

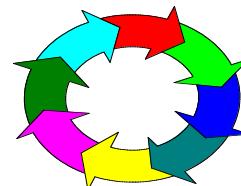
Il ciclo di vita dei sistemi informatici

1

1

Il ciclo di vita

- Comprende le attività svolte durante il periodo di esistenza di un sistema informatico
 - ⇒ definizione strategica
 - ⇒ pianificazione
 - ⇒ controllo di qualità
 - ⇒ analisi dei requisiti
 - ⇒ progettazione
 - del sistema
 - esecutiva
 - ⇒ realizzazione e collaudo in fabbrica
 - ⇒ certificazione
 - ⇒ installazione
 - ⇒ collaudo del sistema installato
 - ⇒ esercizio
 - ⇒ diagnosi e manutenzione
 - ⇒ evoluzione
 - ⇒ messa fuori servizio



2

Definizione strategica

- ⇨ Vengono prese decisioni sull' area aziendale che deve essere oggetto di automazione

Pianificazione

- ⇨ Vengono definiti gli obiettivi, evidenziati i fabbisogni e viene condotto uno *studio di fattibilità* per individuare possibili strategie di attuazione e avere una prima **idea dei costi, dei benefici e dei tempi**.

Controllo di qualità

- ⇨ Viene predisposto un *piano di controllo di qualità* per il progetto, allo scopo di garantire il rispetto delle specifiche e di controllare che il sistema realizzato si comporti come previsto

Analisi dei requisiti

- ⇨ Formalizza i requisiti avvalendosi di tecniche di modellazione della realtà e produce macro-specifiche per la fase di progettazione

Progettazione del sistema

- ⇨ Interpreta i requisiti in una soluzione architettonale di massima. Produce specifiche indipendenti dai particolari strumenti che saranno usati per la costruzione del sistema

Progettazione esecutiva

- ⇨ Vengono descritti struttura e comportamento dei componenti dell' architettura, producendo specifiche che possano dar luogo, attraverso il ricorso a strumenti di sviluppo opportuni, a un prodotto funzionante

Realizzazione e collaudo in fabbrica

Alfa test fatti in azienda dalla software house

- ⇨ Il sistema viene implementato sulla piattaforma prescelta e viene **testato internamente (α -test)** sulla base dei casi prova definiti durante la fase di analisi

3

Certificazione

- ⇨ L'attività di certificazione del software ha lo scopo di verificare che esso sia stato sviluppato secondo i criteri previsti dal metodo tecnico di progetto, in conformità alle specifiche di sistema e a tutta la documentazione di progetto

Installazione

- ⇨ Il sistema viene installato e configurato, e vengono recuperati gli eventuali dati pregressi

Collaudo del sistema installato

Beta test fatto dagli utenti

- ⇨ **Gli utenti testano "in vitro" il prodotto installato (β -test).** Si possono evidenziare *errori bloccanti* (malfunzionamenti che pregiudicano l' attività di collaudo), *errori non bloccanti* (malfunzionamenti che non pregiudicano l' attività di collaudo), problemi di *operatività* (una funzionalità richiesta non viene attuata adeguatamente) e *funzionali* (una funzionalità richiesta non è implementata)

Esercizio

- ⇨ Quando il collaudo dà esito positivo il sistema viene avviato ("**messo in produzione**"), inizialmente affiancando e poi sostituendo gradualmente l' eventuale sistema preesistente

Diagnosi

- ⇨ Durante l' esercizio gli utenti rilevano eventuali errori

Manutenzione

Paga la software house

- ⇨ **Gli errori** che si manifestano durante il funzionamento vengono segnalati e corretti (**manutenzione correttiva**). Può inoltre essere necessario intervenire sul software per adattarlo ai cambiamenti del dominio applicativo (**manutenzione adattativa**)

Evoluzione

MEV: il cliente cambia idea e si creano nuove funzionalità

- ⇨ Si valutano le possibilità di far evolvere il sistema incorporando nuove funzionalità o migliorandone l' operatività (**manutenzione evolutiva** o **perfettiva**)

Paga l'azienda

4

2

Analisi dei requisiti

- Scopo dell' analisi è produrre, utilizzando tecniche di modellazione della realtà, un documento di **specifiche dei requisiti** che diventi l' input per le successive fasi di progettazione e realizzazione
- Oggetto dell' analisi è l' organizzazione nel suo complesso:
 - ⇒ sottosistemi aziendali
 - ⇒ risorse
 - ⇒ processi
 - ⇒ flussi informativi
 - ⇒ ...



5

Specifiche dei requisiti

- La specifica dei requisiti è un accordo tra il produttore di un servizio e il suo consumatore
- In questa fase del ciclo di vita, attraverso la specifica dei requisiti l' utente finale e il progettista si accordano sulle funzionalità messe a disposizione dal software
- La difficoltà per questo tipo di specifica è data dalla diversità dei linguaggi usata dalle due parti
- Qualità per la specifica dei requisiti:
 - ⇒ **Chiarezza**: ogni specifica deve indicare quanto più chiaramente possibile le operazioni e i soggetti del processo che descrive
 - ⇒ **Non ambiguità**: il processo descritto dalla specifica deve essere definito in modo completo e dettagliato
 - ⇒ **Consistenza**: le specifiche non devono contenere punti contraddittori

6

Importanza dei Requisiti

Stadio	Costo relativo per la correzione
Requisiti	0.1 - 0.2
Progettazione	0.5
Codifica	1
Test	2
Accettazione	5
Manutenzione	20

*Più tardi viene scoperto un errore nel ciclo di sviluppo del software,
maggiore è il costo di riparazione.*

7

Analisi: alcune definizioni

- **De Marco:**
 - ⇒ l' analisi è lo studio di un problema, *prima* di intraprendere qualche azione
- **Coad:**
 - ⇒ l' analisi è lo studio del dominio di un problema, che porta a una specifica del comportamento *esternamente osservabile*; una descrizione *completa, coerente e fattibile* di ciò che occorre realizzare; una trattazione quantitativa delle caratteristiche operazionali (cioè *affidabilità, disponibilità, prestazioni*)
- **Davis:**
 - ⇒ l' analisi del problema è il momento in cui viene definito lo spazio del prodotto; la descrizione del prodotto comporta la scelta di una soluzione e l' esplicitazione del comportamento esterno del prodotto dimostrando che esso *soddisfa i requisiti*

Correttezza: il software soddisfa i requisiti (non vuol dire che non ha bug)

8

Cosa e come modellare

- Il processo di analisi è **incrementale** e porta per passi successivi alla stesura di un insieme di documenti in grado di rappresentare un modello dell' organizzazione e comunicare, in modo **non ambiguo**, una descrizione **esauriente, coerente e realizzabile** dei vari **aspetti statici, dinamici e funzionali** di un sistema informatico

9

Metodi di analisi

Differenti problemi richiedono differenti approcci e differenti strumenti di analisi

- Analisi orientata agli **oggetti** = statico
- Analisi orientata alle **funzioni** = funzionale
- Analisi orientata agli **stati** = dinamico

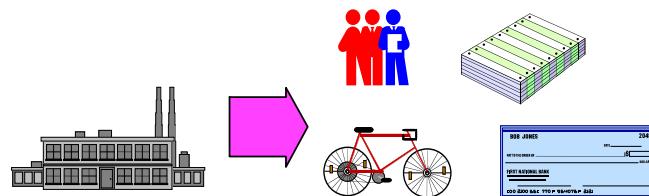
10

Analisi orientata agli oggetti

- L' enfasi è posta:

- ⇒ sull' identificazione degli **oggetti**
- ⇒ sulle **interrelazioni** tra oggetti

Come esempio gli schemi ER

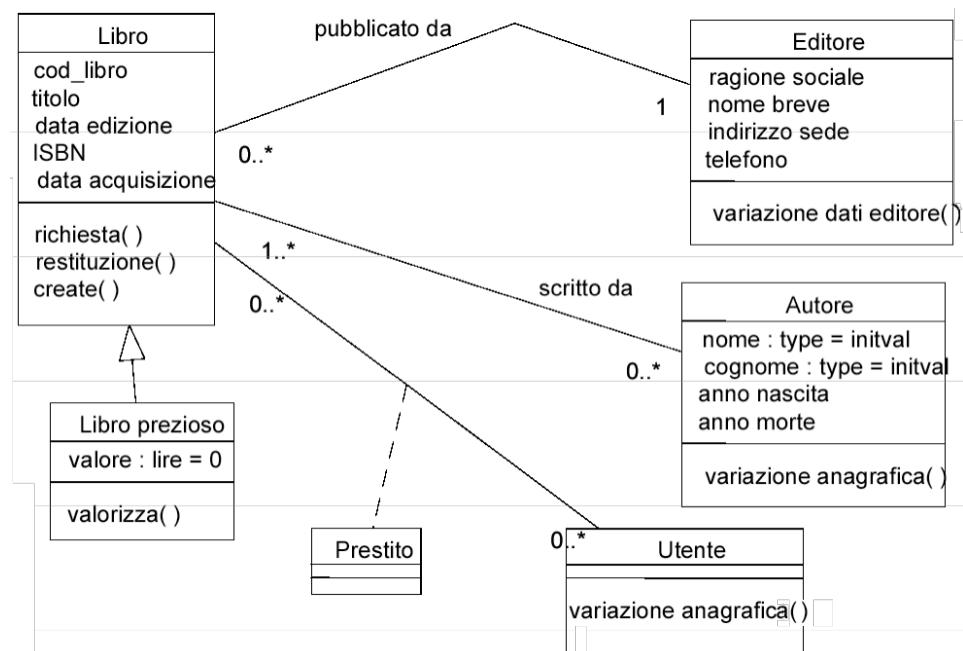


- *Nel tempo le proprietà strutturali degli oggetti osservati restano abbastanza stabili, mentre l'uso che degli oggetti si fa può mutare in modo sensibile*

11

Analisi orientata agli oggetti

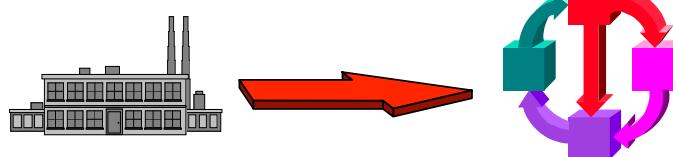
Leggere dritto:
Libro pubblicato da
1 Editore
Editore pubblica 0-n
Libri



12

Analisi orientata alle funzioni

- L'obiettivo è rappresentare un sistema come:
 - ⇒ un insieme di **flussi informativi**
 - ⇒ una **rete di processi** che trasformano flussi informativi



- Ciò corrisponde alla progressiva costruzione di una **gerarchia funzionale**

13

Analisi orientata alle funzioni

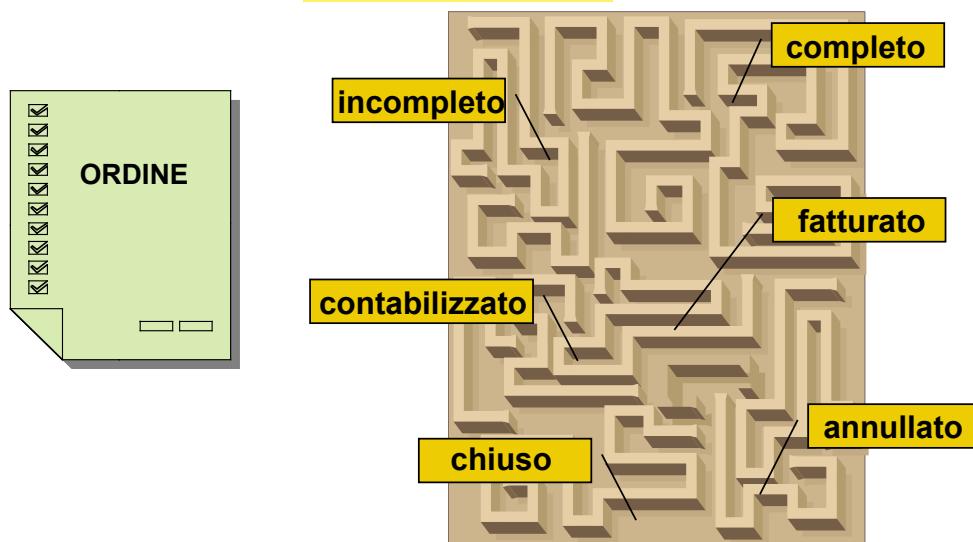
1 : Creazione_Vendita_Produzione

- 201 : Progettazione campionario
 - 3011 : Indagine di mercato
 - 3012 : Creazione stilistica
 - 3013 : Creazione prototipi
 - 40131 : Creazione cartamodello
 - 40132 : Realizzazione prototipo
 - 3014 : Definizione modelli campionario
 - 40141 : Scelta prototipi
 - 501411 : Riunione prima selezione
 - 501412 : Evidenziazione difficoltà realizzative
 - 501413 : Produzione schizzo
 - 40142 : Definizione tecnica modelli
 - 40143 : Codifica modelli
- 3015 : Produzione campionario
 - 40151 : Lancio in produzione del campionario
 - 40152 : Produzione campionario
 - 40153 : Rientro e controllo
- 202 : Pianificazione operativa
 - 3021 : Definizione obiettivi
 - 3022 : Elaborazione delle previsioni di vendita
 - 3023 : Stesura piano operativo
- 203 : Vendita
 - 3031 : Presentazione campionario
 - 3032 : Acquisizione ordini e controllo campagna vendita
 - 3023 : Revisione previsioni di vendita
- 204 : Produzione
 - 3041 : Approvvigionamento materie prime e prodotti finiti
 - 3042 : Ciclo di produzione
 - 40421 : Programmazione produzione
 - 40422 : Confezione
 - 40423 : Rientro e controllo della produzione

14

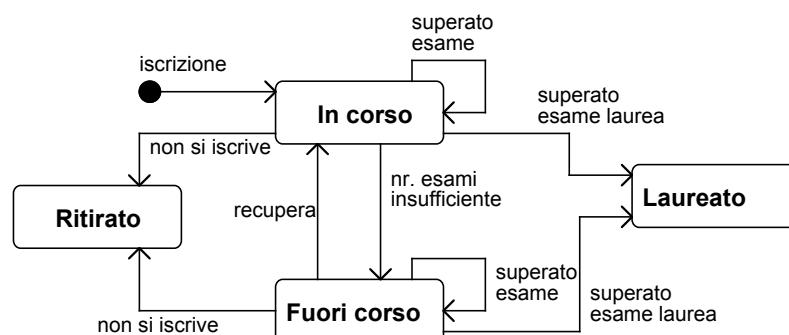
Analisi orientata agli stati

- Per alcune categorie di applicazioni può essere utile pensare fin dall'inizio in termini di **stati operativi**, in cui si può trovare il sistema allo studio, e **transizioni di stato**



15

Analisi orientata agli stati



16

Uso dei metodi d' analisi

- La tendenza attuale è integrare metodi dei tre tipi, tenendo però conto della tipologia di applicazione

- ⇒ **Applicazioni orientate agli oggetti**

- l'aspetto più significativo è costituito dalle informazioni, le funzioni svolte sono relativamente semplici

- ⇒ **Applicazioni orientate alle funzioni**

- la complessità risiede nel tipo di trasformazione input-output operata

- ⇒ **Applicazioni orientate al controllo**

- l'aspetto più significativo da modellare è la sincronizzazione fra diverse attività cooperanti nel sistema

17

Il ruolo dell' astrazione

- Molteplici sono le relazioni in gioco fra oggetti, funzioni e stati e molteplici i livelli di possibile dettaglio:

- ⇒ Gli **oggetti** possono essere descritti a partire da termini molto generici (*edificio, strada*) fino ad arrivare a livello di dettaglio specifici (*la torre degli Asinelli, via Saragozza*)
 - ⇒ Le **funzioni** possono essere espresse inizialmente in modo vago (*controllare il livello di gas nocivi nell' aria*) e successivamente precise (*la programmazione del livello di soglia per l' allarme della centralina viene attivata premendo il pulsante P*)
 - ⇒ Gli **stati** possono essere decretati a un elevato livello di astrazione (*la centralina è in stato di errore*) o specificati in maggior dettaglio (è acceso il segnalatore di errore nel sensore S)

18

Meccanismi di Astrazione

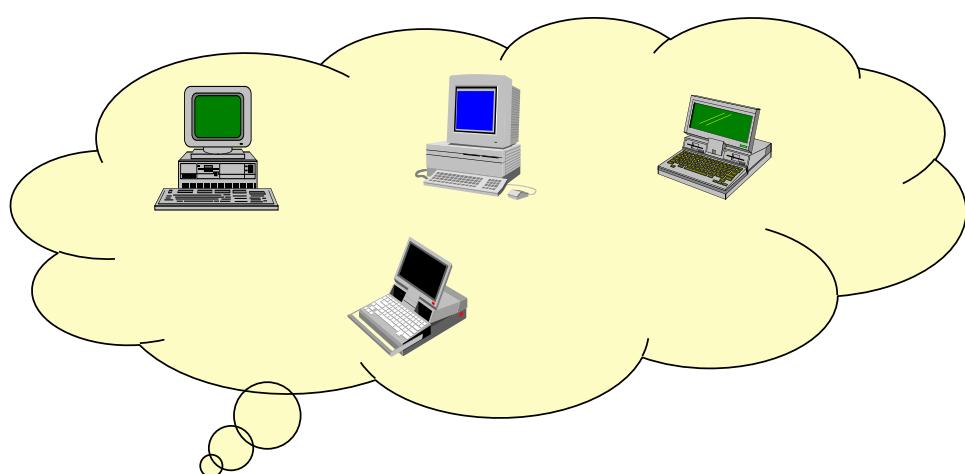
- I principali meccanismi di astrazione usati durante il processo di analisi per costruire una base di conoscenza sul problema sono **classificazione**, **generalizzazione**, **aggregazione** e **associazione**

19

Classificazione

- La **classificazione** consente di raggruppare in classi oggetti, funzioni, o stati in base alle loro proprietà

Funziona per
gruppi di oggetti



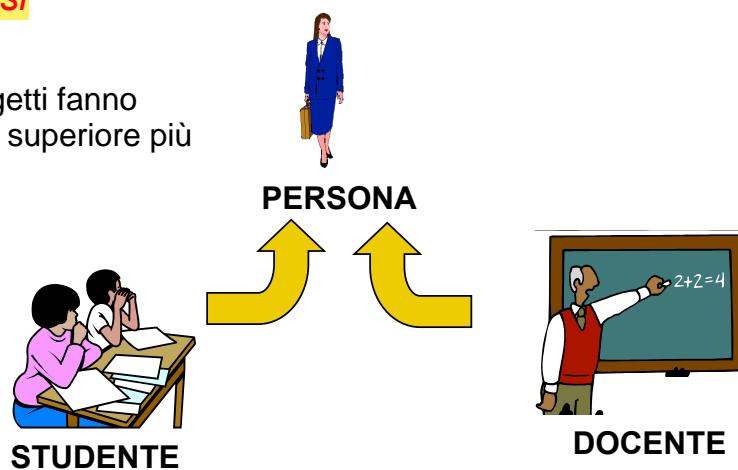
la classe dei computer

20

Generalizzazione

- La generalizzazione cattura le relazioni **è-un** ovvero permette di astrarre le caratteristiche comuni fra più classi definendo **superclassi**

Quando due o più oggetti fanno parte di una categoria superiore più generale

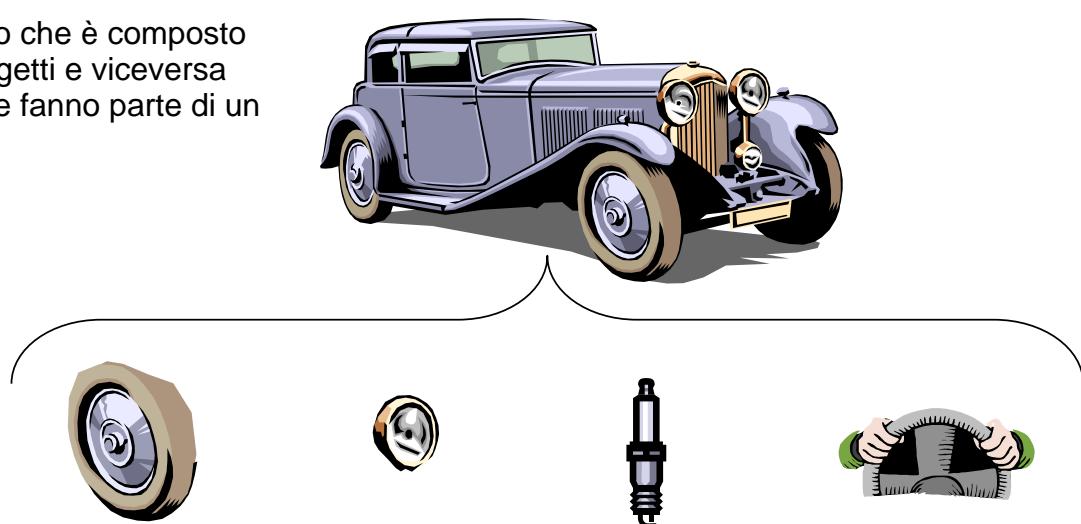


21

Aggregazione

- L'aggregazione esprime le relazioni **parte-di** che sussistono tra oggetti, tra funzioni, tra stati

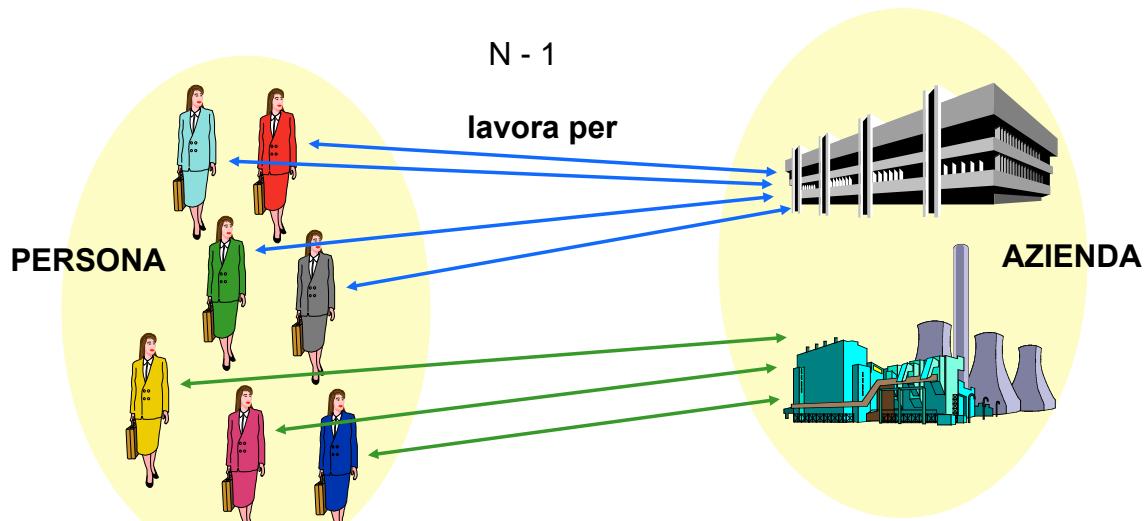
Un oggetto che è composto da altri oggetti e viceversa
oggetti che fanno parte di un
oggetto



22

Associazioni

- Oltre ai meccanismi citati è importante modellare le associazioni che sussistono fra le varie classi



23

Linguaggi per la specifica dei requisiti

- Linguaggi informali**
 - Il linguaggio naturale, alla base della comunicazione durante le interviste tra analista e utente, non può essere adottato come unico mezzo per produrre documenti di specifica per le innumerevoli ambiguità di significato
- Linguaggi semiformali**
 - notazione grafica, che presenta una semantica sfumata, accoppiata con descrizioni in linguaggio naturale (esempi : E/R, DFD)
- Linguaggi formali**
 - linguaggi di specifica basati sulla logica dei predicati
 - linguaggi di specifica algebrici
 - linguaggi concettuali per basi di dati

24

Linguaggi per la specifica dei requisiti

La diversità degli obiettivi posti dalla specifica dei requisiti implica l'utilizzo di notazioni diverse per la rappresentazione delle informazioni

Può essere ambiguo ma è più facile comprendere

Formalismi operazionali: definiscono il sistema descrivendone il comportamento (normalmente mediante un modello)

"E è il percorso che si ottiene muovendosi in modo che la somma delle distanze tra due punti fissi p1 e p2 rimanga invariata"

Non è ambiguo ma deve avere
è più difficile
da
comprendere

Formalismi dichiarativi: definiscono il sistema dichiarando le proprietà che esso

$$ax^2+by^2+c=0$$

L'approccio operazionale fornisce una rappresentazione più semplice poiché più simile al modo di ragionare della mente umana:

- Facilità di acquisizione
- Facilità di verifica della correttezza
- Facilità di comprensione da parte dei programmatore

L'approccio dichiarativo fornisce una rappresentazione che non si presta ad ambiguità

25

3

Progettazione

- Riguarda tutte quelle attività che permettono di passare dalla raccolta ed elaborazione dei requisiti di un sistema software alla sua effettiva realizzazione, pertanto fa da ponte tra la fase di specifica e la fase di codifica
- Durante la fase di progettazione si decidono le modalità di passaggio da "che cosa" deve essere realizzato (specifiche dei requisiti) a "come" la realizzazione deve avere luogo
- Il sistema complessivo viene suddiviso in più sottosistemi **Vantaggi:**
 - ⇒ la complessità delle singole parti è minore della complessità totale originaria
 - ⇒ i sottosistemi ottenuti possono essere realizzati ed analizzati da gruppi diversi di programmatore in modo il più possibile indipendente

26

Progettazione

- Due esigenze contrastanti:
 - ⇒ progetto sufficientemente astratto per poter essere agevolmente confrontato con le specifiche da cui viene derivato
 - ⇒ progetto sufficientemente dettagliato in modo tale che la codifica possa avvenire senza ulteriori necessità di chiarire le operazioni che devono essere realizzate
- A seconda della tecnica impiegata per la progettazione, la realizzazione del sistema può risultare più o meno naturale ed immediata
 - ⇒ *Ad esempio:*



27

Progettazione

- **Non esiste un metodo generale per la progettazione del software**
 - ⇒ Occorre considerare le tipologie di software (software sequenziale, concorrente ed in tempo reale)
- A una stessa specifica possono corrispondere più progetti, ossia più metodi di soluzione diversi
- Le scelte di progetto devono poter cambiare in risposta a mutate esigenze di vario tipo senza che per questo tutto il progetto e perciò tutto il software prodotto debba essere modificato radicalmente
- **Il progetto di un sistema software è perciò un'attività altamente creativa**, che richiede un insieme di abilità a coloro che vi sono preposti

28

Obiettivi della progettazione

- Produrre software con le caratteristiche di qualità che sono state dettagliate nella fase di analisi e specifica dei requisiti. Ad esempio:
 - ⇒ affidabilità
 - ⇒ modificabilità
 - ⇒ comprensibilità
 - ⇒ riusabilità
- ...obiettivi che si possono riassumere nella diminuzione dei *costi* e *tempi* di produzione e nell'aumento della *qualità* del software

I costi maggiori riguardano la fase di manutenzione del software



capacità di poter far fronte a modifiche da effettuare senza che l'intera struttura dell'applicazione già costruita debba essere messa nuovamente in discussione ed elaborata

Il paradigma a oggetti

1

1

Il paradigma a oggetti

□ I concetti fondamentali:

- ⇒ oggetto
- ⇒ astrazione
- ⇒ classe
- ⇒ encapsulamento
- ⇒ ereditarietà
- ⇒ polimorfismo - late binding
- ⇒ delegazione

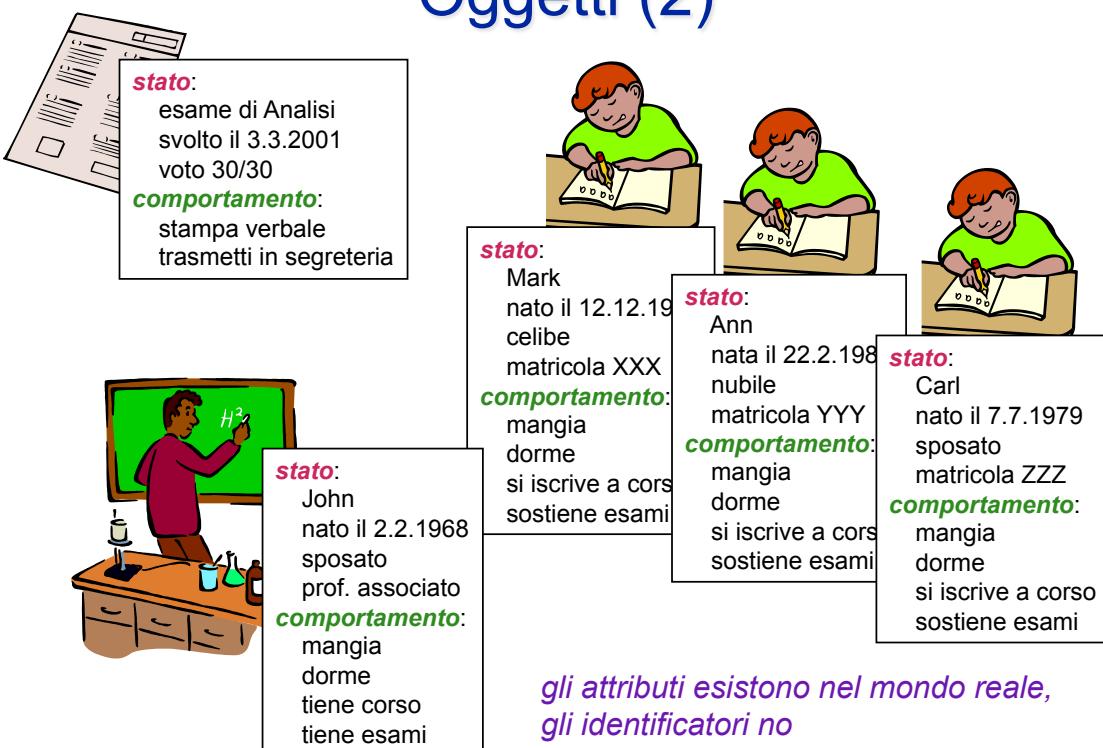
2

Oggetti

- Sono gli elementi di base del paradigma, e corrispondono a entità (non necessariamente "fisiche") del dominio applicativo
 - ⇒ Esempi (in un'aula universitaria): le sedie, gli studenti che le occupano, il professore che tiene la lezione, il corso seguito dagli studenti
 - Un oggetto è un individuo sostanziale che possiede un identità e un insieme di proprietà, che ne rappresentano lo stato e il comportamento
- Un oggetto è una tripla
- Ogni oggetto è caratterizzato da:
 - ⇒ una identità (OID, Object Identifier) che gli viene associata all'atto della creazione, non può essere modificata ed è indipendente dallo stato corrente dell'oggetto
 - ⇒ uno stato definito come l'insieme dei valori assunti a un certo istante da un insieme di attributi
 - ⇒ un comportamento definito da un insieme di operazioni
 - Poiché un oggetto può anche includere riferimenti ad altri oggetti, risulta possibile creare oggetti complessi.

3

Oggetti (2)



4

Operazioni e interfaccia

Signature è
formata da
nome,
parametri,
valore
restituito

- Ogni operazione dichiarata da un oggetto specifica il **nome** dell'operazione, gli oggetti che prende come **parametri** e il **valore restituito** (**signature**)
 - ⇒ L'oggetto su cui l'operazione opera è definito implicitamente

Interfaccia
è l'insieme
di tutte le
signature

- L'insieme di tutte le signature delle operazioni di un oggetto sono dette **interfaccia** dell'oggetto
 - ⇒ L'interfaccia specifica l'insieme completo di tutte le richieste che possono essere inviate all'oggetto

Interfaccia è l'insieme di tutti i servizi che l'oggetto può offrire

5

Tipo di dati astratto

- E' una rappresentazione di un insieme di oggetti "simili", caratterizzato da una **struttura** per i dati e da un'**interfaccia** che definisce quali sono le operazioni associate agli oggetti, ovvero l'insieme dei **servizi** implementati
- Un tipo è **sottotipo** di un **supertipo** se la sua interfaccia contiene quella del **supertipo**
 - ⇒ Un sottotipo eredita l'interfaccia del suo supertipo
 - ⇒ L'interfaccia non vincola l'implementazione del servizio offerto ovvero il comportamento effettivo
 - ⇒ Oggetti con la stessa interfaccia possono avere implementazioni completamente diverse

Da intendere come una classe astratta con metodi

6

Classe

- Fornisce una realizzazione di un tipo di dati astratto, specifica cioè un'implementazione per i metodi a esso associati
 - ⇒ **Esempi:** classe delle sedie, degli studenti, dei professori, dei corsi

1 - 1

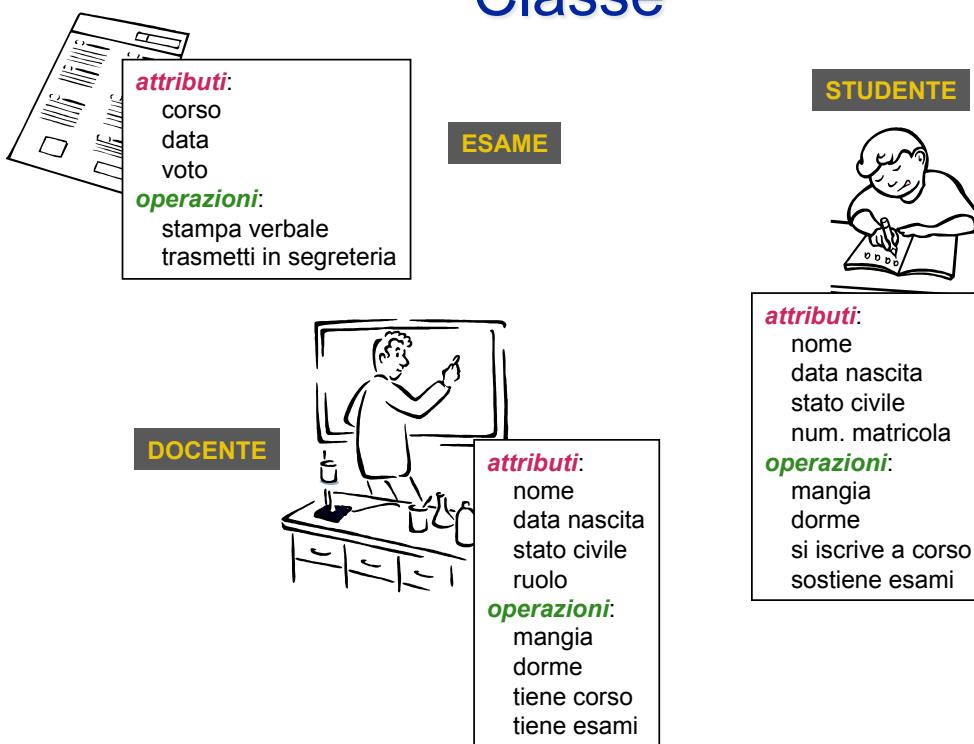
Un oggetto è sempre istanza di esattamente una classe

- Tutti gli oggetti di una classe hanno gli stessi attributi e metodi. Esistono metodi di due tipi: quelli che restituiscono astrazioni significative sullo stato dell'oggetto cui sono applicati, e quelli che ne alterano lo stato

Due tipi di metodi: get e set

7

Classe



8

Incapsulamento

Diminuisce la possibilità di errori

- Protegge l'oggetto nascondendo lo stato dei dati e l'implementazione delle sue operazioni
- Un oggetto incapsula i dati (**attributi**) e le procedure (**operazioni**) che li possono modificare
- *Il principio di **incapsulamento** sancisce che gli attributi di un oggetto possono essere letti e manipolati solo attraverso l'interfaccia che l'oggetto stesso mette a disposizione*
 - ⇒ I dettagli dell'implementazione di una classe sono **privati**, cioè manipolabili direttamente solo dai metodi della classe e quindi protetti
 - ⇒ L'accesso dall'esterno agli attributi della classe avviene attraverso una ristretta **interfaccia pubblica**, costituita da un sottoinsieme dei metodi della classe
 - ⇒ Un oggetto esegue una operazione quando riceve una richiesta (**messaggio**) da un oggetto client

9

Vantaggi dell'incapsulamento

- Per l'**utilizzo** di una classe è sufficiente conoscerne l'**interfaccia pubblica**; i dettagli implementativi sono nascosti all'interno. La classe viene quindi vista come una "scatola nera"
- La **modifica dell'implementazione** di una classe non si ripercuote sull'applicazione, a patto che non ne venga variata l'interfaccia
- Poiché la manipolazione diretta degli attributi della classe avviene esclusivamente tramite i suoi metodi, viene fortemente ridotta la possibilità di commettere **errori** nella gestione dello stato degli oggetti
- Il **debugging** delle applicazioni è velocizzato, poiché l'incapsulamento rende più semplice identificare la sorgente di un errore

10

Operazioni e Metodi

- Un **metodo** cattura l'implementazione di una operazione

- I metodi possono essere classificati in:

Constructor ↳ **costruttori**, per costruire oggetti a partire da parametri di ingresso restituendo l'OID dell'oggetto costruito

Garbage Collector ↳ **distruttori**, per cancellare gli oggetti ed eventuali altri oggetti ad essi collegati

Get ↳ **accessori**, per restituire informazioni sul contenuto degli oggetti (proprietà derivate)

Set ↳ **trasformatori**, per modificare lo stato degli oggetti e di eventuali altri oggetti ad essi collegati

- I metodi possono essere:

 ↳ **pubblici**

 ↳ **protetti**

 ↳ **privati**

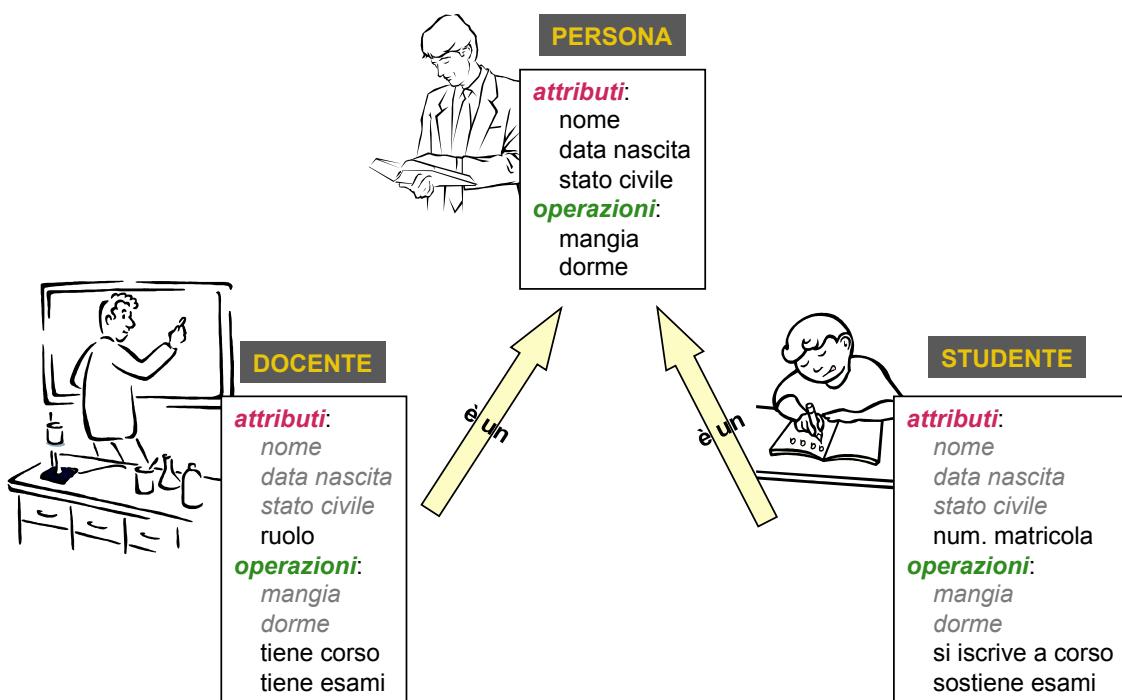
11

Ereditarietà

- Il meccanismo di **ereditarietà** permette di basare la definizione e implementazione di una classe su quelle di altre classi.
- E' possibile definire relazioni di **specializzazione/ generalizzazione** tra classi: la classe generalizzante viene detta **superclasse**, la classe specializzante viene detta **sottoclasse** o **classe derivata**
 - ↳ **Esempio:** le classi studente e professore sono entrambe derivate dalla classe persona
- Ciascuna sottoclasse eredita dalla sua superclasse la struttura ed i comportamenti, ovvero gli attributi, i metodi e l'interfaccia; può però specializzare le caratteristiche ereditate e aggiungere caratteristiche specifiche non presenti nella superclasse

12

Ereditarietà (2)



13

Ereditarietà (3)

- Si parla di **ereditarietà multipla** quando una sottoclasse può essere derivata contemporaneamente da più superclassi
 - ⇒ in caso di conflitti tra attributi o metodi ereditati da due superclassi, occorre individuare opportune strategie di risoluzione
- Poiché una classe derivata può essere ulteriormente specializzata, si vengono a formare **gerarchie** di classi, strutturate come alberi in caso di ereditarietà singola e come reticolati in caso di ereditarietà multipla
- Date due classi A e B di cui B è una sottoclasse di A, esiste di fatto la relazione **B is-a A** (B è un A)
 - ⇒ gli oggetti istanze di B possano a tutti gli effetti essere utilizzati al posto di oggetti istanze di A (ad esempio, uno studente è una persona)
 - ⇒ **Non è vero il contrario** (non è detto che una persona sia uno studente)

14

Polimorfismo

capacità di assumere forme molteplici

- Nel paradigma a oggetti si usa questo termine per alludere alla possibilità di creare metodi con lo stesso nome ma implementazioni differenti
- Tramite il meccanismo di **overload** è possibile definire, all'interno di una stessa classe, più metodi con lo stesso nome ma **signature** (insieme dei parametri) differenti
 - ⇒ A fronte di un messaggio inviato per invocare il metodo, sarà il sistema a scegliere l'implementazione da considerare, sulla base della struttura del messaggio stesso

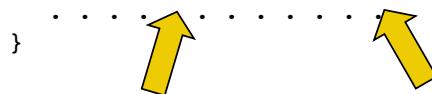
Overload: Sovraccaricare lo stesso metodo con più implementazioni

15

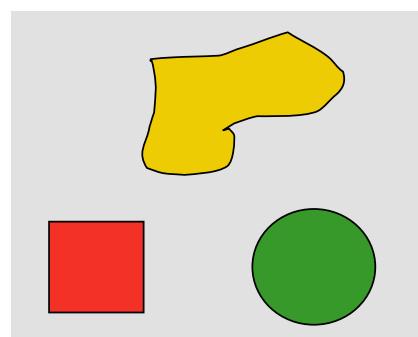
Polimorfismo (2)

- Possibilità di **ridefinire**, all'interno di una sottoclasse, l'implementazione di un metodo ereditato (**override**)

```
class figuraGeometrica
{ // attributi
    int posizioneX; int posizioneY;
    int coloreContorno;
    int coloreRiempimento;
    // metodi
    public:
        void trasla(int shiftX, int shiftY);
        void ruota(int angoloRotazione);
    . . .
}
```



```
class quadrato:figuraGeometrica { int lato; int angolo
} class cerchio:figuraGeometrica { int raggio;
}
```

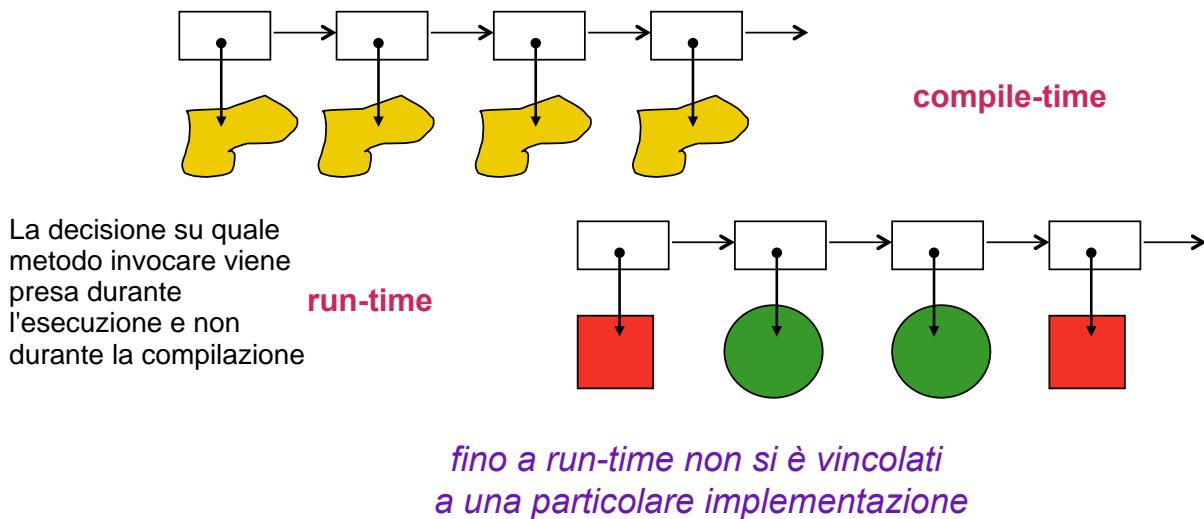


ridefinizione di **trasla** e **ruota**

16

Istanziamento dinamico (*late binding*)

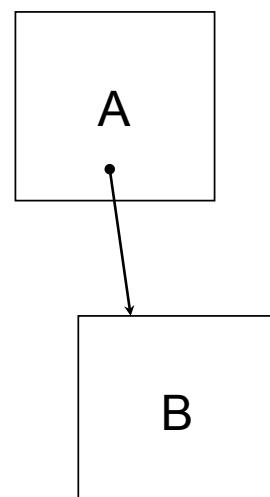
- Il polimorfismo, abbinato all'istanziamento dinamico, permette a ciascun oggetto di rispondere a uno stesso messaggio in modo appropriato a seconda della classe da cui deriva



17

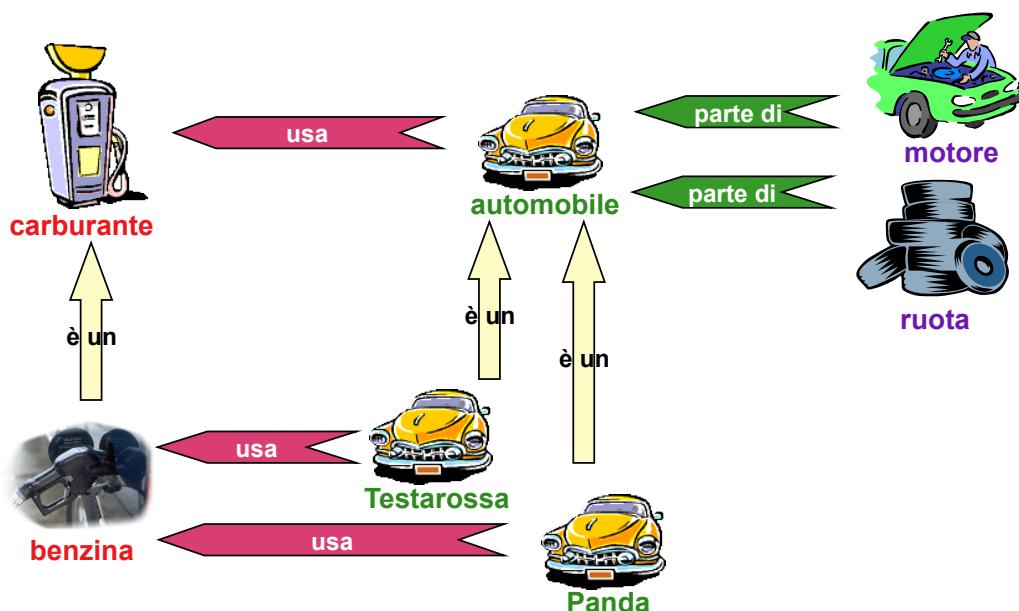
Delegazione

- Si parla di delegazione quando un oggetto A contiene al suo interno un riferimento a un altro oggetto B, cosicché A (che risulta essere in questo caso un *oggetto complesso*) può delegare alcune funzioni alla classe a cui appartiene B
 - Esempio: Dovendo definire una classe persona, gli attributi nome, cognome e indirizzo saranno dichiarati come puntatori a oggetti di classe stringa, delegando così a quest'ultima classe le operazioni di manipolazione
- La delegazione costituisce il meccanismo fondamentale per implementare *associazioni tra classi*
 - Esempio: per rappresentare l'associazione di inclusione tra un aeroplano e il suo motore, si includerà in ogni oggetto di classe aeroplano un puntatore a un oggetto di classe motore



18

Un esempio



19

2

Lo sviluppo di sistemi a oggetti

Imparare una nuova tecnica di progettazione è molto più difficile che imparare un nuovo linguaggio, poiché richiede di modificare sostanzialmente il nostro modo di pensare

- Il bisogno di sviluppare e mantenere sistemi di grandi dimensioni e complessi in ambienti dinamici crea un forte interesse in nuovi approcci al problema del design
- L'obiettivo principale dell'approccio orientato agli oggetti (OO, *object-oriented*) è migliorare la produttività aumentando l'estendibilità e la riusabilità del software e controllando la complessità e i costi della manutenzione

20

Dall'approccio funzionale...

- La **decomposizione funzionale** è un'analisi di tipo top-down tradizionalmente impiegata nel paradigma procedurale, basata sui concetti di **procedura** e **flusso di dati**
 - ⇒ La domanda fondamentale è: *cosa fa il sistema, qual è la sua funzione?*
 - ⇒ Ad alto livello di astrazione, il sistema viene caratterizzato tramite *un'unica funzionalità*
 - ⇒ I blocchi di base dell'applicazione sono i task (*compiti*), che durante l'implementazione daranno luogo a procedure, e sono legati alla specifica soluzione proposta
- Principali problemi:
 - ⇒ Nessun modello unificante per integrare le diverse fasi: c'è una forte **discrepanza** tra concetto di flusso di dati utilizzato nell'analisi e concetto di gerarchia di compiti utilizzato nella progettazione
 - ⇒ **Mancanza di iterazione** nella progettazione: si adotta il **modello a cascata**, in cui le attività sono viste come una progressione lineare
 - ⇒ **Mancanza di estendibilità**: non si considerano le possibili evoluzioni del sistema
 - ⇒ **Poca attenzione al problema della riusabilità**: ogni sistema viene ricostruito a partire da zero, per cui i costi di manutenzione sono alti
 - ⇒ La **progettazione dei dati viene trascurata**, poiché le strutture dati sono determinate dalle strutture procedurali

21

...all'approccio a oggetti

- ★ **ANALISI**: va dall'inizio del progetto fino all'analisi delle specifiche utente e allo studio di fattibilità (*cosa* il sistema deve fare)
- ★ **DESIGN**: progettazione logica e fisica del sistema (*come* lo deve fare)
- ★ **IMPLEMENTAZIONE**: scrittura del codice, test di verifica, validazione, manutenzione
 - ⇒ I confini tra le fasi non sono più distinti, infatti il centro di interesse è lo stesso: gli **oggetti** e le loro **interrelazioni**
 - ⇒ Il processo di sviluppo OO è **iterativo**: si adotta il **modello a fontana**, in cui lo sviluppo raggiunge un alto livello per poi ritornare a un livello precedente e risalire di nuovo
 - ⇒ L'ereditarietà permette di aggiungere nuove caratteristiche a un sistema riducendo i costi di manutenzione (**estendibilità**), e di costruire nuove funzionalità a partire dall'esistente (**riusabilità**) riscrivendo solo quella parte di codice inadeguato e solo per gli oggetti che ne hanno bisogno

22

Benefici dell'approccio a oggetti

- La decomposizione è orientata alla modellazione
 - ⇒ I blocchi di base dell'applicazione sono entità che interagiscono, modellate come classi di oggetti, e sono legate alla formulazione originale del problema
 - ⇒ I risultati dell'analisi non sono un semplice input del design, ma ne sono parte integrante: **analisi e design lavorano insieme** per sviluppare un modello del dominio del problema
- Il progetto dettagliato è rimandato nel tempo e nascosto all'interno di ciascuna classe
 - ⇒ Algoritmi e strutture dati non sono più "congelati" a un alto livello del progetto
 - ⇒ Si ha più flessibilità, poiché un cambiamento nell'implementazione non implica variazioni consistenti alla struttura del sistema

23

Benefici dell'approccio a oggetti

- I sistemi sviluppati a oggetti risultano più stabili nel tempo di quelli progettati per decomposizione funzionale
 - ⇒ Le caratteristiche dei domini applicativi variano più lentamente nel tempo rispetto alle funzionalità richieste ai sistemi
- La produttività è alta
 - ⇒ Fasi diverse dell'analisi dei requisiti e del ciclo di vita possono essere svolte contemporaneamente
- C'è la possibilità di sviluppare rapidamente **prototipi** che possono risultare di valido ausilio per la certificazione dell'analisi dei requisiti
- E' possibile che il design e l'implementazione a classi richiedano tempi elevati, volendo provvedere generalità e riusabilità; a fronte di ciò si ha però una drastica riduzione dei **costi di manutenzione**

24

Object-oriented analysis

“Di che cosa necessita il programma?”

“Quali classi saranno presenti?”

“Qual è la responsabilità di ciascuna classe?”

□ Attività:

- ⇒ determinare la funzionalità del sistema
 - ⇒ creare una lista delle classi che sono parte del sistema
 - ⇒ distribuire le funzionalità del sistema attraverso le classi individuate
- In una buona analisi ...
- ⇒ le classi sono relativamente “piccole” e molte sono abbastanza generali da poter essere riusate in futuri progetti
 - ⇒ le responsabilità e il controllo sono distribuiti, in altre parole il progetto non ha un “centro” esplicito
 - ⇒ ci sono poche assunzioni riguardo al linguaggio di programmazione da usare

25

Object-oriented design

“Come gestirà la classe le sue responsabilità?”

“Quali informazioni sono necessarie alla classe?”

“Come comunicheranno le classi tra loro?”

□ Attività:

- ⇒ determinare metodi e attributi di ciascuna classe
- ⇒ progettare algoritmi per implementare le operazioni
- ⇒ progettare le associazioni

□ In un buon design...

- ⇒ i percorsi di accesso ai dati sono ottimizzati
- ⇒ le classi sono raggruppate in moduli

26

Alcuni approcci object-oriented

- Booch OOD
- Coad-Yourdon OOA/OOD
- Jacobson OOSE
- Rubin-Goldberg OBA
- Rumbaugh OMT
- Shlaer-Mellor OOA
-



Unified Modeling Language (UML)

1

1

Introduzione



- UML nasce come *standard aperto* dalla collaborazione fra tre dei massimi esperti di OOA: **Grady Booch**, **Ivar Jacobson** e **Jim Rumbaugh** (*i tres amigos*), ed è inteso come sintesi dei molti metodi attualmente usati
- È stato accettato da molti altri esperti del settore, tra cui Coad, Yourdon e Odell, e da tutte le grandi compagnie dell'informatica (tra cui Compaq-Digital, Ericsson, Hewlett-Packard, IBM, Microsoft, Rational Software, ...)
- E' uno **standard dell'OMG** dal 1997
- Esistono potenti strumenti **CASE** per UML. Da un modello UML è possibile generare automaticamente lo "scheletro" del codice di un sistema (le strutture dati complete e i prototipi delle funzioni)
- Microsoft** ha adottato UML come linguaggio standard per la sua libreria di componenti

2

UML ...

- è un **linguaggio**, non un **metodo** (come quelli di Yourdon e DeMarco, o di Rumbaugh o Jacobson)
- definisce una notazione standard, basata su un **metamodello** integrato degli “oggetti” che compongono un sistema software
- non prescrive una sequenza di processo, cioè non dice “prima bisogna fare questa attività, poi quest’altra”
- quindi può essere (ed è) utilizzato da persone e gruppi che seguono metodi diversi (è “indipendente dai metodi ”)
- è un linguaggio non proprietario, standard; i suoi autori non hanno il copyright su UML
- la versione diventata standard OMG ha ricevuto i contributi di molti altri metodologi e delle più importanti società di software mondiali
- la sua **evoluzione** è a carico dell’OMG, e soggetta a procedure ben definite per ogni cambiamento. Versione attuale: 2.5 (2013)

3

Generalità

- UML fornisce i costrutti per le seguenti fasi dello sviluppo dei sistemi software:
 - ⇒ Analisi dei requisiti tramite i casi d’uso
 - ⇒ Analisi e progetto OO
 - ⇒ 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 proiezione
- A 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

4

Diagramma vs. modello

In UML c'è distinzione fra i concetti di modello e di diagramma:

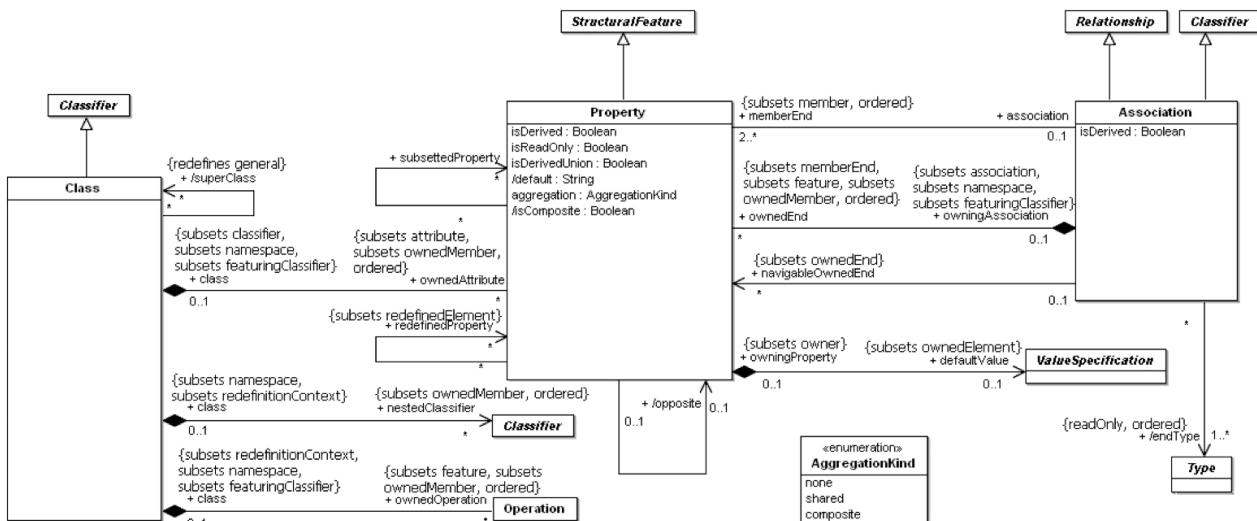
- Un **modello** contiene elementi di informazione circa il sistema sotto osservazione
- Un **diagramma** è una particolare visualizzazione di alcuni tipi di elementi di un modello

Un certo elemento può comparire in più diagrammi ma è univoca la sua definizione all'interno del modello

5

Il metamodello di UML

Una piccola porzione del metamodello UML 2 ...



6

2

La struttura di UML

□ La struttura di UML è composta da:

⇒ **costituenti fondamentali**: gli elementi di base

- **entità**
- **relazioni**
- **diagrammi**

⇒ **meccanismi comuni**: tecniche comuni per raggiungere specifici obiettivi

- specifiche
- ornamenti
- distinzioni comuni
- meccanismi di estendibilità

⇒ **architettura**: l'espressione dell'architettura del sistema

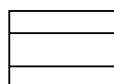
7

Entità

Sono gli elementi di modellazione

Strutture:

- Classe



- Interfaccia



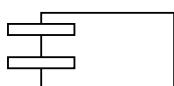
- Collaborazione



- Caso d'uso



- Componente

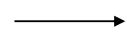


- Nodo

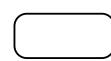


Comportamenti:

- Interazione

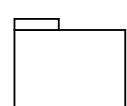


- Stato



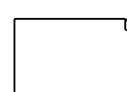
Raggruppamenti:

- Package



Informazioni:

- Annotazione

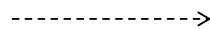


8

Relazioni

Legano tra loro le entità

- Dipendenza



- Associazione



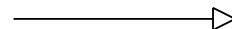
- Aggregazione



- Contenimento



- Generalizzazione



- Realizzazione



- Composizione



9

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

10

Diagrammi

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Sono descritte la struttura dati degli oggetti del sistema e le loro relazioni; è il diagramma più importante, da cui si può generare il codice

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

11

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

mostra un insieme di oggetti di interesse e le loro relazioni

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

12

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Dinamici:

- mostra i package e le loro relazioni di dipendenza, contenimento e specializzazione
- a dei casi d'uso
- a degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

13

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Dinamici:

- descrive l'architettura software del sistema
- a degli stati
- a di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

14

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- ✓ Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

describe la struttura del sistema hardware e l'allocazione dei vari moduli software

15

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

mostra la struttura interna di classificatori strutturati

16

Diagrammi

Statici:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

elenca i casi d'uso del sistema e le loro relazioni

UML

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

17

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma degli automi di Harel per descrivere gli stati degli oggetti di una classe
- Diagramma degli oggetti
- Diagramma dei package
- Diagramma dei componenti
- Diagramma di deployment
- Diagramma delle strutture composite

usa la notazione degli automi di Harel per descrivere gli stati degli oggetti di una classe

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
 - ✓ Diagramma di sequenza
 - ✓ Diagramma di comunicazione
 - ✓ Diagramma di sintesi dell'interazione
 - ✓ Diagramma dei tempi

18

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma dei componenti
- Diagramma dei package
- Diagramma delle strutture composite
- Diagramma di deployment
- Diagramma delle sequenze
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- ✓ Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- ✓ Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

19

Diagrammi

Sono viste sul modello UML

Statici:

- Diagramma delle classi
- Diagramma dei componenti
- Diagramma dei package
- Diagramma delle strutture composite
- Diagramma di deployment
- Diagramma delle sequenze
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- ✓ Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

Dinamici:

- Diagramma dei casi d'uso
- Diagramma degli stati
- Diagramma di attività
- Diagramma di interazione
- ✓ Diagramma di sequenza
- ✓ Diagramma di comunicazione
- ✓ Diagramma di sintesi dell'interazione
- ✓ Diagramma dei tempi

20

Specifiche

- Sono la descrizione testuale della semantica di un elemento



Caso d'uso: "APRI CONTO CORRENTE BANCARIO"

Scenario base:

- 1 il cliente si presenta in banca per aprire un nuovo c/c
- 2 l'addetto riceve il cliente e fornisce spiegazioni
- 3 se il cliente accetta fornisce i propri dati
- 4 l'addetto verifica se il cliente è censito in anagrafica
- 5 l'addetto crea il nuovo conto corrente
- 6 l'addetto segnala il numero di conto al cliente

Varianti:

- 3(a) se il cliente non accetta il caso d'uso termina
- 3(b) se il conto va intestato a più persone vanno forniti i dati di tutte
- 4(a) se il cliente (o uno dei diversi intestatari) non è censito l'addetto provvede a registrarlo, richiede al cliente la firma dello specimen e ne effettua la memorizzazione via scanner

21

Ornamenti

- Rendono visibili gli aspetti particolari della specifica dell'elemento



Finestra
{autore = Smith}

+dimensioni: Rettangolo=(100,100)
#visible: Booleano=falso
+dimensioniPredefinite: Rettangolo

+crea()
+nascondi()

22

Distinzioni comuni

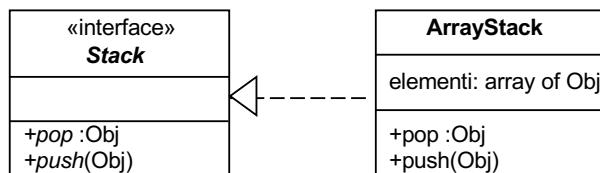
□ Classificatore/istanza

- ⇒ Separa la nozione astratta di un'entità dalle sue concrete istanze
- ⇒ Un'istanza ha di solito la stessa forma del classificatore, ma con il nome sottolineato



□ Interfaccia/implementazione

- ⇒ Separa “cosa” un oggetto fa da “come” lo fa
- ⇒ Un’interfaccia definisce un contratto che ciascuna sua implementazione garantisce di rispettare

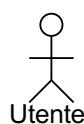


23

Meccanismi di estendibilità

- Uno **stereotipo** rappresenta una variazione di un elemento di modellazione esistente, con la stessa forma ma diverso scopo. Permette quindi di introdurre nuovi elementi di modellazione a partire da quelli esistenti
 - ⇒ predefiniti
 - ⇒ introdotti dall’utente

Attore è stereotipo
(con parentesi angolari <>>)



- Una **proprietà** è un valore associato a un elemento del modello, espresso da una stringa associata all’elemento

```
{ author = "Joe Smith", status = analysis } { abstract }
```
- Un **vincolo** è una frase di testo che definisce una condizione o una regola che riguarda un elemento del modello e deve risultare sempre vera

```
{ disjoint, complete } { subset }
```
- Un **profilo** è un insieme di stereotipi, valori etichettati (che definiscono proprietà) e vincoli, usato per personalizzare UML

24

Architettura

- **Vista dei casi d'uso**
 - ⇒ Descrive le funzionalità del sistema come vengono percepite dagli utenti, dagli analisti e dagli esecutori del testing. Non specifica l'organizzazione del software ma è la base per le altre viste
- **Vista logica**
 - ⇒ Stabilisce la terminologia del dominio del problema sotto forma di classi e oggetti, illustrando come essi implementano il comportamento richiesto
- **Vista dei processi**
 - ⇒ È una variante orientata ai processi della vista logica; modella i thread e i processi sotto forma di classi attive
- **Vista di implementazione**
 - ⇒ Descrive i moduli implementativi e le loro dipendenze, illustrandone la configurazione così da definire il concetto di versione del sistema
- **Vista di deployment**
 - ⇒ Mostra la distribuzione fisica del sistema software sull'architettura hardware

25

Viste/diagrammi (sistema complesso)

	casi d'uso	logica	dei processi	implementativa	di deployment
casi d'uso	X				
classi/oggetti		X	X		
componenti				X	
distribuzione					X
stato		X	X	X	X
attività	X	X	X	X	X
interazione	X	X	X	X	X

aspetti statici - aspetti dinamici

26

Viste/diagrammi (sistema medio)

	casi d'uso	logica	dei processi	implementativa	di deployment
casi d'uso	X				
classi/oggetti		X	X		
componenti				X	
distribuzione					X
stato		X			
attività	X				
interazione		X	X		

aspetti statici - aspetti dinamici

27

Viste/diagrammi (sistema piccolo)

	casi d'uso	logica	dei processi	implementativa	di deployment
casi d'uso	X				
classi/oggetti		X			
componenti				(X)	
distribuzione					(X)
stato		(X)			
attività					
interazione		X			

aspetti statici - aspetti dinamici

28

3

Diagrammi dei casi d'uso

- Rappresentano i **ruoli** di utilizzo del sistema da parte di uno o più **utilizzatori (attori)**:
 - ⇒ esseri umani (dipendenti, clienti)
 - ⇒ organizzazioni, enti, istituzioni
 - ⇒ altre applicazioni o sistemi (hardware e software), sottosistemi
- Descrivono l'**interazione** tra attori e sistema, non la logica interna della funzione né la struttura del sistema
- Sono espressi in forma **testuale**, comprensibile anche per i non "addetti ai lavori"
- Possono essere definiti a livelli diversi (l'intero sistema o parti del sistema), ma sempre dal punto di vista dell'utente

29

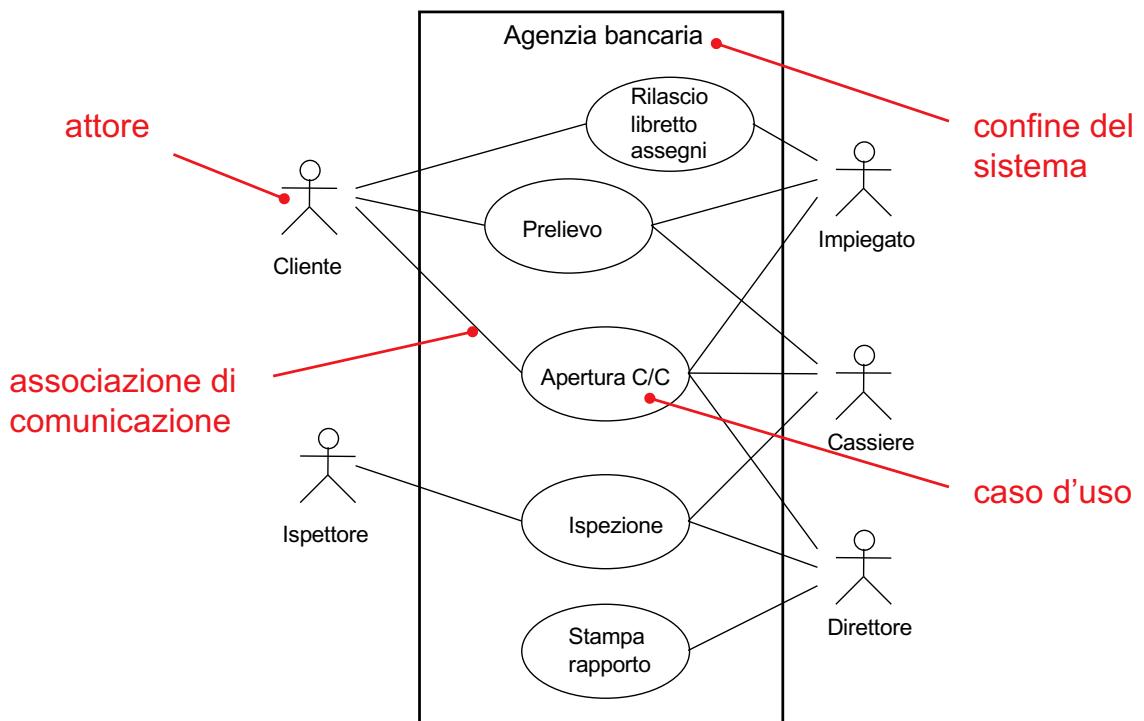
Attore vs. caso d'uso

- Un **attore** identifica il ruolo che un'entità esterna assume quando interagisce direttamente con il sistema
 - ⇒ ... è **sempre esterno al sistema**, anche se il sistema ne può mantenere una rappresentazione interna
 - ⇒ ... **spedisce o riceve messaggi dal sistema**, o scambia informazioni con esso
 - ⇒ ... esegue i casi d'uso
 - ⇒ ... è **modellato con una classe, non un oggetto**
- Un **caso d'uso** è la specifica di una sequenza di azioni che un sistema, un sottosistema o una classe può eseguire interagendo con attori esterni
 - ⇒ ... è una funzionalità come percepita da un attore
 - ⇒ ... **produce un risultato osservabile** utile all'attore
 - ⇒ ... **viene sempre attivato da un attore**
 - ⇒ ... è completo

Un attore è il ruolo esterno che compie un'azione (caso d'uso)

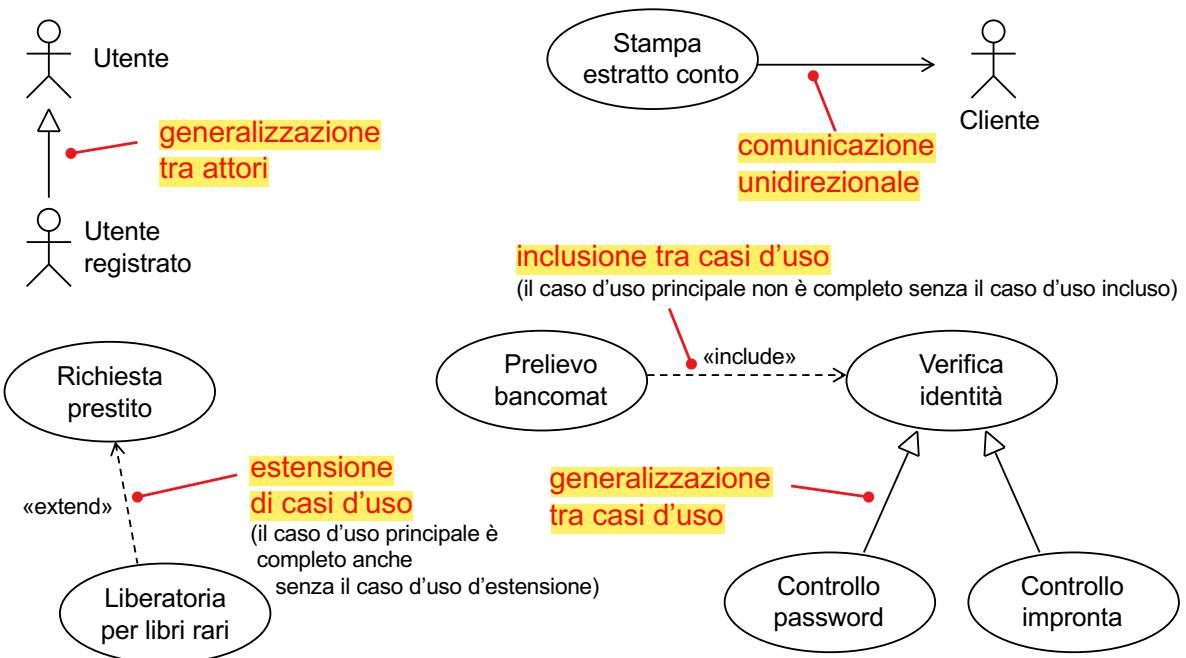
30

I casi d'uso di una banca



31

Relazioni nei diagrammi dei casi d'uso



32

Punti di vista



UTILIZZATORE

- Casi d'uso
 - ⇒ telefonare
 - ⇒ ricevere telefonate
 - ⇒ inviare messaggi
 - ⇒ memorizzare un numero
 - ⇒

PROGETTISTA

- Funzionalità interne
 - ⇒ trasmissione / ricezione
 - ⇒ alimentazione (batteria)
 - ⇒ I/O (display, tasti, ...)
 - ⇒ gestione rubrica
 - ⇒

33

Ruolo dei casi d'uso

- **Nelle fasi iniziali della progettazione servono per chiarire cosa dovrà fare il sistema**
 - ⇒ Ragionare sui casi d'uso con il committente è uno dei modi più efficaci ed efficienti per scoprire ed analizzare i requisiti ai quali il sistema dovrà fornire un'implementazione
 - ⇒ Dialogare su come il sistema verrà utilizzato, nella comunicazione con persone non esperte nella progettazione, è certamente più facile che non guardare a come dovrà essere costruito
 - ⇒ Raggiungere un accordo con il committente sulle modalità di utilizzo del sistema consente al progettista di affrontare con maggiore tranquillità il suo mestiere specifico di progettazione
- **I casi d'uso guidano l'intero progetto di sviluppo**
 - ⇒ Costituiscono il punto di partenza per la progettazione del sistema
 - ⇒ Sono il riferimento primario per la definizione, la progettazione, l'esecuzione dei test per la verifica di quanto prodotto
 - ⇒ Rappresentano delle naturali unità di rilascio, per i progetti che seguono un approccio incrementale alla pianificazione della realizzazione e dei rilasci

34

Identificare i casi d'uso

1. Individuare i confini del sistema
2. Identificare tutte le tipologie di utilizzatori del sistema (esseri umani o altri sistemi), che verranno modellati come attori
3. Per ogni tipologia di attore, rilevare in quale modo utilizzerà il sistema, partendo dagli obiettivi che egli deve raggiungere. A ogni modalità di utilizzo corrisponde un caso d'uso
4. Per ogni caso d'uso, descrivere lo scenario base (la sequenza di passi più semplice possibile che conduce al successo del caso d'uso, le risposte attese dal sistema), e le principali varianti a tale scenario. Così facendo, tipicamente, possono emergere necessità di interazione del sistema con altri soggetti (esseri umani o altri sistemi), che verranno rappresentati nel modello come attori aggiuntivi

35

Scenari

- Ogni specifica esecuzione (istanza) di un caso d'uso è detta scenario
 - ⇒ Ad esempio, in un caso d'uso "acquisto di un prodotto", ogni specifico acquisto effettuato da uno specifico cliente in uno specifico momento costituisce uno scenario particolare
- Esistono scenari di successo e scenari di fallimento
- Gli scenari possibili sono innumerevoli
- La prassi più diffusa per la descrizione degli scenari di un caso d'uso è quella di definire uno scenario base, cioè lo scenario più semplice possibile che porta al successo del caso d'uso
- Allo scenario base vengono quindi agganciate le varianti, che lo rendono più complesso e possono portare al successo o al fallimento del caso d'uso

36

Scenari

Caso d'uso: "APRI CONTO CORRENTE BANCARIO"

Scenario base:

- 1 il cliente si presenta in banca per aprire un nuovo c/c
- 2 l'addetto riceve il cliente e fornisce spiegazioni
- 3 se il cliente accetta fornisce i propri dati
- 4 l'addetto verifica se il cliente è censito in anagrafica
- 5 l'addetto crea il nuovo conto corrente
- 6 l'addetto segnala il numero di conto al cliente

Varianti:

- 3(a) se il cliente non accetta il caso d'uso termina
- 3(b) se il conto va intestato a più persone vanno forniti i dati di tutte
- 4(a) se il cliente (o uno dei diversi intestatari) non è censito l'addetto provvede a registrarlo, richiede al cliente la firma dello specimen e ne effettua la memorizzazione via scanner

37

Specifiche del caso d'uso

- UML non suggerisce il modo per specificare un caso d'uso, lasciando spazio libero a tutte le possibili forme di documentazione testuale
- La specifica del caso d'uso, comunque effettuata, ha un ruolo centrale nella comunicazione tra i diversi soggetti coinvolti nello sviluppo di un sistema, dal committente agli utilizzatori, dai progettisti agli specialisti di test
- Un caso d'uso può essere anche descritto da un **diagramma di attività o di sequenza**

38

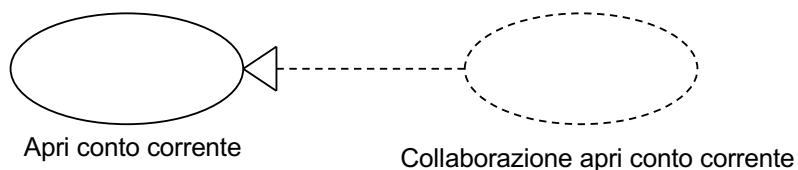
Specifiche del caso d'uso

Nome	
Identificatore	
Breve descrizione	<i>fissa l'obiettivo del caso d'uso</i>
Attori primari	<i>avviano il caso d'uso</i>
Attori secondari	<i>interagiscono con il caso d'uso dopo che è stato avviato</i>
Precondizioni	<i>condizioni che devono essere vere prima che il caso d'uso possa essere eseguito</i>
Sequenza principale degli eventi	<i>i passi che costituiscono il caso d'uso</i>
Postcondizioni	<i>condizioni che devono essere vere quando il caso d'uso termina</i>
Sequenze alternative degli eventi	<i>un elenco di alternative alla sequenza principale</i>

39

Realizzare i casi d'uso

- La realizzazione dei casi d'uso può essere espressa con una **collaborazione** costituita da classi che interagendo tra loro svolgono i passi specificati nel caso d'uso
- La collaborazione che realizza un caso d'uso può essere descritta:
 - ⇒ a livello **statico** mediante un diagramma delle classi che evidenzia le classi o gli oggetti coinvolti nella collaborazione
 - ⇒ a livello **dinamico** mediante un diagramma di interazione che evidenzia i messaggi che gli oggetti si scambiano nell'ambito della collaborazione



40

4

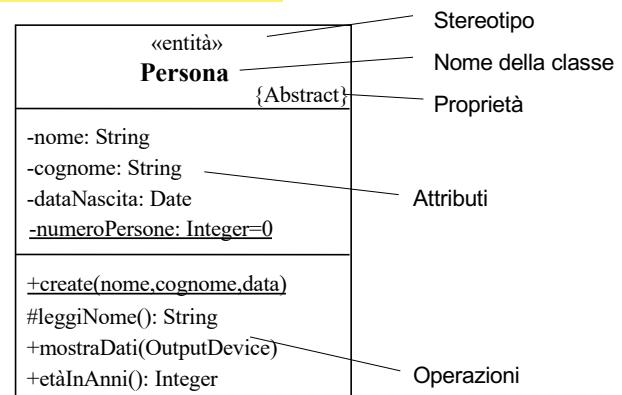
Diagrammi delle classi

- Sono il nucleo fondamentale di UML
- Descrivono la struttura statica del sistema in termini di classi e loro relazioni reciproche

⇒ Una **classe** descrive un gruppo di oggetti con proprietà, comportamento e relazioni comuni

⇒ Un **attributo** è un valore che caratterizza gli oggetti di una classe

⇒ Un' **operazione** è una trasformazione che può essere applicata a (o invocata da) gli oggetti di una classe. Ogni operazione ha come argomento implicito l'oggetto destinazione



41

Notazione

- Per gli **attributi** della classe:

visibilità nome molteplicità : tipo = valoreDefault

⇒ Visibilità

- pubblica +
- privata -
- protetta #
- package ~

⇒ Molteplicità

- per esempio: String [5], Real [2..*], Boolean [0..1]

⇒ Tipo

- Integer, UnlimitedNatural, Real
- Boolean
- String

⇒ Ambito

- istanza
- classe

42

Notazione

□ Per le **operazioni** della classe:

visibilità nome (parametro, ...): tipoRestituito

signature

⇒ Parametri

direzione nomeParametro: tipoParametro=valoreDefault

⇒ Direzione

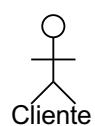
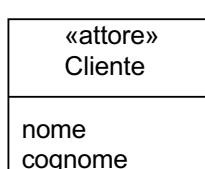
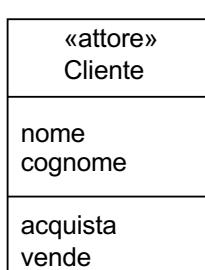
- in
- out
- inout
- return (si usa quando l'operazione restituisce più valori)

⇒ Ambito

- istanza
- classe

43

Diversi livelli di astrazione



44

Le relazioni tra classi

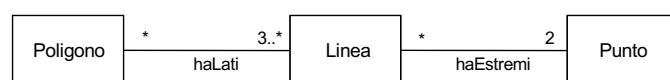
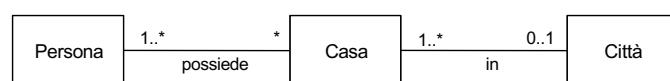
- Generalizzazione →
- Associazione ——————
- Dipendenza -----→
- Aggregazione ——————◊
- Composizione ——————◆
- Raffinamento -----→

45

Associazione

- E' una connessione tra classi, tipicamente bidirezionale
- Molteplicità:

- ⇒ Esattamente 1
 - ⇒ Opzionale 1
 - ⇒ Da x a y inclusi
 - ⇒ Solo i valori a,b,c
 - ⇒ 1 o più
 - ⇒ 0 o più
- | |
|-------|
| 1 |
| 0..1 |
| x..y |
| a,b,c |
| 1..* |
| * |

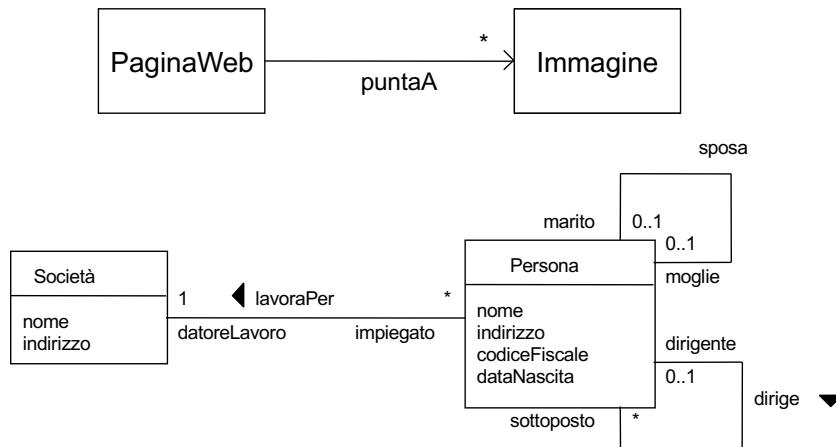


46

Associazione

- E' possibile indicare il **verso di lettura** di una associazione, definire associazioni **monodirezionali**, specificare **ruoli**

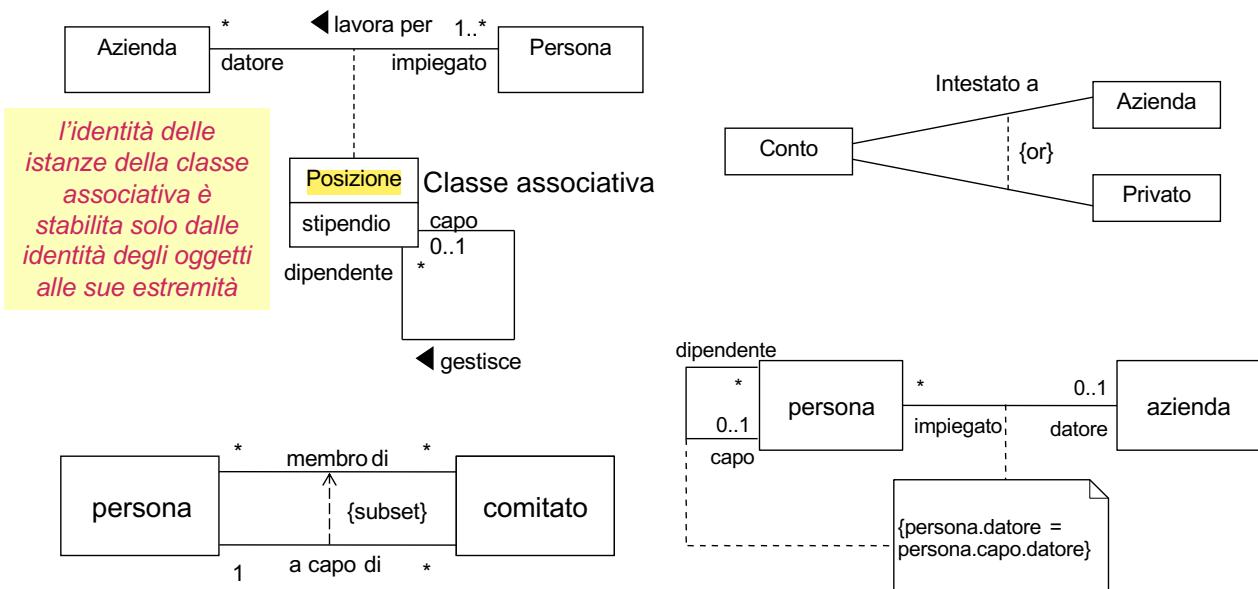
Bisogna leggere la classe poi il verbo e poi la cardinalità vicino alla classe destinazione



47

Associazione

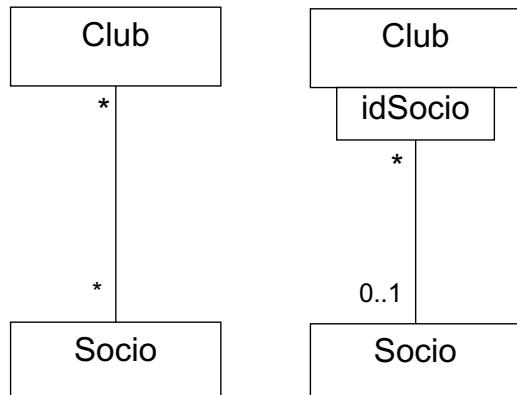
- E' possibile specificare **vincoli** e **classi associative**



48

Associazione

- Le **associazioni qualificate** riducono un'associazione molti-a-molti a una del tipo uno-a-uno, specificando un attributo che permette di selezionare un unico oggetto destinazione svolgendo il ruolo di identificatore o chiave di ricerca

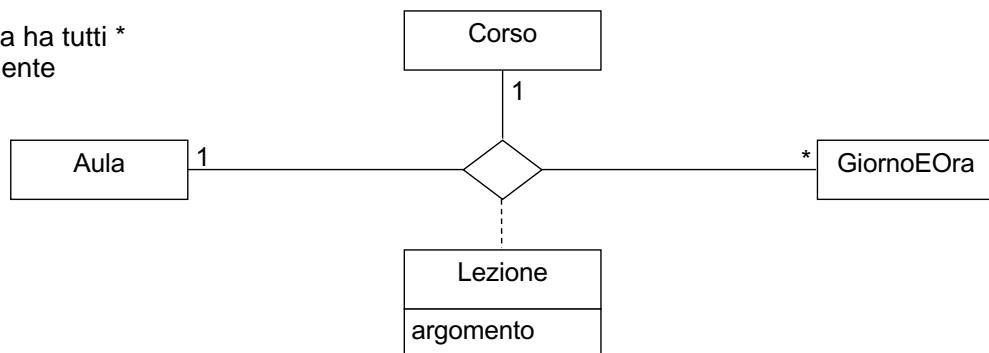


49

Associazione

- E' possibile definire **associazioni n-arie** (cioè tra n classi)
 - Ogni istanza dell'associazione è una tupla formata da n oggetti delle rispettive classi
 - La molteplicità di un ruolo rappresenta il numero di istanze dell'associazione quando sono stati fissati n-1 oggetti
 - I numeri di istanze dell'associazione quando è fissato un solo oggetto sono implicitamente assunti essere tutti a "molti"

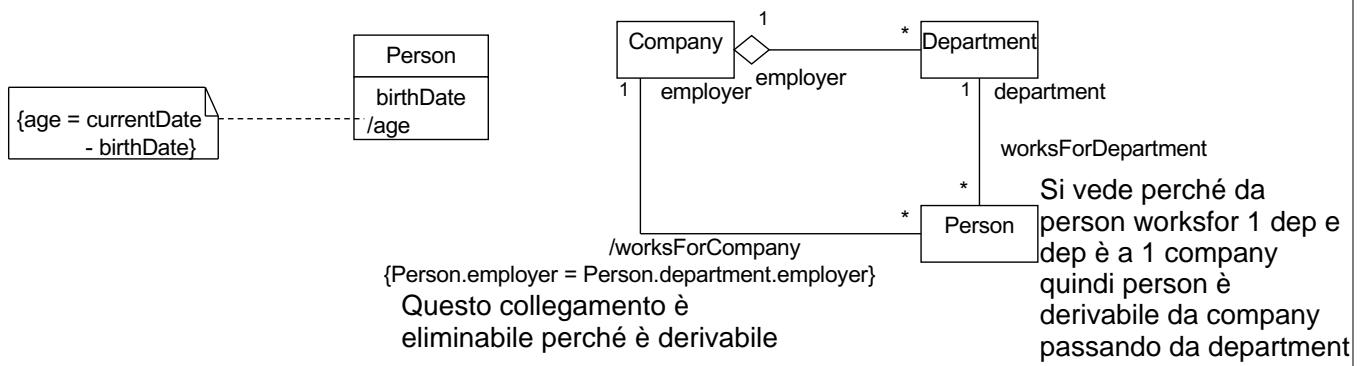
Una vera ternaria ha tutti *
ma capita raramente



50

Elementi derivati

- Un **elemento derivato** può essere calcolato a partire da un altro ma viene mostrato, per motivi di chiarezza o per scelte di progettazione, nonostante non aggiunga alcuna ulteriore informazione semantica
 - ⇒ viene indicato posizionando uno slash prima del suo nome
 - ⇒ i dettagli su come calcolarlo possono essere inseriti in una nota o essere rappresentati con una stringa di vincoli

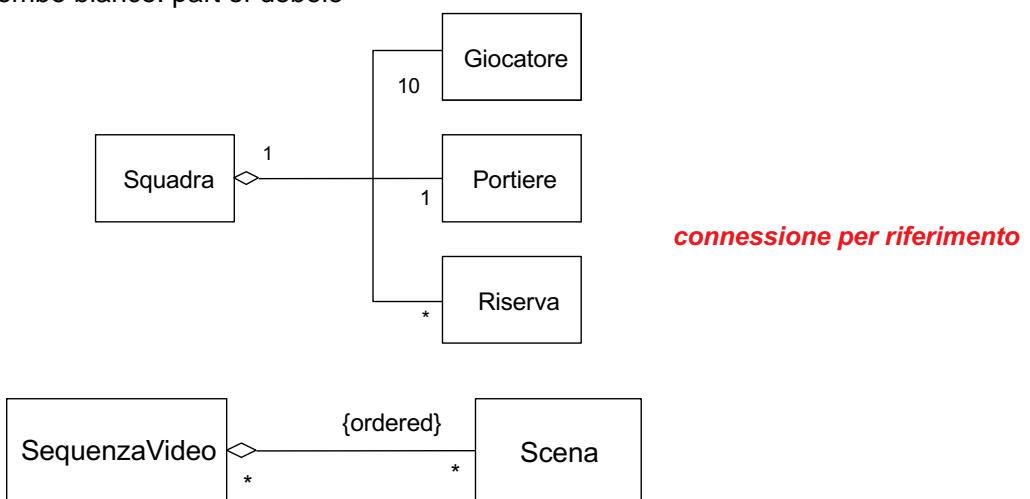


51

Aggregazione

- E' un caso speciale di associazione con semantica *part-of*
 - ⇒ Sia il tutto che le parti esistono indipendentemente

Aggregazione = Rombo bianco: part of debole

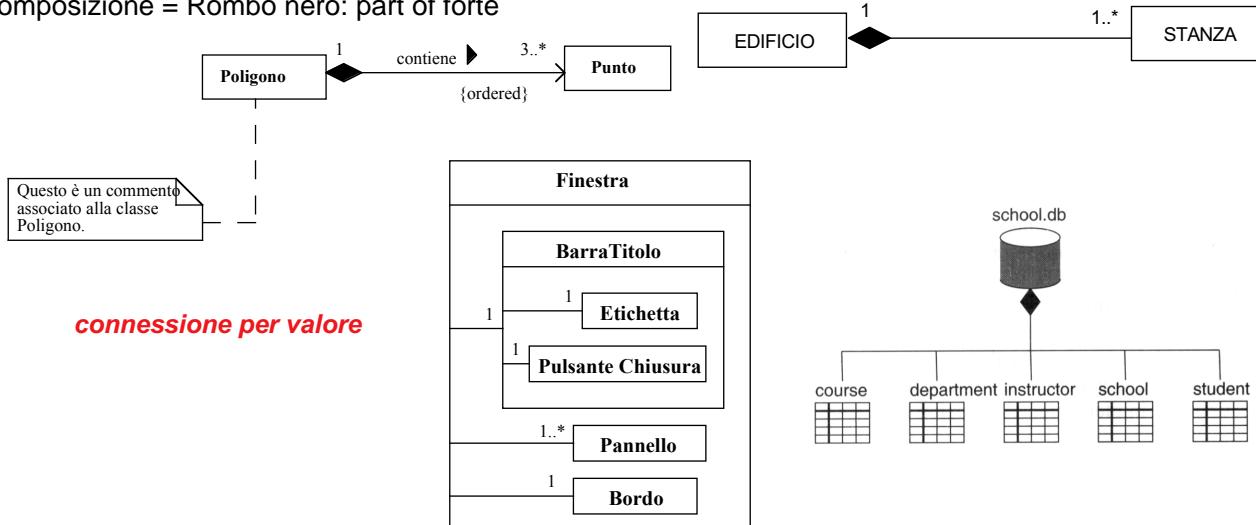


52

Composizione

- E' un'aggregazione in cui il tutto "possiede" le sue parti
 - ⇒ Le parti esistono solo in relazione al tutto
 - ⇒ Ogni parte appartiene a esattamente un tutto

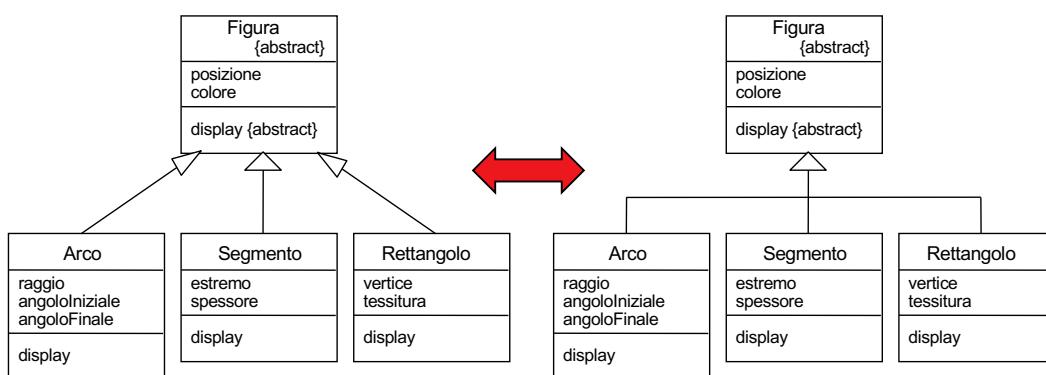
Composizione = Rombo nero: part of forte



53

Generalizzazione

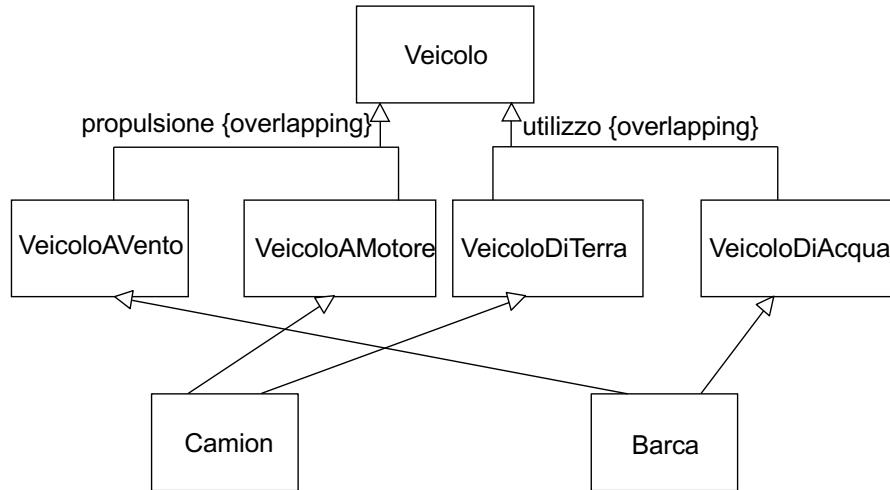
- Tutti gli attributi, le operazioni e le relazioni della superclasse vengono ereditati dalle sottoclassi



54

Generalizzazione

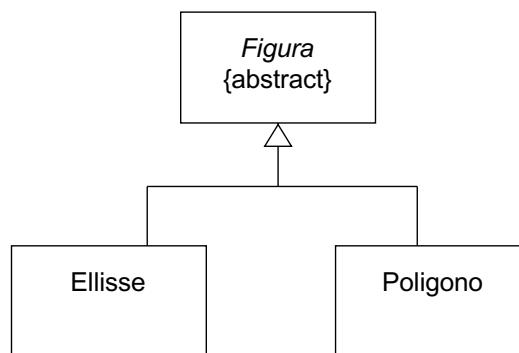
- E' supportata l'**ereditarietà multipla**
- Possono essere indicati **insiemi di generalizzazione** e vincoli (*overlapping, disjoint, complete, incomplete*)



55

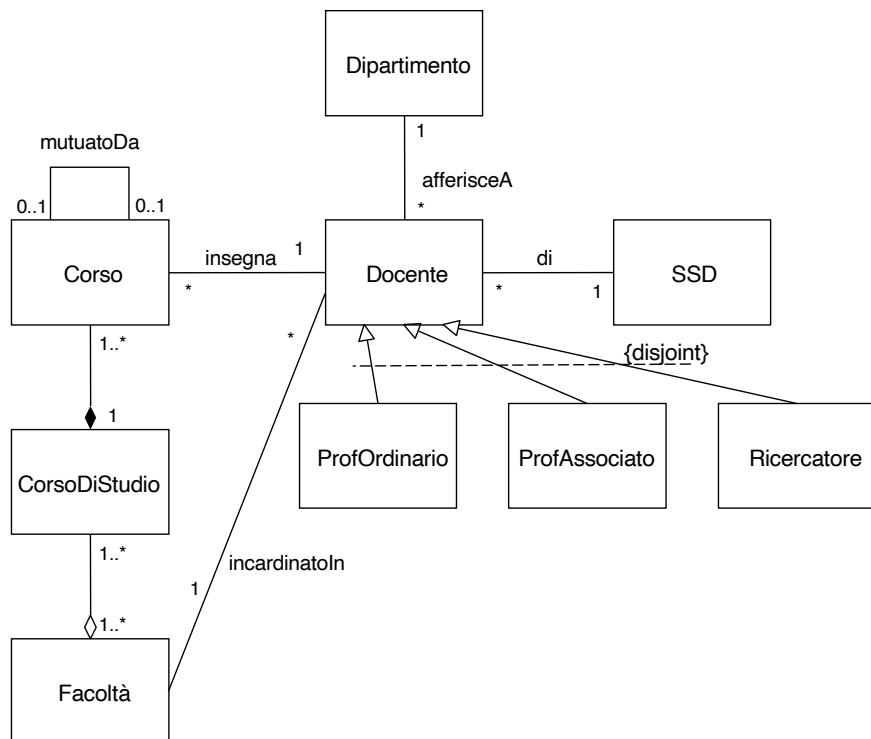
Classi astratte

- Sono **classi che non possono essere istanziate da oggetti**
- Sono utili come radici di **gerarchie di specializzazione**



56

Un esempio

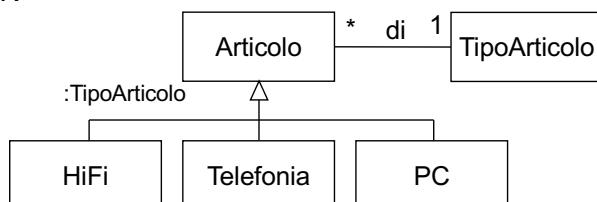


57

Powertyping

- Un **powertype** è una (meta)classe le cui istanze sono classi che specializzano un'altra classe

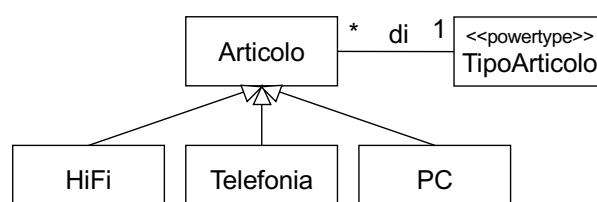
La classe "Categoria" in ER



La soluzione con il tipo invece permette di specificare un attributo comune a tutti i tipi (sconto applicato ad un certo tipo)

in UML 2

La specializzazione permette di specificare altri attributi unici per loro (prezzo individuale di un articolo di un certo tipo)



in UML 1.4

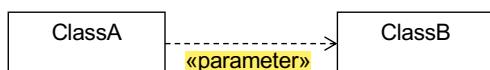
58

Dipendenza

- In generale, **A dipende da B quando una variazione in B può comportare una variazione in A**
- Nel caso delle classi, una dipendenza indica che una classe cliente dipende da alcuni servizi di una classe fornitrice, ma non ha una struttura interna che dipende da quest'ultima
 - ⇒ Lo stereotipo più comunemente usato è «use»



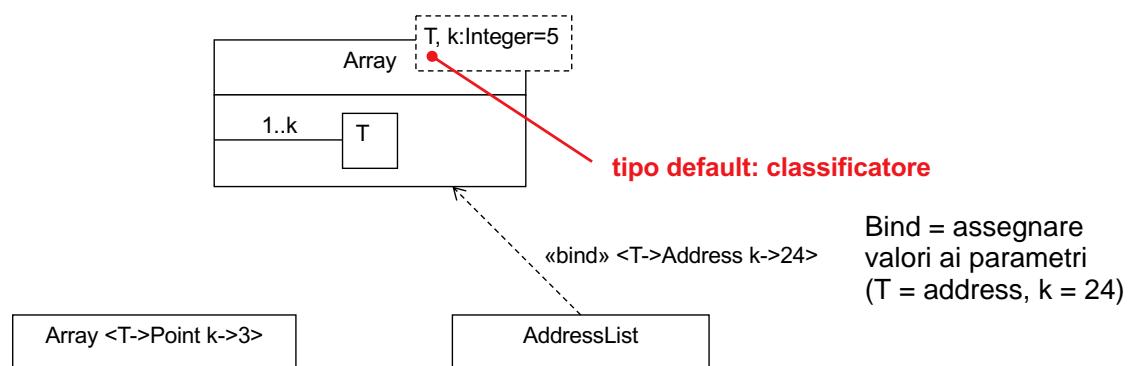
- ⇒ Più specificamente, si può rappresentare il fatto che **un'operazione della classe cliente ha argomenti che appartengono al tipo di un'altra classe**



59

Template

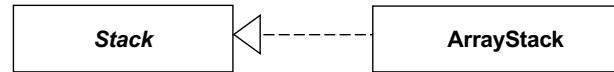
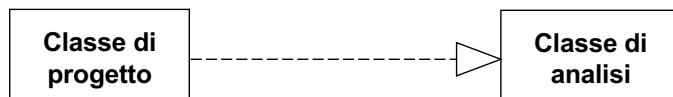
- Un **template** (o **classe parametrizzata**) è utilizzato per descrivere **una classe in cui uno o più parametri formali non sono istanziati**
 - ⇒ Un template definisce una famiglia di classi in cui ogni classe è specificata istanziando i parametri con i valori attuali
 - ⇒ Un template non è utilizzabile direttamente
- Un **bound element** è una classe che istanzia i parametri di un template, e può essere utilizzato esattamente come una classe



60

Raffinamento

- Esprime una relazione tra due descrizioni dello stesso concetto a diversi livelli di astrazione
 - ⇒ Tra un tipo astratto e una classe che lo realizza (*realizzazione*)
 - ⇒ Tra una classe di analisi e una di progetto
 - ⇒ Tra una implementazione semplice e una complessa della stessa cosa



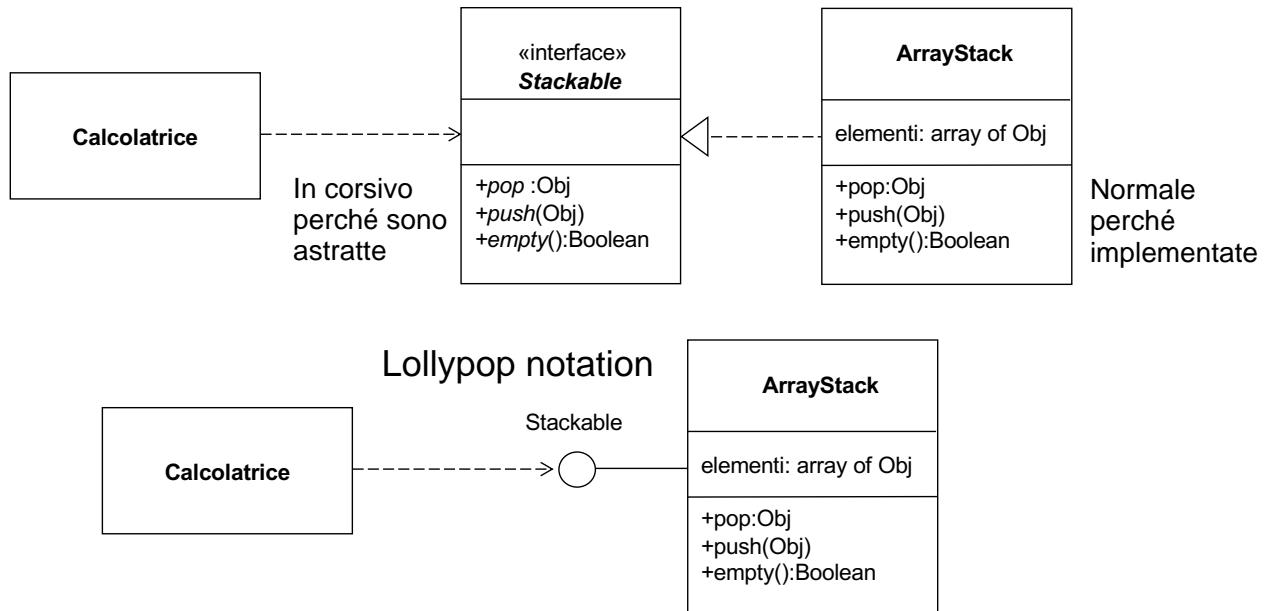
61

Interfaccia

- Una **interfaccia** è un insieme di funzionalità pubbliche identificate da un nome
- Specifica le operazioni pubbliche di una classe, di un componente, di un pacchetto o di altre entità, separandone le specifiche dall'implementazione
- Un'interfaccia non ha alcuna specifica di struttura interna (attributi, stato o associazioni); è una classe astratta, senza attributi né associazioni e con solo operazioni astratte (senza implementazione)
 - ⇒ La notazione estesa prevede una rappresentazione simile a quella delle classi, con «interface» come stereotipo e senza comportamento per gli attributi; la notazione minimizzata prevede un piccolo cerchio collegato all'entità (classe, componente o package) che la supporta, col nome dell'interfaccia vicino (*lollipop notation*)
 - ⇒ Un'altra classe che usa l'interfaccia può essere collegata ad essa da una freccia di dipendenza, eventualmente con lo stereotipo «use»

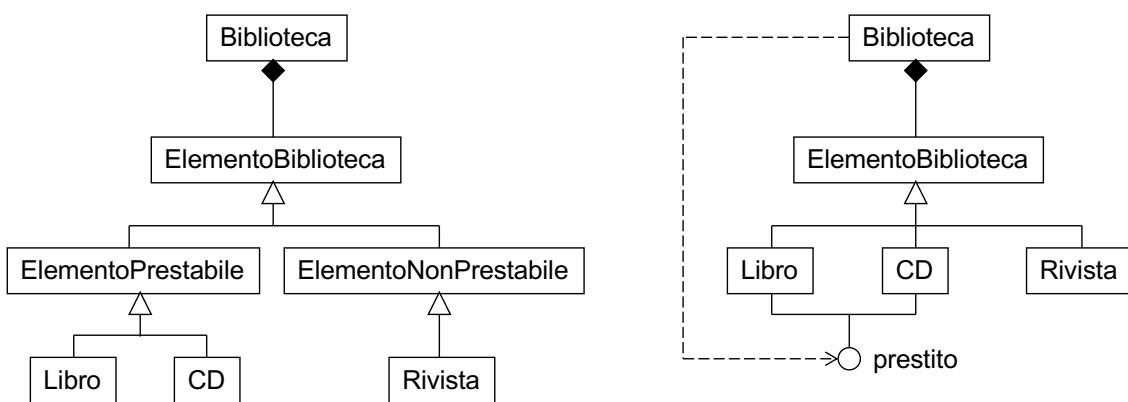
62

Interfaccia



63

Interfaccia vs. ereditarietà



64

Analisi vs. progettazione

□ Classi di analisi

- ⇒ rappresentano un'astrazione nel dominio del problema
- ⇒ corrispondono chiaramente a concetti concreti del mondo del business
- ⇒ escludono tutti i dettagli implementativi
- ⇒ hanno un insieme ridotto, coeso e ben definito di responsabilità
- ⇒ indicano gli attributi che saranno *probabilmente* inclusi nelle classi di progettazione
- ⇒ le loro operazioni specificano i principali servizi offerti dalla classe

□ Classi di progettazione

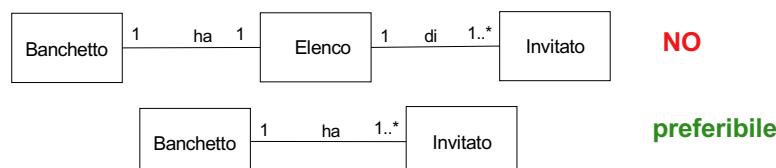
- ⇒ le loro specifiche sono complete per cui possono essere direttamente implementate
- ⇒ nascono dal dominio del problema per raffinamento delle classi di analisi, oppure dal dominio della soluzione

65

Identificare le classi d'analisi

□ Le classi corrispondono a entità fisiche e a concetti del dominio applicativo

- ⇒ Evitare di rappresentare soluzioni implementative



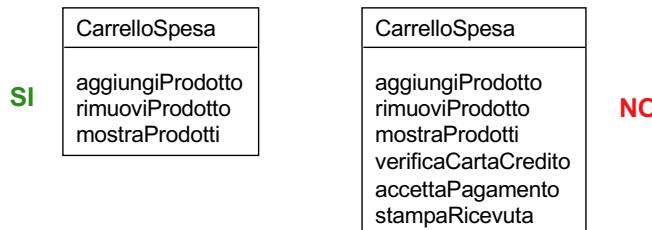
- ⇒ Evitare le classi ridondanti, irrilevanti, vaghe
- ⇒ Evitare le classi “onnipotenti”



66

Identificare le classi d'analisi

- ⇒ Una classe è associata a un **piccolo e ben definito insieme di responsabilità** (normalmente tra 3 e 5)

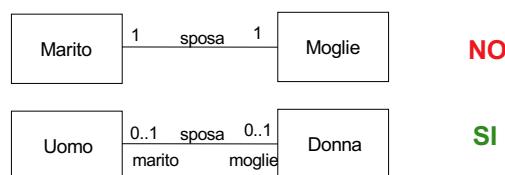


- ⇒ Nessuna classe può essere isolata
- ⇒ **Evitare di avere poche classi troppo complesse, ma anche tante classi troppo semplici**

67

Identificare le classi d'analisi

- ⇒ I nomi delle classi devono riflettere la loro natura intrinseca e non il ruolo giocato nelle associazioni

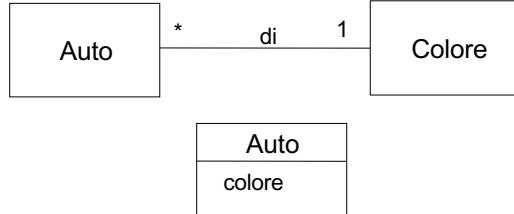


- ⇒ Evitare le gerarchie di specializzazione profonde

68

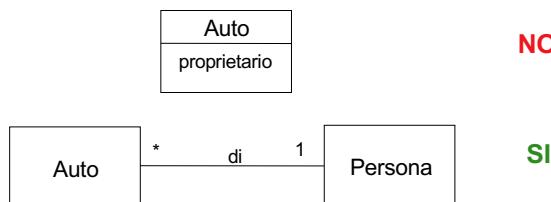
Identificare le classi d'analisi

- ⇒ I nomi che descrivono oggetti dovrebbero essere espressi come attributi



Questo funziona se ho un set di colori predefiniti o se Colore ha dei suoi attributi

- ⇒ Se una proprietà esiste indipendentemente, o compare più volte all'interno del diagramma, dovrebbe essere espressa come classe

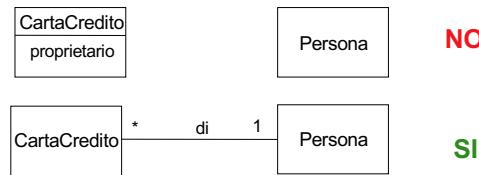


69

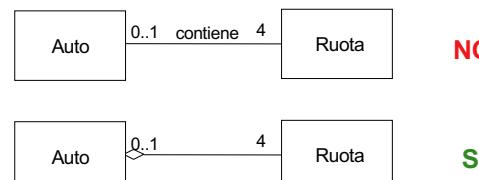
Identificare le associazioni d'analisi

- Le associazioni sono tipicamente indicate da verbi che esprimono collocazione fisica (*contenuto in*), azioni (*gestisce*), comunicazioni (*parla a*), proprietà (*possiede*), soddisfacimento di condizioni (*sposato a*)

- ⇒ Ogni riferimento da una classe a un'altra è un'associazione



- ⇒ Un'aggregazione è un'associazione con semantica *part-of*

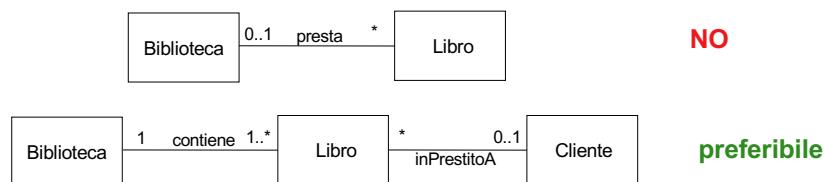


- ⇒ Evitare le associazioni irrilevanti o che esprimono soluzioni implementative

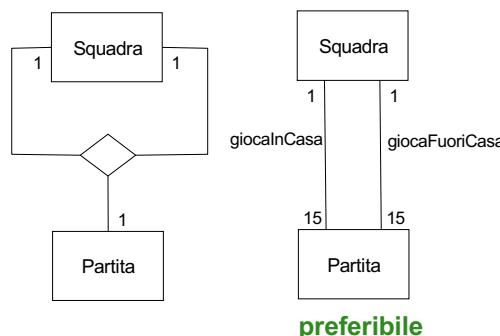
70

Identificare le associazioni d'analisi

- ➡ Un'associazione deve descrivere una proprietà strutturale del dominio, non un evento transitorio



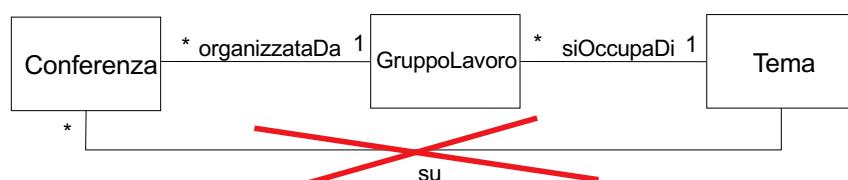
- ➡ Molte associazioni ternarie possono essere scomposte in due associazioni binarie



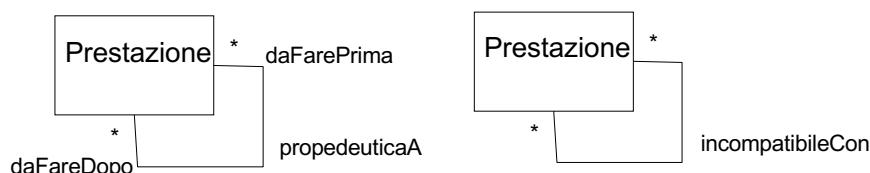
71

Identificare le associazioni d'analisi

- ➡ Evidenziare le associazioni derivate, che cioè possono essere espresse in termini di altre associazioni



- ➡ Quando appropriato, specificare i ruoli



72

Identificare gli attributi

- Le **proprietà** di classi e associazioni sono attributi
- Gli attributi spesso corrispondono a nomi seguiti da possessivi (ad esempio, *il colore della macchina*)
 - ⇒ Omettere o evidenziare gli attributi derivati

Persona
dataNascita
età

NO

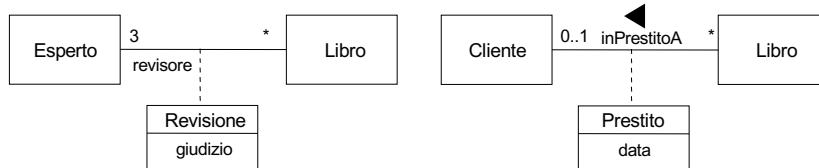
Persona
dataNascita

SI

Persona
dataNascita
/ età

SI

- ⇒ Se una proprietà dipende dalla presenza di un'associazione, rappresentarla con un attributo dell'associazione



73

Identificare gli attributi

- ⇒ Non aggiungere agli attributi gli identificatori degli oggetti, a meno che non risultino esplicitamente dalle specifiche

Prodotto
idProgressivo
nome

NO

Prodotto
nome

SI

Prodotto
codProdotto
nome

SI

- ⇒ Quando gli attributi di una classe possono essere raggruppati in due o più insiemi, probabilmente la classe dovrebbe essere suddivisa in due o più classi

Persona
nome
cognome
dataNascita
indirizzo
redditoLordo
redditoNetto
aliquotaMax

PersonaAnagrafica
nome
cognome
dataNascita
indirizzo

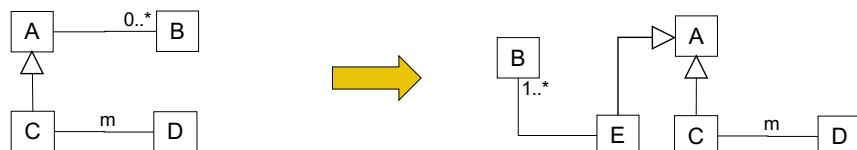
PersonaFiscale
redditoLordo
redditoNetto
aliquotaMax

preferibile

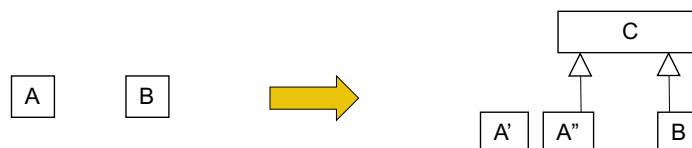
74

Raffinamenti

- La possibilità di raffinare il modello deriva dalla natura iterativa dell'approccio a oggetti
- I raffinamenti tramite ereditarietà possono avvenire top-down (definizione di specializzazioni di classi esistenti) o bottom-up (generalizzazione di due o più classi con caratteristiche comuni)
 - ↳ Valutare l'utilità di aggiungere nuove classi in caso di asimmetrie in associazioni o generalizzazioni



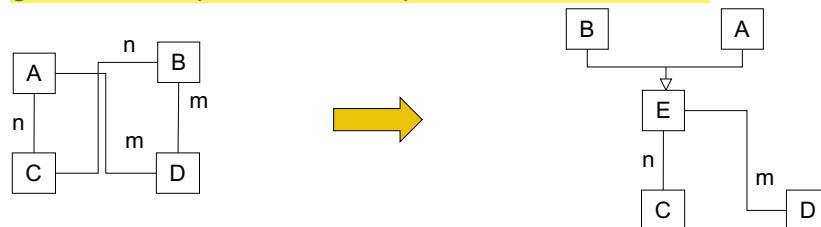
- ↳ In caso di difficoltà nel generalizzare, forse una classe sta giocando due ruoli differenti: può convenire spezzarla in due classi



75

Raffinamenti

- ↳ Se esistono più associazioni con lo stesso nome e scopo, conviene generalizzare per creare la superclasse che le unisce



- ↳ Se un ruolo incide sostanzialmente sulla semantica della classe, può convenire trasformarlo in una nuova classe



- ↳ Una classe senza attributi, né operazioni, né associazioni può essere eliminata
- ↳ Se nessuna operazione usa un'associazione, forse quella associazione è inutile

76

Identificare le classi di progettazione

- Con le classi di progettazione si specifica esattamente **come le classi assolveranno le loro responsabilità**
- Ciascuna classe deve essere:
 - ⇒ **completa**, ossia fornire ai suoi clienti tutti i servizi che essi si aspettano
 - ⇒ **sufficiente**, ossia i suoi metodi devono essere esclusivamente finalizzati allo scopo della classe
 - ⇒ **essenziale**, ossia non mettere a disposizione più di un modo per effettuare la stessa operazione
 - ⇒ **massimamente coesa**, ossia modellare un unico concetto astratto
 - ⇒ **minimamente interdipendente**, ossia essere associata all'insieme minimo di classi che consente di realizzare le proprie responsabilità

77

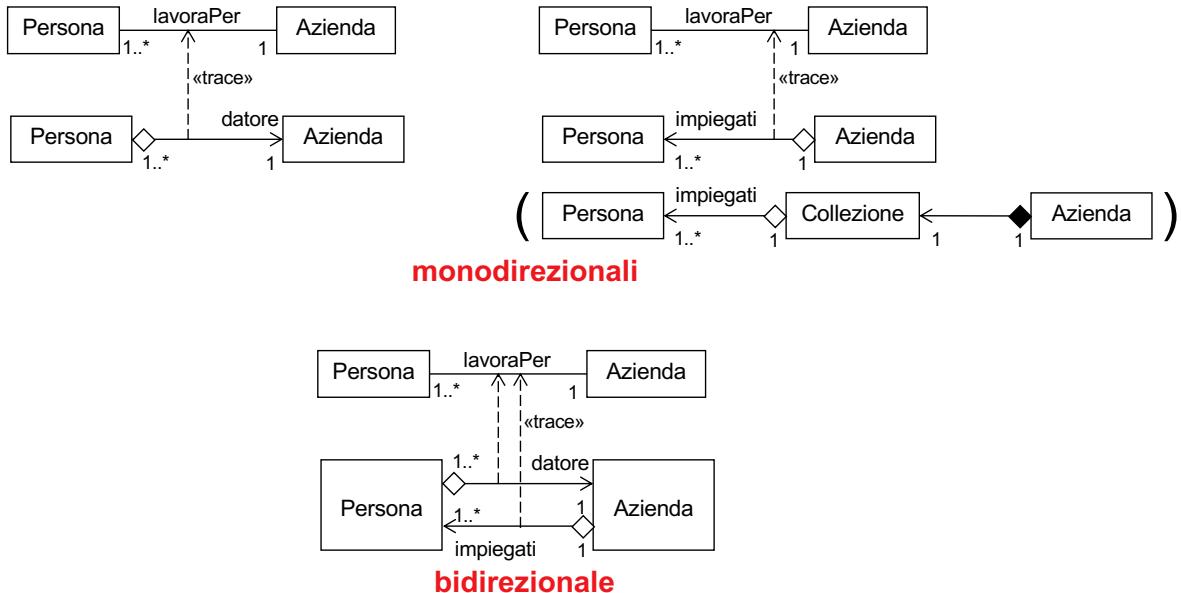
Identificare le associazioni di progettazione

- Costrutti come le associazioni bidirezionali o le classi associative non sono direttamente implementabili
- Le associazioni di progettazione si ottengono da quelle di analisi attraverso una trasformazione basata principalmente sul carico di lavoro cui ciascuna associazione è sottoposta
- Le **associazioni di progettazione devono specificare:**
 - ⇒ il **nome**
 - ⇒ il **verso di navigabilità**
 - ⇒ la **molteplicità a entrambi gli estremi**
 - ⇒ il **nome del ruolo destinazione**

78

Identificare le associazioni di progettazione

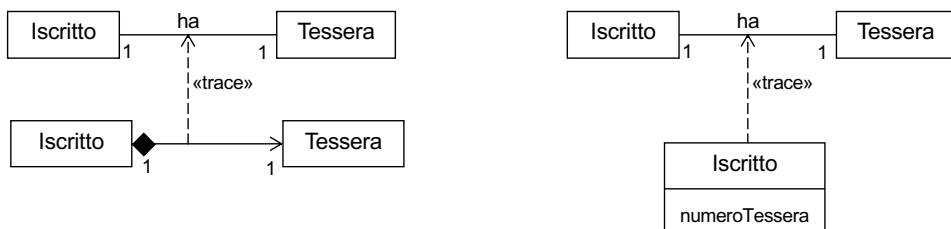
□ Associazioni multi-a-uno o multi-a-molti



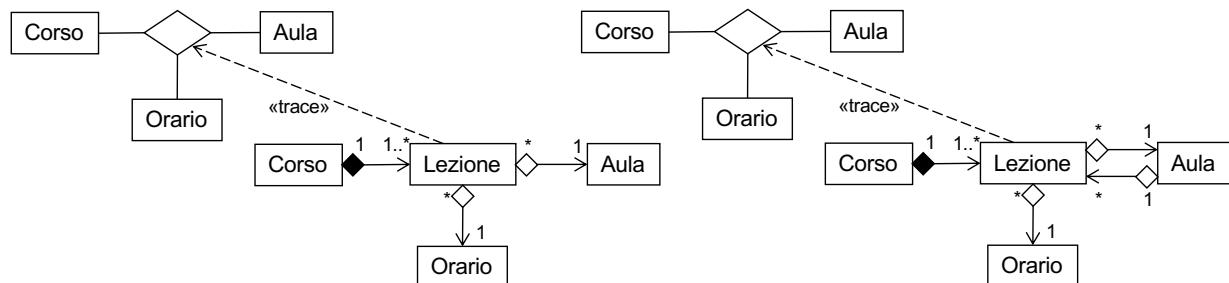
79

Identificare le associazioni di progettazione

□ Associazioni uno-a-uno



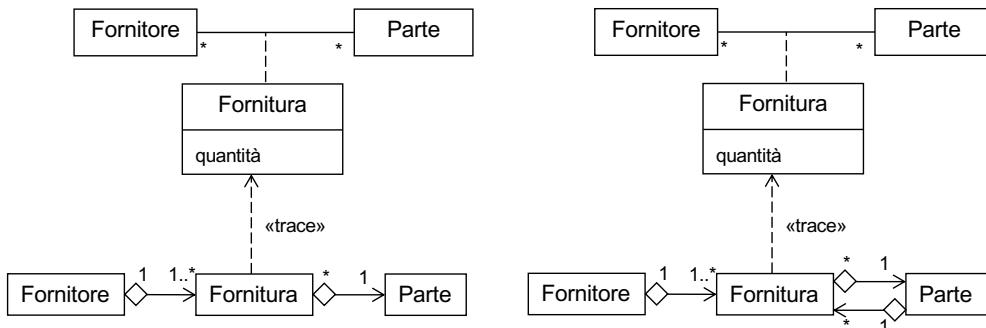
□ Associazioni ternarie



80

Identificare le associazioni di progettazione

□ Classi associative



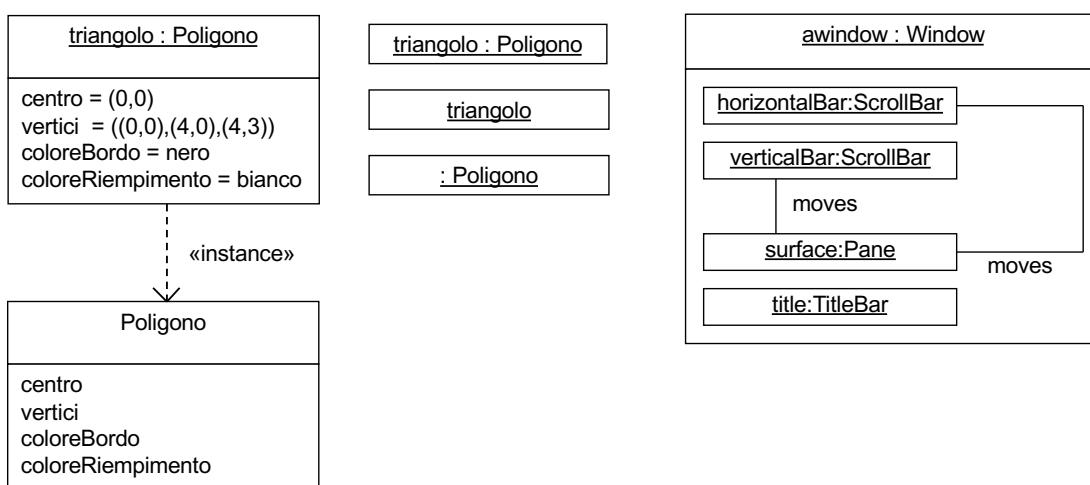
81

5

Diagrammi degli oggetti

- Un **oggetto rappresenta** una particolare **istanza** di una classe.
- Un **oggetto composto** è un oggetto di alto livello che contiene altri oggetti.

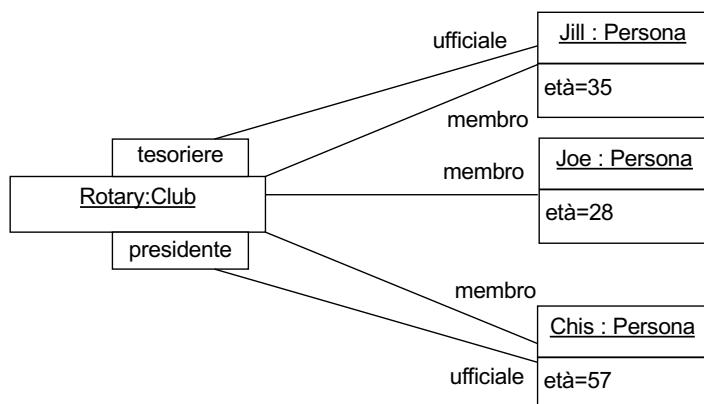
Un oggetto viene sottolineato



82

Diagrammi degli oggetti

- E' un grafo di istanze di elementi, e rappresenta un'istanza di un diagramma delle classi
 - ⇒ Il suo utilizzo è limitato principalmente a mostrare esempi di strutture dati
 - ⇒ Poiché un diagramma delle classi può contenere anche istanze, un diagramma degli oggetti può essere considerato come un caso particolare di diagramma delle classi in cui compaiono solo oggetti

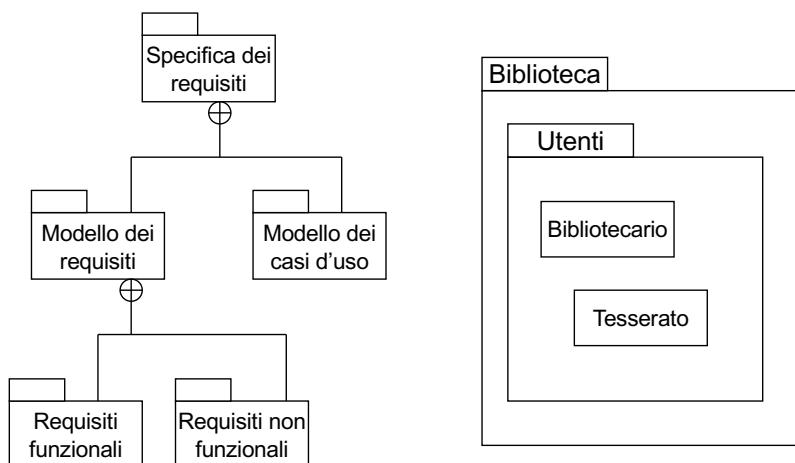


83

6

Diagrammi dei package

- Un package è un raggruppamento di elementi del modello semanticamente correlati
- Relazioni tra package:
 - ⇒ Si possono rappresentare relazioni di contenimento; si consiglia di mostrare massimo due livelli. I package annidati vedono lo spazio dei nomi dei package che li contengono, il contrario non è vero



84

Diagrammi dei package

⇒ Esistono quattro tipi principali di dipendenza tra package

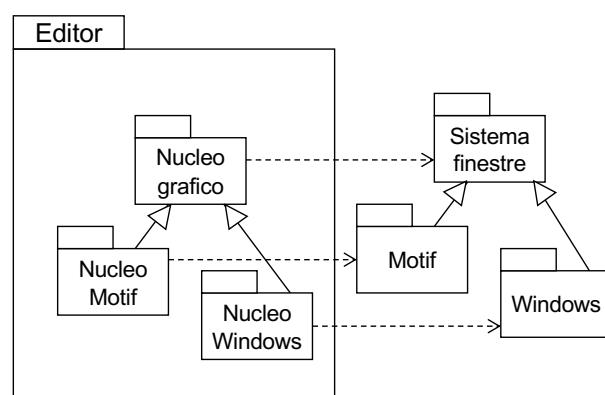
- «use» (default), quando un elemento del package cliente usa in qualche modo un elemento del package fornitore
- «import», quando gli elementi pubblici dello spazio dei nomi del package fornitore vengono aggiunti come elementi pubblici allo spazio dei nomi del package cliente
- «access», quando gli elementi privati dello spazio dei nomi del package fornitore vengono aggiunti come elementi privati allo spazio dei nomi del package cliente
- «trace» rappresenta l'evoluzione di un elemento in un altro elemento più dettagliato



85

Diagrammi dei package

⇒ Esiste una generalizzazione tra due package quando il package specifico si deve conformare all'interfaccia del package generale



86

Individuare i package d'analisi

- I package d'analisi sono gruppi di elementi del modello accomunati da forti correlazioni semantiche
- La fonte migliore per individuarli è il **diagramma delle classi**. I **migliori candidati per essere raggruppati nello stesso package sono:**
 - ⇒ le classi appartenenti a **gerarchie di composizione**
 - ⇒ le classi appartenenti a **gerarchie di specializzazione**
- Anche il **diagramma dei casi d'uso** può servire: uno o più casi d'uso che supportano un processo aziendale o un attore potrebbero indicare un package
- Per minimizzare le interdipendenze si possono poi spostare classi tra package, aggiungere package, eliminare package
- Numero ideali di classi per package: tra 4 e 10
- Conviene evitare la dipendenze circolari

87

7

Diagrammi di interazione

- Rappresentano la struttura dell'interazione tra oggetti durante uno scenario
- Esistono **quattro tipi di diagrammi di interazione**, ognuno rivolto a un particolare aspetto:
 - ⇒ **Diagramma di sequenza**: enfatizza la sequenza temporale degli scambi di messaggi
 - ⇒ **Diagramma di comunicazione**: enfatizza le relazioni strutturali tra gli oggetti che interagiscono
 - ⇒ **Diagramma di sintesi dell'interazione**: illustra come un comportamento complesso viene realizzato da un insieme di interazioni più semplici
 - ⇒ **Diagramma di temporizzazione**: enfatizza gli aspetti real-time di un'interazione

88

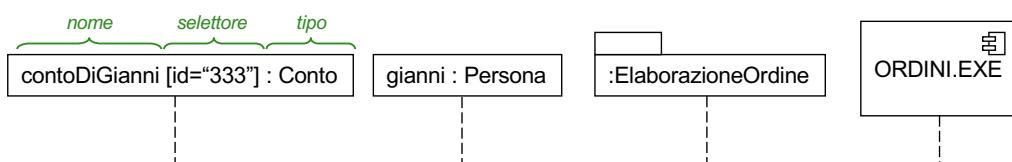
Terminologia

- Una **interazione** è un'unità di comportamento di un classificatore che ne costituisce il contesto; essa comprende un insieme di messaggi scambiati tra linee di vita all'interno del contesto per ottenere un obiettivo
- Il **contesto** può essere dato dall'intero sistema, da un sottosistema, da un caso d'uso, da un'operazione, da una classe
- Una **linea di vita** rappresenta come un'istanza di un classificatore partecipa all'interazione
- Un **messaggio** rappresenta un tipo specifico di comunicazione istantanea tra due linee di vita in un'interazione, e trasporta informazione nella prospettiva che seguirà una attività

89

Linee di vita

- Sintassi:



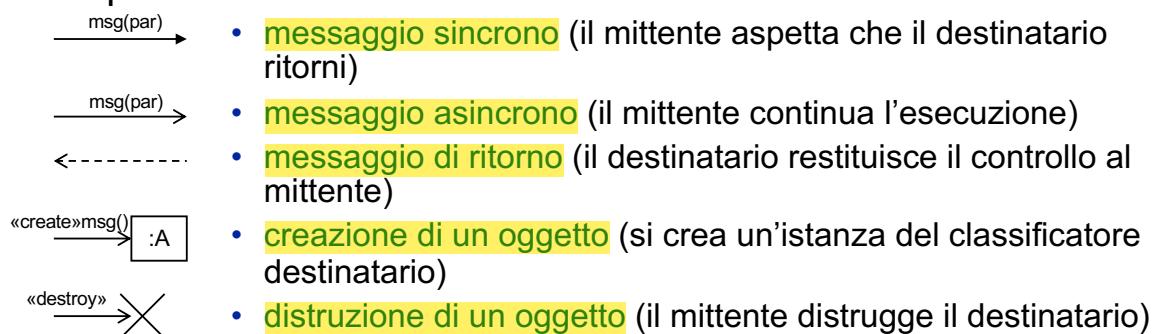
- Sono disegnate con lo stesso simbolo del loro classificatore
- Possono avere una “coda” a forma di riga verticale tratteggiata
- Non rappresentano specifiche istanze del classificatore, ma *modi* in cui le istanze partecipano all'interazione

90

Messaggi

- ⇒ messaggi di chiamata
- ⇒ messaggi di creazione
- ⇒ messaggi di distruzione
- ⇒ invio di segnali

- Per ogni messaggio di chiamata ricevuto da una linea di vita, deve esistere un'operazione corrispondente nel classificatore di quella linea di vita



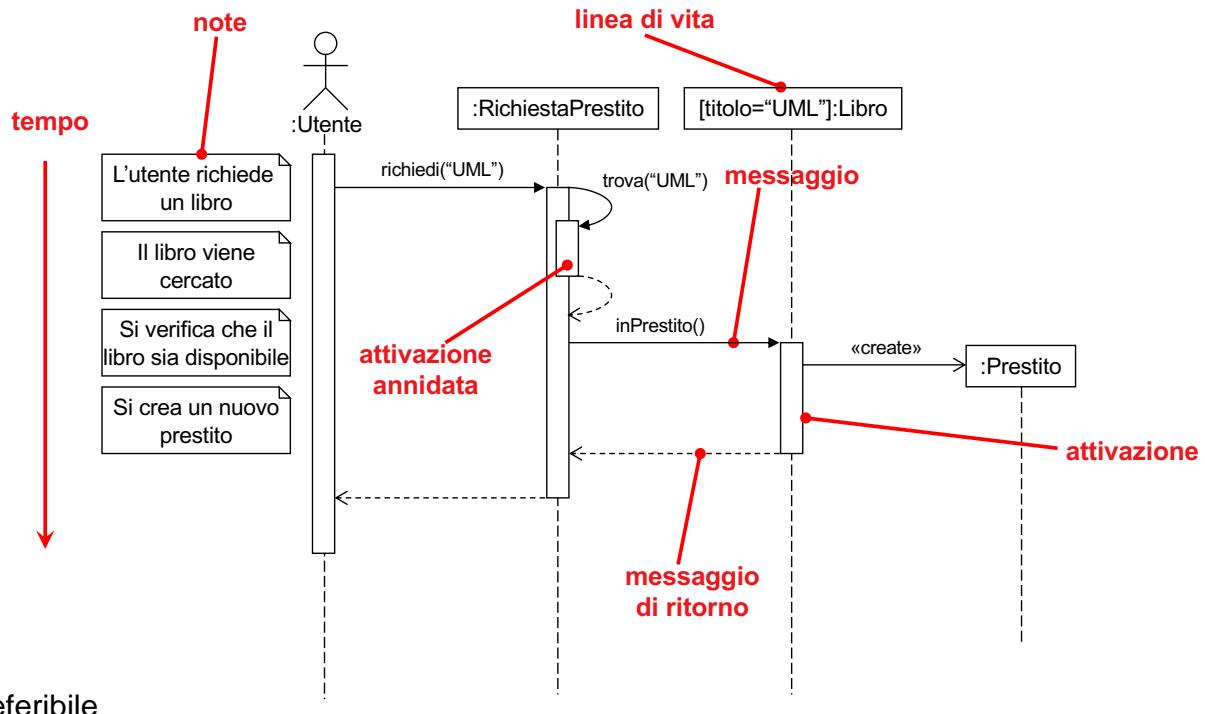
91

Diagrammi di sequenza

- Mostrano le interazioni tra linee di vita come una sequenza di messaggi ordinati temporalmente
- Sono la forma più ricca e flessibile di diagramma di interazione
 - ⇒ Hanno due **dimensioni**: la **dimensione verticale** rappresenta il **tempo** mentre quella **orizzontale** rappresenta le **linee di vita**
 - ⇒ Un'**attivazione** mostra il periodo durante il quale una linea di vita esegue un'**azione** o direttamente o attraverso una procedura subordinata; rappresenta sia la durata dell'azione nel tempo sia la relazione di controllo tra l'attivazione e i suoi chiamanti
 - ⇒ Si possono specificare **nodi decisionali**, **iterazioni**, **attivazioni annidate**
 - ⇒ È consigliato descrivere il flusso tramite un'insieme di **note** poste accanto agli elementi

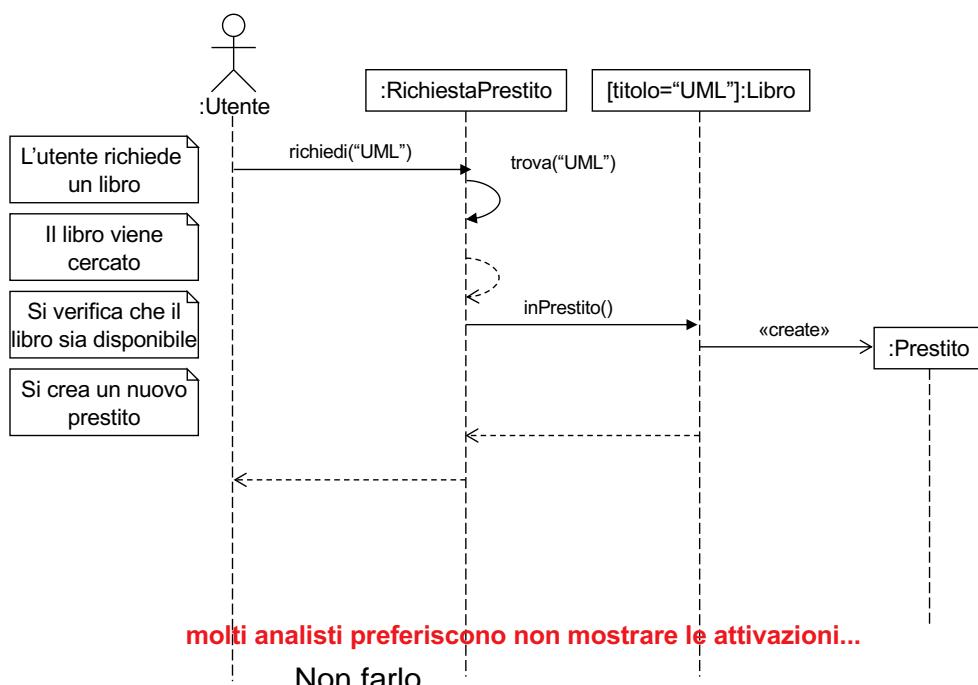
92

Richiesta prestito



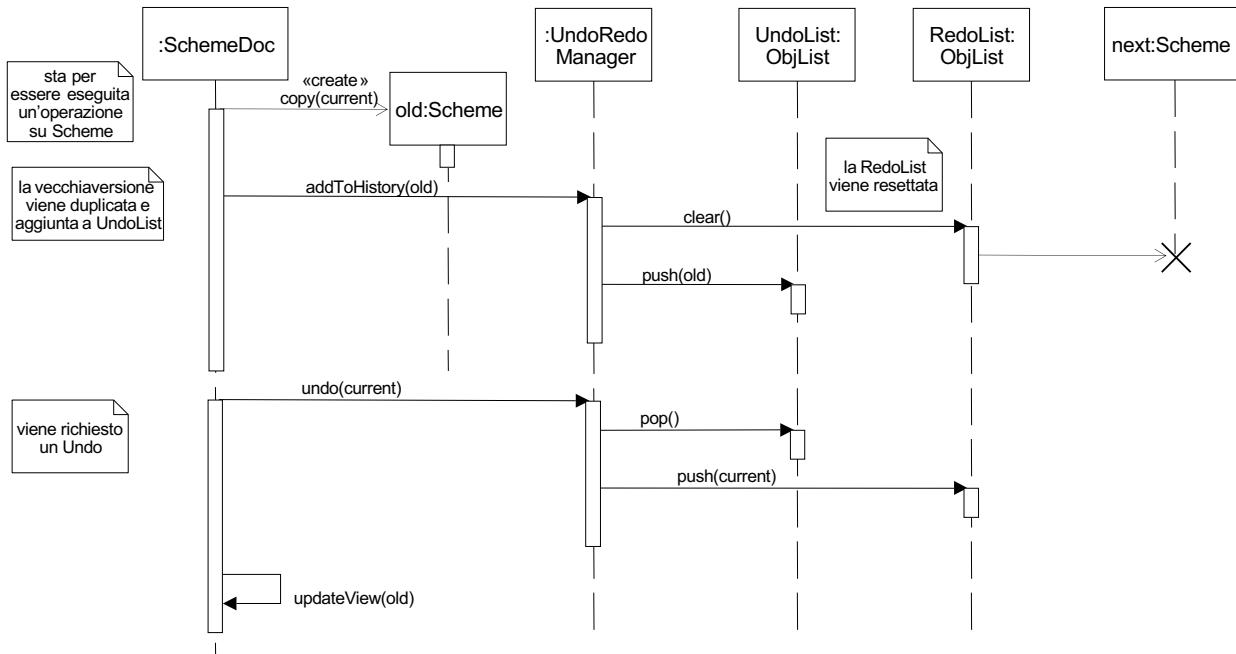
93

Richiesta prestito



94

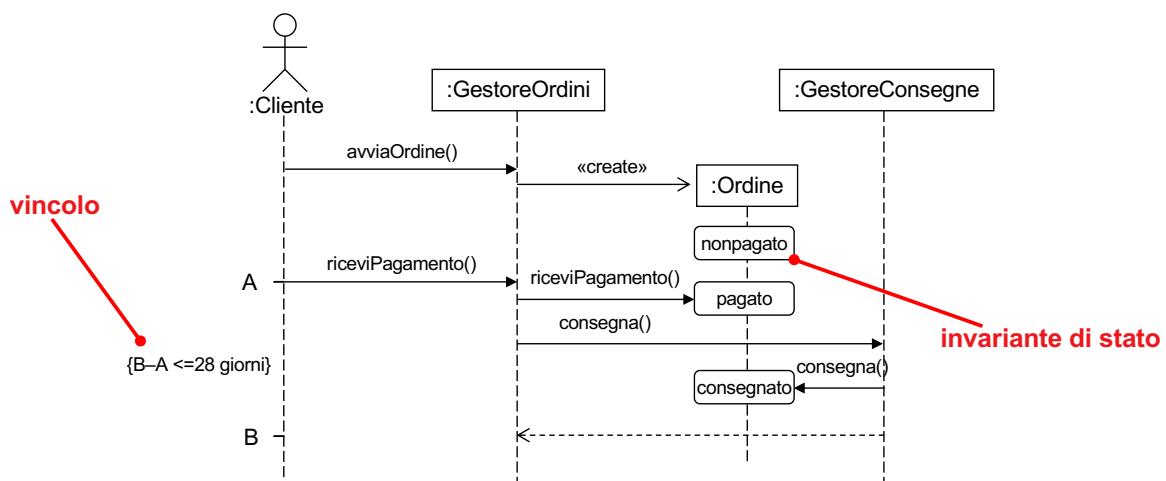
Gestione undo/redo



95

Invarianti di stato e vincoli

- Quando un'istanza riceve un messaggio, il suo stato può cambiare
- Lo **stato** delle istanze può essere mostrato sulla linea di vita
- Un **vincolo** posto sulla linea di vita indica una condizione sulle istanze che deve essere vera da lì in avanti

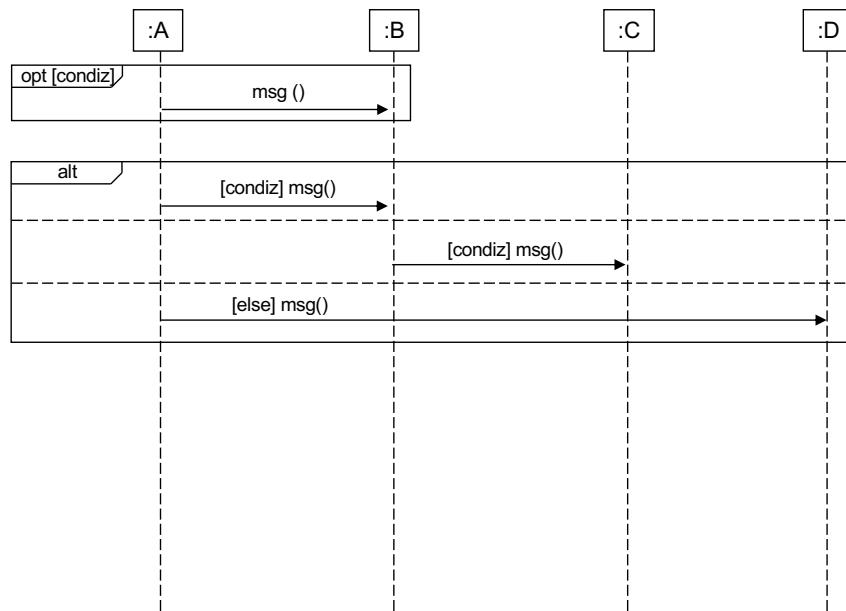


96

Frammenti combinati

Opt funziona come un if
quindi se condiz è true
allora A manda un msg a B

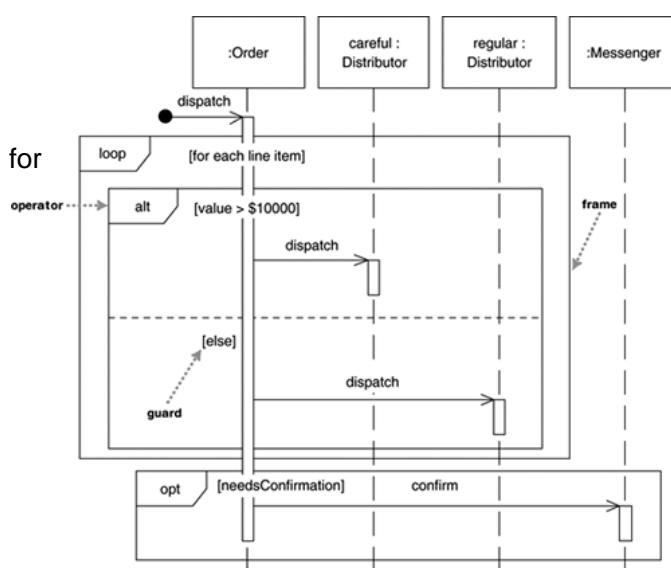
Alt funziona come
uno switch
quindi manda il msg
in base al valore della
condiz altrimenti
esegue else



97

Frammenti combinati

Loop funziona come un for

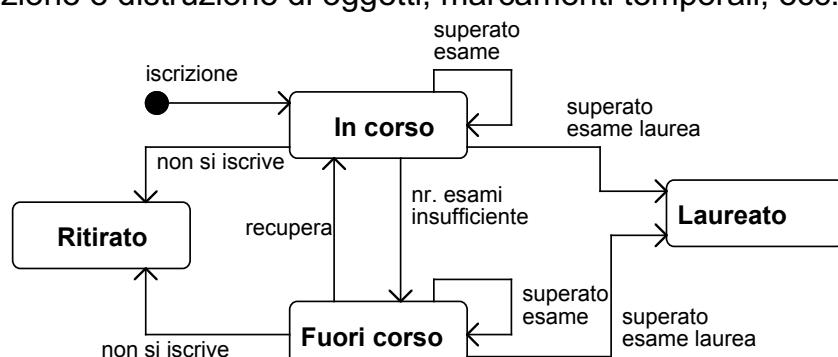


98

8

Diagrammi di stato

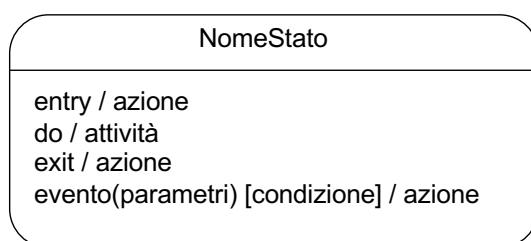
- I diagrammi di stato descrivono in modo esaustivo l'evoluzione temporale delle istanze di un classificatore (classe, caso d'uso, sottosistema) in risposta alle interazioni con altri oggetti
- Ogni classe può avere associato un diagramma di stato
 - ⇒ UML adotta la [notazione di Harel](#), che può esprimere sottostati, stati composti, parallelismo, stati storici, gestione eventi, operazioni, creazione e distruzione di oggetti, marcamenti temporali, ecc.



99

Stati ed eventi

- Lo **stato** di un oggetto in un certo istante è un'astrazione dell'insieme dei valori dei suoi attributi e dei suoi collegamenti
 - ⇒ Le differenti configurazioni di valori e collegamenti vengono raggruppate in stati a seconda di come incidono sul comportamento macroscopico dell'oggetto
- Un **evento** provoca la transizione tra uno stato e l'altro; un oggetto rimane in uno stato per un tempo finito non istantaneo corrispondente all'intervallo tra due eventi
- Uno stato può contenere azioni e attività:
 - ⇒ Le **azioni** sono operazioni istantanee, atomiche e non interrompibili; sono associate a transizioni attivate da eventi
 - ⇒ Le **attività** sono operazioni che richiedono un certo tempo per essere completate e possono quindi essere interrotte da un evento



100

Transizioni

- Una **transizione** marca il passaggio di un oggetto da uno stato a un altro, ed è associata a uno o più eventi e, optionalmente, a condizioni e azioni
- Un **evento** avviene a un preciso istante di tempo, e si assume che abbia durata nulla
 - ⇒ Gli eventi possono essere raggruppati in classi, eventualmente descritte da attributi
- Una **condizione** è un'espressione booleana che deve risultare vera affinché la transizione possa avvenire
- Un'**azione** è un'operazione istantanea, atomica e non interrompibile che viene eseguita all'atto della transizione

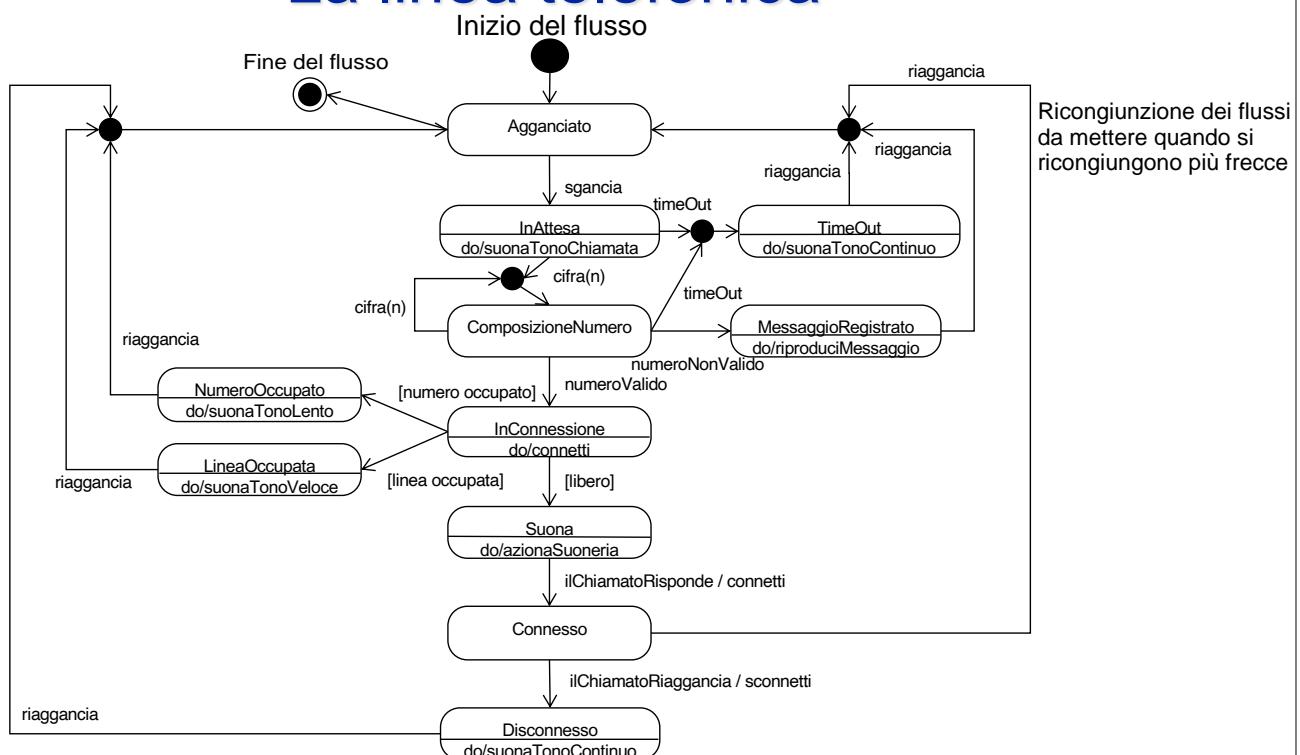
evento(parametri) [condizione] / azione
→

- Una transizione che esce da uno stato e non riporta alcun evento indica che la transizione avviene al termine dell'attività



101

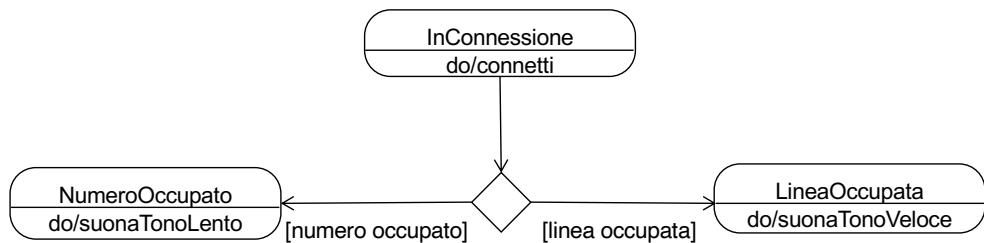
La linea telefonica



102

Pseudo-stato di selezione

- Consente di dirigere il flusso nell'automa secondo le condizioni specificate sulle sue transizioni di uscita



103

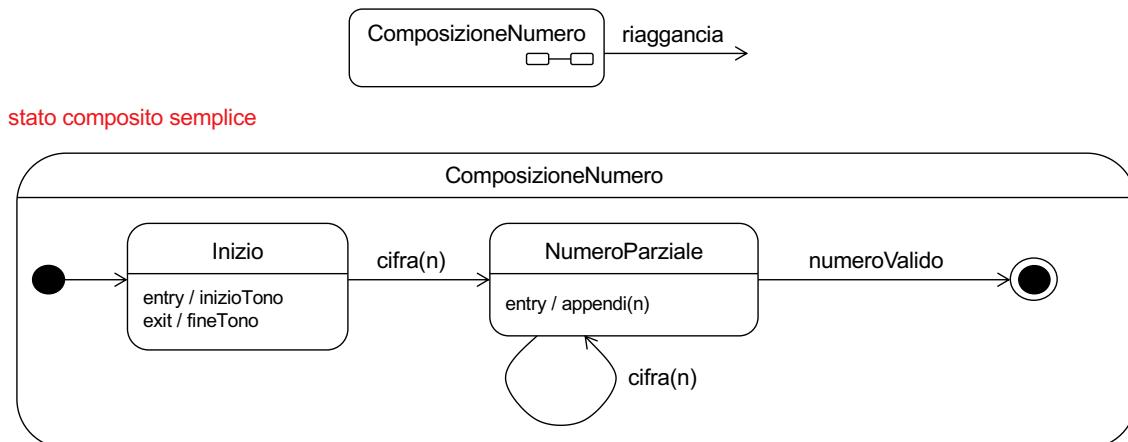
Tipi di eventi

- **Evento di variazione:** si verifica nel momento in cui una condizione diventa vera
 - ◆ è denotato da un'espressione booleana, ad esempio: bilancio<0
 - ◆ può essere considerato come una condizione verificata continuamente sebbene in realtà verrà controllata solo al variare dei parametri coinvolti
- **Evento di segnale:** si verifica nel momento in cui un oggetto riceve un oggetto segnale da un altro oggetto
- **Evento di chiamata:** è l'invocazione di una specifica operazione nell'istanza del classificatore che fa da contesto al diagramma
 - ◆ è denotato dalla signature dell'operazione
 - ◆ può essere associato a una sequenza di azioni separate da ";"
- **Evento temporale:** si verifica allo scadere di un periodo di tempo
 - ◆ **when**(data=01/01/2008): specifica il momento della transizione
 - ◆ **after**(10 seconds): specifica che la transizione deve avvenire dopo 10 secondi dall'entrata dell'automa nello stato attuale; è anche possibile specificare il momento in cui inizia a decorrere il periodo aggiungendo una frase del tipo "since..."

104

Stati composti

- Uno stato composito è uno stato che contiene altri stati annidati, organizzati in uno o più automi
- Ogni stato annidato eredita tutte le transizioni dello stato che lo contiene

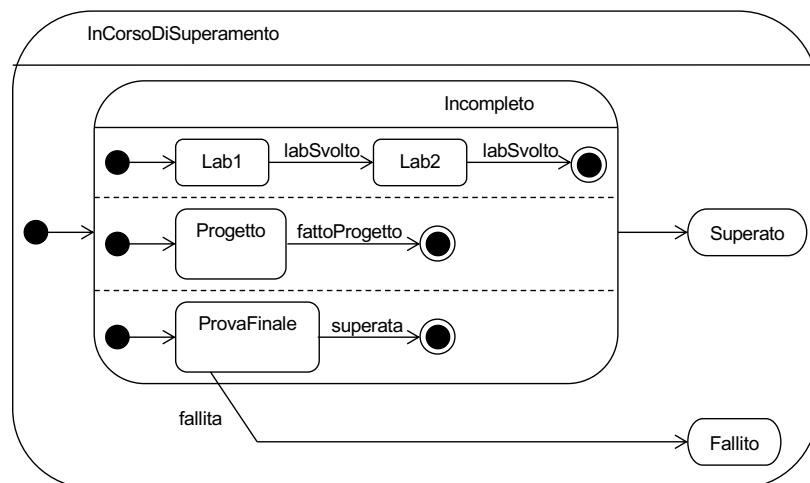


105

Stati composti

- Lo pseudo-stato finale di un automa viene applicato solo a quell'automa

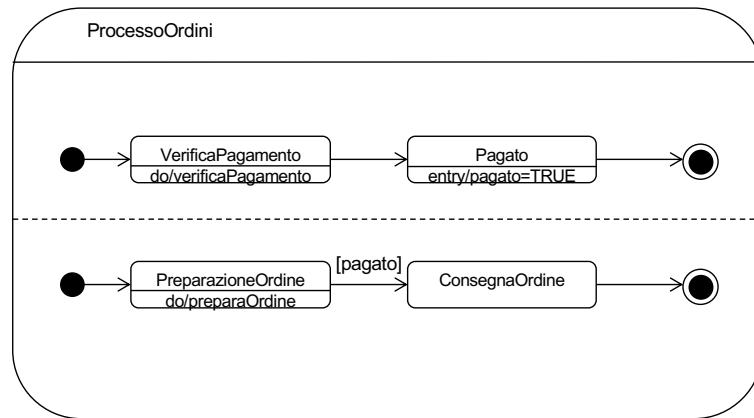
stato composito ortogonale



106

Comunicazione tra automi

- Si fanno comunicare in modo asincrono i due automi attraverso la variabile “pagato”



107

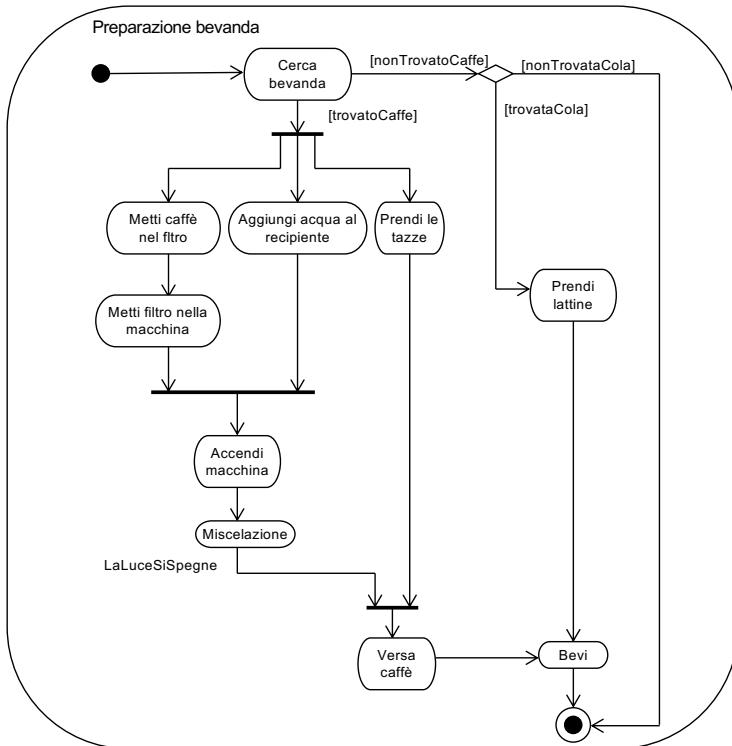
9

Diagrammi di attività

- Modellano un processo come un’attività costituita da un insieme di nodi connessi da archi
- In UML 2 hanno una nuova semantica basata sulle reti di Petri, differenziandosi completamente dai diagrammi degli stati
- Un’attività può essere associata a qualunque elemento di modellazione, che ne diviene il contesto:
 - ⇒ caso d’uso
 - ⇒ operazione
 - ⇒ classe
 - ⇒ interfaccia
 - ⇒ componente
 - ⇒ collaborazione
- I diagrammi di attività possono anche essere usati per modellare efficacemente processi di business e workflow

108

Preparazione di una bevanda



109

Attività

- Sono modellate come reti di nodi connessi da archi
- Categorie di nodi:
 - ⇒ nodi azione, che rappresentano compiti atomici all'interno dell'attività
 - ⇒ nodi controllo, che controllano il flusso all'interno dell'attività
 - ⇒ nodi oggetto, che rappresentano oggetti usati nell'attività
- Categorie di archi:
 - ⇒ flussi di controllo, che rappresentano il flusso di controllo attraverso l'attività
 - ⇒ flussi di oggetti, che rappresentano il flusso di oggetti attraverso l'attività

110

Nodi azione

☐ Nodo azione di chiamata

⇒ chiama un comportamento

Crea ordine

⇒ chiama un'attività

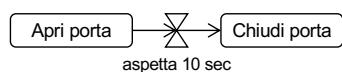
Crea ordine

⇒ chiama un'operazione

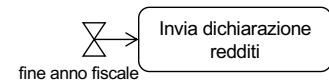
Stampa ordine
(Ordine::stampa)

☐ Nodo azione di accettazione evento temporale

⇒ produce un evento temporale ogni volta che la condizione temporale diventa vera



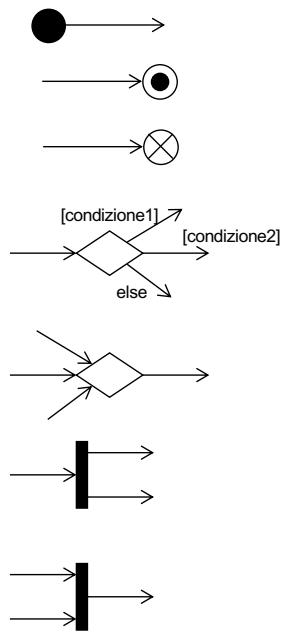
⇒ diventa attivo solo quando si attiva l'arco



111

Nodi controllo

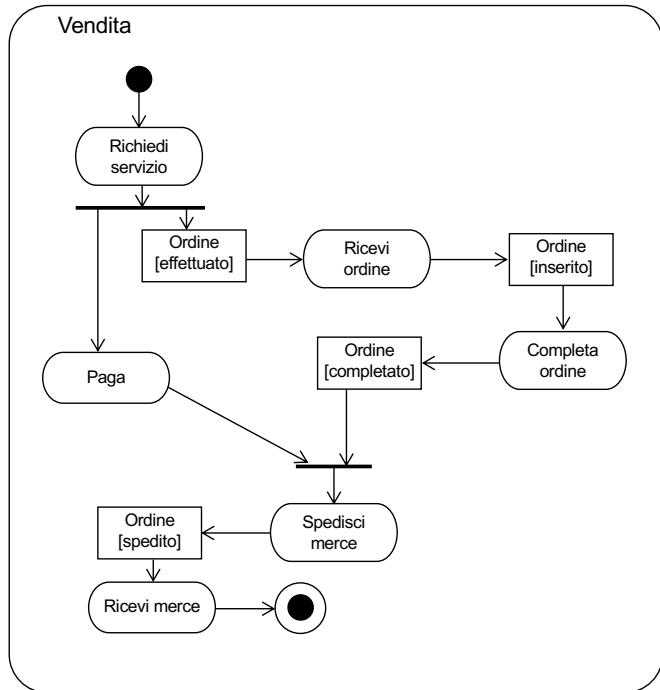
- ☐ **Nodo iniziale**: indica l'inizio del flusso
- ☐ **Nodo finale dell'attività**: termina un'attività
- ☐ **Nodo finale del flusso**: termina uno specifico flusso
- ☐ **Nodo decisione**: divide il flusso in più flussi alternativi
- ☐ **Nodo fusione**: ricongiunge i flussi a valle di un nodo decisione
- ☐ **Nodo biforcazione**: divide il flusso in più flussi concorrenti
- ☐ **Nodo ricongiunzione**: sincronizza flussi concorrenti



112

Nodi oggetto

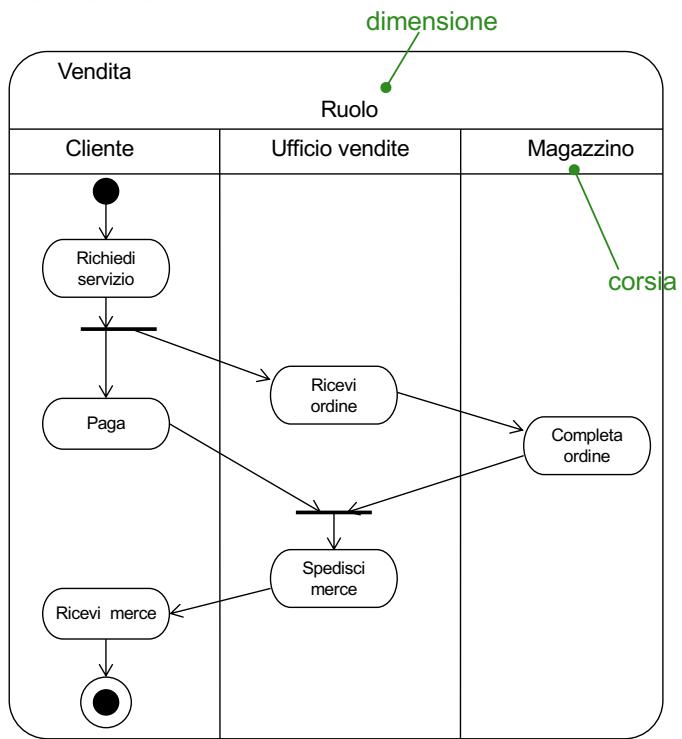
- I nodi oggetto indicano che sono disponibili istanze di una data classe in un punto specifico dell'attività
- Gli archi in entrata e uscita dai nodi oggetto rappresentano flussi di oggetti creati e consumati da nodi azione
- E' possibile rappresentare esplicitamente lo stato di un oggetto



113

Corsie

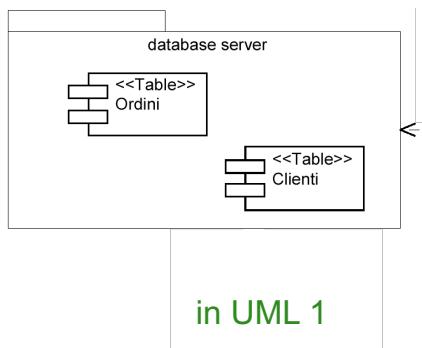
- Le attività possono essere partizionate in **corsie** che raggruppano insiemi di azioni correlate
- Le corsie possono corrispondere a
 - ⇒ casi d'uso
 - ⇒ classi
 - ⇒ componenti
 - ⇒ unità organizzative
 - ⇒ ruoli
- La semantica di ogni insieme di corsie è descritta da una **dimensione**



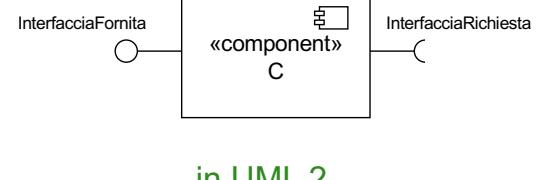
114

10 Diagramma dei componenti

- Mostra i componenti e le loro interdipendenze
- Un **componente** è una parte modulare di un sistema che **incapsula i suoi contenuti** (black box)
- I componenti possono avere attributi e operazioni, e possono partecipare ad associazioni e generalizzazioni



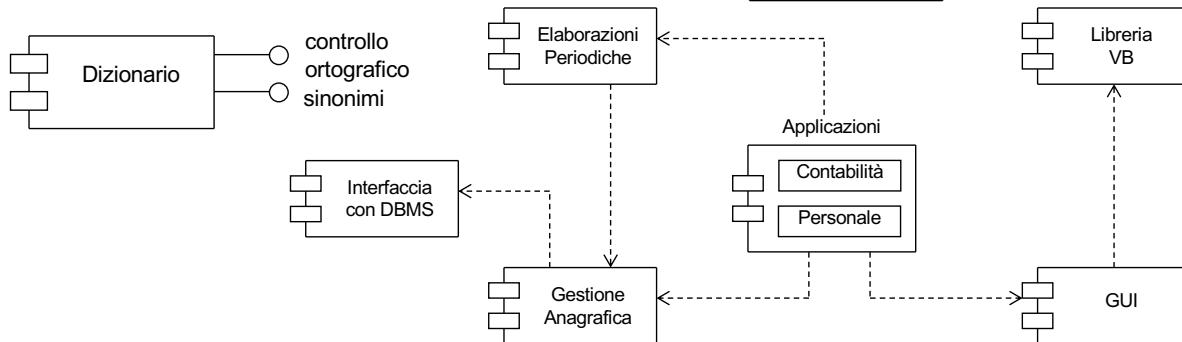
Il Form conosce il DataBase Server, e può essere coinvolto da ogni suo cambiamento



115

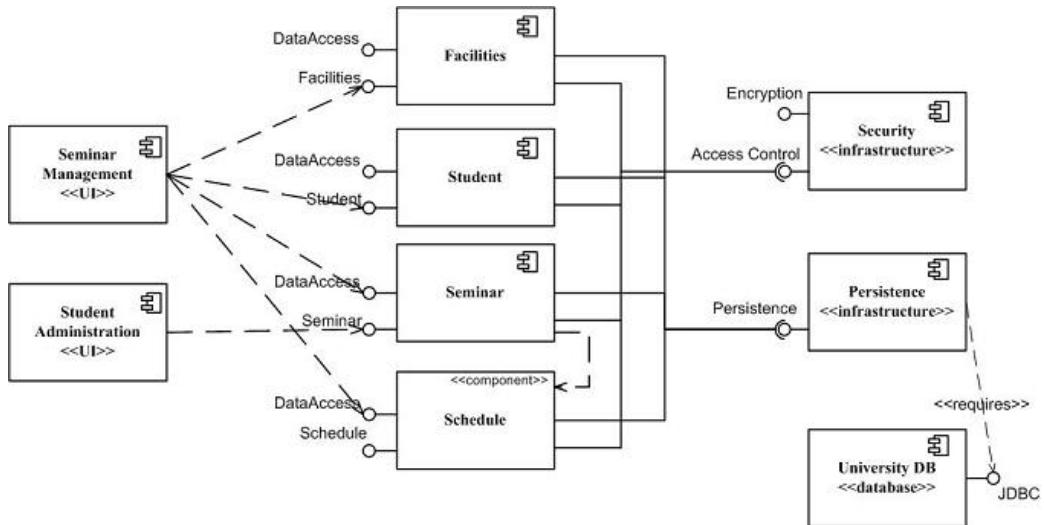
Componenti

- I componenti possono contenere oggetti, ad indicare che essi ne sono parte integrante
- I componenti sono connessi tra loro mediante **dipendenze**, possibilmente tramite **interfacce**, ad indicare che un componente usa i servizi di un altro componente



116

Esempio

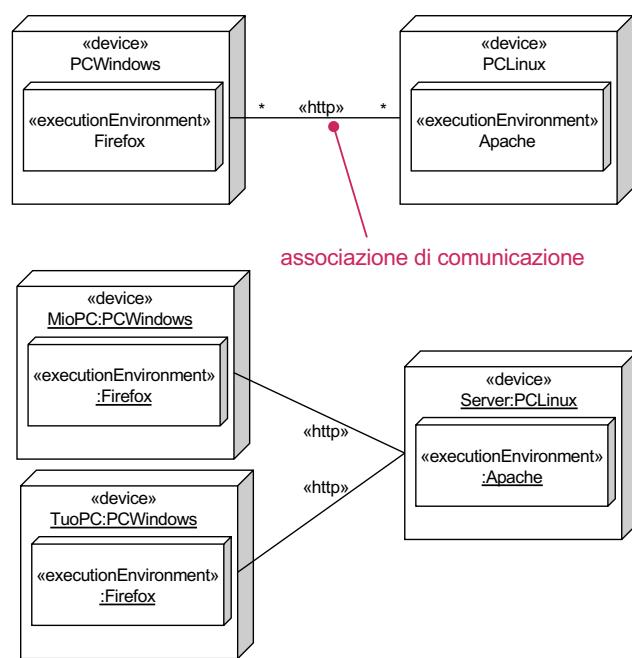


117

11

Diagramma di deployment

- Specifica l'hardware su cui verrà eseguito il software e il modo in cui il software è dislocato sull'hardware
- Può avere due forme:
 - ⇒ **descrittore**, che contiene nodi, relazioni tra nodi e manufatti; modella *tipi* di architetture
 - ⇒ **istanza**, che contiene istanze di nodi, di relazioni tra nodi e di manufatti; modella un deployment dell'architettura su un particolare sito



118

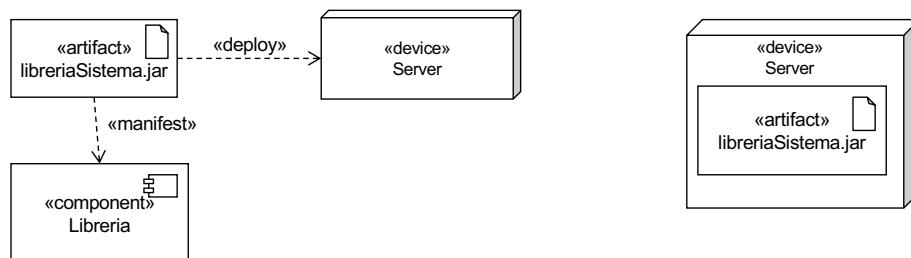
Nodi

- Un *nodo* rappresenta un tipo di risorsa computazionale su cui i manufatti possono essere dislocati per l'esecuzione
- Due stereotipi standard:
 - ⇒ «device» rappresenta un tipo di periferica fisica, per esempio un PC
 - ⇒ «executionEnvironment» rappresenta un tipo di ambiente software di esecuzione, per esempio un web server
- I nodi possono essere annidati in nodi
- Un'associazione tra nodi rappresenta un canale di comunicazione tra di essi
- Si possono usare ulteriori stereotipi e icone per aumentare la leggibilità del diagramma

119

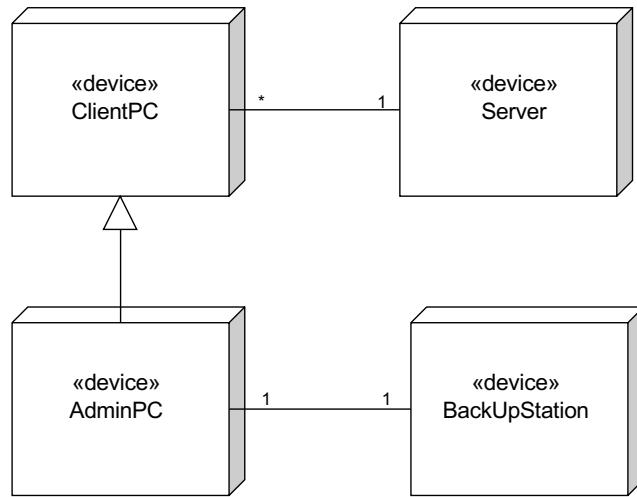
Manufatti

- Un manufatto rappresenta un'entità concreta del mondo reale, per esempio:
 - ⇒ file sorgenti
 - ⇒ file eseguibili
 - ⇒ script
 - ⇒ tabelle di database
 - ⇒ documenti
 - ⇒ modelli UML
- I manufatti vengono dislocati sui nodi



120

Esempio



121

12

Benefici di UML

- Superamento della "guerra dei metodi"
 - ⇒ decine di metodi di analisi e disegno proposti e praticati
 - ⇒ difficoltà per chi vuole passare all'approccio object-oriented: qual è il metodo migliore?
 - ⇒ quale strumento scegliere, se non c'è chiarezza nel campo dei metodi?
- Risposta ai problemi legati allo sviluppo di sistemi complessi con ambienti visuali
 - ⇒ ritorno di attenzione sul processo di lavoro e sugli approcci utilizzati, non solo sulle tecnologie
 - ⇒ il metamodello comune favorisce le possibilità di comunicazione tra strumenti di supporto alla progettazione, e più in generale tra i diversi ambienti utilizzabili dai progettisti nello sviluppo

122

Complessità

- Il metamodello di UML è molto complesso, perché ha l'ambizione di poter rappresentare qualunque tipo di sistema software, a livelli di astrazione differenziati
- Il numero dei diagrammi è elevato, e per molti diagrammi è possibile scegliere tra forme di rappresentazione leggermente diverse tra loro
- UML non suggerisce, né tantomeno prescrive una sequenza di utilizzo dei diversi diagrammi, lascia anzi molte strade aperte, tra le quali i progettisti sono liberi di scegliere

123

Personalizzazioni

- Le realtà che sviluppano software sono molto eterogenee: chi sviluppa per conto proprio non ha le stesse esigenze di documentazione e comunicazione di chi opera in un gruppo di lavoro all'interno di un'azienda
- Tra aziende diverse possono esserci differenze anche notevoli nel livello di formalizzazione richiesto ai progettisti, nelle tecniche da adottare, negli approcci da seguire, nel tipo di documentazione da produrre
- I progetti non sono tutti uguali: variano per dimensioni, per tipologia, per criticità, e per molti altri fattori
- UML può essere utilizzato da tutti, perché è sufficientemente complesso per potersi adattare a tutte le esigenze....
- ...ma non ha senso che tutti utilizzino UML esattamente nello stesso modo: per scegliere il percorso "ottimale", e quali diagrammi utilizzare davvero tra tutti quelli che UML mette a disposizione, è necessario verificare quali siano le esigenze da soddisfare

124

Quindi:

- UML è uno **standard**, e questo è un bene (uniformità nei concetti e nelle notazioni utilizzate, interoperabilità tra strumenti di sviluppo, indipendenza dai produttori, dalle tecnologie, dai metodi)
- UML è **articolato**: può rappresentare qualunque sistema software, a diversi livelli di astrazione
- UML è **complesso**: va adattato (“ritagliato”) in base alle specifiche esigenze dei progettisti e dei progetti, utilizzando solo ciò che serve nello specifico contesto

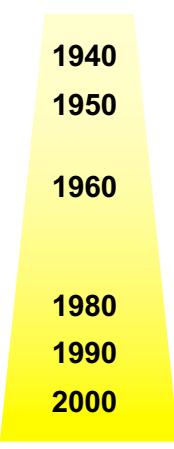
Ingegneria del Software

1

Introduzione

L'**ingegneria del software** tratta la realizzazione di sistemi software (sw) di dimensioni e complessità talmente elevate da richiedere uno o più team di persone per la loro costruzione.

La nascita e lo sviluppo del settore è una conseguenza diretta dell'aumento di complessità dei programmi.

- 
- 1940** I programmi sono sviluppati e utilizzati da una sola persona
 - 1950** Nasce la figura del programmatore
Sono realizzati i primi grandi sistemi commerciali (OS 360 IBM)
 - 1960** La complessità dei sistemi è tale da richiedere figure e metodologie che vedono il prodotto software come il risultato di una vera e propria opera ingegneristica
 - 1980** Il tempo dedicato alla progettazione supera il tempo dedicato alla programmazione
 - 1990** Nasce la certificazione del software (ISO-9000)
 - 2000** ISO-9000 si trasforma in Vision 2000

2

Definizioni

- *L'ingegneria del software è l'approccio sistematico allo sviluppo, all'operatività, alla manutenzione e al ritiro del software.*
- *L'ingegneria del software è la disciplina tecnologica e manageriale che riguarda la produzione sistematica e la manutenzione dei prodotti software che vengono sviluppati e modificati entro i tempi e i costi preventivati.*
- *L'ingegneria del software è un corpus di teorie, metodi e strumenti, sia di tipo tecnologico che organizzativo, che consentono di produrre applicazioni con le desiderate caratteristiche di qualità.*

3

1

La qualità del software

Le qualità su cui si basa la valutazione di un sw possono essere classificate in:

Interne: riguardano le caratteristiche legate allo sviluppo del sw; **non sono visibili agli utenti** **I**

Esterne: riguardano le funzionalità fornite dal prodotto; **sono visibili agli utenti** **E**

Le due categorie sono ovviamente strettamente legate: non è possibile ottenere le qualità esterne se il sw non gode delle proprietà interne

Relative al prodotto: riguardano le caratteristiche stesse del sw e sono **P** sempre valutabili

Relative al processo: riguardano i metodi utilizzati durante lo sviluppo del sw. **PC**

4

La qualità del software

E P Correttezza: un sw è corretto se rispetta le specifiche di progetto

E P Affidabilità: un sw è affidabile se l'utente può dipendere da esso

E P Robustezza: un sw è robusto se si comporta in modo ragionevole anche in circostanze non previste dalle specifiche di progetto (es. input incorretti, rotture di dischi)

N.B. La valutazione della correttezza e dell'affidabilità è basata sulle specifiche di progetto, mentre la robustezza riguarda tutti i casi non trattati.

E P Efficienza: un sw è efficiente se usa intelligentemente le risorse di calcolo

E P Facilità d'uso: un sw è facile da usare se l'interfaccia che presenta all'utente gli permette di esprimersi in modo naturale

I P Verificabilità: un sw è verificabile se le sue caratteristiche (correttezza, performance, ecc.) sono facilmente valutabili

I P Riusabilità: un sw è riusabile se può essere usato, in tutto o in parte, per costruire nuovi sistemi

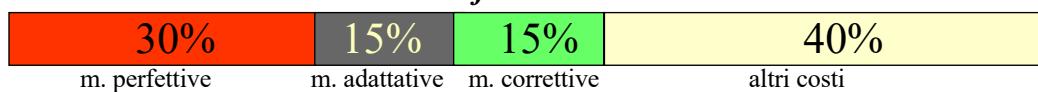
E P Portabilità: un sw è portatile se può funzionare su più piattaforme (es. Java)

5

La qualità del software

I P Facilità di manutenzione: un sw è facile da manutenere non solo se è strutturato in modo tale da facilitare la ricerca degli errori (*modifiche correttive*) ma anche se la sua struttura permette di aggiungere nuove funzionalità al sistema (*modifiche perfettive*) o di adattarlo ai cambiamenti del dominio applicativo (*modifiche adattative*).

Costo del Software



E P Interoperabilità: fa riferimento all'abilità di un sistema di coesistere e cooperare con altri sistemi (es. un word processor in cui possono essere creati grafici)

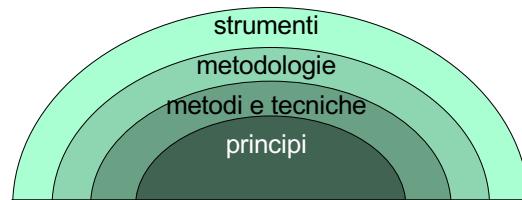
PC Produttività: misura l'efficienza del processo di produzione del software in termini di velocità di consegna del sw.

PC Tempestività: misura la capacità del processo di produzione del software di valutare e rispettare i tempi di consegna del prodotto.

PC Trasparenza: un processo di produzione del software si dice trasparente se permette di capire il suo stato attuale e tutti i suoi passi

6

Principi di progettazione



- Con il termine **software design** si intende il **processo che trasforma**, attraverso numerosi passi intermedi, le specifiche dell'utente in un insieme di specifiche direttamente utilizzabili dai programmatore
- Il risultato del processo di design è l'**architettura del software**, ossia l'insieme dei moduli che compongono il sistema, la descrizione della loro funzione, e delle relazioni esistenti tra di essi
- Tutte le fasi della progettazione sono ispirate a un insieme di **principi** su cui si basano le *tecniche* e i *metodi* utilizzati nelle fasi operative

7

Principi di progettazione: formalità

- L'utilizzo di **formalismi e di metodologie standardizzate** nelle fasi di progettazione, implementazione e documentazione del sistema **permette di ridurre fortemente gli errori di progetto** (es. incompletezza, inconsistenza, ambiguità)

8

Principi di progettazione: anticipazione dei cambiamenti

- La progettazione di un sistema informatico non deve mirare a soddisfare solo le specifiche **attuali** ma deve prevedere anche quelle **future**, poiché la capacità di prevedere i cambiamenti a cui il software sarà sottoposto durante il suo ciclo di vita determina la sua **semplicità di manutenzione** e la sua **riusabilità**
- I **cambiamenti** possono essere:
 - ⇒ **Noti a priori**: ogni software segue un cammino evolutivo rispetto alla sua prima release. Anche i servizi che non verranno inizialmente implementati devono comunque essere presi in considerazione durante la fase progettuale
 - ⇒ **Non noti a priori**: al fine di poter affrontare anche modifiche non prevedibili durante la fase di design, la progettazione deve cercare di rendere il progetto facilmente modificabile
- I cambiamenti possono riguardare:
 - ⇒ **Periferiche e hardware**
 - ⇒ **Dominio di applicazione**
 - ⇒ **Algoritmi e Strutture dati**: questi due elementi incidono fortemente sulle prestazioni del software. Accade spesso che nelle prime versioni del software si utilizzino algoritmi e strutture dati semplici al fine di velocizzare il completamento del sistema e di rendere più facile il debugging

9

Principi di progettazione: separazione degli argomenti

- Indica la necessità di individuare i diversi aspetti di un problema complesso e di trattarli separatamente al fine di semplificare la soluzione
- La suddivisione può essere fatta sulla base del:
 - ⇒ **Tempo** (alla base dei modelli di ciclo di produzione del software, che identificano e separano le attività da svolgere)
 - ⇒ **Livello di qualità** (dapprima si progetta il software in modo corretto quindi lo si ristruttura parzialmente al fine di aumentarne l'efficienza)
 - ⇒ **Vista** (nella fase di analisi dei requisiti può essere conveniente analizzare distintamente i flussi di dati tra le diverse attività e il flusso di controllo che le governa)
 - ⇒ **Livello di astrazione** (le specifiche vengono progressivamente raffinate) → **astrazione**
 - ⇒ **Dimensione** → **modularizzazione**

10

Principi di progettazione: modularità

- Con il termine **modulo** si indica il componente di base di un sistema software che raccoglie un insieme di funzionalità tra loro strettamente legate
- Benefici:
 - ⇒ capacità di scomporre un sistema complesso in parti più semplici
 - ⇒ capacità di comporre un sistema complesso a partire dai moduli esistenti
 - ⇒ capacità di capire un sistema in funzione delle sue parti
 - ⇒ capacità di modificare un sistema modificando soltanto un piccolo insieme delle sue parti
- Linee guida per la modularizzazione:
 - ⇒ Tutti i servizi strettamente connessi devono appartenere allo stesso modulo
 - ⇒ Ogni modulo deve essere realizzato in modo indipendente da ogni altro
 - ⇒ I programmatore devono essere in grado di operare su un modulo avendo una conoscenza minima del contenuto degli altri

11

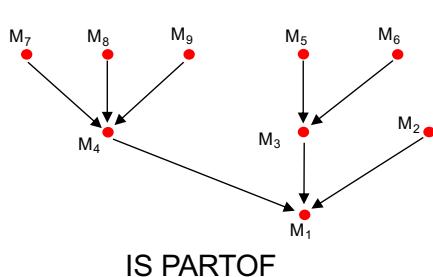
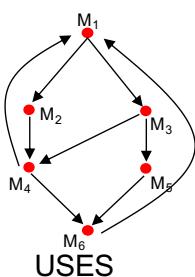
Principi di progettazione: modularità

- La definizione dell'**interfaccia** dei moduli deve rispettare il concetto di **information hiding**: l'interfaccia deve cioè contenere tutte le informazioni necessarie ad un corretto utilizzo del modulo evitando di mostrare i dettagli implementativi. Questo principio permette ai progettisti di modificare l'implementazione del modulo senza che ciò incida sulle altre componenti del sistema
 - ⇒ **Funzionalità a disposizione**: deve essere ben chiaro quali servizi sono realizzati dal modulo
 - ⇒ **Modalità di fruizione di un servizio**: per ogni servizio è necessario indicare la sequenza di routine da chiamare
 - ⇒ **Definizione dei parametri di input**: il tipo, il numero e la semantica dei parametri di input devono essere specificati in modo chiaro
 - ⇒ **Descrizione dell'output**: semantica e tipologia dei valori restituiti dalle routine devono essere completamente specificati. In particolare, per ogni routine deve essere presente una tabella dei codici di errore che riporti, oltre al tipo dell'errore verificatosi, i motivi che lo hanno provocato

12

Principi di progettazione: modularità

- La suddivisione di un sistema in moduli rende necessario tener traccia delle interazioni tra gli stessi. Le relazioni che devono essere descritte sono:
 - ⇒ **Di utilizzo (USES)**: indica quali moduli vengono utilizzati per completare i servizi forniti da un particolare modulo
 - ⇒ **Di composizione (IS PART OF)**: descrive la struttura del sistema a diversi livelli di astrazione permettendo ai progettisti di realizzare una documentazione più chiara e completa (al termine della fase progettuale gli unici moduli che compongono il sistema sono quelli al massimo livello di dettaglio)
 - ⇒ **Temporale**: descrive la sequenza con cui devono essere realizzati i diversi moduli
- Un valido strumento per la rappresentazioni di queste relazioni è il **grafo**



13

Principi di progettazione: astrazione

- E' uno strumento fondamentale per capire e analizzare problemi complessi, poiché consente di identificare gli aspetti fondamentali di un fenomeno e ignorare i suoi dettagli
- I modelli che vengono costruiti per i fenomeni sono sempre astrazioni della realtà, che trascurano alcuni aspetti ritenuti meno importanti per concentrarsi su altri

14

Principi di progettazione: generalità

- Ogni volta che si deve risolvere un problema, si cerca di capire qual è il **problema più generale** che gli si nasconde dietro
- Il problema generale:
 - ⇒ può essere più **semplice** di quello specifico
 - ⇒ la sua soluzione può essere più **riusabile**
 - ⇒ può essere **già risolto** in un'applicazione commerciale
- Importante per la realizzazione di software *off-the-shelf*

15

2

Misurazione

- Nel ciclo di vita del software la misurazione serve a prevedere o stimare tempi di consegna, costo di lavorazione, qualità del prodotto
- Come in ogni altro settore ingegneristico, alle misure viene assegnato il compito di normalizzazione tra oggetti e fenomeni distinti, per un loro confronto o per effettuare correlazioni, al fine di valutare e prendere decisioni
- La tipicità (e “non fisicità” del software) rendono però in parte ambigue le misure in questo settore. Ciò non ha impedito all’ingegneria del software di proporre un vasto insieme di misure e di metodi per la misurazione

16

Misurazione

- Scopi:
 - ⇒ la **previsione** delle caratteristiche che avrà il software in una fase del ciclo di vita diversa da quella in cui si effettua la valutazione
 - ⇒ la **stima** delle caratteristiche possedute dal software, nella fase e nello stadio di sviluppo in cui si effettua la valutazione
- Fasi:
 - ⇒ in fase di **progettazione**: per prevedere la manutenibilità e prevenire problemi nel software rilasciato in esercizio
 - ⇒ in fase di **collaudo e/o test**: per confrontare quanto fornito con le specifiche date
 - ⇒ dopo il **rilascio in esercizio**: per misurare l'impatto del prodotto sulla efficienza ed efficacia del lavoro svolto, confrontare le prestazioni con quelle di altri prodotti comparabili, individuare aree di possibile miglioramento, decidere il momento del ritiro dalla produzione
- In sostanza, **si misura per prendere decisioni ed agire**

17

Stima dei costi

- **Fonti di costo**
 - ⇒ Costo delle risorse per lo sviluppo del sw:
 - costo del personale tecnico
 - costo del personale di supporto
 - costo delle risorse informatiche
 - materiali di consumo
 - costi generali della struttura
- **Fattori di costo**
 - ⇒ Numero di istruzioni da codificare (benefici del riuso) 
 - ⇒ Capacità, motivazione e coordinamento degli addetti allo sviluppo
 - ⇒ Complessità del programma
 - ⇒ Stabilità dei requisiti
 - ⇒ Caratteristiche dell'ambiente di sviluppo

18

Le dimensioni del software

□ Metriche dimensionali

- ⇒ si basano sul numero di istruzioni del programma
- ⇒ LOC (*Lines Of Code*) o DSI (*Delivered Source Instructions*)

M = mesi uomo tot., E = numero tot. errori, \$ = costo tot., PD = pagine doc.

Produttività: $P = LOC/M$

Qualità: $Q = E/LOC$

Costo unitario: $C = \$/LOC$

Indici di qualità

Documentazione: $D = PD/LOC$

□ Metriche funzionali

- ⇒ si basano sulle caratteristiche funzionali del programma
- ⇒ Metodo dei Punti Funzione (*Function Points*)

19

Il metodo Function Points

- Fra le metriche del software riguardanti la dimensione, i FP sono la più vecchia (Allan Albrecht, metà degli anni 70) e tuttora la più diffusa:
 - ⇒ Restituisce un parametro adimensionale
 - ⇒ Misura la dimensione di un sw in termini delle funzionalità offerte all'utente (niente a che vedere con le funzioni dei linguaggi di programmazione!)
 - ⇒ La misurazione si basa sul disegno logico del software espresso in una forma qualsiasi: specifiche in linguaggio naturale, schemi Entity-Relationship, diagrammi di flusso dei dati, ecc.
 - ⇒ Può essere utilizzato a partire dalla prima fase dello sviluppo per poi ripetere la misura nel caso le specifiche siano cambiate
 - ⇒ E' indipendente dall'ambiente tecnologico in cui si sviluppa il progetto
 - ⇒ Consente confronti fra differenti progetti e organizzazioni

20

Il metodo Function Points

- Può essere usato da un'organizzazione come:
 - ⇒ Uno strumento per determinare la **complessità** di un pacchetto applicativo acquistato attraverso la quantificazione di tutte le sue funzioni
 - ⇒ Uno strumento che aiuti gli utenti a determinare il **beneficio** per le loro organizzazioni derivante da un pacchetto applicativo commerciale, attraverso la quantificazione delle sole funzioni che soddisfano i loro requisiti
 - ⇒ Uno strumento per **misurare** un prodotto, a sostegno di analisi sulla qualità e sulla produttività
 - ⇒ Un veicolo per **stimare** costi e risorse necessarie per lo sviluppo e la manutenzione del software
 - ⇒ Un fattore di normalizzazione per effettuare **confronti** sul software

21

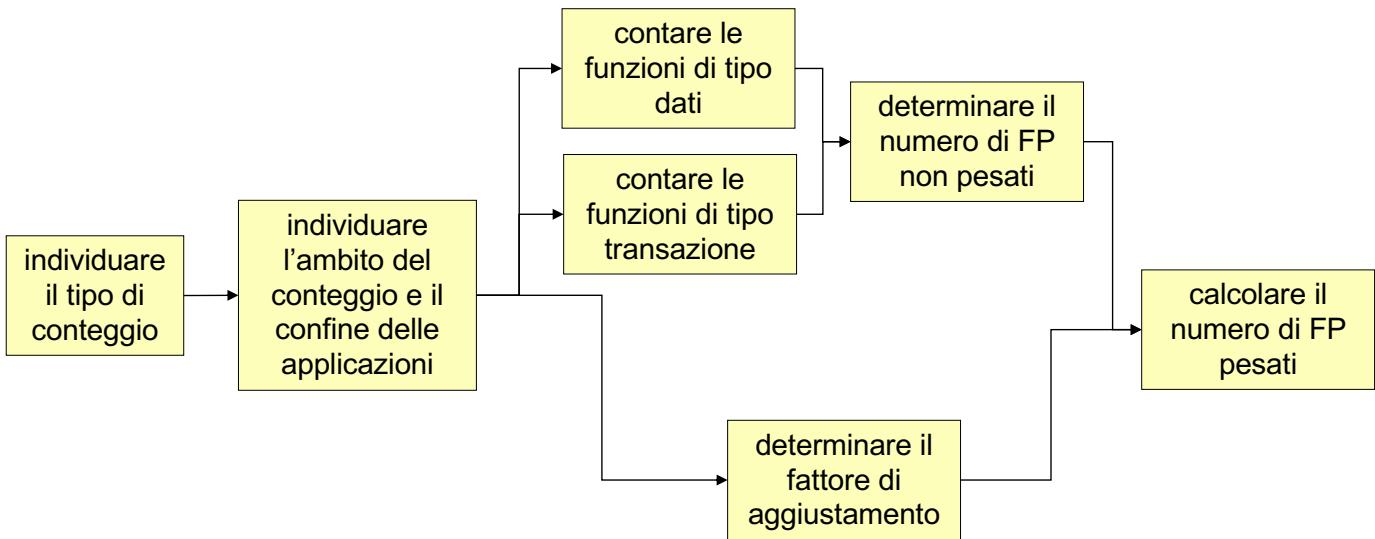
Conteggio dei function point

- Il **metodo** consiste nell'**identificare 5 tipi di funzioni** (funzionalità):
 - ⇒ **funzioni di tipo dati**
 - file interni logici
 - file esterni di interfaccia
 - ⇒ **funzioni di tipo transazione**
 - input esterno
 - output esterno
 - interrogazioni esterne
- Una volta identificate le funzioni, a ciascuna di esse si assegna un **peso** calcolato sulla base della quantità di dati e sulla complessità delle relazioni tra loro
- La somma dei pesi di tutte le funzioni costituisce il **Numeri di Function Points Non Pesato**
- Infine, questo numero è moltiplicato per un **fattore di aggiustamento** ottenuto considerando un insieme di 14 **Caratteristiche Generali del Sistema**

Tipo	Peso (semplice)	Peso (medio)	Peso (complesso)
Input Esterno	3	4	6
Output Esterno	4	5	7
Interrogazione esterna	3	4	6
File Interno Logico	7	10	15
File Esterno di Interfaccia	5	7	10

22

Conteggio dei function point



23

Tipi di conteggio

- ➡ *per progetti di sviluppo*: calcolo dei FP di un software da realizzare ex novo più eventuale conversione dei dati dalla vecchia applicazione
- ➡ *per progetti di manutenzione evolutiva*: misura le modifiche a un software esistente, comprendendo funzioni aggiunte, modificate, cancellate e di conversione
- ➡ *per una applicazione esistente*: consente il calcolo dei FP cosiddetti *installati* e il loro aggiornamento
 1. calcolo dei FP iniziali, differisce dal calcolo per i progetti di sviluppo perché non prevede funzioni di conversione
 2. aggiornamento dei FP dopo ogni manutenzione evolutiva, differisce dal calcolo per un progetto di manutenzione evolutiva perché i punti delle funzioni cancellate sono sottratte invece che sommate
- **Quindi i FP possono essere usati per misurare una applicazione durante tutto il suo tempo di vita**

24

Ambito del conteggio e confine delle applicazioni

- Identificare l'**ambito del conteggio** significa identificare le funzionalità che devono essere considerate in un conteggio
- Il **confine** è la linea di separazione tra le applicazioni che si stanno misurando e le applicazioni esterne o l'utente
- **Regole:**
 - ⇒ Il confine è determinato basandosi sul punto di vista dell'utente
 - ⇒ Il confine tra applicazioni collegate è basato su aree funzionali distinte dal punto di vista dell'utente e non in funzione degli aspetti tecnologici

25

Funzioni di Tipo Dati

- **File interno logico** (Internal Logical File: ILF)
 - ⇒ è un gruppo di dati o informazioni di controllo logicamente collegati e riconoscibili dall'utente che sono mantenuti all'interno dei confini dell'applicazione
 - ⇒ Il compito primario di un ILF è di contenere dati mantenuti attraverso uno o più processi elementari dell'applicazione che si sta contando
- **File esterno di interfaccia** (External Interface File: EIF)
 - ⇒ è un gruppo di dati o informazioni di controllo logicamente collegati e riconoscibili dall'utente che sono referenziati dall'applicazione ma sono mantenuti all'interno dei confini di un'altra applicazione
 - ⇒ Il compito primario di un EIF è di contenere dati referenziati da uno o più processi elementari dell'applicazione che si sta contando
 - ⇒ Questo significa che un EIF contatto per un'applicazione deve essere un ILF in un'altra applicazione

26

Esempio

□ ILF:

- ⇒ Dati sulle entità gestite dall'applicazione come: informazioni sugli impiegati, sui prodotti, sui clienti, ecc.
- ⇒ Dati sulle transazioni effettuate dall'applicazione come: registrazioni di prelievi da un conto corrente, di spese fatte con credit card, di movimentazione di magazzino, ecc.
- ⇒ Dati sulla sicurezza dell'applicazione (come password, accessi,...)
- ⇒ Dati di HELP
- ⇒ Dati di log (registrazione delle operazioni effettuate)

□ EIF:

- ⇒ Dati su entità gestite da altre applicazioni
- ⇒ Dati sulla sicurezza mantenuti all'esterno dell'applicazione
- ⇒ Dati di HELP mantenuti all'esterno dell'applicazione
- ⇒ Dati di log mantenuti all'esterno dell'applicazione

27

Funzioni di tipo transazione

□ Input Esterno (External Input: EI)

- ⇒ è un processo elementare dell'applicazione che elabora dati o informazioni di controllo provenienti dall'esterno del confine dell'applicazione
- ⇒ Il compito principale di un EI è di mantenere uno o più ILFs e/o di modificare il comportamento del sistema

□ Output Esterno (External Output: EO)

- ⇒ è un processo elementare dell'applicazione che manda dati o informazioni di controllo all'esterno del confine dell'applicazione
- ⇒ Il compito principale di un EO è di presentare informazioni all'utente attraverso una logica di processo diversa dal, o in aggiunta al, recupero di dati o informazioni di controllo
- ⇒ La logica di processo deve contenere almeno una formula matematica o calcolo, creare dati derivati, mantenere uno o più ILFs o modificare il comportamento del sistema

□ Interrogazione Esterna (External Inquiry: EQ)

- ⇒ è un processo elementare che manda dati o informazioni di controllo fuori dal confine dell'applicazione
- ⇒ Il compito principale di una EQ è di presentare informazioni all'utente attraverso il recupero di dati o informazioni di controllo da un ILF o EIF
- ⇒ La logica di processo non contiene formule matematiche o calcoli e non crea dati derivati

28

Fattore di aggiustamento

- Il numero totale di FP viene moltiplicato per un **fattore di aggiustamento** per tenere conto di quelle funzionalità generali del sistema non sufficientemente rappresentate dalle funzioni dati e transazionali
- Il valore del fattore di aggiustamento varia fra 0.65 e 1.35 (+/-35%) e viene calcolato sulla base del grado di influenza di ciascuna delle 14 Caratteristiche Generali del Sistema

29

Fattore di aggiustamento

- Il grado di influenza di una caratteristica è compreso tra 0 (nessuna influenza) e 5 (forte influenza):

$$\text{fattore di aggiustamento} = 0.65 + (\text{TDI} * 0.01)$$

con TDI (Total Degree of Influence) somma dei gradi di influenza per ciascuna caratteristica:

- comunicazione dati
- distribuzione dell'elaborazione
- prestazioni
- utilizzo estensivo della configurazione
- frequenza delle transazioni
- inserimento dati interattivo
- efficienza per l'utente finale
- aggiornamento interattivo
- complessità elaborativa
- riusabilità
- facilità di installazione
- facilità di gestione operativa
- molteplicità di siti
- facilità di modifica

30

Il numero ciclomatico

- Modello di metrica del software proposto da McCabe nel 1976
- Il numero ciclomatico è una definizione operativa di complessità del flusso di controllo di un programma, ed è legato all'identificazione di tutti i cammini che permettono di raggiungere una copertura accettabile del programma
 - ⇒ Misura della sola complessità del software intesa in riferimento alla sua produzione, comprensione e modifica
 - ⇒ Viene preso in considerazione il solo flusso di controllo, senza alcun riferimento alla complessità dei dati (grafo del flusso di controllo)
 - ⇒ Metrica svincolata dalle particolarità di un linguaggio

31

Il numero ciclomatico

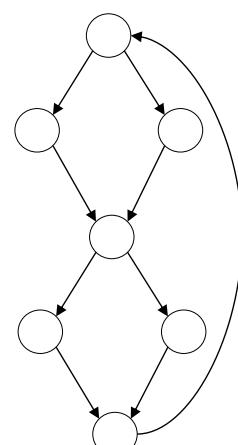
- Il numero ciclomatico di un grafo fortemente connesso è il numero minimo di archi che occorre eliminare per trasformarlo in un albero

ESEMPIO: numero ciclomatico = 3

- Il numero ciclomatico di un grafo fortemente connesso si calcola come:

$$e - n + 1$$

dove e è il numero degli archi ed n è il numero dei nodi



32

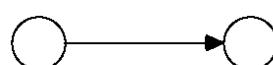
Il numero ciclomatico

- Se un programma è ben formato esistono sempre un nodo iniziale e uno terminale. Inoltre, esiste sempre almeno un cammino che permette di collegare il nodo iniziale con uno qualunque degli altri nodi ed almeno un cammino che permette di collegare uno qualunque dei nodi con il nodo terminale
- Si rende fortemente connesso il grafo del flusso di controllo del programma aggiungendo un arco orientato che va dal nodo terminale al nodo iniziale (+ 1 arco)
- Il numero ciclomatico del programma, assunto come misura della complessità del suo flusso di controllo, è il numero ciclomatico del grafo G modificato, $v(G)$, ed esprime il *numero di cammini linearmente indipendenti nel grafo di controllo*:

$$v(G) = e - n + 2$$

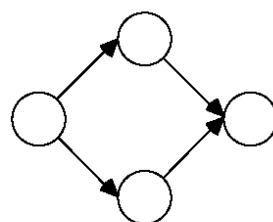
33

Il numero ciclomatico

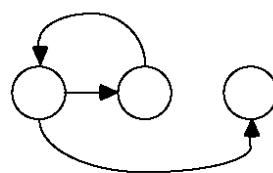


$$v(G)=1$$

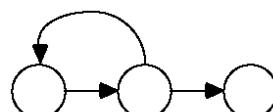
ESEMPI:



$$v(G)=2$$



$$v(G)=2$$



$$v(G)=2$$

34

Il numero ciclomatico

- *Teorema di Mills:*

$$v(G) = d + 1$$

dove d è il numero dei punti di decisione del programma
(assumendo che un punto di decisione a k uscite contribuisca come
 $k-1$ punti di decisione a 2 uscite)

- Se il programma ha procedure al suo interno, il numero ciclomatico dell'intero grafo è dato dalla somma dei numeri ciclomatici dei singoli grafi indipendenti:

$$v(G) = e - n + 2p$$

dove e ed n sono rispettivamente archi e nodi del grafo nel suo insieme, p è il numero di grafi (procedure) indipendenti

35

Il numero ciclomatico

- Il numero ciclomatico cattura almeno in parte ciò che intuitivamente è la complessità del flusso di controllo
- Esistono conferme sperimentali che indicano che si ha un buon grado di correlazione tra il numero ciclomatico e grandezze sicuramente influenzate notevolmente dalla complessità del flusso di controllo, come ad esempio il numero degli errori riscontrati
- Raccomandazione: *la complessità ciclomatica di un modulo non dovrebbe superare il valore 10*

36

COnstructive COst MOdel

- Si calcola una stima iniziale dei costi di sviluppo in base alla dimensione del software da produrre, poi la si migliora sulla base di un insieme di parametri

Modello intermedio

- 1) **Stima della dimensione del software:** calcolata come numero di linee di codice scritte (KDSI), può essere fatta sulla base dell'esperienza del manager oppure utilizzando una tecnica analitica basata, ad esempio, sul metodo FP:

FP	C _{Lang}	#Linee
1000	20 (Cobol)	20000
1000	32 (Pascal)	32000
1000	40 (C++)	40000
1000	88 (Assembler)	88000

37

COnstructive COst Model (2)

- 2) **Determinazione della classe del software:** i sw sono suddivisi in tre categorie con caratteristiche di difficoltà crescente. Per ogni categoria è stata sviluppata una diversa formula per il calcolo del costo, espresso in mesi uomo:

Organic	$M_{Nom} = 3.2 \times KDSI^{1.05}$
Semi-detached	$M_{Nom} = 3.0 \times KDSI^{1.12}$
Embedded	$M_{Nom} = 2.8 \times KDSI^{1.2}$

L'appartenenza ad uno dei tre profili viene determinata sulla base dei seguenti parametri:

	Organic	Semi-det.	Embedded
Conoscenza richiesta nel settore applicativo	Limitata	Normale	Completa
Esperienza del team nello sviluppo di software dello stesso tipo	Estesa	Considerabile	Moderata
Necessità di comunicare con sistemi esterni	Limitata	Considerabile	Elevata
Presenza di vincoli di progetto	Limitata	Considerabile	Elevata
Necessità di sviluppare apparecchiature hardware	Limitata	Normale	Elevata
Necessità di sviluppare strutture dati e algoritmi innovativi	Limitata	Normale	Elevata
Esistono premi per la consegna anticipata	Bassi	Normali	Elevati
Dimensione del prodotto	<50 KDSI	<300 KDSI	>300 KDSI

38

COnstructive COst Model (3)

3) Applicazione degli stimatori di costo:

$$M = M_{Nom} \times \prod_{i=1}^{15} c_i$$

	Molto Bassa	Bassa	Normale	Alta	Molto Alta	Extra
Proprietà del prodotto						
- Affidabilità del software richiesto	0.75	0.88	1.00	1.15	1.40	
- Complessità della base di dati		0.94	1.00	1.08	1.16	
- Complessità del prodotto	0.70	0.85	1.00	1.15	1.30	1.65
Caratteristiche dell'hardware						
- Vincoli di efficienza		1.00	1.11	1.30	1.66	
- Vincoli di memoria		1.00	1.06	1.21	1.56	
- Variabilità dell'ambiente di sviluppo		0.87	1.00	1.15	1.30	
- Tempi di risposta	0.87	1.00	1.07	1.15		
Caratteristiche del team						
- Capacità degli analisti	1.46	1.19	1.00	0.86	0.71	
- Esperienza nella classe di applicazioni	1.29	1.13	1.00	0.91	0.82	
- Capacità dei programmati	1.42	1.17	1.00	0.86	0.70	
- Esperienza nel linguaggio di programmazione	1.14	1.07	1.00	0.95		
- Esperienza nell'ambiente di sviluppo	1.21	1.10	1.00	0.90		
Caratteristiche del progetto						
- Modernità del processo di sviluppo	1.24	1.10	1.00	0.91	0.82	
- Utilizzo di tool di sviluppo	1.24	1.10	1.00	0.91	0.83	
- Presenza di un piano temporale di sviluppo	1.23	1.08	1.00	1.04	1.10	

39

3

Produzione

- Il processo di produzione è la sequenza di operazioni che viene seguita per costruire, consegnare e modificare un prodotto
- La complessità dei sistemi informatici e l'elevata instabilità del processo di costruzione dovuta alla volubilità del mercato rendono necessaria l'adozione di modelli di processo potenti e flessibili:
 - ⇒ Modello a cascata
 - ⇒ Modelli incrementali
 - ⇒ Modelli evolutivi
 - ⇒ Modelli agili

40

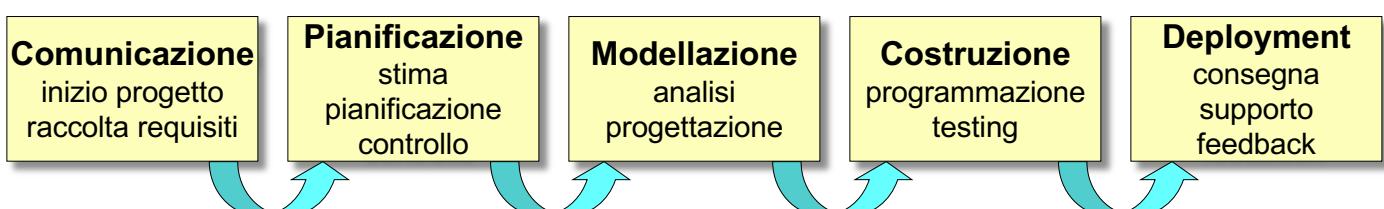
I modelli prescrittivi

- Definiscono un insieme distinto di attività, azioni, compiti, risultati e prodotti che sono necessari per ingegnerizzare un software di alta qualità
- Introducono elementi di stabilità, controllo e organizzazione in un'attività che, se lasciata incontrollata, tende a diventare caotica
- Producono programmi, documenti e dati
- Tutti i modelli prescrittivi comprendono sostanzialmente le stesse attività strutturali generiche:
 - ⇒ **comunicazione** (comprende la raccolta dei requisiti)
 - ⇒ **pianificazione**
 - ⇒ **modellazione**
 - ⇒ **costruzione** (comprende il testing)
 - ⇒ **deployment**
- Ogni modello applica un'enfasi differente a queste attività e definisce un flusso di lavoro che coinvolge ciascuna attività in modo differente

41

Il modello a cascata (Waterfall)

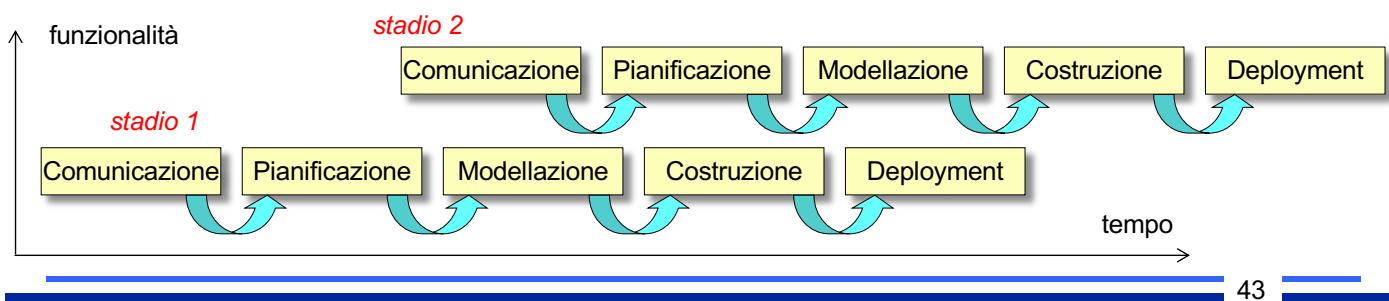
- Introdotto nel 1970, suggerisce un **approccio sistematico e sequenziale lineare**, in cui l'**output di ogni fase rappresenta l'input della successiva**
 - ⇒ E' inadeguato quando (come spesso accade) i requisiti sono incerti o non noti durante le fasi iniziali del progetto
 - ⇒ Non permette di modificare i risultati delle fasi precedenti alla luce di errori riscontrati a posteriori
 - ⇒ Solo al termine del progetto si genera una versione funzionante del programma



42

Il modello incrementale

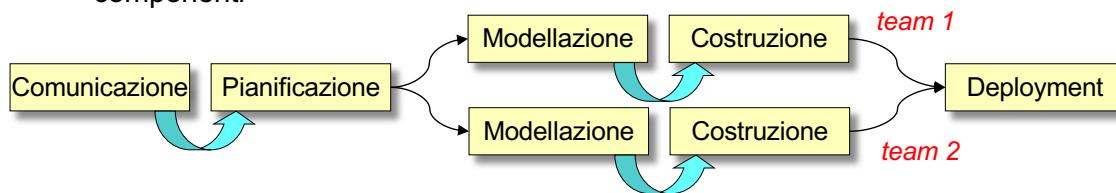
- E' un modello *iterativo* che combina aspetti del modello a cascata applicati a sottosistemi del prodotto finale, producendo il software a *incrementi*
- Consiste nell'applicare più sequenze lineari, scalate nel tempo, ognuna delle quali produce uno **stadio** operativo del software
 - ⇒ Il primo stadio consiste in genere in un "prodotto base", ossia un prodotto che soddisfa i requisiti fondamentali tralasciando alcune caratteristiche supplementari
 - ⇒ In seguito a una valutazione dell'utente, si stende un piano per lo stadio successivo, che preveda l'aggiunta di nuove funzionalità
- E' adatto a progetti in cui i requisiti iniziali sono ben definiti ma la dimensione del sistema scoraggia l'adozione di un processo puramente lineare



43

Il modello RAD

- Rapid Application Development è un modello di processo incrementale che punta a un ciclo di sviluppo molto breve
- Si tratta di un adattamento del modello a cascata, nel quale l'obiettivo di accelerare lo sviluppo è raggiunto grazie a strategie costruttive fondate sull'uso di **componenti**
- Ogni applicazione modularizzabile in modo che ciascuna funzionalità principale possa essere completata in meno di 3 mesi è candidata al RAD
- Ogni funzionalità viene affrontata da un team RAD distinto e poi integrata a formare un unico prodotto
- RAD fallisce se:
 - ⇒ gli utenti non riescono a tenere il passo
 - ⇒ il sistema non è modularizzabile
 - ⇒ sono richieste alte prestazioni da ottenere tramite l'ottimizzazione delle interfacce tra i componenti



44

Incrementale vs. iterativo

□ Similarità

- ⇒ Prevedono entrambi più versioni successive del sistema
- ⇒ Ad ogni istante dopo il primo rilascio esiste una versione in esercizio e una versione in sviluppo

□ Differenze

- ⇒ Sviluppo *incrementale*: ogni versione aggiunge nuove funzionalità o sottosistemi
- ⇒ Sviluppo *iterativo*: da subito sono presenti le funzionalità/sottosistemi di base che vengono successivamente raffinate e migliorate. I requisiti possono cambiare

45

I modelli evolutivi

□ Osservazioni:

- ⇒ I sistemi software evolvono nel tempo, e i loro requisiti cambiano durante lo sviluppo
 - Anche se è impossibile realizzare un prodotto completo e competitivo nei tempi dettati dal mercato, potrebbe essere possibile realizzare una versione limitata per rispondere alla pressione della concorrenza
 - ⇒ A volte il cliente riesce a definire solo **obiettivi generali** per il software, ma non riesce ad identificare requisiti dettagliati in termini di input, elaborazione o output
- I modelli evolutivi sono iterativi, e caratterizzati in modo tale da consentire lo sviluppo di versioni sempre più complete del software
- ⇒ Si produce una versione limitata, sulla base di requisiti ben noti, e successivamente si realizzano delle estensioni
 - ⇒ Si fa largo uso di **tecniche di prototipazione**

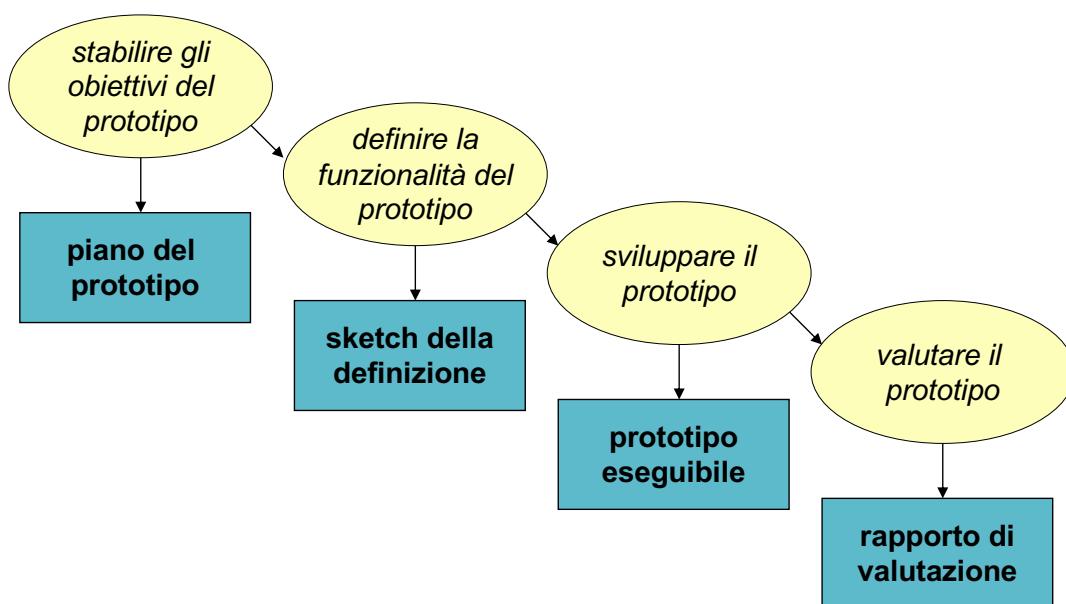
46

Prototipazione

- Un prototipo è una versione approssimata, parziale (funzionante), dell'applicazione che deve essere sviluppata
- Obiettivi:
 - ⇒ Un prototipo software permette di animare e dimostrare i requisiti
 - L'uso principale consiste nell'aiutare i clienti e gli sviluppatori a capire meglio i requisiti inizialmente vaghi o insufficienti
 - Il prototipo può essere usato per l'addestramento dell'utente prima che sia consegnato il sistema finale
- Benefici:
 - ⇒ Equivoci fra gli utenti del sw e gli sviluppatori sono messi in evidenza
 - ⇒ Possono essere evidenziate funzionalità mancanti o confuse
 - ⇒ Un sistema funzionante è disponibile molto presto nel processo
 - ⇒ Il prototipo può servire come base per derivare una specifica del sistema

47

Prototipazione



48

Prototipazione

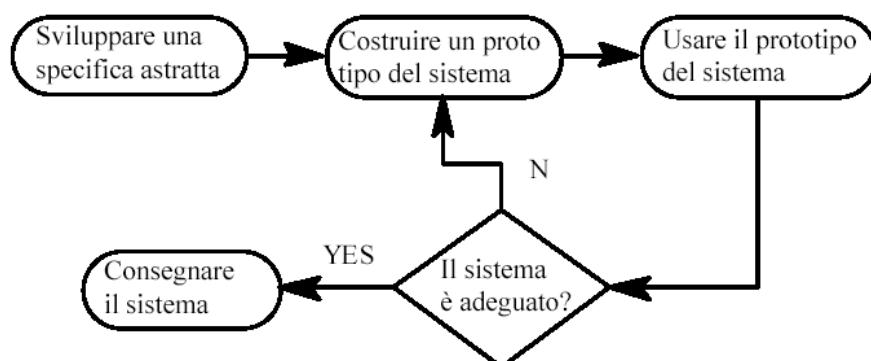
□ Due tecniche:

- ⇒ L'obiettivo della *prototipazione evolutiva* è di fornire un sistema funzionante all'utente finale
 - Lo sviluppo parte con i requisiti che sono meglio capiti
 - il software del prototipo vale circa il 14-15% del prodotto finito
 - il prototipo viene fatto evolvere nel prodotto finale, senza gettarlo
- ⇒ L'obiettivo del *prototipo usa e getta* è di validare o derivare i requisiti del sistema
 - Il processo di prototipazione parte con i requisiti che non sono ben capiti
 - il software del prototipo vale circa il 5-10% del volume del prodotto finito

49

Prototipazione evolutiva

□ Viene usata per sistemi in cui le specifiche non possono essere sviluppate in anticipo, per esempio sistemi AI e interfacce utente



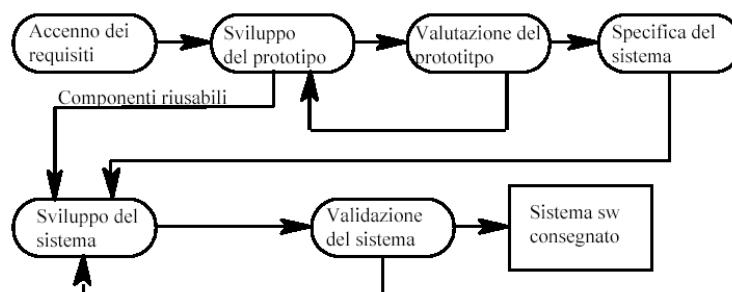
□ Ma...

- ⇒ cambiamenti continui tendono a corrompere il sistema, per cui il mantenimento a lungo termine diviene costoso
- ⇒ sono richieste grandi capacità di progettazione e programmazione
- ⇒ si deve accettare che il tempo di vita del sistema sia corto

50

Prototipazione usa-e-getta

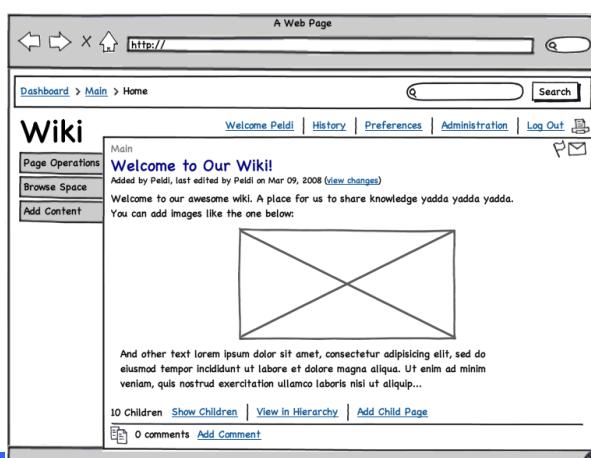
- Usata per ridurre il rischio dei requisiti incerti
- Il prototipo è sviluppato da una specifica iniziale, consegnato per sperimentazione e quindi gettato
- Il prototipo NON deve essere considerato un sistema finale perché
 - ⇒ alcune caratteristiche del sistema possono non essere state considerate
 - ⇒ non c'è specifica per il mantenimento a lungo termine
 - ⇒ il prototipo non è strutturato bene e sarebbe difficile da mantenere



51

Prototipazione dell'interfaccia utente

- E' impossibile specificare in anticipo il *look and feel* di una interfaccia utente in maniera efficace, quindi prototipare è essenziale
- Lo sviluppo di GUI (Graphical User Interface) sta diventando una attività che prende la maggior parte del costo dello sviluppo del sistema
- Generatori di interfacce utente possono essere usati per "disegnare" l'interfaccia e simularne la funzionalità

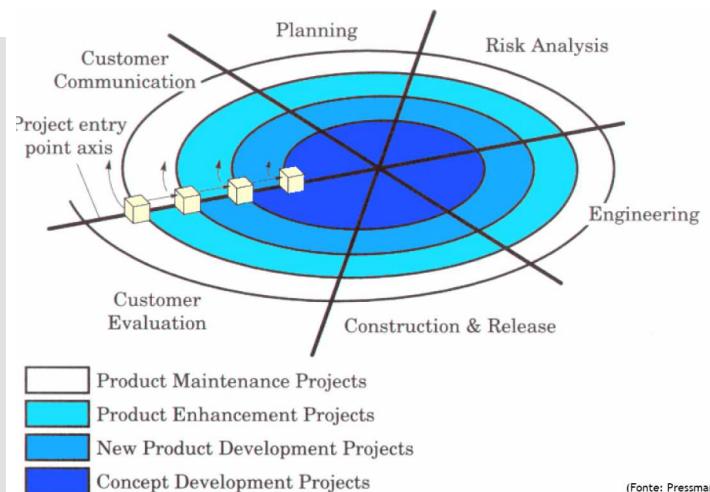


52

Il modello a spirale

- Fa crescere incrementalmente il grado di definizione e implementazione del sistema (a partire da un modello cartaceo o da un prototipo), riducendo il livello di rischio e producendo un insieme di milestone per garantire la fattibilità delle soluzioni intraprese

1. **Customer communication:** Colloquio tra cliente e team di sviluppo
2. **Planning:** Raccolta requisiti e definizione piano di progetto
3. **Risk analysis:** Stima e prevenzione dei rischi tecnici e di gestione
4. **Engineering:** Modellazione e progettazione
5. **Construction & release:** Realizzazione, collaudo e installazione
6. **Costumer evaluation:** Rilevazione delle reazioni da parte del cliente

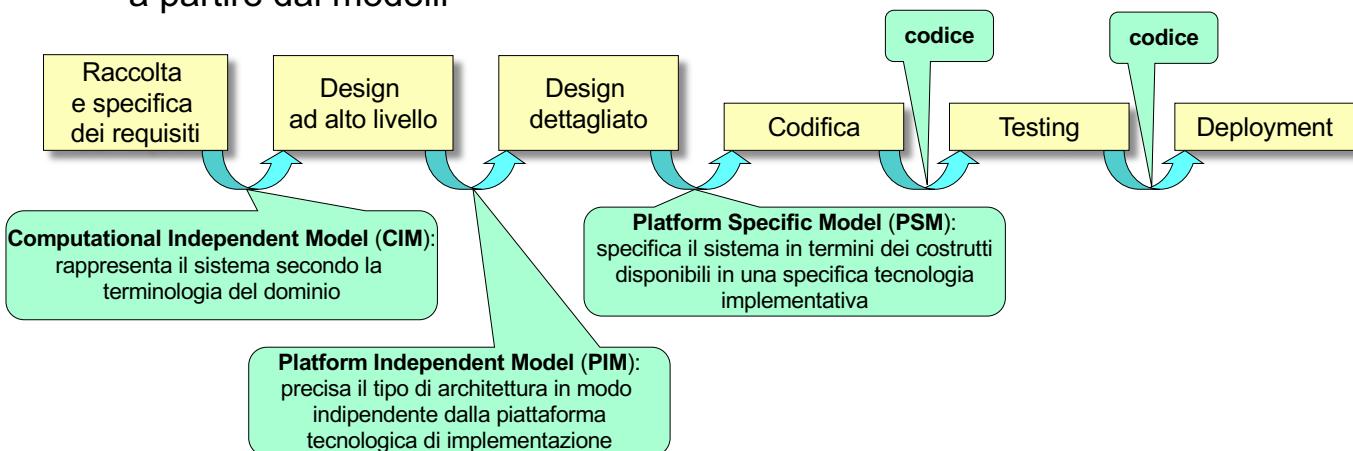


(Fonte: Pressman)

53

Model-Driven Development

- MDD è un tipo di sviluppo in cui si creano modelli formali del software che vengono poi fatti evolvere mentre il sistema viene progettato e implementato
- I modelli diventano la guida del processo di sviluppo; infatti, MDD prevede l'uso di strumenti per la generazione automatica del codice e dei test case a partire dai modelli



54

I modelli agili

- I modelli prescrittivi, basati su una ferrea disciplina, trascurano la fragilità delle persone che realizzano il software
- I modelli di processo agili presentano le seguenti caratteristiche:
 - ⇒ incoraggiano la soddisfazione del cliente e una consegna incrementale anticipata del software
 - ⇒ impiegano team di progettazione compatti e molto motivati
 - ⇒ impiegano metodi informali
 - ⇒ producono un livello minimo di prodotti di ingegneria del software
 - ⇒ incoraggiano semplicità di sviluppo
 - ⇒ richiedono comunicazione continua tra sviluppatori e utenti

55

Extreme programming

- E' il più diffuso modello di processo agile, nato nel 1999
- XP adotta un approccio object-oriented e include 4 attività strutturali:
 - ⇒ **Pianificazione**
 - definisce un insieme di *user story* che descrivono le funzionalità del software
 - a ogni user story il cliente assegna un valore che ne definisce la priorità
 - i progettisti assegnano a ogni user story un costo (in settimane di sviluppo)
 - se una user story richiede più di 3 settimane di sviluppo, si chiede al cliente di frammentarla
 - il cliente e i progettisti decidono quali user story inserire nella prossima release, e le ordinano per valore o per rischio decrescenti

56

Extreme programming

- E' il più diffuso modello di processo agile, nato nel 1999
- XP adotta un approccio object-oriented e include 4 attività strutturali:

⇒ Design

- persegue la massima semplicità
- viene scoraggiata la progettazione di funzionalità aggiuntive
- incoraggia l'uso di *schede CRC* (Classe-Responsabilità-Collaborazione)
- se viene individuato un problema di design, si crea immediatamente un **prototipo operativo** (*spike solution*) che viene poi valutato
- incoraggia il **refactoring**, ossia un **processo di "ripulitura" e riorganizzazione del software** che non ne altera il comportamento esterno
- l'architettura viene considerata un elemento transitorio

57

Extreme programming

- E' il più diffuso modello di processo agile, nato nel 1999
- XP adotta un approccio object-oriented e include 4 attività strutturali:

⇒ Programmazione

- si basa sul **pair programming**, in cui 2 persone (in genere con ruoli leggermente differenziati) collaborano alla stessa workstation per sviluppare il software così da fornire un meccanismo di soluzione in tempo reale dei problemi e una garanzia di qualità

⇒ Testing

- già prima dell'inizio della programmazione vengono definiti degli *unit test*, ossia test di ogni singolo componente, che vengono ora implementati attraverso uno strumento di supporto che ne consenta l'automazione
- si incoraggia il **test di regressione** a ogni modifica del software

58

Extreme programming

- Dopo il primo rilascio del progetto, il team XP calcola la **velocità del progetto**, intesa come il numero di user story implementate nella prima release
- La velocità del progetto è utilizzata per
 - ⇒ stimare le date di consegna e le pianificazioni per le successive release
 - ⇒ determinare se le user story sono state sottovalutate, ed eventualmente modificare il contenuto delle prossime release o le loro date di consegna

59

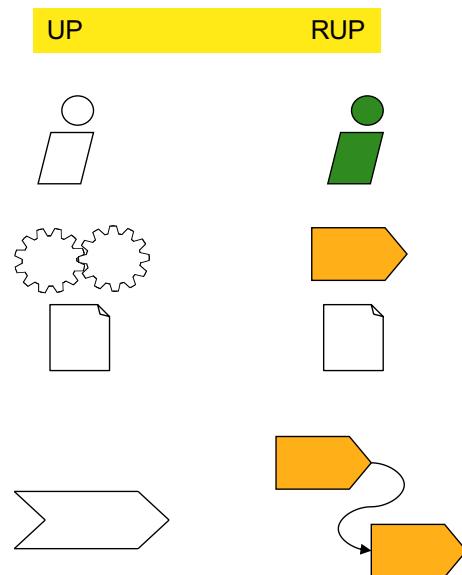
Unified Process

- Unified Process (UP) è il processo di sviluppo del software ideato da Booch, Rumbaugh, Jacobson (gli autori di UML)
 - ⇒ Guidato dai casi d'uso
 - ⇒ Centrato sull'architettura
 - ⇒ Iterativo e incrementale
 - ⇒ Model-based e component-based
 - ⇒ Object-oriented
 - ⇒ Configurabile

60

Un modello di UP

- **CHI:** Una **risorsa** o **ruolo** definisce il comportamento e le responsabilità di un individuo o un gruppo
- **COSA:** Il **comportamento** è espresso in termini di **attività** e **manufatti**
- **QUANDO:** Si modellano **flussi di lavoro**, ossia sequenze di **attività correlate eseguite da ruoli che producono manufatti**



61

Manufatti

- **Set di gestione**
 - ↳ elaborati di pianificazione (software development plan, studio economico, ...)
 - ↳ elaborati operazionali (stato di avanzamento, descrizione versione, ...)
- **Set dei requisiti**
 - ↳ documento di visione
 - ↳ modello dei casi d'uso
 - ↳ modello di business
- **Set di progettazione**
 - ↳ modello di design
 - ↳ modello architetturale
 - ↳ modello di test
- **Set di implementazione**
 - ↳ codice sorgente ed eseguibili
 - ↳ file di dati
- **Set di rilascio agli utenti**
 - ↳ script di installazione
 - ↳ documentazione utente
 - ↳ materiale formativo

62

Flussi di lavoro

- I flussi di lavoro non sono rigidamente sequenziali, e vengono svolti dal progetto in ogni iterazione
 - ⇒ **Requisiti**: fissa ciò che il sistema deve fare
 - ⇒ **Analisi**: mette a punto i requisiti e li struttura
 - ⇒ **Progettazione**: concretizza i requisiti in un'architettura del sistema
 - ⇒ **Implementazione**: costruisce il software
 - ⇒ **Test**: verifica che l'implementazione rispetti i requisiti
 - ⇒ **Deployment**: descrive la configurazione del sistema
 - ⇒ **Gestione configurazione**: mantiene le versioni del sistema
 - ⇒ **Gestione progetto**: descrive le strategie per gestire un processo iterativo
 - ⇒ **Ambiente**: descrive le infrastrutture di sviluppo

63

Fasi

- Le fasi sono sequenziali, e corrispondono a milestone significativi per committenti, utenti, management
 - ⇒ **Inception (avvio)**: definisce gli obiettivi del progetto, ne investiga la fattibilità, ne stima i costi, il potenziale di mercato e i rischi, analizza i prodotti concorrenti
 - ⇒ **Elaboration**: pianifica il progetto e ne definisce le caratteristiche funzionali, strutturali e architettoniche
 - ⇒ **Construction**: sviluppa il prodotto attraverso una serie di iterazioni, effettua il testing, prepara la documentazione
 - ⇒ **Transition**: consegna il sistema agli utenti finali (include marketing, installazione, configurazione, formazione, supporto, mantenimento)
- Ogni fase può essere composta da una o più iterazioni; il numero esatto dipende dalle scelte del Project Manager e dai rischi del progetto

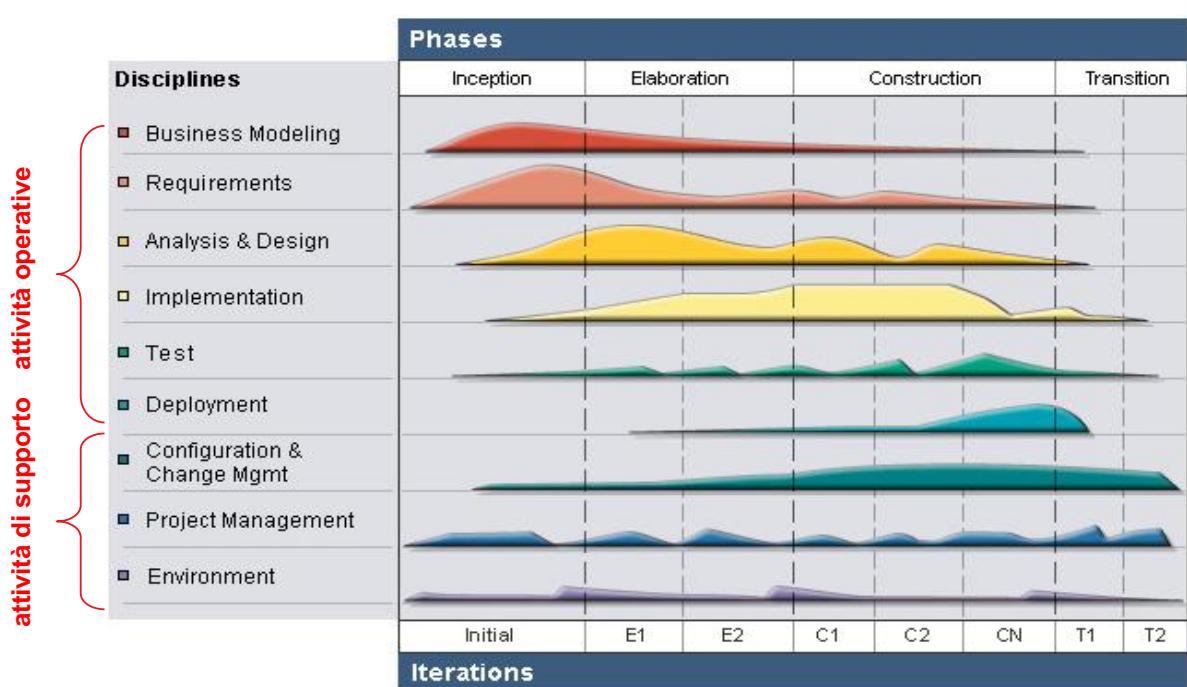
64

Milestone

- Inception
 - ⇒ Documenti fattibilità
- Elaboration
 - ⇒ Specifica dei requisiti software
 - ⇒ Architettura consolidata e verificata
- Construction
 - ⇒ Versione sistema in pre-produzione (Beta)
- Transition
 - ⇒ Versione sistema in produzione

65

Fasi e flussi di lavoro



66

4

Verifica del software

La fase di verifica del software ha lo scopo di controllare se il sistema realizzato risponde alle specifiche di progetto. La verifica non coinvolge solo il prodotto finale ma segue passo per passo il progetto e lo sviluppo del prodotto

Le tecniche di verifica del sw possono essere classificate come:

- **Dinamiche o di testing**: il corretto funzionamento del sistema viene controllato sulla base di prove sperimentali che ne verifichino il comportamento in un insieme rappresentativo di situazioni. Sono le più utilizzate nella pratica.
- **Statiche o di analisi**: il corretto funzionamento del sistema viene verificato analizzando direttamente la struttura dei moduli e il codice che li realizza. Sono applicabili durante l'intero ciclo di vita.

67

Testing

“Le operazioni di testing possono individuare la presenza di errori nel software ma non possono dimostrarne la correttezza” (Dijkstra 1972)

Scopo del testing è quello di verificare il comportamento del sistema in un insieme di casi sufficientemente ampio da rendere plausibile che il suo comportamento sia analogo anche nelle restanti situazioni.

Vista l'impossibilità pratica di verificare un sistema in tutte le possibili circostanze (*testing esaustivo*) è necessario individuare dei criteri per la selezione dei casi significativi.

Le operazioni di testing si suddividono in:

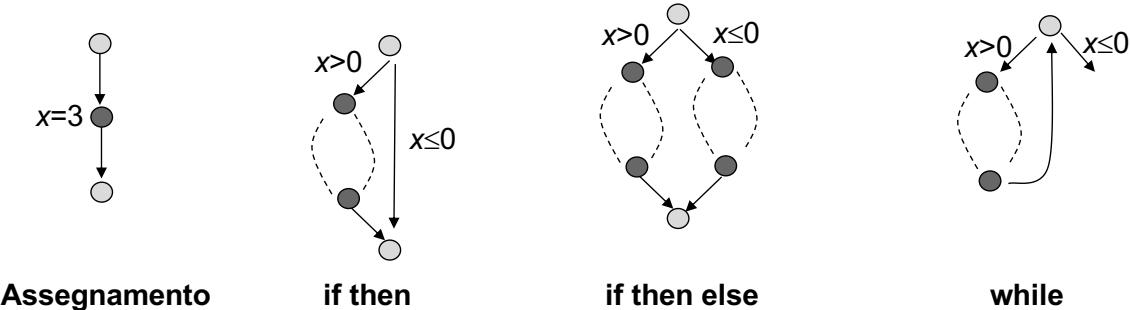
- **Testing in the small**: riguardano moduli singoli e porzioni specifiche del codice che rivestono una particolare importanza o che hanno una particolare complessità
- **Testing in the large**: riguardano il sistema nella sua globalità

68

Testing in the small

Valuta il corretto funzionamento di una porzione del codice analizzando in modo approfondito il suo comportamento in relazione all'input.

Grafi di controllo



69

Testing in the small

Criterio di copertura dei programmi (statement test)

“Selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T , ogni istruzione elementare di P venga eseguita almeno una volta”

- Si basa sull'osservazione che un errore non può essere scoperto se la parte di codice che lo contiene non viene eseguita almeno una volta
- Può essere eseguito solo conoscendo la struttura interna della porzione di codice (**white-box testing**)

```
read(x);
read(y);
if x!=0 then x:=x+10;
y:=y/x;
.....
```

test={($x=20, y=30$)}

70

Testing in the small (2)

Criterio di copertura delle decisioni (branch test)

“Selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T , ogni arco del grafo di controllo di P sia attraversato almeno una volta”

- Il criterio richiede che per ogni condizione presente nel codice sia utilizzato un test che produca il risultato TRUE e FALSE
- Si basa sul flusso di controllo e non sull'insieme di istruzioni
- Può essere eseguito solo conoscendo la struttura interna della porzione di codice (*white-box testing*)

```
read(x);
read(y);
if (x=0 or y>0)
    then y:=y/x;
    else x:=y+2/x;
....
```

test={(x=5, y=5), (x=5, y=-5)}

71

Testing in the small (3)

Criterio di copertura delle decisioni e delle condizioni

“Selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T , ogni arco del grafo di controllo di P sia attraversato e tutti i possibili valori delle condizioni composte siano valutati almeno una volta”

- Il criterio richiede che, per ogni porzione di condizione composta presente nel codice, sia utilizzato un test che produca il risultato TRUE e FALSE.
- Il criterio produce un'analisi più approfondita rispetto al criterio di copertura delle decisioni
- Può essere eseguito solo conoscendo la struttura interna della porzione di codice (*white-box testing*)

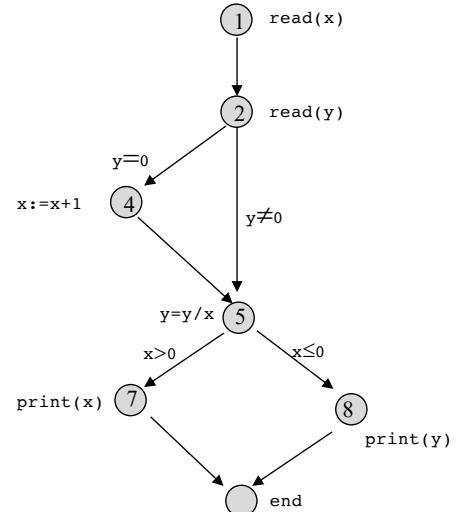
72

Testing in the small (4)

```
1. read(x);
2. read(y);
3. if (y=0)
4.   then x:=x+1;
5. y:=y/x;
6. if (x>0)
7.   then print(x);
8. else print(y);
```

test1={(x=2, y=10), (x=-2, y=0)}

test2={(x=2, y=10), (x=-1, y=0)}



Entrambi i test set soddisfano il criterio di copertura delle decisioni e delle condizioni, ma solo *test2* permette di individuare l'errore.

73

Testing in the large

L'esplosione combinatoria delle possibili situazioni che si hanno quando si esaminano sistemi di grandi dimensioni rende impossibile l'utilizzo di tecniche white-box.

Si rende quindi necessario valutare il funzionamento del sistema sulla base delle corrispondenze input-output. Il sistema è considerato una scatola nera (**black-box testing**).

L'insieme di test da utilizzare viene selezionato sulla base delle specifiche di progetto che permettono di definire i diversi valori di input e i corrispondenti valori in output.

"Il programma riceve come input una fattura di cui è nota la struttura dettagliata. La fattura deve essere inserita in un archivio ordinato per data. Se esistono altre fatture con la stessa data fa fede l'ordine di arrivo. È inoltre necessario verificare che: 1) il cliente sia già stato inserito in archivio, vi sia corrispondenza tra la data di inserimento del cliente e quella della fattura,"

Test Set

- 1) Fattura con data odierna
 - 2) Fattura con data passata e per la quale esistono altre fatture
 - 3) Fattura con data passata e per la quale non esistono altre fatture
 - 4) Fattura il cui cliente non è stato inserito.
-

74

Testing in the large (2)

Test di modulo: verifica se un modulo è stato implementato correttamente in base al suo comportamento esterno

Test d'integrazione: verifica il comportamento di sottoparti del sistema sulla base del loro comportamento esterno. Viene solitamente svolto simulando il comportamento dei moduli che producono l'input del sottosistema in analisi

Test di sistema: verifica il comportamento dell'intero sistema sulla base del suo comportamento esterno

Il test d'integrazione permette di:

- 1) Anticipare la scoperta di eventuali errori alla fase di sviluppo
- 2) Semplificare la ricerca degli errori poiché questi risultano circoscritti alla sottoporzione in esame
- 3) Permettere il rilascio di sottoparti autonome del sistema

75

Analisi del software

Analizzare un software significa ispezionarne il codice per capirne le caratteristiche e le funzionalità.

- Può essere effettuata sul codice oppure su pseudocodice.
- Permette la verifica di un insieme di esecuzioni mentre il testing verifica singoli casi
- È soggetta agli errori di colui che la effettua
- Si basa su un modello della realtà e non su dati reali

I due principali approcci all'analisi del software sono **Code walk-through** e **Code inspection**

76

Code walk-through

È un tipo di analisi informale eseguita da un team di persone che dopo aver selezionato opportune porzioni del codice e opportuni valori di input ne simulano su carta il comportamento.

- Il numero di persone coinvolte deve essere ridotto (3~5)
- Il progettista deve fornire in anticipo la documentazione scritta relativa al codice
- L'analisi non deve durare più di alcune ore
- L'analisi deve essere indirizzata solamente alla ricerca dei problemi e non alla loro soluzione
- Al fine di aumentare il clima di cooperazione all'analisi non devono partecipare i manager

77

Code inspection

L'analisi, eseguita da un team di persone e organizzata come nel caso del code walk-through, mira a ricercare classi specifiche di errori. Il codice viene esaminato controllando soltanto la presenza di una particolare categoria di errore, piuttosto che simulando una generica esecuzione.

Le classi di errori che vengono solitamente ricercate con questa tecnica sono:

- Uso di variabili non inizializzate
- Loop infiniti
- Letture di dati non allocati
- Deallocazioni improprie di memoria

78

Analisi di flusso dei dati

- Un tipo particolare di code inspection
- L'analisi dell'evoluzione del valore associato alle variabili durante l'esecuzione di un programma è intrinsecamente dinamica. Ciononostante, alcuni aspetti di questo problema possono essere analizzati anche staticamente
- A ogni comando è possibile associare staticamente il tipo di operazioni eseguite sulle variabili:
 - ⇒ *definizioni* (d)
 - ⇒ *usi* (u)
 - ⇒ *annullamenti* (a)
- Sequenze di comandi, corrispondenti a possibili esecuzioni, sono riducibili staticamente a sequenze di tali operazioni

79

Analisi di flusso dei dati: esempio

```
1 procedure swap (x1, x2: real)
2 var x: real;
3 begin
4   x2 := x;
5   x2 := x1;
6   x1 := x;
7 end;
```

- ⇒ Per la variabile x, la sequenza (assegnamenti 4,5,6) può essere ridotta a:
 - un annullamento (il valore associato alla variabile x non è infatti definito al momento dell'attivazione del sottoprogramma)
 - un uso (linea 4)
 - un secondo uso (linea 6)

La sequenza di operazioni sulla variabile x può quindi essere riassunta con la stringa **auu**

- ⇒ Per la variabile x1 la sequenza di operazioni corrispondenti può essere riassunta dalla stringa **dud** (è uno dei parametri formali della procedura e quindi il valore è definito al momento della chiamata)
- ⇒ Per la variabile x2, la sequenza di operazioni corrispondenti è **ddd**

80

Analisi di flusso dei dati

- L'esame delle sequenze ottenute per ogni variabile può rilevare la presenza di anomalie
 - ⇒ La sequenza **aau**, ad esempio, ottenuta per la variabile x permette di dedurre che il valore usato nei due comandi di assegnamento alle linee 4 e 6 non è definito, i due usi della variabile sono infatti preceduti da un annullamento
- In generale, ogni sequenza contenente un uso non preceduto da una definizione senza annullamenti intermedi è sintomo di una possibile anomalia dovuta all'uso di valori non definiti
 - ⇒ Nel programma swap, che dovrebbe scambiare il contenuto dei parametri x1 e x2 facendo uso di una variabile locale x, le variabili x ed x2 nell'assegnamento di linea 4 ($x2 := x;$) sono state erroneamente invertite
 - ⇒ Anche la sequenza **ddd** ottenuta per la variabile x2 permette di rilevare l'anomalia nel programma swap: il valore associato a x2 all'atto della chiamata non è usato prima di essere sostituito da un nuovo valore, è quindi assegnato inutilmente alla variabile
- In generale, ogni sequenza contenente due definizioni consecutive è sintomo di una possibile anomalia

81

Analisi di flusso dei dati

- Regole generali, la cui violazione permette di dedurre la presenza di possibili anomalie nel programma:
 1. L'uso di una variabile x deve essere sempre preceduto in ogni sequenza da una definizione della stessa variabile x, senza annullamenti intermedi
 - ⇒ Un uso non preceduto da una definizione corrisponde infatti al potenziale uso di un valore non determinato di una variabile; al momento dell'uso, infatti, il valore della variabile non è ancora stato definito. Allo stesso modo se tra l'uso e la precedente definizione compare un annullamento, il valore della variabile non è definito all'atto del uso
 2. Una definizione di una variabile x deve sempre essere seguita da un uso della variabile x, prima di un'altra definizione o di un annullamento della stessa variabile x
 - ⇒ Una definizione non seguita da un uso prima di ulteriori definizioni o annullamenti della variabile corrisponde all'assegnamento di un valore non successivamente utilizzato e quindi potenzialmente inutile. Ciò può essere sintomo di un'anomalia dovuta all'omissione del comando che avrebbe dovuto usare il valore assegnato alla variabile.

82

D deve essere sempre seguito da U e U deve essere sempre preceduto da D

Analisi di flusso dei dati

□ Esempio:

- ⇒ **adudu**, **duadudu** sono legali secondo le regole (1) e (2)
- ⇒ **aduddu**, **dauduu**, **duaudu** non soddisfanno invece le regole (1) e (2)
 - Nella prima compaiono due definizioni consecutive, contrariamente a quanto richiesto dalla regola (2)
 - Nella seconda tra il primo uso e la precedente definizione compare un annullamento, la regola (1) non è quindi soddisfatta
 - Nella terza, è interposto un annullamento tra un uso (il secondo uso della sequenza) e la definizione precedente (la prima definizione della sequenza)

83

Analisi di flusso dei dati

□ Non tutte le sequenze **au** e **dd** corrispondono necessariamente ad anomalie:

- ⇒ La sequenza **au** può per esempio comparire in un generatore di numeri casuali, che legge il contenuto non inizializzato di una cella di memoria per determinare il seme della generazione
- ⇒ La sequenza **dd** può essere dovuta ad una cattiva strutturazione del programma, per cui la prima definizione della sequenza non è usata nell'esecuzione considerata, ma lo è in un'altra esecuzione, che richiede la percorrenza di un cammino diverso:

```
1 .....
2 x := .....
3 if .... then x := .....
4 ... := ...x...
5 .....
```

84

5

La certificazione

La Comunità Europea basa il sistema di controllo della qualità nei diversi settori della produzione su due distinte classi di regole:

Regole tecniche: sono emesse dalla pubblica amministrazione e dagli organi dello Stato (leggi, decreti e regolamenti) nel rispetto delle regole comunitarie (*direttive*); la loro osservanza ha carattere obbligatorio

Norme tecniche consensuali: sono elaborate e pubblicate dagli organismi di normazione riconosciuti con la collaborazione anche di rappresentanti governativi e possono avere validità nazionale (UNI e CEI), europea (EN), internazionale (ISO e IEC). La loro applicazione non è obbligatoria, ma può essere imposta da opportune direttive, leggi o regolamenti

Le norme tecniche consensuali rappresentano una razionalizzazione dei molti approcci utilizzati tra clienti e fornitori per garantire la qualità del prodotto o del servizio

La famiglia di norme ISO 9000 ricade tra le norme tecniche consensuali

85

La normativa ISO 9000

Gli obiettivi primari delle norme della famiglia ISO 9000 sono due:

Gestione per la qualità: offrono una guida alle aziende che desiderano progettare e attuare un efficace sistema qualità nella loro organizzazione o migliorare il sistema di qualità esistente. Uno degli obiettivi principali della gestione per la qualità è il miglioramento delle attività e dei processi

Assicurazione della qualità: definiscono i requisiti generali a fronte dei quali un cliente valuta l'adeguatezza del sistema qualità del fornitore, nonché la sua capacità di soddisfare i requisiti stabiliti

86

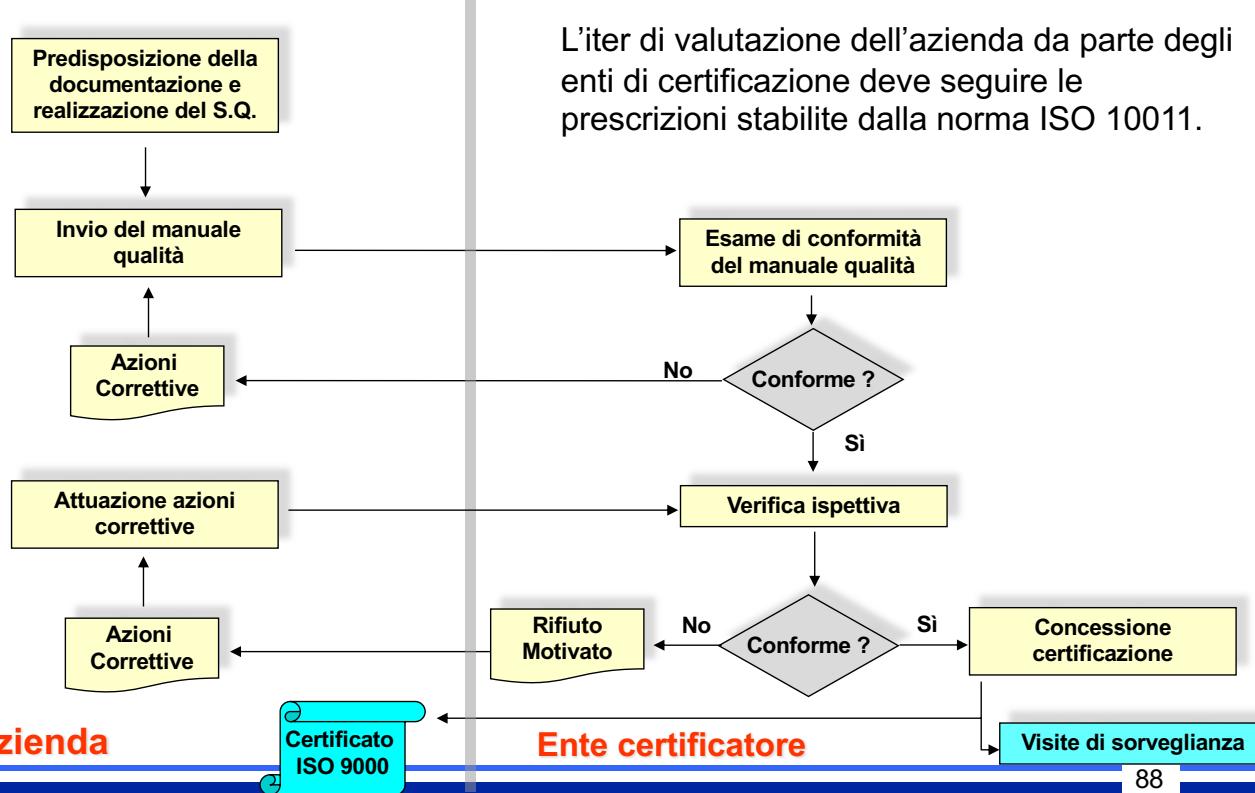
La certificazione ISO 9000

Con il termine **certificazione** si intende l'atto mediante il quale un **organismo di certificazione accreditato** a livello nazionale o internazionale dichiara che, con un livello confidenziale attendibile, un determinato prodotto, processo, servizio o sistema qualità aziendale è conforme a una specifica norma o documento normativo a essa applicabile

L'**accreditamento** è definito come il riconoscimento formale di idoneità di un laboratorio a effettuare specifiche prove o determinati tipi di prova. L'uso del termine è ora esteso al processo di riconoscimento delle competenze degli organismi di certificazione

87

La certificazione ISO 9000



88

La certificazione ISO 9000

La certificazione ISO 9000 può essere rilasciata anche su sotto-porzioni dell'azienda oggetto di valutazione. La definizione dell'estensione dell'analisi e quindi la scelta degli elementi e dei processi oggetto di verifica sono un problema controverso e motivo di discussione tra le parti. È compito dell'organismo di certificazione, in collaborazione con l'azienda da certificare, definire i confini e l'estensione della certificazione.

Il **manuale qualità** comprende la specifica di tutti i processi su cui è applicato il sistema qualità. Esso comprende inoltre la descrizione di tutta la documentazione che viene redatta a supporto di tale sistema.

La **verifica ispettiva** è la fase fondamentale della certificazione. Durante la visita ispettiva i valutatori interpellano, secondo un piano concordato in base alle caratteristiche dell'azienda e al risultato dell'analisi del manuale qualità, vari responsabili aziendali le cui funzioni hanno impatto sulla qualità (direzione generale, ufficio acquisti, direzione tecnica, responsabili laboratori, ecc.). Vengono quindi visitati i reparti in cui si svolgono le attività ed effettuate le verifiche relative alla corretta applicazione delle procedure aziendali.

A certificazione avvenuta l'organismo di certificazione effettua visite di mantenimento della certificazione (**visite di sorveglianza**). La frequenza delle visite di sorveglianza non è uguale per tutti gli organismi e può variare da 1 a 4 all'anno.

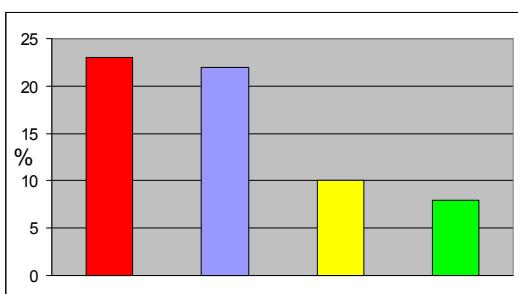
89

La certificazione ISO 9000

Non conformità secondo la norma è il “non soddisfacimento di requisiti specificati”. La definizione riguarda lo scostamento o l'assenza di una o più caratteristiche di qualità o di elementi del sistema qualità rispetto ai requisiti specificati.

Le non conformità che vengono riscontrate durante le verifiche di certificazione possono essere di vario tipo:

- **Non conformità rispetto ai requisiti della norma**: qualora uno o più requisiti della norma non vengano rispettati;
- **Non conformità sulla documentazione**: attività non procedurali o formalizzate in documenti, oppure attività che non seguono le prescrizioni delle procedure;
- **Non conformità sull'attuazione delle procedure**: qualora le prescrizioni delle procedure non vengano attuate in modo efficace.



Principali non conformità per le software house

- Non c'è evidenza sul ciclo di sviluppo
- Parte della manualistica che descrive la metodologia di sviluppo non è conforme
- Le procedure di controllo della sub-fornitura dello sviluppo del software non sono conformi
- La politica per la qualità non risulta compresa e attuata appieno

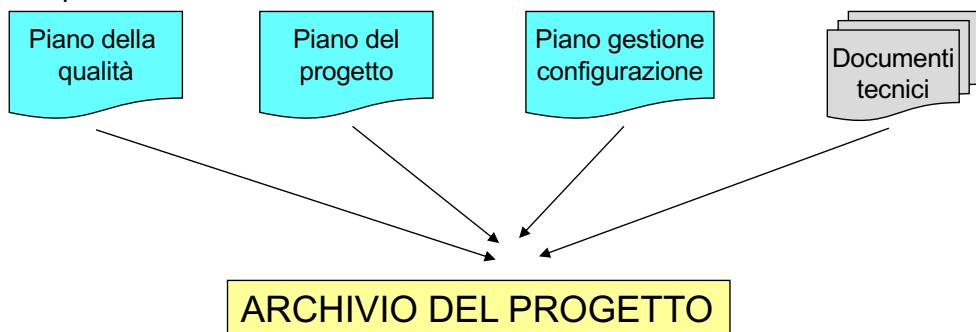
90

I documenti del progetto

La gestione della qualità si realizza tramite la standardizzazione di tutte le operazioni che riguardano il processo. Elemento primario a tale fine è l'insieme dei documenti che costituiscono l'archivio del progetto. La concessione della certificazione ISO 9000 si basa in gran parte sulla correttezza di tale documentazione.

I documenti del progetto si dividono in:

- Documenti tecnici
- Documenti di pianificazione



91

6

La manutenzione

Correttiva

→ Rimedia ai malfunzionamenti provocati dai difetti derivanti da errori di analisi, progettazione, codifica, test

Adattiva

→ mantiene inalterato il livello di servizio del sistema al mutare delle condizioni operative

Perfettiva

→ migliora qualitativamente le caratteristiche funzionali o tecniche del sistema

Evolutiva

→ migliora qualitativamente e quantitativamente le caratteristiche del sistema

92

Manutenzione correttiva

- Errori:
 - ⇒ commessi dall'uomo
 - ⇒ interessano tutte le fasi del ciclo di sviluppo
- Difetti:
 - ⇒ si riscontrano nei programmi
 - ⇒ si manifestano nella produzione di risultati sbagliati
- Malfunzionamenti:
 - ⇒ interessano i sistemi
 - ⇒ ne compromettono l'intera funzionalità
 - ⇒ ne annullano il valore informativo

93

Manutenzione correttiva

- Esempi:
 - ⇒ tutto ciò che fa “andare male” i programmi
- Costi:
 - ⇒ altissimi (40%), soprattutto quando non si risale oltre i difetti
- Risultati:
 - ⇒ aumento dell'entropia del programma
 - ⇒ degrado del sistema
 - ⇒ abbattimento dell'affidabilità
 - ⇒ ripristino della qualità

94

Manutenzione adattiva

- Esempi:
 - ⇒ ricalcolo tasse e imposte
 - ⇒ aggiornamento/gestione listini e tariffari
 - ⇒ modifiche a routine di calcolo
- Costi:
 - ⇒ alti (20-30%) ma spesso imputati allo sviluppo
- Risultati:
 - ⇒ nessun aumento dei valori informativi del sistema
 - ⇒ ripristino della qualità

95

Manutenzione perfettiva

- Esempi:
 - ⇒ ricerca performance
 - ⇒ estensioni funzioni applicative
 - ⇒ nuove interfacce
 - ⇒ modifiche architetturali
- Costi:
 - ⇒ quasi sempre imputati allo “sviluppo”
- Risultati:
 - ⇒ aumento del valore informativo del sistema
 - ⇒ aumento dell'utilizzabilità
 - ⇒ aumento della complessità
 - ⇒ spesso, degrado della qualità

96

Manutenzione evolutiva

- Esempi:
 - ⇒ nuove funzioni “embedded” nei vecchi programmi
 - ⇒ passaggio da interfaccia a caratteri a interfaccia a finestre
 - ⇒ passaggio da file indexed a DBMS relazionali
 - ⇒ potenziamento reporting
- Costi:
 - ⇒ alti (spesso nascosti fino all'esercizio)
- Risultati:
 - ⇒ aumento tendenziale dell'entropia
 - ⇒ diminuzione della robustezza del sistema
 - ⇒ aumento della potenza
 - ⇒ aumento della qualità, ma solo se si progetta bene l'operazione!

97

Manutenzione

Man. correttiva	Man. adattiva	Man. perfettiva	Man. evolutiva
freni rotti	pieno di benzina	condizionatore	gancio traino
scheda bruciata	cambio batterie	nuove casse	lettore CD
serratura guasta	imbiancatura	nuovi mobili	una stanza in più
mal di denti	alimentazione	jogging	corso di inglese



98

Manutenzione

- Ciò che rende un sistema manutenibile è la sua architettura originale

- ⇒ Automobili

- funzione stabile nel tempo
 - costruzione modulare
 - architettura "smontabile"
 - progettazione mirata all'intervento successivo



- ⇒ Hi-Fi

- architettura a "moduli" (schede)
 - componenti legate strettamente
 - progettazione poco mirata (meglio buttare)



- ⇒ Casa

- costruzione non modulare
 - elementi nascosti (tubi, cavi)
 - progettazione mirata all'impianto iniziale



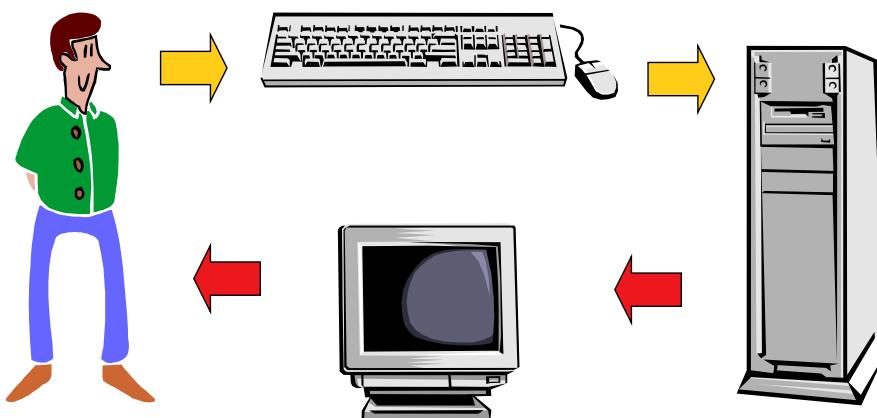
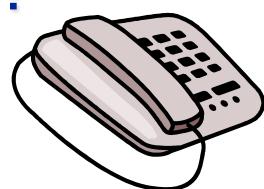
- ⇒ *Spesso i programmi sono costruiti come case ma vengono utilizzati e debbono essere mantenuti come automobili*

Progettazione di Interfacce Utente

1

Cos'è un'interfaccia?

- Nel gergo generale...
 - ⇒ ...permette il dialogo tra due entità (*partner*)
- Nel gergo elettronico...
 - ⇒ ...permette il transito di informazione tra due dispositivi o sistemi



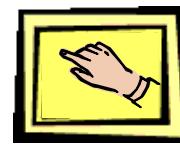
2

Tecnologia vs. ergonomia

- Sul lato fisico...
 - ⇒ ... tecnologia



- Sul lato cognitivo
 - ⇒ ...ergonomia cognitiva



3

I paradigmi di interazione

- Terminale scrivente: **scrivi e leggi**
- Terminale video: **scegli e riempি**
- Personal Computer: **what if**
- Sistemi multimediali: **parla ed ascolta**
- Realtà virtuale: **entra ed agisci**

4

Dalla parte dell'utente

- Le GUI esaltano le potenzialità del cervello umano:
 - ⇒ riconoscere e associare
 - ⇒ generalizzare e dedurre
- Come:
 - ⇒ molte informazioni contemporaneamente
 - ⇒ metafore
 - ⇒ colore

5

Riconoscere o ricordare?

- Lista di codici registrazione ordinazioni al ristorante, facili da ricordare e raggruppati per significato.
 1. Leggerli per 30 secondi
 2. Chiudere le dispense e cercare di riscriverli correttamente anche se in un qualunque ordine

FISS

SPAG

BRAC

ACQU

CART

RISO

POLL

VINO

SELF

BROD

PESC

LASA

BOLL

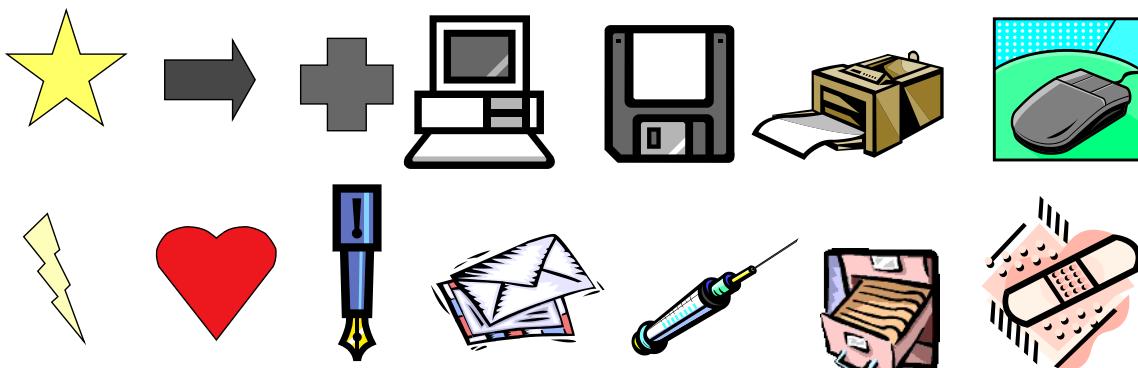
CONI

PUNTEGGIO: 11/14

6

Riconoscere o ricordare?

- Un insieme di icone d'aspetto familiare raggruppate per significato
 1. Osservarle per 30 secondi
 2. Chiudere le dispense e cercare di riscriverne i nomi (secondo la propria interpretazione) in un qualunque ordine



PUNTEGGIO: 11/14

7

Riconoscere o ricordare?

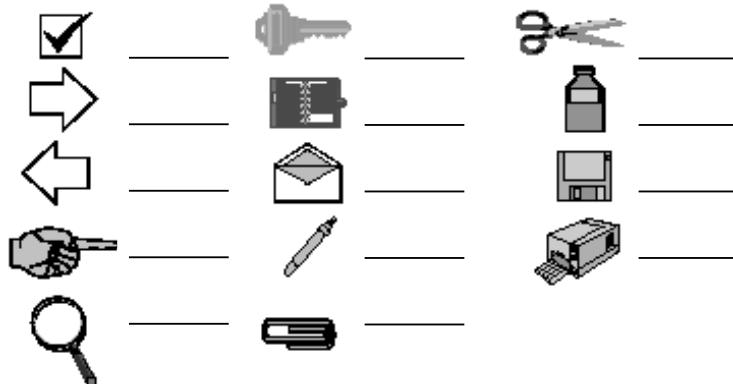
- Un insieme di icone che associano oggetti del mondo reale alle più diffuse funzioni computerizzate
 1. Osservarle per 30 secondi
 2. Voltare pagina e cercare di riscrivere il nome di ciascuna funzione accanto all'immagine dell'icona corrispondente



8

Riconoscere o ricordare?

- Un insieme di icone che associano oggetti del mondo reale alle più diffuse funzioni computerizzate
 1. Osservarle per 30 secondi
 2. Voltare pagina e cercare di riscrivere il nome di ciascuna funzione accanto all'immagine dell'icona corrispondente

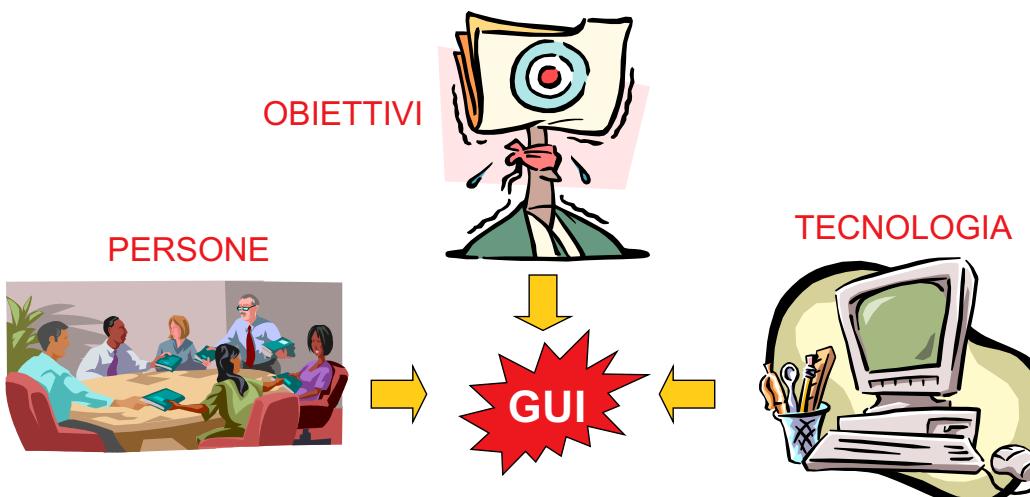


PUNTEGGIO: 14/14

9

Fattori chiave nel progetto dell'interfaccia

- Chi userà l'interfaccia?
- Per cosa l'interfaccia verrà usata?



10

Scelta della tecnologia per l'interfaccia

- ...**in funzione degli obiettivi**:
 - ⇒ rapidità o efficacia
 - ⇒ che cosa è la qualità e quanto è importante
 - ⇒ change management
 - ⇒ utilizzo di strumenti di produttività individuale
 - ⇒ strategie a lungo o a breve termine (elevato o modesto investimento)
- ...**in funzione degli utenti**:
 - ⇒ numero d'utenti
 - ⇒ esperienza nell'utilizzo della tecnologia
 - ⇒ età media
 - ⇒ motivazione o scetticismo
 - ⇒ eterogeneità dei gruppi d'appartenenza
 - ⇒ turnover
 - ⇒ utilizzatori assidui o saltuari
 - ⇒ versioni standard o ad hoc

11

Interfacce code-based

- Interazione attraverso comandi (codici)
 - ⇒ Ottimale per moli di lavoro elevate che richiedono attenzione in punti lontani dal video (es. Check-In in aeroporto)
 - ⇒ Occorre mantenere basso il numero di codici utilizzabili
 - ⇒ Nessuna riusabilità delle conoscenze acquisite

Mole di lavoro da svolgere	Qualità	Facilità di apprendimento	Riutilizzo conoscenza	Soddisfazione
😊	😐	🙁	🙁	🙁

```
> copy utenti.txt D:  
> print utenti.txt  
> delete utenti.txt
```

12

Interfacce 3270

□ Interfaccia a caratteri

- ⇒ Ottimale per data-entry ed editing di dati altamente strutturati
- ⇒ Workflow fortemente predefinito (bassa flessibilità)
- ⇒ Navigazione e tasti funzionali complicano apprendimento e riusabilità delle conoscenze acquisite

Mole di lavoro da svolgere	Qualità	Facilità di apprendimento	Riutilizzo conoscenza	Soddisfazione

NOME : _____

COGNOME : _____

SESSO : _____

RESIDENZA : _____

13

Pseudo-GUI

□ Interfaccia grafica che richiama la strutturazione di un'interfaccia a caratteri

- ⇒ Ottimale per applicazioni che debbano gestire dati fortemente strutturati garantendo una buona flessibilità
- ⇒ Se standard consente riusabilità delle conoscenze acquisite

Mole di lavoro da svolgere	Qualità	Facilità di apprendimento	Riutilizzo conoscenza	Soddisfazione

Anagrafica Clienti

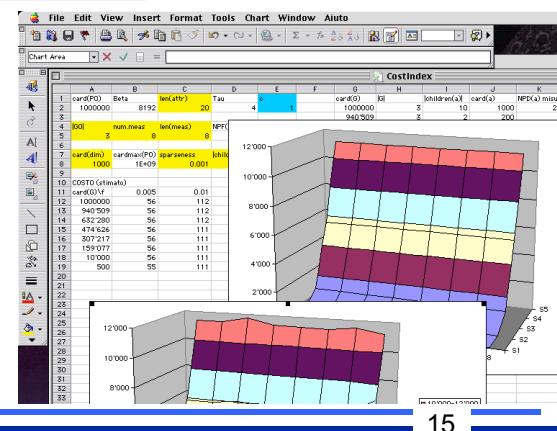
Codice:	<input type="text"/>	Tipo:	<input type="radio"/> Potenziale	<input type="radio"/> Acquisto
Cognome:	<input type="text"/>	Settore Mercologico:	<input type="text"/>	
Nome:	<input type="text"/>		<input type="button" value="▼"/>	
Ruolo:	<input type="text"/>			
Società:	<input type="text"/>			
Indirizzo:	<input type="text"/>			
C.A.P.:	<input type="text"/>	Provincia:	<input type="text"/>	
Città:	<input type="text"/>			

14

Standard GUI

- Progettata e sviluppata per un ambiente grafico
 - ⇒ Esaltate le potenzialità di manipolazione diretta (cut & paste, drag & drop, etc.)
 - ⇒ Ottimale per user-driven applications (flessibilità)

Mole di lavoro da svolgere	Qualità	Facilità di apprendimento	Riutilizzo conoscenza	Soddisfazione
				



Special GUI

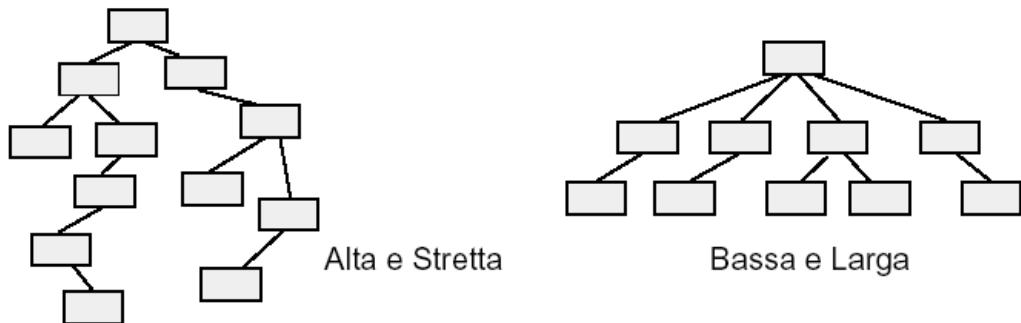
- Enfasi massima alla presentazione grafica
 - ⇒ Obiettivo prioritario è l'autoespliсazione (EIS, videogames)
 - ⇒ Il cliente “si serve” da solo...
 - ⇒ L’utente target potrebbe non avere esperienza sull’utilizzo dei computer

Mole di lavoro da svolgere	Qualità	Facilità di apprendimento	Riutilizzo conoscenza	Soddisfazione



Strutturazione

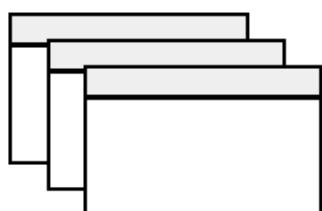
- Una struttura “bassa e larga” fornisce all’utente una visione migliore delle possibilità offerte e facilita la navigazione



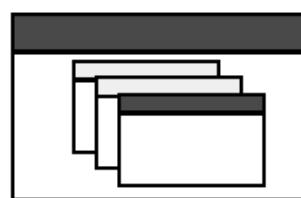
17

Strutture di riferimento

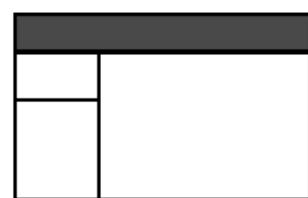
Multi-Window



Multi-Document (MDI)



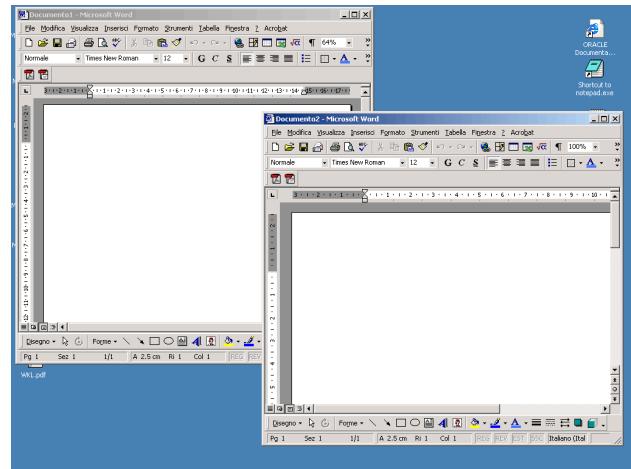
Multi-Paned



18

Modello multi-window

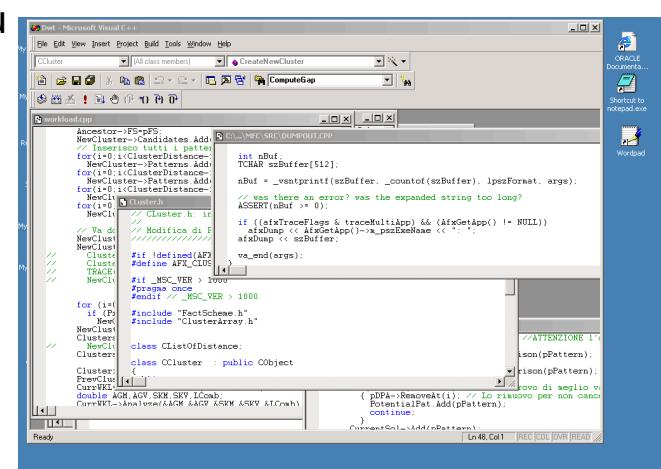
- ⇒ molte *main window* (ciascuna con un menu)
- ⇒ rapporto 1:1 tra *main window* e *business object*
- ⇒ molte *child windows* (senza menu) possibili per ciascuna *main window*
- Più *main window* attivabili contemporaneamente: estrema flessibilità
- Navigazione complessa



19

Modello multi-document

- ⇒ una sola *top window* con menu
- ⇒ la *top window* guida una serie di *document window*
- ⇒ la *top-window* deve sempre rimanere aperta
- Flessibilità inferiore a quella del multi-window model
- Vi sarà sempre un solo menu attivabile
- Ottimale anche per utenti inesperti



20

Modello multi-paned

- ⇒ una “window” alla volta con o senza menu
- ⇒ eventuale suddivisioni in aree (*pane*) monofunzionali e monoposizionali
- Assenza di flessibilità
- Per special GUI in applicazioni self-service



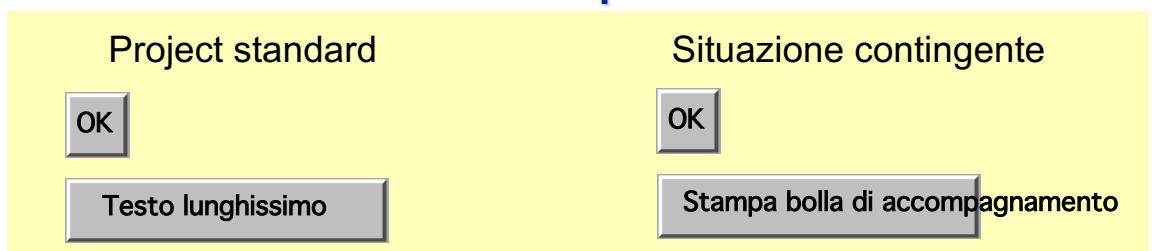
21

Project standard

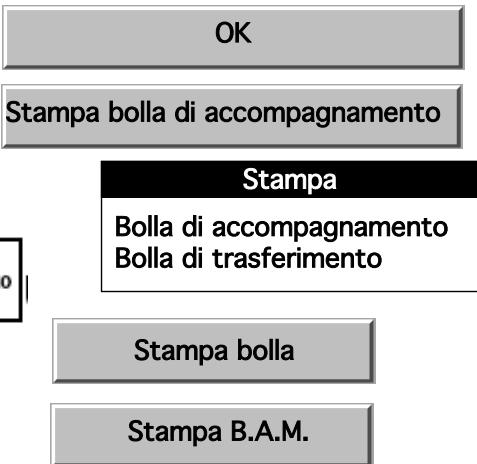
- Definizione degli standard per:
 - ⇒ terminologia
 - ⇒ metafore, icone
 - ⇒ caratteristiche delle finestre (menu, bottoni, dimensioni, posizione, ecc.)
- Obiettivo prioritario: agevolare l'utilizzo da parte dell'utente
 - ⇒ consistenza esterna
 - ⇒ i tool già utilizzati in azienda (standard de facto)
 - ⇒ consistenza interna subordinata all'usabilità

22

Esempio



- Allargare tutti i bottoni della window
- Allargare solo il bottone “incriminato”
- Ridisegnare la window ed inserire la scelta nel menù
- Un simbolo al posto del testo
- Testo più corto compreso ed approvato dall’utente
- Abbreviazione compresa ed approvata dall’utente



23

Esempio

- Priorità consigliate**
 1. Testo più corto compreso ed approvato dall’utente
 2. Abbreviazione compresa ed approvata dall’utente
 3. Allargare tutti i bottoni della window/gruppo
 4. Allargare solo il bottone “incriminato”
 5. Un simbolo al posto del testo
 6. Ridisegnare la window ed inserire la scelta nel menù

24

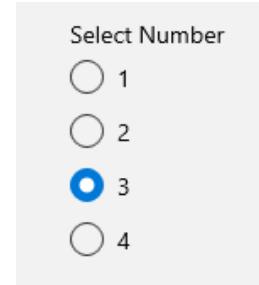
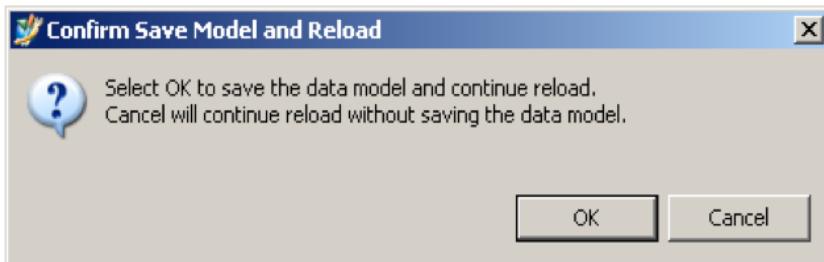
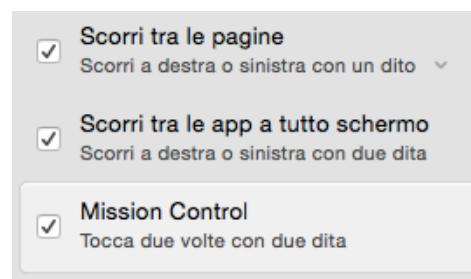
Comunicazione visiva nelle GUI

- **Affordance**: enfatizza gli aspetti di un oggetto che invitano a manipolarlo in un certo modo
- **Metafora**: una parola, una frase o una figura che dipinge un oggetto o un concetto attraverso una somiglianza o un'analogia con un altro oggetto o concetto del mondo reale
- **Layout**: è determinato dalla posizione del testo, dei disegni e dei controlli all'interno di un'area considerata
- **Colori**: utili per focalizzare l'attenzione o per creare associazioni
- **Icone**: disegni piccoli, semplici e metaforici
- **Font**: leggibilità in relazione al tipo e alle caratteristiche del carattere

25

Affordance

- Tridimensionalità
- Ombreggiatura
- Puntamento



26

Metafore

- La prima tra le scelte progettuali...

simbolo di divieto
+
evocazione del "fumo"



Metafore comuni

Documento
Cartellina
Schedario
Scheda
Lettera
Taglia e cuci
Cestino
Bottone
Gomma

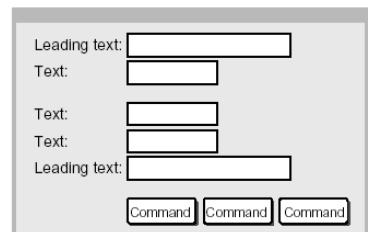
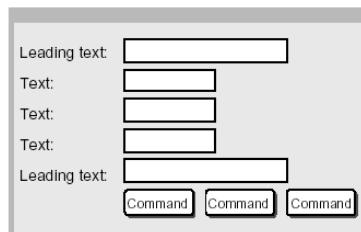
Associazioni

File
Directory
Storage System
Record
E-mail
Scrivi e leggi da un buffer
Cancella
Comando
Undo

27

Layout

- La posizione degli elementi è un importante strumento di comunicazione
 - ⇒ Le distanze devono essere scelte in relazione al grado di associazione tra gli elementi
- Fra gli standard di progetto...
 - ⇒ distanza tra campi correlati
 - ⇒ distanza tra i gruppi
 - ⇒ distanza (superiore, laterale, inferiore) tra riquadro ed elementi contenuti
 - ⇒ distanza (superiore, laterale, inferiore) tra margine dell'area principale ed elementi contenuti



28

Colori

Culture	Rosso	Blu	Verde	Bianco	Giallo
USA	Pericolo	Mascolinità	Sicurezza	Purezza	Codardia
Francia	Aristocrazia	Libertà, pace	Criminalità	Neutralità	Temporaneità
Egitto	Morte	Virtù, fede	Fertilità, forza	Gioia	Prosperità
India	Vita, creatività	Gioia, potenza	Prosperità	Purezza	Successo
Giappone	Pericolo	Malvagità	Futuro, energia	Morte	Nobiltà
Cina	Felicità	Paradiso	(Ming) Paradiso	Purezza	Nascita



dimenticatevi l'estetica, il colore è comunicazione!

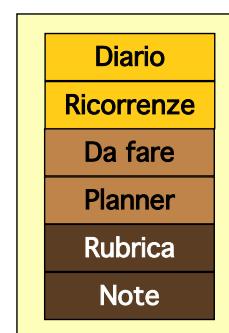
(Jan B. White)

29

Colori



decorazione



codifica

30

Colori

- Non abusare dei colori in un “ambiente” monocromatico: il risalto è eccessivo
- Se il colore è usato come codice: solo 3-5 colori, ricordarsi la semantica
- Colori vivaci per aree piccole e neutri per aree grandi
- Ricercare un contrasto efficace tra testo e sfondo
- Sfondo chiaro (bianco, grigio, giallo) è ottimale per testi scuri
- I colori troppo brillanti causano alterazione visiva sui tempi lunghi: sono pertanto sconsigliabili per applicazioni gestionali, mentre risultano ottimali nelle applicazioni self-service

31

Icone

- Struttura
 - ⇒ immagine
 - ⇒ sfondo
 - ⇒ testo (facoltativo)
- Caratteristiche
 - ⇒ Facilmente distinguibili
 - ⇒ Elevato valore informativo
 - ⇒ Presentazione esplicita della metafora
 - ⇒ Incrementano la velocità e la correttezza della selezione
 - ⇒ Autoesplicative anche se prive di testo
- Linee guida
 - ⇒ Disegni semplici e schematici
 - ⇒ Colori differenti in icone differenti
 - ⇒ Il testo è il titolo della finestra collegata
 - ⇒ Evitare i puzzle!



Piscina



Servizio
Elicotteri



Traghetti

32

Icônes

1. Desktop icon

- ⇒ Obiettivo: partenza, riapertura
- ⇒ Per applicazioni collegate per l'utente, icônes similaires graphiquement
- ⇒ Si minimise:
 - icônes similaires pour fenêtres différentes de la même application
 - le texte est fondamental pour icônes similaires représentant des fenêtres différentes
 - texte = window title

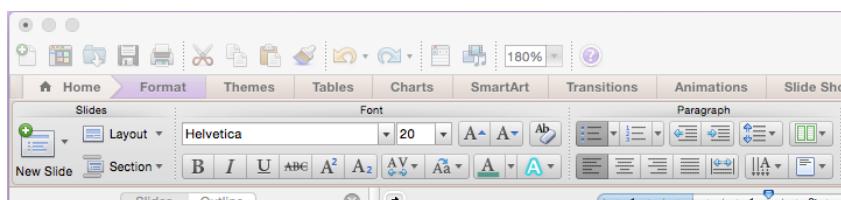


33

Icônes

2. Menu icon - Palette Icon

- ⇒ Toujours visibles à côté des menus
 - overview of functions always available
 - un modo veloce di selezionare
 - per comandi esprimibili più facilmente con disegni che con parole
 - invito alla sperimentazione



34

Icone

3. *Button icon*

- ⇒ In aggiunta al testo di un bottone
 - Rafforza graficamente la funzione del bottone



35

Font

□ Linee guida:

- ⇒ Sans Serif per singole righe
- ⇒ Serif per testi articolati su molte righe
- ⇒ Attenzione al maiuscolo
- ⇒ Spaziatura proporzionale

Questo è il font Helvetica

Questo è il font Times, più adatto per coprire più righe

ATTENZIONE
non abusare del maiuscolo

Questo è il font Courier

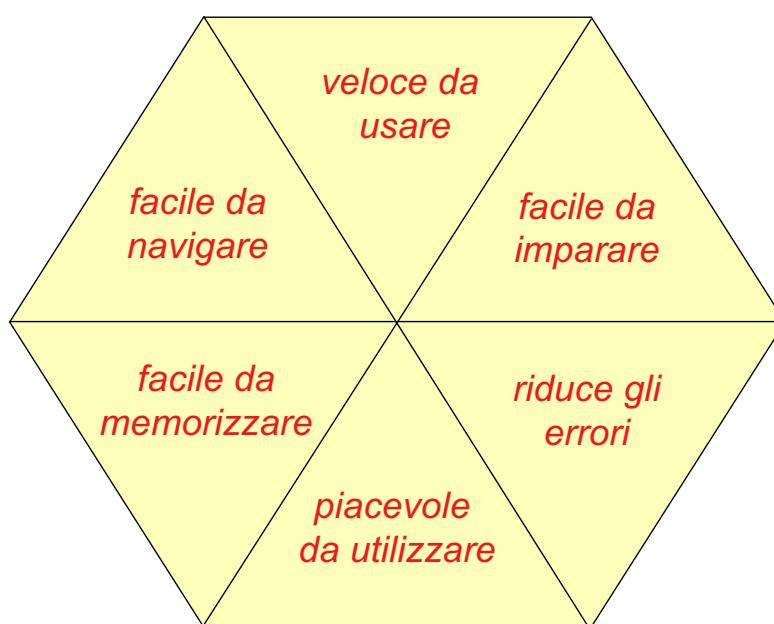
36

Usabilità

- L'efficacia, efficienza e soddisfazione con cui determinati utenti eseguono determinati compiti in particolari ambienti
 - ⇒ **Efficienza**: in che misura i compiti previsti dal funzionamento vengono eseguiti
 - ⇒ **Efficienza**: risorse da impegnare per eseguire i compiti previsti
 - ⇒ **Soddisfazione**: misura dell'accettabilità del funzionamento da parte dell'utente
- ...ma anche comprensibilità, apprendibilità, operabilità

37

Usabilità



i 6 criteri di usabilità

38

Apprendibilità

□ Obiettivo

- ⇒ 80% dei nuovi utenti in grado di svolgere compiutamente una singola attività dell'applicazione in 30 minuti

□ Quando

- ⇒ Turn-over alto
- ⇒ Utenti saltuari
- ⇒ Riduzione del training
- ⇒ Sistemi solitamente sottoutilizzati per mancanza di training
- ⇒ Breve ciclo di vita dei prodotti

39

Velocità

□ Obiettivo

- ⇒ 10 inserimenti ogni 2 minuti

□ Quando

- ⇒ Utilizzo giornaliero e intensivo
- ⇒ Attività ripetitiva

Soddisfazione

□ Obiettivo

- ⇒ 9 su 10 dichiarano che è “bello da usare”

□ Quando

- ⇒ Sistema self-service
- ⇒ Business Process Re-engineering incentrato sul nuovo sistema

40

Facilità di navigazione

□ Obiettivo

- ⇒ Possibilità di innescare 6 diverse attività su un singolo oggetto senza ritornare al menu principale

□ Quando

- ⇒ Il cliente “guida il gioco”
- ⇒ Richiami notevoli tra attività
- ⇒ Si attende una decisione... (ristorante)

FLESSIBILITÀ

RIGIDITÀ

41

Memorabilità

□ Obiettivo

- ⇒ Riutilizzo, senza ulteriore training, di una applicazione inattiva da 12 mesi

□ Quando

- ⇒ Utenti saltuari
- ⇒ Applicazioni per circostanze “eccezionali”
- ⇒ Applicazioni di utilizzo secondario
- ⇒ Applicazioni attivate in date precise (scadenze)

42

Prevenzione degli errori

□ Obiettivo

- ⇒ Riduzione della percentuale degli errori incorreggibili (catastrofici)

□ Quando

- ⇒ Risultati/prodotti ottenuti “faticosamente”
- ⇒ Risultati correlati a fattori di sicurezza
- ⇒ Risultati immediatamente visibili al cliente esterno

43

Metodologia di progetto

□ Prima del termine dello studio di fattibilità

1. definire le attività legate alla realizzazione dell’interfaccia
2. definire i parametri di riferimento ed i criteri di usabilità
3. pianificare le attività di valutazione dell’usabilità
4. realizzare il modello concettuale dell’interfaccia

□ Precocemente nella fase di analisi e progettazione

5. Definire e realizzare le strutture base (dialogo, look & feel)
6. Stabilire gli standard di progetto per l’interfaccia
7. Prototipare le parti ritenute critiche
8. Verificare l’allineamento con modello concettuale e standard

□ Nella fase di sviluppo

9. Ultimare l’interfaccia in dettaglio legandola alla logica applicativa

44

Test con l'utente

Simulatore (l'utente è passivo)

Dimostratore (l'utente agisce sulle parti critiche)

Prototipo (l'utente agisce sull'intero sistema in beta-release)

□ Elementi da verificare:

- ⇒ Il modello concettuale è sufficientemente rappresentato
- ⇒ Rispetto al progetto l'interfaccia è adatta e gli standard sono rispettati
- ⇒ Adeguato bilanciamento tra flusso predefinito e flessibilità
- ⇒ Possibilità d'utilizzo alternativo tra mouse e tastiera
- ⇒ Livello d'integrazione dell'utente con l'interfaccia
- ⇒ E' utilizzata la terminologia utente