsabilità degli oggetti
- Ruoli, obiettivi e capacità che un oggetto o una classe ha di partecipare a una relazione con un altro oggetto
- Collaborazione tra oggetti per raggiungere un obiettivo.

Ci sono due tipologie di responsabilità, quella di fare e quella di conoscere. La granularità (fine o grossa) rappresenta la specificità delle azioni in base al loro scopo. Le responsabilità vengono assegnate durante la modellazione o durante la codifica. In contesto UML le responsabilità sono individuate mentre si creano i modelli statici e dinamici del sistema. I pattern sono principi generali e soluzioni idiomatiche per la creazione di software, codificati in un formato che descrive il problema e la soluzione

## 9 GRASP: Progettazione di oggetti con responsabilità

I GRASP definiscono nove principi di progettazione OO di base o blocchi di costruzione elementari della progettazione. GRASP è l'acronimo di General Responsibility Assignment Software Patterns, ovvero di "Pattern generali per l'assegnazione di responsabilità nel software".

**Creator** chi crea un oggetto particolare. In generale, i contenitori creano le cose contenute. La classe padre può contenere o aggregare con una composizione della classe figlia, utilizzarla strettamente o possederne i dati per l'inizializzazione

### Vantaggi

- Creator fornisce un accoppiamento basso, il che implica minori dipendenze di manutenzione e maggiori opportunità di riuso

**Information expert** è la classe che ha le informazioni necessarie per adempiere ad alcune responsabilità (classi rilevanti nel modello di progettazione, altrimenti in quello di dominio o analisi). Le classi che possiedono le informazioni necessarie per soddisfare una richiesta hanno responsabilità. (es. Per calcolare il totale di una vendita vengono chiamate in causa più classi)

#### **Vantaggi**

- L'incapsulamento delle informazioni viene mantenuto, poiché gli oggetti usano le proprie informazioni per adempiere ai propri compiti

**Low Coupling** dipendenza bassa, impatto dei cambiamenti basso e riuso elevato. L'accoppiamento indica quanto fortemente un elemento è connesso ad altri elementi, ha conoscenza di altri elementi e ne dipende.

#### **Vantaggi**

- Una classe o componente con un accoppiamento basso non è influenzata dai cambiamenti nelle altre classi e componenti
- Semplice da capire separatamente dalle altre classi e componenti
- Conveniente da riutilizzare

**High Coesion** indica quanto siano correlate e concentrate le responsabilità di un elemento. Le classi con coesione bassa sono difficili da comprendere, riutilizzare e mantenere. Una classe con coesione alta ha un numero di metodi relativamente basso, con funzionalità altamente correlate e non fa troppo lavoro

#### **Vantaggi**

- Modularità del software
- Riutilizzo delle classi
- Classi facili da mantenere

**Controller** il primo oggetto oltre lo strato di UI che è responsabile di ricevere e gestire un messaggio di un'operazione di sistema. L'oggetto deve rappresentare il sistema oppure uno scenario di un caso d'uso, in modo da verificare la sequenza delle operazioni e lo stato corrente delle attività. La logica è svincolata dall'UI e si possono usare interfacce diverse.

### **Vantaggi**

- Maggiore potenziale di riuso e interfacce inseribili
- Opportunità di ragionare sullo stato del caso d'uso (gestire le sessioni)



## 10 Design Patterns

### Pattern

- Descrive un problema specifico che ricorre più volte
- Descrive il nucleo della soluzione a quel problema in modo da poter utilizzare tale soluzione molte volte senza mai applicarla nello stesso modo

### Design Pattern

- Regola tripartita
- Esprime una relazione tra contesto problema e soluzione
- Nomina, astrae e identifica aspetti chiave di una struttura di design comune
- Identifica classi e istanze che partecipano, ruoli, modalità di collaborazione e distribuzione delle responsabilità

Sono divisi in 3 categorie:

- **Creational:** Pattern che forniscono un'astrazione del processo di istanziazione degli oggetti. Permettono di rendere un sistema indipendente da come gli oggetti sono creati/rappresentati/composti al suo interno
- **Structural:** Dedicati alla composizione di classi e oggetti per formare strutture complesse. È possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti. Sono particolarmente utili per fare in modo che librerie di classi sviluppate indipendentemente possano operare insieme
- **Behavioral:** Sono dedicati all'assegnamento di responsabilità tra gli oggetti e alla creazione di algoritmi. L'utilizzo di questi pattern permette di dedicarsi principalmente alle connessioni tra oggetti, tralasciando la gestione dei flussi di controllo

## 10.1 Alcuni Design patterns

### 10.1.1 Adapter

**Problema** Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a componenti simili ma con interfacce diverse?

**Soluzione** Converti l'interfaccia di un componente in un'altra interfaccia, attraverso un oggetto **adapter** intermedio.

### 10.1.2 Factory

**Problema** Chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica di creazione complessa, quando si desidera separare la responsabilità di creazione per una coesione migliore, e così via?

**Soluzione** Crea un oggetto **pure fabrication** chiamato **factory** che ne gestisce la creazione

### 10.1.3 Singleton

**Problema** E' consentita esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

**Soluzione** Definisci un metodo statico della classe che restituisce l'oggetto singleton

### 10.1.4 Strategy

**Problema** Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?

**Soluzione** Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune

### 10.1.5 Composite

**Problema** Come trattare un gruppo o una struttura composta di oggetti polimorficamente dello stesso tipo nello stesso modo di un oggetto non composto?

**Soluzione** Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

### 10.1.6 Façade

**Problema** E' richiesta un'interfaccia comune e unificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema.

**Soluzione** Definisci un punto di contatto singolo con il sottosistema (un oggetto **façade**) che copre il sottosistema. Questo oggetto presenta un'interfaccia singola e unificata.

### 10.1.7 Observer

**Problema** Diversi tipi di oggetti **subscriber** sono interessati ai cambiamenti di stato o agli eventi di un oggetto **publisher**

**Soluzione** Definisci un'interfaccia **subscriber** o **listener**