

***Object Oriented  
Analysis and Design  
OOA/OOD  
Metodologie***

***Prof. Paolo Nesi***

**Dipartimento di Sistemi e Informatica  
Universita' di Firenze  
Via S. Marta 3, 50139, Firenze  
tel: 055-4796523, fax: 055-4796363  
email: nesi@dsi.unifi.it, nesi@ingfi1.ing.unifi.it  
www: <http://www.dsi.unifi.it/~nesi>**

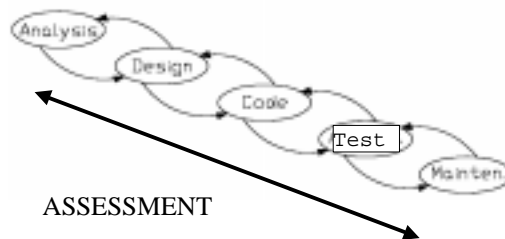
**Perché una Metodologia**

- Una metodologia e' un insieme di regole e linee guida che permettono, almeno sulla carta, di ottenere un processo di sviluppo:
  - Ripetibile
  - Indipendente dalle capacita' di chi la utilizza
  - Documentata
  - Portabile su vari linguaggi
  - Riusable in vari contesti
  - etc..
- Questa deve tenere conto degli aspetti strutturali, comportamentali e funzionali del sistema.
- Deve anche essere mirata o confezionabile per ogni particolare contesto di sviluppo: sistemi gestionali, sistemi embedded, sistemi distribuiti, architetture parallele, sistemi avionici, sistemi medicali, sistemi di controllo per macchine utensili, etc.....

## Perché una Metodologia

- Le metodologie sono spesso supportate da modelli grafici di rappresentazione delle entità e delle loro relazioni.
- Tali modelli grafici sono spesso generabili attraverso strumenti di tipo CASE (Computer Aid Software Engineering) su una GUI
- Tali strumenti offrono spesso la possibilità' di generare codice in svariati linguaggi di programmazione.
- Strumenti raffinati integrano
  - il CASE, la metodologie e il linguaggio con
- gli strumenti di:
  - test: debug, capture and playback,
  - class browser per la navigazione fra le classi,
  - configuration management,
  - per il lavoro cooperativo,
  - versioning,
  - process and product assessment,
  - documentation
  - etc.

## Ciclo di Vita a Cascata

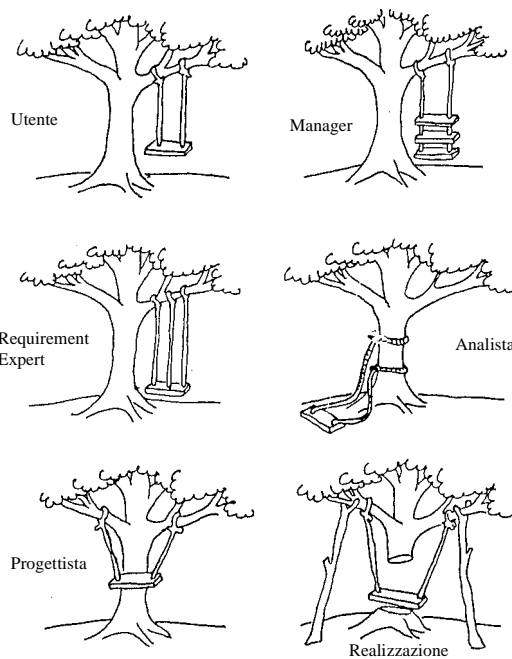


- Collezionamento dei requisiti
- Analisi dei Requisiti (analisi)
- progetto astratto (decomposizione/composizione alto livello)
- progetto dettagliato (comunicazione fra moduli, realizzazione interna)
- codifica
- collaudo (verifiche di consistenza e completezza, simulazione)
- Manutenzione correttiva

## Ciclo di Vita a Cascata

- Le fasi sono troppo distinte
- Fasi diverse sono tipicamente eseguite da persone diverse
- La comunicazione fra fasi diverse e pertanto fra team diversi avviene tramite documentazione scritta utilizzando modelli diversi: e.g., Z per i requisiti, DFD, per il design; C come linguaggio, etc..
- Il passaggio da un modello all'altro costa fatica.
- Chi usa un modello per descrivere il sistema e' diverso da chi lo interpreta, una sorta di telefono senza fili.
- Vi sono iterazioni ma spesso sono costose visto che il tempo viene speso a spiegarsi a vicenda come erano state pensate e come sono state interpretate le descrizioni riportate nei documenti.

## L'altalena



## La produzione del software

- Lo sviluppo di un sistema secondo i principi della "buona" ingegneria del software, comporta fasi di analisi e di progettazione prima della codifica.
- Nella realtà, ciò viene effettuato molto raramente.
- La ragione non sta solo nella costante pressione cui sono sottoposti i programmatori, ma anche nella grande separazione che c'è tra analisi, progetto e programmazione.
- Con l'avvento delle tecniche ad oggetti, questa separazione viene a cadere, e le attività divengono un processo continuo che si basa sullo stesso modello, sempre più raffinato e particolareggiato.
- Nelle metodologie OO il costo in ore uomo per lo sviluppo è maggiormente distribuito nelle varie fasi produttive (analisi, progetto e codifica), spostando il baricentro verso l'analisi dalla codifica.
- Molte metodologie OO fanno riferimento a modelli evolutivi che portano a lavorare per prototipi.

*Paolo Nesi, 1998/2000*

7

## Approcci per Prototipi

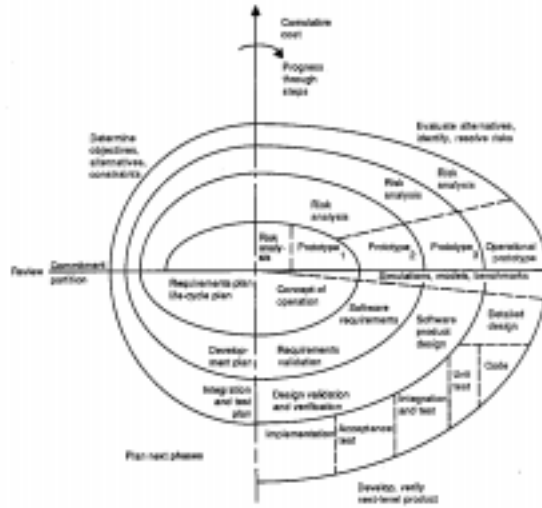
- Definizione dell'architettura generale
- Implementazione di un primo prototipo con funzionalità minime
- Incremento graduale delle funzionalità del sistema e della documentazione
- processo iterativo di Analisi, Progetto, Codifica e Test.
  
- La documentazione cresce di conseguenza
- E' possibile tenere sotto controllo la produzione in base alle funzionalità implementate
- Il collaudo viene effettuato in modo incrementale
- Il sistema può essere visionato dal committente per avere un feedback anche durante le prime fasi del suo sviluppo.

**Degenerazione verso modelli evolutivi meno controllabili, come il modello a fontana.**

*Paolo Nesi, 1998/2000*

8

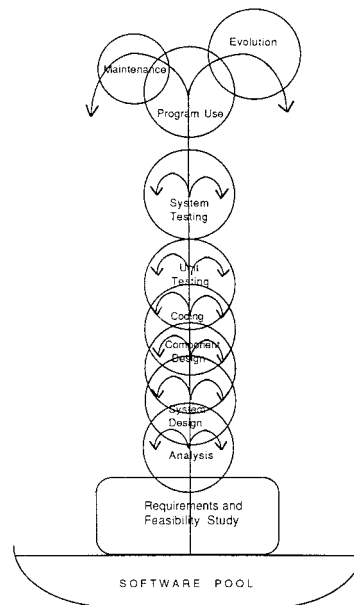
## Ciclo di Vita a Spirale (Boehm 1986)



Paolo Nesi, 1998/2000

9

## Ciclo di Vita a Fontana



Henderson-Sellers

Real-World System

Paolo Nesi, 1998/2000

10

## Concetti di Base, Pro

- Nello sviluppo OO si ha sempre a che fare con oggetti durante tutto il ciclo di vita. Il modello diventa uniforme.
- Uso di modelli evolutivi, non si ha un transizione brusca fra una fase e la successiva
- la complessità passa dalla fase di progetto e codifica a quella di analisi: rivalutazione del personale
- introduzione di tecniche di riuso formalizzate: la classe e' un modello formale per l'interfaccia di riuso di moduli.
- La manutenzione e' semplificata poiché il collaudo viene distribuito lungo tutto il ciclo di vita e scelte architetturali sbagliate vengono facilmente identificate nei primi cicli, nei primi prototipi.
- Incremento di produttività anche del 100 % rispetto ai metodi tradizionali

## Concetti di Base, Contro 1/2

- Il tipo di gestione manageriale cambia totalmente
- Le modalità di gestione delle persone cambiano totalmente
- Le metodologie di analisi, progetto cambiano totalmente.
- Il modo di fare test, la documentazione e l'assessment cambiano radicalmente
- I Costi di riuso sono difficilmente quantificabili
- Le librerie di terzi sono un terreno molle: qualità, OO-ness, permanenza sul mercato, etc.
- I linguaggi sono ancora poco stabili (C++ abbastanza, Java e' in evoluzione continua)

## Concetti di Base, Contro 2/2

- I programmatori devono essere riconvertiti.
- Un analista ha bisogno di 3 anni per essere formato partendo come programmatore OO
- Un manager di almeno 5 anni di esperienza partendo come programmatore OO.
- I vecchi manager dovrebbero reimparare a programmare ad oggetti non solo sulla carta ma facendo il programmatore 7 ore/giorno per almeno 6 mesi.
- Decomposizione facile del problema e quindi anche possibilità di delegare parte delle realizzazione a terzi

## Procedurale vs Object oriented

- Procedurale (decomposizione funzionale)

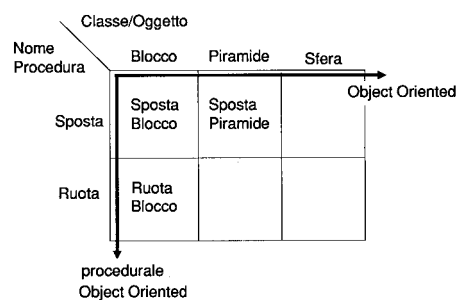
**DATI + ALGORITMI = PROGRAMMI**

ci si chiede *come deve essere eseguita* una certa operazione

- Object-Oriented (decomposizione in oggetti)

**OGGETTI + OGGETTI = OGGETTI**

ci si chiede *CHI, o meglio cosa, deve eseguire* una certa operazione



## L'esempio della Ericsson

- Nel 1968, Jacobson ideò la prima metodica di analisi e progetto OO, per lo sviluppo del software delle centrali telefoniche.
- L'architettura OO risultante è ancora usata in Ericsson, e si prevede lo sarà nei prossimi 50 anni.
- Ovviamente, il software è modificato ed adattato in continuazione, ma l'architettura base resta la stessa.
- Il software così prodotto è di elevata qualità, ben documentato e facilmente modificabile a costi ridotti.
- Come conseguenza di *questo fondamentale vantaggio competitivo*, la Ericsson è diventata uno dei più grandi produttori di centrali telefoniche.
- Alcune società concorrenti, come ITT e Philips (settore telefonico), che nel 1968 erano ben più grandi, si sono ritirate o si sono ridimensionate.

## Metodologie Orientate agli Oggetti

- A livello metodologico, negli ultimi anni l'OOP è stato adottato da molti sviluppatori.
- I meccanismi dell'OOP rendono facile la decomposizione del sistema in classi ed oggetti, identificando facilmente quali sono le attività che devono essere eseguite in parallelo

Molti di questi approcci sono reinterpretazioni ed estensioni di tecniche tradizionali basate sull'analisi strutturata -- e.g., OMT, Coad and Yourdon, Martin and Odell.....

altri sono approcci completamente ad oggetti Booch, and Wirsf-Brock et al., etc..

alcuni sono basati su di uno specifico linguaggio di programmazione ad oggetti....



## **Analisi e progetto ad oggetti (OOA e OOD)**

- A partire dalla fine degli anni '80, sono stati sviluppati dei metodi formali di OOA e OOD
- Ciò ha generato una "guerra dei metodi", con oltre 80 metodi proposti, di cui solo una dozzina sono effettivamente utilizzati.
- Un metodo di analisi c/o progetto comprende:
  - una notazione per esprimere il modello del sistema
  - un elenco di documenti testuali associati
  - dei passi sistematici per effettuare l'analisi (il *processo*).
- Il processo di produzione del software può variare a seconda del sistema da progettare ed è maggiormente legato alle preferenze della singola organizzazione.

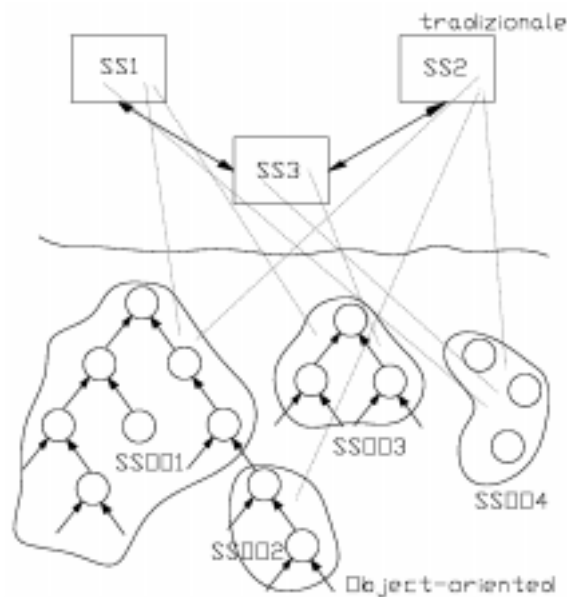
## **Analisi e Sviluppo ad Oggetti**

- Le classi vengono identificate sulla base delle tipologie di oggetti reali mentre gli oggetti identificando le loro istanze (e.g., un asse di moto da controllare, l'asse di 35mm posto a 45 gradi, sono la classe ed una sua istanza).
- Le classi vengono organizzate secondo gerarchie:
  - strutturale - legami IS-PART-OF, IS-REFERRED-BY
  - concettuale - relazioni di specializ./generaliz. fra classi al fine di favorire il polimorfismo, etc.
- Nella fase di progetto vengono definite nel dettaglio le relazioni e le operazioni, i metodi di ogni classe.
- Segue la fase di codifica in cui le classi vengono realizzate per mezzo di un linguaggio di programmazione, segue poi il test, etc.
- Il test come verifica e validazione della specifica può essere presente in varie fasi dello sviluppo in funzione del tipo di approccio che viene utilizzato, sono ovviamente da preferire i metodi per i quali la verifica e validazione può essere eseguita fin dalle fasi di analisi e durante tutto il ciclo di sviluppo.
- Idem per Documentazione, assessment.....

## Decomposizione Strutturale vs Object Oriented

- Metodi tradizionali si basano su una decomposizione strutturale sia per l'identificazione dei moduli che per la definizione dei sottosistemi
- Nel modello ad oggetti l'identificazione dei sottosistemi dipende dalle relazioni fra le classi.
- Il concetto di sottosistema e' più vicino a quello di package:
  - Alberi e sotto-alberi con classi semplici connesse
  - Librerie separabili: GUI
  - Funzionalità trasversali: print, load, save, ... metodi polimorfici
  - Classi sciolte di servizio
  - Drivers,

## Da Tradizionale a OO



## Ciclo di Vita OO

- Il ciclo di vita di un sistema orientato agli oggetti non è così semplice come è stato descritto poiché le fasi di analisi, progetto e test non sono separabili.
- Parti diverse dello stesso sistema si possono trovare nello stesso istante in fasi diverse del ciclo di vita. Per esempio, in un sistema sotto sviluppo possono convivere classi appena analizzate con classi di cui è già stato fatto il progetto e la codifica.
- Questo accade poiché l'OOP consente uno sviluppo sia bottom-up (partendo dalla classe che modella il sistema) che top-down (partendo dalle classi elementari per poi definire quelle che ne fanno uso) rendendo semplice il riuso, e spingendo verso un approccio di tipo prototyping.

## Macro e Micro Ciclo

Identificazione di un macro-ciclo e di un micro-ciclo di vita.

### PER ESEMPIO

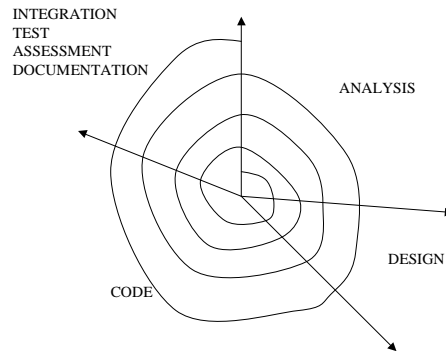
- modello a spirale per il macro
- modello a fontana per il micro

### MACRO CICLO

- Analisi, progetto, codifica, verifica, val rischi, etc. ogni 4 mesi
- Funzionalità definite per ogni prototipo
- Ultimi Cicli di
  - Integrazione
  - Ottimizzazione
  - Dimostrazione e Validazione

## Il Micro Ciclo a Spirale

- Analisi, progetto, Test, etc. sequenziali senza possibilità di modifiche
- Attività di tipo mensile o quindicinale
- Nessuna sovrapposizione fra seguenti spire della spirale

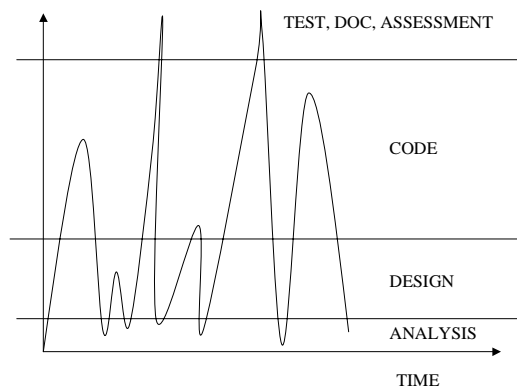


*Paolo Nesi, 1998/2000*

23

## Il Micro Ciclo a Fontana

- Analisi, progetto, Test, etc. non sequenziali e non di durata fissa
- Attività di tipo mensile o quindicinale
- Sovrapposizioni non controllate



*Paolo Nesi, 1998/2000*

24

## Analisi dei Requisiti

- Processo attraverso il quale si astraggono le necessita' del committente
- L'analista che conosce il dominio del problema si trova in questo caso in enorme vantaggio rispetto a chi ne e' estraneo.
- Durante questa fase di definiscono anche dei casi d'uso cioe' si descrive come il sistema si deve comportare nei riguardi di altri sistemi o utenti nei casi piu' interessanti, complessi e importanti.
- Realizzazione di un documento che in certi casi puo' fare parte anche del contratto fra il committente ed il fornitore/realizzatore del sistema software. Pertanto spesso e' scritto in modo abbastanza formale ma certe volte anche in modo troppo legale.
- Vi sono anche specifici linguaggi e modelli per la definizione dei requisiti e la loro formalizzazione in modo che sia possibile verificarne la consistenza e completezza.

## Analisi Object Oriented

- Processo attraverso il quale si *modella il dominio del problema* e si astraggono quindi le necessita' per la sua soluzione.
- Il dominio e' la porzione del mondo che contiene il problema.
- Si cerca di fare un'analisi del dominio generalizzando quelli che sono i requisiti del sistema e quindi la tradizionale analisi dei requisiti. Questo permette di avere un'analisi del problema piu' astratta che porta a un *modello piu' riusabile*.
- Questo significa che l'analista dovrebbe conoscere il dominio del problema o che deve avvalersi di esperti.
  
- Durante questa fase si analizzano i casi d'uso al fine di identificare le entita' piu' importanti del sistema e le loro relazioni.

## Identificazione Classi: Linee Guida

- Secondo il Paradigma ad oggetti:
  - le categorie del mondo reale di traducono in *classi*
  - le istanze del mondo reale in *oggetti*
- Si può anche aggiungere che:
  - le categorie generiche si traducono in *template*
  - i modelli di interazione comuni in *interfacce*

categorie del mondo reale (classi): ruoli, cose, unita' organizzative, ubicazioni, dispositivi

## Analisi Object Oriented

- L'analisi si concentra su *cosa deve fare* il sistema e non sul come lo deve fare.
- Questa fase deve produrre un documento che descrive il problema in termini di classi ed oggetti.
- Lo scopo dell'analisi è di identificare le classi e porle in relazione. Per esempio utilizzando la notazione vista durante la presentazione del paradigma ad oggetti.
- Definizione dell'albero delle classi con le varie relazioni dell'OOP.
- Il documento di analisi può essere utilizzato anche per la verifica delle funzionalità con il committente poiché è tipicamente intuitivo.

## Linee Guida per l'OOA

- Identificare le entità/classi principali del sistema reale, cercando di definire entità aventi bassa coesione con le altre *entità principali*: concetti ortogonali, ....
- definire la struttura delle entità principali e le loro relazioni di specializzazione, aggregazione, associazione.
- Albero delle classi con relazioni: IS\_A, IS\_PART\_OF e IS\_REFERRED\_BY.
- Definire *i servizi* delle classi principali.
- Se possibile ridurre la coesione al solo passaggio di messaggi fra oggetti.
- definire le *entità di servizio*, classi più piccole, e le relazioni fra loro e le classi importanti.
- Inserire tali classi nella gerarchia di classi.

## Analisi Object Oriented

### Le classi principali sono il motore del sistema (KEY CLASS):

- il Gestore della seriale
- Il garage
- Il gestore degli oggetti disegnabili
- Le liste in genere
- Un iteratore il Database

Spesso sono classi che danno luogo a thread separati, godono comunque di una certa autonomia di stato.

Non sono sicuramente le finestre del vostro sistema.

Tenete fuori dall'analisi l'identificazione delle finestre dell'interfaccia utente.

## Analisi Object Oriented

### Sono classi base o di servizio:

- I numeri complessi
- il vettore
- il buffer
- una struttura dati che va nel database.
- ...

Classi elementari che non hanno autonomia ma sono semplicemente utilizzate da molte altre con legami di tipo IS-PART-OF

Sono classi delicate proprio perché utilizzate da molte altre classi.

Sono tipicamente classi passive, non Thread.

## Identificazione dei Servizi

- Durante l'analisi si devono identificare le relazioni e quindi i servizi delle classi. Anche questi hanno un riscontro con il mondo reale:

- Specializzazione, generalizzazione -> IS-A
- Estensione, ruoli, comportamenti -> IS-A
- Composizione/decomposizione -> IS-PART-OF
- Aggregazione -> IS-PART-OF
- Associazione, liste, strutture dinamiche -> IRB
- Uso -> chiamata a metodo, IS-PART-OF
- Generazione -> AN-INSTANCE-OF
- Modello generico -> Template, Abstract Template
- Interfaccia -> interfaccia
- ...
- ..

La ricerca dei servizi e delle relazioni aiuta anche a identificare eventuali altre classi ed oggetti.



## Specializzazione vs Specificazione

Nel turbine del processo di specializzazione si deve mettere in chiaro che non esiste un solo punto di vista nella identificazione e nella definizione delle classi.

Un domanda fondamentale:

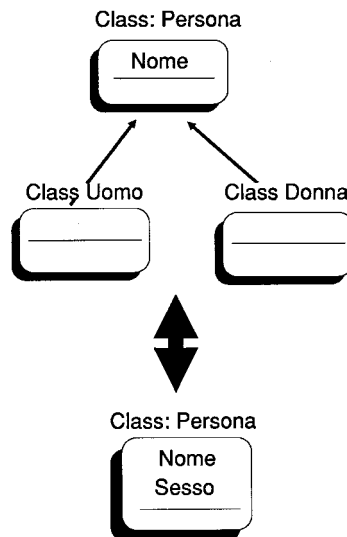
*Quando si deve smettere di specializzare e si deve invece iniziare a specificare le differenze attraverso le variabili definite come attributi?*

Una risposta a questa domanda potrebbe essere:

*Quando la presenza di un parametro fra gli attributi fa s che i metodi debbano essere parametrizzati al punto da rendere difficoltosa la loro stesura.*

i.e., quando e' più conveniente tenere separati due entità diverse: per comportamento o per struttura (due tipi diversi).

## Specializzazione vs Specificazione



## Processo di Generalizzazione

- Processo attraverso il quale si identificano membri comuni di classi in relazione gerarchica e si portano verso la radice.
- Può portare a definire classi astratte
- Classi astratte sono tipicamente non utilizzare per produrre oggetti
- Classi astratte pure sono più sicure perché non istanziabili.

Viene tipicamente effettuato nei cicli interni del micro o macro ciclo per ottimizzare il modello ad oggetti ottenuto.

- Porta ad una riduzione del codice ed a un incremento del riuso e della manutenibilita'
- I costi devono essere previsti
- ..

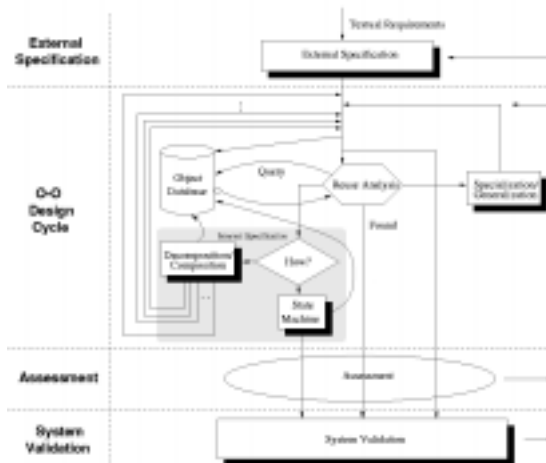
## Progetto Object Oriented

- Fase in cui i dettagli che sono rimasti in sospeso nella fase di analisi vengono definiti. In modo da definire in un certo dettaglio **COME deve essere codificato** il sistema.
- definizione di dettaglio delle modalità di comunicazione fra oggetti
- definizione delle relazioni indirette fra oggetti tramite i metodi
- definizione delle relazioni generiche fra oggetti tramite Object Diagrams, diagrammi fra Istanze
- Identificazione delle interfacce da utilizzare

## Codifica Object Oriented

- Realizzare le entità di servizio dalla più generica alla più specialistica, testarle in modo incrementale.
- Realizzare le entità principali dalla più generica alla più specialistica.
- Utilizzare un file per la Definizione ed uno per la Realizzazione
- Tenere sotto controllo le dimensioni delle classi: numeri attributi, numero di metodi, loro complessità
- Non sfruttare al massimo il linguaggio ma mantenersi conformi ad un modello ideale OO

## Object Oriented Development Life-Cycle



## Riuso Object Oriented

- Classe: Riuso degli aspetti statici e dinamici del sistema
- Implicito nella specializzazione e nel polimorfismo
- Specifico per le classi e i sottosistemi
  
- A livello di analisi, progetto e codifica.
- Di librerie/package purché questi siano composti da gerarchie piccole o molto complete. Gestione dei conflitti fra librerie di classi.
  
- Modellazione e caratterizzazione del software da riutilizzare. La definizione della classe può essere utilizzata per verificare la sua usabilità in un altro contesto.
- Istituzione del repository dal quale fare il riuso: sito web per la pubblicazione del riusabile
- Istituzione del manuale di riuso
- Gestione del materiale riusabile: versioni, specializzazione, costo di manutenzione, documentazione
  
- Definizione di un team per il riuso

## Consigli per una definizione di sistemi che siano il più OO Possibile

### Privilegiare:

- l'ereditarietà (Inheritance);
- il polimorfismo (Inheritance + IS-REFERRED-BY).
- La definizione di template

### Dosare l'uso di:

- ereditarietà multipla (Multiple Inheritance);
- ereditarietà parziale (Partial Inheritance) (per levarsi dai guai e per ridurre il numero di attributi ridondanti);
- condivisione dei dati fra oggetti (Data/Object Sharing);
- condivisione del codice o struttura fra classi che non sono in relazione gerarchica (Structural Sharing); (e.g., C++, Friend).
- Interfacce e delle loro specializzazioni

### Non definire mai:

- attributi pubblici nelle classi poiché questi risulterebbero essere attributi globali del sistema.

## ESERCIZIO

- Definire un sistema che mantenga traccia di tutte le auto che entrano ed escono dai cancelli: primario e secondario della fabbrica, mantenendo traccia del giorno, dell'orario di ingresso e di quello di uscita, nonché del nome della/e persona/e a bordo e della persona che gli/li ospita se sono visitatori. Ad ogni cancello e' posta una persona.

## Sommario della Metodologia

- Iniziare raccogliendo dei casi d'uso del sistema. Di solito bastano 6-15 storie sul funzionamento del sistema in risposta a richieste esterne, incluse le motivazioni della storia stessa.
- Iniziare l'OOA a partire dai Casi di uso o scenari, individuando classi potenziali, discutendole ed elencandone le responsabilità. Non considerate la GUI o il DB, ma solo il modello del sistema.
- Suddividere il sistema in sottosistemi ed iniziate l'approfondimento di uno di essi o delegarne lo studio a persone diverse.
- Definite le classi, le loro responsabilità, le dipendenze da altre classi, l'ereditarietà. Definire gli attributi: aggregazioni e associazioni, Mettete tutto in vari diagrammi di classi .
- Passate a definire la struttura di interazione, le operazioni complesse. Utilizzare diagrammi ad oggetti e diagrammi di stato. Provate a descrivere i casi di uso o scenari di partenza con diagrammi ad oggetti.
- Scrivete un prototipo del sottosistema, dapprima senza GUI e accesso al DB. Ciò inevitabilmente porta a modifiche nel diagramma delle classi.

- Testate il sistema realizzato rispetto ai casi di uso identificati.
- Effettuate una valutazione del sistema

Passate a raffinare il prototipo, aggiungendo funzionalità, secondo l'approccio incrementale-iterativo.

Fate un piano per mettere in ordine di precedenza le funzionalità.

- Per ogni ciclo completo come il precedente vi possono essere dei micro-cicli che durano 2-3 settimane mentre il ciclo completo può durare anche alcuni mesi.
- Cercate di lasciare separata la gestione dell'interfaccia utente e della parte Database.
- Questi devono essere dei sottosistemi separati e non devono influenzare la rappresentazione ad oggetti del dominio del problema.



## ***UNIFIED MODELING LANGUAGE***

**Paolo Nesi**

**Dipartimento di Sistemi e Informatica  
Universita' di Firenze  
Via S. Marta 3, 50139, Firenze  
tel: 055-4796523, fax: 055-4796363  
email: [nesi@dsi.unifi.it](mailto:nesi@dsi.unifi.it), [nesi@ingfi1.ing.unifi.it](mailto:nesi@ingfi1.ing.unifi.it)  
www: <http://www.dsi.unifi.it/~nesi>**

## UML è una cosa seria!

- E' opera di **tre** dei massimi esperti di OOA e OOD, Booch, Jacobson e Rumbaugh (i *tres amigos*), e nasce come *standard aperto*, sintesi di molti metodi.
- E' **stato accettato** da molti altri esperti del settore, tra cui Coad, Yourdon e Odell, e da tutte le **grandi compagnie di informatica**.
- E' uno standard dall'OMG (Object Management Group), creato nel 1989 con 440 aziende: Microsoft, Digital, HP, NCR, SUN, OSF, etc.
- Esistono ottimi strumenti CASE per UML, e praticamente tutti i produttori di tali strumenti ne hanno annunciato il supporto. Da un modello UML è possibile generare automaticamente lo "scheletro" del codice di un sistema (le strutture dati complete ed i prototipi delle funzioni).
- **Microsoft ha adottato UML** come linguaggio standard per la sua "Repository" di componenti.
- L'uso di UML e di strumenti CASE permette *effettivamente* di sviluppare sistemi meglio documentati, di migliore qualità e più facili da mantenere nel tempo.

## La situazione prima di UML

- metodi più diffusi con le percentuali di mercato, stimate alla fine del 1996 da R.A. Johnson, University of Arkansas.

Metodo	Utilizzo
OMT di Rumbaugh et al.	20%
Booch	15%
SWaer & Mellor	15%
Martin-Odell	7%
Coad & Yourdon	5%
Wirfs-Brock, CRC	5%
Jacobson, OORE	5%
Fusion	3%
HOOD	3%

- La frammentazione del mercato era di ostacolo agli investimenti ed allo sviluppo dell'Object Oriented Technology!

## La nascita di UML

- Alla fine del 1994, James Rumbaugh e Grady Booch si sono associati per definire un Linguaggio di Modellazione Unificato (UML) di terza generazione, nato dalla fusione dei due metodi omonimi.
- Nel 1995, anche Jacobson si è unito formando il gruppo dei "tre amigos", operante presso la Rational Software Corp. di Santa Clara, California.
- La quota di mercato dei metodi dei "tre amigos", pari al 40%, la pubblicità data al nuovo metodo, l'apertura a suggerimenti ed interventi esterni e l'oggettiva necessità di uno standard hanno dato una spinta decisiva.
- UML si è evoluto nelle versioni 0.8 (1995), 0.9 (1996), 1.0 (gennaio 1997) e 1.1 (settembre 1997).
- Le ultime due versioni sono state presentate all'OMG come standard dalle seguenti società:

Digital Equipment	Ericsson	Hewlett-Packard
i-Logix	IBM	ICON Computing
Intellicorp	MCI Systemhouse	Microsoft
ObjecTime	Oracle	PLATINUM Technology
Rational Software	Sterling Software	Unisys

## Linguaggio di Modellazione Unificato

- Il linguaggio UML supporta le seguenti fasi dello sviluppo dei sistemi software:
  - Analisi dei requisiti tramite casi d'uso.
  - Analisi e progetto (OOA, OOD)
  - Modellazione dei componenti
  - Modellazione della struttura e della configurazione.
- Il modello OOA/OOD viene espresso tramite dei diagrammi grafici.
- Ogni entità del modello può comparire in uno o più diagrammi, che ne rappresentano una vista.
- Ad ogni entità si possono anche associare vari tipi di documentazione testuale.
- Nei vari diagrammi, tutti i concetti e le entità che presentano similitudini, sono espressi con la medesima notazione.



## I sette diagrammi di UML

### USO/REQUISITI

- *diagramma dei casi d'uso (use case diagram)*: elenca i casi d'uso del sistema e le loro relazioni.

### OOA/OOD

- *diagramma delle classi (class diagram)*: descrive la struttura dati degli oggetti del sistema e le loro relazioni. E' il diagramma più importante, dal quale si può generare il codice.

### COMPORAMENTO/OOD

- *diagramma sequenziale (sequence diagram)*: mostra le interazioni tra gli oggetti durante scenari di funzionamento del sistema, privilegiando la sequenzialità temporale.
- *diagramma di collaborazione (collaboration diagram)*: mostra le interazioni tra gli oggetti durante scenari di funzionamento del sistema, privilegiando la struttura del sistema stesso.
- *diagrammi di stato e di attività (state e activity diagram)*: usano la notazione di Harel e descrivono gli stati di un oggetto e le sequenze eventi-azioni-transizioni di una funzione.

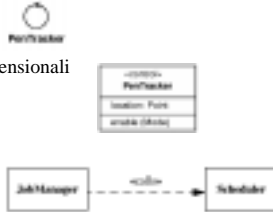
## I sette diagrammi di UML

### IMPLEMENTAZIONE/CODIFICA

- *diagramma dei componenti (component diagram)*: descrive l'architettura fisica del sistema.
- *diagramma di dispiegamento (deployment diagram)*: descrive la struttura del sistema hardware e l'allocazione dei vari moduli software.

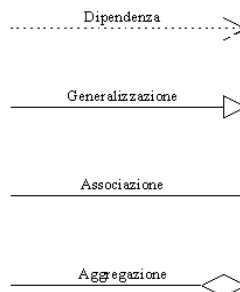
## Linguaggio UML: generalità

- diagrammi UML sono grafi contenenti nodi e connessioni
- Vi sono tre relazioni topologiche importanti:
  - la connessione (di linee a forme bidimensionali)
  - il contenimento
  - la prossimità (di due simboli)
- Di solito, le dimensioni non hanno rilevanza.
- In UML, vi sono quattro costrutti grafici:
  - le icone
  - i simboli bidimensionali
  - i collegamenti
  - le stringhe integrate (f: Function, from: Real, to: Real)
- Le icone hanno dimensioni e forma fisse, e non contengono nulla



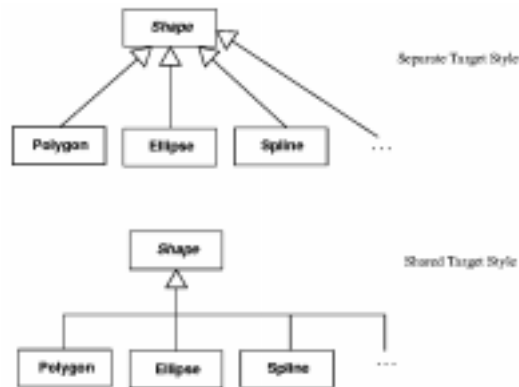
## Linguaggio UML: generalità

- I simboli bidimensionali hanno dimensioni variabili, possono essere espansi e contenere altre entità, possono essere suddivisi in compartimenti
- I collegamenti sono linee curve o spezzate che terminano con connessioni a due simboli. I collegamenti rappresentano diversi tipi di relazioni:



## Linguaggio UML: generalità

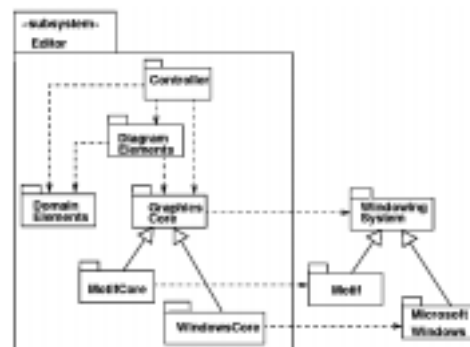
- Talora, più collegamenti si possono riunire in alberi, ma vanno sempre intesi come l'unione di più collegamenti semplici



- Le stringhe hanno molti usi e possono apparire in liste

## Packages

- è un gruppo di elementi del modello
- può contenere altri pacchetti
- L'intero modello si può considerare come un P. di alto livello, che contiene tutto
- è delimitato da un rettangolo grande con un rettangolino in alto a sinistra
- se è minimizzato, il rettangolo ne contiene solo il nome
- Se è esploso, il nome può essere contenuto nel rettangolino



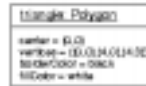
## Nomi, tipi e istanze

- Un nome è una stringa che identifica un elemento del modello entro una certa area di azione, scope
- Un nome completo (pathname) è una stringa che identifica completamente un elemento del modello

Nomi:           ContoBancario  
                  MatriceBanda  
                  indirizzo

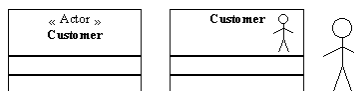
Nome completo:  
                  MathPack::Matrici::MatriceBanda.dimensione

- Un modello di solito descrive elementi generici (*tipi*) che specificano elementi particolari (*istanze*)
- Ad esempio: classi-oggetti, associazione-legame, parametro-valore, operazione-chiamata
- In UML, i simboli per le istanze sono identici a quelli per i tipi corrispondenti, con il nome dell'istanza sottolineato ed opzionalmente seguito da due punti e dal nome del tipo



## Stereotipi

- Uno stereotipo è una nuova classe di elementi del modello, predefinita o fattibile ad opera dell'utente
- La notazione per lo stereotipo è una stringa entro doppie parentesi angolari: <<control>>, <<Actor>>, posta sopra o vicino al nome dell'elemento.
- Attraverso gli Stereotipi, UML presenta la possibilità di essere esteso o adattato ad un particolare ambiente, progetto o utente.
- Gli stereotipi sono oggetti predefiniti la cui semantica può essere estesa dal progettista.

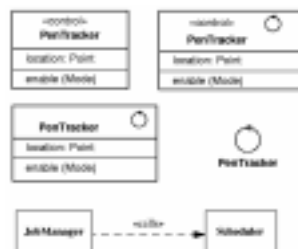


## Stereotipi

«interface»	class	
«control»	class	
«active object»	class	
«utility»	class	
«actor»	class	
«bind»	refinement relation	
«imports»	dependency relation	
«calls»	dependency relation	
«friend»	dependency relation	
«instantiates»	dependency relation	
«extends»	use case relation	
«uses»	use case relation	
«self»	association role	
«local»	association role	
«parameter»	association role	
«global»	association role	
«signal»	event	
«becomes»	node dependency relation	

## Icone ed Etichette

- Sono permesse anche icone, o la composizione di queste con le stringhe:



- Un'etichetta (label) è una stringa associata a un simbolo grafico, per contenimento o per vicinanza

## Proprietà

- Una proprietà di qualche elemento del modello è espressa da una stringa associata al simbolo grafico dell'elemento
- Le proprietà sono contenute entro parentesi graffe e separate da virgole
- Una proprietà è espressa come coppia

*parola chiave = valore*

ove *parola chiave* è il nome della proprietà e *valore* è una stringa che ne denota il valore

Esempio:

```
{author = "Joe Smith", deadline = 31-March-1997, status = analysis}
```

- Le espressioni di tipo sono denotate in UML da stringhe, senza vincoli imposti da UML

Esempi:

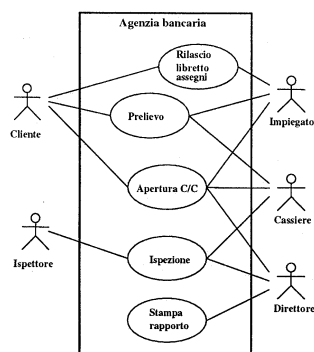
BankAccount

BankAccount \* (\*) (Person\*, int)

array [1..20] of reference to range (-1.0..1.0) of Real

## Diagramma dei casi d'uso

- Casi d'uso sono negli ovali mentre le icone con "l'omino" rappresentano gli attori.
- Si identificano i possibili "attori" del sistema (entità esterne al sistema che interagiscono con esso, fra cui, ad esempio, gli utenti), i possibili "scenari" di utilizzo dello stesso e le relazioni fra attori e scenari (in quali scenari sono coinvolti i diversi attori). Ogni scenario viene poi dettagliato mediante uno degli altri diagrammi della vista dinamica.



## Diagramma dei casi d'uso

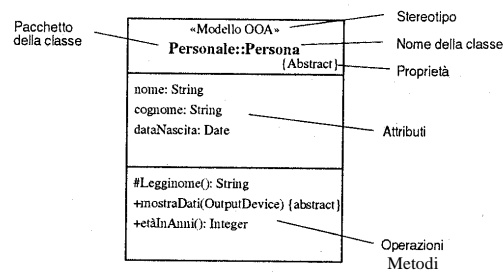
- Da un diagramma dei Casi di Uso è possibile risalire al diagramma che descrive le sue relazioni.
- Dal diagramma che descrive un sistema per la vendita per telefono ed il sistema di ordini ad esso collegato viene derivato un diagramma delle relazioni (Use Case Relationships) che metta in evidenza le relazioni tra i vari scenari e tra i vari attori e gli stessi.



61

## Diagramma delle classi

- E' il nucleo Fondamentale di tutto l'UML.
- Mostra la struttura statica del sistema: gli oggetti in esso contenuti e le loro relazioni reciproche.
- Una classe è la descrizione di uno o più oggetti con uguali attributi ed operazioni. Il simbolo di classe è un rettangolo diviso in tre regioni:



- Notazione per gli attributi della classe:

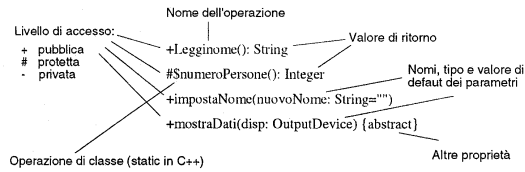
nomeAttributo: Tipo = valoreDefault

Paolo Nesi, 1998/2000

62

## Diagramma delle classi

- Notazione per le operazioni della classe:



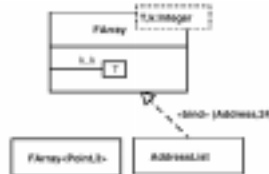
- Nel diagramma delle classi si mostrano solo gli attributi e le operazioni più significative.
- A classi, attributi ed operazioni, come ad ogni elemento di UML, si può associare anche della documentazione testuale.
- Le operazioni possono essere descritte in un PDL (process description language), e possono avere pre- e post-condizioni.
- In tutti i diagrammi possono apparire commenti, che vengono racchiusi in rettangoli con un angolo “ripiegato”

## Diagramma delle classi

- Con i Class Diagram è possibile specificare la vista statica di una classe. La classe può essere rappresentata a diversi livelli di dettaglio fino alla sua struttura interna.
- Esempio relativo a un sistema a finestre.



- E' possibile specificare classi parametriche ed il modo in cui altre classi sono derivate da esse una volta legati tali parametri a dei o tipi valori particolari. Questo meccanismo permette, per esempio, di definire una classe che rappresenta una lista di indirizzi come se fosse un array (classe definita a priori) di 24 elementi di tipo indirizzo.

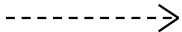
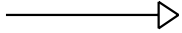
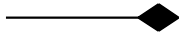
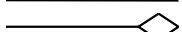
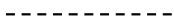




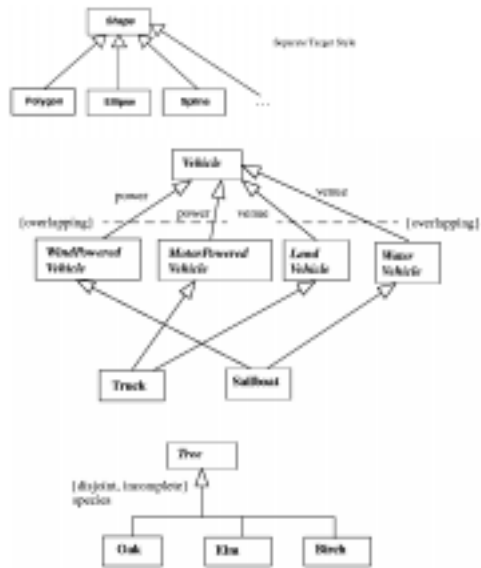
## Diagramma delle classi

- E' possibile specificare le relazioni che intercorrono tra più classi dando un nome ed un senso logico di percorrenza.
- Tali relazioni possono presentare una molteplicità ed un nome che richiama il ruolo delle due classi coinvolte nella relazione.
- È possibile inoltre utilizzare attributi per specificare proprietà aggiuntive di una relazione.
- E' possibile dichiarare delle relazioni generiche fra package che poi si concretizzano con relazioni reali solo nel dettaglio dei diagrammi delle classi.

## Le relazioni fra le classi

- Tra le classi vi possono essere quattro tipi fondamentali di relazione:
- Dipendenza 
- Ereditarietà 
- Aggregazione 
- Associazione 
- Realizzazione
- Vincoli 

## Specializzazione



Paolo Nesi, 1998/2000

67

## Dipendenza

- Dipendenza: relazione fra due packages. Significa solo che il Banking potrà utilizzare istanze del package Customer.



Paolo Nesi, 1998/2000

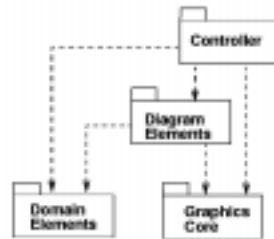
68

## Dipendenza

Various usage dependencies among classes



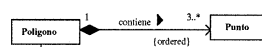
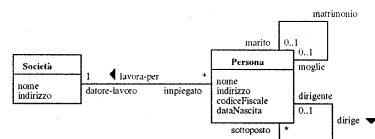
Dependencies among packages



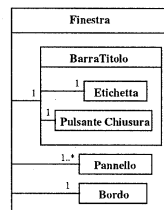
Paolo Nesi, 1998/2000

69

## Aggregazione e Dipendenza



Questo è un commento associato alla classe Poligono.

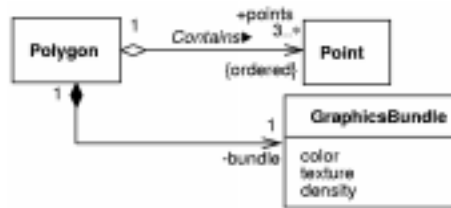


Paolo Nesi, 1998/2000

70

## Associazione e Aggregazione

- Si possono specificare relazioni di aggregazione tra classi specificando se si tratta di una composizione (le parti esistono in funzione del tutto e solo se questo esiste) o di una semplice associazione.
- Nel descrivere la vista statica di una classe per la rappresentazione dei poligoni si possono rappresentare questi ultimi come insiemi di punti, di cui si ammette l'esistenza anche se non vi è un poligono, e di altre caratteristiche grafiche (colore, perimetro..) che hanno senso solo se esiste un poligono a cui sono riferite.

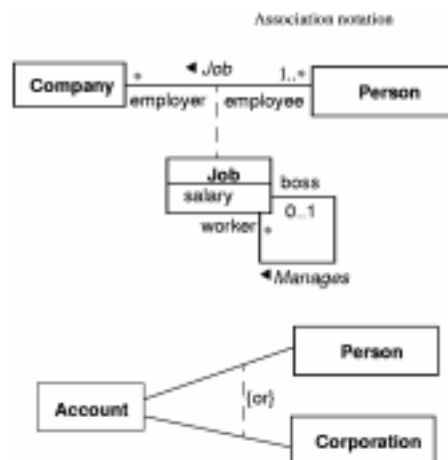


Paolo Nesi, 1998/2000

71

## Relazioni di Associazione

- PE: in una azienda possono, ad esempio, lavorare più impiegati (ognuno con 1 o più impieghi) questa relazione di lavoro è caratterizzata da un salario e dal fatto che un dirigente controlla più operai che a loro volta sono diretti da un unico responsabile.



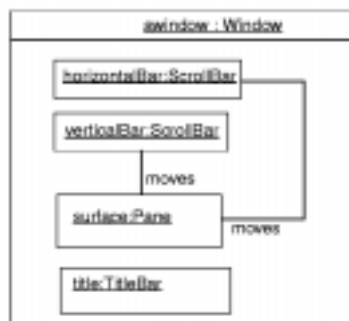
Paolo Nesi, 1998/2000

72

## Esempi di Realizzazione

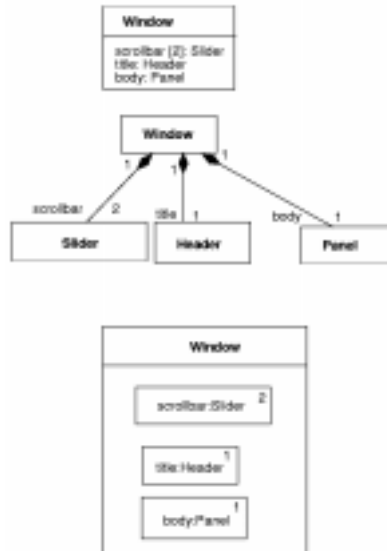
## Oggetti Composti

- Oggetto awindow della classe Window composto da uno scrollbar verticale e orizzontale, da un pane e da un titolo.
- La sottolineatura differenzia le istanze (gli oggetti) dalle classi.



## Oggetti Composti

- Diversi modi di rappresentare relazioni IS-PART-OF



Paolo Nesi, 1998/2000

75

## Object Diagrams

- Un object diagram rappresenta una particolare "configurazione" del sistema, un particolare insieme di oggetti e le rispettive relazioni.
- La sintassi degli Object Diagram ricalca quella dei Class Diagram, è possibile tramite questo tipo di diagrammi descrivere oggetti derivati tramite istanziazione dalle classi. Per esempio, dalla descrizione di una struttura dati ad oggetti per la rappresentazione di un punto specificata tramite un Class Diagram è possibile rappresentare in un Object Diagram una particolare istanza di tale classe (un oggetto) assegnando anche dei particolari valori ai suoi attributi.

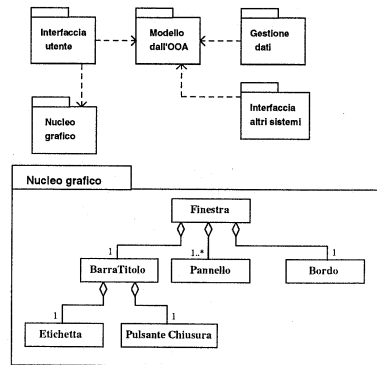


Paolo Nesi, 1998/2000

76

## I Pacchetti (Package)

- Sono sottoinsiemi del modello che raggruppano insieme classi in qualche modo collegate o affini.
- I pacchetti possono contenere altri pacchetti, e definiscono una portata di nome.



Paolo Nesi, 1998/2000

77

## Tipi

- Un tipo è la descrizione di oggetti avente:
  - Stato astratto (attributi astratti)
  - Specifica delle operazioni concrete
  - Nessuna implementazione delle operazioni
- Un tipo è un'entità più astratta della classe, che anche se astratta può avere attributi ed implementare operazioni (simile al Template del C++).
- Una classe può implementare uno o più tipi, una o più interfacce.
- Concetti di interfaccia e tipo sono stati distinti
- Una classe che implementa un tipo appartiene allo stereotipo (opzionale) "implementation class".
- Un oggetto può avere più tipi, ma solo una classe.
- Il simbolo di un tipo è uguale a quello di una classe, con lo stereotipo "type". Attributi ed operazioni sono astratti, e quindi in corsivo.

Paolo Nesi, 1998/2000

78

## Interfacce

- Un'interfaccia specifica un insieme di operazioni pubbliche di una classe, di un componente, di un pacchetto o di altre entità. è usata in Java.
- Un'interfaccia non ha alcuna specifica di struttura interna (attributi, stato o associazioni).
- Essa equivale a una classe astratta, senza attributi né associazioni e con solo operazioni astratte.
- Un'interfaccia si denota col simbolo di classe e lo stereotipo <<interface>>. Il compartimento degli attributi è sempre vuoto, e può mancare (deve in Java).
- Una classe, un componente o un package può sopportare un'interfaccia. Ciò si denota con un cerchio collegato all'entità che la supporta, col nome dell'interfaccia vicino.
- Un'altra classe che usa l'interfaccia può essere collegata ad essa da una freccia di dipendenza, eventualmente con lo stereotipo <<uses>>.
- In alternativa, si può usare la relazione <<realizza>>

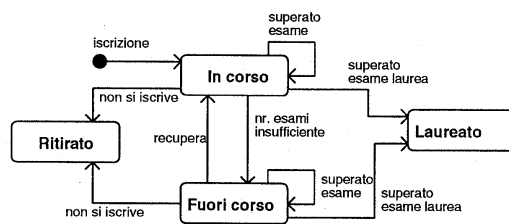
## Tipi e interfacce: notazione





## Diagrammi di Stato e Attivita'

- I diagrammi di stato descrivono l'evoluzione temporale di un oggetto in risposta alle interazioni con altri oggetti.
- Ogni classe può avere associato un diagramma di stato. Il diagramma riporta l'evoluzione dello stato della classe a prescindere dal fatto che questo è implementato con metodi diversi.
- UML adotta quasi completamente la notazione di Harel, che può esprimere sottostati, stati composti, parallelismo, stati storici, gestione eventi, operazioni, creazione e distruzione di oggetti, marcamenti temporali, ecc.

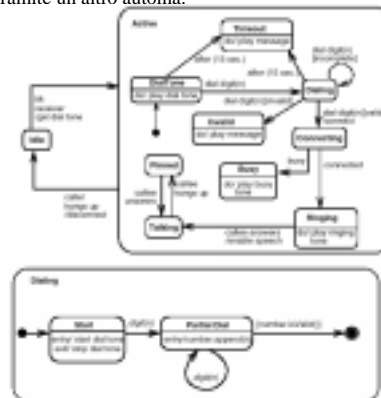


Paolo Nesi, 1998/2000

81

## Statechart Diagram

- Statechart Diagram possono essere utilizzati per rappresentare l'evoluzione dello stato degli oggetti di una classe.
- Ex. Comportamento di un apparecchio telefonico per il quale vengono definiti due stati principali, attivo ed inattivo (idle). Lo stato dialing viene a sua volta descritto tramite un altro automa.

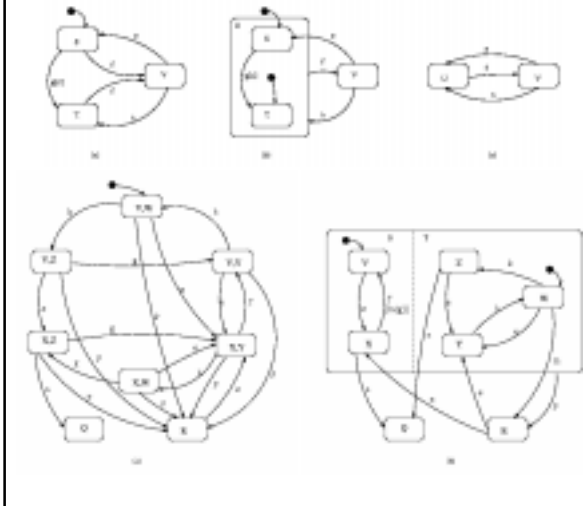


Paolo Nesi, 1998/2000

82

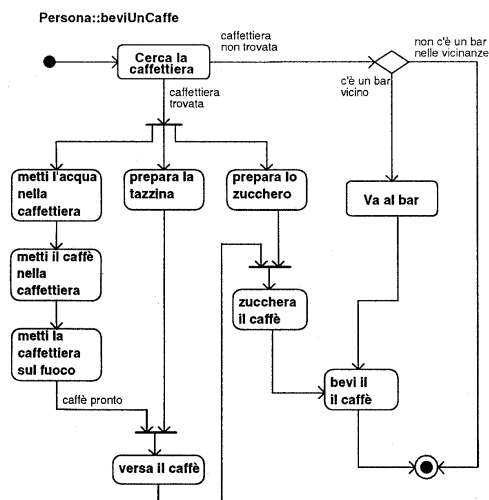
## Statechart Diagram

- Statecharts have been firstly introduced in 1987 as a visual notation for representing complex state machines, in a more synthetic manner with respect to the usually adopted notations based on state diagrams. With this notation, complex state machines are represented as combinations of simpler machines, through the XOR and AND mechanisms. In this way, the explosion of the number of states of conventional state diagrams is strongly reduced. On the other hand, this notation may be less intuitive than conventional state diagrams.



## Diagrammi di Attivita'

- Sono una specie di Flow Chart e sono utilizzati per descrivere singole attività delle classi, cioè il comportamento di metodi:

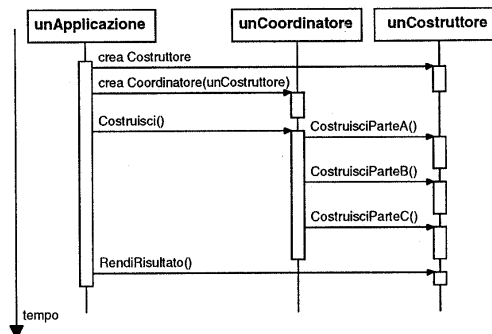


## Diagrammi di interazione

- Si dividono in
  - diagrammi sequenziali e
  - diagrammi di collaborazione
- Le interazioni sono tipicamente dovute a scambio di messaggi, chiamate di metodi

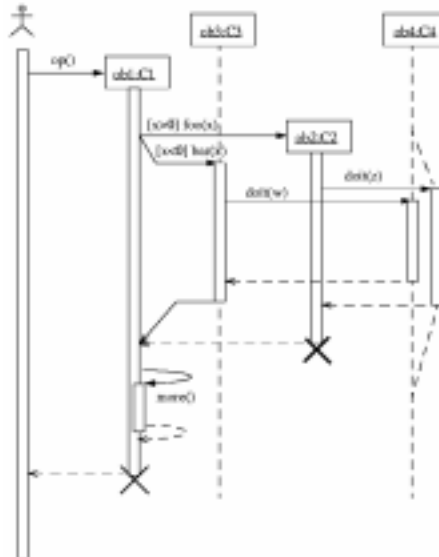
## Diagrammi Sequenziali

- Il diagramma sequenziale evidenzia il flusso temporale di elaborazione.
- Sequenza temporale delle interazioni fra oggetti del sistema, attraverso chiamate/messaggi.
- Non sono evidenziate le relazioni fra gli oggetti



## Diagrammi Sequenziali

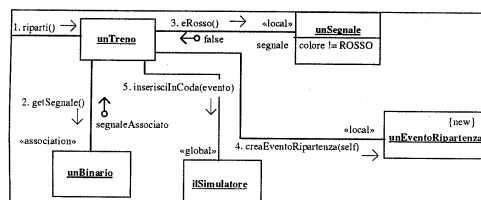
- Ex: tempistiche relative alla creazione e distruzione di oggetti, una scelta condizionata, meccanismi di ricorsione, di iterazione e di esecuzione parallela.



87

## Diagrammi di Collaborazione

- diagramma di collaborazione evidenzia le relazioni tra gli oggetti ed il modo con cui si possono scambiare i messaggi e le relazioni che vi intercorrono.
- E' una diversa forma per rappresentare lo stesso tipo di informazioni presenti in un sequence diagram.
- Nei collaboration diagram si rappresentano gli oggetti e le relazioni come in un object diagram, e si annotano le relazioni con l'indicazione dei messaggi scambiati fra gli oggetti.



Scenario di simulazione:

Un treno fermo ad un segnale riceve il messaggio "riparti".  
 Esso dapprima ottiene il segnale che lo blocca dal binario in cui è posizionato.  
 Poi chiede al segnale se è rosso. Se no, crea un oggetto di classe Evento-  
 Ripartenza (inviando un messaggio a tale classe) e lo inserisce nella  
 coda degli eventi del simulatore, al tempo corrente.

## Diagrammi di Collaborazione

- I possibili stereotipi di collaborazione sono:
  - *associazione*: il ricevente è visibile in quanto associato al mittente;
  - *global*: il ricevente è una variabile globale;
  - *local*: il ricevente è una variabile locale dell'operazione eseguita dal mittente;
  - *parameter*: il ricevente è un parametro dell'operazione eseguita dal mittente;
  - *self*: il ricevente è lo stesso mittente

## Diagrammi di Collaborazione

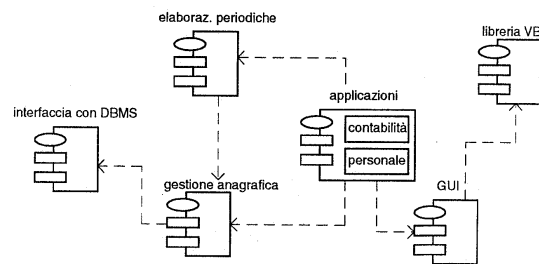
- Ex: un controller (rappresentato attraverso un oggetto) e le operazioni che questo compie per la stampa di linee su di una finestra. Vari elementi che vengono invocati (sono rappresentati come oggetti tramite la notazione Object Diagram) e con quale ordine. È possibile inoltre definire operazioni iterative e cicli.





## Diagramma dei Componenti

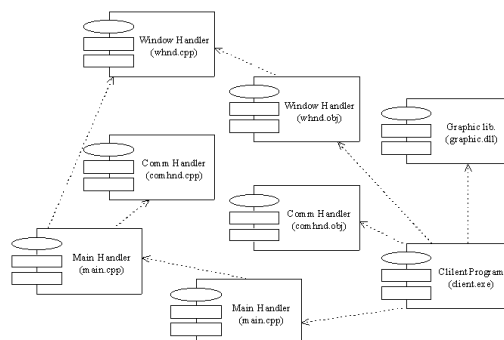
- Il **diagramma dei componenti** mostra graficamente i vari moduli software, file sorgenti, del sistema e le loro interdipendenze.
- I moduli possono essere legati da relazioni di dipendenza (le solite frecce tratteggiate), e di contenimento (un modulo contiene altri moduli).
- Il significato di un diagramma dei moduli è legato allo specifico linguaggio e sistema operativo in cui è realizzato il sistema software descritto.



Paolo Nesi, 1998/2000

93

## Diagrammi di Componenti

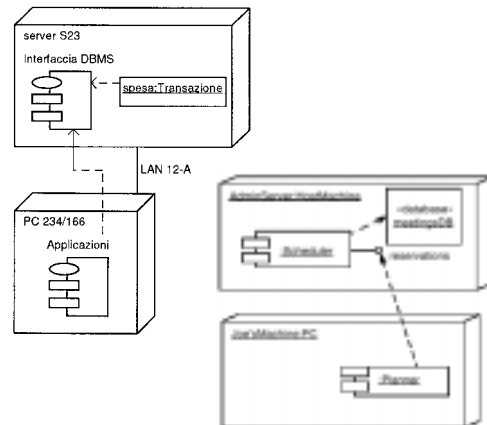


Paolo Nesi, 1998/2000

94

## Diagrammi di Dispiegamento

- diagrammi di dispiegamento mostrano graficamente i vari "nodi" hardware su cui sono allocati i processi e le applicazioni.



Paolo Nesi, 1998/2000

95

## Object Constraint Language

- In un linguaggio di specifica non è sempre possibile esprimere tutto in modo semplice e conciso, per permettere di chiarire alcuni particolari di un elemento in un diagramma è possibile aggiungere a questo delle note attraverso un oggetto grafico comune a tutti i tipi di diagrammi.
- Spesso le note contengono commenti, appunti o domande del progettista che verranno poi sottoposte al committente per dirimere eventuali dubbi in fase di specifica; il tipo di nota può essere rappresentato attraverso l'uso di stereotipi.
- OCL viene utilizzato anche per definire dei vincoli
- Questi possono essere tipicamente delle precondizioni e postcondizioni sulla specifica come sul comportamento del sistema
- Adatto per verifica formale dei requisiti. External verification, verification of consistency.

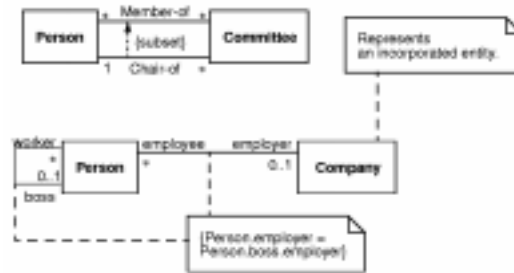
Paolo Nesi, 1998/2000

96



## Object Constraint Language

- Condizione di subset
- commenti
- relazioni
- ...
- .



## Occorre sempre usare tutto UML?

- UML è un linguaggio molto ricco, ma per questo presenta moltissimi costrutti e difficile da imparare.
- In pratica, se ne può efficacemente usare anche solo un sottoinsieme, "cucito" secondo le proprie esigenze ed il linguaggio Target utilizzato (certi dettagli troppo implementativi lo impongono).
- Object Constraint Language (OCL) è complesso e non ancora ben definito. È necessario definire esattamente come deve essere utilizzato in ogni progetto.
- In generale, UML è un linguaggio fin troppo ricco per cui occorre sapersi limitare e fermare l'analisi ed il progetto al *giusto* livello. Oltre, si può passare efficacemente al codice.

## Diagrammare o non diagrammare

- *diagrammi dei casi d'uso*: aiutano a documentare il sistema, ma aggiungono poca informazione. Attenzione! Si parla del diagramma, non dei casi d'uso!
- *diagrammi delle classi*: sono essenziali per ogni analisi. Sono il fondamento del sistema e la base per la generazione automatica del codice. Alcuni costruiti non sono disponibili in tutti i linguaggi:
  - classi associative (sostituibili dalle classi)
  - associazioni qualificate e n-arie (eredità dei diagrammi E-R)
  - "class utility", classi parametriche, vari tipi di vincoli sulle associazioni sono utili solo per certi linguaggi (C++) e in fase di progetto avanzato.
- *diagrammi sequenziali e di collaborazione*: usarli solo per descrivere quei 10-20 casi d'uso c/o scenari critici di funzionamento. Servono per una migliore comprensione del sistema, non a descrivere tutto il suo funzionamento!

## Diagrammare o non diagrammare

- *diagrammi di stato*: usarli per le singole classi i cui oggetti hanno stato. Vanno complementati con tabelle attributi/stati e operazioni/stati. Danno dei problemi visto che lo stato e' un proprietà trasversale al quale partecipano molti metodi. L'evoluzione descritta deve essere di alto livello.
- *diagrammi di attività*: equivalgono ai diagrammi di flusso tradizionali. Meglio usare direttamente un linguaggio di alto livello.
- *diagrammi dei componenti*: dei "makefile" grafici! Meglio usare dei veri makefile!
- *diagrammi di dispiegamento*: aggiungono di per se poca informazione, ma sono utili per la descrizione strutturale di grossi sistemi. Tipicamente distribuiti.

## Conclusioni 1/2

- Il nuovo linguaggio di modellazione unificato di Booch, Jacobson e Rumbaugh, con contributi provenienti da molti altri esperti in metodiche OO, ha tutte le carte in regola per diventare uno standard nel settore dell'OOA e OOD.
- E' un metodo completo, flessibile e non ambiguo, adatto sia all'OOA che all'OOD.
- Se ne possono usare efficacemente anche dei sottoinsiemi, e modelli esistenti possono essere convertiti in modelli UML in modo relativamente semplice.
- UML è una notazione grafica usata come supporto alla documentazione e "formalizzazione" dei progetti.
- I progetti sono realizzati usando un processo di produzione, che specifica le varie attività e passi da svolgere per realizzare il sistema.
- L'insieme linguaggio + processo si chiama *metodo di progettazione*.
- Il linguaggio di modellazione è per molti versi la parte più importante di un metodo: ne è la parte pubblica, quella che utenti, analisti e sviluppatori devono comprendere.

## Conclusioni 2/2

- Una volta standardizzata la notazione grafica sarà utilizzabile indipendentemente dal contesto e dagli specifici campi applicativi.
- I tre autori di UML, insieme ad esperti di tutte le società elencate prima stanno lavorando ad un processo standard di produzione di software Object Oriented
- Con l'avvento di UML finalmente anche il software potrà essere sistematicamente progettato e documentato prima della sua codifica.
- Alcuni si sono spinti a prevedere che tra pochi anni il 70% dei sistemi saranno progettati usando UML
- In ogni caso, l'adozione di tecniche ad oggetti per progettare e sviluppare il software, e l'uso di una notazione standard come UML, costituisce un vantaggio competitivo essenziale per chiunque produca o dia in appalto la produzione di sistemi software

## Riferimenti

- Booch, Object Oriented Modeling, Prentice Hall, 1994
- OMG, Object Management Group
- P. Nesi, Better Managing Object Oriented projects, IEEE Software, 1998
- P. Nesi, Corso Object Oriented Modelling, CESVIT, ETNOTEAM, ESPITI, 1997
- TABOO, 1998
- Tutorial Prof. M. Marchesi, Univ. Cagliari
- UML and C++, Lee, Prentice Hall
- UML Notation Guide, ver. 1.1
- UML OCL specification
- UML Sematic Model
- UML Summary
- UML, Rational Rose, user manual
- WEBBook, Prof. A. Fuggetta, Univ. Milano