ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dispensa di *"Ingegneria Del Software (modulo 2)"*

A.A. 2024/25

Ivan De Simone

UML - parte 1

Introduzione all'Object Orientation

Perché l'OO è popolare?
La speranza è che incrementi la produttività.
È un modo naturale di strutturare il mondo, tramite:
- Oggetti
- Messaggi
- Responsabilità

Serve capire quali entità indipendenti prendono parte alla computazione.
Grazie al linguaggio naturale, che è ambiguo, possiamo non esprimere ciò che non sappiamo nella formulazione di un problema.

Che cos'è lo sviluppo di software object-oriented?
Un modo per vedere il mondo dell'applicazione.
Un modo per descrivere un modello dell'applicazione.
Una metodologia comprensiva che permette di sviluppare un sistema software usando concetti simili all'interno dell'intero processo di sviluppo.

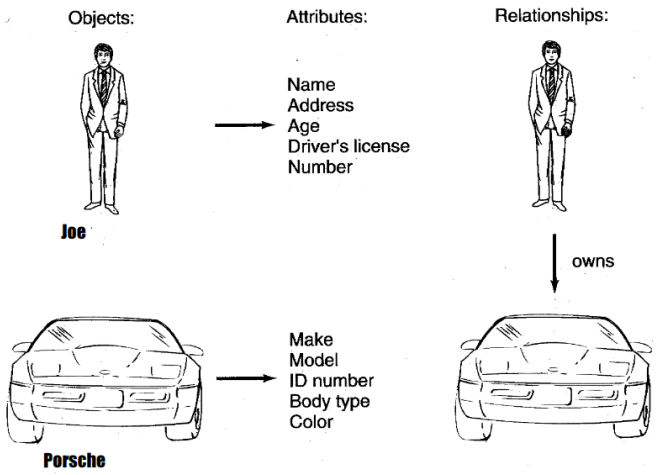
Lo **sviluppo di sistemi object oriented** intende raggiungere un'alta qualità in termini di:
- Information hiding
- Astrazione
- Modularizzazione
- Riuso

Un approccio object oriented, più o meno, forza lo sviluppatore software ad applicare questi concetti.
Non sono ingredienti totalizzanti, sono features (cioè non sono questi a definire un linguaggio OO).
L'Object Orientation è caratterizzato dall'elaborazione come interazione tra elementi indipendenti.

Tra la fine degli anni '80 e l'inizio degli anni '90 vennero sviluppate numerose metodologie OO, caratterizzate da differenti notazioni e/o processi.
Gli approcci principali erano Booch, Rumbaugh e Jacobson, riuniti nel linguaggio UML.

L'idea chiave è di rappresentare il mondo in termini di **oggetti che interagiscono tra loro**, e usare questa rappresentazione in tutto il ciclo di vita delle fasi di sviluppo:
- OO Concept Modeling
- OO Analysis
- OO Design
- OO Programming

Un semplice modello OO



Concetti chiave

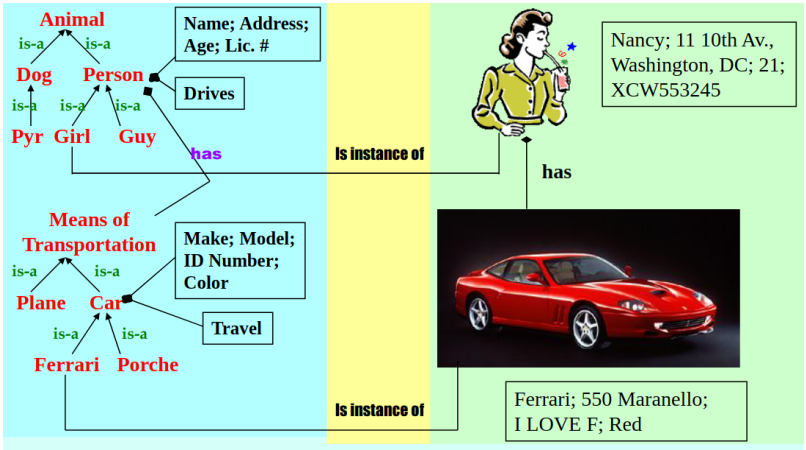
Classi e gerarchie di classi
- Attributi
- Metodi
- Ereditarietà
- Relazioni con altre classi

Oggetti: istanze di classi
- Attributi con valori assegnati
- Relazioni istanziate

Messaggi e metodi per rispondere ad un messaggio

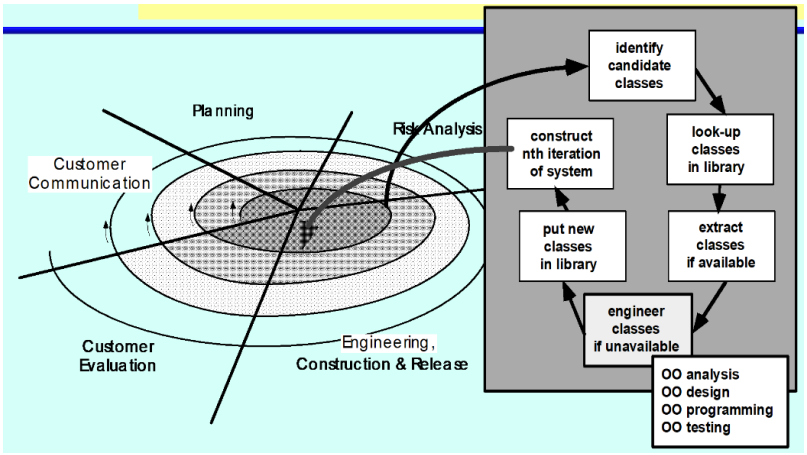
Un modello OO

Cerco di rappresentare ciò che viene presentato nel mondo

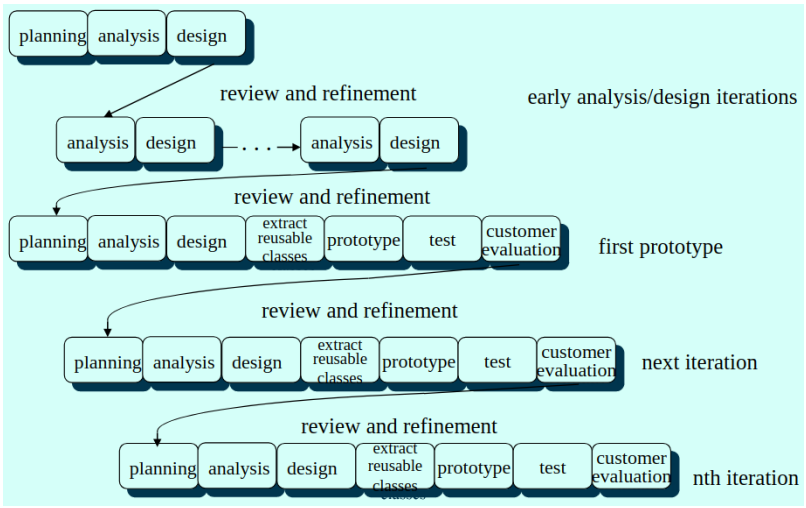


Nancy è una Girl, con una serie di attributi che sono ereditati da Person. Identifichiamo quella specifica ragazza con la sequenza di attributi: *Nancy; 11 10th, Av. Washington, DC; 21; XCW553245*.
La 550 Maranello è un'istanza di Ferrari, con una serie di attributi che sono ereditati da Car.
Identifichiamo quella specifica Ferrari con la sequenza di attributi: *Ferrari; 550 Maranello; I LOVE F; Red*.
Quando si eredita, alcuni attributi possono diventare fissi.

Il modello di processo OO



Tipico processo per un progetto Object-Oriented

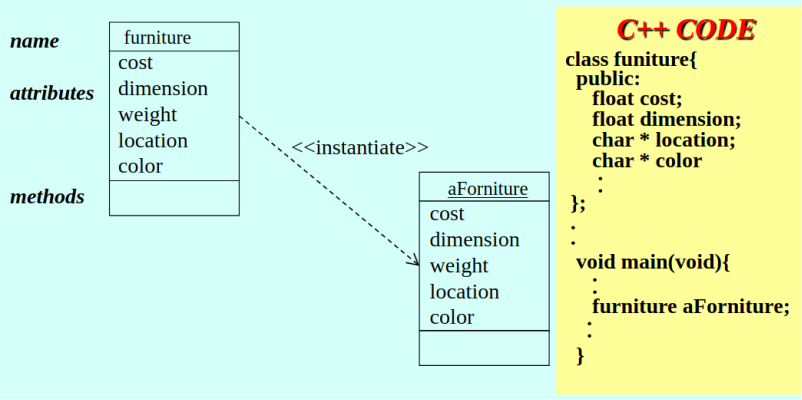


Alcuni termini

Classi

Una **classe** è una collezione di oggetti simili. Spesso è definita come:
- Template
- Descrizione generalizzata
- Pattern
- "Planimetria" (*blueprint*), descrivente una collezione di elementi simili

Una classe identifica **proprietà (attributi)** che appartengono a tutti gli oggetti della classe e **comportamenti (operazioni)** di tutti gli oggetti della classe.
Una volta che è stata definita una classe di elementi, si può definire una specifica istanza di quella classe.

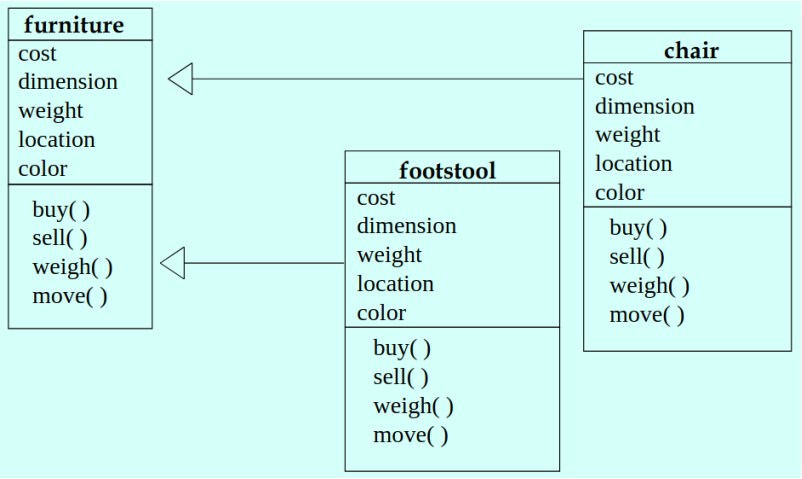


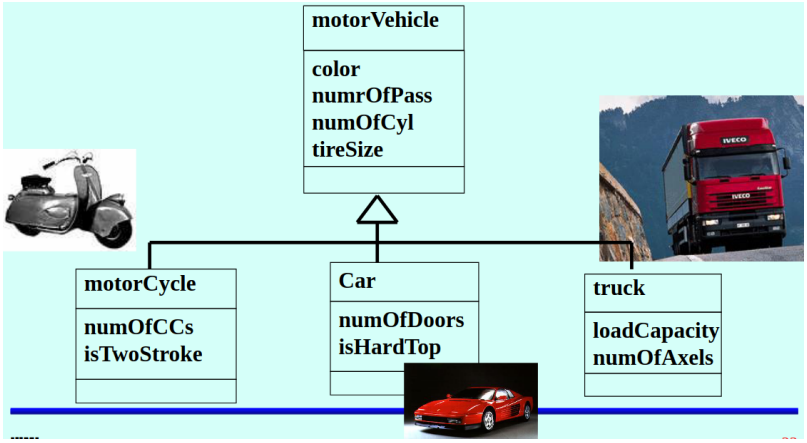
Operazioni (servizi)

Una **procedura eseguibile** che è incapsulata in una classe ed è progettata per operare su uno o più attributi definiti come parte della classe.
Spesso i libri di testo dicono che un'operazione è invocata tramite message passing.
Il termine "operazione" ha molti sinonimi: servizio, function entry (concurrent Pascal), member function (C++), metodo, ...

Ereditarietà

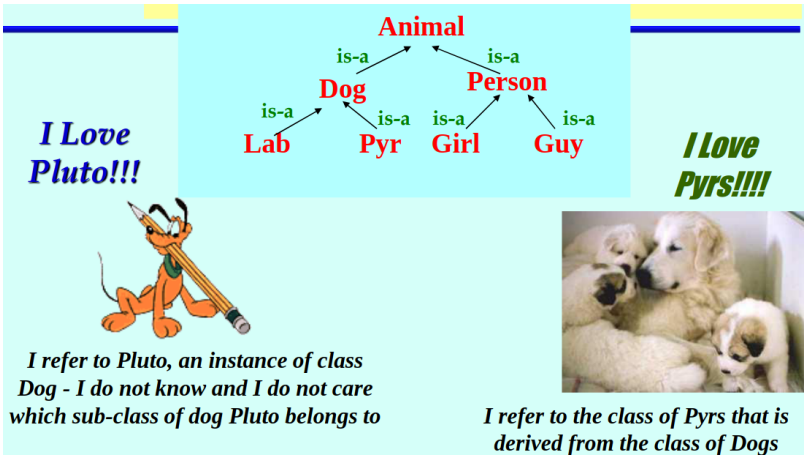
L'**ereditarietà** è l'abilità di **definire classi che sono estensioni** di altre classi con attributi e metodi nuovi o specializzati.
Per esempio la classe Dog eredita dalla classe Animal, significando che Dog ha (eredita) tutti gli attributi e i metodi di Animal, e può ridefinire alcuni di essi e aggiungerne di nuovi.
Si dice: Dog "è un" Animal, Dog "estende" Animal, la classe Animal "contiene" la classe Dog, Animal "generalizza" Dog, ...



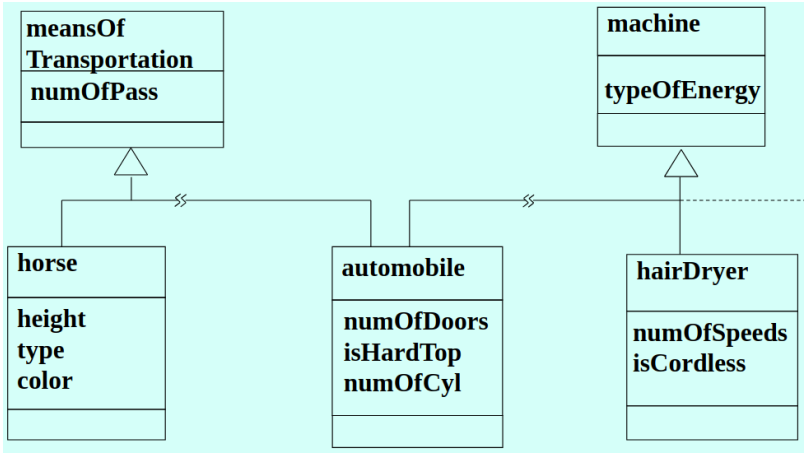


Ereditarietà NON È istanziazione

Istanziare vuol dire che esiste un elemento vivo di quella classe.
Un'istanza può essere a qualunque livello dell'albero gerarchico.



Ereditarietà multipla



Polimorfismo

Il **polimorfismo** è l'abilità di usare lo stesso nome per metodi che effettuano operazioni dello "stesso tipo" su oggetti diversi.
In matematica ci sono numerosi esempi di polimorfismo: il + è usato per sommare qualunque tipo di numero (Naturale, Intero, Reale, Complesso, ...), ma anche vettori e matrici.
Il polimorfismo aiuta a gestire grandi insiemi con operazioni simili, senza il bisogno di ricordare nomi bizzarri (es. `printf`, `fprintf`, `sprintf`, ...).

Varie forme di polimorfismo

- **Ad-hoc**, anche chiamato "**overloading**": numerose funzioni sono definite con lo stesso nome ma diversi parametri.
  Es. `print(file)`, `print(string)`, `print(number)`.
- **Generico**: un template generale definisce una struttura comune ad un insieme di classi/funzioni.
  Es. `template <class A> void swap(A &x, A &y) { A t=x; x=y; y=t; }`.
- **Ereditario** o **overriding**: funzioni sono ridefinite nella sottoclasse.

```
void DrawCorrect(Graph &t){
     t.draw( );
}
```

```
class Graph{//base class
public: virtual void draw( ){
      cout<<"in base\n"; }
};

class LineGraph : public Graph
public: virtual void draw( ){
      cout<<"in LineGraph\n"; }
};

class PieChart : public Graph{
public:virtual void draw( ){
      cout<<"in piechart\n"; }
};
```

```
void main(void){
   LineGraph lg;
   PieChart pc;
   DrawCorrect(lg);
   DrawCorrect(pc);
   Graph *list[10];
   int i;
   for(i=1;i<10;i++)
     list[i]= … ;
   for(i=1;i<10;i++)
     DrawCorrect(*list[i]);
}
```

Il polimorfismo in Java è possibile con overloading, overriding o generici.
In C non esiste ma è simulabile con i puntatori void o con le macro.

```c
// esempio di macro swap in C
// generici simulati passando alla macro il tipo che voglio usare (T)
#define swapM(x,y,T) {\
  T t = x;\
  x = y;\
  y = t;\
}
```

Problemi! Se volessimo definire una moltiplicazione generica in C:

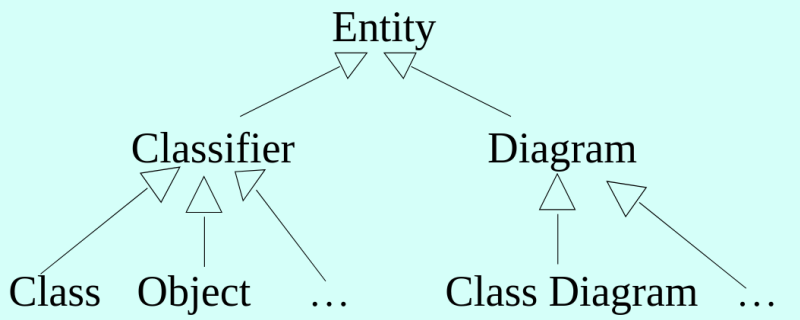
```c
#define mult(a, b) a*b

int a = 7;
int b = 10;
int c = mult(a+1, 1+b);
// c = a + 1 * 1 + b  -->  quindi alla fine ottengo a + b
```

In conclusione: possiamo definire in C macro per simulare i generici ma dobbiamo tenere presente che sono soggette a delle macro sostituzioni.

Alcuni dettagli di UML

Unified Modeling Language prova ad integrare gli approcci più vecchi. Sviluppato da Rational come CASE tool (strumento software che supporta lo sviluppo e la progettazione di sistemi orientati agli oggetti), assumendo Booch, Rumbaugh e Jacobsen. Standardizzato da OMG (Object Management Group). Supportato da quasi tutti gli OO CASE tool, con alcune limitazioni. Attualmente è alla versione 1.3.

UML ha molte entità:



Classificatori UML

UML parte dal modellamento delle classi, ma nel corso degli anni ha sviluppato come obiettivo quello di poter modellare qualunque cosa. Esistono meta-modelli per modellare facendo uso di modelli. Alcuni esempi di classificatori (UML classifiers) sono:

- Class
- Interface
- Datatype
- Component
- Node
- Use case
- Subsystem
- ...

Diagrammi UML

UML in modo standard ha almeno 9 tipi di diagrammi. Sarebbero teoricamente infiniti, data la possibilità di creare dei meta-modelli, ma dal punto di vista pratico 9 diagrammi bastano e avanzano. Essi sono:
Diagrammi strutturali:

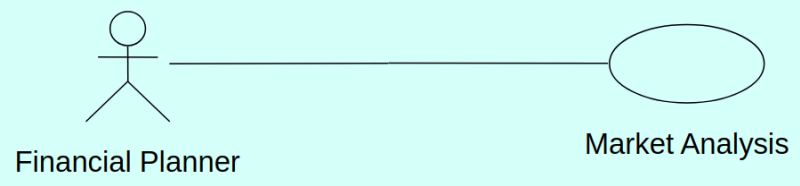
- Class diagram
- Object diagram
- Component diagram
- Deployment diagram

Diagrammi comportamentali:

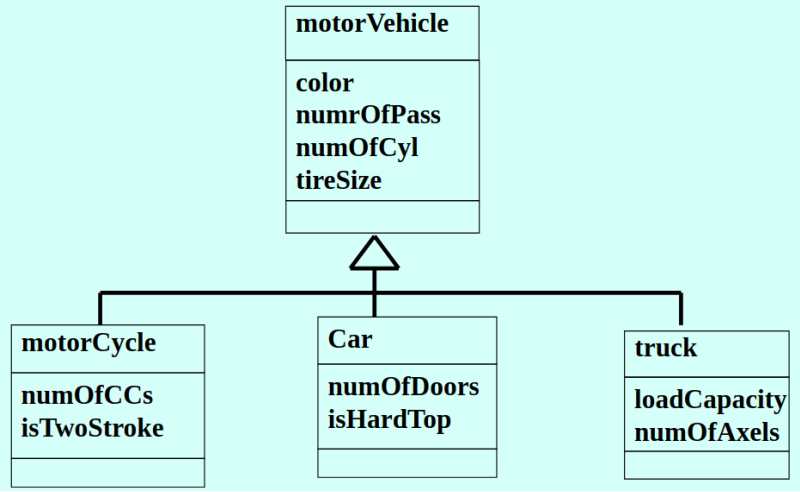
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Use case diagrams

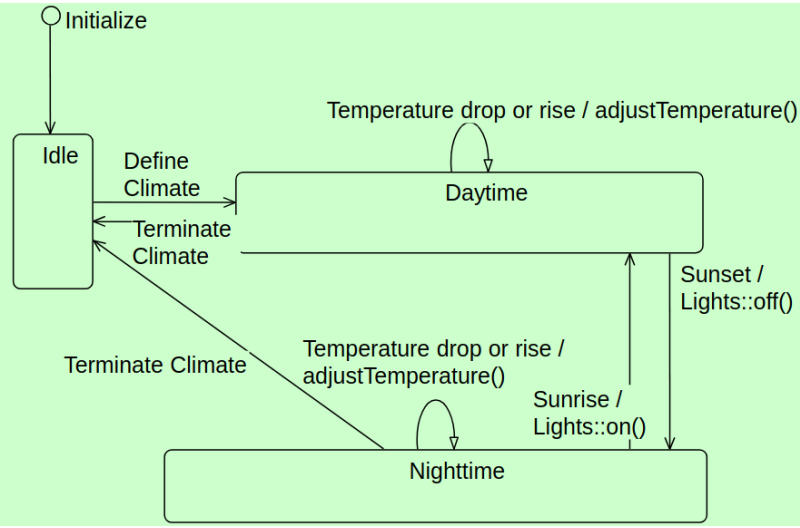
I diagrammi di use case definiscono i requisiti nelle fasi iniziali di analisi.



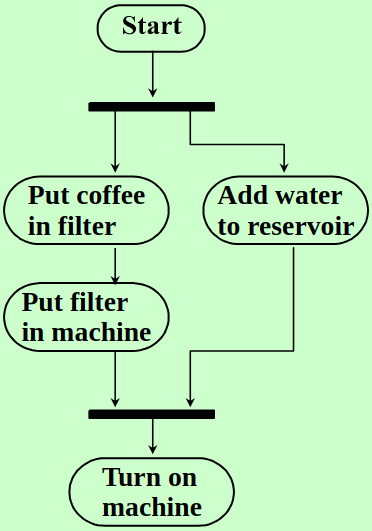
Class diagrams (motor vehicle)



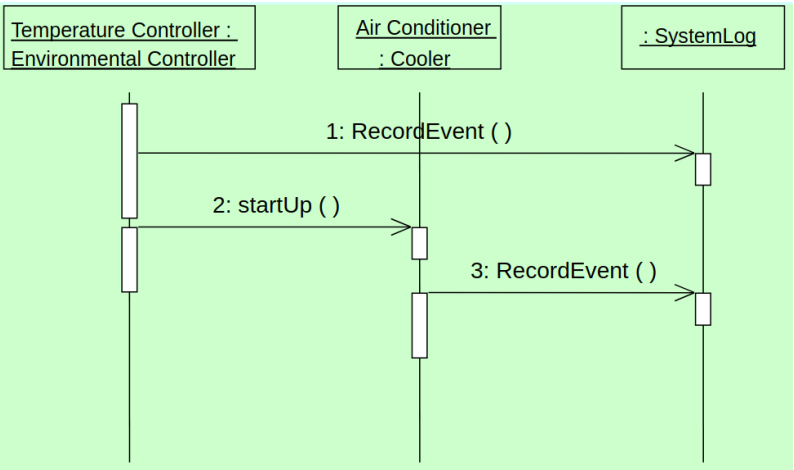
Statechart diagrams (air cond.)



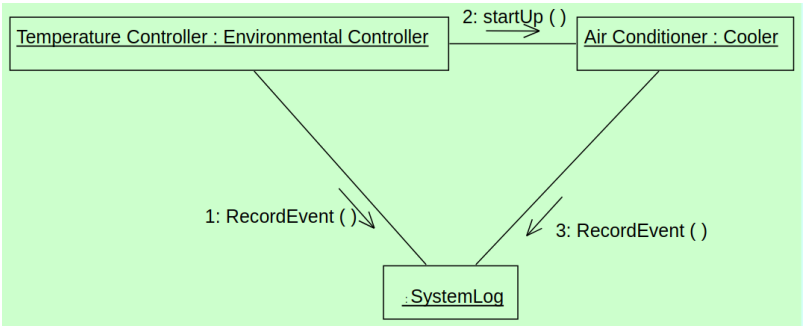
Activity diagrams (coffee machine)



Sequence diagrams (air cond.)



Collaboration diagrams (air cond.)



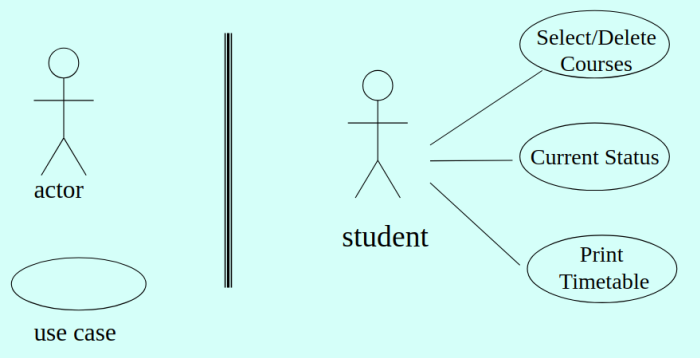
Modellazione concettuale Object Oriented (OOCM)

Obiettivi
- Analisi del contesto: comprendere il contesto operazionale del sistema.
- Analisi dei requisiti: capire i requisiti effettivi del sistema.
A volte ci si riferisce a questa fase come sola "analisi dei requisiti", ma si intendono entrambe le attività.
Per poter capire cosa vuole il cliente devo innanzitutto analizzare il contesto in cui si trova, per poi procedere all'analisi dei requisiti vera e propria. Non sono in grado di strutturare tutto a priori in un colpo solo, quindi cerco di rappresentare il mondo come una narrazione. La potenza della narrazione è l'ambiguità, che ci permette di nascondere quello che ancora non sappiamo.

Use cases per la modellazione concettuale OO

Una descrizione di caso d'uso (Use Case Description) è uno scenario che descrive una modalità di utilizzo per un sistema.
Una Use Case Description include:
> Un **diagramma**, composto da **attori**, che sono delle entità rappresentanti persone o dispositivi che utilizzano il nostro sistema, e **casi d'uso**, che sono possibilità di uso del sistema.
> Una **descrizione testuale**, che pone in sequenza le attività.
Quindi io racconto il mio sistema evidenziando la presenza di attori, e lo faccio in casi di uso concreti che descrivo tramite descrizioni testuali.

Esempio: *uno studente che vuole controllare i corsi da seguire.* Lo studente è l'attore, ed ha tre possibili casi d'uso (narrazioni).



Use case

Uno **use case** rappresenta una tipica interazione tra gli attori ed il sistema. Deve essere centrato sul descrivere un'attività che sia utile all'utente finale e che sia atomica, ovvero non scomponibile. Quindi uno use case descrive uno scenario, ad esempio come il sistema è usato.
Es. *Un editor di testo: rendi del testo in grassetto, crea un indice, cancella una parola.*

Fatti da considerare nella scrittura di uno use case
Quanto sarà grande il sistema? Bisogna decidere la granularità con cui descrivere lo use case.
Spesso lo use case cattura una funzione che sia visibile all'utente.
Il caso d'uso non è una qualunque forma di interazione: descrive sempre una funzionalità richiesta per il sistema che ha come scopo l'ottenimento di un valore per l'utente finale.

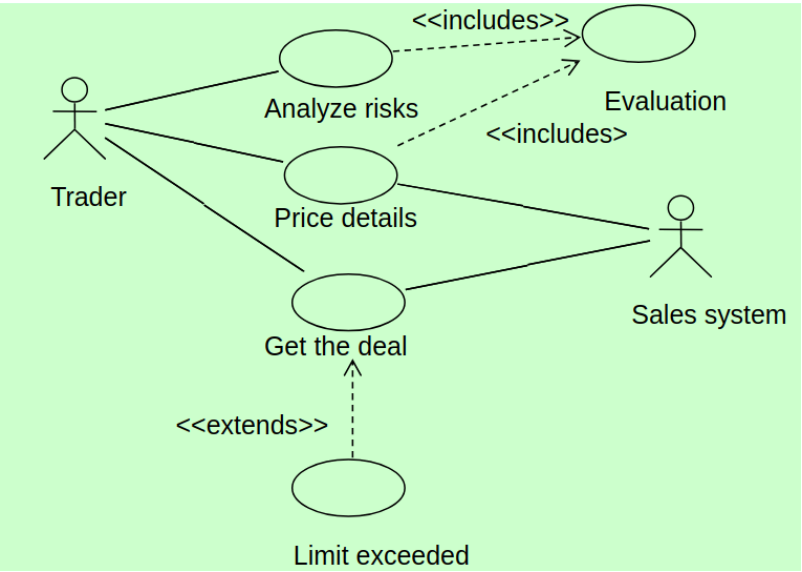
Quando e come
La prima cosa da fare nello sviluppo è trascrivere le narrazioni e catturare il contesto in cui ci si trova. Si passa poi ai casi d'uso, ovvero ogni singola cosa che il cliente vuole fare con il sistema. La prima scrittura dello use case non deve essere necessariamente precisa, verrà approfondita durante lo sviluppo se serve. L'importante è assegnare un nome ed una breve descrizione.

Sviluppare uno use case
Esempi di domande da porsi quando si approccia un problema:

- Quali sono le principali attività o funzioni che sono effettuate dall'attore?
- Quali informazioni del sistema l'attore acquisirà, produrrà o cambierà?
- Sarà necessario che l'attore informi il sistema riguardo a cambiamenti nell'ambiente esterno?
- Quali informazioni desidera l'attore dal sistema?
- L'attore desidera essere informato riguardo cambiamenti inaspettati?

Esempio: NYSE

Supponiamo di voler modellare il NYSE. Ci sono traders e sistemi di vendita...
Use case diagram

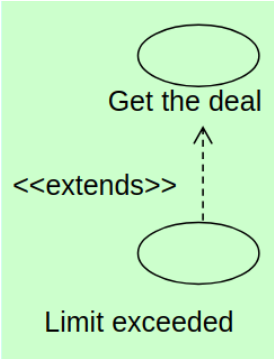


Attori

Un **attore** è un *ruolo* che l'utente assume rispetto al sistema.
Gli attori svolgono i casi d'uso -> definire prima gli attori, poi i loro casi d'uso.
Gli attori non devono necessariamente essere umani.
Gli attori devono ottenere valore dagli use case in cui partecipano.

Relazione extends

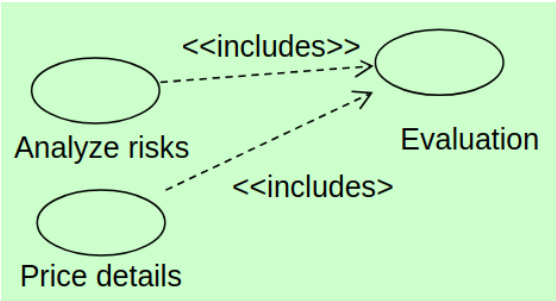
Extends: uno use case è simile ad un altro ma fa qualcosa in più.



Catturare per primo lo use case semplice, nella sua forma normale.
Per ogni step chiedersi:
- cosa potrebbe andare storto?
- come potrebbe funzionare diversamente?
Segnare ogni variazione come un'estensione dello use case semplice.
L'extends mi serve per estensioni che non hanno valore singolarmente, ma solo congiunte allo use case base.

Relazione includes

Includes: una parte del comportamento è simile in più di uno use case.



Evita il copia e incolla di parti delle descrizioni dei casi d'uso.

Confronto extends-includes

Le due relazioni hanno diversi scopi.
extends:
- lo stesso attore svolge lo use case e tutte le estensioni.
- l'attore è collegato allo use case base.
includes:
- spesso non c'è un attore associato allo use case comune.
- diversi attori chiamano i casi d'uso possibili.

Descrizione testuale

Descrizione generica, scritta step-by-step, delle interazioni tra l'attore (o gli attori) ed un caso d'uso.
Le descrizioni devono essere brevi, chiare e precise.

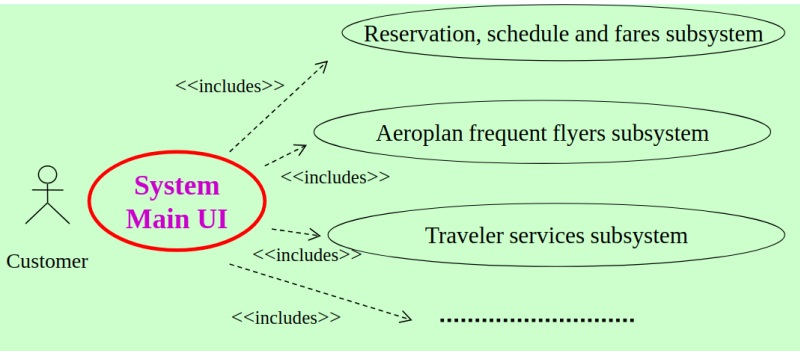
Esempio di descrizione

Use case: Get the deal

1. Enter the user name & bank account
2. Check that they are valid
3. Enter number of shares to buy & share ID
4. Determine price
5. Check limit
6. Send order to NYSE
7. Store confirmation number

Nota che
Non tutti i casi d'uso sono stati listati (ogni diagramma fornisce una vista parziale).
La relazione di include supporta la fattorizzazione delle specifiche comuni del sistema.

Attenzione al killer!



Descrizione testuale (solo per lo use case frequent flyer)
The user enters the subsystem to gain more information about its frequent flyer status. Inside the subsystem, the user can access (a) general information about the frequent flyer program -the reward schema, how to enroll, how to get miles with partner companies, and (b) specific information on her/his status, such as the miles earned, the status level. The user can also update her/his address.
Nota che questo è un formato molto differente!

Argomenti non trattati

- Generalizzazione negli use cases
- Generalizzazione negli attori
- Presenza dei punti di estensione

Esercizi proposti

Taxi drivers

Develop an OO Concept Model for a system supporting the reservation and scheduling for taxi drivers.

OOCM for Stocks Trading Service

By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the user to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a particular stock is no longer available.

OOCM for Network Printing Service

There is a super high-resolution color laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc...). In addition, users can monitor their own print jobs and delay or delete jobs as they see fit.
To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.

OOCM for Component Brokerage System

This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.
Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.
Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.

OOCM for Bug Tracking System

Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.

Analisi Object Oriented (OOA)

Il nostro punto di partenza è che abbiamo raccolto i casi d'uso del sistema (ovviamente non tutti). Ora dobbiamo cercare di metterli in ordine in una qualche forma per capire il sistema da sviluppare.

OOA: una vista generica

1. Estrarre delle possibili classi candidate
2. Stabilire relazioni base tra le classi
3. Definire una gerarchia di classi
4. Identificare gli attributi per ogni classe
5. Specificare i metodi che servono gli attributi
6. Indicare come le classi e gli oggetti sono relazionati
7. Costruire un modello comportamentale
8. Ripetere i primi cinque step

Estrazione delle classi

Regola empirica: i nomi presenti nelle narrazioni sono candidati per le classi.

Es. *Get the deal*:

1. Enter the **user** name & **bank account**
2. Check that they are valid
3. Enter number of shares to buy & **share ID**
4. Determine price
5. Check limit
6. Send order to **NYSE**
7. Store **confirmation number**

Altro es. *Lufthansa*:
The **user** enters the subsystem to gain more information about its **frequent flyer status**. Inside the subsystem, the user can access (a) general information about the **frequent flyer program** -the **reward schema**, how to enroll, how to get **miles** with **partner companies**, and (b) specific information on her/his **status**, such as the **miles earned**, the **status level**. The user can also update her/his **address**.

Class diagram

È centrale per la modellazione Object Oriented. Mostra la struttura statica del sistema:
- I tipi degli oggetti
- Le relazioni (associazione, sottotipi, dipendenza)

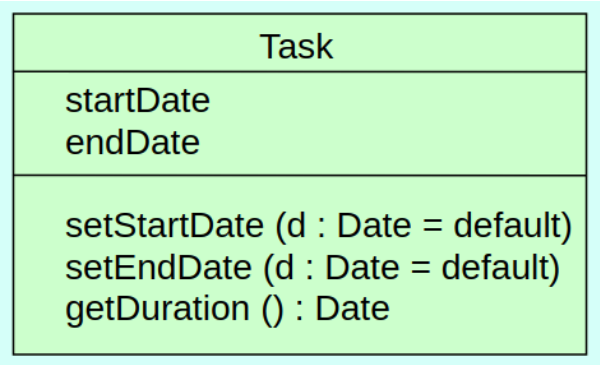
Abbiamo 3 prospettive

- **Concettuale (OOA)**: mostra concetti del dominio, è indipendente dall'implementazione.
- **Specifica (OOD)**: struttura generale del sistema, definisce le interfacce del software.
- **Implementazione (OOP)**: ha i dettagli dell'implementazione, spesso è l'unica usata.

I nomi sono messi in relazione tramite il diagramma a classi, che è fondamentale in quanto mi serve per concettualizzare il modello del mio sistema: tramite questo razionalizzo la storia che mi racconta il cliente.
Noi non siamo esperti del dominio su cui lavorerà il software, operiamo sulla conoscenza degli altri. Il primo scopo del diagramma a classi è formalizzare la conoscenza che noi abbiamo acquisito dagli altri.
In soldoni il class diagram è come una mappa concettuale. Questo diagramma ci sarà utile in fase di progettazione: il linguaggio rimarrà quello, riducendo la formazione di errori (non dovendo tradurre).

Classe

Una **classe** è un insieme di oggetti.
Definisce un nome, degli attributi e delle operazioni.



Classe vs Tipo

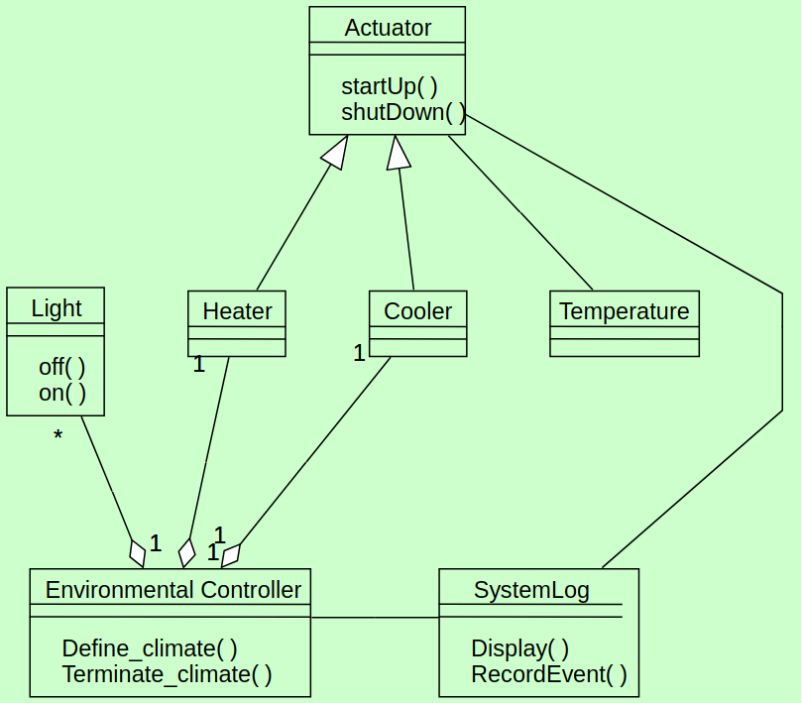
Certe volte si sovrappone il concetto di classe con quello di tipo: questo è vero quando si parla di implementazione, la differenza è molto sottile.
Un **Tipo** è un protocollo compreso da un oggetto, definisce un insieme di operazioni che possono essere usate. Una **Classe** è un costrutto orientato all'implementazione, implementa uno o più tipi.
In Java un tipo può essere visto come un'interfaccia, in C++ come una classe astratta.

Associazione

Le classi definiscono delle relazioni tra istanze di classe, che sono chiamate **associazioni**.
Es. *Uno studente è iscritto ad un corso*, *un professore insegna il corso*.

Esempio di class diagram

Torniamo all'esempio del sistema di aria condizionata

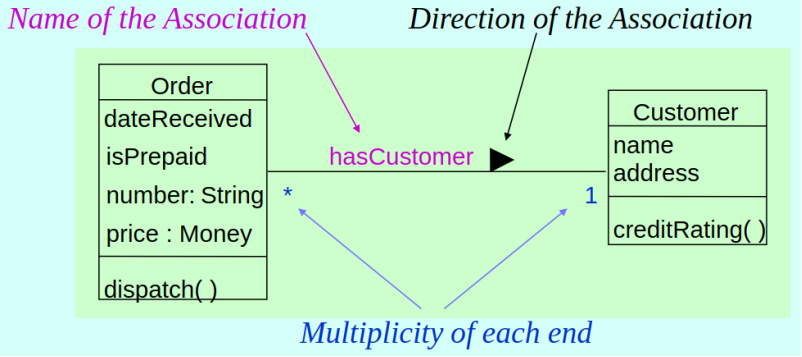


Classi e diagrammi

Una classe può essere parte di numerosi diagrammi.
I diagrammi dovrebbero illustrare aspetti specifici:
- non troppe classi,
- non troppe associazioni,
- nascondere attributi e/o operazioni irrilevanti.
Sono necessarie numerose iterazioni per creare un diagramma come si deve.

Associazione: relazione tra classi

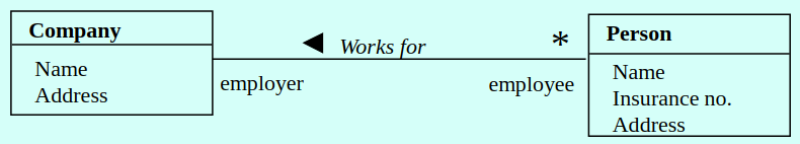
Un ordine arriva da un solo cliente, un cliente può fare numerosi ordini.



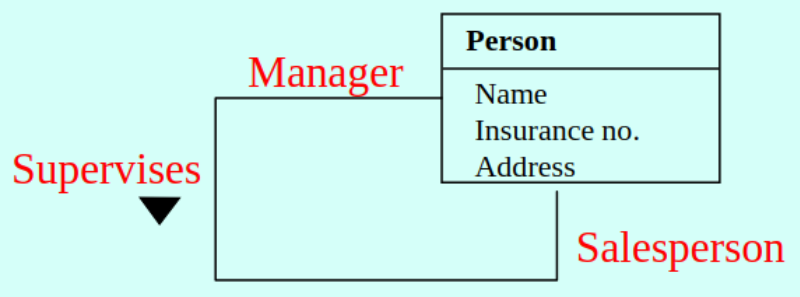
I nomi devono essere esplicativi, evitare quelli che non significano nulla: *associated_with*, *has*, *is_related_to*...
Il nome di un'associazione è spesso un verbo: *has_part*, *is_contained_in*...

Ruoli

Un **ruolo** identifica un estremo di un'associazione. Un'associazione ha due ruoli, che possono essere esplicitamente etichettati oppure denominati implicitamente come la classe target. Vicino alla classe metto il ruolo che essa assume nell'associazione, questo diventerà un attributo dell'altra classe nel codice.



Es. *Person avrà come campo employer* in questo caso.
Il nome del ruolo è obbligatorio per le associazioni tra oggetti della stessa classe.



Molteplicità

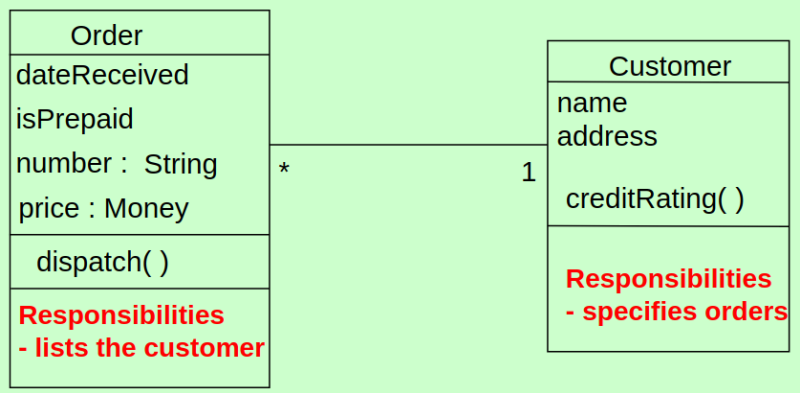
La molteplicità indica quanti oggetti possono partecipare nella relazione.

Ci sono diverse possibilità per esprimerla (se non è espressa è 1):

- *****: da 0 ad infinito
- **1**: solo 1
- **1..100**: da 1 a 100 (inclusi)
- **2,4,5**: 2, 4 oppure 5
- ***..5**: al massimo 5
- **2..***: da 2 ad infinito

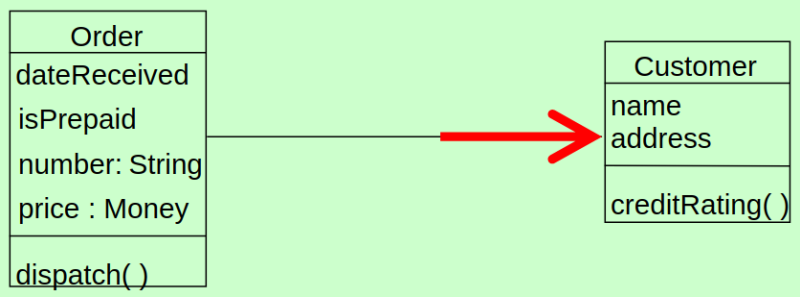
Responsabilità

Il cliente specifica gli ordini, gli ordini listano il cliente.



Navigabilità

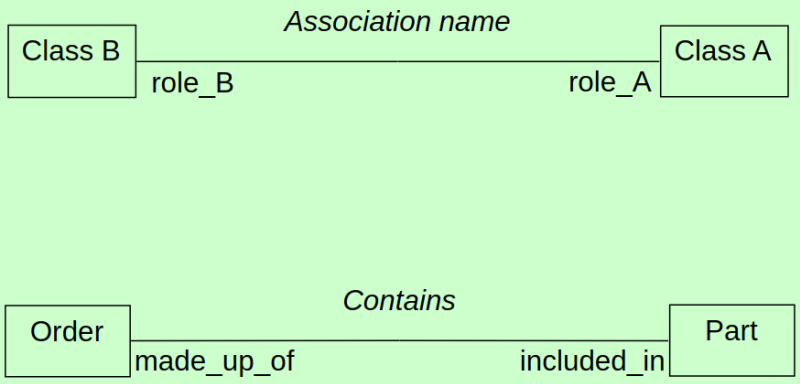
La direzione dell'associazione è indicata dalla freccia.



L'ordine deve essere in grado di determinare il cliente. Il cliente non sa tutti gli ordini.
Associazione bidirezionale: navigabilità in entrambe le direzioni, richiede i ruoli per un'identificazione corretta.

Riassunto

Notazione base per le associazioni

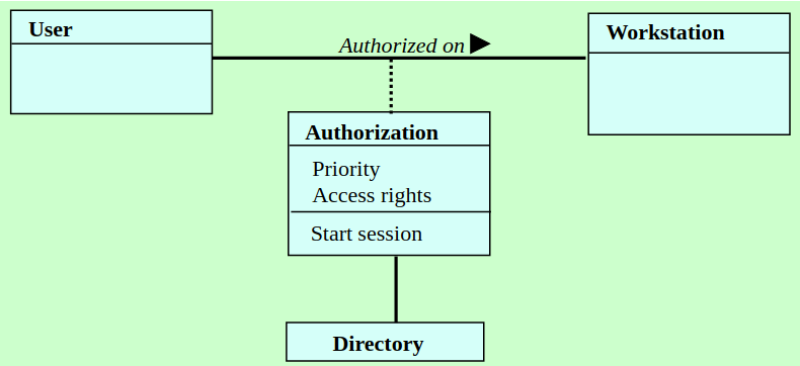


Convenzioni per i nomi

Le convenzioni per i nomi spesso consentono di inferire i nomi dei messaggi dal diagramma.
Per rendere i nomi più leggibili la convenzione è di usare la scrittura camelCase.
Uno studio dice che nomi con più di 10 caratteri sono illeggibili.

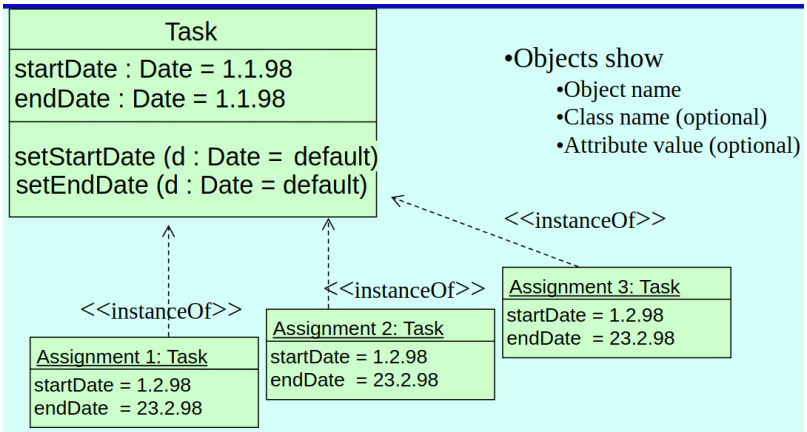
Classi-associazioni

Le relazioni (associazioni) possono diventare esse stesse delle classi.
Utile se gli attributi non appartengono a nessuna classe ma all'associazione. Lo svantaggio è la difficoltà di configurazione.

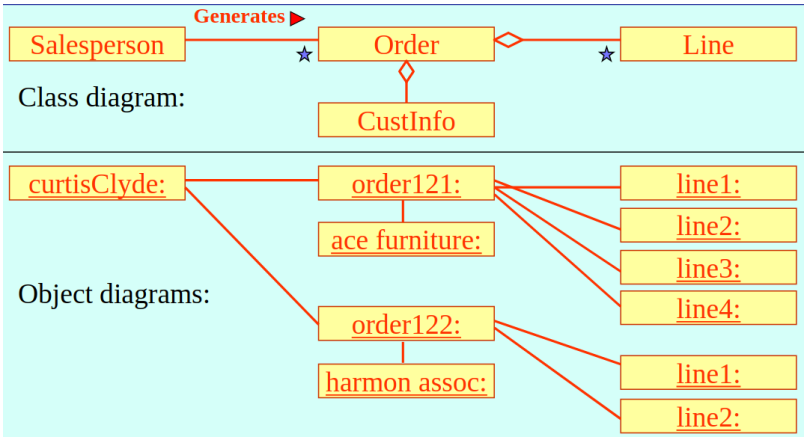


Oggetti

Come detto, una classe è un modello: definisce la struttura di un "gruppo" di oggetti. Ogni volta che istanzio una classe creo un oggetto di tipo Classe. In UML bisogna sempre distinguere oggetti da sottoclassi.
Gli oggetti mostrano: il nome dell'oggetto, il nome della classe (opzionale), il valore degli attributi (opzionale).

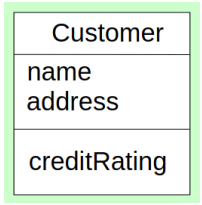


Esempi di classi e oggetti



Attributi

Gli **attributi** sono ciò che usiamo per definire lo stato di un oggetto,
che lo distingue da tutti gli altri oggetti.



Gli attributi lavorano su tre livelli:

- concettuale: indica che il cliente ha un nome
- di specifica: il cliente può dirti il nome ed impostarlo
- implementazione: è disponibile una variabile d'istanza

Differenza tra attributi ed associazioni
A livello concettuale non c'è così tanta differenza, definisco le associazioni e gli attributi, i quali hanno un valore singolo (0..1).
A livello di implementazione vado ad associarli. Gli attributi memorizzano valori, non riferimenti. Non c'è condivisione dei valori degli attributi tra istanze diverse.
Quando si definisce una classe bisogna capire cosa rendere un attributo e cosa una relazione/associazione. Se un'entità è abbastanza grossa da avere vita propria diventerà una classe, se è un valore semplice o scalare (numeri, stringhe, date...) sarà un attributo, non c'è una regola precisa che separi le due cose.

Operazioni

Le **operazioni** sono i processi, le attività, che vengono portate avanti da una classe. A livello concettuale sono viste come responsabilità. A livello di specifica sono date dai messaggi che si scambiano le classi.
Normalmente non si mostrano le operazioni che manipolano gli attributi (getter e setter).

Sintassi UML per le operazioni
`visibility name(parameter list): return-type-expression`
Es. *+ assignAgent(a: Agent): Boolean*

- visibility = public (+), protected (#), private (-)
    - L'interpretazione dipende dal linguaggio
    - Non è necessaria a livello concettuale
- name = stringa
- parameter list = argomenti (sintassi come quella degli attributi)
- return-type-expression = specifica dipendente dal linguaggio

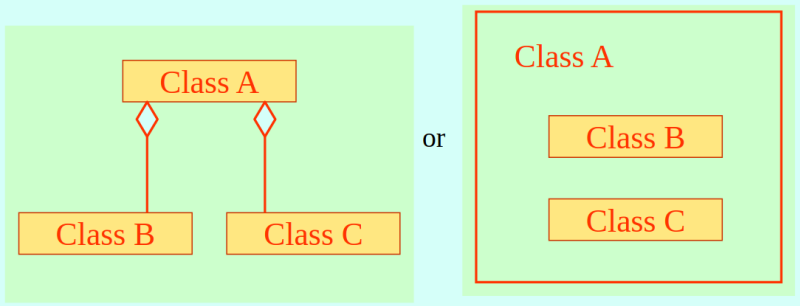
Tipi di operazioni
Tipicamente le operazioni possono essere suddivise in due categorie: *query* e *modificatori*.
- Query: ritorna un qualche valore senza modificare lo stato interno della classe. Possono essere eseguite in qualsiasi ordine.
- Modificatore: cambia lo stato interno.
Operazioni getting e setting: getting -> query, setting -> modificatore.

Aggregazioni

Una forma particolare di associazione è l'**aggregazione**. Per riconoscerla mi devo chiedere se ho una relazione di *part-of* -> i componenti sono parte dell'oggetto aggregato. Es. *La macchina ha un motore e delle ruote come sue parti.* L'aggregazione è transitiva. Non la vedo come un link ma come un campo.

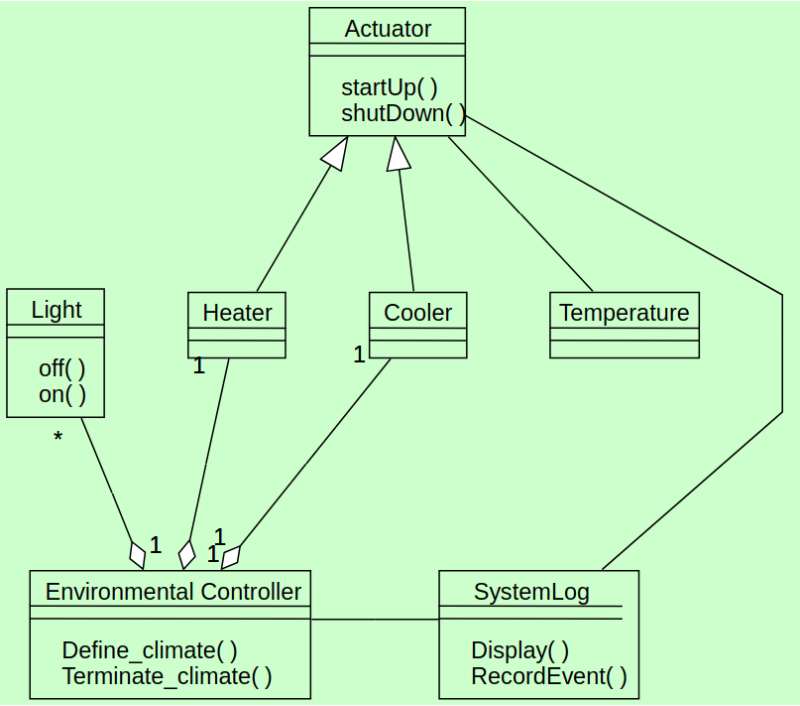
Notazione

Il diamantino vuoto (rombo) rappresenta l'aggregazione: definisce un oggetto che è parte di un altro oggetto.

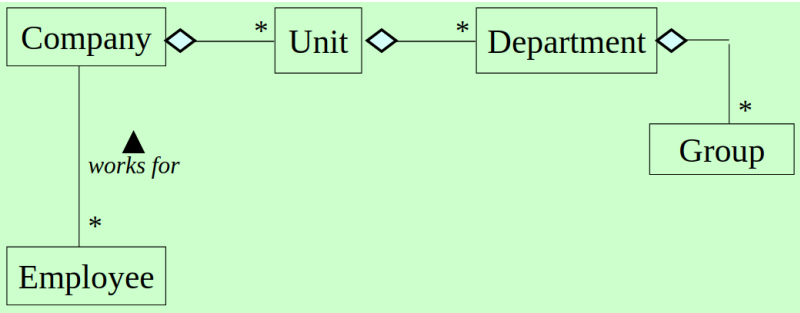


Esempio dell'aria condizionata

Il controllo ambientale contiene al suo interno una luce, un riscaldatore ed un raffreddatore.

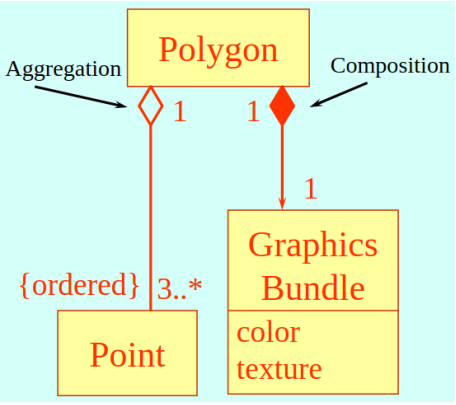


Altro esempio



Composizione

Oltre all'aggregazione, UML specifica la cosiddetta **composizione**. È una particolare aggregazione che appartiene fortemente alla classe: se muore la classe, muore anche il componente. Viene rappresentata con il diamantino pieno (rombo).

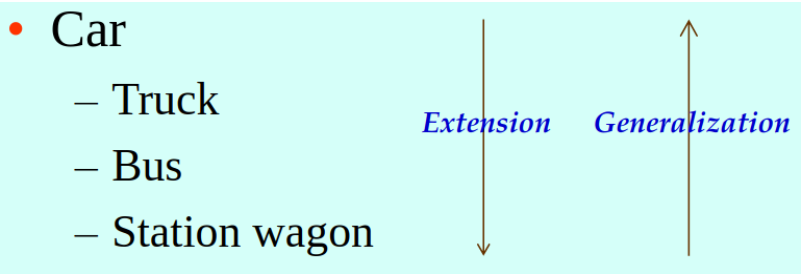


Esercizio proposto

Sviluppare il class diagram per il caso del NYSE.

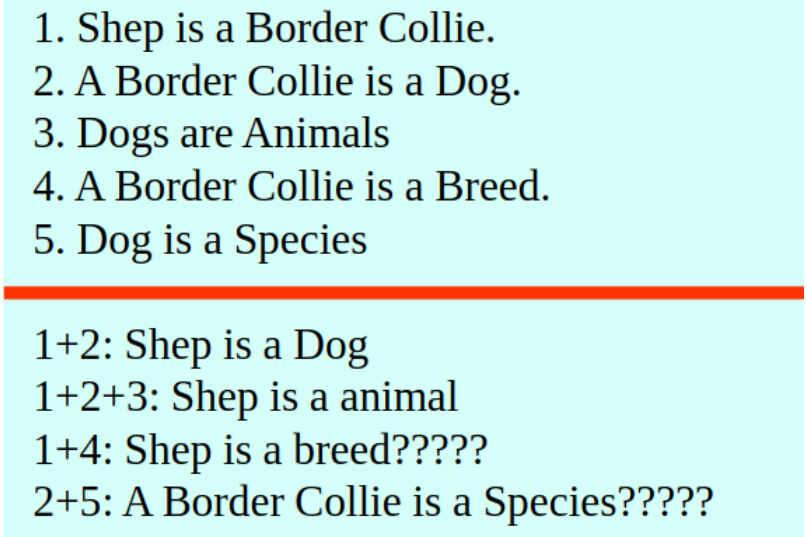
Generalizzazione vs estensione

Generalizzazione ed estensione sono molto simili, ma hanno alcune differenze.



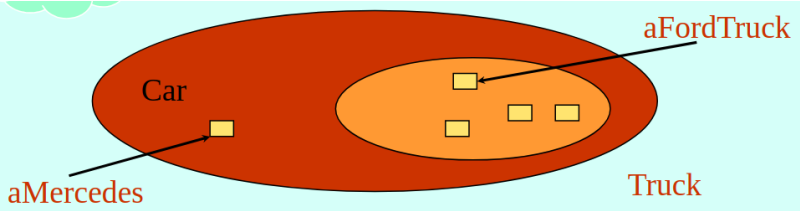
Istanziazione e generalizzazione

La generalizzazione è transitiva (is kind of), l'istanziazione non lo è (is instance of).



Concetto di generalizzazione

Una classe definisce in modo implicito un insieme di oggetti. *aCar appartiene all'insieme di tutte le macchine Car* -> istanziazione.
La generalizzazione rappresenta una relazione di sottoinsieme. *L'insieme dei camion Truck è sottoinsieme dell'insieme dei veicoli Car.* Non devo per forza arrivare in fondo all'albero gerarchico per istanziare gli oggetti.



Come definire classi (rivisto)

- Cercare i nomi negli Use case.
- Definire una classe per ogni nome (e in più aggiungerne altre).
- Documentare le regole che determinano l'insieme degli oggetti appartenenti alla classe.
- Aggiungere associazioni per modellare le relazioni.
- Pensare alle relazioni di sottoinsieme per formare generalizzazioni.

A quale classe appartiene un oggetto?

Definizione di appartenenza alla classe:

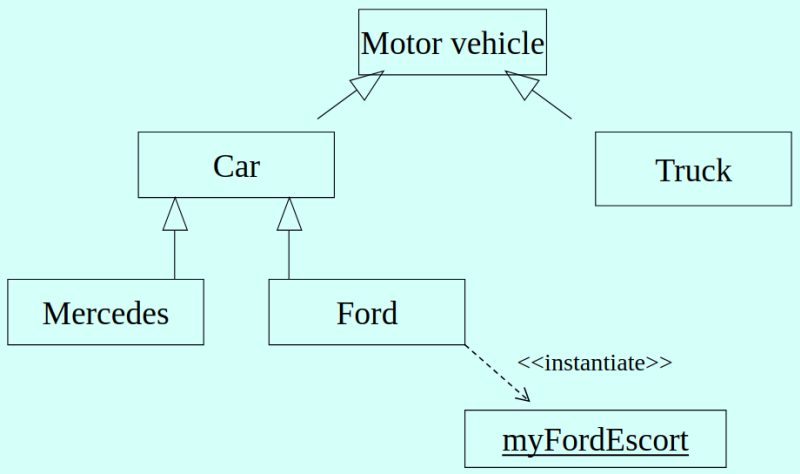
- Implicita tramite regole:
  - le regole definiscono condizioni per essere un membro della classe
  - se i valori degli attributi sono disponibili, la classe può essere determinata
  - logica terminologica dell'IA (subsunzione, classificatore)
- Esplicita tramite enumerazione:
  - l'istanziazione definisce l'appartenenza alla classe
  - problema: operazioni proibite che violano le restrizioni

Cambiare le classi

UML rende possibile cambiare classe ad un oggetto dinamicamente (usando il tipo stereotipo).
Questo è effettuato in C++ e Java avendo una classe base comune, poi cambiando l'oggetto puntato/riferito con un costruttore adatto.

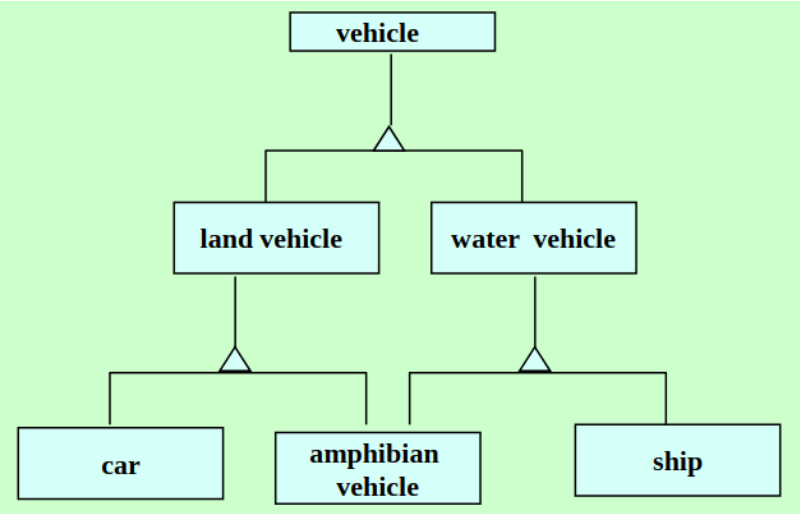
Ereditarietà

Un altro tipo particolare di associazione è la relazione di **ereditarietà**, tramite cui attributi ed operazioni della classe antenata sono ereditati nella sottoclasse. Può essere di due tipi:
- Estensione: aggiungere nuovi attributi e/o operazioni.
- Restrizione: aggiungere restrizioni sugli attributi dell'antenato. Es. *Un Cerchio è un Ellisse con gli assi lunghi uguali.*
Nel class diagram si rappresenta con un triangolino vuoto.



Ereditarietà multipla

Con l'ereditarietà multipla una classe eredita le caratteristiche da più di una superclasse.



Vantaggi:
- più vicina al modo di pensare umano
- maggiore flessibilità per specificare le classi
- maggiori chance di riuso

Svantaggi:
- mancanza di chiarezza (quale metodo è eseguito)
- implementazione più complicata
- risoluzione dei conflitti necessaria per caratteristiche multiple ereditate

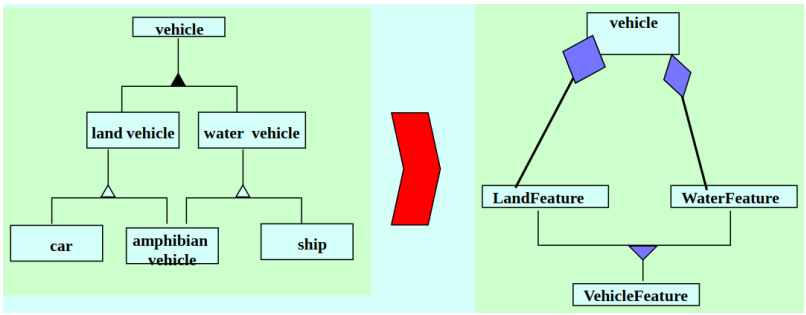
Ereditarietà virtuale

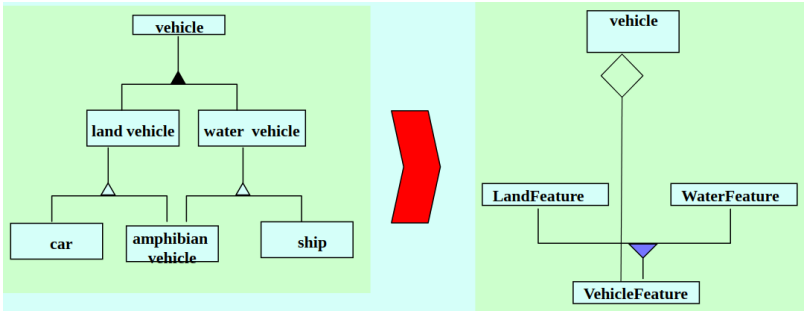
L'ereditarietà virtuale è una forma particolare che eredita una sola volta la classe base, quindi un campo all'interno della classe base viene condiviso se entra a far parte di ereditarietà multipla.
Se nell'ereditarietà ho un misto di virtual e non-virtual, avrò un numero di campi della classe base pari a uno per ogni eredità non virtual + 1 se c'è almeno una eredità virtual.
Nei diagrammi l'ereditarietà virtuale viene rappresentata con il triangolino pieno.

Evitare l'ereditarietà multipla

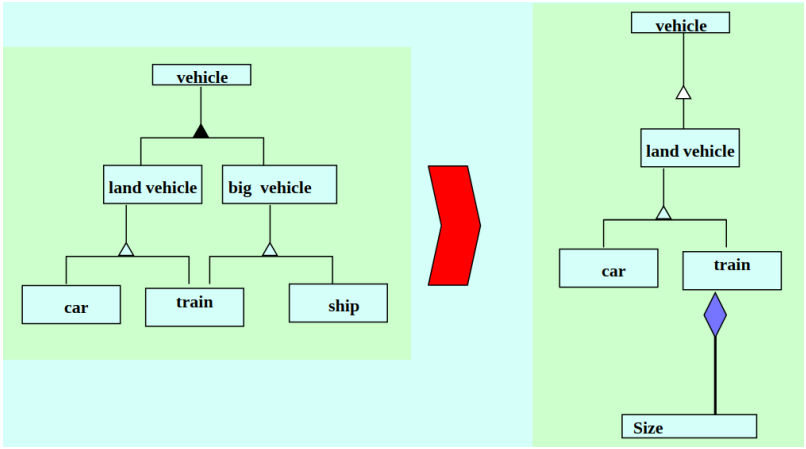
Di base è una questione di implementazione. La via più semplice spesso è ristrutturare il modello, possibile tramite tecniche diverse:
- Delegazione e aggregazione
- Ereditarietà basata sulla caratteristica più importante e delegazione del resto
- Generalizzazione basata sulle diverse dimensioni

Delegation & aggregation

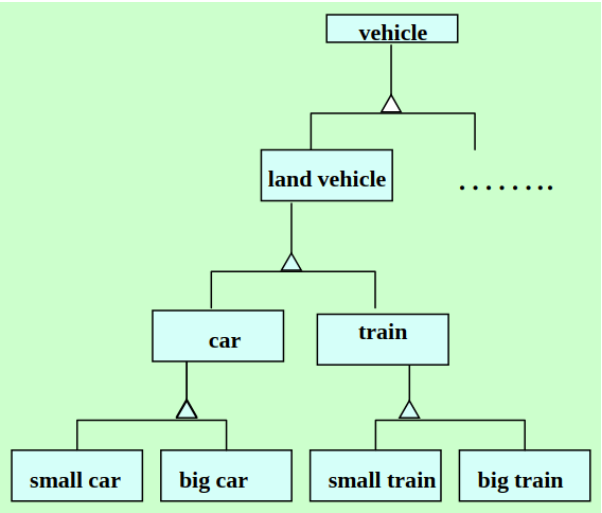




Most important feature & aggregation



Generalization based on different dimensions



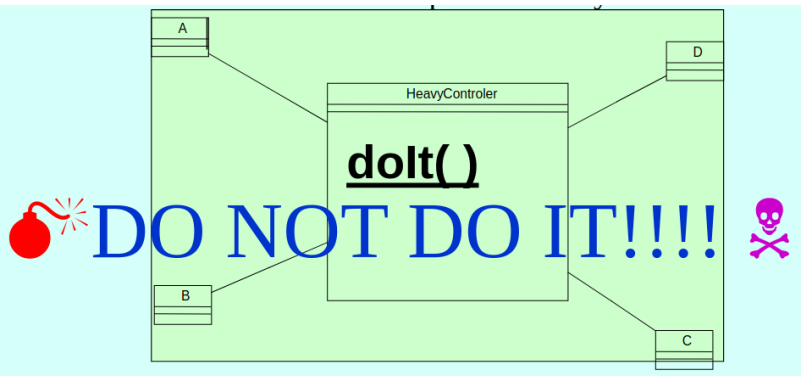
Conclusioni

I class diagram sono la spina dorsale degli approcci Object Oriented per lo sviluppo.
> Non usare tutte le notazioni: cominciare con cose semplici.
> Considera la prospettiva: non troppi dettagli in analisi, la specifica spesso è meglio dell'implementazione.
> Concentrati su aree chiave: meglio pochi diagrammi aggiornati che molti modelli obsoleti.

Creare un class diagram
Iniziare con le cose semplici, classi principali ed associazioni ovvie. Poi aggiungere attributi, molteplicità ed operazioni. Quante meno cose complesse ci sono, meglio è.

Regole pratiche

- Una classe può essere parte di numerosi diagrammi.
- I diagrammi dovrebbero illustrare aspetti specifici.
    - non troppe classi
    - non troppe associazioni
    - nascondere attributi ed operazioni irrilevanti
- Fare tanti diagrammi che diano punti di vista diversi del sistema.
- Servono numerose iterazioni per creare un diagramma.
- Evitare classi "pesanti", in cui il controller fa tutto e le altre classi contengono solo i dati.

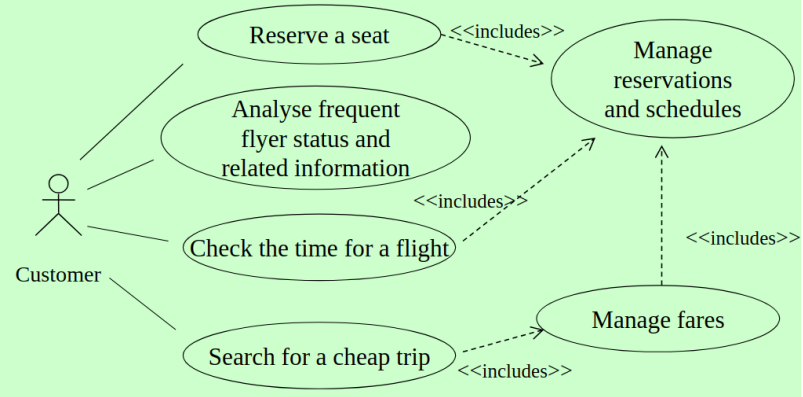


Esercizi proposti

Lufthansa example (with partial solution)

Model the classes and the objects in the Lufthansa web site. The starting point is the analysis of the use cases.
[Shorter form] Focus on the frequent flyer subsystem.

*The **user** enters the subsystem. To do so it needs to be a **frequent flyer**. Inside the subsystem, the user can access (a) general information about the **frequent flyer program** -the **reward schema**, how to enroll, how to earn **miles** with **partner companies**, and (b) specific information on her/his **miles earned**, the **status level**. The user can also update her/his **address**.*



Taxi drivers

Develop an OO Analysis Model for a system supporting the reservation and scheduling for taxi drivers that was discussed in the first day of the course.

Stocks Trading Service

By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the user to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a particular stock is no longer available.

Network Printing Service

There is a super high-resolution color laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc...). In addition, users can monitor their own print jobs and delay or delete jobs as they see fit.
To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.

Component Brokerage System

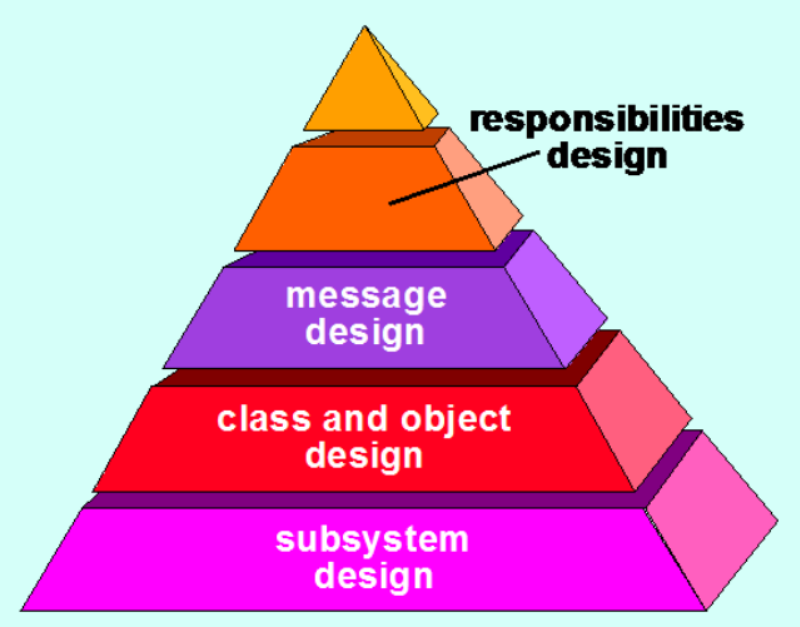
This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.
Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.
Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.

Bug Tracking System

Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.

UML - parte 2

Design Object Oriented in UML



Progettazione Object Oriented

Il primo vantaggio del design Object Oriented è quello di mantenere la stessa struttura di base quando si passa dall'analisi al design. Abbiamo un racconto in cui non dobbiamo cambiare le descrizioni, come avremmo dovuto fare passando da un linguaggio ad un altro.

Problemi di design

Ci sono diverse caratteristiche che dobbiamo tenere in considerazione quando sviluppiamo un sistema software. Sono problemi che piano piano andiamo a sbrogliare durante la fase di design.

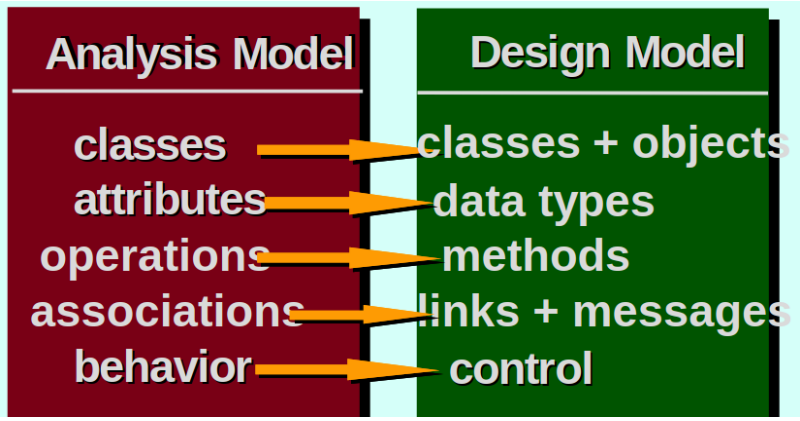
- **Decomponibilità**: la facilità con cui un metodo di design permette di decomporre un problema grande in sotto-problemi più semplici da risolvere.
- **Componibilità**: la modalità con cui un metodo di design assicura che i componenti del programma (moduli), una volta progettati e creati, possano essere riusati per comporre altri sistemi.
- **Comprensibilità**: la facilità con cui un componente del programma può essere compreso, senza riferimenti ad altre informazioni o altri moduli, così che sia possibile capire cosa avviene nel sistema.
- **Continuità**: la capacità di apportare piccole modifiche in un programma e di far sì che esse si manifestino con modifiche corrispondenti in uno o pochissimi moduli.
- **Protezione**: una caratteristica architetturale che riduce la propagazione di effetti collaterali se avviene un errore in un determinato modulo.

Input per il design

- **Componenti del dominio del problema**: i sottosistemi che sono responsabili dell'implementazione diretta dei requisiti del cliente, ovvero il sistema che devo andare a costruire.
- **Componenti di interazione umana**: i sottosistemi che implementano l'interfaccia utente (inclusi sistemi di GUI riutilizzabili), quindi il modo in cui strutturo l'interfaccia che presento all'utente.
- **Componenti di gestione dei compiti**: i sottosistemi responsabili del controllo e della coordinazione dei task concorrenti che potrebbero essere impacchettati all'interno di un sottosistema oppure tra sottosistemi differenti. In sintesi la gestione del controllo, cioè come eseguo gli algoritmi necessari.
- **Componenti di gestione dei dati**: i sottosistemi responsabili della memorizzazione e del recupero degli oggetti.

Dall'analisi al design

La parte più critica del sistema è la fase di analisi.



Cosa distingue l'OOD dall'OOA?

- Il livello di dettaglio: fissiamo i nomi, fissiamo la signature dei metodi, la visibilità, la molteplicità, gli algoritmi per i metodi...
- Notazioni addizionali per i diagrammi.

Perciò nell'OOD abbiamo ancora i diagrammi di classe, sono però raffinati per matchare il design del sistema. Oltre ai class diagram, abbiamo numerosi altri diagrammi:
Diagrammi strutturali:
- Object diagram
- Deployment diagram

Diagrammi comportamentali:
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

UML include ulteriori diagrammi che non useremo, ad esempio il component diagram.

Diagrammi strutturali per l'OOD

Class diagram: la loro struttura è la stessa dell'OOA.
Object diagram: hanno a che fare con oggetti, ossia istanze di classi, quindi sono assolutamente equivalenti ai class diagram. Per questo non li analizzeremo a fondo.

Diagrammi comportamentali per l'OOD

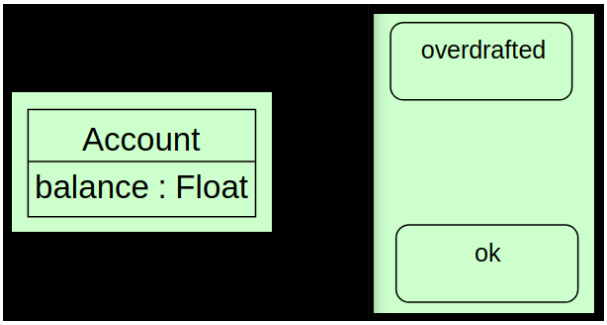
Statechart diagram: descrivono l'evoluzione degli stati di qualsiasi classificatore. Sono solitamente usati per gli oggetti.
Activity diagram: descrivono l'evoluzione delle attività nel sistema.
Sequence diagram: descrivono le interazioni tra gli oggetti ordinate temporalmente.
Collaboration diagram: descrivono le interazioni tra gli oggetti sulla base dell'organizzazione.
Sequence e collaboration sono detti diagrammi di interazione.

Statechart diagram vs interaction diagram

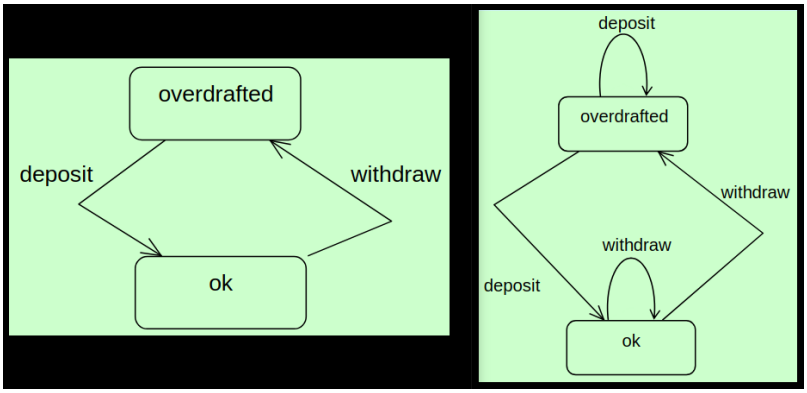
I **diagrammi di interazione** mostrano come gli oggetti interagiscono tra di loro. I **diagrammi statechart** mostrano il comportamento di un singolo oggetto: come evolve il proprio stato in base ai messaggi ricevuti. Posso pensare oggetti che in base ai cambiamenti dei valori delle variabili cambiano il proprio stato. *Analogia con automa a stati finiti.* Altri nomi possono essere state transition, state diagram, Harel diagram.

Statechart diagram

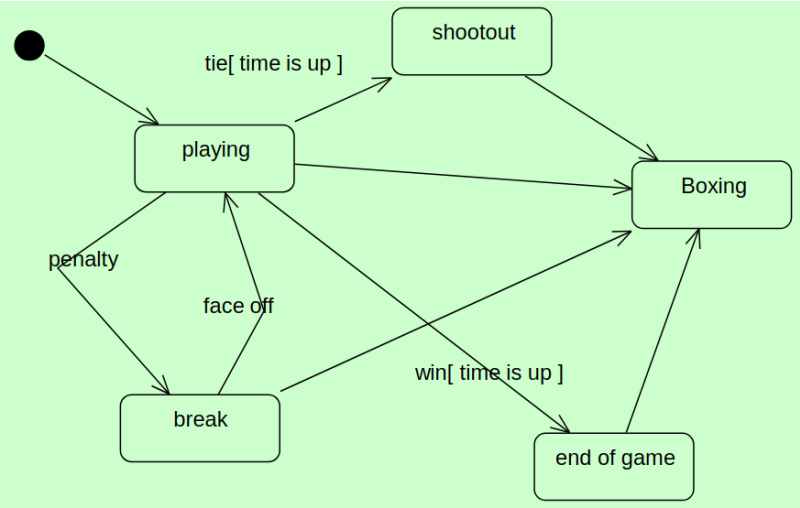
Lo **stato** è l'insieme di valori che descrive un oggetto in uno specifico momento. Lo stato è determinato basandosi sui valori degli attributi.



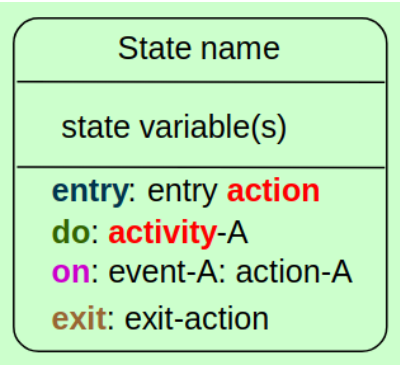
Lo stato può cambiare a seguito di un evento, come la ricezione di un messaggio, oppure può rimanere uguale.



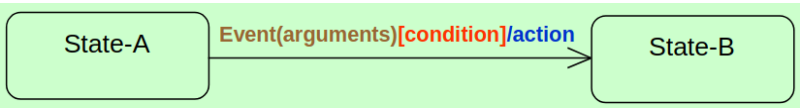
Esempio di diagramma statechart: Hockey Game



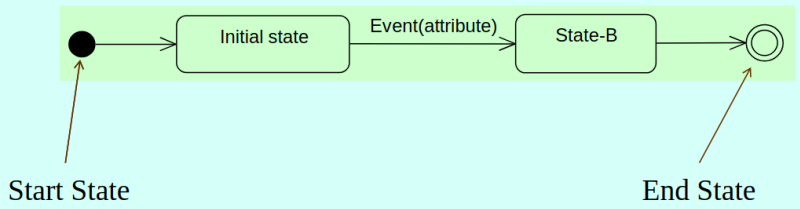
Notazioni dei diagrammi statechart



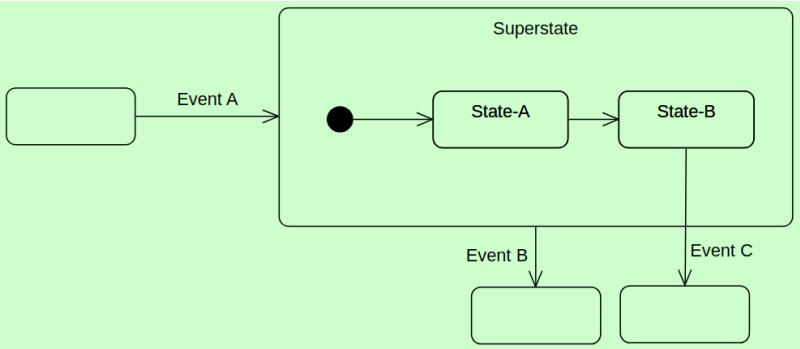
I diagrammi di stato sono rappresentati tramite quadrati con i bordi arrotondati (uno per ogni stato) che contengono al loro interno variabili di stato, le quali descrivono univocamente lo stato ed hanno associate ad esse attività e/o azioni. Un'**attività** può essere lunga e ammette interruzioni, un'**azione** avviene rapidamente.
- **entry**: un'azione effettuata all'ingresso nello stato.
- **do**: un'attività in corso effettuata mentre si è nello stato. Es. *mostrare una finestra*.
- **on**: un'azione effettuata come risposta ad uno specifico evento.
- **exit**: un'azione effettuata all'uscita dallo stato.



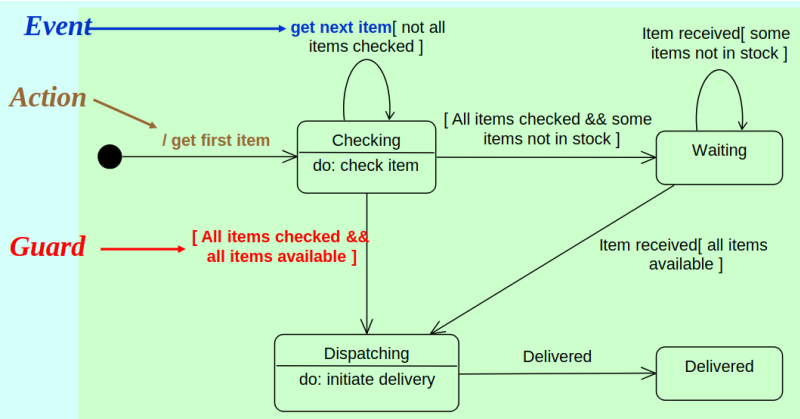
Sulla base di un evento possiamo decidere di passare da uno stato all'altro. La transizione tra stati è descritta dalla freccia.
- **Event**: messaggio da inviare.
- **Guard condition**: la transizione avviene solo quando la guardia è true. Le guardie per le transizioni di uscita da uno stato sono mutualmente esclusive.
- **Action**: processo che si assume avvenga rapidamente e che non può essere interrotto.
Ciascuna parte può essere omessa.



Stato iniziale = pallina nera, stato finale = doppio cerchietto.



Esempio di diagramma statechart: Order Management



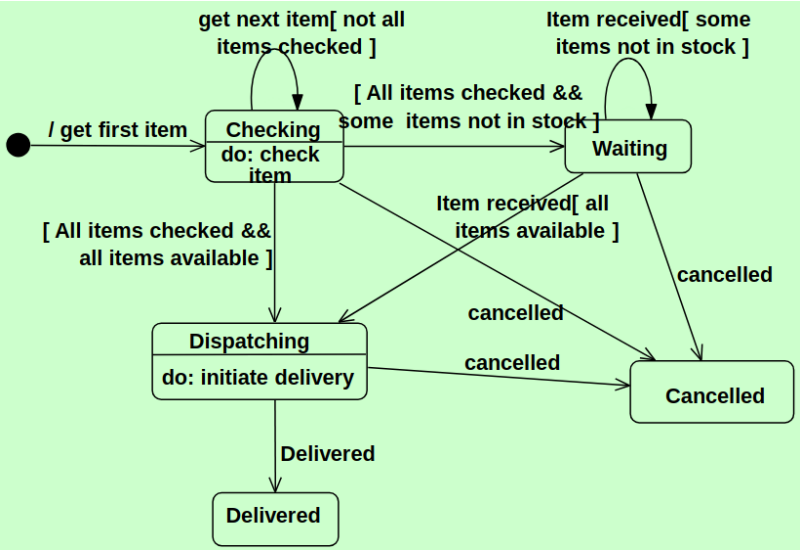
Problema: cancellazione dell'ordine

Vogliamo essere in grado di cancellare un ordine in qualsiasi momento.

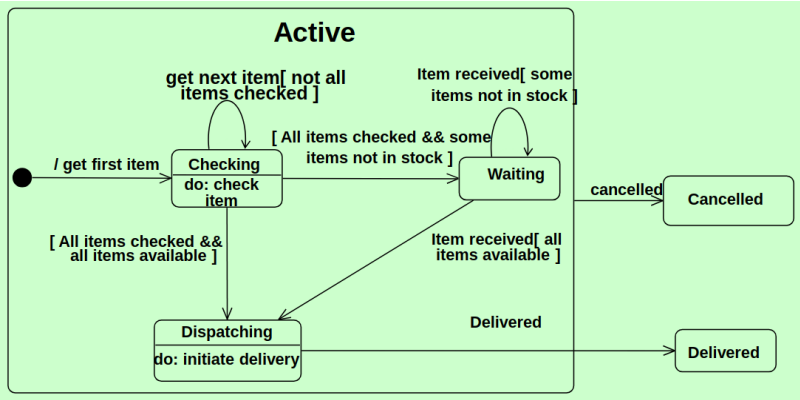
Soluzioni:

- Transizioni da ogni stato verso uno stato "cancelled".
- Super-stato e transizioni singole.

Transizioni verso "cancelled"



Super-stato e sotto-stati



Conclusioni

I diagrammi statechart non devono necessariamente fare riferimento a classi o oggetti, possono anche riferirsi a sotto-sistemi... In pratica questo è comunque il modo più comune per usare i diagrammi statechart.

Argomenti non trattati

- Stati annidati vs stati concorrenti
- Stati storici/di cronologia
- Transizioni interne

Esercizi proposti

Define the statechart diagram of a basketball game.

Define a statechart diagram of your software development process.

Possibile soluzione

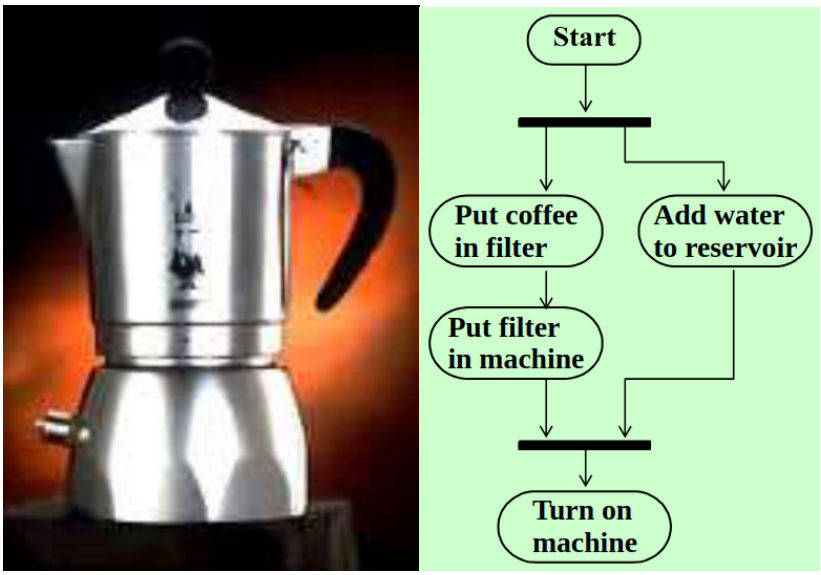
First, we create the diagram. Then, we need to set the appropriate use cases and classes. Eventually, we move to the statechart diagrams.

Activity diagram

Usati per descrivere il flusso di lavoro e i processi paralleli.
Le attività a livello concettuale sono compiti da svolgere, a livello di specifica/implementazione sono metodi di una classe.

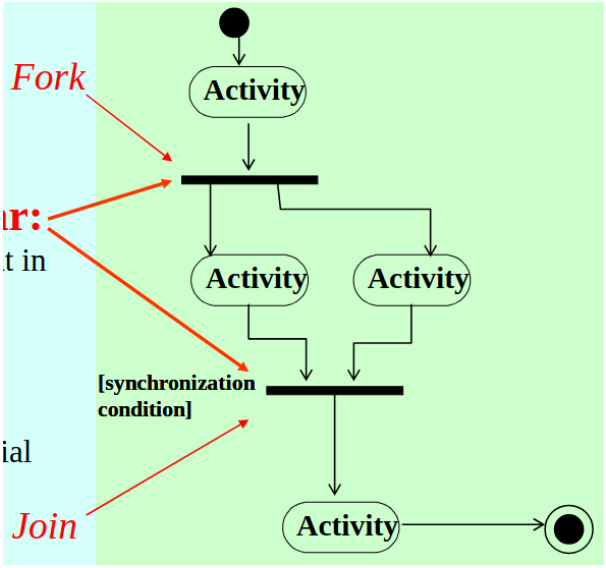
Esempio di diagramma di attività

La macchinetta del caffè.



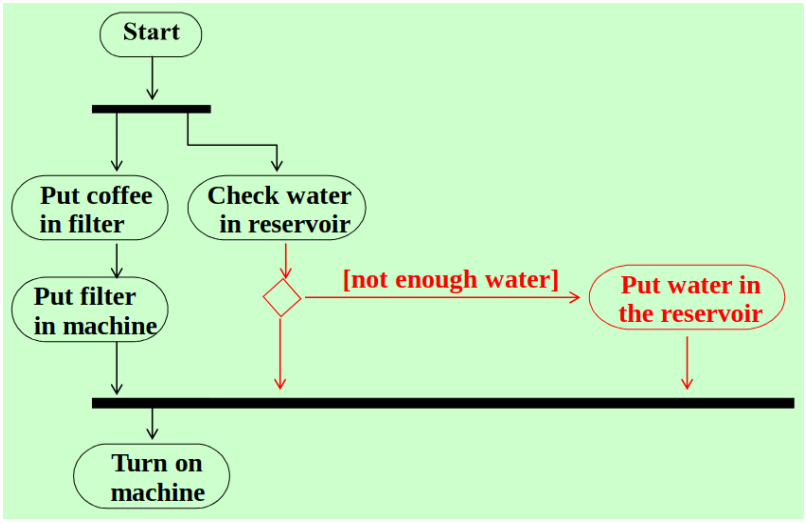
Barre di sincronizzazione

Le attività possono essere portate avanti in parallelo, in qualsiasi ordine (fork). Arrivati ad una barra con più frecce entranti serve che tutte le attività precedenti siano terminate per proseguire (join).
Un activity diagram mostra un ordine parziale delle attività.

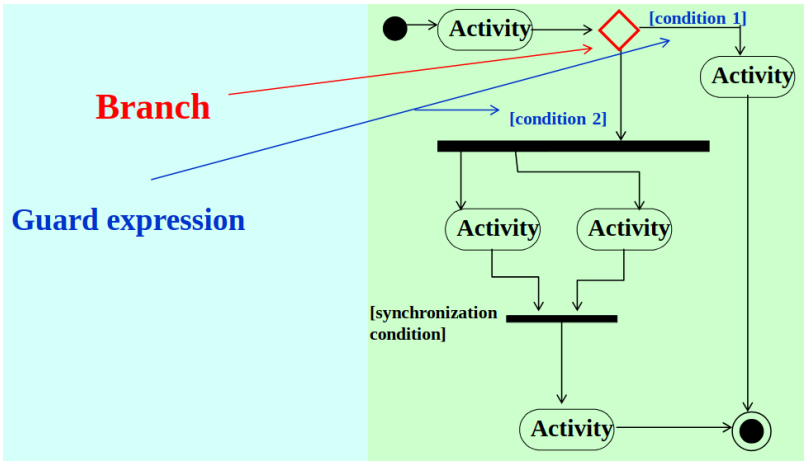


Condizioni

Anche nei diagrammi di attività posso andare a definire delle condizioni, ovvero due parti alternative: non possono procedere in parallelo, viene eseguito uno solo dei due rami.
La condizione si rappresenta con un rombo (branch) che ha due guardie di uscita: in base al loro valore si decide dove proseguire.



Struttura di un activity diagram



Argomenti non trattati

- Attività vs azioni
- Swimlanes e flusso degli oggetti
- Cambiamenti nello stato degli oggetti

Esercizi proposti

Define an activity diagram for the previously defined basketball game.

Define an activity diagram for your software development process.

Taxi drivers

Develop the relevant OO Statechart Diagrams and Activity Diagrams for a system supporting the reservation and scheduling for taxi drivers that was discussed in the first day of the course.

Books

Develop the relevant OO Statechart Diagrams and Activity Diagrams for the system that manages the search, the selection, and the purchase of books at Chapters (http://www.chapters.ca) that was discussed in the first day of the course.

Stocks Trading Service

Develop the relevant OO Statechart Diagrams and Activity Diagrams for:
By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the user to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a particular stock is no longer available.

Network Printing Service

Develop the relevant OO Statechart Diagrams and Activity Diagrams for:
There is a super high-resolution color laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc...). In addition, users can monitor their own print jobs and delay or delete jobs as they see fit.
To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.

Component Brokerage System

Develop the relevant OO Statechart Diagrams and Activity Diagrams for:
This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.
Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.
Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.

Bug Tracking System

Develop the relevant OO Statechart Diagrams and Activity Diagrams for:
Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.

Diagrammi di interazione

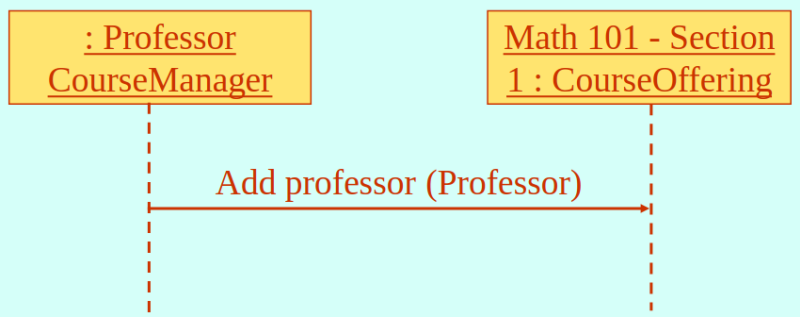
Si focalizzano sulle entità reali, cioè sugli oggetti (le classi "non esistono", sono solo uno stampino).
Mostrano in dettaglio come gli oggetti comunicano tra di loro. Abbiamo 2 possibili viste:
- Vista basata sul tempo (time-based)
- Vista basata sull'organizzazione (organization-based)

Sequence diagram

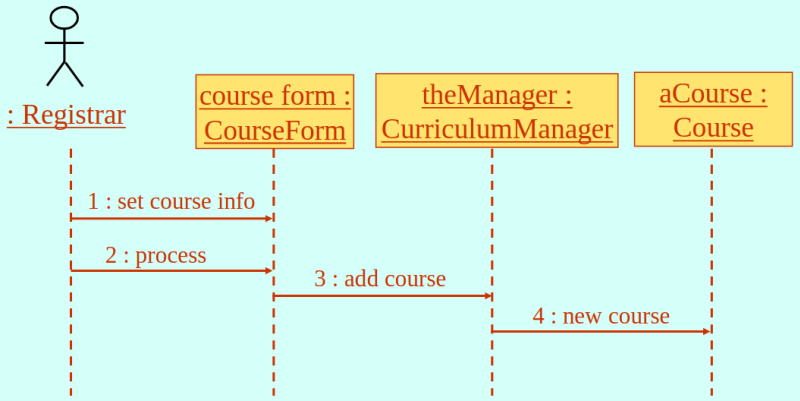
I diagrammi di sequenza mostrano le interazioni degli oggetti disposte in sequenza temporale. Si focalizzano sugli oggetti (e sulle classi) e sullo scambio dei messaggi per realizzare le funzionalità dello scenario.
Gli oggetti sono organizzati su una linea orizzontale e gli eventi su una linea del tempo verticale.

Timelines

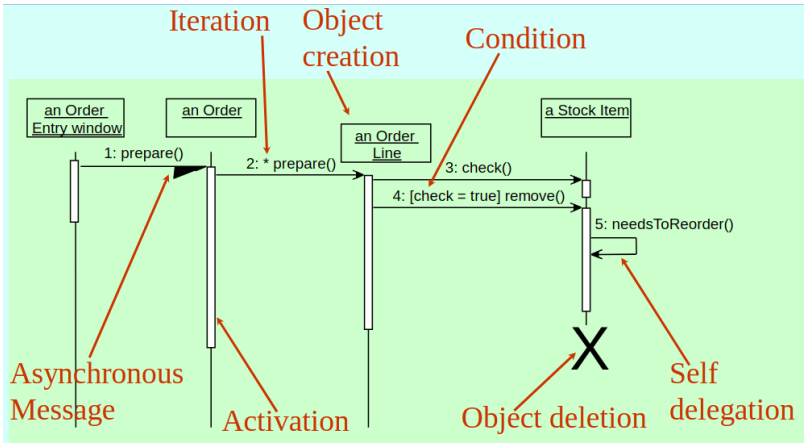
I messaggi puntano dal cliente al fornitore.



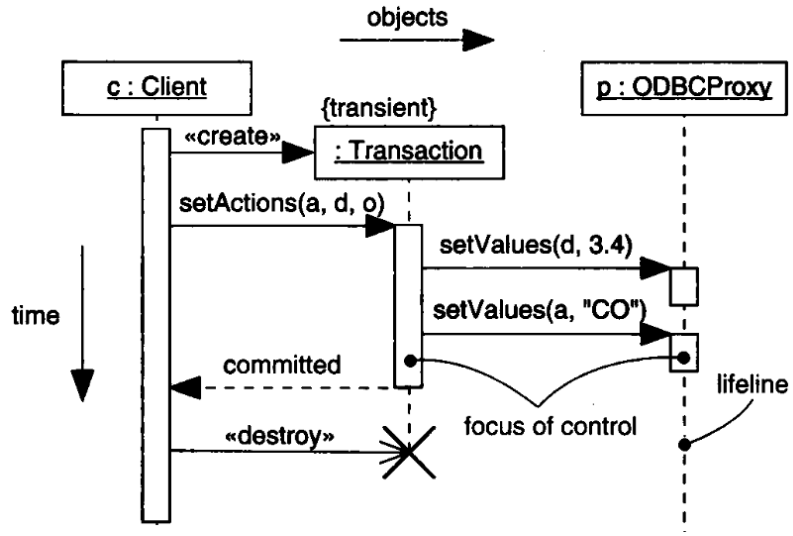
Esempi di sequence diagram



Esempio più dettagliato



Esempio di transizione

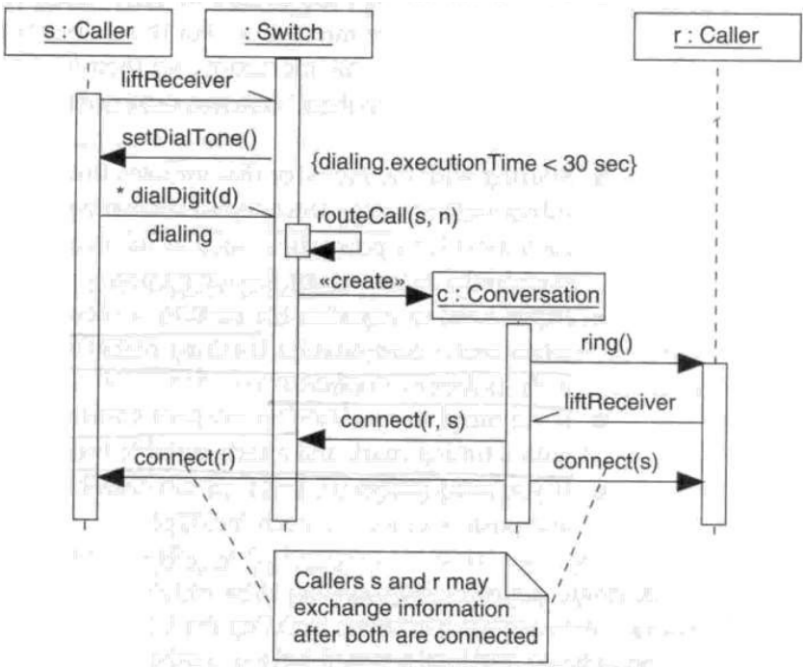


Contenuto dei sequence diagram

Oggetti: si scambiano messaggi tra di loro.
Messaggi: possono essere:
- Sincroni = "chiamate" (call events), denotati dalla freccia intera.
- Asincroni = "segnali" (signals), denotati dalla mezza freccia.
- Ci sono inoltre messaggi "create" e "destroy".

I messaggi asincroni non bloccano il chiamante. Possono svolgere 3 funzioni:
> Creare un nuovo thread
> Creare un nuovo oggetto
> Comunicare con un thread già in esecuzione

Esempio completo di sequence diagram



Complessità

KISS = Keep It Small and Simple
I diagrammi servono a chiarire le cose: se una cosa è semplice si aggiunge al diagramma, se è complessa si disegnano più diagrammi separati.

Esercizi proposti

Use the sequence diagram to model how you receive the request to develop a new piece of code.

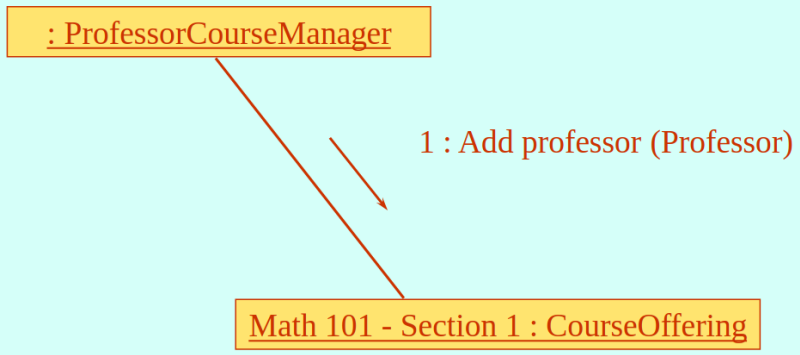
Use the sequence diagram to model how a message is sent over an Ethernet connection.

Dove sono i confini?

Un confine modella la comunicazione tra il sistema ed il mondo esterno. Es. *un'interfaccia utente.* Può essere utile mostrare le classi di confine nei diagrammi di interazione, catturando i requisiti dell'interfaccia e tralasciando come essa verrà implementata.

Collaboration diagram

Mostra come gli oggetti interagiscono rispetto alle unità organizzative (confini).
Sequenze di messaggi determinate da numerazione.
- *1, 2, 3, 4, ...*
- *1, 1.1, 1.2, 1.3, 2, 2.1, 2.1.1, 2.2, 3...* -> mostra quale operazione chiama quale altra operazione.



Collaboration vs sequence

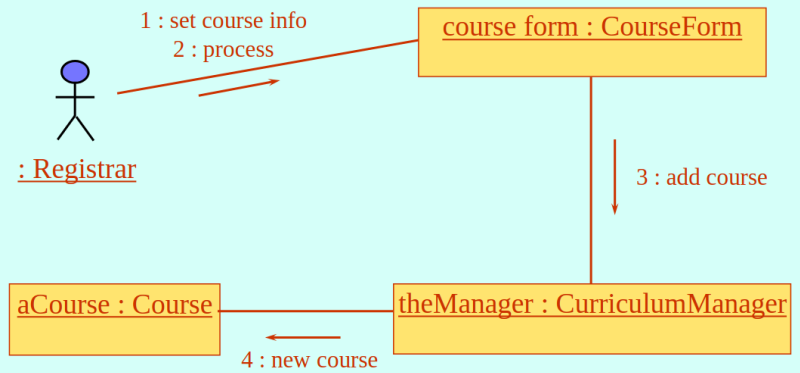
I diagrammi di sequenza sono migliori per vedere il flusso del tempo. Le sequenze di messaggi sono più difficili da capire nei diagrammi di collaborazione.
Il flusso di controllo dell'organizzazione si vede meglio tramite i diagrammi di collaborazione: il loro layout può mostrare connessioni statiche degli oggetti. Il controllo complesso è difficoltoso da esprimere in ogni caso!

Contenuto dei collaboration diagram

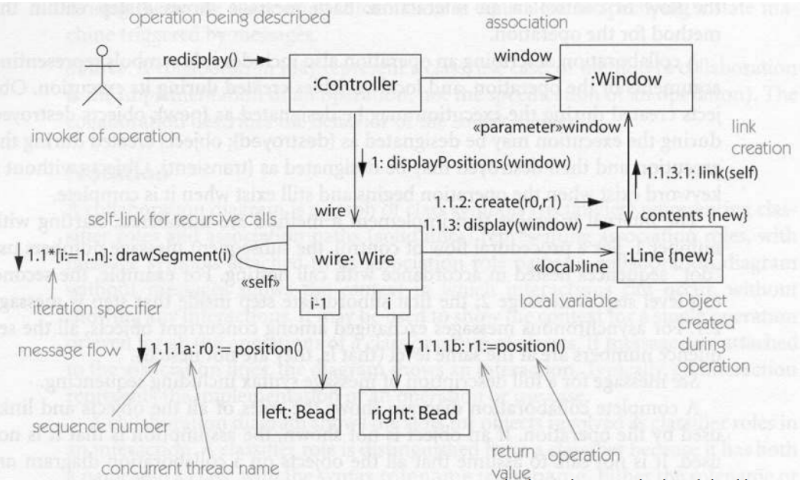
Oggetti: si scambiano messaggi tra di loro.
Messaggi: sono numerati e possono avere loop. Possono essere:
- Sincroni = "chiamate" (call events), denotati dalla freccia intera.
- Asincroni = "segnali" (signals), denotati dalla mezza freccia.
- Ci sono inoltre messaggi "create" e "destroy".

Quasi le stesse cose dei diagrammi di sequenza.

Esempio di collaboration diagram



Possono diventare molto complessi



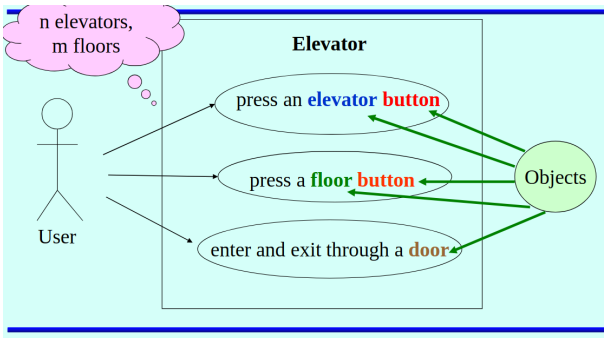
Esercizi proposti

Use the sequence diagram to model how you receive the request to develop a new piece of code. What is the difference with before?

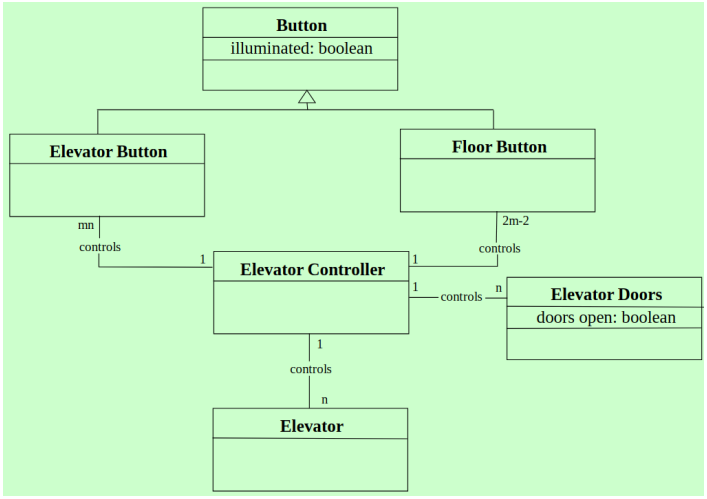
Use the sequence diagram to represent how a cellular phone moves from cell to cell and handle roaming.

Un esempio completo: l'ascensore

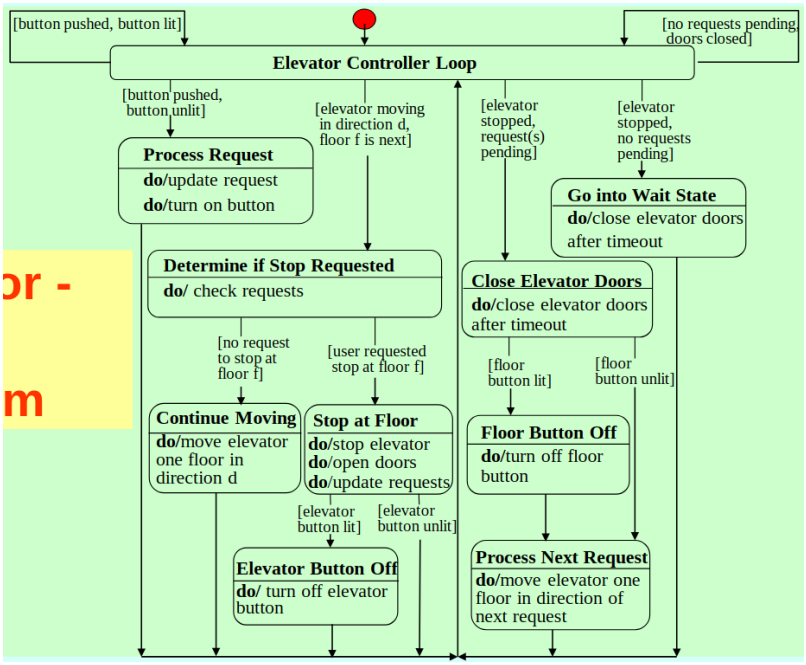
Use cases



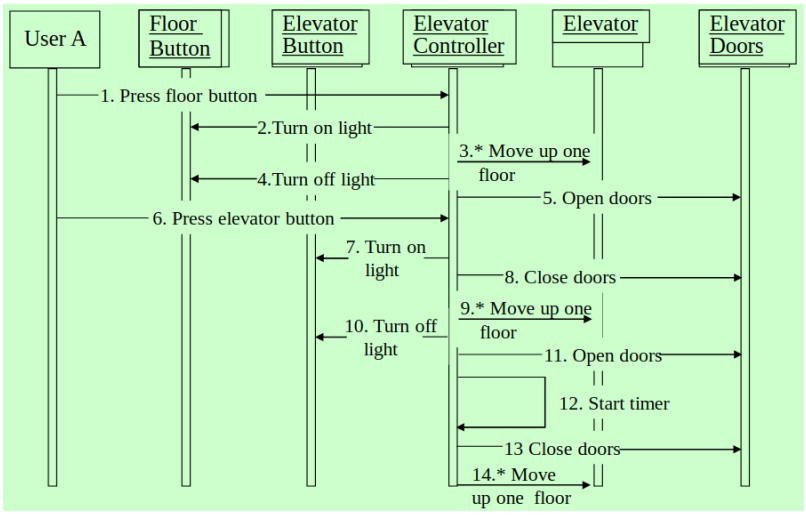
Primo class diagram



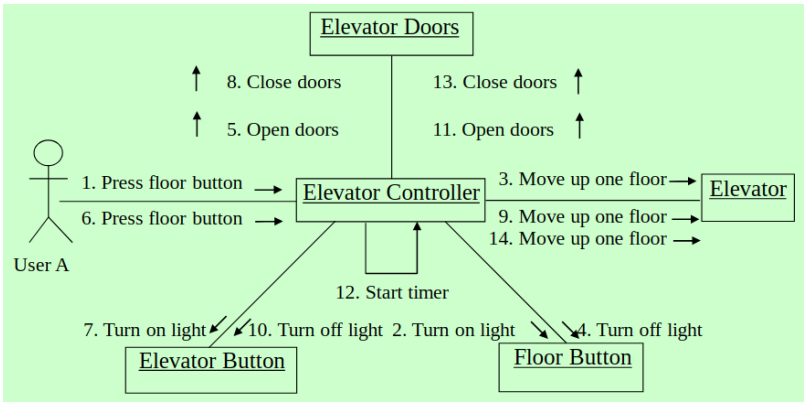
State diagram



Sequence diagram

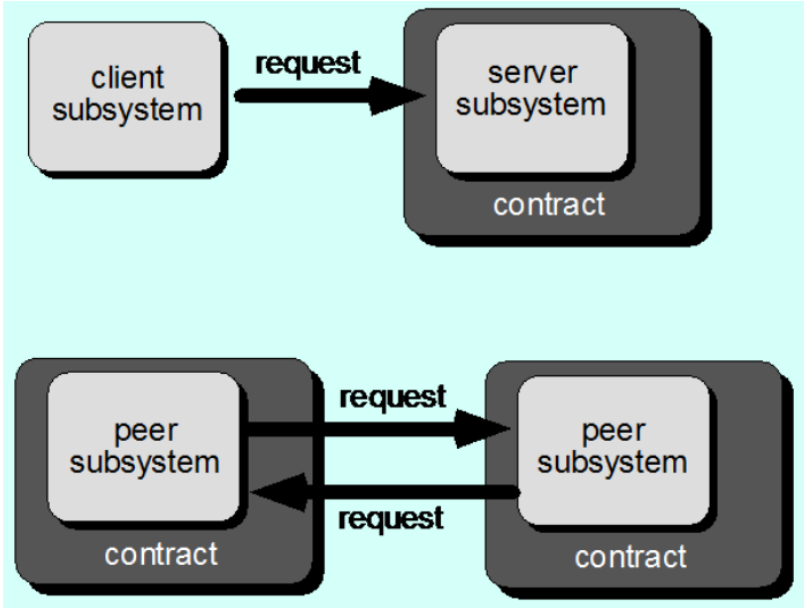


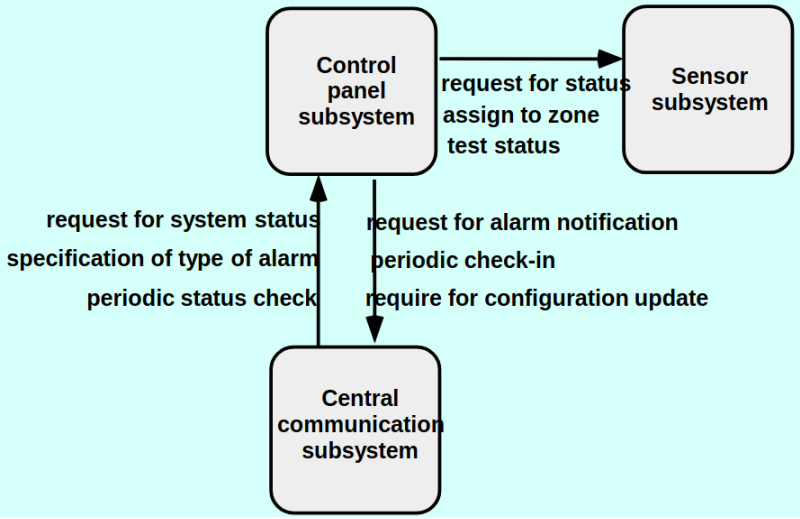
Collaboration diagram



Design di sistemi e sottosistemi

UML ha anche una parte che descrive sistemi e sottosistemi, fatta sostanzialmente a blocchi.





Come suddividere un sistema in sottosistemi più piccoli

Principio romano: divide et impera = dividere un grosso sistema in parti maneggiabili.
- Metodi strutturati: decomposizione funzionale.
- Object Orientation: raggruppare le classi in unità di livello superiore, ovvero i package (o componenti).

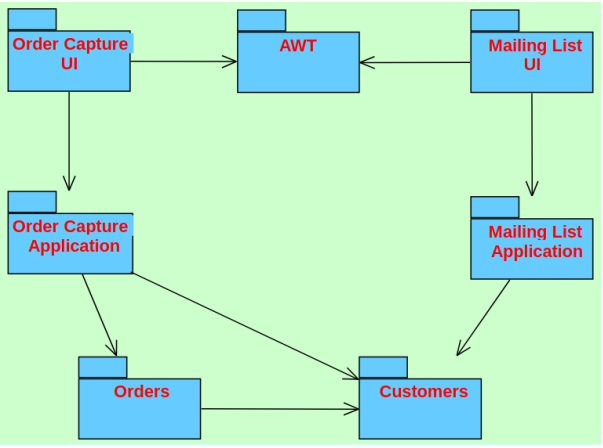
Quanto bisogna suddividere? In linea di massima il più possibile, non è sempre possibile farlo in quanto bisogna mantenere coerenza e coesione. All'interno dei linguaggi di programmazione i sistemi diventano package o moduli.

Package

In UML sono presenti i **package diagram**, che mostrano i package e le dipendenze tra di essi. Una dipendenza esiste tra due elementi se modifiche alla definizione di un elemento possono causare modifiche all'altro. Lo scopo (e l'arte) del design su larga scala è minimizzare le dipendenze, vincolando gli effetti dei cambiamenti.
Una dipendenza è rappresentata nel diagramma da una freccia: `x -> y` significa x importa y.

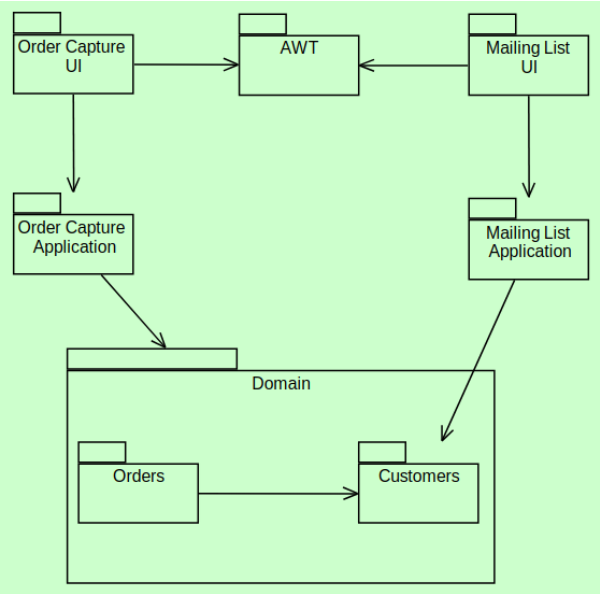
Esempio

Un sistema per spedire e gestire ordini.



Package annidati

Un package può avere dei sotto-package, di cui abbiamo due interpretazioni:
- trasparente: tutto il contenuto dei package interni è visibile.
- opaca: solo le classi del contenitore sono visibili.
Si possono rappresentare graficamente mettendoli uno dentro l'altro. Se il diagramma diventa troppo pasticciato si usa la relazione di sotto-package.
A seconda del linguaggio di programmazione posso avere informazioni dettagliate o meno, e posso importare l'intero package o anche soltanto un sotto-package.



Ridurre la complessità

Come è possibile ridurre la complessità dell'interfaccia dei package?
Bisogna assegnare a tutte le classi nel package solo la visibilità `package`. Si definisce poi una classe `public` che fornisce i comportamenti pubblici del package. Tutte le operazioni pubbliche sono da delegare alla classe appropriata all'interno del package.

Regole approssimative

1. Evitare cicli nella struttura delle dipendenze.
2. Se ci sono troppe dipendenze provare a fare refactoring del sistema.
3. Ristrutturare il sistema se il diagramma non sta interamente in una pagina A4.

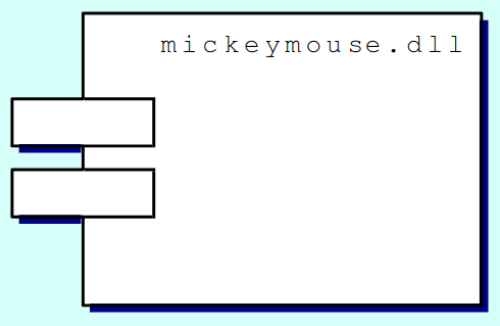
Package vs diagram

I package rappresentano le divisioni fisiche dello sviluppo. Il loro obiettivo è di semplificare la definizione di workpackages e di sviluppare sistemi software. Il package è l'entità di consegna (del deliverable), ha valore in quanto entità di lavorazione. Al cliente io vendo il package!
I diagrammi rappresentano l'assemblamento logico delle informazioni. Il loro obiettivo è di semplificare la comprensione del dominio, visualizzandolo.

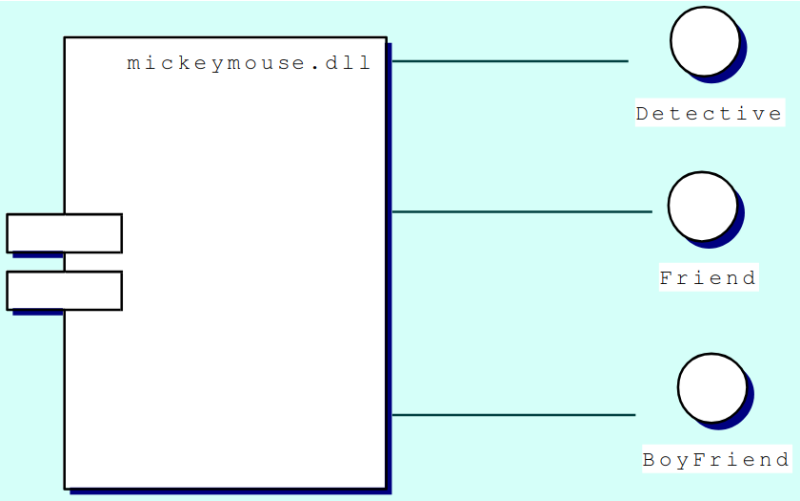
Componenti in UML

Rappresentazione in UML

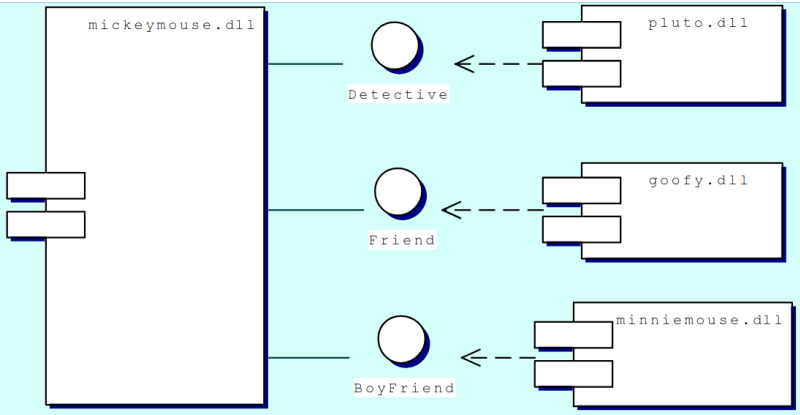
Componente, rappresentato con un rettangolo avente due "linguette" sovrapposte a sinistra.
Tipico esempio: *.dll, ovvero Dynamic Link Library* (usate in windows).



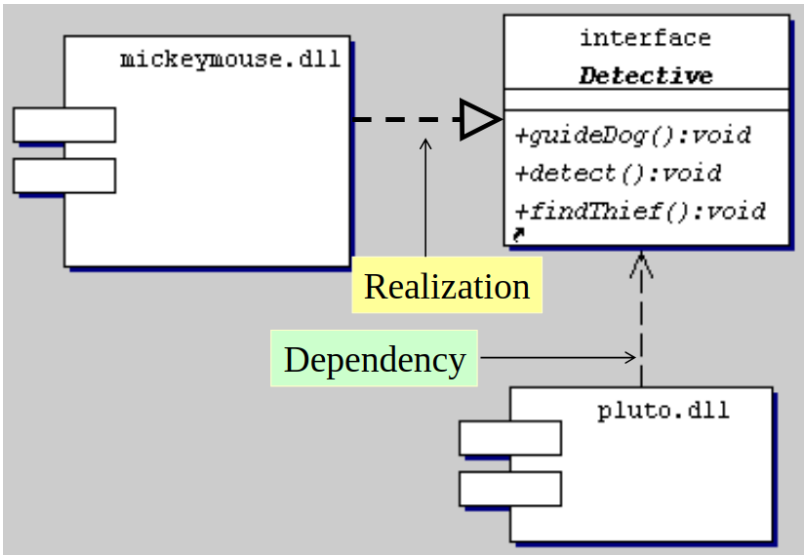
Interfacce, rappresentate da cerchi.



Collaborazioni, ovvero l'implements, rappresentate da una freccia con linea tratteggiata.
mickeymouse espone l'interfaccia Detective, che è implementata da pluto.

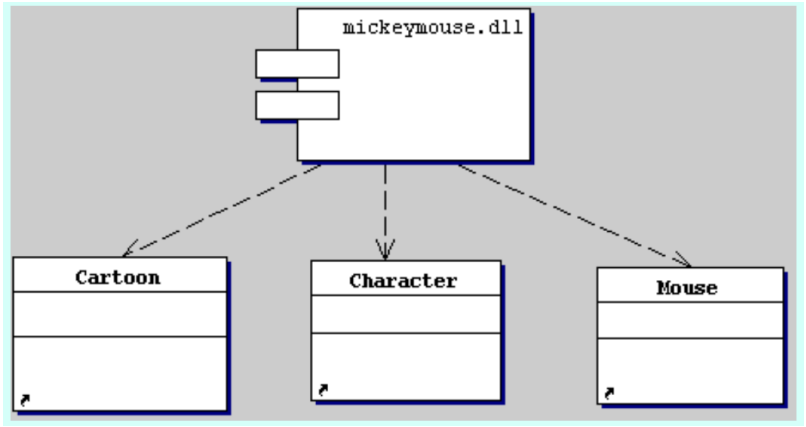


Vista dettagliata di una collaborazione

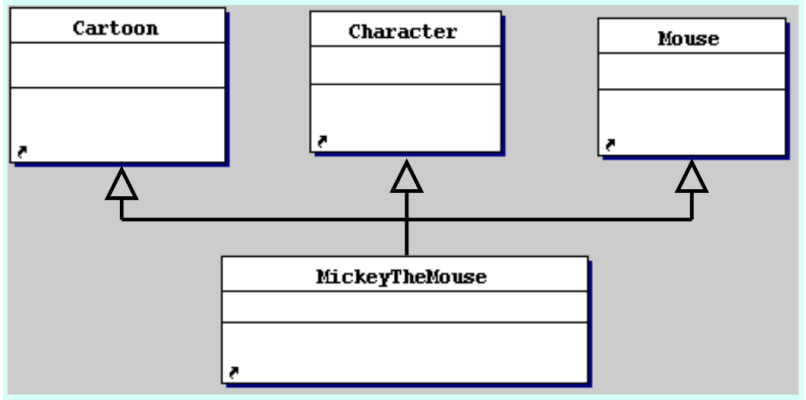


Componenti e classi

Posso strutturare gli oggetti istanziando le classi.

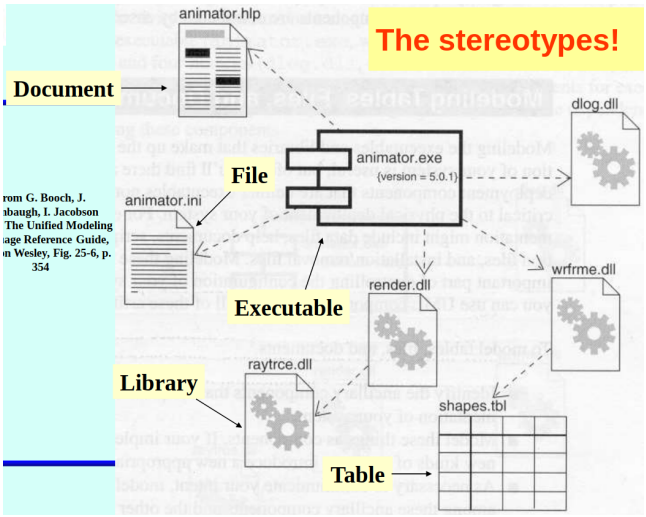


Che è totalmente diverso dal seguente, che indica MickeyMouse come un'implementazione di ...

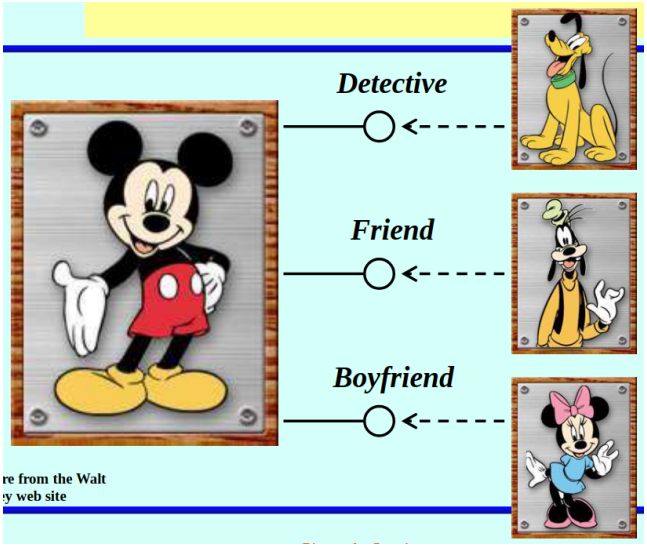


Stereotipi

Ancora più che nelle classi, è utile usare gli **stereotipi**: delle icone o dei nomi che rappresentano delle classi di componenti. UML definisce 5 tipi di stereotipi che possono essere applicati ai componenti:
- Eseguibile
- Libreria
- Tabella
- File
- Documento



Possiamo definire le nostre icone. Non mettiamo un'immagine perché bella, la mettiamo per aiutare la comprensione.



Nodi

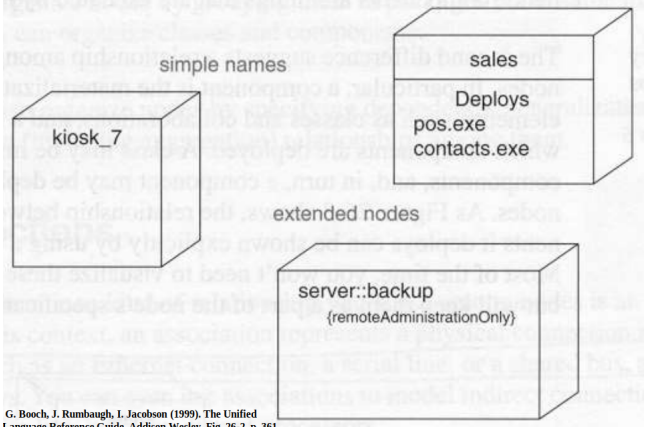
Un nodo è un elemento fisico che esiste a runtime e rappresenta una risorsa computazionale, che generalmente ha almeno un po' di memoria e spesso capacità di processare. I componenti girano sui nodi.

Rappresentazione grafica in UML

In UML rappresentiamo i nodi come dei cubi.

È possibile associare delle icone ai nodi.
Nodo con un componente che vi lavora sopra.

I nodi possono essere identificati da nomi estesi.

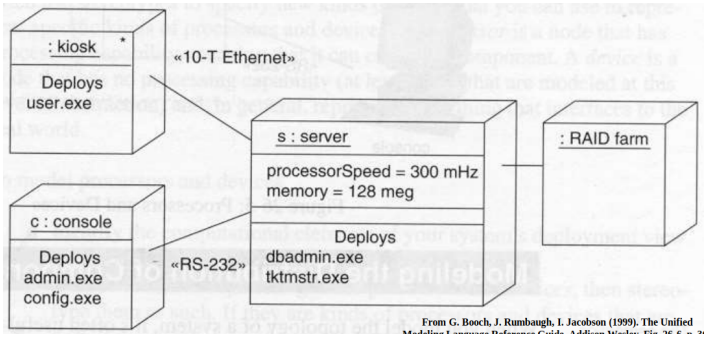


Componenti e nodi

I componenti sono simili ai nodi (hanno nomi, partecipano nelle relazioni di dipendenza, generalizzazione e associazione, possono essere annidati, possono avere istanze, possono partecipare alle interazioni, ...).
Comunque, i componenti sono cose che partecipano all'esecuzione di un sistema, i nodi sono entità che eseguono i componenti. I componenti rappresentano il packaging fisico di elementi altrimenti logici, i nodi rappresentano la dislocazione fisica dei componenti.

Un diagramma di deployment con nodi che contengono componenti

Un diagramma di deployment mostra dove vado a mettere i nodi.



Analisi e progettazione di sistemi Real Time

La caratteristica fondamentale dei sistemi Real Time è il determinismo. In questi sistemi la correttezza dell'esecuzione non dipende solo dal risultato, ma anche dal tempo.

Real Time in UML

Lavorando con UML dobbiamo considerare due diversi aspetti:
- Le feature di UML che possono essere usate per supportare applicazioni Real Time.
- Le estensioni di UML proposte che gestiscono applicazioni Real Time -> le feature auto-estendenti di UML possono essere usate per questo scopo.

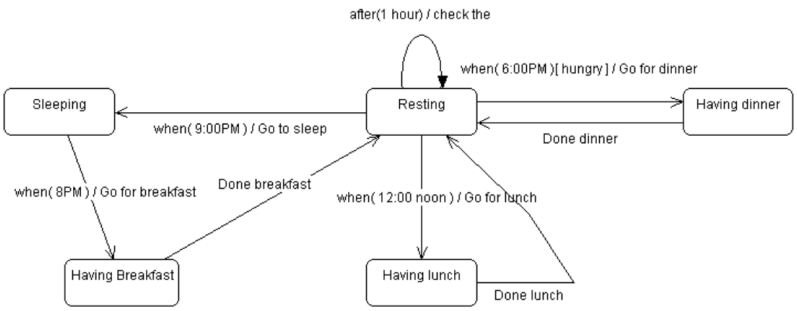
Costrutti Real Time

UML ha delle caratteristiche per supportare applicazioni Real Time:
- Evento di cambio
- Evento di tempo
- Vincoli di tempo

Eventi di tempo e cambio

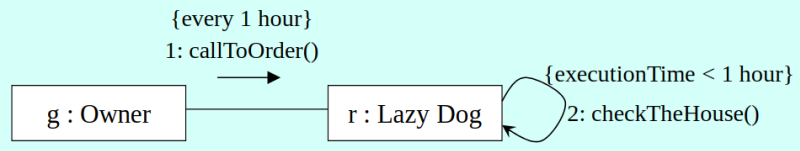
Gli eventi di tempo e cambio specificano quando un'azione è da effettuare ad un preciso istante di tempo.

- **when** è un evento di cambio che specifica data e ora in cui l'evento avverrà.
    - when (12:00 AM) / goForLunch()
- **after** è un evento di cambio che specifica dopo quanto tempo un evento sarà eseguito.
    - after (5 minutes) / put the egg in the pot



Vincoli di tempo

I vincoli di tempo possono essere aggiunti ai messaggi.
Ci sono tre variabili built-in che possono essere usate: startTime, stopTime e executionTime.
Il costrutto "every" identifica un messaggio che viene reinviato a intervalli di tempo dati.



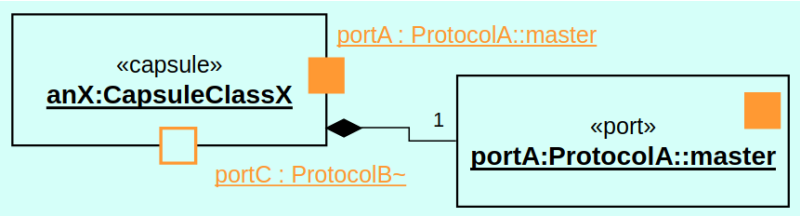
Sfortunatamente non esistono strumenti o linguaggi di programmazione che facciano rispettare tali vincoli.

Esercizio proposto

Model the problem of calling the taxi company to get a taxi to the airport in time to get the plane.
Start from the taxi class diagram of last week.

ROOM: Realtime Object Oriented Modeling

Notazione abbreviata per istanze di capsule, forma iconificata.

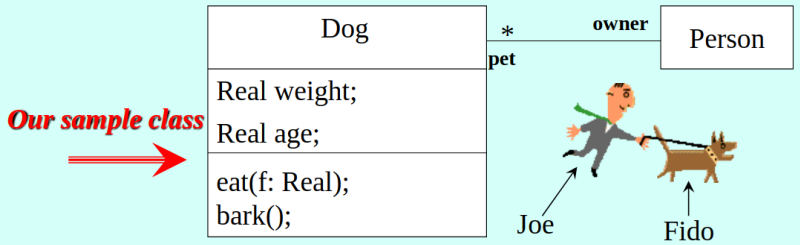


Object Constraint Language - OCL

OCL è il linguaggio di espressione per UML.
- È un linguaggio puro, non ha side effect.
- È un linguaggio di modellazione, non può essere eseguito.
- È un linguaggio formale, ha una semantica ben definita.

Scopo di OCL

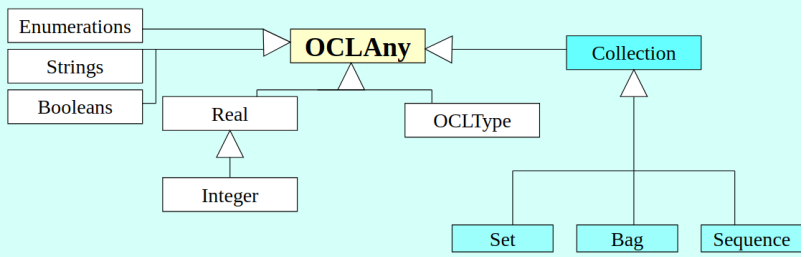
Lo scopo è di definire in modo chiaro e non ambiguo le guardie in UML.



I vincoli in Java sono gestiti con le eccezioni (RuntimeException, così non c'è obbligo di gestione).

Feature principali di OCL

- Gli attributi hanno la stessa sintassi di Java (con *self* al posto di *this*).
    - es. self.weight < 100
- Le espressioni sono fortemente tipate e viene controllata la conformità dei tipi.
    - es. self.weight < "abc" NO!
- Si possono gestire precondizioni e postcondizioni, ovvero cosa succede prima e dopo che un metodo venga chiamato. Esse sono assegnate alle operazioni e si implementano usando le annotazioni di assert (@BeforeEach, @AfterAll, ...).
    - es. Dog::bark()
      - pre: neighborhood is happy
      - post: neighborhood is angry
- Le associazioni di cardinalità 1 sono trattate come attributi.
    - es. Fido.owner = Joe
- Le associazioni di cardinalità diversa da 1 sono trattate come insiemi e hanno operazioni ben definite.
  `.` e `->` possono essere usati polimorficamente.
    - es. Joe->pet è un insieme di tutti i *pets* di Joe.



Collezioni di metodi

Possiamo avere collezioni di metodi in OCL.

- size
- includes
- count
- includesAll
- isEmpty
- notEmpy
- sum
- exists
- forAll
- iterate
- select
- reject
- collect

Introduzione al refactoring

I sistemi software sono entità in evoluzione che richiedono costante aggiornamento e mantenimento. Abbiamo bisogno di definire modi per rendere l'aggiornamento ed il mantenimento "semplici" e non "orrendamente difficoltosi".
La soluzione è lo sviluppo incrementale con cicli di vita brevi e un costante **refactoring**.

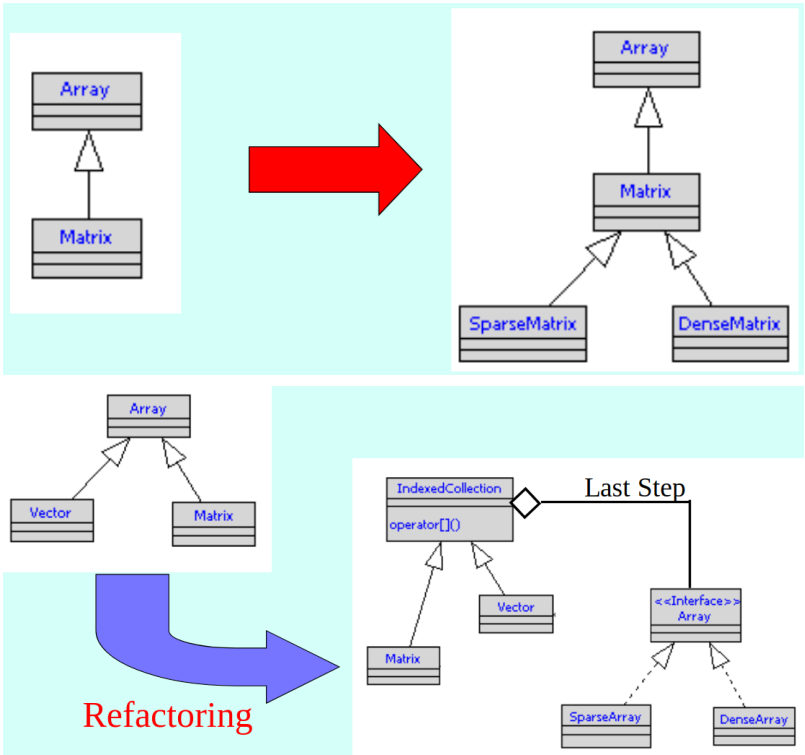
Idea chiave

L'idea del refactoring è fare operazioni sul codice che modifichino la sua struttura mantenendone invariato il comportamento, ovvero non cambiare la funzionalità, bensì cambiare la facilità di lettura del codice. Il refactoring è importante soprattutto quando si fa sviluppo incrementale: si cerca di rendere il codice sempre più semplice per favorire il riuso.

Refactoring di diagrammi UML

Dobbiamo revisionare il nostro sistema, ristrutturarne la progettazione sforzandoci di raggiungere semplicità e una maggiore aderenza alla struttura del dominio, mantenendo il suo comportamento.

Esempio



Tecniche di refactoring

Le operazioni più frequenti di refactoring sono legate alle classi.
Es: *Break Down = dividere una classe troppo grande in classi più piccole.*
Altro es: *Push Up/Down = spostare un campo o metodo nella classe superiore o inferiore nella gerarchia.*

	Class	(Static) (member) data	(Static) (member) function	Temp. data
Add				
Delete				
Rename				
Push Up/Down				
Break Down / Condense				
Re-qualify				

Design pattern e refactoring

I design pattern possono essere istruttivi per il refactoring:
- Suggeriscono la struttura interna del codice.
- Guidano lo sviluppo di nuove strutture di oggetti e classi.
- Aiutano a separare le strategie di implementazione dallo scopo dell'implementazione.

Guida al refactoring con design pattern

Creazione consistente tra classi: **Abstract factory**.
Creazione dipendente da bisogni ambientali: **Factory method**.
Creazione basata su un'entità prototipo: **Prototype**.
Numero di feature variabile: **Decorator**.
Struttura ad albero: **Composite**.
Doppio dispatching: **Visitor**.
Implementazione flessibile di un algoritmo: **Strategy**.
Implementazione flessibile di una classe: **Bridge**.

Criteri per il refactoring

Rifattorizzare quando il programma mostra carenze e non per "miglioramento concettuale".
Scrivere il codice esattamente una volta (usare la relazione di include).
Eliminare entità enormi (metodi, classi, packages, ...).

Esercizi proposti

Taxi drivers

Develop the remaining OO Design Diagrams for the system supporting the reservation and scheduling for taxi drivers that was discussed in the first day of the course. Then think at how you could refactor the system developed so far.

Books

Develop the remaining OO Design Diagrams for the system that manages the search, the selection, and the purchase of books at Chapters (http://www.chapters.ca) that was discussed in the first day of the course. Then think at how you could refactor the system developed so far.

Stocks Trading Service

Develop the remaining OO Diagrams for:
By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the ser to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a articular stock is no longer available.
Then refactor.

Network Printing Service

Develop the remaining OO Diagrams for:
There is a super high-resolution color laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc...). In addition, users can monitor their own print jobs and
delay or delete jobs as they see fit.
To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.
Then refactor.

Component Brokerage System

Develop the remaining OO Diagrams for:
This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.
Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.
Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.
Then refactor.

Bug Tracking System

Develop the remaining OO Diagrams for:
Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.
Then refactor.

Esercizio comprensivo finale

Define a system to support all the features (calling, billing, roaming, + the extra gadgets of your choice) of a new form of cellular service.
- While at home or within 10 meters of your home, same charges as the local phone.
- While in the city limits, only charge for airtime.
- Within Canada, roaming access from the local provider that set himself the charges + airtime.
- Outside Canada, no service in place.

Design patterns

Introduzione

I **design pattern** sono delle entità che definiscono uno scheletro di soluzione ad un problema generale ripetuto.
Sono descrizioni di oggetti e classi comunicanti, personalizzate per risolvere un problema di progettazione generale in un contesto particolare.
Es. *costruire edifici circolari:*

Definizione

"Un pattern descrive un problema che si presenta numerose volte nel nostro ambiente, e descrive il nucleo della soluzione a quel problema in modo che si possa usare tale soluzione un altro milione di volte, senza mai doverla fare due volte alla stessa maniera." *(Alexander et. al., 1977)*

Design pattern nello sviluppo

Possiamo tradurre il concetto di design pattern nello sviluppo software. Dobbiamo definire:
- I "mattoni"
- La "configurazione dei mattoni"
L'Object Orientation fornisce un modo naturale per esprimere i design pattern: gli oggetti della progettazione sono i nostri "mattoni". Informalmente, un design pattern è una particolare "configurazione" di tali oggetti: un insieme di oggetti e delle loro relazioni (ereditarietà, composizione, aggregazione, associazione, creazione, ...).
I design pattern OO hanno un potenziale eccellente per essere i componenti giusti per il riuso.

Composizione di un design pattern

Un design pattern è composto di quattro elementi:

- Il **nome del pattern**: è usato per descrivere un problema, la sua soluzione e le sue conseguenze in una o due parole.
- Il **problema**: descrive un particolare problema di progettazione e il suo contesto.
- La **soluzione**: descrive gli elementi di progettazione, le loro relazioni, le loro responsabilità e collaborazioni.
- Le **conseguenze**: sono il risultato ed i trade-off dell'applicare il design pattern.

L'approccio Gamma

Gamma distingue 3 tipi di pattern:
- **Creazionali**: pattern che hanno a che fare con la creazione degli oggetti.
- **Strutturali**: pattern che hanno a che fare con la composizione di classi e oggetti.
- **Comportamentali**: pattern che hanno a che fare con l'interazione tra oggetti e la condivisione delle responsabilità.

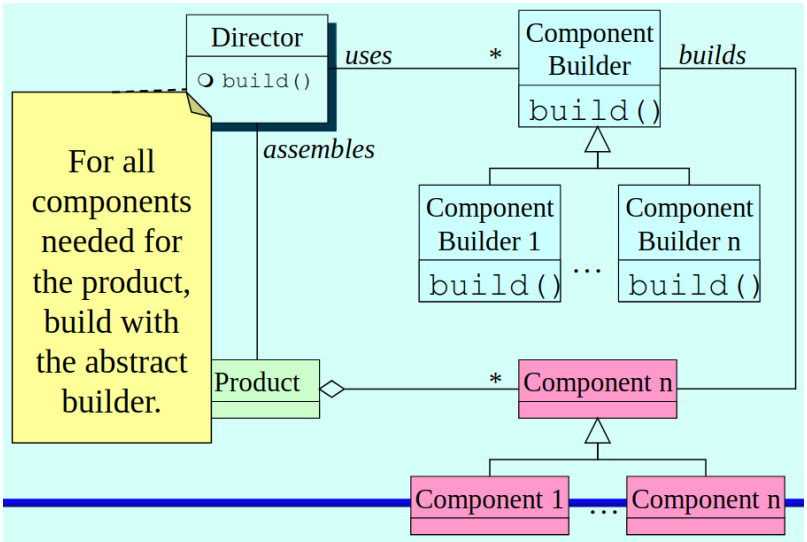
Pattern creazionali

Questi pattern sono legati alla creazione degli oggetti. Astraggono l'istanziazione dell'oggetto. Incapsulano la conoscenza delle classi concrete e nascondono le informazioni riguardanti la creazione dell'oggetto.
Servono quando gli oggetti sono complessi o ci sono difficili relazioni da mantenere.
Cinque pattern creazionali sono: **Abstract factory**, **Builder**, **Factory method**, **Prototype** e **Singleton**.

Builder pattern

Questo pattern è usato per creare un oggetto complesso separando il suo processo di costruzione dalla sua rappresentazione.
Il processo di costruzione è delegato ad un direttore della costruzione degli oggetti. Il direttore mantiene una lista degli oggetti complessi da creare e dirige il processo di costruzione verso il componente costruttore appropriato.
Ci consente di avere diverse implementazioni e/o interfacce delle parti di un oggetto e di avere un controllo più preciso sul processo di costruzione.
Struttura di base:
- **Builder**: una classe separata che fornisce metodi per configurare le proprietà dell'oggetto. Questi metodi di solito restituiscono il Builder stesso.
- **Product**: l'oggetto complesso che deve essere costruito.
- **Director** (opzionale): una classe che utilizza il Builder per costruire un oggetto.

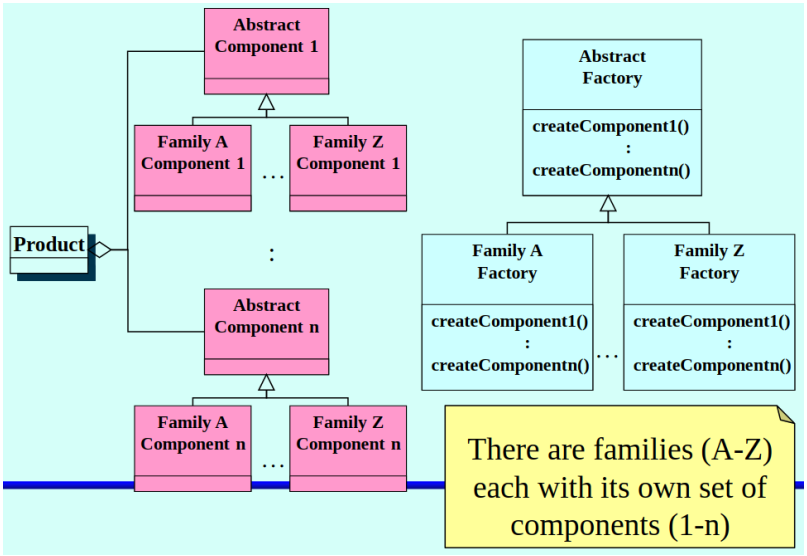
Il direttore utilizza un insieme di Builder, organizzati in una gerarchia. Il prodotto ha un riferimento al direttore e n componenti, ognuno dei quali ha associato un Builder. Per ogni componente del prodotto si chiama il direttore, che sceglie il Builder appropriato, il quale richiama la `build()`.



```
class Car {
  private int wheels;
  private String color;

  // Costruttore privato per evitare l'uso diretto
  private Car(Builder builder) {
    this.wheels = builder.wheels;
    this.color = builder.color;
  }

  // Classe Builder statica
  public static class Builder {
    private int wheels;
    private String color;

    public Builder setWheels(int wheels) { ... }

    public Builder setColor(String color) { ... }

    public Car build() {
      return new Car(this); // Costruisce l'oggetto Car
    }
  }
}

public static void main(String[] args) {
  Car car = new Car.Builder()
    .setWheels(4)
    .setColor("Red")
    .build();
}
```

Abstract factory

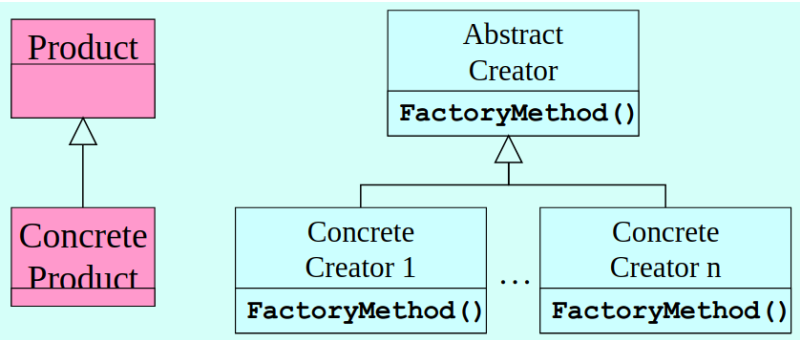
Fornisce un'interfaccia per creare famiglie di oggetti imparentati o dipendenti senza specificare le loro classi concrete.
Può essere usato quando c'è bisogno di avere famiglie multiple di prodotti, per nascondere le implementazioni del prodotto e presentare solo le interfacce.
Supporta la consistenza tra i prodotti e rende semplice cambiare le famiglie del prodotto. È molto difficoltoso supportare nuovi tipi di prodotto in ogni famiglia.
Invece di usare costruttori concreti per creare gli oggetti, si utilizza un'interfaccia che delega la creazione delle istanze a classi specifiche che implementano questa interfaccia. Struttura di base:
- **Abstract Factory**: un'interfaccia o classe astratta che dichiara un set di metodi per creare oggetti astratti.
- **Concrete Factory**: implementazioni dell'Abstract Factory che creano oggetti specifici di una famiglia.
- **Abstract Product**: interfacce o classi astratte che definiscono il tipo di oggetti da creare.
- **Concrete Product**: implementazioni concrete degli oggetti da creare.
Un prodotto può essere composto da diversi tipi di componenti che appartengono a famiglie diverse. Per ogni famiglia vado a specificare tutti i componenti possibili, ottenendo così un creatore di componenti di tale famiglia. Per usarlo istanzio prima la famiglia, poi la uso per creare i componenti. Dovendo specificare tutti i componenti per ogni famiglia, è difficoltosa l'estensione delle famiglie di prodotti.



```
AbstractFactory f;  // Abstract factory
AbstractComponent1 c1;  // Abstract product
AbstractComponent2 c2;
// We want component 1 from Family A.
f = new FamilyAFactory(); // FamilyAFactory (concrete factory)
implements AbstractFactory
c1 = f.createComponent1();  // concrete product
// We want component 2 from Family C.
f = new FamilyCFactory();
c2 = f.createComponent2();
```

Factory method

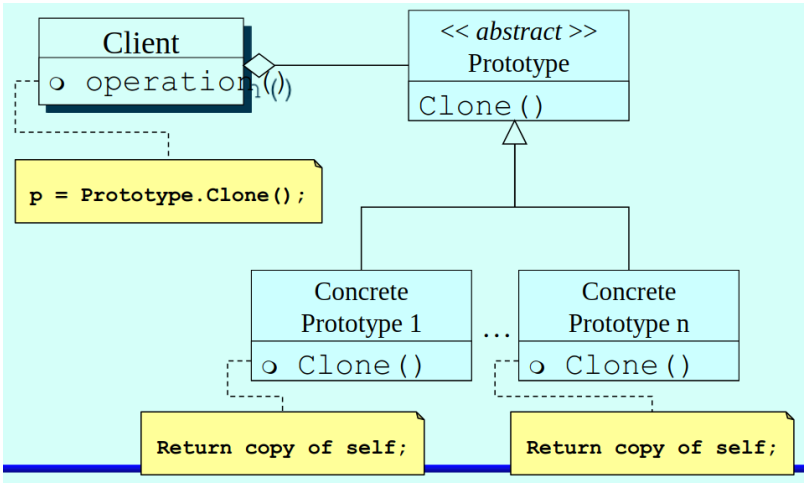
È una versione leggermente più semplice dell'Abstract factory, usato per creare un oggetto quando le informazioni necessarie per costruirlo sono disponibili solo a runtime (quindi non durante la compilazione).
Può essere usato quando una classe non può anticipare la classe dell'oggetto che deve creare.
In questo pattern possiamo creare un'interfaccia per creare un oggetto e lasciare che sia la sottoclasse a decidere quale classe istanziare. Questo pattern connette gerarchie di classi parallele. La situazione è meno coerente perché decido solo a runtime che cosa fare.
Struttura di base:
- **Product**: un'interfaccia o classe astratta che definisce l'interfaccia degli oggetti creati dal Factory Method.
- **Concrete Product**: implementazioni concrete dell'interfaccia Product.
- **Creator**: una classe astratta o un'interfaccia che dichiara il Factory Method.
- **Concrete Creator**: classi che estendono Creator e sovrascrivono il Factory Method per restituire un'istanza del Concrete Product.



```java
// Interfaccia Product: definisce il tipo di oggetti che il factory method creerà
interface Product { void use(); }

// Concrete Product: implementano Product
class ConcreteProductA implements Product {
  @Override
  public void use() { ... }
}

class ConcreteProductB implements Product {
  @Override
  public void use() { ... }
}

// Creator: classe astratta che dichiara il factory method
abstract class Creator {
  abstract Product factoryMethod();

  public void someOperation() {
    Product product = factoryMethod();
    product.use();
  }
}

// Concrete Creator: sovrascrivono il factory method per creare i Concrete Product
class ConcreteCreatorA extends Creator {
  @Override
  Product factoryMethod() {
    return new ConcreteProductA();
  }
}

class ConcreteCreatorB extends Creator {
  @Override
  Product factoryMethod() { ... }
}

public static void main(String[] args) {
  Creator creatorA = new ConcreteCreatorA();
  creatorA.someOperation();
  Creator creatorB = new ConcreteCreatorB();
  creatorB.someOperation();
}
```

Prototype pattern

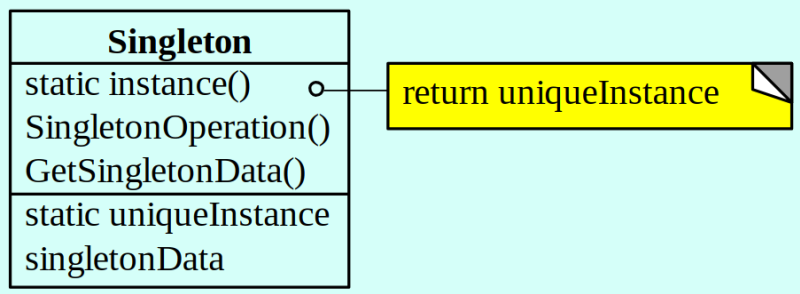
Permette all'utente di specificare i tipi degli oggetti da creare usando un'istanza prototipica, e di creare nuovi oggetti clonando il prototipo. Questo è utile quando il costo di creazione di un nuovo oggetto è elevato o quando il processo di costruzione è complesso.
Può essere usato per evitare la formazione di gerarchie di classi parallele utilizzanti il pattern Factory method.
Consente all'utente di aggiungere e rimuovere oggetti a runtime. Potrebbe ridurre il numero di sottoclassi.
Struttura di base:
- **Prototype**: un'interfaccia che dichiara il metodo `clone()` per duplicare gli oggetti.
- **Concrete Prototype**: classi che implementano il metodo `clone()`, restituendo un nuovo oggetto con gli stessi valori di quello originale.
Il metodo `clone()` è unico e viene via via raffinato nelle sottoclassi. Sfruttando il polimorfismo con il tipo Any (o Object), manteniamo una collezione di oggetti della classe gerarchicamente più alta e cambiamo nel codice il tipo dell'oggetto assegnandone uno più specifico. Per sapere di che tipo è l'oggetto in questione usiamo la reflection, con `getClass()`.




```
// mantengo tutti i prototipi
Vector<Object> vettorePrototipi = new Vector...
// un oggetto di un tipo generico
Object oggettoQualunque;
// assegno l'oggetto del tipo che mi interessa clonando il prototipo
oggettoQualunque = vettorePrototipi.elementAt(5).clone();
// reflection per sapere il tipo
oggettoQualunque.getClass();
```

Singleton pattern

È uno dei pattern più importanti, garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a quell'istanza. Utile quando un'istanza di una classe deve controllare risorse condivise.
Settiamo il costruttore privato per impedire la creazione di nuove istanze. Nella classe definiamo una singola istanza statica e privata. Definiamo poi un metodo statico e pubblico che ritorna l'oggetto se esiste, altrimenti lo crea e poi lo restituisce.
Questo pattern funziona grazie alla possibilità di impostare la visibilità public e private.




```
class Singleton {
  private static Singleton theInstance = null;  // singola istanza
  private Singleton() {}  // costruttore privato
  static Singleton getInstance() {
    if (theInstance == null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

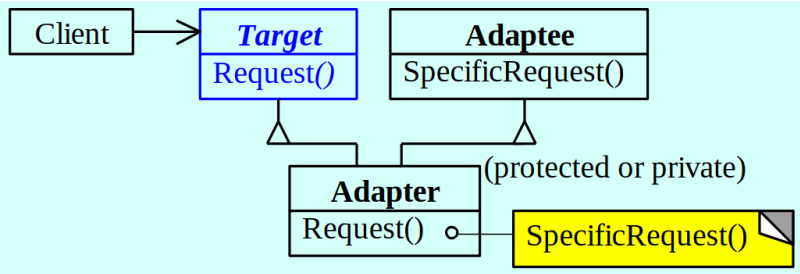
Pattern strutturali

Questi pattern hanno a che fare con come le strutture sono formate tramite la composizione di classi e oggetti.
Ci sono due tipi di pattern strutturali:
- Pattern strutturali **di classe**, che usano l'ereditarietà per comporre interfacce o implementazioni.
- Pattern strutturali **di oggetto**, che descrivono i modi di comporre oggetti per realizzare nuove funzionalità.
Sono meno importanti dei pattern creazionali, servono per disaccoppiare l'implementazione del codice dal suo accesso.
Esempi di pattern strutturali: **Adapter**, **Bridge**, Composite, **Decorator**, Facade, Flyweight, **Proxy**.

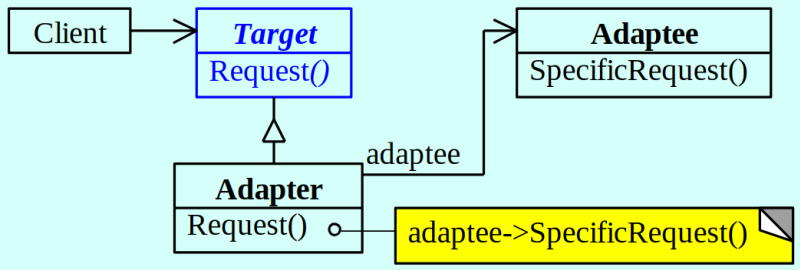
Adapter

L'Adapter permette a due classi con interfacce incompatibili di lavorare insieme: agisce come un ponte tra le classi.
È utile quando si vuole utilizzare una classe esistente ma la sua interfaccia non corrisponde a quella necessaria.
La classe più generale ha un riferimento alla classe con cui vogliamo comunicare. Creiamo poi una sottoclasse specializzata che ridefinisce il metodo con cui si effettua il collegamento.

Con ereditarietà multipla



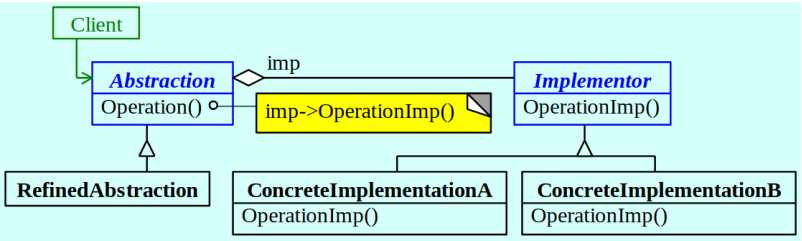
Senza ereditarietà multipla




```
// classe con cui comunicare
class Adaptee {
  Object specificRequest() { ... }
}

// classe adapter
class Adapter {
  Adaptee adaptee;

  Object request() {
    adaptee.specificRequest();
  }
}

// classe specializzata
class Target extends Adapter {
  @Override
  Object request() { ... }
}
```

Bridge

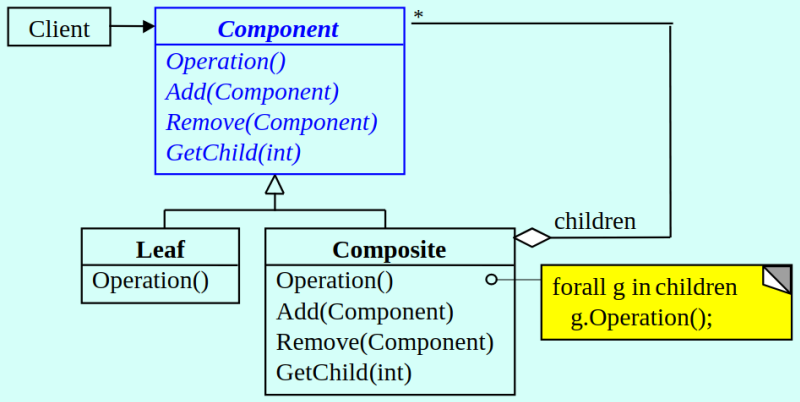
Il Bridge separa un'interfaccia dalla sua implementazione, permettendo ad entrambe di variare indipendentemente.
Questo pattern suddivide una classe in due gerarchie separate: una per l'astrazione (l'interfaccia) e una per l'implementazione. L'astrazione contiene un riferimento all'implementazione.
Implementor è l'interfaccia che definisce le operazioni di base che le **ConcreteImplementor** devono realizzare. **Abstraction** è una classe astratta che mantiene un riferimento a un oggetto Implementor e definisce l'interfaccia per l'uso del client. **RefinedAbstraction** estende Abstraction e può aggiungere comportamenti aggiuntivi o personalizzati.
Data la presenza delle interfacce in Java, il Bridge è meno critico a livello di codice dell'Adapter.



```java
// interfaccia che definisce l'implementazione
interface Implementor { void operationImp(); }

// implementazione concreta
class ConcreteImplementorA implements Implementor {
  @Override
  public void operationImp() { ... }
}

// interfaccia astratta che utilizza l'implementazione
abstract class Abstraction {
  Implementor implementor;

  public Abstraction(Implementor implementor) {
    this.implementor = implementor;
  }

  abstract void operation();
}

// estensione dell'astrazione
class RefinedAbstraction extends Abstraction {
  public RefinedAbstraction(Implementor implementor) {
    super(implementor);
  }

  @Override
  void operation() {
    implementor.operationImp();
  }
}

public static void main(String[] args) {
  // uso di ConcreteImplementorA
  Abstraction abstractionA = new RefinedAbstraction(new
ConcreteImplementorA());
  abstractionA.operation();
}
```

Composite pattern

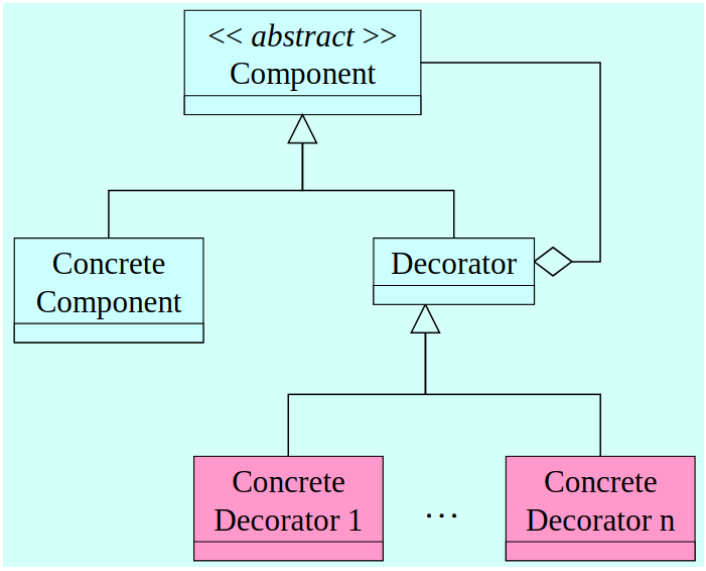
Il Composite permette di trattare oggetti singoli e composizioni di oggetti in modo uniforme. Questo pattern è utile quando si ha una struttura gerarchica di elementi, come un albero, e si vuole gestire sia gli oggetti semplici sia quelli complessi usando la stessa interfaccia.
Struttura di base:
- **Component**: interfaccia o classe astratta che definisce i metodi comuni per gli oggetti.
- **Leaf**: oggetto semplice che non ha figli.
- **Composite**: oggetto complesso che può contenere altri oggetti Component.

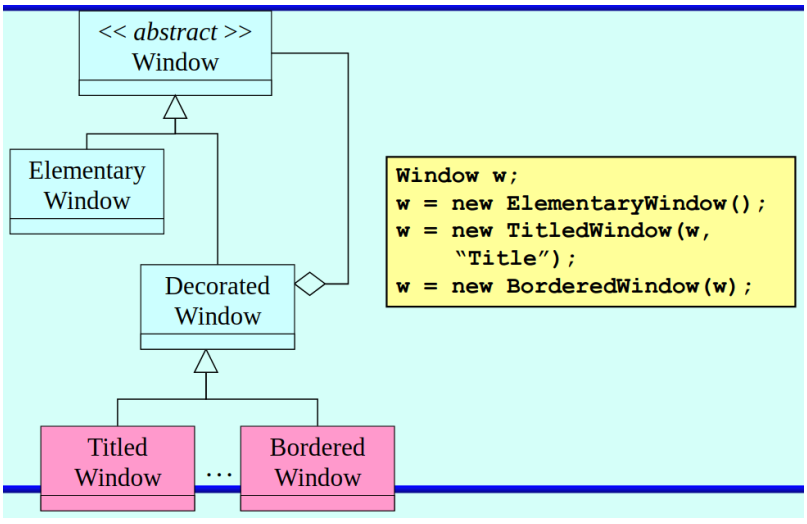



```java
// interfaccia comune
interface Component { void operation(); }

// oggetto semplice
class Leaf implements Component {
  @Override
  public void operation() { ... }
}

// oggetto composto
class Composite implements Component {
  private List<Component> children = new ArrayList<>();

  public void add(Component c) {
    children.add(c);
  }

  public void remove(Component c) {
    children.remove(c);
  }

  @Override
  public void operation() { ... }
}
```

Decorator pattern

L'idea è di attribuire responsabilità aggiuntive a un oggetto dinamicamente, in modo flessibile, così che l'oggetto possa essere esteso senza usare sottoclassi.
È utile quando dobbiamo definire una classe di oggetti con un insieme variabile di proprietà.
Abbiamo un componente e un decoratore, il quale può essere di tipi diversi. In questo modo il componente può scegliere concretamente la specializzazione. Un decoratore e i suoi componenti non sono identici.
L'oggetto decorato contiene al suo interno l'oggetto originario. Bisogna fare attenzione perché il nuovo oggetto non ha riferimenti, quindi lo creo e poi lo butto via. Un oggetto può essere decorato più volte.
Struttura di base:
- **Component**: definisce l'interfaccia per gli oggetti che possono essere decorati.
- **ConcreteComponent**: implementazione di Component che rappresenta l'oggetto di base da decorare.
- **Decorator**: classe astratta che serve come base per tutti i decoratori concreti.
- **ConcreteDecorator**: classe che estende Decorator e aggiunge nuove funzionalità.

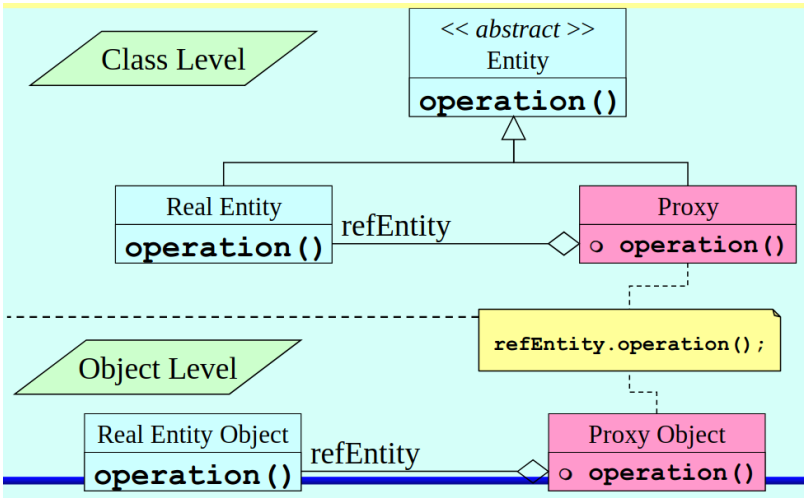




```java
// interfaccia o classe astratta per l'oggetto base
interface Component { void operation(); }

// implementazione concreta del componente
class ConcreteComponent implements Component {
  @Override
  public void operation() { ... }
}

// classe astratta che implementa Component
abstract class Decorator implements Component {
  Component component;

  @Override
  public void operation() {
    component.operation();
  }
}

// decoratore concreto che aggiunge funzionalità
class ConcreteDecoratorA extends Decorator {
  @Override
  public void operation() { // add functionalities
    super.operation();
  }
}

public static void main(String[] args) {
  Component component = new ConcreteComponent();
  component = new ConcreteDecoratorA(component);
  component.operation();
}
```

Proxy pattern

L'implementazione effettiva è nascosta nell'oggetto reale e un oggetto proxy viene usato per la presentazione. I proxy possono essere utilizzati per accesso remoto, accesso virtuale e protezione.
L'idea di fondo è di accedere ad un'entità remota come se fosse in locale, creando un'istanza locale che si occupa della comunicazione con l'estremo remoto e che può aggiungere funzionalità.
Il pattern proxy può avvenire sia a livello di classe che di oggetto.



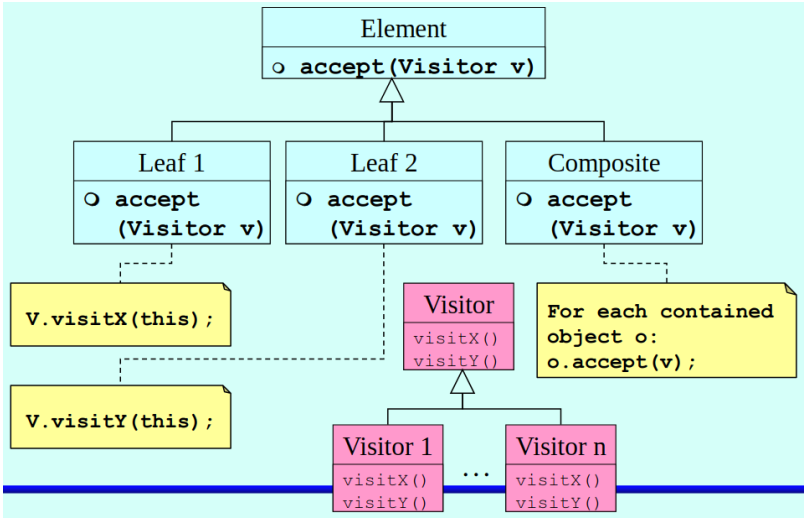

```java
// interfaccia comune per il RealSubject e il Proxy
interface Subject { void request(); }

// implementazione dell'oggetto reale
class RealSubject implements Subject {
  @Override
  public void request() { ... }
}

// classe che controlla l'accesso al RealSubject
class Proxy implements Subject {
  private RealSubject realSubject;

  @Override
  public void request() {
    if (realSubject == null) {
      realSubject = new RealSubject();
    }
    // Logica aggiuntiva (controllo accesso, caching, ...)
    realSubject.request();
  }
}
```

Pattern comportamentali

Questi pattern hanno a che fare con gli algoritmi e l'assegnamento di responsabilità tra oggetti. Essi descrivono i modelli di interazione degli oggetti, e caratterizzano controlli di flusso complessi che sono difficili da seguire a runtime.
Alcuni pattern comportamentali sono: **Visitor**, **Strategy**, **Chain of responsibility**, **Mediator**, State, Command, Interpreter, Iterator, Memento, Observer, Template Method.

Visitor pattern

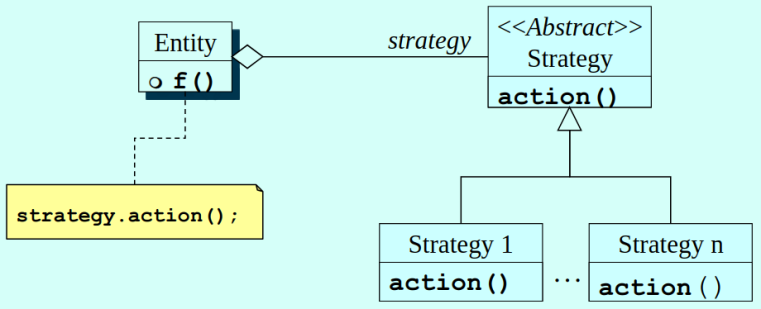
Il Visitor definisce un'operazione da effettuare sugli elementi di una struttura di oggetti, consentendo di separare un algoritmo dalla struttura su cui opera. Il Visitor permette di avere nuove operazioni senza cambiare le classi degli elementi su cui lavora. Potrebbe forzare una rottura nell'incapsulamento.
Ho un elemento che può essere foglia oppure composite, il quale accetta un Visitor. L'operazione viene effettuata sulla foglia oppure replicata su tutti gli elementi del composite. L'operazione fatta concretamente è quindi definita dall'oggetto foglia.
Struttura di base:
- **Visitor**: definisce il metodo `visit` per ciascun tipo di elemento nella struttura.
- **Element**: interfaccia o classe astratta che definisce il metodo `accept`.
- **ConcreteElement**: oggetto concreto che implementa Element e accetta un Visitor.



```
// implementa la logica specifica per ogni tipo di elemento
class Visitor {
  public void visit(ConcreteElementA e) { ... }

  public void visit(ConcreteElementB e) { ... }
}

// interfaccia che definisce il metodo accept
interface Element {
  void accept(Visitor v);
}

// ConcreteElements: implementano il metodo accept per il Visitor
class ConcreteElementA implements Element {
  @Override
  public void accept(Visitor v) {
    v.visit(this);
  }
}

class ConcreteElementB implements Element {
  @Override
  public void accept(Visitor v) {
    v.visit(this);
  }
}

public static void main(String[] args) {
  List<Element> elements = ...
  Visitor visitor = new Visitor();

  for (Element e : elements) {
    e.accept(visitor);
  }
}
```

Strategy pattern

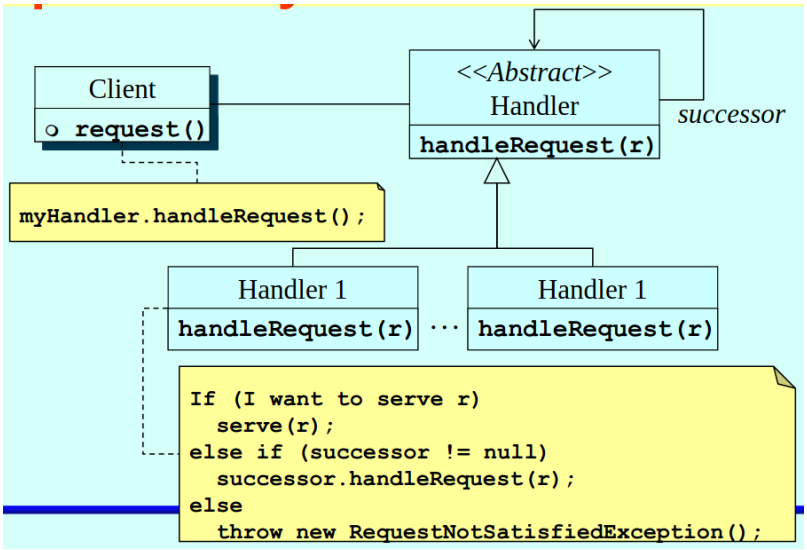
Strategy permette di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Questo pattern consente di selezionare l'algoritmo da utilizzare in modo dinamico a runtime, evitando di usare costrutti condizionali per decidere quale logica eseguire. Consente di avere scelte di implementazione ed un numero di sottoclassi ridotto.
Utile soprattutto quando abbiamo bisogno di numerose classi imparentate che differiscono solo nel loro comportamento.
Abbiamo un'entità su cui vogliamo applicare un algoritmo (ad esempio un array da ordinare), scelto tra diverse opzioni. Per determinare l'operazione concreta da effettuare, passiamo un riferimento all'oggetto che incapsula il comportamento, il quale contiene la funzione da chiamare.
Gestire con cautela in quanto è alla base della dependency injection.
Struttura di base:
- **Strategy**: interfaccia comune per tutti gli algoritmi.
- **ConcreteStrategy**: classi che forniscono un'implementazione dell'algoritmo.
- **Context**: classe che utilizza un oggetto Strategy e delega a esso la chiamata dell'algoritmo.




```
// interfaccia comune per tutte le strategie
interface Strategy { void execute(); }

// implementazione concreta dell'algoritmo A
class ConcreteStrategyA implements Strategy {
  @Override
  public void execute() { ... }
}

// utilizza una strategia per eseguire un'operazione
class Context {
  private Strategy strategy;

  public void setStrategy(Strategy s) { ... }

  public void executeStrategy() {
    if (strategy != null) strategy.execute();
  }
}
```

Chain of responsibility pattern

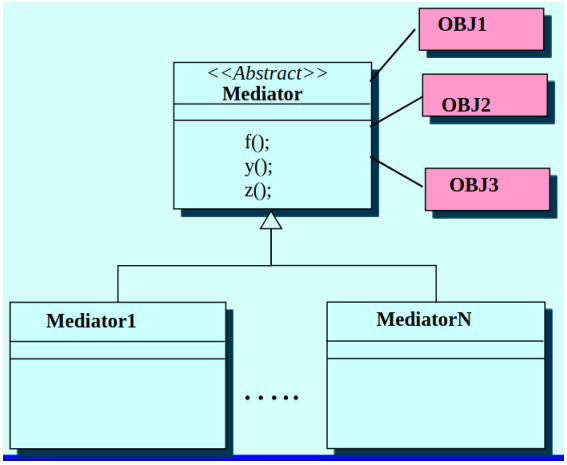
Può essere usato quando abbiamo una richiesta che deve essere soddisfatta da uno di tanti oggetti, ma non sappiamo in anticipo quale andrà a gestire la richiesta. Evita l'accoppiamento del mittente di una richiesta al suo destinatario. La ricezione della richiesta non è garantita.
Leghiamo a catena gli oggetti handler e passiamo la richiesta lungo la catena di oggetti fino a che uno di questi non la gestisce. Se non esiste nessun gestore si lancia un'eccezione (RuntimeException).
Es. *nel sistema Android il meccanismo degli Intent: diverse app registrate per poter servire un'azione.*




```java
// interfaccia per gestire le richieste
abstract class Handler {
  Handler nextHandler;

  public abstract void handleRequest(String request);
}

// gestisce la richiesta o la passa al successivo
class ConcreteHandlerA extends Handler {
  @Override
  public void handleRequest(String request) {
    if (request.equals("A")) {
      // gestione
    } else {
      // passaggio
      nextHandler.handleRequest(request);
    }
  }
}
```

Mediator pattern

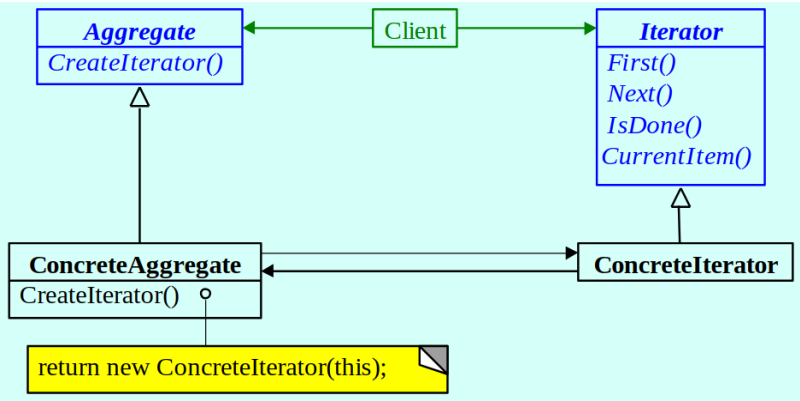
Utilizzato quando abbiamo interazioni complesse tra oggetti e non vogliamo gestirle all'interno dell'oggetto stesso.
Tutti gli oggetti sono memorizzati in uno spazio comune: il concetto del mediatore è simile ad una lavagna usata da tanti oggetti per condividere conoscenza e centralizzare il controllo.
Il Mediator ha un insieme fisso di primitive che permettono di rendere gli oggetti comunicanti. È richiesto che ogni partecipante conosca tali primitive per partecipare alla discussione.




```
// gestisce e coordina la comunicazione tra i colleghi
class Mediator {
  private List<Colleague> colleagues;

  public void addColleague(Colleague c) { ... }

  public void send(String msg, Colleague c) { ... }
}

// classe base per gli oggetti che comunicano tramite il mediatore
abstract class Colleague {
  protected Mediator mediator;

  public Colleague(Mediator m) { ... }
}

// collega concreto che comunica tramite il mediatore
class ColleagueA extends Colleague {
  public ColleagueA(Mediator m) { super(m); }

  public void send(String msg) { ... }

  public void receive(String msg) { ... }
}
```

Iterator pattern

L'Iterator fornisce un modo per accedere sequenzialmente agli elementi di una collezione senza esporre la sua rappresentazione interna. L'idea principale è separare la logica di iterazione dalla struttura della collezione.
Struttura di base:
- **Iterator**: interfaccia che definisce i metodi per iterare sugli elementi, come `hasNext()` e `next()`.
- **ConcreteIterator**: implementa Iterator e mantiene lo stato della posizione corrente nella collezione.
- **Aggregate**: interfaccia o classe che definisce un metodo per creare un iteratore.
- **ConcreteAggregate**: collezione concreta che implementa il metodo per restituire un iteratore.



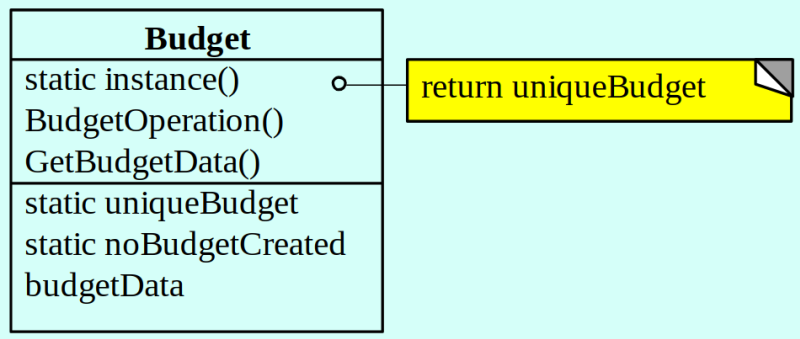
```java
// interfaccia per iterare sugli elementi
interface Iterator<T> {
  boolean hasNext();
  T next();
}

// implementa l'iterazione sugli elementi di una collezione
class ConcreteIterator<T> implements Iterator<T> {
  private List<T> collection;
  private int index = 0;

  public ConcreteIterator(List<T> collection) {
    this.collection = collection;
  }

  @Override
  public boolean hasNext() {
    return index < collection.size();
  }

  @Override
  public T next() {
    if (hasNext()) {
      return collection.get(index++);
    }
    return null;
  }
}

// interfaccia per creare un iteratore
interface Aggregate<T> {
  Iterator<T> createIterator();
}

// collezione concreta che restituisce un iteratore
class ConcreteAggregate<T> implements Aggregate<T> {
  private List<T> items = ...

  public void addItem(T item) { ... }

  @Override
  public Iterator<T> createIterator() {
    return new ConcreteIterator<>(items);
  }
}
```

Esempio tipico

Vogliamo progettare un **sistema di contabilità** per una piccola cittadina.
Esiste un **budget** composto da numerosi **conti** e il sistema dovrebbe essere in grado di ottenere le informazioni aggregate da questi conti.
Ci concentriamo sulla creazione e sull'analisi della struttura, non sul modificarla.
Requisiti
Il budget deve essere unico.
Numerosi conti possono essere aggiunti o rimossi dal budget, ogni conto può essere monolitico oppure formato da altri conti.
Deve essere possibile scorrere attraverso tutti i "conti esterni" appartenenti al budget.

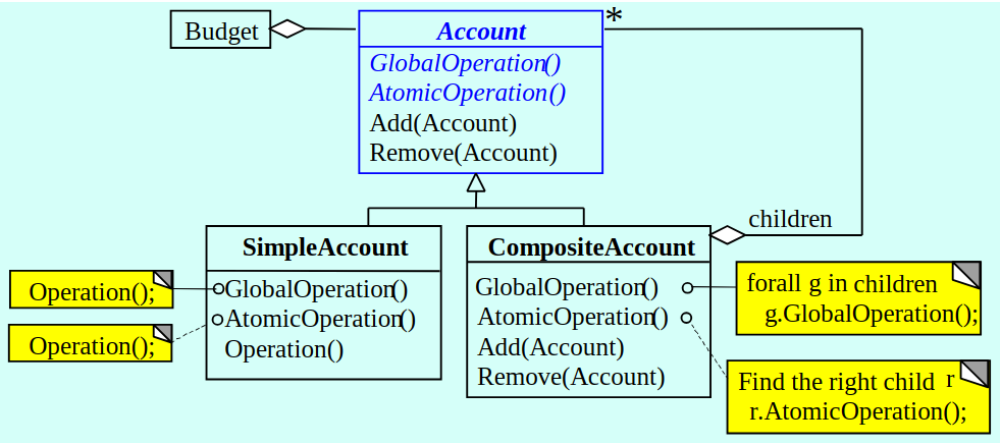
Soluzione

Unicità del budget

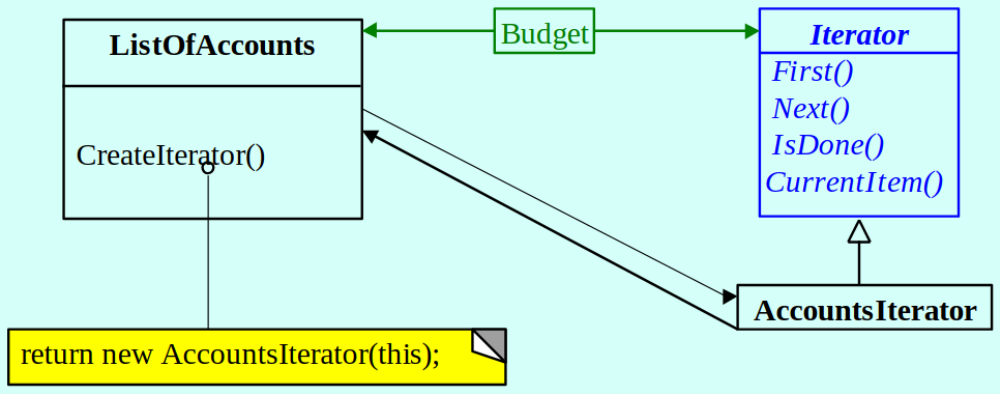



```
public class Budget { // Singleton
  public static Budget instance() {
    if (uniqueBudget == null)
      uniqueBudget=new Budget();
    return uniqueBudget;
  }
  ...
  private Budget() { ... }
  ...
  private static Budget uniqueBudget = null;
  ...
}
...
Budget townshipBudget = Budget.instance();
// Budget wrongBudget = new Budget(); WRONG!
```

Struttura dei conti



Scorrere attraverso i conti



Esercizi proposti

Imagine that you are the designer for a Windows-like operating system.
Think of features of the operating system that will benefit from the use of design patterns.

Find all the possible (... well ... at least one ...) patterns present in the O/S that you currently use.

Architetture software

Introduzione

Cerchiamo di capire come le diverse parti del sistema sono costruite. Durante la fase di progettazione di sistemi ci si occupa di pianificare come costruire il sistema e di decidere la struttura (architettura) del sistema.
Nel modello procedurale l'attività di progettazione è eseguita in sequenza, dopo l'analisi e prima di scrivere codice. Nel modello agile l'attività di progettazione è eseguita in parallelo, quindi durante l'analisi e la scrittura di codice.
EVITARE l'hacking in code, ovvero scrivere codice senza progettazione, la quale viene fatta successivamente ad-hoc. Il problema è che diventa difficile fare modifiche e miglioramenti successivamente.

Architettura del sistema

Partizionare il sistema in sotto-sistemi. La **progettazione architetturale** è l'attività di identificazione e definizione di sotto-sistemi. I sotto-sistemi sono definiti a vari livelli:
- Hardware
- Software di sistema (es. *OS, database management system, internet server*)
- Software di applicazione (es. *moduli, le loro feature ed interfacce*)

Architetture hardware

La parte più semplice è sicuramente quella hardware.

- Macchina singola
- Mainframe-based: il mainframe (singolo computer) è acceduto dagli utenti attraverso terminali connessi.
- Rete di computer: client e server.
- Embedded: processori direttamente collegati a sensori ed attuatori.
- Parallelo: molti processori che effettuano enormi computazioni in parallelo.

Architettura di sistemi software

Il mondo dei sistemi architetturali è stato fortemente influenzato dai design pattern. Mentre i design pattern tornano utili quando si scrive codice, gli stili architetturali servono per capire l'organizzazione strutturale del sistema (non vengono usati giornalmente).
Garland e Shaw, cercando di creare dei "pattern" per le architetture, hanno identificato nel 1994 alcuni stili architetturali principali, che danno informazioni su come i componenti sono connessi tra di loro. Essi includono: pipe and filters, data abstractions and OO organization, event based implicit invocation, layered systems, repositories, table driven interpreters. Questi possono essere ritrovati in un grosso numero di applicazioni.
In UML possono essere rappresentati in molti modi, incluso il diagramma di deployment.

Stili architetturali

Definizione: "uno stile architetturale definisce una famiglia di sistemi in termini del modello di organizzazione strutturale. Più specificamente, uno stile architetturale definisce un insieme di componenti e tipi di connettori, ed un insieme di vincoli su come essi possono essere combinati". *(Shaw and Garland, 1996)*

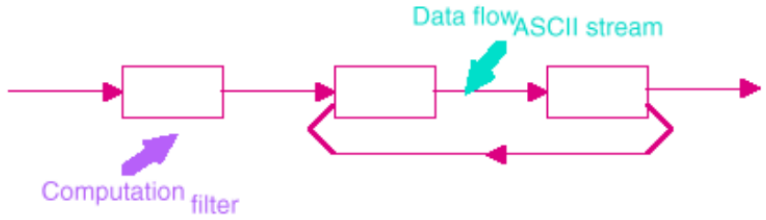
Stili architetturali comuni

Shaw e Garland identificano sette stili architetturali comuni:
- Pipe and filters
- Data abstraction and OO organization
- Layering
- Event-based implicit invocation
- Repositories
- Interpreters
- Process control

Pipes and filters

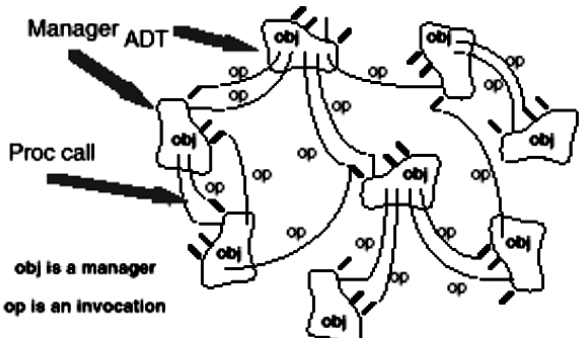
Ogni componente ha un insieme di input e output. Il componente legge flussi di dati in input ed applica trasformazioni locali in modo incrementale. L'output comincia prima che l'input sia stato consumato interamente.
I componenti sono chiamati **filters**, i connettori **pipes**.
I filters devono essere entità indipendenti:
- Non dovrebbero condividere lo stato con altri filters.
- Non dovrebbero conoscere l'identità dei filters superiori ed inferiori.

Nel mondo UNIX è molto usato.



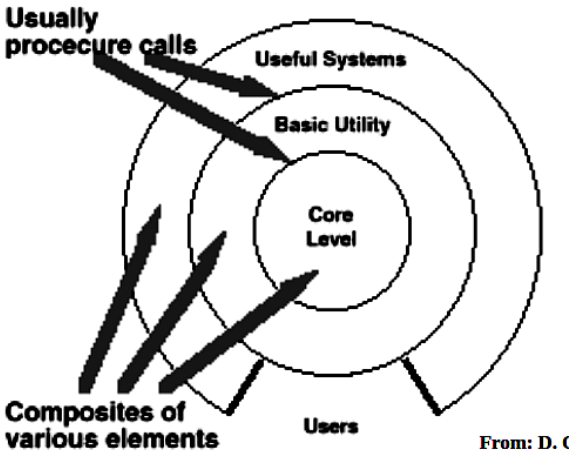
Data abstraction and OO organization

Rappresentazione dei dati catturata come Abstract Data Type. Un ADT (o oggetto) rappresenta un componente "manager":
- È responsabile di preservare l'integrità di una risorsa.
- Nasconde le rappresentazioni da altri oggetti.
Rappresentiamo il mondo tramite oggetti che interagiscono.



Layered systems

Struttura a livelli, come quella del sistema operativo.

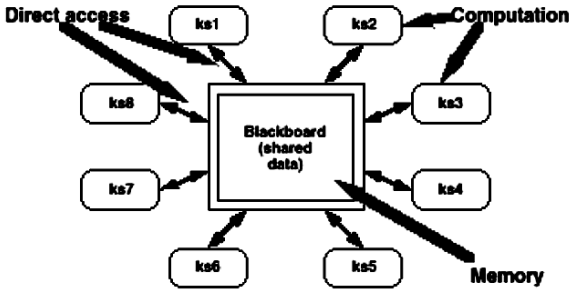


Event-based, Implicit invocation

Stile che ha storicamente origine nei sistemi basati su attori, soddisfazione di vincoli, demoni (daemons) e reti a commutazione di pacchetto.
Le interfacce dei componenti presentano un insieme di procedure ed un insieme di eventi. Gli annunciatori degli eventi non sanno chi reagirà: perciò gli eventi sono in broadcast.
Questo stile fornisce un forte supporto al riuso.
Un esempio sono i sistemi ad eventi, come *una finestra dove a seguito di un click viene eseguita una funzione*.

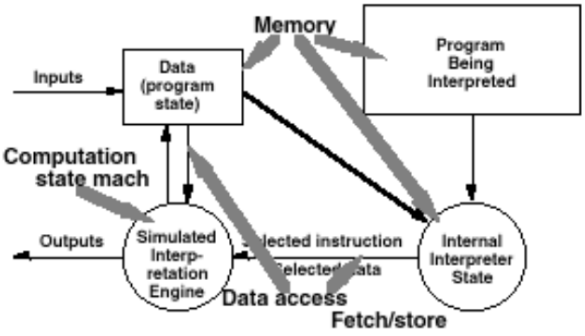
Repositories

Sono architetture basate sui dati memorizzati, che vengono poi manipolati. Due principali sotto-categorie:
- Database: i principali trigger sono i tipi di transazione.
- Architetture a lavagna: il principale trigger è lo stato corrente. Hanno tre parti principali: le risorse di conoscenza, la struttura dati lavagna ed il controllo. I dati sono messi al centro: su di essi sono definite delle regole e delle funzioni che li vanno a manipolare.



Interpreters

Una macchina virtuale è prodotta in software.
L'interprete è un componente intermedio che mi consente di lavorare su un'altra macchina, esso include:
- Pseudo-programma: include il programma ed il record di attivazione.
- Motore di interpretazione: include la definizione dell'interprete ed il suo stato di esecuzione attuale.
Ha quattro componenti: il motore di interpretazione, una memoria, la rappresentazione del controllo di stato e la rappresentazione dello stato attuale del programma che sta venendo simulato.



Una visione più generale

Componenti di un sistema software

Quando parliamo del mondo software, parliamo di componenti. Il nostro scopo è riuscire a gestire questi sistemi:

- Operating system (OS)
    - Proprietario (*Windows*) oppure open source (*Linux*)
- Network management
    - Basato su TCP/IP, LAN oppure altri protocolli di rete industriali
    - Esistono standard di comunicazione ad alto livello, usati per scambiare dati e messaggi tra applicazioni distribuite (*CORBA, J2EE, .NET*)
- DBMS (Database Management System)
    - Relazionali, object-oriented, XML-based o file-based
    - Proprietari, open source o fai da te
- Internet server
    - Rendere le informazioni disponibili su Internet
    - Gestire l'accesso dei client
- PDE (Programming Development Environment)
    - Linguaggio di programmazione
    - Ambiente di modifica e debug
- Modello a componenti
    - Riusare componenti software
- Web services
    - Rendere disponibili su Internet specifici servizi

Central repository

Quando molti utenti condividono un repository dei dati condiviso. Il **data store** è il centro di questa architettura, è implementato tramite un DBMS. Le applicazioni client possono accedere/modificare ed aggiungere/eliminare i dati sul data store.
Diverse possibilità di processing dei dati:
> Thin client: il grosso della computazione è eseguita sul server.
> Balanced intelligence: il client ed il server condividono il carico.
> Fat client: il grosso della computazione è eseguita sul client.

PRO
- Tutti i dati in un singolo posto
- Accesso efficiente per grandi gruppi di utenti a sostanziali quantità di dati
- Divisione dei compiti tra applicazione e repository
- Le applicazioni non hanno bisogno di sapere l'una dell'altra

CONTRO
- Le applicazioni si interfacciano con uno specifico modello di dati
- Cambiare il modello dei dati è difficile e costoso
- Alto overhead di comunicazione
- Forte affidamento sul repository centrale

Client-server (C/S)

Una rete di molti processori e dispositivi. Il **server** offre servizi, come *data storage o accesso a internet*, il **client** usa i servizi offerti, tramite chiamate a procedure remote o object request broker.
I server e i client possono essere eseguiti concorrentemente sulla stessa macchina. I server possono anche agire come client.
Il middleware è un sistema software che abilita la comunicazione tra client e server. L'object request broker (ORB) è un sistema che permette l'accesso client-server usando le tecnologie OO: i servizi disponibili sono visti come oggetti.
L'architettura client-server può essere usata per implementare lo stile central repository.

PRO

- Il sistema è distribuito e modulare
- Gli effetti della rottura di un singolo server sono limitati
- Cambiare la struttura interna di un server è semplice e ha meno effetto sul sistema

CONTRO

- Architettura ed integrazione tecnologica complessa
- Alti costi di sviluppo, testing e manutenzione
- Degradazione delle performance

Inter/intra-net based

Disponibilità più ampia di sistemi software e strumenti. Le applicazioni possono agire come server, accettando connessioni in entrata e provvedendo servizi, e come client, accedendo alla rete richiedendo informazioni.
È basato sullo standard TCP/IP. Può hostare ed accedere servizi web. Per scambiare informazioni usa formati dati basati su caratteri: XML (Extended Markup Language) è lo standard per definire ed interpretare formati dati basati su caratteri.
Una volta che il sistema è connesso ad Internet, la locazione fisica non è più un vincolo.

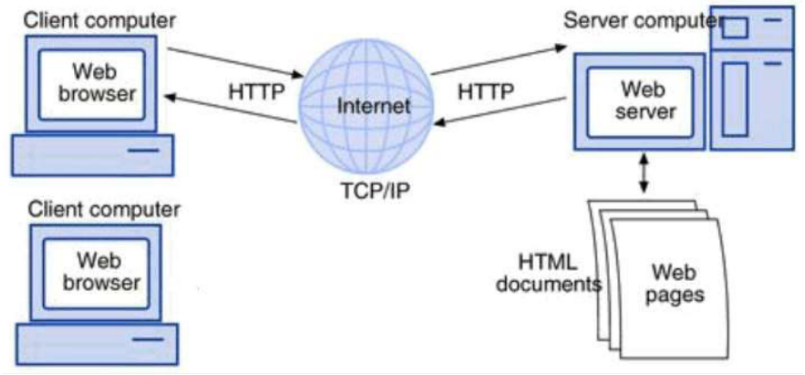
PRO

Gli stessi del client-server, e in più:

- Il sistema è basato su standard aperti, con tecnologie affidabile e molto usate
- Molti software di scambio dei dati ed applicazioni server disponibili liberamente e open source
- Lo scambio di dati è basato sui caratteri, quindi di facile maneggiabilità
- L'architettura può scalare da una rete locale all'intero Internet

CONTRO

- La velocità di scambio dei dati potrebbe essere insufficiente, per applicazioni che scambiano continuamente grosse quantità di dati
- La sicurezza del sistema è un fattore di preoccupazione, sono fondamentali firewall ed altre sicurezze appropriate
- Lo sviluppo di applicazioni non banali è complesso



n-tiers architecture

Una prospettiva diversa per vedere un sistema C/S distribuito. 3-tier architecture:
- Il flusso delle informazioni è lineare
- Ogni livello può essere aggiornato e rimpiazzato immediatamente
- Se il livello intermedio è multi-livello si parla di "n-tier architecture"
I pro e contro sono gli stessi dell'architettura client-server.

Layered architecture

Le architetture a livelli giocano un ruolo essenziale nel mondo della programmazione, poiché hanno il duplice scopo di disaccoppiare logicamente e programmaticamente le complessità sottostanti per consentire l'accesso ad entità sovrastanti.
L'esempio classico, ancora più che *Linux*, è il *TCP/IP*.
Questa architettura è anche chiamata macchina astratta, in quanto è una forma di astrazione, una regola per implementare moduli e servizi.
Le operazioni ed i servizi possono accedere tra di loro solo nello stesso livello oppure in quello adiacente. I livelli interni non sono a conoscenza dei livelli esterni.
Rappresenta un modo alternativo di gestire le architetture viste in precedenza (gestire, non sostituire).
I servizi sono cosa un livello può fare, le interfacce sono i modi di accedere un livello ed i protocolli sono i modi di implementare un livello.
PRO
- I livelli sono indipendenti: è modulare quindi è semplice progettare, testare e ed installare nuovi componenti
- Supporta lo sviluppo incrementale
CONTRO
- I dati passano attraverso ogni livello, quindi c'è overhead di scambio
- A volte è difficile seguire la struttura a livelli esattamente

Livelli di un'architettura a livelli

- Inner layers: effettuano operazioni vicine al set di istruzioni macchina ed al kernel del sistema operativo.
- Intermediate layers: effettuano servizi di utility e funzioni di applicazioni software.
- Outer layers: servizi chiamati direttamente dalle applicazioni esterne.

Architetture di rete a livelli

Un'architettura di rete: definisce un insieme di livelli ed i protocolli usati per la comunicazione tra processi peer. Due maggiori architetture di rete a livelli:

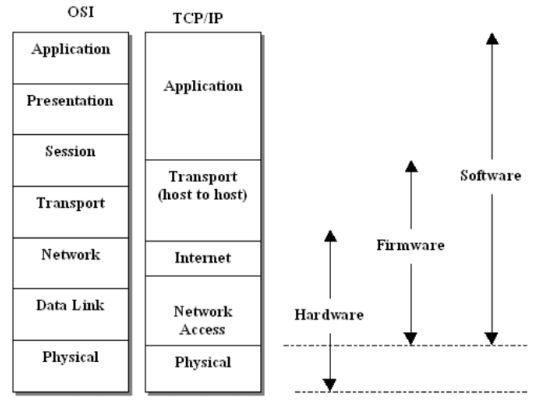
- Open Systems Interconnection (OSI) Reference Model
  - Sviluppato da ISO
  - Utile per discutere la progettazione di reti di computer e la loro costruzione
- TCP/IP (Transmission Control Protocol / Internet Protocol)
  - Usato per consegnare dati su Internet
  - IP: usato in reti in cui ciascun nodo singolo è inaffidabile
  - TCP: controllo end-to-end ed altro, più affidabile

OSI è più astratto, regolare e dettagliato. TCP/IP è più largamente usato poiché alla base di Internet.

Architettura di rete OSI

Layer name	Layer description
Application Layer	Interface to an application program. E.g., a file transfer program.
Presentation Layer	Data formatting and management functions. E.g. text compression, conversion, encryption.
Session Layer	Negotiates the establishment of a session with the destination node.
Transport Layer	Breaks messages into packets, and adds header information to them.
Network Layer	Manages packet routing, adding proper addresses to outgoing packets.
Data Link Layer	Adds header and trailer information. Performs error checking at destination.
Physical Layer	Transmission of electrical, optical or radio signals on the physical medium.

Architettura di rete TCP/IP



Layer name	Layer description
Application Layer	Includes several high-level protocols: -The *File Transfer Protocol* (FTP) provides file transfers between computers -The *TELNET* protocol supports remote login and execution of programs. -The *Simple Mail Transfer Protocol* (SMTP) handles electronic mail. -The *HyperText Transfer Protocol* (HTTP) supports the World Wide Web. -Other protocols support news-groups, spreading the names of new nodes around the Internet, and other functions.
Transport Layer	This layer provides a reliable service (TCP), used by the sender and receiver to exchange control packets that support reliable delivery of data between those two points.
Internet Layer	This layer implements an unreliable, connectionless network delivering IP packets.
Host-to-network Layer	Handles all aspects of connecting the computer to the physical network.

Parallel architecture

Adatta per effettuare computazioni time-critical. Ha molti processori che lavorano allo stesso compito, scambiandosi risultati intermedi.
Ci sono due modi di implementare un'architettura parallela: il primo è avere processori semplici e fortemente connessi, migliaia o milioni di processori contigui connessi assieme. Il secondo è avere come processori dei computer reali, connessi attraverso reti standard o Internet. La computazione viene distribuita su tanti terminali, è necessario software appropriato per consentire ai pc di lavorare in parallelo. Abbiamo due modelli di elaborazione parallela:
- A memoria condivisa: ha i suoi limiti, ad es. nel cloud computing non abbiamo le macchine in loco.
- A memoria privata con comunicazione tra processi (IPC).
La computazione parallela non è sempre più veloce dell'esecuzione classica, dobbiamo capire come gestirla adeguatamente. I problemi principali da affrontare sono:
1. Dove si trova il dato, è il primo collo di bottiglia
2. Lo scheduling, ovvero come vado a prendere i dati
3. La coerenza dei dati, sincronizzazione, complessità dell'algoritmo

Approccio grid computing

Un insieme di progetti che puntano ad effettuare enormi computazioni utilizzando una rete di computer connessi via Internet. Alcuni esempi di applicazioni:
- Drug design
- Nuclear physics computation
- Processing outer space signals
- Humane Genome Project

Architettura di applicazioni

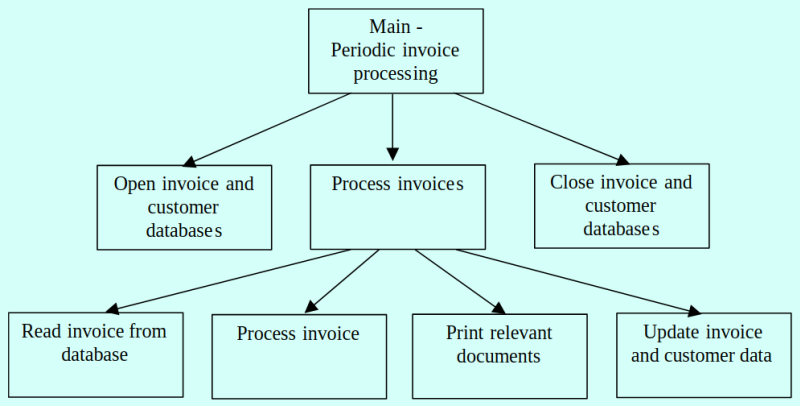
Un modo di strutturare un'applicazione o un sotto-sistema.
Modelli architetturali di applicazioni per gli utenti, ad esempio *il software scritto dagli sviluppatori*.
Modelli di controllo di un'applicazione: in che modo i moduli che la compongono sono controllati per far si che funzionino correttamente.

Centralized control

I moduli sono passivi, cominciano l'esecuzione solo se comandata dal modulo di controllo. Quando l'esecuzione termina il controllo ritorna al modulo di controllo. Due modelli:
- Call and return
- Manager

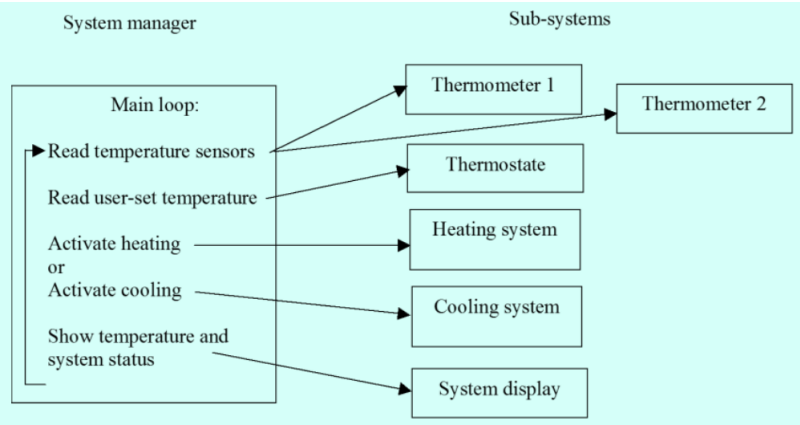
Call and return

Viene lanciato il programma Main. Esso chiama funzioni di livello inferiore, che potrebbero chiamare a loro volta altre funzioni, e il risultato viene restituito al chiamante. L'applicazione termina quando il programma Main fa return.
Una funzione "non-foglia" può essere considerata come il modulo di controllo delle sue sotto-funzioni.
Attenzione ad usare la ricorsione.
Sono possibili chiamate a procedure remote per funzioni su computer di reti diverse.



Manager

Il modulo manager controlla l'avvio, lo stop e la coordinazione degli altri moduli. Fa questo utilizzando un loop di controllo infinito. Periodicamente verifica lo stato dei sensori e dei moduli di input.
I moduli possono essere funzioni, oggetti o sotto-sistemi generici.
È usato nel controllo industriale e nei sistemi real time.



Event-driven control

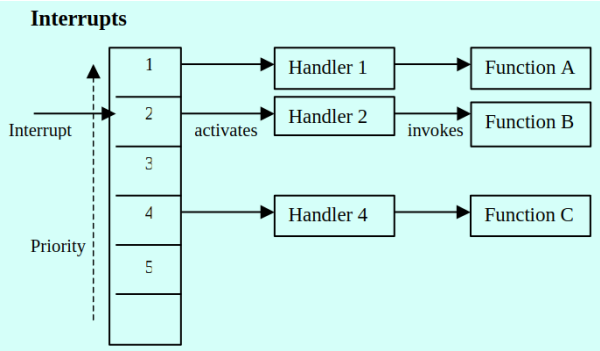
Un registro nel sistema associa i moduli con i possibili eventi. Un evento avviene fuori dal controllo del modulo che lo gestisce. Anche qui due modelli (entrambi necessitano di un gestore di eventi, solitamente il SO):
- Broadcast
- Interrupt-driven

Broadcast

Tutti gli eventi sono comunicati in broadcast ai moduli in grado di gestirli. Azioni appropriate sono poi intraprese dai moduli che hanno verificato la necessità di rispondere.
L'approccio è modulare: aggiungere nuovi componenti ed eventi è abbastanza semplice. Il modello è complesso in termini di coordinazione.
Es. *Gestione delle moderne interfacce grafiche (GUI).*

Interrupt-driven

Usato per sistemi real time con un tempo di risposta critico.
L'interrupt è classificato con una priorità ed associato ad un gestore definito (solitamente una funzione).
Quando viene ricevuto un interrupt di priorità maggiore il sistema blocca il processo corrente, chiama la funzione associata, quando essa termina il controllo viene restituito al processo messo in pausa.
Quando viene ricevuto un interrupt con priorità minore finisce nella lista di attesa.
È difficoltoso da implementare:
- le funzioni potrebbero dover essere stoppate asincronamente
- le funzioni ad alta priorità potrebbero bloccare il sistema



Processi multi-thread

Più processi eseguono in parallelo tramite un hardware multiprocessore oppure sequenzialmente gestiti con priorità. Un processo può essere avviato chiamando una funzione come thread indipendente oppure mandando un messaggio ad un oggetto come thread indipendente.
Il multi-threading non è una forma di controllo, può essere controllato in modo centralizzato o event-driven.
Un semaforo è un singolo punto di controllo che gestisce l'accesso ad una risorsa che può essere acceduta da un solo processo alla volta.

Decomposizione modulare

Modello data flow

Anche chiamato modello pipes and filters, le informazioni fluiscono tramite i pipes e le trasformazioni tramite i filters.

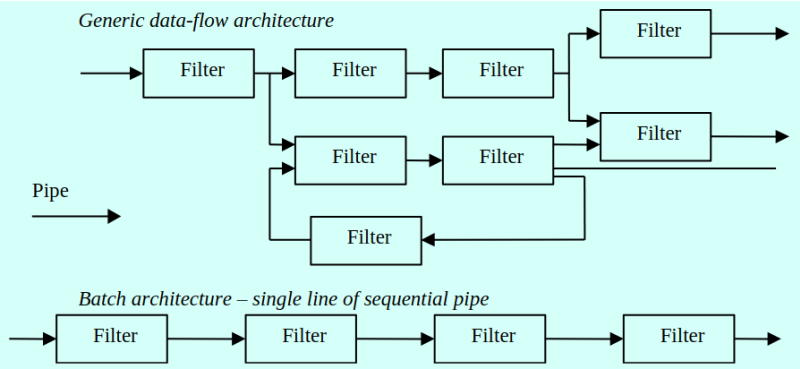
`Input --> trasformazione --> output`

PRO

- I filters sono indipendenti e possono essere riusati
- È intuitivo
- Un filter può iniziare l'output mentre l'input sta venendo processato
- La trasformazione dei dati può essere fatta in sequenza o in parallelo
- Aggiungere e rimuovere filter è relativamente semplice

CONTRO

- Non tutte le applicazioni son facilmente modellabili
- La gestione degli errori è un problema
- Il formato dati di scambio incompatibile compromette la modularità
- Le architetture possono essere molto complesse



Modello object oriented

La decomposizione del sistema è centrata sui dati. Gli oggetti sono l'unità di decomposizione: sono formati da dati, operazioni (metodi) ed interfacce. Comunicano tra di loro scambiando messaggi.
L'incapsulamento permette di nascondere i dettagli dell'implementazione.
L'ereditarietà consente di creare nuove classi a partire da altre esistenti, specificando solo le differenze.

PRO

- È intuitivo
- Stesso modello OO per tutte le fasi dello sviluppo
- Fase di specifica: gli oggetti sono più stabili delle funzioni
- Gli oggetti sono modulari ed indipendenti
- Lo scambio di messaggia avviene in sequenza o in parallelo
- Si possono riusare gli oggetti
- I cambiamenti all'implementazione interna di un oggetto hanno un impatto minimale sul resto del sistema
- Testare il sistema è semplice

CONTRO
- Lo scambio di messaggi porta un alto overhead
- Lo sviluppo OO è difficoltoso per programmatori di linguaggi procedurali
- Può portare a "ravioli code": analogo allo "spaghetti code" della programmazione procedurale, ovvero incasinato.

Effettuare la progettazione

Quando effettuiamo la progettazione dobbiamo considerare:

- Astrazione
    - Considerare di un entità solo gli aspetti rilevanti al problema, ignorando o nascondendo gli altri.
    - I sistemi e i moduli possono essere visti come vari livelli di astrazione.
- Scalabilità
    - Il sistema può essere visto in diverse scale.
    - Dividere un sistema in sotto-sistemi, poi in sotto-sotto-sistemi e così via.
- Astrazione procedurale
    - Sistema decomposto in termini di funzioni.
    - Ogni funzione è caratterizzata dalla sua signature.
    - Usare le funzioni senza sapere l'implementazione interna.
- Astrazione dei dati
    - ADT (Abstract Data Type): tipi di dato definiti in termini delle operazioni (metodi) che possono essere svolte su di essi.
    - Solo le signature delle operazioni sono specificate.
    - Usato dalle classi in linguaggi di programmazione OO.

Decomposizione modulare

Sistema decomposto in sotto-sistemi e successivamente in moduli -> principio dividi e conquista.
Astrazione: per usare un modulo serve soltanto l'interfaccia.
Bottom-up design: un insieme di moduli riutilizzabili per costruire un sistema più complesso.
Top-down design: decomporre il sistema fino ad arrivare all'ultimo livello in cui i moduli possono essere implementati direttamente senza ulteriore decomposizione.

Considerare alternative

I progettisti devono considerare molte alternative: scegliere e/o mischiare per trovare la soluzione migliore.
I criteri utilizzati per scrutinare le alternative:
- Aderenza ai requisiti del cliente
- Semplicità e qualità
- Costi e risorse

Raffinamento iterativo, applicato a gerarchie di classi e metodi: iterativamente aggiungere sempre più classi specializzate e successivamente rifinire le operazioni in istruzioni più dettagliate.

Strategy pattern in Java

Codice di riferimento

```
testScope
  | t |
  t := 42.
  self testBlock: [Transcript show: printString]

testBlock: aBlock
  | t |
  t := nil.
  aBlock value
```

Questo blocco di codice SmallTalk va ben oltre i semplici puntatori a funzione, ed è lo scopo finale dello strategy pattern. Calcolare `testScope` restituisce 42 ... come mai?

Classi in scope interni

In Java e C++ è possibile mettere una definizione di classe all'interno della definizione di un'altra classe.
In C++ questo è legato solo alla visibilità e ai nomi.
In Java la questione è più profonda, possiamo distinguere tra:
- **nested class**, che sono inserite dentro altre classi al solo scopo di packaging/naming.
- **inner class**, i cui oggetti dipendono dall'esistenza dell'oggetto della classe che li annida.
- **local class**, che sono dichiarate localmente all'interno di un blocco di codice, piuttosto che come un membro della classe.

Nested classes

È possibile mettere una definizione di classe all'interno della definizione di un'altra classe. Questa è chiamata una **nested class**.
Le nested class permettono di nascondere l'esistenza di una classe al mondo esterno. Inoltre, consentono di raggruppare classi che sono legate logicamente e di controllare la visibilità di una dall'interno di un'altra.
Il nome di una nested class è locale alla classe che la contiene. La nested class è nello scope della classe che la contiene. Le dichiarazioni in una nested class possono usare solo membri statici dalla classe contenitore.

Inner classes

Java porta questo approccio più lontano, definendo le cosiddette **inner class**.
In Java, semplicemente dichiarare una classe dentro il corpo di un'altra, senza ulteriori qualificazioni, risulta in una inner class. Per dichiarare una nested class bisogna aggiungere la keyword `static` a fronte della dichiarazione.

Esempio di nested e inner class in Java

```
public class X {
  int instanceVar; // Belongs to an instance of class X
  static int staticVar; // Belongs to class X

  private class Inner {
    int y;
    Inner() { y = instanceVar + staticVar; }
  }

  private static class Nested {
    int x;
    Nested() {
      x = staticVar;
      // x += instanceVar; Error!
    }
  }
}
```

Esempio di nested class in C++

```
class X {
  public:
  int instanceVar; // Belongs to an instance of class X
  static int staticVar; // Belongs to class X

  class Nested {
    int x;
    Nested() : x(staticVar) {
      // instanceVar++; Error!
    }
  };
};
```

Commenti sulle inner class

Ciascuna istanza ha un'istanza che la racchiude, e può utilizzare i suoi membri.
Le inner class non possono avere membri static.
Le inner class non possono avere lo stesso nome della classe che le racchiude.
Se una inner class è una local class (vedi dopo), ha accesso ai membri della classe che la racchiude, alle variabili locali final e ai parametri.
Una inner class può anche essere una classe anonima (vedi dopo), ovvero una classe senza nome definita all'interno di un'espressione. Sono simili alle local class ma possono avere solo una istanza.

Local classes

Sia Java che C++ permettono la definizione di classi all'interno del corpo di una funzione, queste sono dette **local class**.
Le local class sono molto utili quando abbiamo bisogno di codice e strutture dati per effettuare task specifici all'interno di una funzione, ma non vogliamo renderli disponibili al mondo esterno.
Perciò le local class intendono forzare un maggiore incapsulamento e information hiding a livello di scope di funzione. Il nome di una local class è locale allo scope che la racchiude. La local class è nello scope che la racchiude ed ha lo stesso accesso ai nomi all'esterno della funzione di tale funzione che la contiene.
In Java le local class sono anche inner class, e non semplicemente nested class.

Interfacce

L'interfaccia è un modo per definire tipi tramite il nome dei metodi supportati.
La dichiarazione di un'interfaccia introduce un nuovo tipo riferimento, i cui membri sono classi, interfacce, costanti e metodi. Questo tipo non ha variabili d'istanza e tipicamente dichiara uno o più metodi astratti. Classi non correlate possono implementare l'interfaccia fornendo implementazioni per i suoi metodi astratti.
Le interfacce non possono essere istanziate, possono solo essere implementate da classi o estese da altre interfacce.
Le interfacce contengono solo costanti, signature dei metodi, metodi default, metodi statici e/o tipi annidati. Tutto ciò che compare in un'interfaccia è public.
I programmi possono usare le interfacce per eliminare la necessità per classi correlate di condividere una superclasse astratta comune e per aggiungere metodi ad Object.

Dichiarazione e implementazione

Nella sua forma più comune, un'interfaccia è un gruppo di metodi correlati senza corpo. Le interfacce hanno le stesse specifiche di accesso delle classi. Le interfacce supportano l'ereditarietà singola e multipla, visto che l'assenza dei dati d'istanza elimina l'ambiguità, di ad esempio di C++.

```
public interface Bicycle {
  void changeCadence(int newValue);
  void changeGear(int newValue);
  void speedUp(int increment);
  void applyBrakes(int decrement);
}
```

Le interfacce possono avere metodi default, ovvero un metodo che è definito a meno che non venga definito nella classe che implementa l'interfaccia. L'utilizzo dei metodi default è sconsigliato.
I metodi default sono definiti con `default`, i metodi statici con `static`.
Tutti i metodi astratti, default e statici sono implicitamente public.
Un'interfaccia può contenere dichiarazioni di costanti. Tutte le costanti sono implicitamente public, static e final.

```
public interface GroupedInterface extends Interface1, Interface2,
Interface3 {
  // constant declarations
  // base of natural logarithms
  double E = 2.718282;
  // method signatures
  void doSomething (int i, double x);
  int doSomethingElse(String s);
}

class BianchiBicycle implements Bicycle {
  int cadence = 0, speed = 0, gear = 1;
  // The compiler will now require that methods
  // changeCadence, changeGear, speedUp, and
  // applyBrakes all be implemented. The compilation will
  // fail if those methods are missing from this class.
  void changeCadence(int newValue) {
    cadence = newValue;
  }
  //...
  void printStates() {
    System.out.println("cadence:" + cadence + " speed:" + speed + "
gear:" + gear);
  }
}
```

Metodi nelle interfacce

I metodi default e quelli astratti nelle interfacce sono ereditati come i metodi d'istanza. Comunque, quando il supertipo di una classe o interfaccia fornisce metodi default multipli con la stessa signature, il compilatore Java segue delle regole di ereditarietà per risolvere il conflitto di nomi. Queste regole sono guidate per seguire due principi:
1) I metodi d'istanza hanno la priorità sui metodi default dell'interfaccia.
2) I metodi di cui è già stato fatto override da altri candidati sono ignorati. Questa circostanza può capitare quando i supertipi condividono un antenato comune.

1. I metodi d'istanza hanno la priorità sui metodi default dell'interfaccia.

```
public class Horse {
  public String identifyMyself() { return "I am a horse.";}
}
public interface Flyer {
  default public String identifyMyself() {
    return "I am able to fly."; }
}
public interface Mythical {
  default public String identifyMyself() {
    return "I am a mythical creature."; }
}

public class Pegasus extends Horse implements Flyer, Mythical {
  public static void main(String... args) {
    Pegasus myApp = new Pegasus();
    System.out.println(myApp.identifyMyself());
  }
}
```

Il metodo Pegasus.identifyMyself() stampa "I am a horse".

2. I metodi di cui è già stato fatto override da altri candidati sono ignorati.

```
public interface Animal {
  default public String identifyMyself() {
    return "I am an animal."; }
}
public interface EggLayer extends Animal {
  default public String identifyMyself() {
    return "I am able to lay eggs."; }
}
public interface FireBreather extends Animal { }

public class Dragon implements EggLayer, FireBreather {
  public static void main (String... args) {
    Dragon myApp = new Dragon();
    System.out.println(myApp.identifyMyself());
  }
}
```

Il metodo Dragon.identifyMyself() stampa "I am able to lay eggs.", perché è quello più specifico.

Se due o più metodi default definiti indipendentemente confliggono, oppure un metodo default e un metodo astratto, il compilatore Java produce un errore. Per risolverlo bisogna esplicitamente fare override del supertipo dei metodi.
Se abbiamo due metodi allo stesso livello, abbiamo un errore perché il compilatore non può decidere da solo.
Consideriamo l'esempio sulle macchine computer-controlled che ora possono volare. Abbiamo due interfacce (OperateCar e FlyCar) che forniscono un'implementazione default per lo stesso metodo.

```
public interface OperateCar {
  // ...
  default public int startEngine(EncryptedKey key) {
    // Implementation
  }
}
public interface FlyCar {
  // ...
  default public int startEngine(EncryptedKey key) {
    // Implementation
  }
}
```

Una classe che implementa sia OperateCar che FlyCar deve fare override del metodo startEngine(). Potremmo invocare una qualunque delle implementazioni default usando la keyword `super`.

```
public class FlyingCar implements OperateCar, FlyCar {
  // ...
  public int startEngine(EncryptedKey key) {
    OperateCar.super.startEngine(key);
    FlyCar.super.startEngine(key);
  }
}
```

Specifichiamo il nome davanti per disambiguare, perché con le interfacce c'è l'ereditarietà multipla. Il nome che precede super (in questo esempio, OperateCar o FlyCar) deve riferirsi ad una super-interfaccia diretta che definisce o eredita il metodo invocato.
Questa forma di invocazione dei metodi non è ristretta alla differenziazione tra interfacce multiple implementate che contengono metodi default con la stessa signature. Si può usare la keyword `super` per invocare un metodo default sia nelle classi che nelle interfacce.

I metodi d'istanza ereditati da una classe possono fare override dei metodi astratti di un'interfaccia. Consideriamo le seguenti interfacce e classi:

```
public interface Mammal {
  String identifyMyself();
}
public class Horse {
  public String identifyMyself() {
    return "I am a horse.";
  }
}
public class Mustang extends Horse implements Mammal {
  public static void main(String... args) {
    Mustang myApp = new Mustang();
    System.out.println(myApp.identifyMyself());
  }
}
```

Il metodo Mustang.identifyMyself() stampa "I am a horse". La classe Mustang eredita il metodo identifyMyself() dalla classe Horse, la quale fa override del metodo astratto con lo stesso nome nell'interfaccia Mammal.

Riassunto

In basso a destra "nasconde" e non "fa override" perché definendo static il binding viene fatto a tempo di compilazione.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Metodi static e default

I metodi statici sono per comportamenti dell'interfaccia generali e non permettono l'accesso a dati non-statici nè overriding.
I metodi default possono subire override e sono un meccanismo potente per consentire l'evoluzione delle interfacce, supportando le implementazioni non obbligatorie di tutti i metodi dentro alle classi.

Esempio da Oracle

Supponiamo di avere:

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
}
```

Successivamente la vogliamo evolvere in:

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
  boolean didItWork(int i, double x, String s);
}
```

Il codice delle vecchie implementazioni ora si romperebbe, dato che non ha il metodo didItWork.
Ma con il default tutto è sistemato:

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
  default boolean didItWork(int i, double x, String s) {
    // Method body
  }
}
```

Evolviamo l'interfaccia con un metodo default in modo che le classi sprovviste di tale metodo continuino a funzionare. È quindi un fattore di retrocompatibilità.

Classi anonime

Le classi anonime permettono di scrivere codice più conciso, dichiarando ed istanziando una classe nello stesso momento. Sono come le local class tranne per il fatto che non hanno un nome.
Il pattern di utilizzo più comune è quando abbiamo bisogno di usare una local class solo una volta. La classe è definita dentro un'altra espressione.

Listeners

Un event listener è usato per processare eventi. Per esempio, componenti grafici come un JButton o un JTextField sono noti come sorgenti di eventi. Questo significa che possono generare eventi, quando un utente clicca sul JButton o scrive nel JTextField. Il compito dell'event listener è di catturare questi eventi e farci qualcosa.

Gli event listener sono in realtà diversi tipi di interfacce. Ce ne sono di molti tipi: ActionListener, ContainerListener, TextListener, WindowListener per citarne alcuni.
Ciascuna interfaccia definisce uno o più metodi che devono essere implementati da una classe per processare l'evento.

C'è una buona flessibilità con gli event listener in quanto più di un componente grafico può essere associato con lo stesso event listener. Questo vuol dire che se abbiamo un insieme di componenti simili che fanno praticamente la stessa cosa, i loro eventi possono essere gestiti da un singolo event listener.
Ad esempio, un JButton necessita un oggetto di una classe che implementi un ActionListener per processare il bottone. Se ci sono numerosi bottoni che svolgono compiti simili quando cliccati, possono essere assegnati allo stesso ActionListener.

Quando vogliamo "ascoltare" un click sul bottone dobbiamo avere una classe che implementi l'interfaccia ActionListener, una semplice interfaccia con un solo metodo.

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

Per implementare l'interfaccia ActionListener la classe deve avere il metodo actionPerformed(). Questo metodo è chiamato quando avviene un evento di click sul bottone.
L'oggetto ActionEvent chiamato 'e' contiene informazioni sull'evento. Per esempio, possiamo capire quale bottone è stato cliccato chiamando il metodo e.getActionCommand().

SimpleCalc

Vogliamo costruire una semplice calcolatrice (classe SimpleCalc).
Per prima cosa, dobbiamo creare la GUI, contenente i nostri bottoni ed un campo di testo per fornire informazioni all'utente.

```
public SimpleCalc() {
  guiFrame=new JFrame();
  numberCalc=new JTextField();
  buttonPanel = new JPanel();
  //Make a Grid that has three rows and four columns
  buttonPanel.setLayout(new GridLayout(4,3));
  guiFrame.add(buttonPanel,BorderLayout.CENTER);
  for (int i=1;i<10;i++) { //Add the number buttons
    addButton(buttonPanel, String.valueOf(i));
  }
  JButton addButton = new JButton("+"); //Add the operation buttons
  addButton.setActionCommand("+");
  buttonPanel.add(addButton);
  buttonPanel.add(subButton);
  buttonPanel.add(equalsButton);
  guiFrame.setVisible(true);
}
private void addButton(Container parent, String name) {
  JButton but = new JButton(name);
  but.setActionCommand(name);
  parent.add(but);
}
```

Con SimpleCalc implementiamo l'interfaccia ActionListener in tre diversi modi per mostrare le opzioni a disposizione per ascoltare gli eventi:
- dentro alla classe che contiene
- come una inner class
- come una inner class anonima

Implementazione dentro alla classe che contiene

Questo approccio può essere una comoda scorciatoia se abbiamo numerosi componenti grafici che agiscono esattamente nella stessa maniera quando cliccati.
In questo caso i bottoni numerati da 1 a 9 agiranno allo stesso modo: quando uno di essi viene cliccato, il numero che rappresenta viene piazzato nel JTextField.
Implementiamo l'interfaccia dentro SimpleCalc, significando che la classe SimpleCalc dovrà implementare il metodo actionPerformed().

```
public class SimpleCalc implements ActionListener{
// All the buttons are doing the same thing.
// It's easier to make the class implementing the
// ActionListener controlling the clicks from one place
@Override
public void actionPerformed(ActionEvent event) {
  //get the Action Command text from the button
  String action = event.getActionCommand();
  //set the text using the Action Command text
  numberCalc.setText(action);
  // Do the job
}
```

Perché ciò funzioni, abbiamo bisogno di aggiungere l'ActionListener ad ogni bottone. Dato che i bottoni numerati sono creati usando addButton(), dobbiamo aggiungergli una linea extra.

```
private void addButton(Container parent, String name) {
  JButton button = new JButton(name);
  button.setActionCommand(name);
  button.addActionListener(this);
  parent.add(button);
}
```

addActionListener prende come parametro la classe che implementa ActionListener con il this.

Implementazione con una inner class

Potremmo implementare ActionListener in una classe separata, ma tale classe deve gestire solamente l'evento di un click sul bottone. Perciò potremmo usare una classe dentro SimpleCalc.
Questo ha inoltre il vantaggio di consentire alla inner class l'accesso ai componenti definiti dentro la classe esterna.
Quando i bottoni di operazione aritmetica (+ e -) sono cliccati, le operazioni da eseguire sono molto simili. La sola differenza è se i numeri saranno sommati o sottratti. Questo rende ideale, per entrambi i bottoni, utilizzare una inner class che implementa ActionListener.

```
public class SimpleCalc{
  private class OperatorAction implements ActionListener {
    public void actionPerformed(ActionEvent event) { }
  }
}
```

OperatorAction è l'inner class ed il suo compito principale è impostare il currentCalc e calcOperation di SimpleCalc. currentCalc mantiene il numero nel JTextField e calcOperation è l'intero rappresentante se è una somma (1) o una sottrazione (2). Questo viene poi usato dal bottone '='.

```
private class OperatorAction implements ActionListener{
  private int operator;
  public OperatorAction(int operator) {
    this.operator = operator;
  }
  public void actionPerformed(ActionEvent event) {
    currentCalc = Integer.parseInt(numberCalc.getText());
    calcOperation = operator;
  }
}
JButton additionButton = new JButton("+");
additionButton.setActionCommand("+");
OperatorAction additionAction = new OperatorAction(1);
additionButton.addActionListener(additionAction);

JButton subtractionButton = new JButton("-");
subtractionButton.setActionCommand("-");
OperatorAction subtractionAction = new OperatorAction(2);
subtractionButton.addActionListener(subtractionAction);
```

Implementazione con una classe anonima

Spesso, ogni bottone causa un'azione diversa. È sempre possibile avere una classe associata a ciascun comportamento, è sufficiente che ogni classe implementi l'interfaccia ActionListener.
Tale approccio causa una proliferazione di classi "standard", rendendo il codice illeggibile.
Possiamo quindi usare le classi anonime, facendo in modo che il codice diventi più semplice ed il comportamento associato ad ogni bottone sia piazzato dove il bottone viene istanziato.

Un click sul bottone dell'uguale causa l'esecuzione dell'operazione aritmetica e la successiva visualizzazione del risultato nel JTextField. È il candidato perfetto per una classe (inner) anonima.

```
JButton equalsButton = new JButton("=");
equalsButton.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) { ... }
});
```

Invece di scrivere una classe separata che implementasse l'interfaccia ActionListener, abbiamo definito una classe anonima proprio lì.
Percepiamo la nuova classe anonima come un pezzo di codice creato al volo per l'interfaccia ActionListener che implementa il metodo actionPerformed.
Abbiamo l'impressione che un pezzo di codice possa essere il parametro della chiamata del metodo addActionListener. Non è così, è interamente una classe che viene creata, ed un oggetto di essa viene istanziato e passato come parametro. Praticamente abbiamo istanziato una classe anonima che implementa l'interfaccia.

```
equalsButton.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    if (!numberCalc.getText().isEmpty()) {
      int number = Integer.parseInt(numberCalc.getText());
      if (calcOperation == 1) {
        ... // perform addition
      } else if (calcOperation == 2) {
        ... // perform subtraction
      }
    }
  }
});
```