Analisi e progettazione orientate agli oggetti

Obiettivi della lezione

- Che cos'è la **progettazione**?
  - Un processo che *anticipa* un futuro artefatto mediante problem solving creativo
- Quali sono le regole di **buona** progettazione?
- Aspetti critici nel progetto del software a **oggetti**
  - Analisi delle alternative
  - Iterazioni (è raro azzeccarla alla prima)
  - Fattorizzazioni (semplificare il design)
  - Architetture di riferimento (tecnologie e standard)
- Come si **impara** a progettare a oggetti?

Discorso sul metodo

- La progettazione si apprende studiando i principali paradigmi evolutisi in relazione al progresso della tecnologia e delle applicazioni; esistono oggi parecchi paradigmi di progettazione
  - *Metodologia*
    - Insieme di metodi, regole e postulati impiegati da una disciplina
    - Analisi dei principi di conoscenza in un dominio specifico; studio dei metodi di analisi di tale dominio
  - *Paradigma*
    - Esempio, modello
    - Il paradigma attualmente dominante è l'"object-oriented" (OO), che si basa sulla costruzione di modelli

Analisi e progettazione

L'**analisi** si occupa di:

- Definire la soluzione giusta per il problema giusto

La **progettazione** si occupa di

- Descrivere (anticipare) una soluzione al problema mediante un **modello**

Un *modello* è una rappresentazione semplificata della realtà che contiene informazioni ottenute focalizzando l'attenzione su alcuni aspetti cruciali e ignorando alcuni dettagli

I modelli del sw richiedono più viste

- **Un buon modello include almeno quattro viste:**
  - Vista **Teleologica** (scopo)
  - Vista **Funzionale** (compiti in relazione allo scopo)
  - Vista **Strutturale** (elementi e relazioni)
  - Vista **Comportamentale** (interazione dinamica tra Funzionale e Strutturale)

Esempio: in UML abbiamo

Use Cases Interactions between a User and the system.	**Interaction Diagrams** Behavior of core business elements in process.	**Class Diagrams** Basic business elements and their relationships.
Process Description Overview of process goals, participants and collaboration.	**State Diagrams** Life cycle of core system elements in process.	**Deployment Diagrams** Actual structure of the final system
Funzione	*Comportamento*	*Struttura*

Perché più viste

• La vista **teleologica** descrive scopo e usi del sistema da progettare (eg. Documento di Marketing)

• La vista **funzionale** descrive il comportamento di un sistema in relazione allo scopo (eg. Casi d'uso)

• La vista **comportamentale** descrive il comportamento di un sistema in relazione ai cambiamenti del suo ambiente

• La vista **strutturale** descrive i componenti di un sistema e le loro relazioni

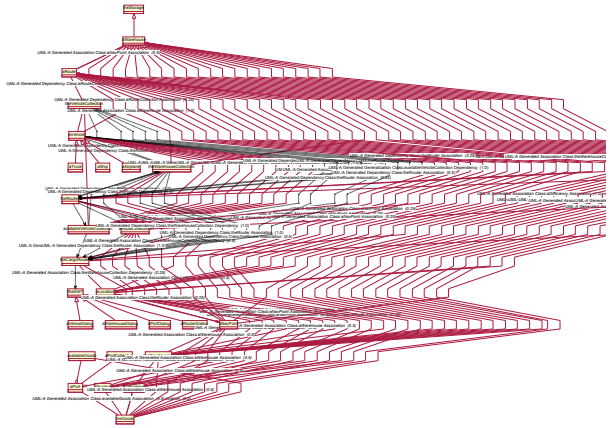
Le prime due viste sono "**soggettive**" (descrivono le intenzioni del progettista), le seconde "**oggettive**" (descrivono proprietà future del sistema)

Progettazione

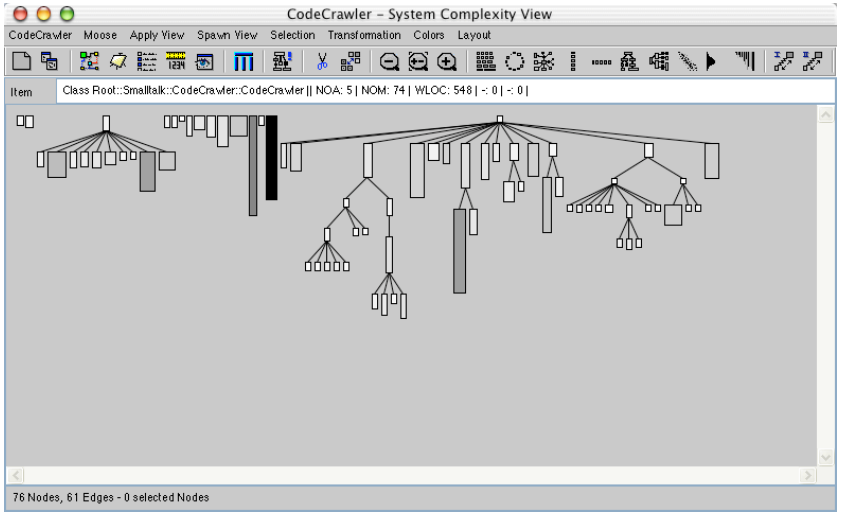
- Dopo l'analisi, distinguiamo almeno due fasi di progettazione:
  - Progettazione **architetturale**
    - Decomporre il sistema in moduli
    - Determinare le relazioni tra i moduli
  - Progettazione **dettagliata**
    - Specifica delle interfacce dei moduli
    - Descrizione funzionale dei moduli

Complessità del software

- Vista "sorgente" di un'applicazione"

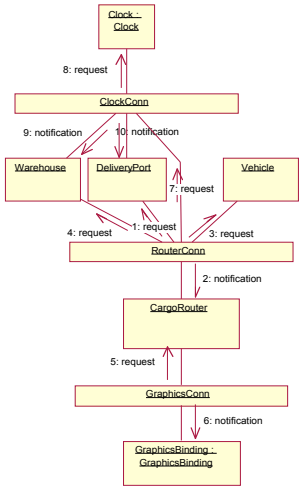


Vista di “reverse engineering”



L’architettura permette di controllare la complessità

- Vista architetturale
- “We can think of the architecture of a system as the common vision that all the workers (i.e., developers and other stakeholders) must agree on or at least accept”



Cosa NON è un’architettura

- La maggior parte delle classi con operazioni, interfacce e attributi privati (nascosti)
- Meno del 10% delle classi sono rilevanti per l’architettura (l’altro 90% non è visibile)
- La maggior parte dei casi d’uso
- I casi di test e le procedure di test

Architettare vs integrare

Approccio tradizionale
- Analisi dei Requisiti
- Design
- Implementazione

Il processo è sequenziale

Approccio Moderno
- Analisi del mercato
- Riuso di componenti OTS
- Adattamento e Integrazione

Il processo è concorrente

Elementi della progettazione

- Procedure
- Funzioni
- Coroutine
- Processi
- Moduli
- Oggetti
- Componenti
- Agenti

} Questi diversi meccanismi linguistici vengono presi come elementi atomici della progettazione architetturale

Livelli di astrazione

- Strutture dati e algoritmo
- Classe (strutture dati e molti algoritmi)
- Package/Moduli (gruppi di classi correlate, magari interagenti via design pattern)
- Moduli/Sottosistemi (moduli interagenti contenenti ciascuno molte classi; solo le interfacce pubbliche interagiscono con altri moduli/sottosistemi)
- Sistemi (sistemi interagenti con altri sistemi - hardware, software ed esseri umani)

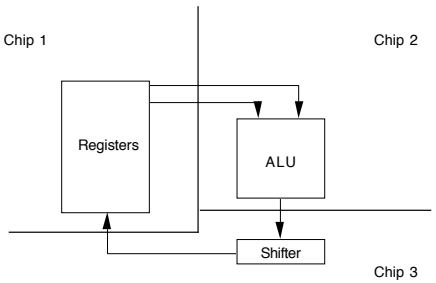
La **progettazione architetturale** riguarda gli ultimi due livelli

Obiettivi di progettazione

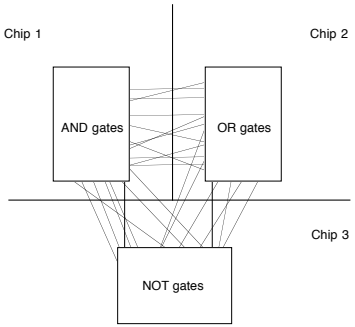
- Semplicità
  - Spesso si coniuga con eleganza
  - Il codice è più semplice da capire e correggere
  - Riduce gli errori
- Affidabilità e robustezza
  - Gestire input inattesi
  - Superare gli errori senza crash
  - Non far danni
- Efficacia
  - Risolvere i problemi giusti
  - Efficienza
  - Integrazione e compatibilità con altro software

Progetto semplice o complesso?

CPU con 3 chip (classi)



CPU con 3 chips (classi)



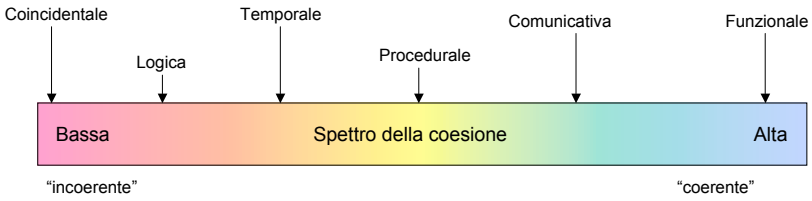
Progetto semplice o complesso?

- I due progetti sono funzionalmente equivalenti, ma quello di destra è
  - Difficile da capire
  - Difficile da correggere
  - Difficile da estendere o migliorare
  - Difficile da riusare.
  - Costoso da manutenere
- Il progetto di sinistra è migliore:
  - Massimizza le relazioni intraclasse (**coesione**)
  - Minimizza le relazioni interclasse (**accoppiamento**)

Coesione dei moduli

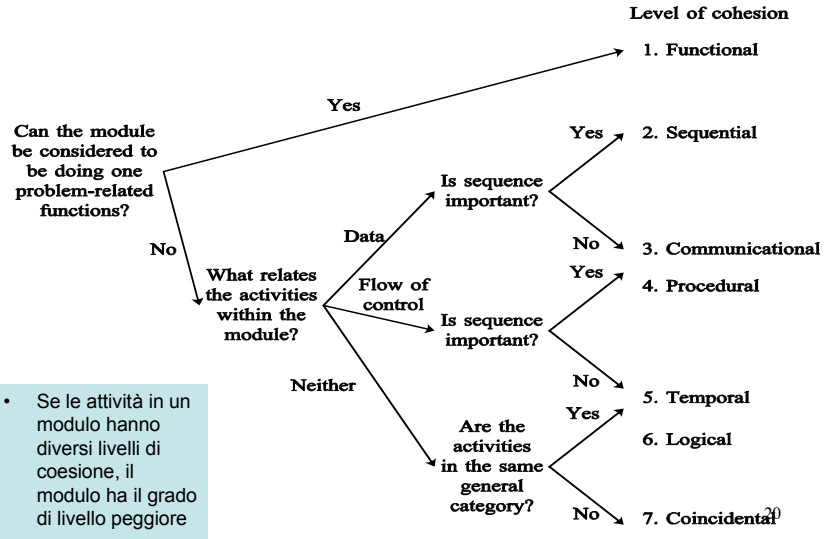
- Misura della coerenza funzionale di un modulo
- Ogni modulo dovrebbe avere **coesione alta**
- Un modulo ha coerenza funzionale se fa una cosa sola - e la fa bene
- Le classi componenti di grossi moduli dovrebbero essere funzionalmente correlate

Tipi di coesione



Coincidentale: Molteplici azioni e componenti completamente scorrelati
Logica: serie di azioni o componenti correlate (e.g. libreria di funzioni di IO)
Temporale: serie di azioni o componenti simultanee (e.g. moduli di inizializzazione)
Procedurale: serie di azioni che condividono sequenze di passi
Comunicativa: coeasione procedurale sugli stessi dati
Funzionale: una sola azione o funzione

Decidere il livello di coesione



- Se le attività in un modulo hanno diversi livelli di coesione, il modulo ha il grado di livello peggiore

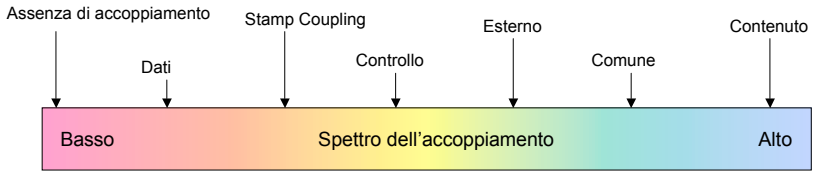
Accoppiamento dei moduli

- **Accoppiamento**: Misura del grado di indipendenza di un insieme di moduli
- I moduli di un sistema dovrebbero riportare un **accoppiamento basso (o lasco)**
- Questo si ottiene quando ci sono solo poche dipendenze tra i moduli, tutte funzionalmente rilevanti e necessarie
- Il software ad alto accoppiamento è fatto male (difficile da capire, mantenere, correggere, ecc…)

L'accoppiamento tra i moduli si riduce:

- eliminando relazioni non necessarie
- riducendo il numero delle relazioni necessarie
- rilassando le relazioni necessarie.

Tipi di accoppiamento



Misura dell'indipendenza funzionale di un insieme di moduli

Contenuto: un modulo che referenzia il contenuto di un altro modulo
Comune: due moduli hanno accesso agli stessi dati globali
Esterno: dati comuni tra due moduli con accesso strutturato
Controllo: un modulo passa un elemento di controllo ad un altro modulo
Stamp: un modulo fornisce parte di una struttura dati per un altro modulo
Dati: un modulo produce una struttura dati per un altro modulo che la consuma

Gerarchia

- La gerarchia dei moduli di un sistema sw riduce le dipendenze vincolando la topologia delle relazioni ra i moduli
- La struttura gerarchica dei moduli forma la base di progetto
  - Decompone il dominio del problema
  - Facilita lo sviluppo parallelo
  - Isola le ramificazioni di modifica e versionamento
  - Permette la prototipazione rapida

Relazioni gerarchiche tra moduli

- `X usa Y` (dipendenza funzionale, es. *call*)
- `X è_composto_da` {Y,Z,W}
  (scomposizione architetturale: solo le foglie sono "vero" codice)
- `X is_a Y` (ereditarietà)
- `X has_a Y` (contenimento)

Errori progettuali più comuni

- Progettazione “depth first”
- Raffinamento diretto della specifica dei requisiti
- Non considerare possibili modifiche
- Progettazione iperdettagliata

OO: Storia breve

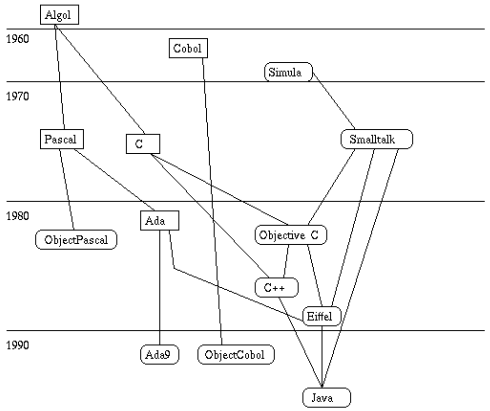
- 1967. Simula 67 - linguaggio per simulazione
- 1980. Primi linguaggi basati su oggetti. Smalltalk e C++
- 1984. MacOs è basato su Object Pascal
- 1987. Next è interamente basato su Objective-C
- 1990. Primi metodi di analisi e progettazione basati su oggetti: Coad, Wirfs-Brock, Booch, Rumbaugh, Jacobson
- 1995. Java: gli oggetti vanno in rete!
- 1997: UML 1.1 diventa uno standard OMG

Principali linguaggi OO

- Simula (anni ’60)
- Smalltalk (fine anni ’70)
- C++ (inizio anni ’80)
- Objective C (fine anni ’80)
- Eiffel (fine anni ’80)
- Visual Basic (inizio anni ’90)
- Delphi (Object Pascal, TurboPascal, 1995)
- Java (1995)
- C# (2000)

Evoluzione dei linguaggi OO

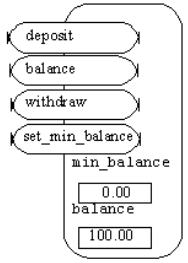
- Dahl …(1960s)…SIMULA
- Dijkstra…(1970s)…Abstraction
- Parnas…(1970s)…Information Hiding



Classi e oggetti

- I linguaggi ad oggetti includono sempre un tipo di modulo chiamato *classe*, che è uno schema dichiarativo che definisce *tipi di oggetti*
- La dichiarazione di classe *incapsula* la definizione dei campi e dei *metodi* degli oggetti creabili come *istanza* della classe

classe



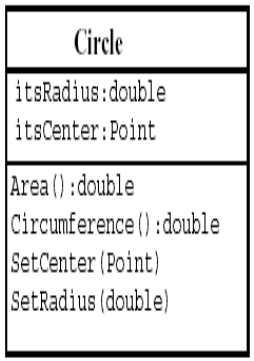
oggetto

Una classe in Java

```
class Main
{
  public static void main(String args[])
  {
    Account paolo = new Account();
    Account figaro = new Account();
    double obtained;
    System.out.println( "Saldo di Paolo = " +
    paolo.account_balance() );
    paolo.deposit(100.00);
    System.out.println( "Saldo di Paolo = " +
    mike.account_balance() );
    obtained = paolo.withdraw(20.00);
    System.out.println( "Paolo ha ritirato : " +
obtained);
    System.out.println( " Saldo di Paolo = " +
    paolo.account_balance() );
    figaro.deposit(50.00);
    System.out.println( " Saldo di Figaro = " +
    figaro.account_balance() );
  }
}
```


L'icona di classe in UML

- Definisce
  - Uno stato persistente
  - Un comportamento
- La classe ha
  - Nome
  - Attributi
  - Operazioni
- Ha una rappresentazione grafica in forma di un rettangolo diviso in tre parti



Terminologia

- **Classe**. Nome collettivo (**dichiarazione di tipo**) di tutti gli oggetti che hanno gli stessi metodi e variabili di istanza.
- **Oggetto**: Entità strutturata (codice, dati) e con stato la cui struttura e stato è **invisibile** all'esterno dell'oggetto.
- **Stato**: Lo stato di un oggetto si accede e manipola mediante messaggi che invocano metodi
- **Variabili di istanza:** Variabili contenute nell'oggetto che rappresentano il suo stato interno
- **Messaggio** Richiesta ad un oggetto di invocazione di uno dei suoi metodi
- **Metodo** Azione che accede o manipola lo stato interno del'oggetto (di solito le variabili di istanza). L'implementazione di tale azione è nascosta al cliente che spedisce messaggi all'oggetto

Metodi di analisi OO

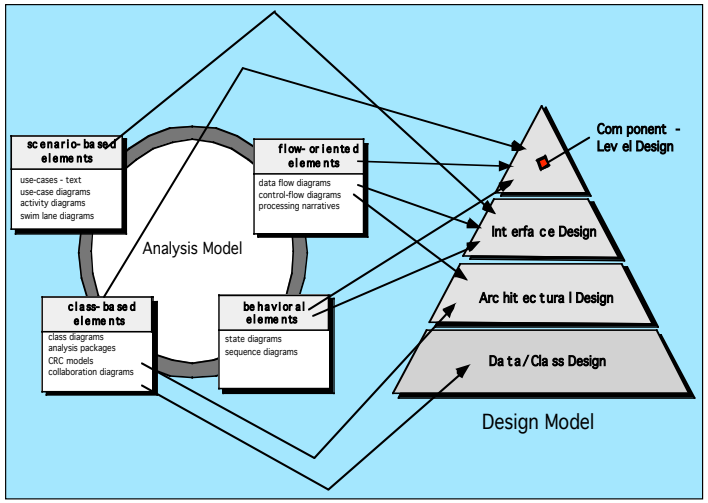
- Analisi OO: inizio di un processo di sviluppo OO
- Metodi di analisi OO:
  - Booch: processo evolutivo basato un “modello” astratto a oggetti
  - Rumbaugh: OMT (Object Modeling Technique) enfasi su modelli dinamici e funzionali
  - Jacobson: OOSE (Object Oriented Software Engineering) enfasi sui casi d’uso
  - Coad and Yourdon: enfasi sui livelli della modellazione
  - Wirfs-Brock: analisi e progetto sono un continuum
- Simili, con differenze noiose
- Booch, Rumbaugh e Jacobson si allearono per creare UML ed il RUP

Il processo OO

I processi OOA hanno tutti la seguente struttura:

1. Elicitazione dei requisiti
2. Identificazione di scenari e casi d’uso
3. “Estrazione” delle classi candidate
4. Identificazione di attributi e metodi
5. Definizione di una gerarchia di classi
6. Costruzione di un modello a oggetti e relazioni
7. Costruzione di un modello di comportamento degli oggetti
8. Revisione dell’analisi rispetto ai casi d’uso
9. Iterare se necessario

Dall’Analisi al Progetto (Pressman)



Il linguaggio di modellazione

- Un **linguaggio di modellazione** permette di specificare, visualizzare e documentare un processo di sviluppo OO
- I **modelli** sono strumenti di comunicazione tra cliente e sviluppatori
- I linguaggi di modellazione più usati sono anche *standardizzati*

Unified Modelling Language

- UML è un sistema di notazioni principalmente grafiche (con sintassi, semantica e pragmatica predefinite) per la modellazione OO
- UML non è un processo, né è proprietario
- Combina le notazioni di Booch, Rumbaugh e Jacobson
- E' uno standard OMG (Object Management Group)
- E' definito mediante un metamodello
- Include:
  - Viste (mostrano diverse facce del sistema: utente, strutturale, operazionale, ecc., anche in relazione al processo di sviluppo)
  - Diagrammi (grafi che descrivono i contenuti di una vista)
  - Elementi di modellazione (costrutti usati nei diagrammi)

Analisi = Processo + Modelli

Processo	**Modello prodotto**
1. Elicitazione dei requisiti d'utente e identificazione dei casi d'uso	Diagrammi e scenari dei casi d'uso
2. Estrazione delle classi candidate, identificazione degli attributi e dei metodi, definizione della gerarchia delle classi	Schede Classe- Responsabilità Collaboratore (CRC)
3. Costruzione di un modello a oggetti e relazioni	Diagramma delle classi
4. Costruzione di un modello operazionale degli oggetti	Diagramma delle interazioni

Casi d'uso

- **Vista del sistema** che mostra il suo comportamento come apparirà agli utenti esterni
- Partiziona le funzionalità in **transazioni** ('casi d'uso') significative per gli utenti ('attori').
- Consiste di **scenari** - interazioni tipiche tra un utente ed un sistema informatico
- Proprietà:
  - Mostra funzioni visibili all'utente
  - Raggiunge obiettivi specifici pr l'utente
  - Non rappresenta l'ordine o il numero di volte che una funzione viene eseguita
- Si ottiene dialogando con l'utente e il cliente
- Si usa per costruire i modelli strutturali e operazionali e per costruire casi di test

Che cos'è la progettazione OO?

Orientato agli oggetti:

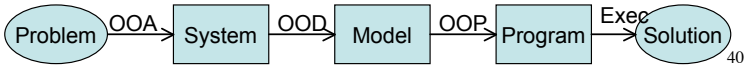
- Decomposizione di un sistema mediante astrazioni di oggetto
- Diversa dalla decomposizione funzionale/procedurale

OO Design, Analysis e Programming:

OOA: Esaminare e decomporre il problema… a*nalysis*

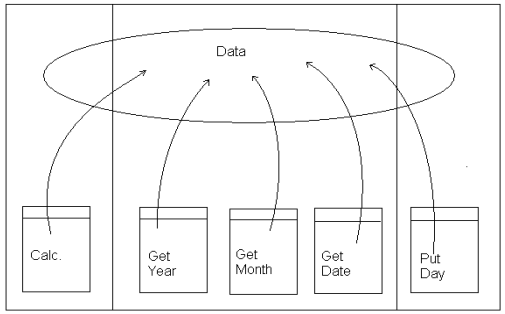
OOD: Costruire un modello…*design*

OOP: realizzare il modello…*programming*



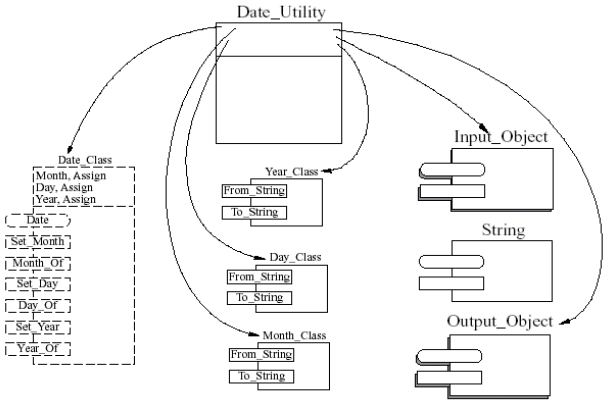
Approccio funzionale (non OO)

- I componenti sono blocchi funzionali
- Astrazione e incapsulamento assenti o poveri
- Non modella il dominio del problema
- Sequenziale (thread singolo)



Approccio OO

- I componenti sono classi e oggetti
- Astrazione e incapsulamento sono i meccanismi principali
- Il sistema finale rispecchia il dominio del problema
- Concorrenza (thread multipli)



Il paradigma OO secondo G.Booch

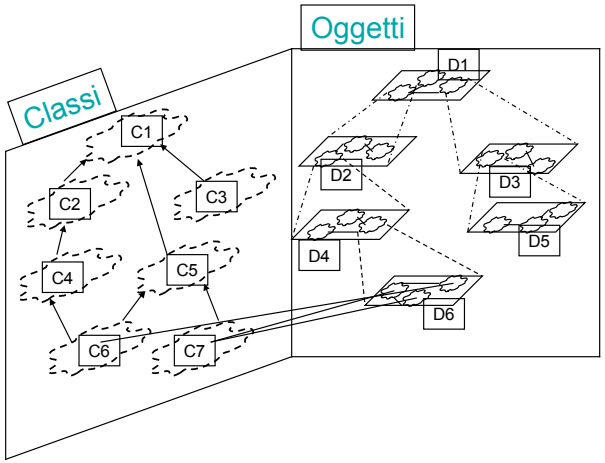
- Definizione e classificazione di nozioni base
- Modelli e notazioni
- Pragmatica
- Base di UML e UP (con i “Three Amigos” di Rational)



The three amigos:

Grady Booch, Jim Rumbaugh, Ivar Jacobson

Forma canonica di un sistema OO



Principali elementi del paradigma OO secondo Booch

- **Astrazione** Il progettista crea le classi e gli oggetti
- **Incapsulamento** Information hiding
- **Modularità** Decomposizione in moduli ad accoppiamento lasco
- **Gerarchia** ordinamento di astrazioni mediante ereditarietà

Elementi minori del paradigma di Booch

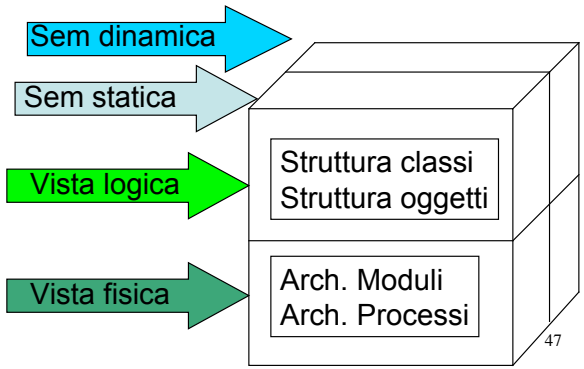
Tipaggio Vincoli su classi e oggetti

Concorrenza Thread multipli

Persistenza Oggetti che sopravvivono nello spazio e nel tempo al programma che li crea

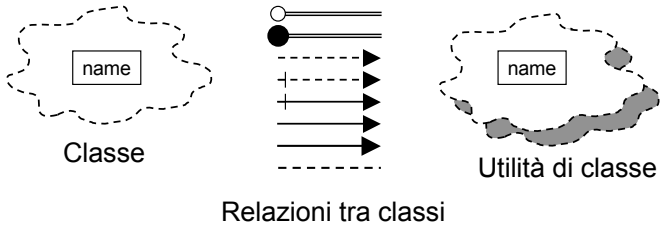
Il "cubo" di Booch

- Vista **logica** e vista **fisica**
- Semantica **statica** e semantica **dinamica**

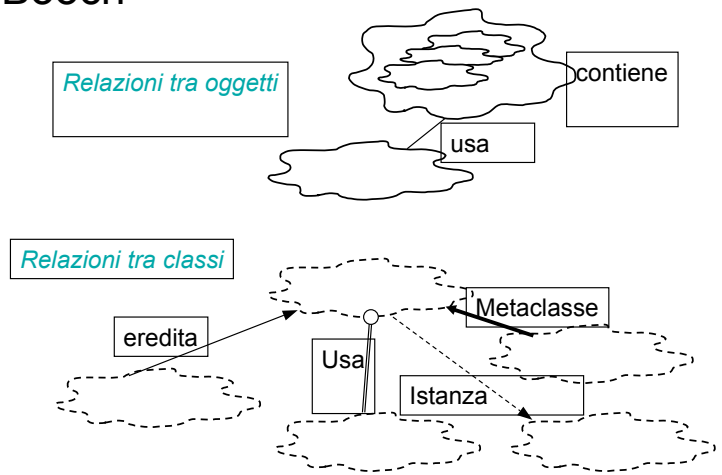


Diagrammi di classe secondo Booch

Classi, relazioni tra classi, utilità di classe



Relazioni tra classi nel paradigma di Booch



Associazioni

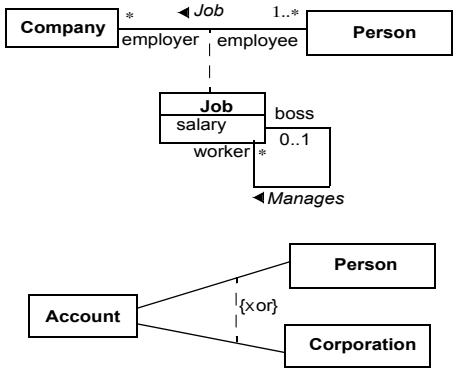


Fig. 3-40, *UML Notation Guide*

Riferimento: Tutorial OMG su UML di Cris Kobryn

Composizione

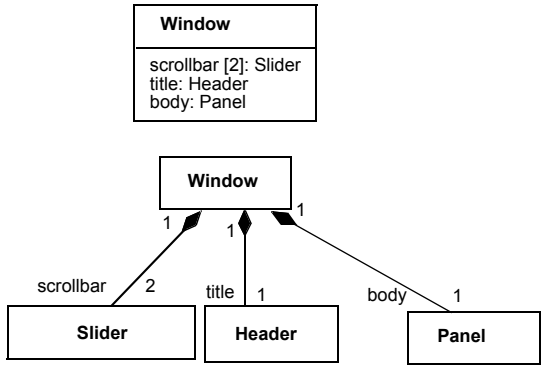


Fig. 3-45, *UML Notation Guide*

Generalizzazione

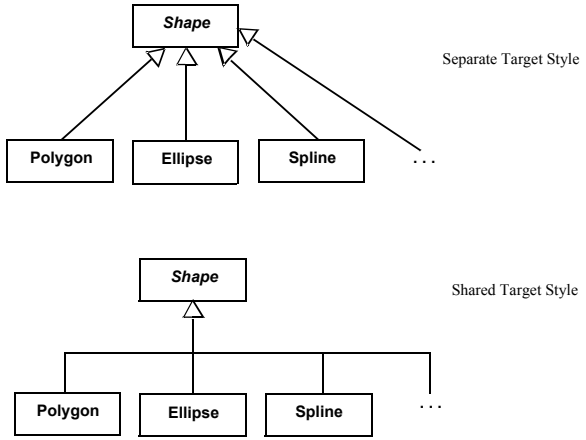


Fig. 3-47, *UML Notation Guide*

Dipendenze

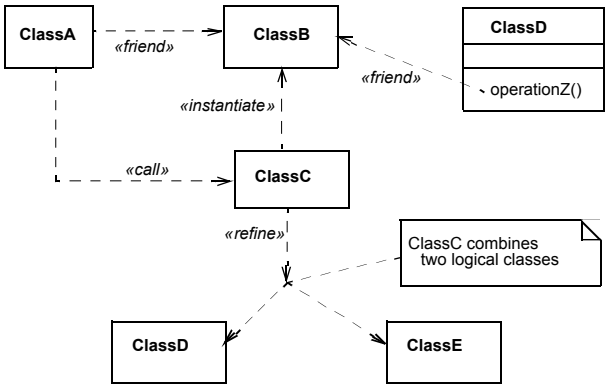
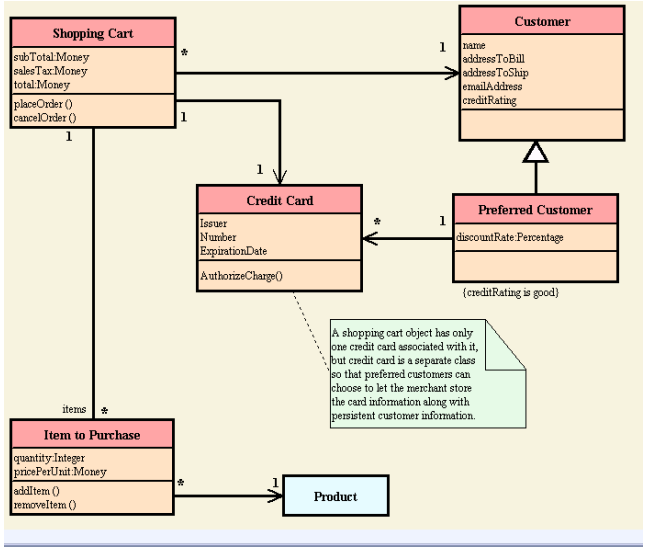


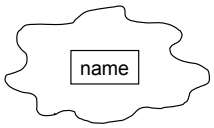
Fig. 3-50, *UML Notation Guide*

Diagramma di classi in UML

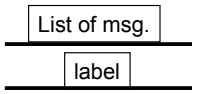


Diagrammi degli oggetti

- Oggetti e relazioni tra oggetti:



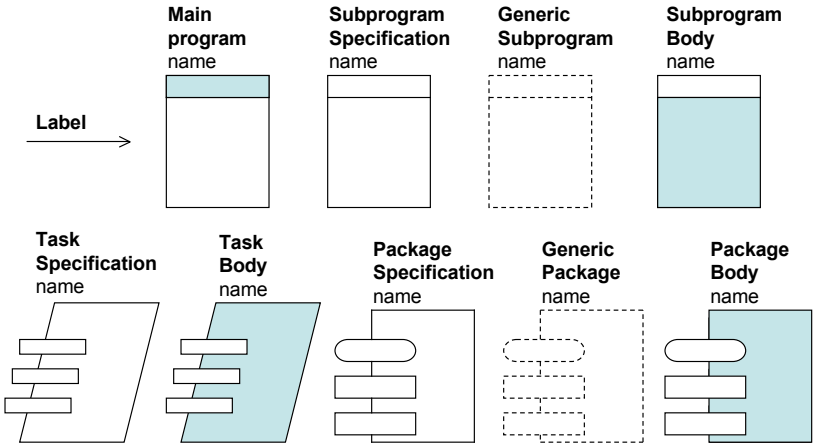
oggetto



relazioni tra oggetti

- Visibilità e sincronizzazione
- Object diagram templates

Diagrammi dei moduli (Gradygrammi)



Diagrammi dei processi

- Connessioni tra processore e dispositivo

- Process diagram templates

Aspetti linguistici degli oggetti

- Identità
- Classificazione
- Ereditarietà
- Polimorfismo
- Information hiding (incapsulamento)

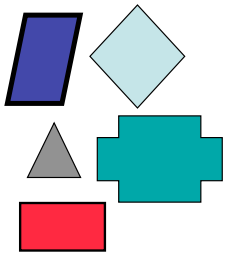
Identità

- Ogni oggetto denota i dati che gestisce
- Ogni oggetto ha una identità
  ⇨ possono esistere due oggetti con dati identici e diversa identità
- L'identità degli oggetti si può realizzare con un id unico o con una chiave (logical pointer)
- Gli oggetti vengono acceduti mediante l'id unico
  ⇨ è possibile mescolare i meccanismi

Classificazione

- Raccolta di oggetti simili: **classe**
  - stessi attributi (variabili d'istanza)
  - stesse operazioni (servizi/messaggi)
- Classe: **astrazione** di attributi rilevanti
  - le classi si riferiscono al dominio dell'applicazione
  - Una classe descrive un insieme (infinite) di oggetti = **istanze** della classe

Esempio: classe poligono



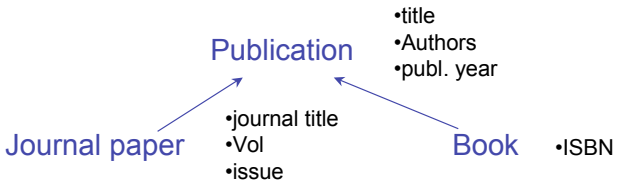
- class polygon
  - attributi
    - set of points
    - line color
    - fill color
  - operazioni
    - draw
    - delete
    - move

Classe vs tipo

- **Tipo di dato** (in senso tradizionale):
  - dominio + operazioni (e.g. Integer)
  - Si usa per dichiarare variabili
- **Classe**:
  - Costrutto linguistico, orientato all'implementazione
  - Realizza uno o più tipi OO
- **Tipo OO (interfaccia)**:
  - Protocollo compreso da un oggetto
  - Insieme di messaggi (operazioni)
- **Tipo di dato astratto:**
  - Specifica domini (sorte), operazioni e assiomi
  - Nozione più astratta rispetto al tipo OO

Ereditarietà

- Uso condiviso di attributi e codice in classi diversi



- Le sottoclassi ereditano tutte le proprietà della superclasse

Usi dell'ereditarietà

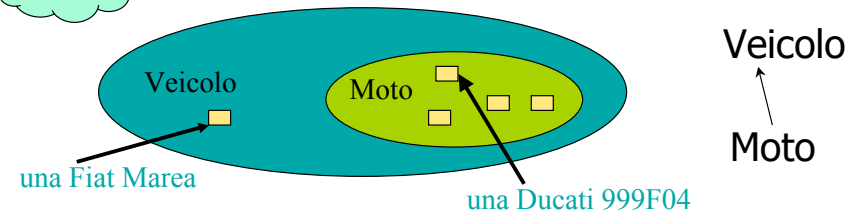
- **Classificare** le entità di un dominio
- Evitare **ridondanze**
  - Il codice identico si scrive una volta sola
- **Diminuizione** della dimensione del codice
- **Riuso** del codice

Generalizzazione

- Classe: definizione implicita di un insieme di oggetti
  - unVeicolo ∈ Veicolo = Insieme di *tutti* i veicoli
- Generalizzazione: relazione di contenimento insiemistico
  - Moto ⊆ Veicolo

classificazione

generalizzazione



Polimorfismo

"*calcola area* di un cerchio"

"*calcola area* di un quadrato"

Le operazioni con lo stesso significato dovrebbero avere lo stesso nome

- **overloading** = stesso nome di funzione, implementazione diversa
- Nei linguaggi procedurali (es.C): solo per i tipi base (e.g.: int * int, real * real)

Information hiding

- **Nascondere** i dettagli inessenziali
- Accesso solo mediante le operazioni **predefinite**

black box

Responsibility-Driven Design

- **Tecnica in cui le responsabilità di un oggetto guidano il suo design**
- **Si concentra sul ruolo di un oggetto e su come il suo comportamento influenza gli altri oggetti**
- **Se si comincia dalle responsabilità di un oggetto poi è più semplice**
  - Creare la sua interfaccia pubblica (come il mondo esterno accede le sue funzioni)
  - Progettare il suo funzionamento interno
  - Tener conto degli eventi da esso riconosciuti

- Prospettive di modellazione (Rebecca Wirfs-Brock)

Esempio: Prospettive di un cavallo

- ***Vista strutturale:*** **ha un corpo, una coda, quattro zampe** ***(componenti)***
- ***Vista comportamentale:*** **cammina, mangia, emette versi** ***(attività)***
- ***Vista delle responsabilità:*** **trasporta persone o merci, corre negli ippodromi** ***(scopo entro un sistema)***

Progettazione guidata dalle responsabilità

Le responsabilità di una classe si classificano come:

- Conoscere
  - A quali domande deve rispondere?
- Fare
  - Quali operazioni deve eseguire?
- Applicare
  - Quali regole deve applicare e/o imporre?

Esempio

- Una festa
  - Quali sono gli oggetti?
    - Padroni di casa
    - Ospiti
    - Invito
      - Indirizzo
        - » Della festa
        - » Dell'ospite
      - Data
      - Tema
    - Cibo e bevande
    - Musica

Person (superclass)

Esempio

  - Quali sono i metodi?
    - comprare
    - cucinare
    - invitare
      - occorre la lista degli (oggetti) invitati
    - pulire la casa
    - RSVP
    - andare alla festa
  - Quali sono le responsabilità dei vari oggetti?

Approccio

- Definire il problema
- Creare gli scenari d'uso
  - Mediante interviste
  - Concentrarsi sulel operazioni principali
- Usare le schede CRC
  - Oggetti
    - Iniziare con quelli più importanti
  - Responsabilità
    - Le cose principali che fanno gli oggetti più importanti
  - Relazioni con altri oggetti

Schede CRC

- Classe, Responsabilità, Collaborazione
  - responsabilità: compiti da eseguire
  - collaborazioni: altri oggetti che cooperano con questo

Class name:	Superclasse:	Sottoclassi:
Responsabilità:	Collaborazioni:	

CRC Card

Class name: **Padrone**	Superclass: **Persona**	Subclasses:
Responsibilities:	Collaborations:	
Invita	**Invito, Persona, Lista**	
Compra	**Denaro, Negozio, Cibo, …**	
Pulisce_casa	**Spugna, Straccio, …**	

CRC Card

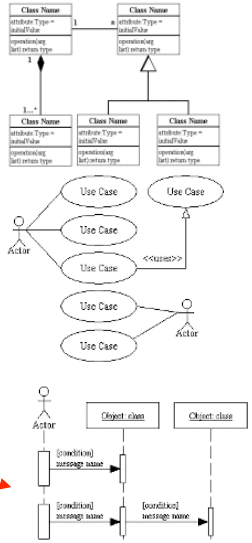
Class name: **Ospite**	Superclass: **Persona**	Subclasses:
Responsibilities:	Collaborations:	
RSVP	**Telefono, Padrone**	

Da CRC a UML

- Le schede CRC definiscono le classi principali e le loro interazioni
  - Strumento di brainstorming
  - Se ne scrivono tante, se ne buttano tante
- UML
  - Per raffinare il progetto
  - Per descrivere il progetto ad altri

UML: Unified Modeling Language

- Notazione per progetto OO
- Associata ad un metodo
- Parecchi tipi di diagrammi
  - Diagrammi di classe
    - Struttura statica
  - Diagrammi Casi d'Uso
    - Funzionalità
  - Diagrammi di sequenza
    - Vista temporale dello scambio di messaggi tra le classi



I tre passi del progetto OO

- Tre passi iterabili
  1. Modellazione dei casi d'uso
     - Determinare come si ottengono i risultati principali
     - Orientata alle azioni
     - Tecnica: linguaggio naturale
  2. Modellazione delle classi (e degli oggetti)
     - Determinare le classi con attributi e relazioni
     - Orientata ai dati
     - Tecnica: schede CRC e poi diagrammi UML di classe
  3. Modellazione dinamica
     - Determinare le azioni eseguite da o su ciascuna classe
     - Orientata alle azioni
     - Tecnica: diagrammi di sequenza e interazione

Il design del sw nel SWEBOK

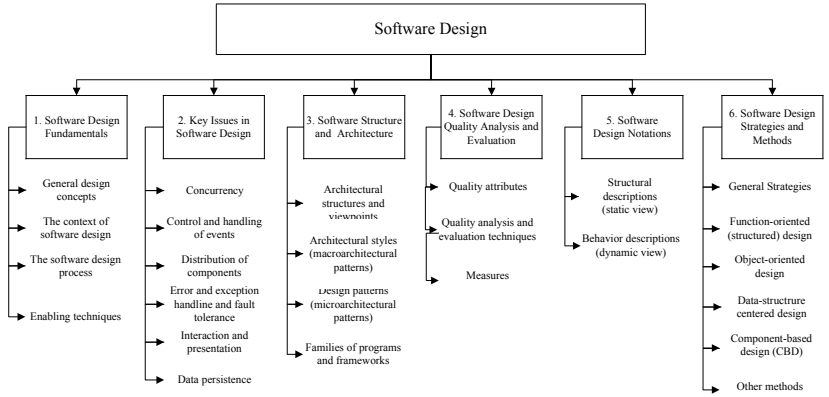
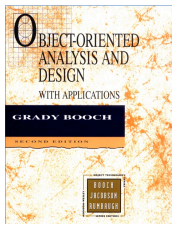
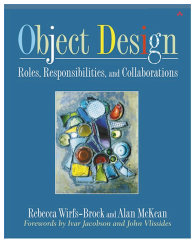


Figure 1 Breakdown of topics for the Software Design KA

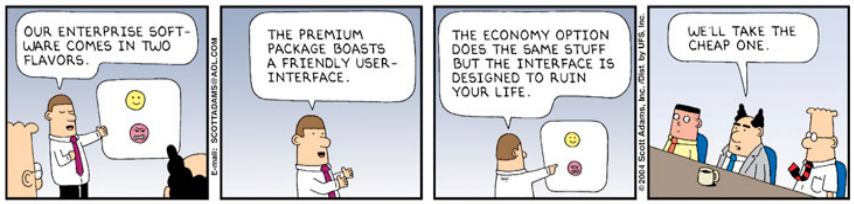
Riferimenti

- Capitolo 3 del SWEBOK "Software design"
- IEEE Recommended Practice for Software Design Descriptions (IEEE 1016-1998)
- Booch, *Object oriented analysis and design with applications*, AW 1993
- Wirfs-Brock and Mckean, *Object Design: Roles, Responsibilities and Collaborations*, AW 2002





Domande?


OUR ENTERPRISE SOFTWARE COMES IN TWO FLAVORS.
THE PREMIUM PACKAGE BOASTS A FRIENDLY USER-INTERFACE.
THE ECONOMY OPTION DOES THE SAME STUFF BUT THE INTERFACE IS DESIGNED TO RUIN YOUR LIFE.
WE'LL TAKE THE CHEAP ONE.
E-mail: SCOTTADAMS@AOL.COM
©2004 Scott Adams, Inc./Dist. by UFS, Inc.