Object Oriented Design

Antonia Sacchitella
Analyst @icubedsrl

Antonia.sacchitella@icubed.it

L’importanza della comunicazione

La comunicazione è uno di **problemi principali** che esiste nel business moderno ed in particolare nello sviluppo software.

Ogni giorno ci si confronta con persone di diversa **provenienza**, diversa **cultura**…ma soprattutto diverse **competenze**.

I 6 ruoli nel team di sviluppo software (1)

Quando si approccia lo sviluppo di un sistema, sono 6 i ruoli (o le figure) che normalmente compongo il team:

«Product owner» – il proprietario del prodotto finale.

«Business Analysts» – si pongono come figura «di mezzo» tra gli owner del prodotto e il team tecnico.

«Software/solution Architect» - progetta i sistemi, identifica il modo in cui gli sviluppatori dovrebbero costruire il sistema e si pone come guida per tutti gli aspetti tecnici relativi al progetto.

icubed

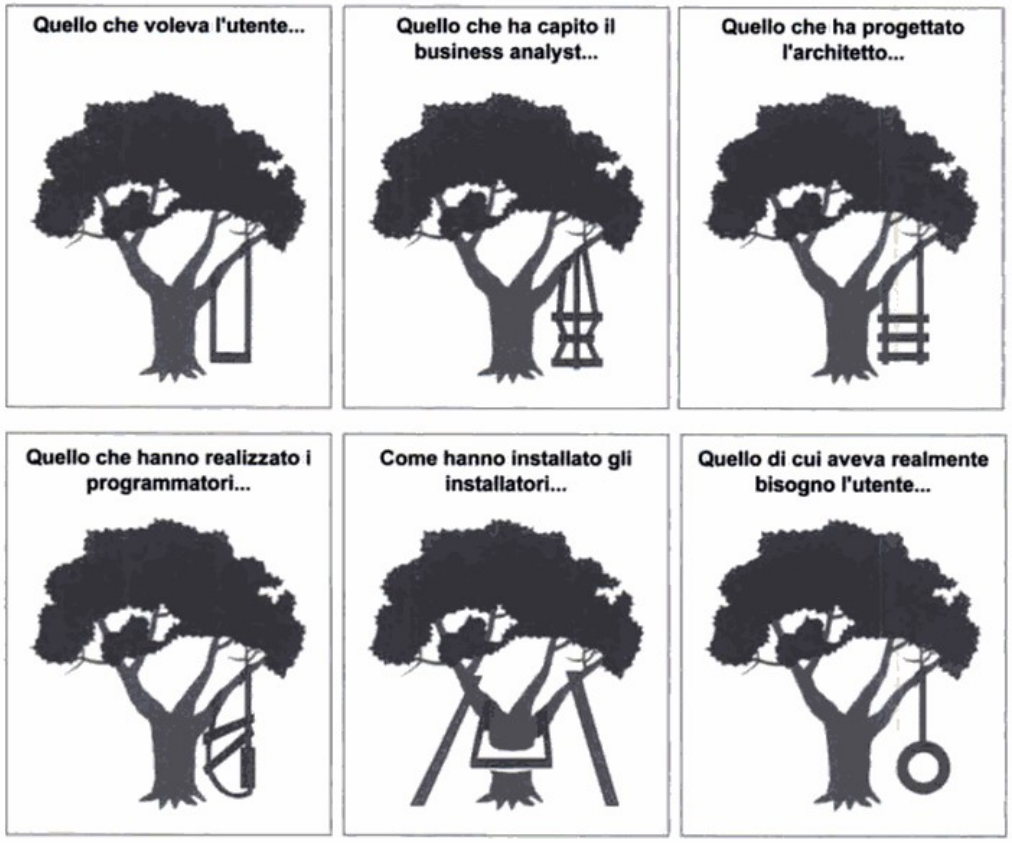
I 6 ruoli nel team di sviluppo software (2)

«Sviluppatori» – Ereditano i disegni funzionali e tecnici redatti dagli analisi e dall'architect

«Team QA» (Quality Assurance) – Detto anche team di «Testing», si assicura che il software scritto dal team di sviluppo corrisponda ai requisiti definiti in fase di analisi. Si assicura che il sistema risponda sia in termini di logica funzionale, sia in termini di performance e di sicurezza.

«Team Operations» – Prende il software finito (e testato) e predispone gli ambienti di esecuzione temporanea (Staging) e finali (Production) occupandosi della configurazione di quelli che sono i requisiti tecnici di esecuzione, i

L’importanza di analisi e design



icubed

Analisi e Disegno: gli errori più comuni

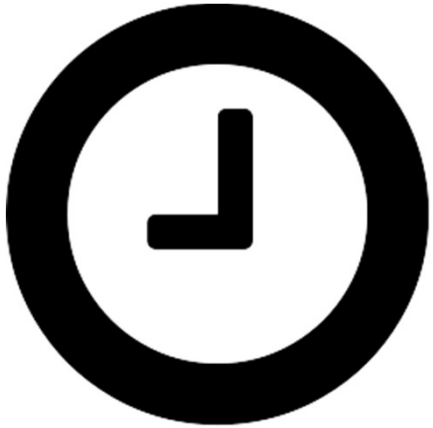
Spesso si tende a sottovalutare o minimizzare la fase di analisi e disegno di un sistema informatico per differenti motivi:

Questa sottovalutazione porta spesso a problemi ben più gravi:

Ritardi nella consegna del prodotto finale

Aumento dei costi del progetto e sforamento del budget

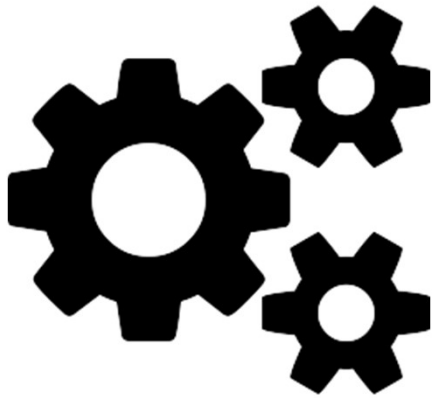
In alcuni casi addirittura al **fallimento del progetto** stesso



icubed

Analisi e Design «orientato agli oggetti»

«E’ un approccio fortemente tecnico per analizzare e disegnare un’applicazione, un sistema oppure un business, applicando tanto il paradigma object-oriented, quanto utilizzando la modellazione visiva durante i cicli di vita dello sviluppo.»



icubed

Analisi e Design «orientato agli oggetti»

E’ un approccio che mostra i benefici in maniera più evidente se applicato in maniera **iterativa e incrementale**, coinvolgendo nel processo di analisi diversi membri del team (funzionale e tecnico).

icubed

Un po’ di storia...

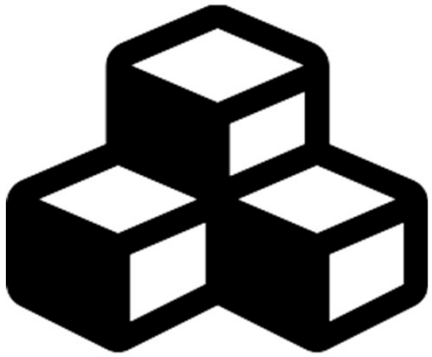
Prima della metà degli anni ‘90 esistevano differenti metodologie, in competizione tra loro, per l’analisi di sistemi complessi; **non esistevano standard** e termini consistenti.

Grady Booch, James Rumbaugh e Ivar Jacobson (“The Three Amigos”), su tutti, furono responsabili dello sviluppo di Unified Modeling Language (UML) nel 1994.

La standardizzazione ufficiale di **UML**, avvenne nel 1997 con l’uscita della versione 1.1: da quel momento la **convergenza delle metodologie** di definizione di sistemi complessi può dirsi completata

Object Oriented Modelling: cos'è?

La modellazione orientata agli oggetti («Object Oriented Modeling» o OOM) è un approccio comune alla modellazione di applicazioni, sistemi e domini aziendali, che utilizza il **paradigma orientato agli oggetti** in tutti i cicli di vita dello sviluppo.



icubed

Object Oriented Modelling: i benefici

La modellazione object-oriented si divide tipicamente in due aspetti del lavoro:

la modellazione di comportamenti **dinamici** come processi aziendali e casi di uso («use case»)

la modellazione di strutture **statiche** come classi e componenti

I principali benefici sono:

Comunicazione efficiente ed efficace

- riduce nella maniera più naturale possibile il «gap semantico» che esiste tra un utente funzionale che conosce il business, e un tecnico che deve realizzare il sistema

Astrazione utile e stabile

- aiuta a codificare in maniera efficiente, producendo descrizioni astratte ma accessibili sia dei requisiti di sistema, sia dei modelli logici in gioco, favorendo una overview dell'intero processo

icubed

Object Oriented Analysis: introduzione

Lo scopo di qualsiasi attività di analisi nel ciclo di vita del software è quello di creare un modello di **requisiti funzionali** del sistema indipendente dai vincoli di implementazione.

Nelle altre metodologie di analisi tradizionali, i due aspetti - processi e dati - sono considerati separatamente. Ad esempio, i dati possono essere modellati da diagrammi ER (Entity-Relation), comportamenti da diagrammi di flusso o da schemi di struttura.

Object Oriented Analysis: obiettivi

I compiti primari in una analisi object-oriented (OOA) sono:

Individuare gli oggetti in gioco

Organizzare gli oggetti

Descrivere come **interagiscono** gli oggetti tra loro

Definire il **comportamento** degli oggetti

Definire le **implementazioni interne** degli oggetti senza scendere in dettagli su come le implementazioni devono essere realizzate

icubed

Object Oriented Design

Durante il processo di Object Oriented Design (OOD), uno sviluppatore applica vincoli di implementazione al **modello concettuale** prodotto in Object Oriented Analysis.

Tali vincoli potrebbero includere le piattaforme hardware e software da utilizzare, requisiti di prestazione e archiviazione, usabilità del sistema e limitazioni imposte dai budget e dal tempo.

icubed

Raccolta dei requisiti

Il ciclo di vita del software è tipicamente suddiviso in fasi che vanno dalle descrizioni astratte del problema, al disegno, alla codifica e test, e infine alla fase di deploy (distribuzione all'utente finale).

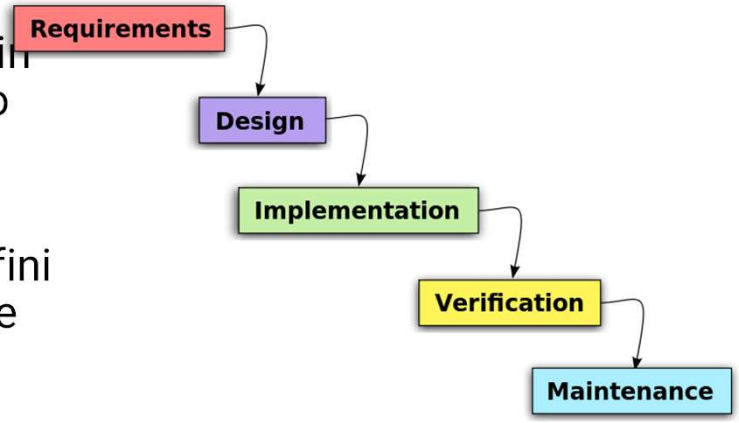
Le prime fasi di questo processo sono analisi e progettazione. La fase di analisi è anche spesso chiamata «Raccolta dei requisiti» o «Requirements Acquisition».

L'utente finale – in generale l'attore che svolge il ruolo di rappresentante del processo di business – espone **le problematiche** che hanno portato alla necessità di adottare/produrre una automazione basata su un modello software.

Metodologia di analisi «waterfall»

Il processo di Object Oriented Analysis e Design è condotto in modo iterativo e incrementale, come formulato dal processo unificato.

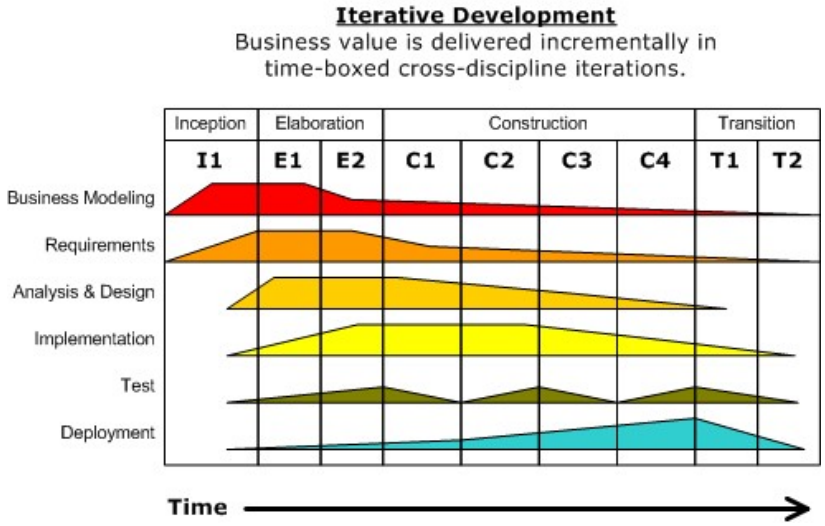
In alcuni approcci allo sviluppo del software - conosciuti collettivamente come modelli «waterfall» (a cascata) - i confini tra ciascuna fase sono destinati a essere abbastanza **rigidi** e **sequenziali**.



icubed

Metodologie di analisi «iterative»

Con i modelli iterativi è possibile lavorare su diverse fasi in **parallelo**. Ed esempio è possibile - e non viene considerato come una fonte di errore - lavorare sull'analisi, il design e addirittura codificare tutto nello stesso giorno, accettando di avere problemi di **impatto minimale** tra una fase e l'altra.



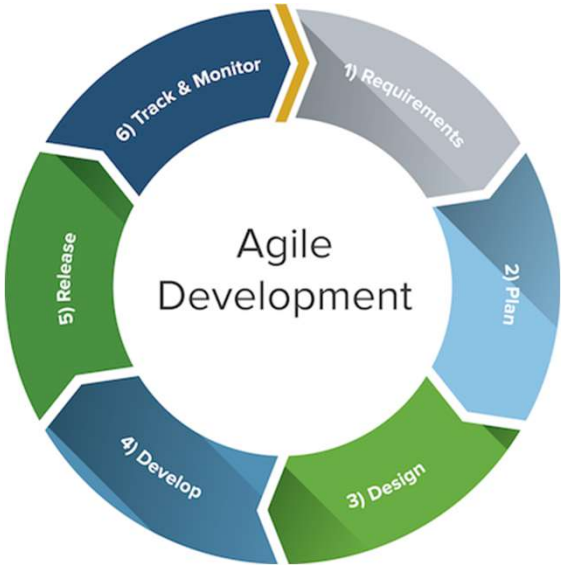
icubed

Metodologia «Agile»

La metodologia «agile» descrive una serie di principi per lo sviluppo del software in base alle quali i requisiti e le soluzioni si evolvono attraverso lo sforzo di collaborazione di organizzazioni intra-funzionali di autoorganizzazione.

Promuove la **pianificazione adattabile**, lo sviluppo evolutivo, la consegna anticipata e il miglioramento continuo, e incoraggia una risposta rapida e flessibile al cambiamento.

Il termine «Agile» è stato adottato dagli autori del «Manifesto per lo sviluppo di software agile» («Agile Manifest») pubblicato nel 2001 da Kent Beck, Robert C. Martin, Martin Fowler e altri.



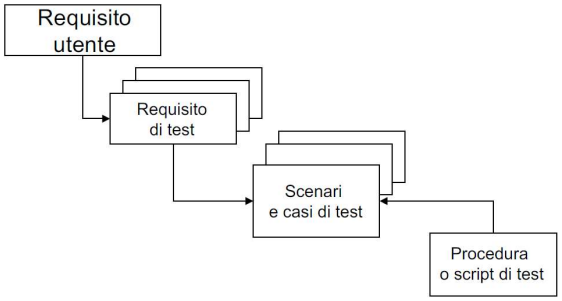
icubed

Stesura dei requisiti di un sistema

Requisiti software («software Requirements») è la descrizione dei servizi che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività del software stesso.

Sono solitamente suddivisi in:

Requisiti utente («user requirements»): descrizione in linguaggio naturale - con l'aggiunta di eventuali diagrammi – dei servizi e delle funzioni che il sistema deve offrire unitamente ai vincoli operativi.



Requisiti di sistema («system requirements»):
specificati mediante la stesura di un documento strutturato, che descrive in modo dettagliato i servizi che il sistema deve fornire unitamente ai vincoli tecnologici che sono eventualmente posti

icubed

Stesura dei requisiti di un sistema

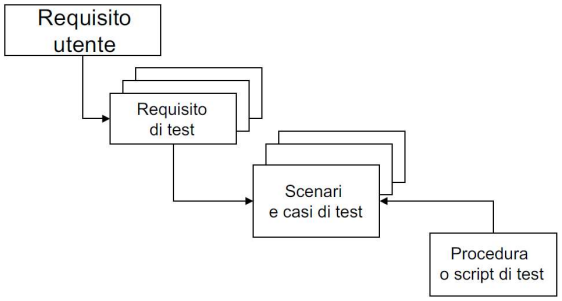
Requisiti software («software Requirements») è la descrizione dei servizi che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività del software stesso. Il documento risultante costituisce un «contratto» tra cliente e fornitore e serve a stabilire:

cosa richiede il cliente che il software oggetto dell'analisi deve possedere

evitando di definire **come** il sistema verrà costruito (parti, scelte architetturali o tecniche)

Esistono delle semplici regole per scrivere un buon documento di requisiti:

- Definire il glossario (**nomi** e **verbi**), unificando termini e sinonimi
- **Specificare** ciò che è troppo generico, introducendo elementi mancanti
- Specificare lo **scenario** delle operazioni (che porteranno poi alla costruzione dei test per i requisiti stessi, scenari di attivazione e procedure di esecuzione test)



icubed

La piramide dei requisiti

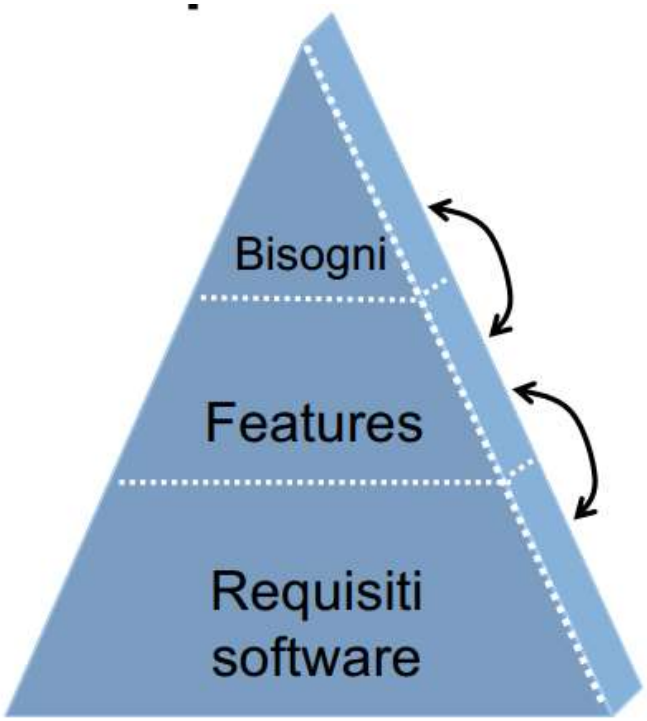
In cima sono i **bisogni** del cliente. Esempio: “I dati dei clienti e dei fornitori dovranno essere conservati per 10 anni all’interno dello storage".

Ai bisogni danno risposta le **features**: ogni feature corrisponde ad un bisogno ed ha proprietà (requisiti funzionali o meno) che il sistema finito dovrà possedere e che verranno controllate mediante test

Un **requisito** può essere una descrizione astratta e generica di un servizio o vincolo di sistema, oppure una specifica concreta e dettagliata di una funzionalità del sistema. I requisiti hanno infatti una doppia funzione:

– Base per un’offerta di contratto: devono essere **aperti** a varie interpretazioni

– Base per un contratto: devono essere **dettagliati** e non soggetti ad interpretazioni arbitrarie



icubed

Differenza tra feature e requisito

Un requisito descrive una **capacità** che deve possedere il sistema; è scritto per testare l'implementazione in relazione agli scenari di uso del sistema stesso

Una feature è un **insieme di requisiti** logicamente correlati; di solito descrive una funzionalità a «grana grossa» di un prodotto

Esempio:

Feature:

- Carrello in un sito di e-commerce

Requisiti:

- L'utente potrà aggiungere elementi al carrello
- L'utente potrà rimuovere elementi dal carrello
- L'utente potrà iniziare il pagamento dal carrello

icubed

Definizione formale dei requisiti

I requisiti sono «proprietà» di un sistema o prodotto «desiderate» dai suoi committenti e «verificabili» in fase di validazione del sistema (tramite il documento di UAT, «User Acceptance Tests»)

Quali proprietà?

- Proprietà «funzionali» oppure «non-funzionali», quindi di business oppure tecniche

Che succede se alcuni requisiti confliggono?

- Ci si basa sul concetto di «Priorità» oppure di «Preferenza» per risolvere la situazione

Chi sono i committenti e come si registrano i relativi requisiti?

- I committenti sono gli «stakeholders» che usano viste specializzate

icubed

Definizione formale dei requisiti

Come si verifica che un requisito sia soddisfatto?

- Tramite azioni di Testing (di unità, E3E, integration, load, stress) eseguite dagli sviluppatori
- Tramite «Validazione», fatta dai committenti che verificano che il sistema svolga i sui compiti

Cosa non è considerabile un requisito?

L'architettura del sistema

Vincoli tecnologici (es. linguaggio di implementazione)

Il processo di sviluppo

L'ambiente di sviluppo

Il sistema operativo di riferimento

Aspetti di riusabilità e portabilità

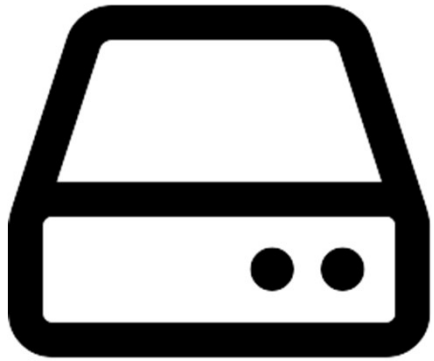
Requisiti «funzionali» e «non funzionali»

Requisiti funzionali: descrivono le interazioni tra il sistema ed il suo ambiente, indipendentemente dall'implementazione

- Es. «La piattaforma e-learning tiene traccia delle attività dello studente»

Requisiti non funzionali: proprietà del sistema misurabili dall'utente e non direttamente correlate al suo comportamento funzionale

- Es. «La piattaforma deve gestire almeno 100 utenti contemporaneamente»
- Es. «Se un utente fa una domanda per email deve avere risposta entro 48h»
- Es. «La piattaforma deve essere disponibile agli studenti 24/7»

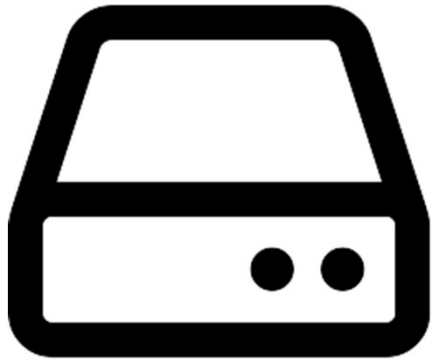


icubed

Requisiti «funzionali» e «non funzionali»

Vincoli (“Pseudo requisiti”): imposti dal cliente o dall’ambiente operativo in cui funzionerà il sistema

- Es. «Il linguaggio di programmazione dev’essere Java»
- Es. «La piattaforma deve interfacciarsi con documenti scritti con Word su Windows 10»

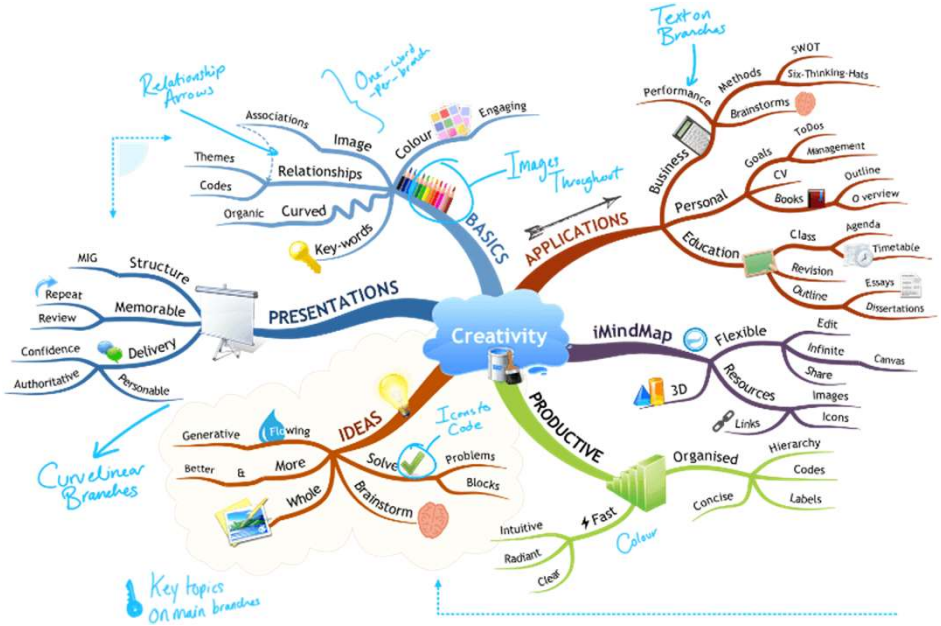


icubed

Mind Maps: organizzazione delle idee

Una **mappa mentale** è una forma di rappresentazione grafica del pensiero teorizzata dal cognitivista inglese Tony Buzan, a partire da alcune riflessioni sulle tecniche per prendere appunti.

Le mappe mentali (mind maps) non vanno confuse con altri tipi di mappe come le mappe concettuali dalle quali si differenziano sia per la strutturazione, sia per il modello realizzativo, sia per gli ambiti di utilizzo.



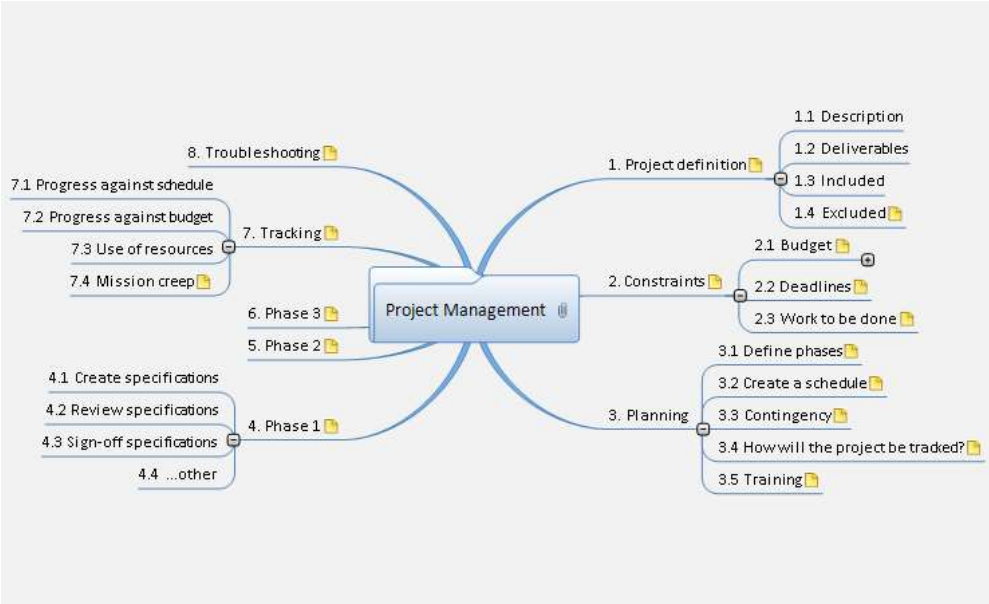
icubed

Struttura di una mappa mentale

Le mappe mentali hanno una struttura gerarchico - associativa. Questo significa che sono solo due le tipologie di connessioni che possono essere create:

gerarchiche (dette anche rami) che collegano ciascun elemento con quello che lo precede;

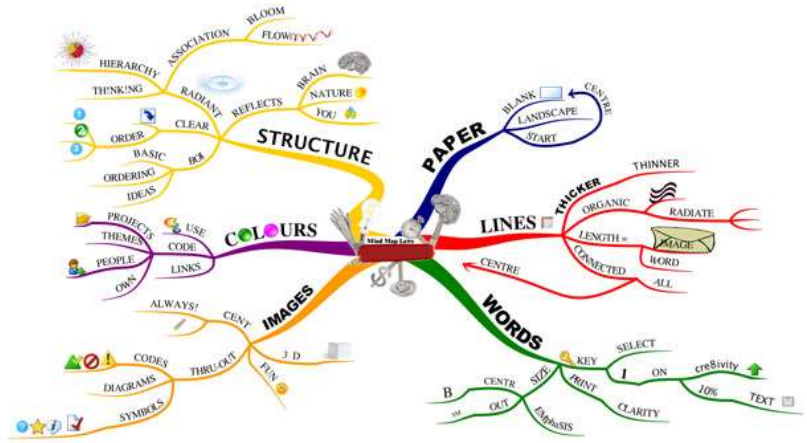
associative (dette anche associazioni) che collegano elementi gerarchicamente disposti in punti diversi della mappa.



icubed

Mind maps: creatività ed associazione

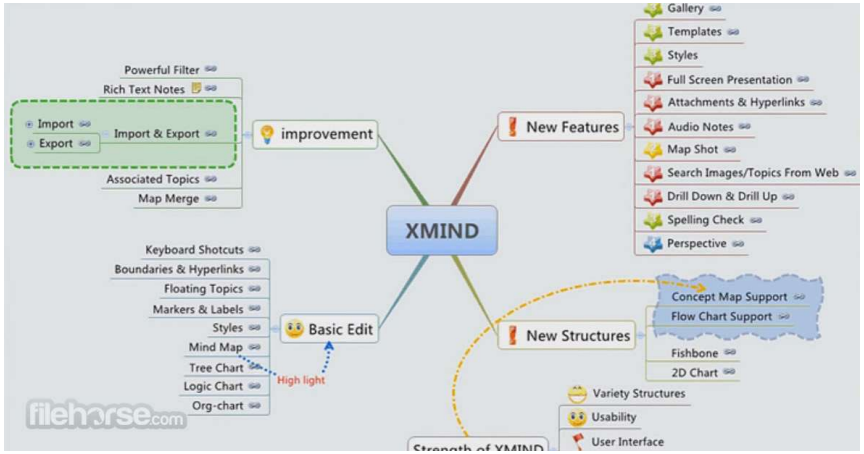
Per questo il suo ideatore Tony Buzan ha formulato il suo modello incentrandolo sull'evocatività: tutti gli elementi di una mappa mentale devono essere ricchi di **immagini** fantasiose e colorate, perché da un lato rendono gradevole la rappresentazione, dall'altro stimolano l'emisfero cerebrale destro, le cui funzioni supportano facoltà come la creatività, la memoria, l'associazione mentale.



icubed

Mind maps: requirements gathering

Quando ci si incontra con i «product owner» - o coloro che contribuiranno alla definizione dei requisiti – ci si trova spesso a portare avanti la discussione senza un reale filo logico; tutti parlano a «ruota libera» e sta alla bravura dell'analista percepire le caratteristiche che dovrà avere un sistema complesso.



Il processo di sviluppo

Il metodo di sviluppo software è composto da un **processo** di sviluppo e un **linguaggio** di definizione del modello.

I processi di sviluppo sono invece **molteplici**; tra tutti quello identificato come RUP (Rational Unified Process) è quello che maggiormente si integra con UML perché nato dall'esperienze delle stesse persone che hanno dato vita a UML stesso.

La flessibilità di UML ci permette di prendere solo le parti necessarie, integrando il resto con altre metodologie

Passi fondamentali di RUP

E’ essenzialmente composto da quattro distinte fasi:

Principio – valutazione dello scopo del processo, dimensionamento dell’effort e dei costi e prima analisi del processo di business

Elaborazione – raccolta dei requisiti dettagliata, analisi di dettaglio, progettazione e prima ipotesi di «planning»

Costruzione – pianificazione dettagliata e multiple iterazioni che portano alla realizzazione del sistema, ciascuna delle quali soddisfa un requisito del prodotto finale

Transizione – addestramento utenti, beta testing, miglioramento delle prestazioni, refactoring e bug fixing

icubed

RUP: la fase di elaborazione

E' la fase in cui si valutano i rischi associati a progetto:

Rischi sulle abilità : sincerarsi di avere le giuste competenze per affrontare il progetto (si mitiga con la formazione o l'acquisizione di nuove risorse con skills)

Rischi tecnologici : verificare di aver selezionato la tecnologia più adatta per realizzare il prodotto finale (si mitiga avendo all'interno del team persone con forte esperienza «cross platform»)

Rischi sui requisiti: assicurarsi di aver identificato in maniera corretta tutti i requisiti richiesti dal cliente

icubed

RUP: rischi sui requisiti

Rappresenta sempre il **problema più grosso** quando si procede ad una analisi di un sistema, e la fase per cui non esistono «rimedi» ad applicazione matematica

L'adozione di **use case** è da considerarsi un'ottima pratica per mitigare le incomprensioni tra cliente e team di sviluppo (analisti, architetti e sviluppatori)

La definizione del **modello logico** è uno step fondamentale per strutturare le entità in gioco e le interconnessioni tra le stesse («class» ed «activity diagram»)

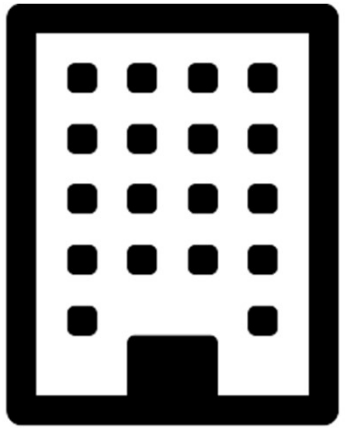
icubed

RUP: la fase di costruzione

Composta da due sotto-fasi subordinate temporalmente:

Pianificazione – definizione dettagliata di una timeline contenente l'elenco delle attività in carico a ciascun attore coinvolto (compreso il cliente), ciascuna con data di inizio e fine precisa, evidenziando parallelismi e eventuali task di dipendenza (senza i quali non è possibile procedere)

Costruzione – molteplici iterazioni, ciascuna delle quali è rappresentata da uno «use case» che viene analizzato, progettato e codificato, producendo (ove possibile) una «demo» che mostra l'espletamento del requisito stesso (es. utilizzando gli Unit Test)



Object Oriented Programming

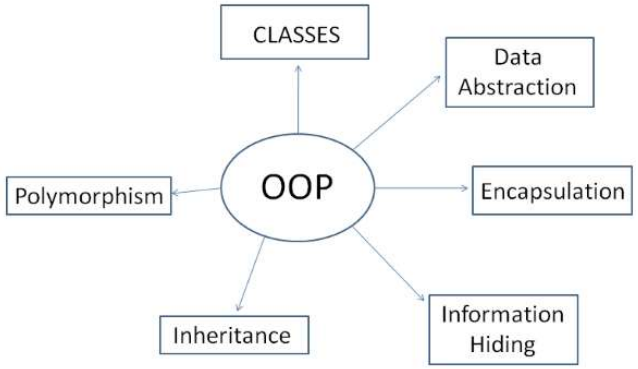
«La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi.»

È particolarmente adatta nei contesti in cui si possono definire delle **relazioni di interdipendenza** tra i concetti da modellare (contenimento, uso, specializzazione).

Fornisce un **supporto naturale** alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre

Permette una più **facile gestione** e manutenzione di progetti di grandi dimensioni

L'organizzazione del codice sotto forma di classi favorisce la modularità e il **riuso di codice**



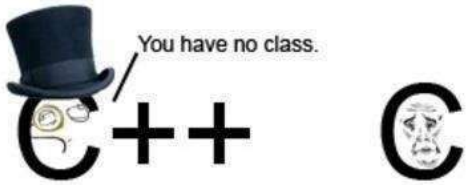
icubed

Quando un linguaggio è «object oriented»?

Un linguaggio di programmazione è definito «ad oggetti» quando permette di implementare tre meccanismi usando la sintassi nativa del linguaggio:

Incapsulamento

- consiste nella separazione della cosiddetta interfaccia di una classe dalla corrispondente implementazione, in modo che i client di un oggetto di quella classe possano utilizzare la prima, ma non la seconda



Ereditarietà

- permette essenzialmente di definire delle classi a partire da altre già definite

Polimorfismo

- permette di scrivere un client che può servirsi di oggetti di classi diverse, ma dotati di una stessa interfaccia comune; a tempo di esecuzione, quel client attiverà comportamenti diversi senza conoscere a priori il tipo specifico dell'oggetto che gli viene passato

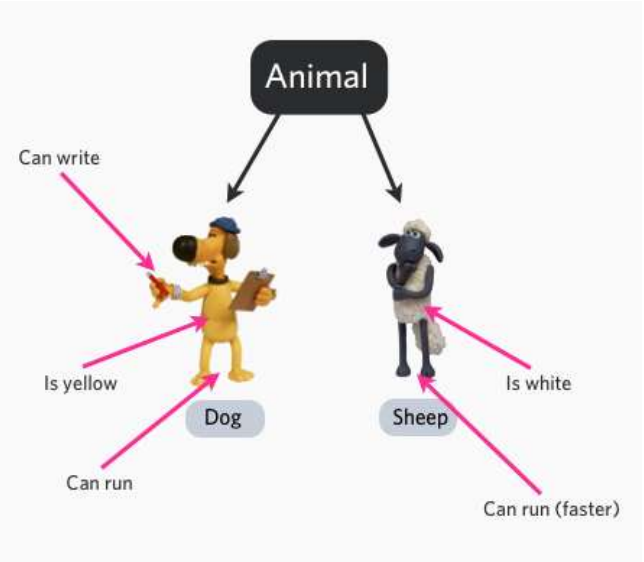
icubed

OOP: le classi e gli oggetti

Una classe prevede di raggruppare in una «zona circoscritta del codice sorgente» la dichiarazione delle strutture dati e delle procedure che operano su di esse.

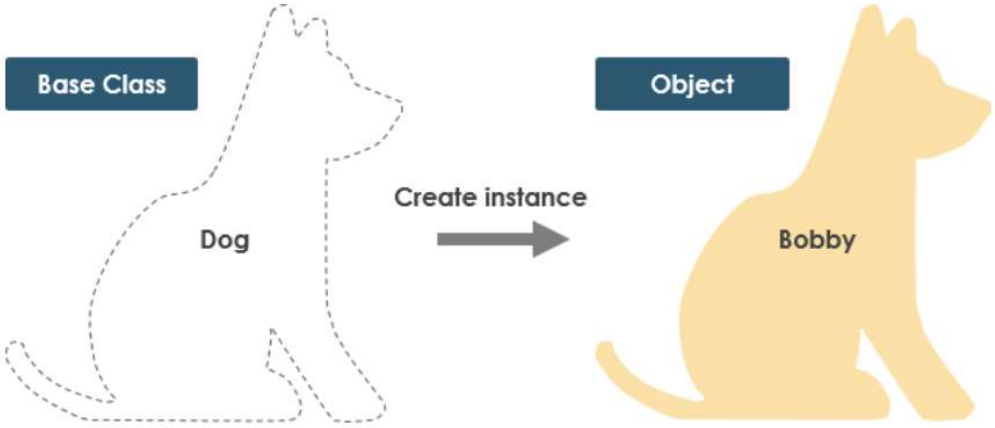
Le classi costituiscono dei modelli astratti, che a tempo di esecuzione vengono invocate per istanziare o creare oggetti modellati sulla classe stessa.

Questi ultimi sono dotati di attributi (dati) e metodi (procedure) secondo quanto definito/dichiarato dalle rispettive classi di appartenenza.



icubed

OOP: le classi e gli oggetti



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

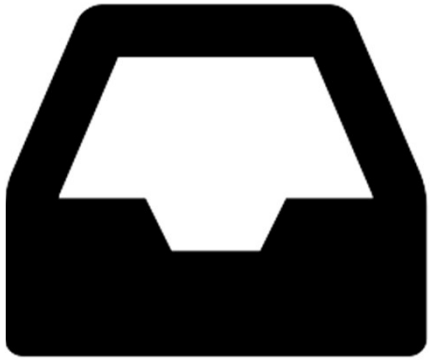
icubed

OOP: l'incapsulamento

L'incapsulamento è la proprietà per cui i dati che definiscono lo stato interno di un oggetto e i metodi che ne definiscono la logica sono accessibili ai metodi dell'oggetto stesso, mentre non sono visibili all'esterno della classe.

Per alterare lo stato interno dell'oggetto, è necessario invocarne i metodi pubblici, ed è questo lo scopo principale dell'incapsulamento.

Se gestito opportunamente, esso permette di vedere l'oggetto come una black-box, cioè una "scatola nera" di cui, attraverso l'interfaccia, è noto **cosa** fa, ma non **come** lo fa.



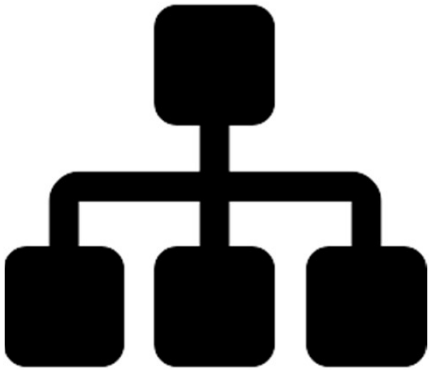
icubed

OOP: l'ereditarietà

Il meccanismo dell'ereditarietà permette di derivare nuove classi a partire da quelle già definite realizzando una gerarchia di classi. Una classe ereditata attraverso questo meccanismo:

Mantiene i metodi e gli attributi delle classi da cui deriva (classi base, superclassi o classi padre)

Può definire i propri metodi o attributi, e ridefinire il codice di alcuni dei metodi ereditati tramite un meccanismo chiamato overriding.



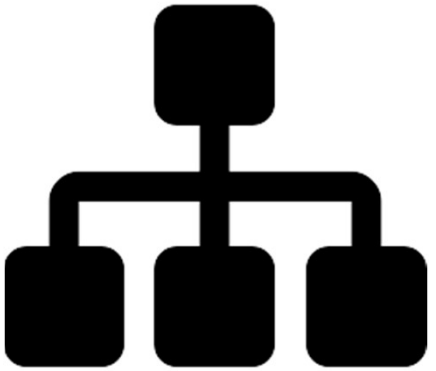
icubed

OOP: l'ereditarietà

Quando una classe eredita da una sola superclasse si parla di eredità singola; viceversa, si parla di eredità multipla. Alcuni linguaggi (tra gli altri, Java, Smalltalk) forniscono supporto esclusivo all'ereditarietà singola. Altri (C++, Python) prevedono anche l'ereditarietà multipla.

L'ereditarietà può essere usata come meccanismo per ottenere l'estensibilità e il riuso del codice, e risulta particolarmente vantaggiosa quando viene usata per definire sottotipi, sfruttando le relazioni esistenti nella realtà di cui la struttura delle classi è una modellizzazione.

Oltre all'evidente riuso del codice della superclasse, l'ereditarietà permette la definizione di codice generico attraverso il meccanismo del polimorfismo.

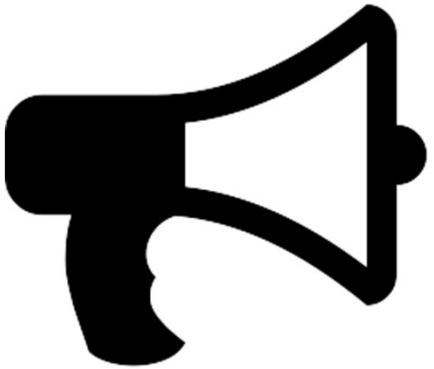


UML: le sue caratteristiche fondamentali

Si tratta di un linguaggio di modellazione usato per capire e descrivere le caratteristiche di un nuovo sistema o di uno esistente.

E’ progettato per essere abbinato alla maggior parte dei linguaggi «object-oriented», e per tale motivo è spesso utilizzato in combinazione con un processo di sviluppo vincolato a quella tipologia di approccio

E’ un vero linguaggio, non una semplice notazione grafica; è dotato di una sintassi chiara, non ambigua; ha regole sintattiche (produrre modelli legali) e semantiche (che definiscono il comportamento di quei modelli)



icubed

Cos'è il linguaggio UML

Unified Modeling Language, è un linguaggio semi-formale e grafico (basato su diagrammi) per:

Specificare

Visualizzare

Realizzare

Modificare

Documentare

gli artefatti di un sistema complesso (solitamente software).

Un «artefatto» è un qualunque prodotto tangibile del progetto: sorgenti, eseguibili, documentazione, file di configurazione, tabelle di database, benchmark, …

icubed

UML: perché imparare un nuovo linguaggio

Acquistereste a «scatola chiusa» un appartamento costruito dando come requisiti all'architetto il fatto che sia dotato di 2 camere da letto, 2 bagni e che sia grande circa 90 metri quadri?

Il medesimo approccio è applicabile quando si approccia la creazione di un sistema software complesso; il processo di realizzazione del prodotto finale sarà coadiuvato da una «piantina» della casa – i diagrammi di cui è dotato UML – che non farà altro che descrivere in maniera semplice, intuitiva ma esauriente a tutti i livelli le caratteristiche tecniche che dovranno essere implementate.

icubed

UML: i principali vantaggi dell'adozione

- Possibilità di avere un'idea chiara del risultato finale di progetto
- Maggiore semplicità nella scrittura del codice
- Facile previsione dei «buchi» di sistema
- Migliore comunicazione con persone non esperte tecnicamente

icubed

UML: la tassonomia dei 13 diagrammi

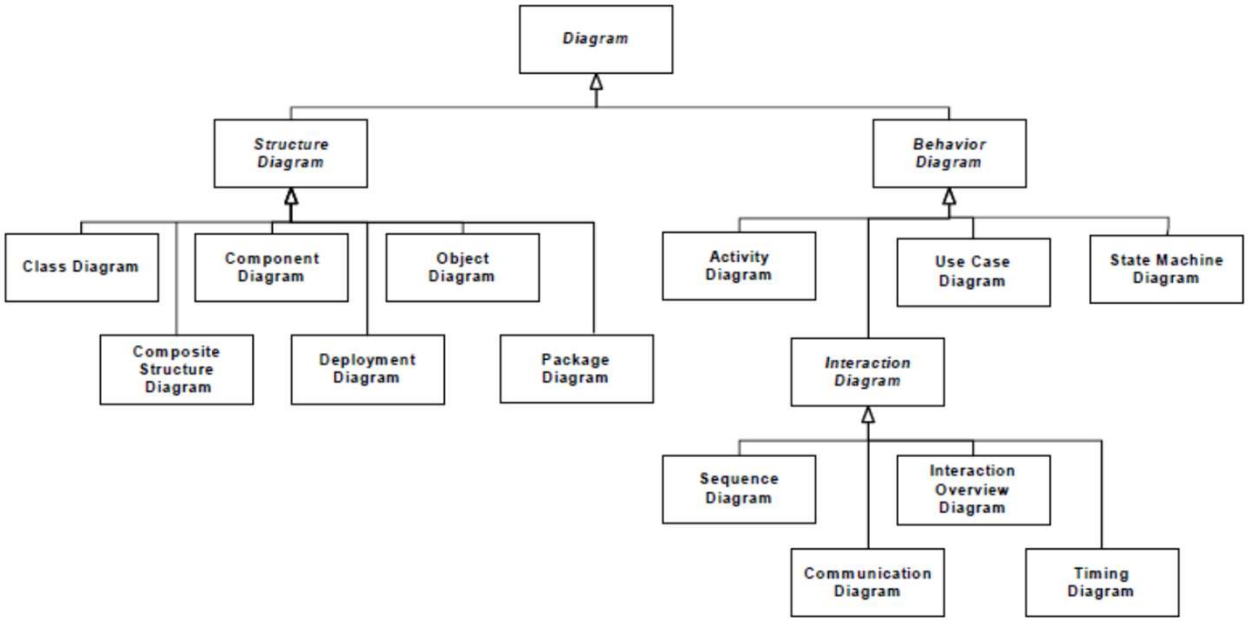
Il linguaggio UML contiene svariati elementi grafici che vengono messi insieme durante la creazione dei diagrammi. Poiché l'UML è un linguaggio,

Utilizza delle regole per combinare i componenti del linguaggio nella creazione dei diagrammi.

UML 2 definisce un totale di 13 diagrammi (contro i 9 di UML 1.x), divisi in due categorie:

Structure diagrams: come è fatto il sistema; forniscono le viste «Logical», «Development» e «Physical»

Behavior diagrams: come funziona il sistema; forniscono le viste «Process» e «Use case»



icubed

Una semplice domanda per iniziare…

Una domanda che ci si pone all'inizio imparando UML è la seguente:

«Ma è proprio necessario sviluppare tutti e tredici i diagrammi che l'UML mette a disposizione per tutti i progetti?»

La risposta a questa domanda può variare in relazione alla complessità del sistema che si intende costruire ma, in linea di massima, tende ad essere affermativa.

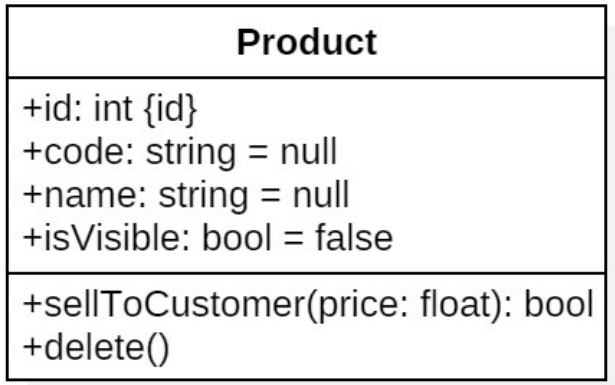
L consiste di 13 diagrammi di base, ma si tenga presente che è assolutamente possibile costruire e aggiungere dei diagrammi differenti dagli standard (che vengono definiti ibridi) rispetto a quelli definiti dal linguaggio, creando di fatto nuovi artefatti in grado di soddisfare esigenze differenti

icubed

Primitive di UML: Classe

Il «Class Diagram» del linguaggio UML consiste di svariate classi connesse tra di loro tramite delle relazioni.

- La classe viene rappresentata da un rettangolo.
- Un «attributo» rappresenta una proprietà di una classe; esso descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe (tipo) e un valore di default.
- Un' «operazione» è un'azione che gli oggetti di una certa classe possono compiere.



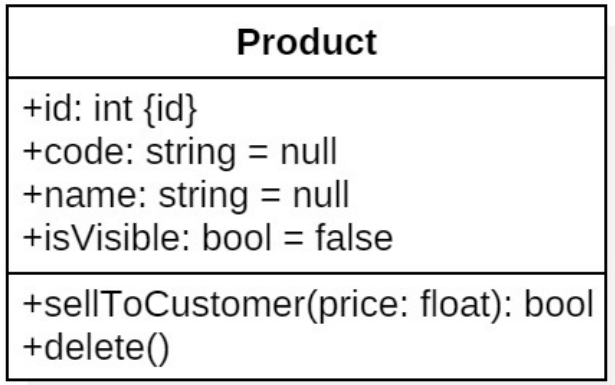
icubed

Primitive di UML: Classe

Associati ad attributi ed operazioni possono essere specificati dei «constraint», cioè vincoli legati all'elemento, e delle «note», che contengono una descrizione associata all'elemento stesso. Allo stesso modo è possibile applicare marcatori di «visibilità»: «public» (+), «private» (-) e «protected» (#)

Prestare molta attenzione ai «nomi» che i clienti usano per descrivere le entità del loro business.

Prestare attenzione ai «verbi» che vengono pronunciati dai clienti. Questi costituiranno, con molta probabilità, i metodi (le operazioni) nelle classi definite.



icubed

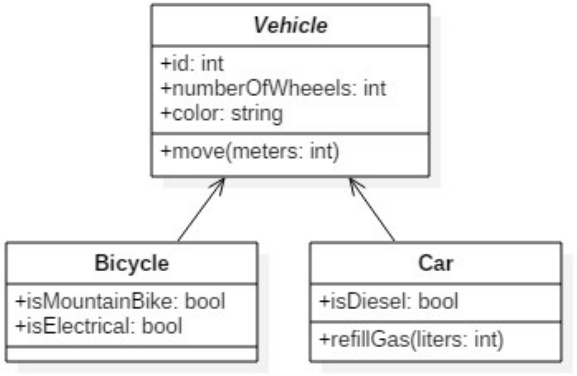
Primitive di UML: Ereditaretà

In UML l'ereditarietà viene rappresentata con una linea che connette la classe padre alla classe discendente e dalla parte della classe padre si inserisce un triangolo (una freccia).

Quando gli attributi e le operazioni per una classe siano generali si applicano a parecchie altre classi che, a loro volta, potranno "specializzare" la classe padre aggiungendo attributi e operazioni propri.

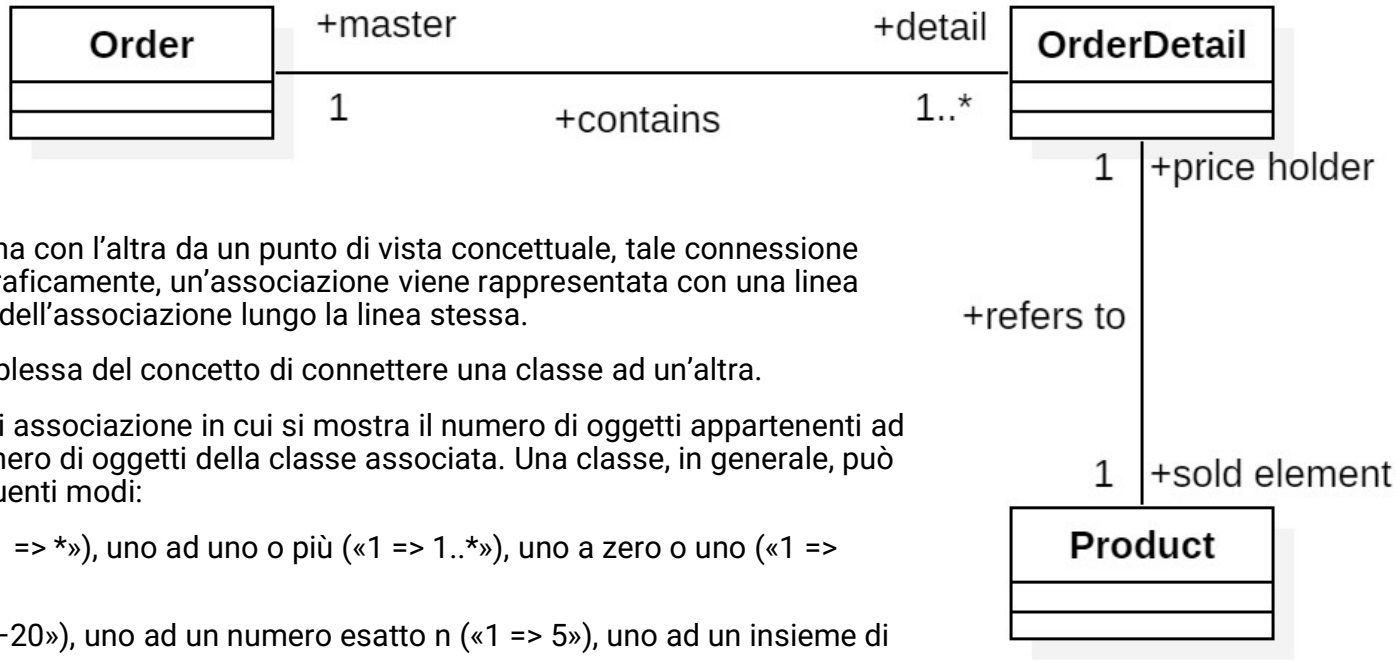
Un'altra possibilità è che si noti che due o più classi hanno un numero di attributi ed operazioni in comune.

Le classi che non permettono di istanziare nessun tipo di oggetto sono dette classi «astratte». In UML una classe astratta si indica scrivendo il suo nome in corsivo.



icubed

Primitive di UML: Associazione



Quando più classi sono connesse l'una con l'altra da un punto di vista concettuale, tale connessione viene denominata «associazione». Graficamente, un'associazione viene rappresentata con una linea che connette due classi, con il nome dell'associazione lungo la linea stessa.

Un'associazione può essere più complessa del concetto di connettere una classe ad un'altra.

La «molteplicità» è un tipo speciale di associazione in cui si mostra il numero di oggetti appartenenti ad una classe che interagisce con il numero di oggetti della classe associata. Una classe, in generale, può essere correlata ad una altra nei seguenti modi:

uno ad uno («1 => 1»), uno a molti («1 => *»), uno ad uno o più («1 => 1..*»), uno a zero o uno («1 => 0..1»)

uno ad un intervallo limitato («1 => 2–20»), uno ad un numero esatto n («1 => 5»), uno ad un insieme di scelte (es.: «1 => 5, 8»)

Un'associazione può essere anche riflessiva.

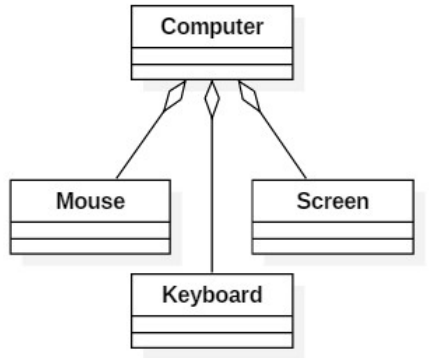
icubed

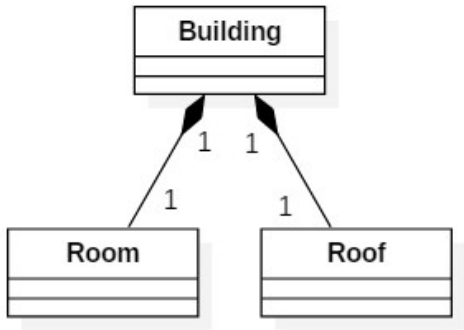
Primitive: Aggregazione e Composizione

Qualche volta una classe può rappresentare il risultato di un insieme di altre classi che la compongono.

Un'aggregazione è rappresentata come una gerarchia in cui l' «intero» si trova in cima e i componenti («parte») al di sotto. Una linea unisce «l'intero» ad un componente con un rombo raffigurato sulla linea stessa vicino all' «intero».

Se si desidera indicare una relazione di aggregazione «più forte», in UML è possibile utilizzare il concetto di «composizione»; essenzialmente si tratta di una dipendenza – esattamente come nel caso dell'aggregazione – che però implica un rapporto di «esistenza» differente tra le parti. Viene rappresentato come una aggregazione con il rombo posto al culmine della linea che è colorato di nero.





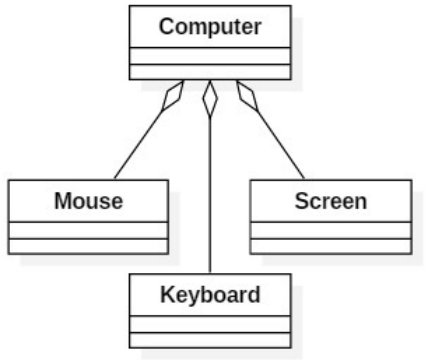
icubed

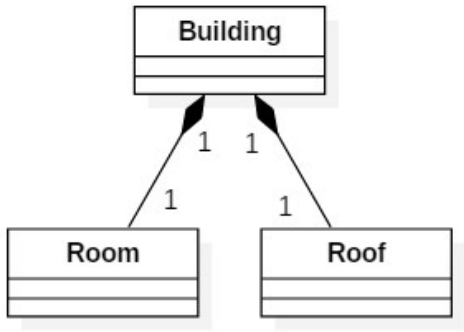
Primitive: Aggregazione e Composizione

In generale la differenza tra una «aggregazione» e una «composizione» è che il tipo di dipendenza che lega le entità in gioco:

in una aggregazione le «parti» possono esistere indipendentemente dalla presenza dell' intero (es. «mouse» e «keyboard» esistono come entità reali indipendentemente dal fatto che esse saranno parte dell'intero «computer»)

In una composizione le parti non possono esistere (o non ha senso che esistano in un modo reale) senza la presenza di un intero che di fatto racchiude la parti. Ad esempio non ha senso che una «room» esista senza un «building» che la racchiuda.





icubed

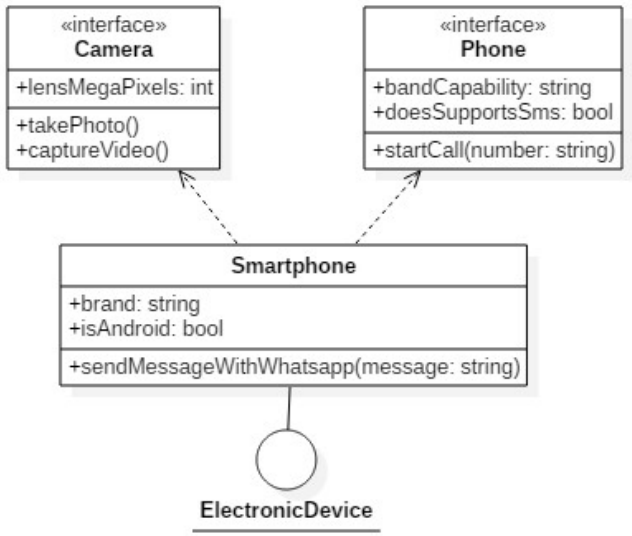
Primitive: Interfacce e Realizzazioni

E’ possibile che alcune classi non siano solo correlate ad una particolare classe padre e che il loro comportamento possa essere afferente a diverse forme o diversi «contratti».

In questi casi è possibile definire un insieme di operazioni e attributi che specifica alcuni aspetti del comportamento di una classe, l’interfaccia

Per modellare un’interfaccia si utilizza lo stesso modo utilizzato per modellare una classe, con un rettangolo. La differenza consiste nel fatto che un’interfaccia è marcata con il testo <<interface>> che precede il nome della stessa.

Esiste una notazione «compatta» per evidenziare che una particolare classe implementa una interfaccia, utilizzando un piccolo cerchio che riporta il nome dell’interfaccia, e una connessione diretta alla classe implementante. La notazione compatta è molto comoda quando il diagramma di classi diventa complesso



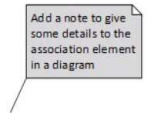
icubed

Primitive: Annotazioni e Stereotipi

UML è un linguaggio in continua evoluzione. Dalla primissima versione universalmente accettata (v1.1 risalente al 1997) è cambiato molto, arricchendosi di nuove funzionalità, diagrammi ma soprattutto «estensioni».

Le «**annotations**» possono essere applicate in qualunque tipo di diagramma, rappresentandole con un rettangolo con l'angolo piegato (quasi fossero un «post-it») e una linea che le collega all'elemento su cui si vuole applicare la nota o la precisazione.

Gli «**stereotypes**» o stereotipi rappresentati dalle doppie parantesi angolari possono essere usati in ogni tipo di diagramma per definire un «prototipo» di un artefatto, cioè un modello logico che astrae una vera implementazione ma che esprime il significato semantico della stessa. Esistono «interface» (Class Diagram) e «include» (Use Case), ne ma possono essere creati di propri per definire una generalizzazione del concetto.



Annotations / Notes

<<Interface>>

Stereotype

Web Server

Iconic Stereotype

icubed

Best practices per il disegno dei diagrammi

E’ fondamentale seguire 3 semplici regole quando si disegna un diagramma UML, qualunque sia il suo scopo e qualunque sia il sistema che andiamo a descrivere:

- Leggibilità
- Focus
- Precisione

icubed

Obiettivi dell'uso dei diagrammi UML

Visualizzazione

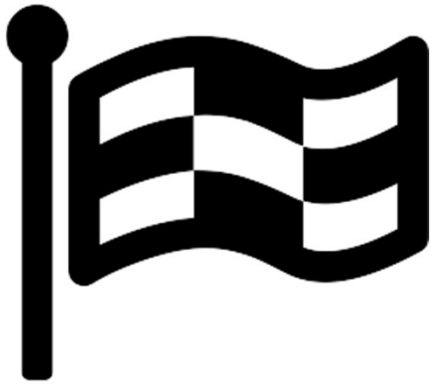
- Si tratta di un linguaggio «visuale», e quindi l'obiettivo principale è quello di fornire informazioni utilizzando la capacità dell'essere umano di ricordare (e comprendere) meglio un disegno piuttosto che un testo scritto.

Specifica:

- Si vuole specificare le «parti» di cui è costituito un sistema e come esse interagiscono per adempiere il workflow. E' quindi importante la precisione e l'utilizzo dei corretti artefatti

Documentazione:

- E' un aspetto spesso sottovalutato nel ciclo di vita di un sistema perché è molto difficile tenere allineata la documentazione con il software in evoluzione. Documentare usando UML è molto più semplice perché il dettaglio implementativo non è così netto.



icubed

Structural Diagrams

Aiutano a definire la struttura e l'architettura complessiva del sistema che si vuole realizzare.

Sono considerati «diagrammi strutturali» i seguenti diagrammi UML:

Class Diagram

Component Diagram

Deployment Diagram

Package Diagram

Object Diagram

Composite Structure Diagram

icubed

Class diagram: scenario reale (1)

Dato il seguente scenario:

Si desidera realizzare un sistema che gestisca un catalogo di prodotti

Ciascun prodotto è caratterizzato da un codice, un nome, un flag di visibilità (visibile, nascosto), una descrizione e una categoria di appartenenza.

Una categoria è il contenitore di uno o più prodotti

Ciascuna categoria è caratterizzata da un codice, un nome e un flag che definisce se si tratta di una categoria di base alimentare o meno

Si desidera mostrare i prodotti sia in una pagina specifica con i suo dettaglio, sia in una pagina con una lista di tutti i prodotti di una certa categoria

Ogni elemento convolto nel processo del sistema deve essere passibile delle classiche operazioni di C.R.U.D. (Create, Read, Update e Delete) su una base dati generica

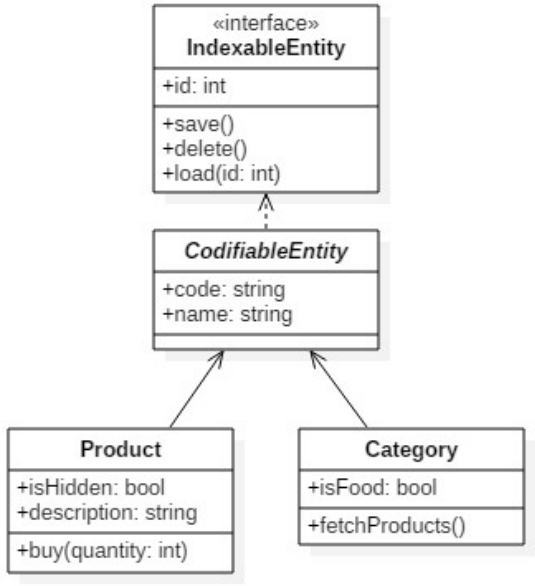
Un elemento deve essere acquistabile specificando la quantità di unità

icubed

Class diagram: scenario reale (2)

Un diagramma di classe ci permette di produrre una rappresentazione schematica delle entità presenti nel modello dati, comprendente tutte le specifiche funzionale che sono state espresse dal committente.

Un'analisi approfondita del dominio dati – cercando di spingere sulla leva delle «analogie» tra gli elementi – permette di identificare delle astrazioni (es. «CodifiableEntity» come classe padre di «Product» e «Category») e una generica interfaccia «IndexableEntity» che espone alcune delle funzionalità di CRUD richieste.



Demo

L'applicazione da progettare riguarda le informazioni sulle contravvenzioni fatte in un comune.

Di ogni **contravvenzione** interessa **il veicolo** a cui è stata effettuata**, il vigile** che l'ha fatta, il numero di verbale e il luogo in cui è stata fatta.

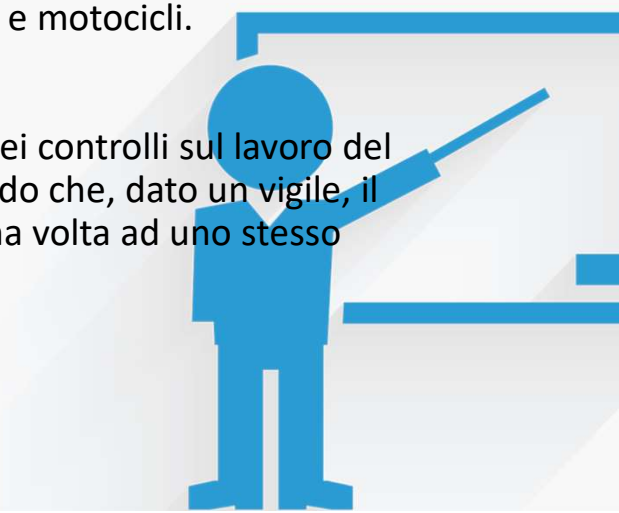
Di ogni vigile interessa il nome, il cognome ed il numero di matricola.

Di ogni veicolo interessa la targa.

Esistono solamente due categorie di veicoli, che sono fra loro disgiunte: automobili e motocicli.

Delle automobili interessa la potenza in kilowatt, dei motocicli il numero di telaio.

Sappiamo che il comune vuole effettuare, come cliente della nostra applicazione, dei controlli sul lavoro del proprio personale. In particolare, si deve progettare il diagramma delle classi in modo che, dato un vigile, il comune possa controllare se un vigile ha elevato una contravvenzione per più di una volta ad uno stesso veicolo.



Esercitazione n.3

Le officine riparano i veicoli.

Di ogni **officina** interessano nome, indirizzo, numero dipendenti, **dipendenti** (con l'informazione su quanti anni di servizio), e **direttore**.

Dei dipendenti e dei direttori interessano: codice fiscale, l'indirizzo, numero telefono.

Dei direttori interessa anche l'età.

Delle **riparazioni** interessano: codice, veicolo, ora e data di accettazione, e ora e data di riconsegna.

Dei **veicoli** interessano: modello, tipo, targa, anno di immatricolazione, e **proprietario**.

Dei proprietari interessa codice fiscale, indirizzo, numero telefono.

icubed

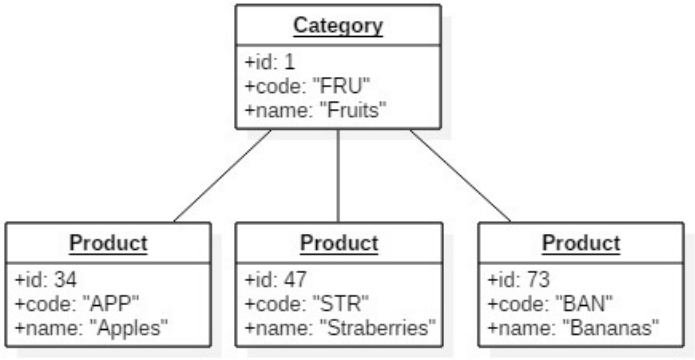
Object diagram: istanze di classi

Il “Diagramma degli oggetti” è un grafo che include oggetti e valori stato. Un diagramma di oggetti statici rappresenta le istanze delle classi descritte nel “Class Diagram”

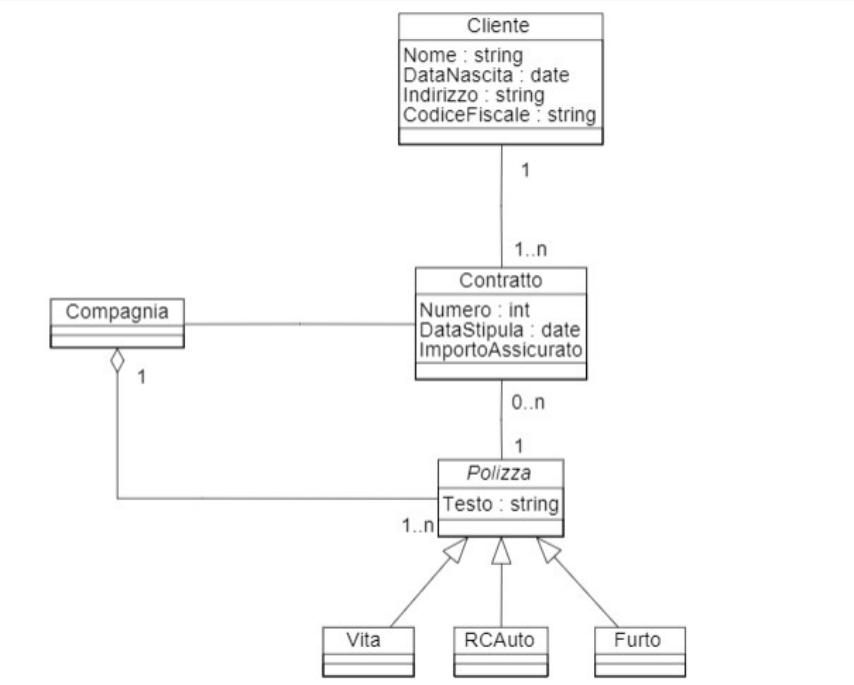
L’utilizzo del diagramma di oggetti è abbastanza limitato ed è solitamente sfruttato per mostrare esempi del funzionamento del software in termini puramente statici.

Mostra le relazioni tra istanza effettive e reali delle classi, come le stesse sono correlate tra loro.

Spesso accompagna gli «Use Case» per aiutare gli utenti i business a spiegare come un particolare comportamento del sistema influenza gli elementi in gioco.



Esercitazione n.4



A partire dal diagramma delle classi.

Riportare il codice delle classi corrispondente.

icubed