PROGETTAZIONE DI SISTEMI ORIENTATI AGLI OGGETTI (OOD)

Michele Marchesi

michele@diee.unica.it

Dipartimento di Ingegneria Elettrica ed Elettronica

Università di Cagliari

SOMMARIO

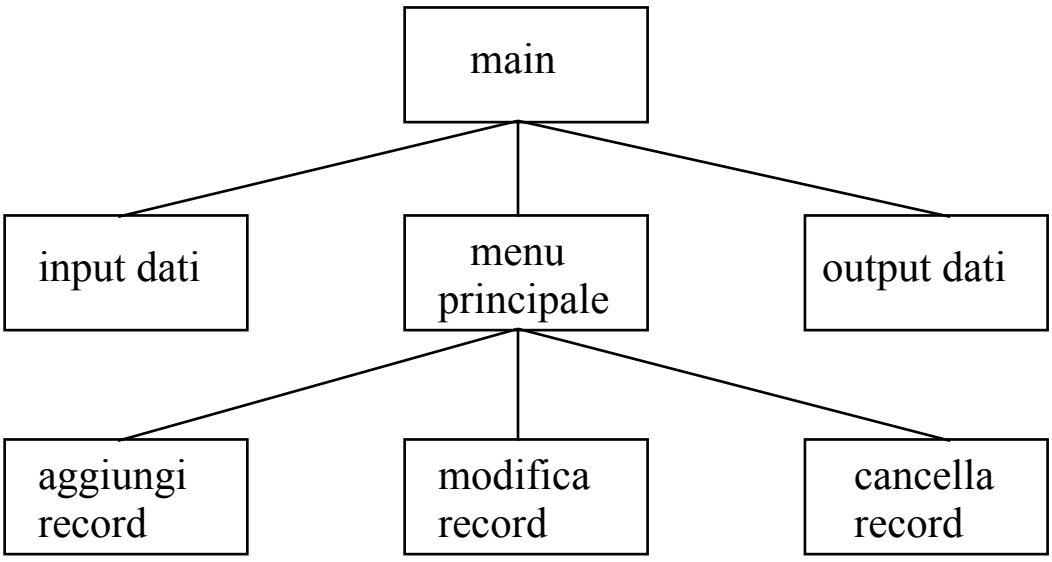
- Che cos'è veramente l'approccio a oggetti
- Analisi e progettazione ad oggetti (OOA, OOD)
- I principi su cui si basa l'OOD
- Cenni di UML
- Le attività fondamentali dell'OOD
- Metodo CRC
- Le classi: individuazione, responsabilità e validazione
- Il modello dinamico
- Le relazioni tra classi
- Principi di buon OOD

Che cos'è la POO?

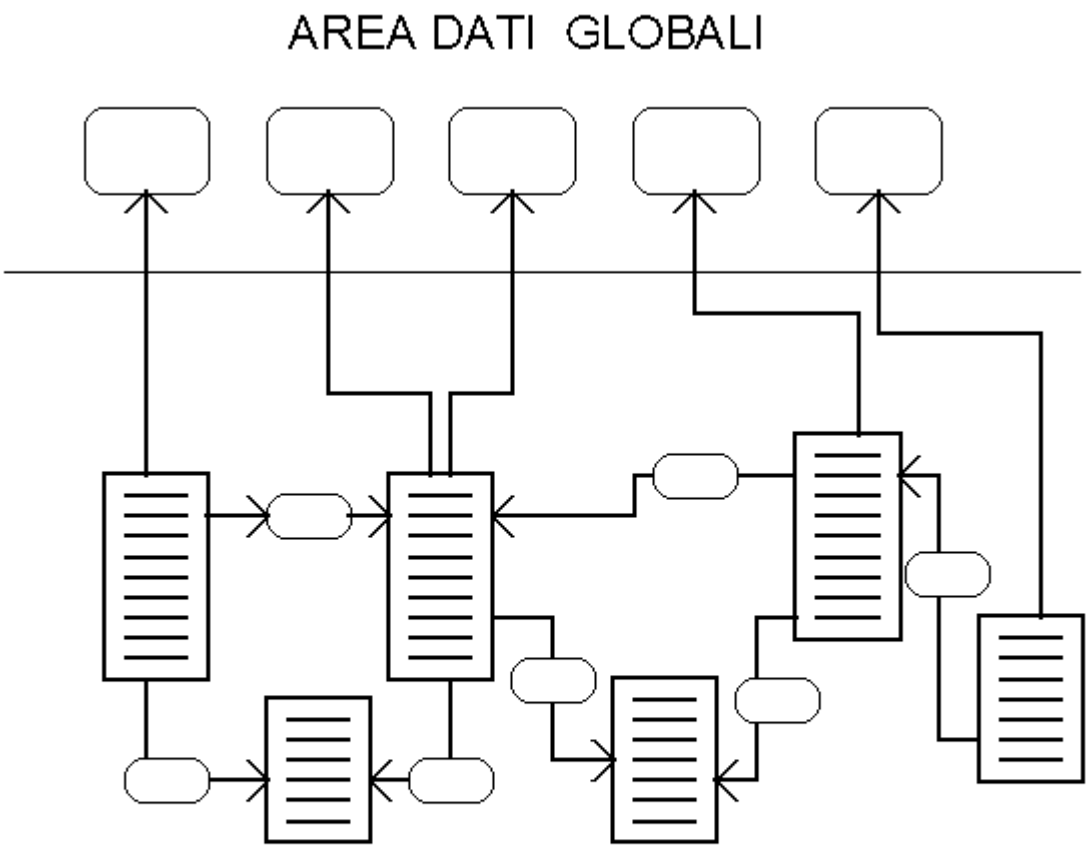
- Oggi, la maggior parte dello sviluppo ex-novo avviene con linguaggi ad oggetti (OOPL): Java, C++, C#, Python, Ruby, Objective-C, ...
- Quindi, tutti proclamano di “programmare ad oggetti”
- Ma è proprio cosi?
- Qual è il vero significato di POO?
  - Per molti, è l'uso di “oggetti” come widget grafici, porte di accesso a DBMS, nel proprio programma
  - Per altri, è la possibilità di definire nuovi tipi di dati (dati + operazioni), e non solo di record
  - Per altri ancora, è programmare definendo classi, in ambienti che spesso richiedono di definire una nuova classe anche per fare le cose più semplici
- In effetti, la POO è un po' tutto questo, ma va molto oltre

Programmazione procedurale

- Un programma procedurale (ad es., in C) è composto da dati e procedure (o funzioni)
- Le procedure si chiamano a vicenda, e operano sui dati
- I dati sono primitivi, o vettori, o record.
- Sono allocati staticamente, o allocati e deallocati dinamicamente
- Le procedure si scambiano i dati (parametri e valori di ritorno), oppure operano su dati globalmente accessibili



Programmazione procedurale



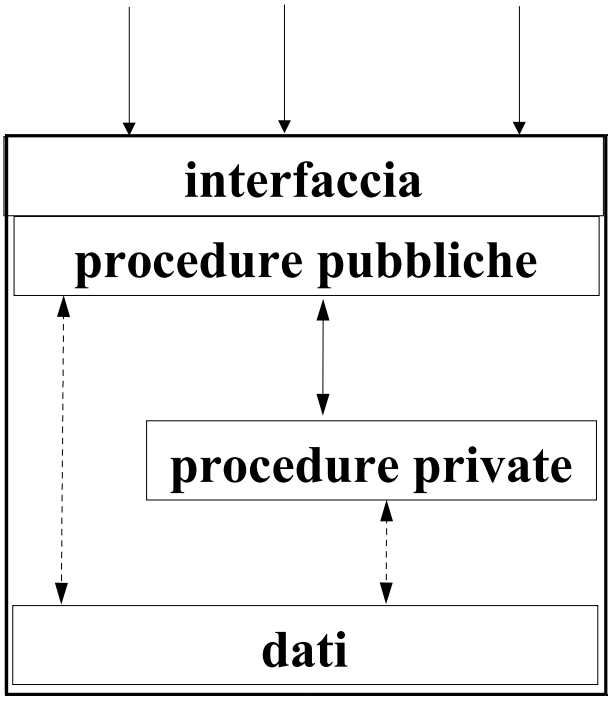
FUNZIONI CHE SI SCAMBIANO DATI ED
ACCEDONO ALL'AREA DATI GLOBALI

I problemi della programmazione procedurale

- Per grandi sistemi, la complessità dell'approccio è molto elevata
- I dati comuni creano forti accoppiamenti tra le funzioni
- Modifiche ai dati possono comportare la modifica a cascata di moltissime procedure, spesso non definibili a priori
- E' difficile scomporre grandi sistemi in sottosistemi (moduli) trattabili separatamente

Programmazione Modulare (PM)

- La soluzione dei linguaggi Ada e Modula-2:
  - Capacità di definire **nuovi tipi di dati**, con le procedure che operano su di essi (**moduli**)
  - Un modulo è accessibile tramite le sue procedure pubbliche (**interfaccia** del modulo)
  - I dati e le altre procedure del modulo sono **privati**, e quindi non visibili all'esterno
- I nuovi tipi di dati sono usati in modo procedurale



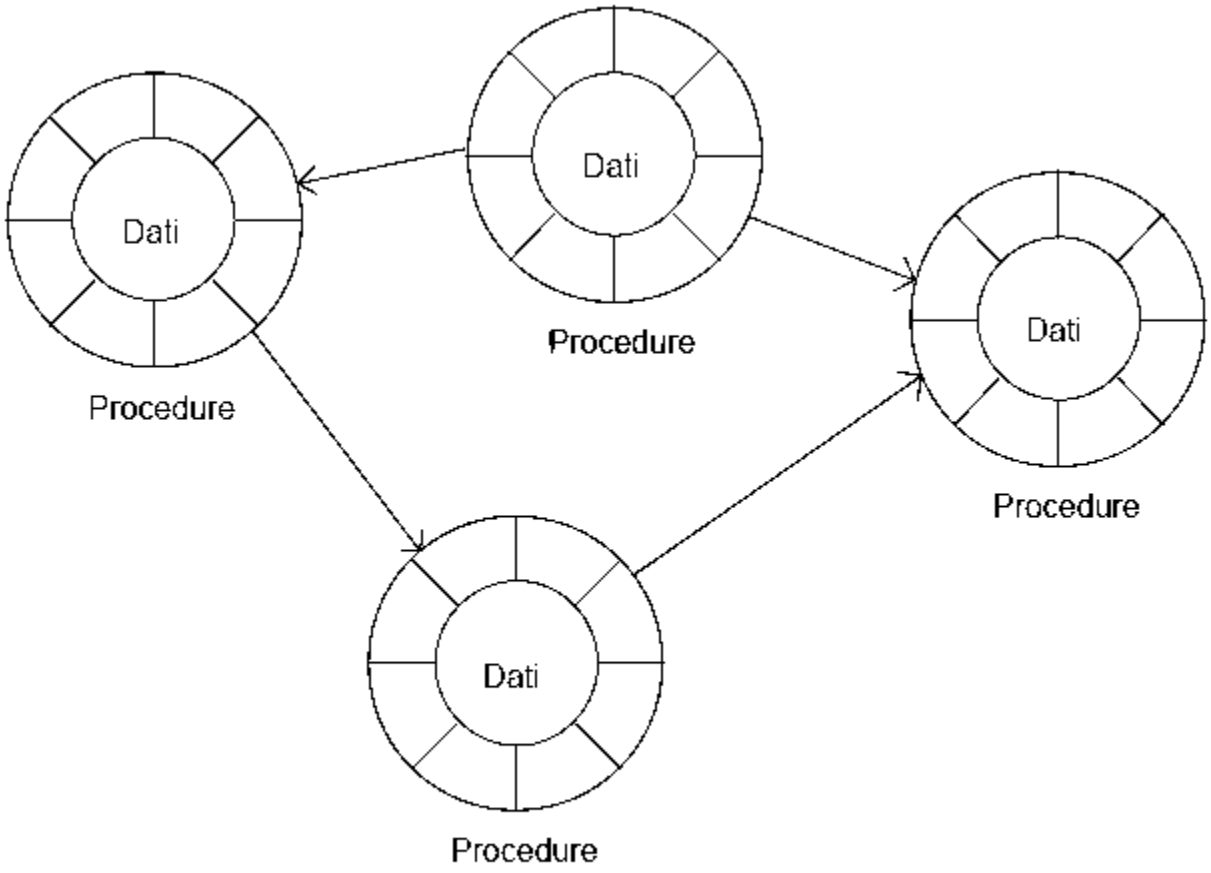
un modulo software

Un programma modulare è costituito da dati definiti dall'utente, ed usati in modo procedurale.
L'accesso ai dati avviene attraverso le interfacce

Programmazione a oggetti (POO)

- Si definiscono **oggetti**, che sono poi i moduli della PM.
- La definizione di un oggetto si chiama **classe**
- Gli oggetti modellano entro il calcolatore in "mondo esterno"
- I sistemi sono più comprensibili e più naturalmente progettabili e modificabili
- Ogni oggetto incapsula i propri dati, e quindi facilita le modifiche al programma
- Un programma ad oggetti è composto da molti oggetti che si scambiamo **messaggi**
- Non c'è uno strato procedurale al di sopra, ma le uniche procedure sono i **metodi** degli oggetti
- I linguaggi ad oggetti: Java, C++, C#, Smalltalk

Programmazione a oggetti (POO)



OGGETTI CHE INCAPSULANO I PROPRI DATI

Programmazione visuale per componenti

- Larga parte dei sistemi di nuovo sviluppo comportano:
  - definizione di un DB ed accesso allo stesso
  - definizione di varie applicazioni basate su finestre
  - definizione di accesso tramite browser HTML
  - capacità di gestire vincoli e reporting
- Per fare tutto ciò, esistono potenti ambienti visuali di sviluppo (ad es. Visual Studio, Eclipse, ...)
- Il programmatore costruisce le applicazioni scegliendo i **componenti base** in modo grafico, e li compone legandoli tra loro ed associando codice agli **eventi** relativi
- Spesso, elaborazioni più complesse sono effettuate da **stored procedure** direttamente entro il DBMS
- Questa è **programmazione per componenti** (**PpC**)
- I componenti sono in realtà pienamente oggetti, sviluppati con POO, ma ciò riguarda chi li scrive, non chi li usa!

PpC vs. POO

- La “vera” POO riguarda piuttosto programmi di elevata complessità e contenuto elaborativo
- Ad esempio:
  - simulatori
  - sistemi di supporto alle decisioni
  - sistemi di rostering, scheduling, ottimizzazione
  - . . .
- In tali programmi, il focus è sul modello dei dati e sulle elaborazioni, non sul DB o sulla GUI (che vanno tenuti il + possibile separati)
- I moderni computer, permettono modelli e simulazioni su grande scala
- La POO è insostituibile in ciò
- Tuttavia, molti dei principi e linee guida dell'OOD/POO sono applicabili anche alla PpC
- Inoltre, spesso si usa PpC quando si potrebbe fare meglio con la POO

Motivazioni e benefici dell'approccio OO

1. Permette di affrontare domini di problemi più difficili, poiché gestisce meglio la complessità.
2. Migliora l'interazione tra l'analista e l'esperto del problema, perché si basa sulla visione umana del mondo.
3. Migliora la coerenza interna dei risultati dell'analisi, mettendo insieme dati e operazioni su di essi.
4. Rappresenta esplicitamente le caratteristiche comuni utilizzando l'ereditarietà.
5. Produce una specifica resistente ai cambiamenti perché si basa sugli oggetti, che sono molto più stabili delle funzioni.
6. Riutilizza i risultati dell'analisi, fornendo una base efficace per l'analisi del dominio e per il riuso.
7. Fornisce una rappresentazione interna coerente, potendo passare dall'analisi alla progettazione alla programmazione utilizzando gli stessi principi, la stessa notazione e gli stessi oggetti.

Analisi e progettazione OO

Analisi

L'analisi è la prima attività svolta nella produzione dei sistemi software, dopo la raccolta dei requisiti.

L'***analisi del dominio*** sviluppa un modello che descrive il *dominio del problema* da affrontare.

> ***Dominio del problema***: la porzione del mondo rilevante per il sistema, sui cui questo deve mantenere informazioni e con cui deve interagire.

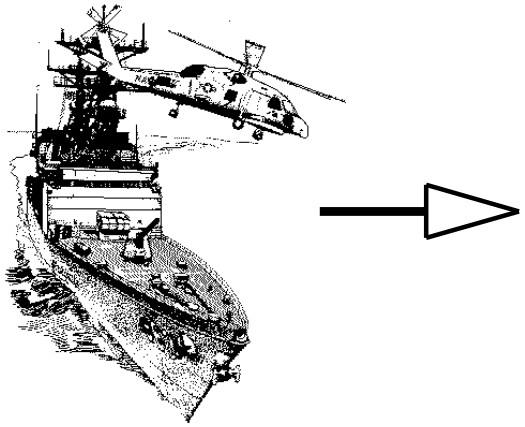
L'***analisi dei requisiti*** si concentra sulla descrizione delle *responsabilità* del sistema in esame.

> ***Responsabilità del sistema*****:** ciò che deve fare il sistema per soddisfare ai suoi requisiti.

L’analisi è il processo di estrarre i "bisogni" di un sistema -- *che cosa* deve fare il sistema per soddisfare il cliente, non *come* il sistema va realizzato

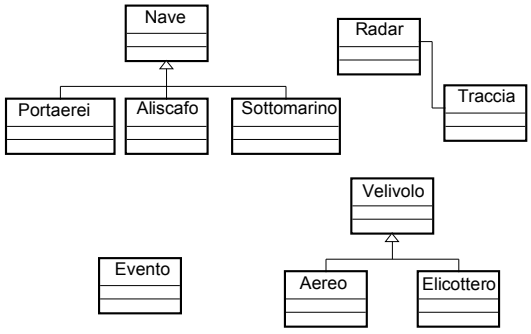
L’analisi si ferma quindi ad un livello di dettaglio che permette di esprimere i concetti fondamentali del sistema, ma non la sua implementazione

L’OOA produce un primo *modello OO del sistema*, che è la base di partenza del successivo processo di sviluppo del software



Dominio del problema

(comando e controllo aeronavale)



Modello OOA del sistema da sviluppare

Progettazione (design)

Determina **l'architettura** del sistema, a vari livelli:

- configurazione hardware e pacchetti software (sistemi operativi, linguaggi, database, ambienti di interfaccia, ecc.) da utilizzare
- *modello logic*o del sistema (schema del database, prototipi delle funzioni, ecc.).
- modalità delle interfacce da utilizzare.
- ***architettura* ad oggetti del sistema: come il sistema è scomposto in sottosistemi, e questi in classi, a vali livelli di dettaglio.**

La progettazione OO **utilizza la stessa notazione dell'analisi**, specializzandone i modelli.

I processi di analisi, progettazione e codifica sono sempre iterativi.

Analisi dei requisiti

- Ha come scopo la definizione dei requisiti e la descrizione di uno specifico sistema da realizzare.
- Inizia di solito con un documento esprimente i requisiti in forma testuale, con una serie di discussioni col cliente, o con entrambi.
- I requisiti sono spesso espressi come *casi d'uso* del sistema, oppure come *user stories*.
- L'analisi si concentra sulla descrizione del sistema, del mondo esterno e dei loro costituenti fondamentali, e non sui dettagli di come tale sistema funziona.
- L'analisi dapprima considera le entità importanti dell'ambiente esterno e del sistema, e poi le raffina alla luce delle *responsabilità del sistema.*
- In un approccio agile, il sistema è sviluppato per iterazioni, realizzando successivamente le funzionalità descritte nelle *user stories*.
- Occorre comunque sempre una progettazione iniziale della struttura del sistema, per definirne almeno a grandi linee l'architettura OO

I principi dell'OOA/OOD

L'OOA/OOD si basa su di una analisi accurata dei principi utilizzati dagli uomini per gestire la complessità:

Astrazione

> ***Astrazione***. Il principio di ignorare gli aspetti di un soggetto che non sono importanti per lo scopo attuale, per concentrarsi maggiormente su quelli che lo sono.

- La maggior parte delle cose reali sono intrinsecamente complesse, e tale complessità va gestita.
- Quando usiamo il concetto di astrazione ammettiamo che stiamo considerando la cosa trattata come complessa; piuttosto che cercare di comprenderla interamente, ne selezioniamo solo una parte.

Astrazione Procedurale. Il principio che ogni operazione ben definita possa essere considerata dai suoi utenti come un'entità singola, nonostante tale operazione sia effettivamente realizzata da una sequenza di operazioni di più basso livello (ad esempio, la funzione sqrt(x).

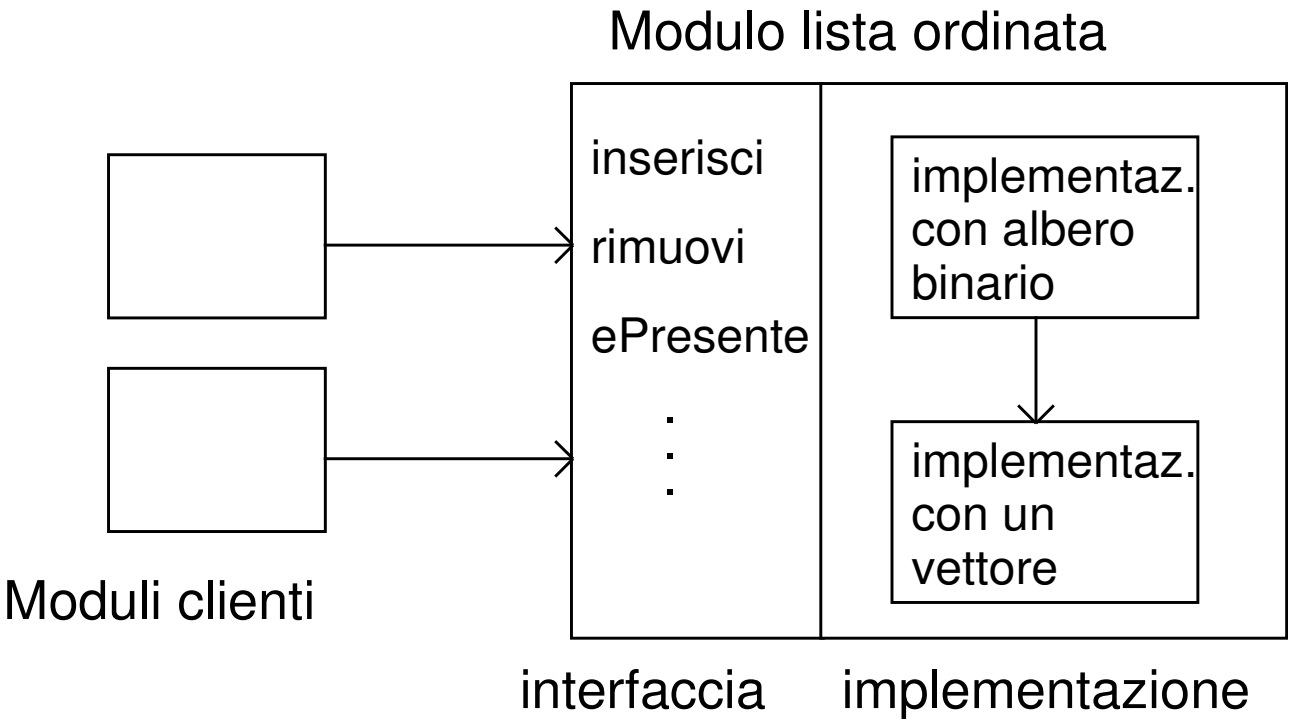
- L'*astrazione procedurale* è una delle forme di astrazione più usate.La scomposizione dei processi in sottoprocessi è uno dei metodi fondamentali per gestire la complessità.
- Un altro meccanismo di astrazione è l'*astrazione dei dati.*
- Questo principio è il fondamento per l'organizzazione del modo di pensare e per la specifica delle responsabilità dei sistemi OO.

Astrazione dei Dati. Il principio di definire un tipo di dato (ADT) in termini delle operazioni applicabili agli oggetti del tipo, col vincolo che i valori di tali oggetti si possano modificare ed osservare solo usando tali operazioni.

- Un ADT è un tipo definito dall'utente ed è composto da:
  - una struttura dati
  - delle procedure per operare su tali dati
- L'implementazione di un ADT si chiama *classe*
- Una classe equivale a un *tipo* (si possono definire variabili e puntatori della classe)
- Ai dati si può accedere *solo* tramite le procedure pubbliche (incapsulamento).
- Un modulo è composto da un'*interfaccia* esterna e da un'*implementazione* interna, nettamente separate

Un modulo riusabile:

- Interfaccia e implementazione sono separati.
- L'interno del modulo si può modificare senza alterarne l'interfaccia e quindi senza impatto sugli utilizzatori (clienti):



Incapsulamento

- Incapsulamento (occultamento dell'informazione): ogni componente del programma deve incapsulare una singola scelta di progetto.
- L'interfaccia di ciascun modulo è definita in modo tale da rivelare il meno possibile del suo funzionamento interno.
- L'incapsulamento minimizza le modifiche da fare sviluppando un nuovo sistema.

Ereditarietà

- Un meccanismo per esprimere le somiglianze tra le classi, semplificando la definizione delle classi simili ad una precedentemente definita.
- Permette di specificare attributi e operazioni comuni una volta sola, e di specializzare ed estenderli in casi specifici

L'identificazione dello scopo e delle caratteristiche di un sistema

Il principio "25 parole o meno". Sempre, sempre, sempre iniziate domandandovi "Qual è lo scopo del sistema, in 25 parole o meno?" Incorniciate questo scopo in una bella targhetta! Usate lo scopo del sistema per farvi guidare in ogni mossa dell'OOA/OOD del sistema.

La raccolta dei requisiti si può fare:

- Con un approccio tradizionale (OOA → OOD → OOP...)
- Tramite casi d'uso
- Tramite User Stories

User Stories

- Appartengono al cliente e sono scritte da lui
- Devono avere valore tangibile per il cliente
- Devono essere capite dal programmatore
- Devono essere stimabili
- Sono scritte su schede
- Sono discusse col cliente
- Sono validate tramite un test di accettazione, anch'esso descritto su una scheda
- **Lo sviluppo del sistema viene guidato dalle storie**
- **Ciò permette di utilizzare il time-boxing**

Scopo della progettazione

- **Definire l'architettura del sistema**, ed in particolare la sua scomposizione in sottosistemi, e questi in oggetti, a vari livelli
- Guidare la codifica
- Ottimizzare l'architettura per poter facilmente aggiungere nuove caratteristiche ed agevolare la manutenzione

Strumenti della progettazione

- Tecniche e metodi di analisi OO
- Principi e linee guida
- Refactoring
- Pattern

Notazione dell'OOD

- Diagrammi informali, documenti di progetto
- Linguaggio UML
- Schede CRC
- Pseudo-codice
- Prototipazione con linguaggi di alto livello
- Codice "*intention revealing*"

Cenni di UML

- UML (Unified Modeling Language) è un **linguaggio grafico** per:
  - specificare
  - visualizzare
  - realizzare
  - documentare

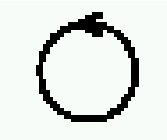
  i manufatti di un sistema software.
- Adottato da OMG (Object Management Group) dal 1997 (UML 1.1)
- Attualmente, siamo a UML 2.0.

Obiettivi di UML

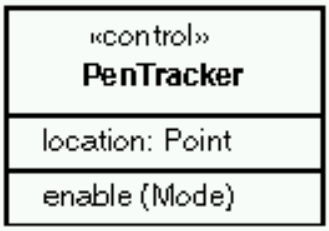
- Definire un linguaggio di modellazione semplice ma ricco di concetti
- Unificare i linguaggi di modellazione OO: Booch, OMT, Objectory e gli altri presenti sul mercato, definendo uno **standard**.
- Adottare le “best practices” dell’industria
- Considerare le tematiche attuali dello sviluppo del software:
  - scalabilità, distribuzione, concorrenza, ecc.
- Abilitare l’interscambio dei modelli OO e aiutare la definizione di “repository” per i modelli
- Poter essere usato con tutti i processi, durante tutto il ciclo di sviluppo e con diverse tecnologie di implementazione

I quattro costrutti grafici di UML

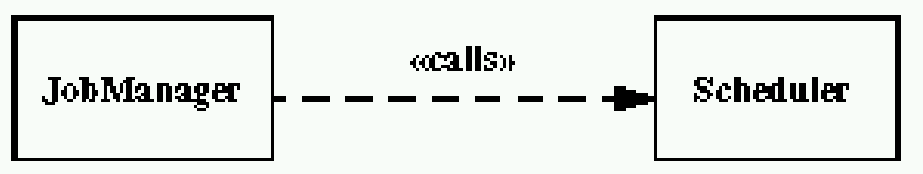
- Le icone:



- I simboli bidimensionali:



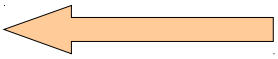
- I collegamenti:



- Le stringhe:

  integrate (f: Function, from: Real, to: Real)

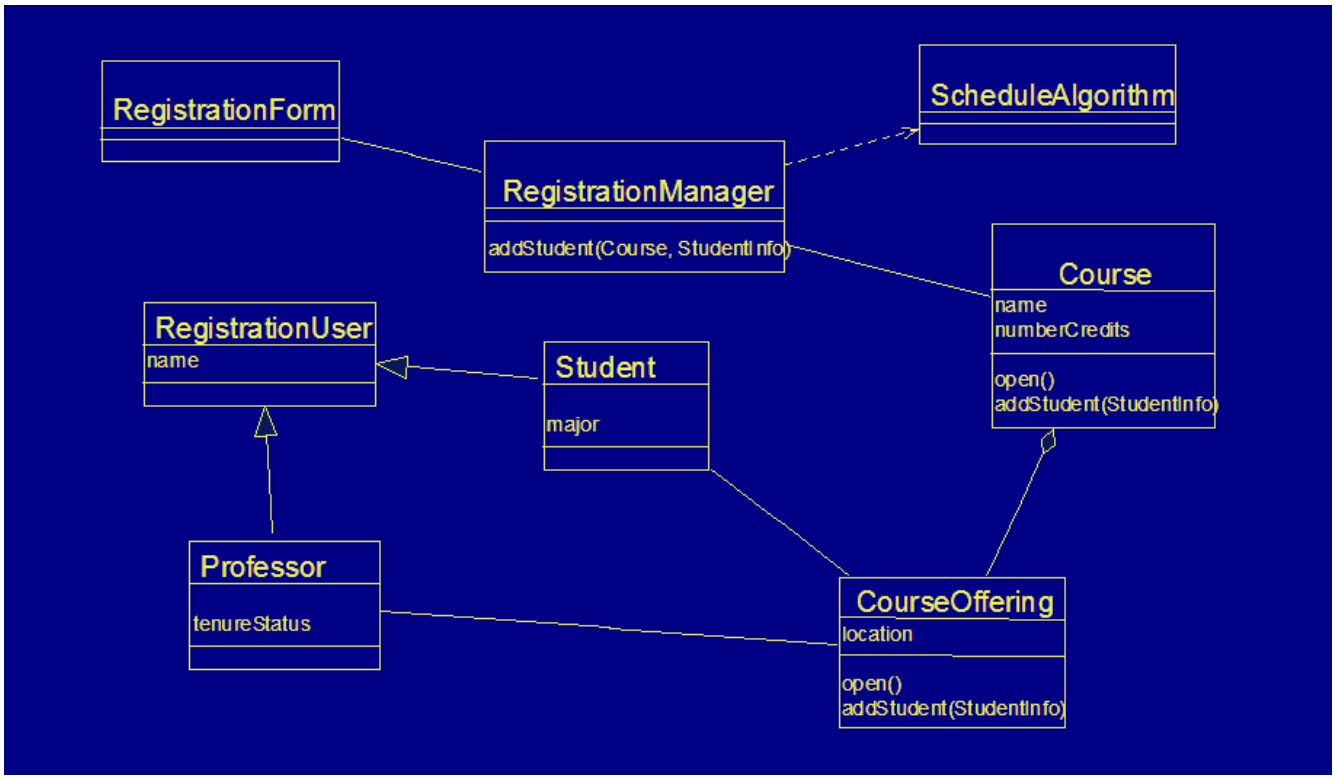
I diagrammi di UML 2.0

- Diagrammi strutturali statici:
  - Diagrammi di classe 
  - Diagrammi degli oggetti
- Diagrammi d'implementazione:
  - Diagrammi dei componenti
  - Diagrammi di dispiegamento (deployment)
- Diagramma dei casi d'uso
- Diagrammi dinamici:
  - Diagramma di stato 
  - Diagrammi di interazione:
    - Diagrammi sequenziali 
    - Diagrammi di collaborazione
  - Diagrammi di attività

Diagrammi di classe

- Mostrano le classi e le loro relazioni
- Sono a un livello di astrazione alto
- Sono utilizzabili per generare il codice con le strutture dati e le dichiarazioni dei metodi
- Costrutti fondamentali:
  - Package
  - Classe (nome, attributi, operazioni)
  - Interfaccia
  - Ereditarietà
  - Associazione
  - Dipendenza

Un diagramma delle classi UML



Vantaggi e svantaggi di UML

- **Vantaggi**:
  - E' standard e universalmente noto
  - Permette di rappresentare i concetti OO
  - Esistono molti strumenti di supporto
- **Svantaggi**
  - E' follemente complesso e ridondante
  - Porta facilmente alla sindrome della "analysis-paralysis"
  - Strumenti e training possono essere molto costosi
- **In conclusione**:
  - A dosi omeopatiche, può essere benefico
  - In dosi massicce è un potente veleno paralizzante per quasi tutti i progetti sw!

Analysis-Paralysis

Copyright © 2002 United Feature Syndicate, Inc.

Analisi e Progettazione agili

- Un sistema software di una certa dimensione **deve** essere progettato correttamente
  - Se no, non sarà manutenibile
- L'approccio “upfront” (OOA → OOD → OOP) non è abbastanza dinamico
- La soluzione agile:
  - OOA/OOD upfront iniziale e limitata:
    - solo per fondare l'architettura
    - con “buoni principi” di OOD
  - Uso di UML, ma anche di schede CRC o notazioni in pseudo-codice
  - Sviluppo iterativo-incrementale guidato da “features”
  - Refactoring (l'OOD evolve col tempo)

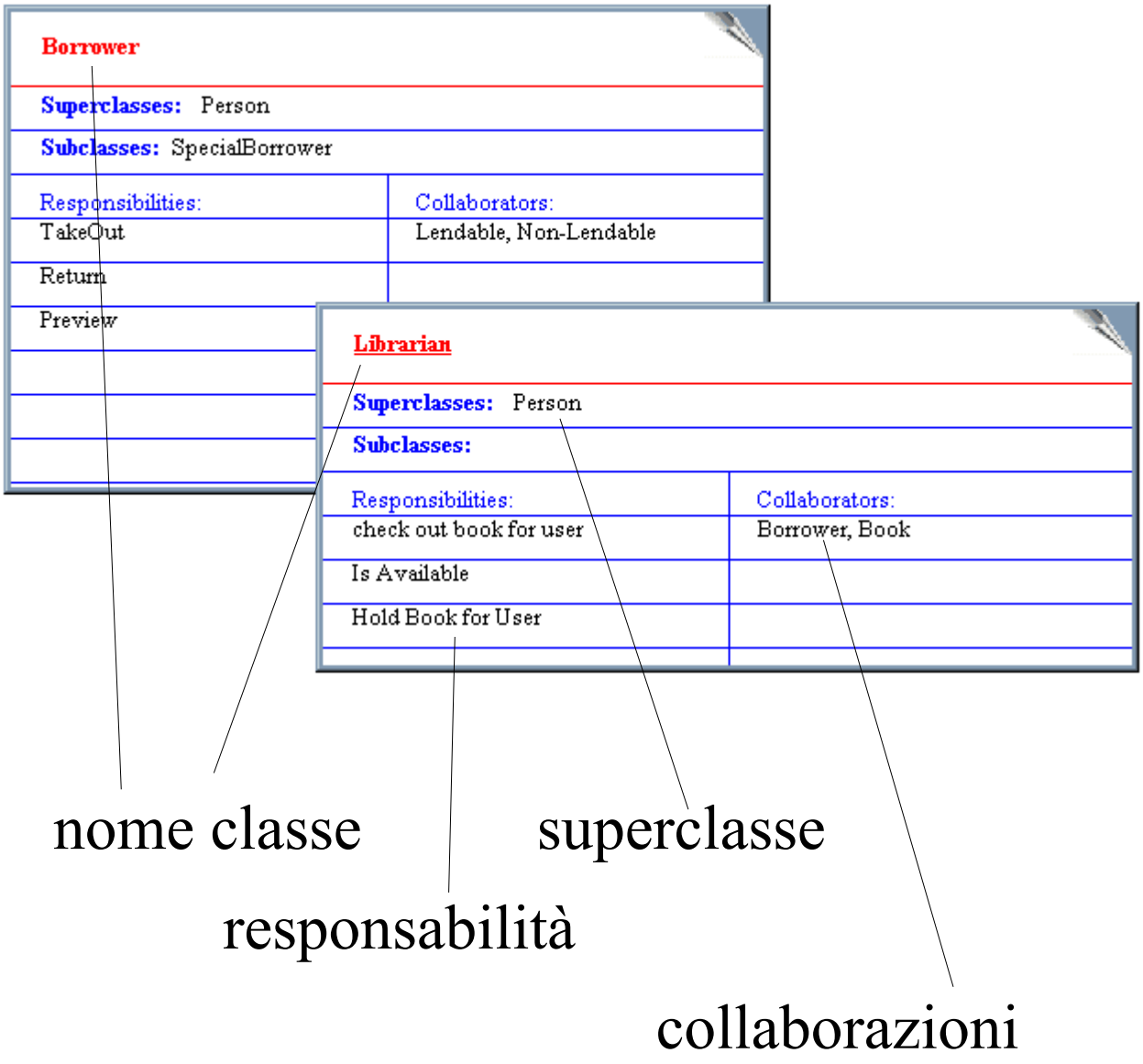
Un buon punto d'inizio: metodo CRC

- E' un metodo indipendente dal linguaggio per progettare software OO
- E' costituito dalle seguenti attività:
  - identificazione degli oggetti a partire dai requisiti del sistema;
  - determinazione delle responsabilità del sistema, ed assegnazione delle responsabilità ad ogni singolo oggetto;
  - determinazione delle collaborazioni tra oggetti necessarie;
  - identificazione delle gerarchie di classi;
  - identificazione dei sottosistemi;
  - specifica dettagliata dell'interfaccia per ciascuna classe.
- E' considerato il miglior metodo per OOD perché non si basa sulla struttura ma sul comportamento

Class-Responsibility-Collaboration

- **Class**:
  - Il metodo si basa sull'individuazione e sulla definizione delle classi
  - Esse sono descritte su schede di cartoncino
  - Date le classi, se ne definiscono anche le gerarchie di ereditarietà
- **Responsibility**:
  - Ogni classe è specificata in funzione delle sue responsabilità
  - Tali responsabilità portano alla specifica dell'interfaccia della classe
- **Collaboration**:
  - Le classi devono collaborare tra loro
  - Ciò porta alla struttura delle dipendenze tra classi

Schede CRC


Borrower
Superclasses: Person
Subclasses: SpecialBorrower
Responsibilities:
Collaborators:
TakeOut
Lendable, Non-Lendable
Return
Preview
Librarian
Superclasses: Person
Subclasses:
Responsibilities:
Collaborators:
check out book for user
Borrower, Book
Is Available
Hold Book for User
nome classe
superclasse
responsabilità
collaborazioni

L'identificazione degli oggetti

Oggetto**. **Un'astrazione di qualche cosa nel dominio del problema, che riflette la capacità del sistema a mantenere dell'informazione su di essa, ad interagire con essa o ad entrambe le cose;
un incapsulamento di valori di attributi e delle operazioni esclusive su tali attributi. (sinonimo: un'*istanza*)

Classe. Una descrizione di uno o più oggetti con un insieme uguale di attributi ed operazioni, compresa la descrizione di come creare nuovi oggetti della classe

- A rigori, il metodo CRC descrive dei *tipi*:

Tipo: La descrizione di uno o più oggetti in grado di rispondere agli stessi messaggi, cioè con la stessa interfaccia (le stesse operazioni pubbliche).

Dove cercare le classi

- La fonte principale in cui cercare è il *documento dei requisiti*, una descrizione testuale del sistema da realizzare e del suo funzionamento, scritto dal o concordato col cliente.
- In alternativa, si cerca nei casi d'uso o nelle Users Stories
- Altre fonti sono:
  - Colloqui coi futuri utenti
  - Altri sistemi nello stesso dominio.
  - Enciclopedie, nomenclature e documenti tecnici che descrivono il dominio.
  - Diagrammi di tutti i tipi (a blocchi, di interfaccia, architetturali, di flusso dei dati di livello elevato...) descriventi il sistema.

Per trovare le classi

- Elencate i nomi (semplici o composti) che compaiono nei documenti raccolti, convertendoli al singolare.
- Eliminate i nomi che ovviamente non si riferiscono a classi.
- Eliminate i nomi che chiaramente indicano attributi (dati di basso livello) e operazioni.
- Scegliete un solo termine significativo se più parole indicano lo stesso concetto.
- Attenzione agli aggettivi e agli attributi: possono indicare oggetti diversi, usi diversi dello stesso oggetto, o essere irrilevanti. Ad es., "impiegato bravo" è irrilevante, ma "impiegato temporaneo" è probabilmente una nuova classe.

- Attenzione alle frasi passive, impersonali o con soggetti fuori dal sistema: rendetele attive ed esplicite, perché possono mascherare entità rilevanti per il sistema.

- Successivamente, lo studio e la costruzione del modello, così come lo sviluppo dei prototipi, porteranno a raffinare l'OOD e a trovare nuove classi.
- L'individuazione delle gerarchie di ereditarietà porta facilmente alla scoperta di nuove classi.

Che cosa cercare (*tipologie* delle classi)

Attori: persone o organizzazioni che hanno un ruolo nel sistema.

- Esempi: *persona, organizzazione (agenzia, associazione, ente, fondazione, società)*.

Ruoli degli attori: analizzate come gli attori partecipano al sistema.

- Esempi: *agente, amministratore, cassiere, civile, cliente, compratore, consulente, delegato, dipendente, dirigente, distributore, donatore, dottore, fornitore, impiegato, ingegnere, insegnante, investitore, lavoratore, maestro, magazziniere, membro, operaio, operatore, partecipante, produttore, professore, proprietario, rappresentante, rappresentante di commercio, responsabile, ricevente, richiedente, rivenditore, sottoscrittore, studente, supervisore, tecnico, ufficiale, utente*.

Posti: ubicazioni ove risiedono le cose, o che contengono altri oggetti.

- Esempi: *aeroporto, agenzia, banca, casa, centro servizi, clinica, contenitore, deposito, entità geografica, fabbrica, garage, hangar, linea di montaggio, magazzino, negozio, ospedale, regione, stanza, stazione, zona.*

Cose tangibili: oggetti concreti del dominio del problema (visitate i luoghi del sistema e scegliete tra le cose tangibili che vedete).

- Esempi: *cassa, cassetto, auto, treno, PC, cellulare...*

Eventi o transazioni: eventi che il sistema deve memorizzare. Una transazione è un momento nel tempo (ad es., una vendita) o un intervallo di tempo (ad es., un affitto).

- Le transazioni derivano da:
  - Una GUI (dati inseriti dall'utente).
  - Un altro oggetto che controlla e fa registrare l'avvenimento di un evento significativo.
  - Un altro sistema.

Esempi: *accordo, acquisto, affitto, assegnazione, autorizzazione, consegna, contratto, deposito, fattura, incidente, invio, locazione, noleggio, nota, ordine, pagamento, parcella, prenotazione, rapporto problema, registrazione, richiesta, ricevimento, rilascio, rimborso, rinuncia, sottoscrizione, trasferimento, tratta, vendita, versamento*.

- Quasi tutte le transazioni consistono di uno o più elementi *riga transazione*.

Associati: oggetti che devono conoscersi, ma la cui associazione non deve ricordare alcun dato.

- Esempi: *aereo-pista, conducente-veicolo, edificio-sensore, piattaforma carico-ordine, ordine-trasporto, camion-piattaforma carico, treno-binario*.

Modelli e loro elementi specifici: oggetti descrittivi (ad es., un modello di auto: *Fiat Punto SX*, e le specifiche istanze: le effettive auto di quel modello che il sistema deve registrare).

- Esempi: *categoria prezzo-elemento con quel prezzo, categoria tassa-tassa specifica, descrizione compito-compito specifico, modello aereo-aereo specifico, modello automobile-automobile specifica*.

Sistemi e Dispositivi: altri sistemi e dispositivi con cui il sistema in esame deve interagire.

- Esempi: *attuatori, DBMS, Internet, LAN, rete, semafori, sensori*.

Collezioni di Oggetti: cercate oggetti che appartengono a una collezione; date un nome alla collezione ed ai singoli oggetti.

- Il nome della collezione, in mancanza di meglio, può essere il nome al plurale dei suoi componenti.
- Esempi di collezioni di persone: *corpo docente, dipartimento, gruppo, organizzazione, società, squadra, team*.
- Esempi di luoghi collezioni: *aeroporto, campus, catena di negozi*.
- Esempi di collezioni di cose: *coda, collezione, gruppo, insieme, lista, log*.

Oggetti contenitori: cercate oggetti che contengono altri oggetti.

- Esempi: *aereo, aeroporto, armadio, banca, camera, cassaforte, caveau, classificatore, deposito, edificio, garage, hangar, magazzino, negozio, ospedale, ripostiglio, sala*.

Criteri di scelta per determinare se includere o no una classe in un modello

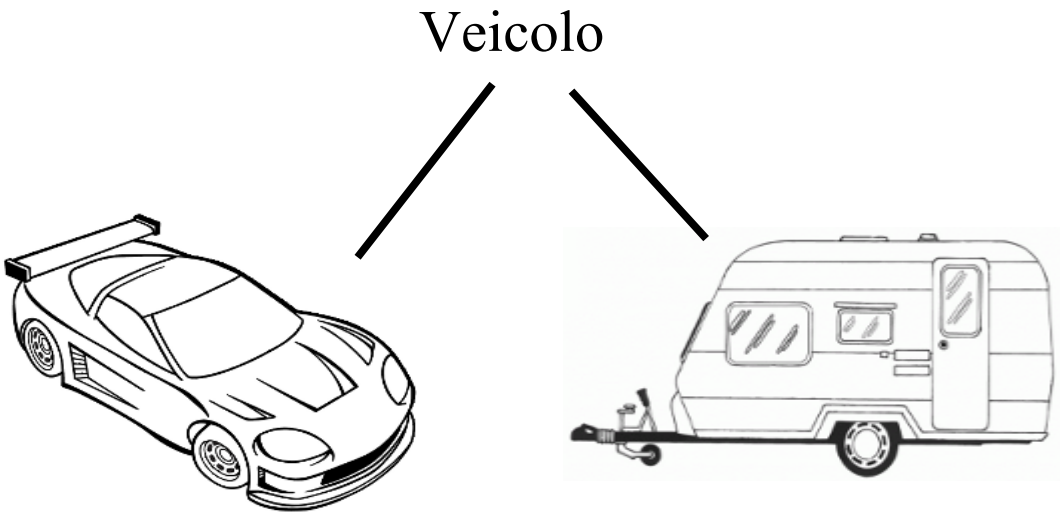
- ***Principio generale***: ponetevi sempre le domande:
  - "E' nella responsabilità del sistema ricordare qualcosa o interagire con questo oggetto?"
  - "Quali sono le sue responsabilità?"
- ***Cose da ricordare***. Il sistema deve ricordare qualcosa sugli oggetti della classe? Se no, la validità e l'importanza della classe è dubbia
- ***Comportamento richiesto***. Un oggetto deve fornire un dato comportamento (elaborazioni)?
- ***Doppioni***. Questo oggetto ha le stesse responsabilità di un altro: ne è dunque un doppione?

- ***(Di solito) più di un oggetto in una classe***. Le classi con un solo oggetto sono sospette
- ***Responsabilità sempre applicabili***. Esistono responsabilità ben definite solo per alcuni oggetti della classe? Se sì, c'è dell'ereditarietà.
  Ad esempio: classe **Veicolo** con responsabilità di ricordare la cilindrata; ma i rimorchi non hanno motore, e quindi non hanno cilindrata! Quindi esiste la struttura:

Veicolo

VeicoloAMotore

Rimorchio



Definizione delle responsabilità

- Le responsabilità delle classi del sistema sono di due tipi:
  - mantenere delle informazioni (uno stato)
  - effettuare delle azioni (un comportamento)
- Le responsabilità servono a definire lo scopo di un oggetto, ed il suo ruolo nel sistema.
- Le responsabilità rappresentano solo i servizi disponibili pubblicamente, non quelli privati interni alla classe, la cui definizione sarebbe prematura
- Le due principali sorgenti di informazione sono i requisiti e le classi appena trovate

Identificazione delle responsabilità

- esaminare accuratamente i requisiti, alla ricerca delle *azioni*. Elencare i verbi che rappresentano azioni fatte da un oggetto del sistema;
- annotare tutte le informazioni che gli oggetti entro il sistema devono mantenere e gestire
- fare una simulazione (mentale, impersonando gli oggetti o usando diagrammi) del sistema in azione. Cercare i posti ove il sistema esegue un'azione in risposta a input esterni. Fare diagrammi di stato del sistema
- utilizzare le classi già identificate: per il semplice fatto di esistere, devono avere almeno una responsabilità
- quali informazioni deve contenere una classe, e quali azioni deve svolgere per ottemperare al suo scopo?

Come assegnare le responsabilità

- Le responsabilità, una volta identificate, vanno assegnate ad una o a più classi
- In caso di dubbio, assegnare le responsabilità alle classi seguendo i seguenti criteri:
  - distribuire in modo bilanciato l'intelligenza nel sistema (molti oggetti che si scambiano messaggi)
  - assegnare le responsabilità al livello più generale possibile salendo la gerarchia delle classi
  - mantenere il comportamente collegato alle informazioni ad esso necessarie
  - mantenere le informazioni su una cosa in un solo posto

- Tutti i principi di “buona” OOD che vedremo, sono utilizzabili per decidere come assegnare le responsabilità
- Dati dei requisiti, non esiste un modo univoco, né un unico modo dimostrabilmente migliore, per determinare le classi del sistema e per assegnar loro le responsabilità.
- Ogni problema ammette soluzioni orientate agli oggetti diverse ed ugualmente valide
- Vi sono però molte più soluzioni scorrette o inefficienti

Specifica delle classi

- Le classi così trovate, oltre a comparire nel diagramma vanno specificate in schede.
- Per ogni classe, si ha:
  - Nome, semplice e descrittivo.
  - Breve descrizione di un oggetto della classe.
  - Lista delle responsabilità.
  - Lista delle collaborazioni
  - Nome della superclasse
  - Astratta o no.

Un altro esempio di scheda CRC:

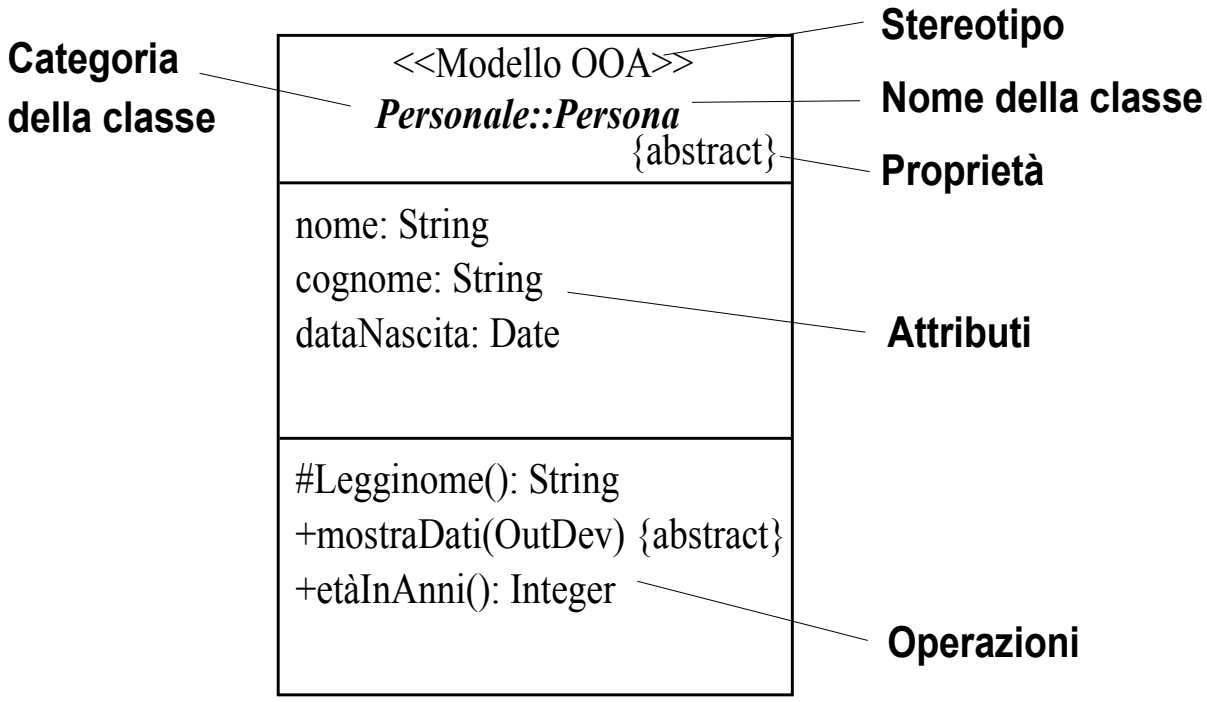
Classe: **Evento**	Superclasse: **Nessuna**	Astratta
Descrizione: Un evento rappresenta qualche cosa di significativo per il sistema, che accade ad un preciso istante di tempo e che muta lo stato del sistema.		
Responsabilità: 1. Un Evento conosce la propria data e ora. 2. Un Evento sa eseguire o attivare le elaborazioni che, in conseguenza al proprio accadere, mutano lo stato del sistema.		

- Questa classe non ha (per ora) collaborazioni

Dalla specifica CRC alla struttura dati

- Le specifiche delle classi su schede CRC arrivano a definire i protocolli, ovvero le "firme" dei metodi delle classi.
- Per poter codificare una classe, occorre però anche darne la struttura dati.
- Tale struttura dati è composta da:
  - attributi
  - associazioni con altre classi
- La struttura dati:
  - permette di soddisfare alle responsabilità di mantenere delle informazioni
  - permette di accedere ad oggetti necessari per effettuare le elaborazioni

Notazione (UML) per le classi



- La sezione superiore contiene il nome della classe (in grassetto) e può inoltre contenere:
  - Se importata, il nome del pacchetto: NomePacchetto::NomeClasse.
  - Lo stereotipo della classe, posto tra "<< >>" sopra il nome (es. controllore, attore, evento, interfaccia utente, database, ecc.).
  - Ulteriori proprietà, poste sotto il nome tra '{}' e scritte in corsivo (ad es. abstract, readonly, progettista=Carlo Rossi...).

Notazione (UML) per le classi

- La sezione mediana contiene gli attributi (solo quelli più significativi)
- La sezione inferiore contiene le operazioni (solo quelle pubbliche più significative)
- La scelta del nome della classe:
  - Il nome deve indicare al singolare un oggetto della classe e di solito inizia con maiuscola.
  - Occorre usare nomi familiari all'utente o all'esperto del dominio del problema.
- Classi astratte:
  - Se una classe non ha istanze, ma è usata solo per specificare proprietà comuni alle sue sottoclassi, si chiama classe astratta ed è denotata dalla proprietà abstract.
  - Il nome delle classi astratte è denotato in corsivo.
  - Anche un'operazione può essere astratta, nel qual caso è denotata in corsivo.

Attributi

Attributo. Un attributo è un dato (informazione di stato) per il quale ciascun oggetto in una classe ha un proprio valore.

- Gli attributi descrivono dati di tipo primitivo; i dati di tipo di classe sono collegati tramite associazioni
- Sia attributi che associazioni possono essere *di classe*, cioè unici nella classe e quindi non parte della struttura dati delle istanze della classe.
- Gli attributi si elencano nella sezione intermedia del rettangolo relativo ad una classe
- Ogni attributo è caratterizzato dal nome, dal tipo e opzionalmente da un valore di default:

  nomeAttributo: Tipo = valoreDefault

- Gli attributi di classe hanno la definizione sottolineata:

  numIstanze: int = 0

Identificazione degli attributi

- Domande da porsi per ogni oggetto di una classe:
  - "Come è descritto nel mondo reale l'oggetto che mi corrisponde?"
  - "Come sono descritto nel contesto delle responsabilità di questo sistema?"
  - "Che cosa devo conoscere?"
  - "Come faccio ad ottenere tali informazioni?"
  - "Quali informazioni di stato devo ricordare nel tempo?"
  - "In quali stati posso essere?"
- Fate sì che ogni attributo catturi un "concetto atomico":
  - Un valore singolo
  - Un gruppo di valori strettamente legati, come un indirizzo o una data
- Esempi di attributi: *data, indirizzo, numero, nome, ora, soglia, stato, stato operativo, telefono, tipo*.

I tipici attributi delle tipologie di classe

Tipologia	Attributi tipici
Attori	codice fiscale, indirizzo, matricola, nome, telefono.
Ruoli degli attori	data, livello autorizzazione, numero, ora, parola d'ordine.
Posti	altezza, indirizzo, latitudine, longitudine, numero, nome.
Cose tangibili	codice, colore, data, marca, modello, numero, ora, peso, quantità, versione.
Eventi o transazioni	data, numero, ora, stato.
Modelli	codice universale prodotto (UPC), codice, descrizione, dimensioni, nome, peso.
Elementi specifici	data acquisto, matricola, numero serie, targa
Sistemi e Dispositivi	nome, numero, stato operativo

Verifica di attributi ed associazioni

- Mettete ciascun attributo ed associazione entro la classe che lo descrive meglio.
- Se memorizzare o meno un attributo ricalcolabile in ogni momento è una decisione di codifica. Nell'OOD, specificate l'operazione di calcolo, senza l'attributo sempre ricalcolabile corrispondente.
- Identificatori impliciti, "id", si possono usare nel testo delle specifiche, quando necessario. Tali identificatori servono a identificare univocamente gli oggetti e sono generati dal sistema
- Attributi che indicano tipo o categoria dell'oggetto non sono necessari: devono essere **operazioni** definite nella gerarchia dei tipi. (Operazione *nomeClasse*()).
- Applicate l'ereditarietà:
  - Posizionate gli attributi e le associazioni più generali più in alto
  - Posizionate gli attributi e le associazioni specializzati più in basso

Attenzione ai casi particolari!

- Un attributo ha valore "*sì o no*"? Il nome stesso può essere un valore di un'enumerazione. Ad es.: *tassabile* (sì o no) va convertito in *statoTassazione* (tassabile, esente, ridotto, ecc.).
- Se un attributo varia nel tempo, ed occorre tener traccia dei cambiamenti, *aggiungete una classe* i cui oggetti contengano il valore, ma anche la data del cambiamento, e eventualmente il responsabile, ecc.
- Attenzione alle classi con un solo attributo: spesso tali classi sono attributi di un'altra classe.
- Attributi con valori "non applicabile" o a valori multipli: possono nascondere ereditarietà o una nuova classe

- Una classe, dopo attento esame, non ha attributi, nemmeno ereditati?
  - Ha associazioni: tutto OK!
  - Può essere un sistema esterno, con cui occorre interagire ma che non ha stato (difficile!).
  - Se esiste un solo oggetto di quella classe, è una "bolla" funzionale: una funzione che va assegnata come operazione ad un'altra classe.
  - Avrebbe attributi nel mondo reale, ma non in questo dominio del problema e entro le responsabilità del sistema. Allora, eliminatela!

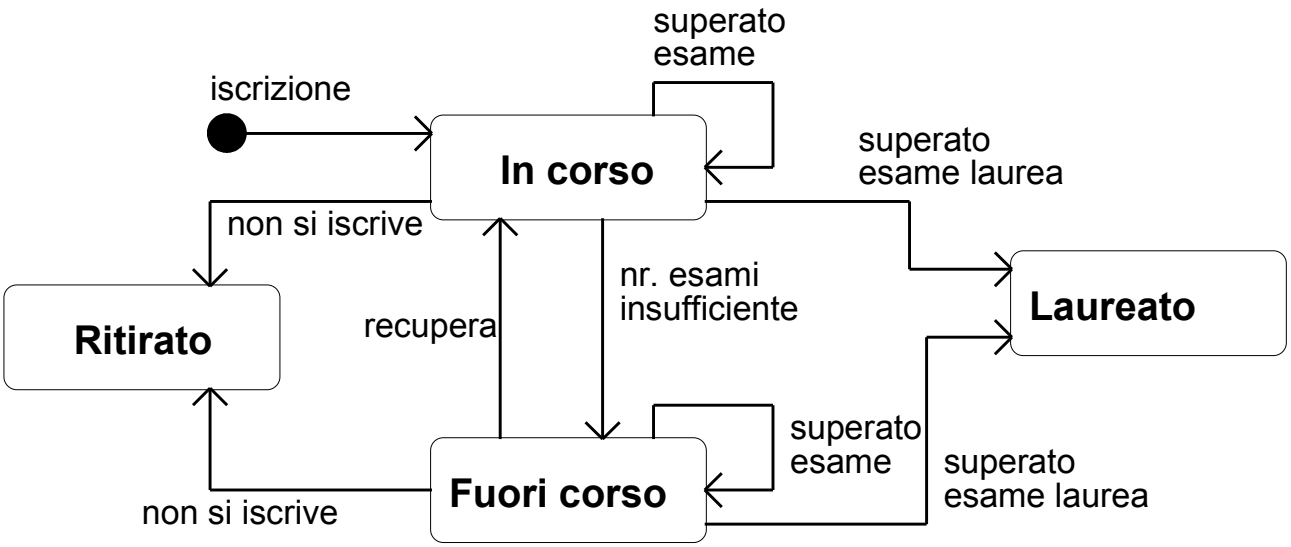
Specifica degli attributi

- Date il nome agli attributi:
  - Vocabolario standard del dominio del problema.
  - Leggibile.
  - Non mettete valori nel nome (ad es., diesel invece di tipoMotore).
- Descrizione di una o due righe.
- Specificare se l'attributo è di classe o no.
- Esprimetene i vincoli:
  - Unità di misura, intervallo, limiti, enumerazioni; precisione; valore di "default".
  - Sempre presente (si o no)?
  - Vincoli di creazione o di accesso?
  - Vincoli dovuti ai valori di altri attributi?
  - Codici di applicabilità negli stati.
- Visibilità: pubblico (+), protetto (#) o privato (-). **Un attributo deve sempre essere protetto o privato!**

Stati degli oggetti

- In senso generale, lo *stato* di un oggetto è dato dal valore dei suoi attributi e delle sue associazioni.
- In molti domini applicativi, esistono oggetti che, a seconda del proprio stato, rispondono in maniera diversa ai messaggi ricevuti. Esempi sono:
  - Dispositivi (stato: spento, in attesa, operativo, guasto, ecc.).
  - Transazioni complesse (stato: in definizione, in esecuzione, completata, fallita, ecc.).
- In tali casi, ha senso definire un *diagramma di stato* per l'oggetto, mostrando i possibili stati e gli eventi che attivano transizioni da uno stato all'altro.
- Anche nell'analisi tradizionale si parla di stato del sistema o di sottosistemi. Nell'OOD lo stato si definisce a livello dei singoli oggetti, ed è quindi più granulare.

- Esempio: stato di uno studente in un sistema informativo universitario:



- Nell'UML i diagrammi di stato seguono quasi totalmente la notazione di Harel, che è molto ricca.
- Uno stato iniziale o di entrata si denota con un pallino nero.

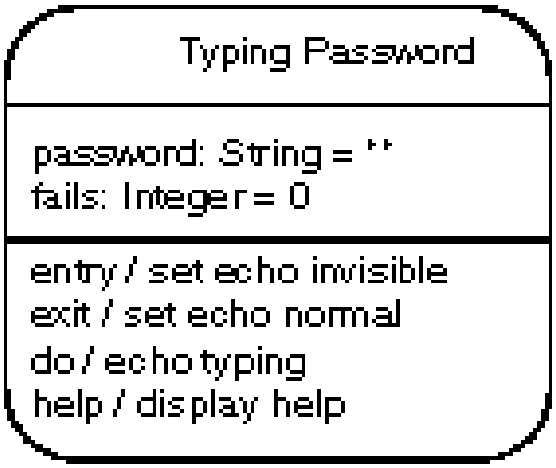
- Uno stato finale o di uscita si denota con un pallino nero circondato da un cerchio:

Identificazione degli stati

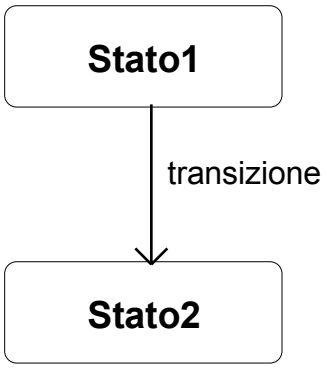
- Esaminate i valori possibili degli attributi.
- Determinate se le responsabilità del sistema comprendono comportamenti diversi per tali valori possibili.
- Descrivete gli stati e le transizioni in un diagramma di stato degli oggetti, da associare alla documentazione della classe.
- Riassumete gli eventuali attributi dipendenti dallo stato usando una tabella attributi/stati
- Riassumete le operazioni dipendenti dallo stato usando una tabella operazioni/stati

Notazione UML

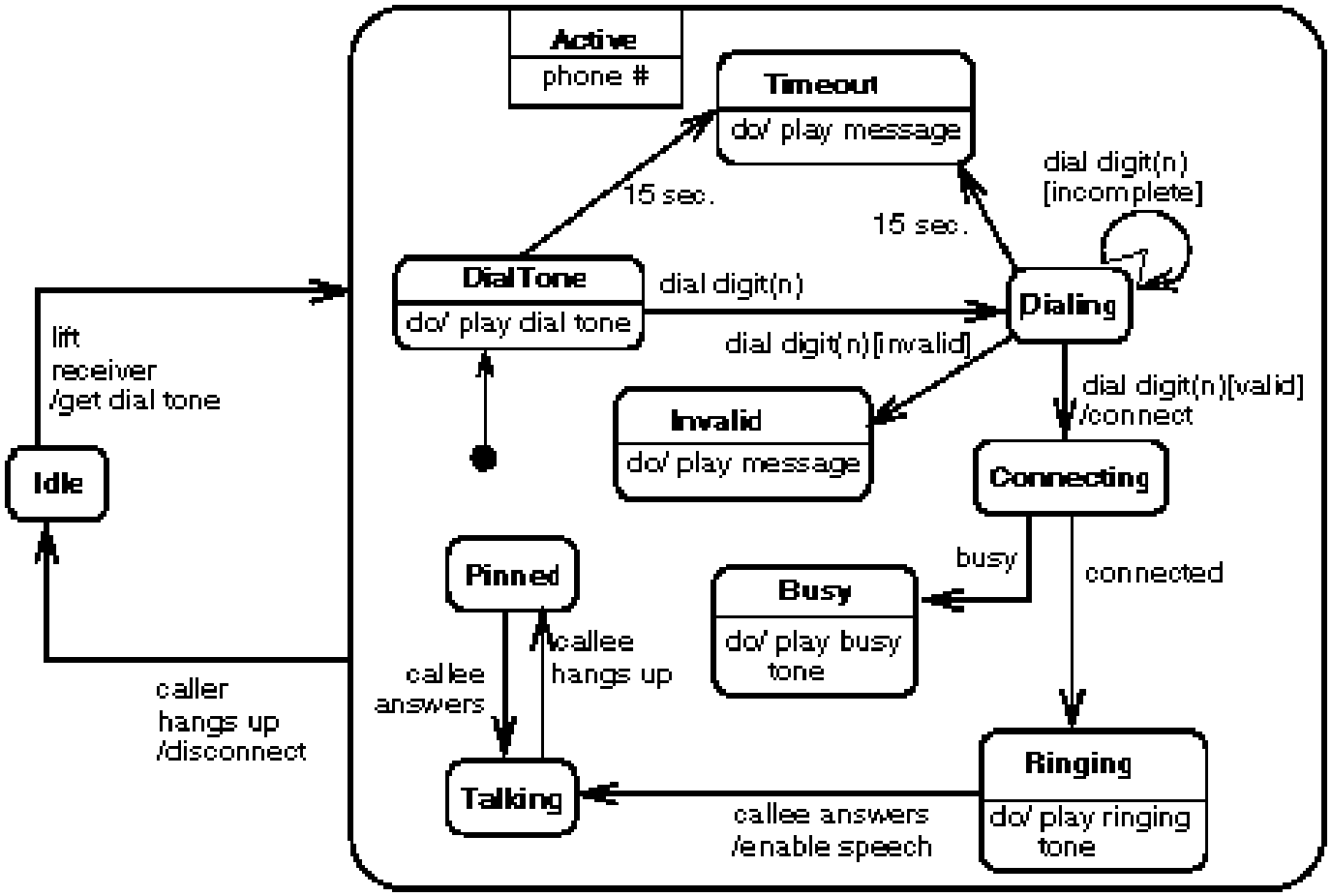
- Uno stato è rappresentato da un rettangolo coi bordi arrotondati, col nome (opzionale) al centro.
- Esso può avere due altri compartimenti:
  - in quello centrale sono elencate le variabili di stato coi valori relativi per quello in oggetto
  - in quello inferiore sono elencate le azioni effettuate in quello stato, coi prefissi “entry /”, “exit /” e “do /”:



- Una transizione è rappresentata da una freccia che congiunge due stati:



Stati multipli



- In UML, uno stato può essere espanso in sottostati, nel qual caso il rettangolo che lo rappresenta contiene un sotto-diagramma di stato

Transizioni

- Una transizione tra due o più stati è scatenata da un evento.
- Un evento è caratterizzato da un nome e da eventuali parametri associati. Gli eventi possono essere descritti in un diagramma delle classi, con lo stereotipo “event” o “signal”
- Una transizione è denotata da una freccia, etichettata col nome dell’evento scatenante, più eventualmente la condizione (un’espressione Booleana), le azioni conseguenti ed una o più “send-clause”:

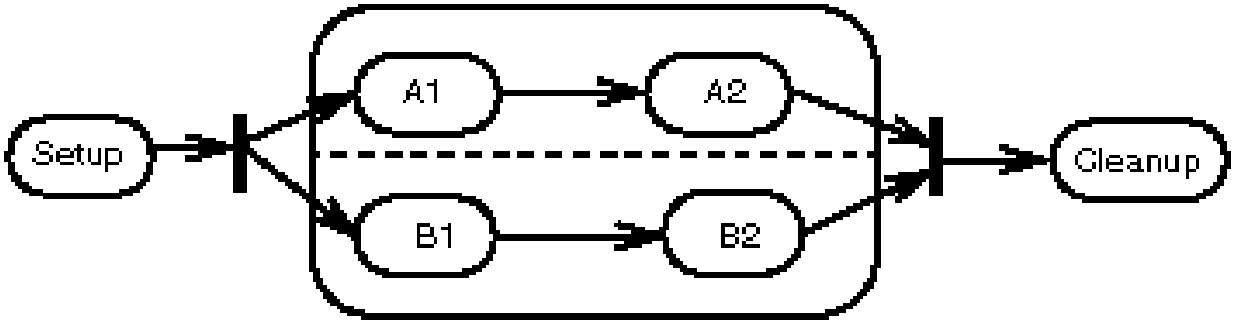
  evento(param...) [condizione] / azioni ^ send-clause

- Esempio:

  right-mouse-down (location) [location in window] / object := pick-object (location) ^ object.highlight ()

Transizioni complesse

- Vi possono essere transizioni complesse, aventi più di uno stato in ingresso e/o in uscita.
- Esse sono denotate da più frecce, e da una barra verticale spessa intermedia, etichettata dal nome della transizione:



- Se la barra è preceduta da più frecce, queste devono essere tutte abilitate perché la transizione scatti.
- Se la barra è seguita da più frecce, queste devono essere collegate a stati eseguibili in modo concorrente.