Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma

Fondamenti di Informatica

Laurea in
Ingegneria Civile e Ingegneria per l'ambiente e il territorio

Paradigmi di Programmazione

Stefano Cagnoni e Monica Mordonini

Problema della ricorsione

Il calcolo del fattoriale

- La funzione fattoriale, molto usata nel calcolo combinatorio, è così definita

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

dove **n** è un numero intero non negativo

Il calcolo del fattoriale

- Vediamo di capire cosa significa…
  0! = 1
  1! = 1(1-1)! = 1·0! = 1·1 = 1
  2! = 2(2-1)! = 2·1! = 2·1 = 2
  3! = 3(3-1)! = 3·2! = 3·2·1 = 6
  4! = 4(4-1)! = 4·3! = 4·3·2·1 = 24
  5! = 5(5-1)! = 5·4! = 5·4·3·2·1 = 120
- Quindi, per ogni **n** intero positivo, il fattoriale di **n** è il prodotto dei primi **n** numeri interi positivi

La ricorsione nella programmazione

- ***Si invoca un metodo*** mentre si esegue ***lo stesso metodo***
- è un paradigma di programmazione che si chiama
  ricorsione
  e un metodo che ne faccia uso si chiama
  metodo ricorsivo
- La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi, ma è anche fonte di molti errori di difficile diagnosi

In un programma…

- Vediamo la sequenza usata per calcolare 3!
  si invoca factorial(3)
  factorial(3) invoca factorial(2)
  factorial(2) invoca factorial(1)
  factorial(1) invoca factorial(0)
  factorial(0) restituisce 1
  factorial(1) restituisce 1
  factorial(2) restituisce 2
  factorial(3) restituisce 6
- Si crea quindi una lista LIFO (uno *stack*) di metodi "in attesa", che si allunga e che poi si accorcia fino ad estinguersi

La ricorsione

- Per capire come utilizzare correttamente ***la ricorsione***, vediamo innanzitutto ***come funziona***
- Quando un metodo ricorsivo invoca se stesso, ***la macchina virtuale esegue le stesse azioni che vengono eseguite quando viene invocato un metodo qualsiasi***
  - sospende l'esecuzione del metodo invocante
  - esegue il metodo invocato fino alla sua terminazione
  - riprende l'esecuzione del metodo invocante dal punto in cui era stata sospesa

La ricorsione

- In ogni caso, anche se il meccanismo di funzionamento della ricorsione può sembrare complesso, la chiave per un suo utilizzo proficuo è
  - dimenticarsi come funziona la ricorsione, ma sapere come vanno scritti i metodi ricorsivi perché il tutto funzioni!
- Esistono infatti due regole ben definite che vanno utilizzate per scrivere metodi ricorsivi che funzionino

La ricorsione: caso base

- **Prima regola**
  - il metodo ricorsivo ***deve fornire la soluzione del problema in almeno un caso particolare***, ***senza ricorrere ad una chiamata ricorsiva***
  - tale caso si chiama ***caso base*** della ricorsione
  - nel nostro esempio, il caso base era

```
if (n == 0)
   return 1;
```

  - a volte ci sono più casi base, non è necessario che sia unico

La ricorsione: passo ricorsivo

- **Seconda regola**
  - il metodo ricorsivo ***deve effettuare la chiamata ricorsiva dopo aver semplificato il problema***
  - nel nostro esempio, per il calcolo del fattoriale di **n** si invoca la funzione ricorsivamente per conoscere il fattoriale di **n-1**, cioè per risolvere un problema più semplice

```
if (n > 0)
   return n * factorial(n - 1);
```

  - il concetto di "problema più semplice" varia di volta in volta: in generale, ***bisogna avvicinarsi ad un caso base***

La ricorsione: un algoritmo?

- Le regole appena viste sono fondamentali per poter dimostrare che la soluzione ricorsiva di un problema sia un algoritmo
  - in particolare, che arrivi a conclusione in un numero **finito** di passi
- Si potrebbe pensare che le chiamate ricorsive si possano succedere una dopo l'altra, all'infinito

La ricorsione: un algoritmo?

- Invece, se
  - ad ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base
  - la soluzione del caso base non richiede ricorsione

allora certamente la soluzione viene calcolata in un numero finito di passi, per quanto complesso possa essere il problema

Ricorsione infinita

- Non tutti i metodi ricorsivi realizzano algoritmi
  - ***se manca il caso base***, il metodo ricorsivo continua ad invocare se stesso all'infinito
  - ***se il problema non viene semplificato ad ogni invocazione ricorsiva***, il metodo ricorsivo continua ad invocare se stesso all'infinito
- Dato che la lista dei metodi "in attesa" si allunga indefinitamente, l'ambiente runtime esaurisce la memoria disponibile per tenere traccia di questa lista, ed il programma termina con un errore

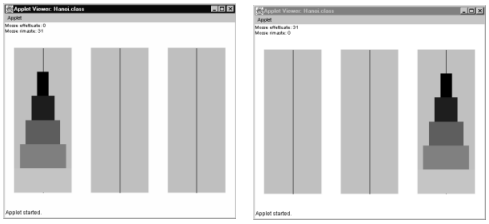
Esempio: Torri di Hanoi

Torri di Hanoi: regole

- Il rompicapo è costituito da tre pile di dischi ("torri") allineate
  - all'inizio tutti i dischi si trovano sulla pila di sinistra
  - alla fine tutti i dischi si devono trovare sulla pila di destra
- I dischi sono tutti di dimensioni diverse e quando si trovano su una pila devono rispettare la seguente regola
  - ***nessun disco può avere sopra di sé dischi più grandi***

Torri di Hanoi: regole

Situazione iniziale Situazione finale



Torri di Hanoi: regole

- Per risolvere il rompicapo bisogna ***spostare un disco alla volta***
  - un disco può essere rimosso dalla cima della torre ed inserito in cima ad un'altra torre
    - non può essere "parcheggiato" all'esterno…
  - in ogni momento deve essere rispettata la regola vista in precedenza
    - ***nessun disco può avere sopra di sé dischi più grandi***

Algoritmo di soluzione

- Per il rompicapo delle Torri di Hanoi è noto un algoritmo di soluzione ricorsivo
- Il problema generale consiste nello spostare **n** dischi da una torre ad un'altra, usando la terza torre come deposito temporaneo
- Per spostare **n** dischi da una torre all'altra si suppone di saper spostare **n-1** dischi da una torre all'altra, come sempre si fa nella ricorsione
- Per descrivere l'algoritmo identifichiamo le torri con i numeri interi 1, 2 e 3

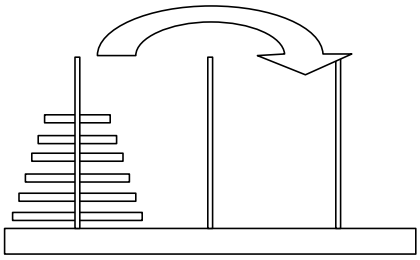
Algoritmo ricorsivo

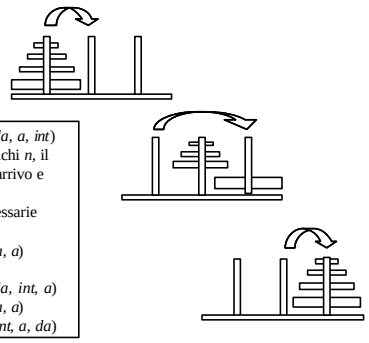
- Il caso base si ha quando **n** vale 1, in questo caso possiamo spostare liberamente il disco da una torre ad un'altra
- Per spostare **n** dischi dalla torre 1 alla torre 3
  1. gli **n-1** dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
  2. il disco rimasto nella torre 1 viene portato nella torre 3
  3. gli **n-1** dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)

Algoritmo ricorsivo

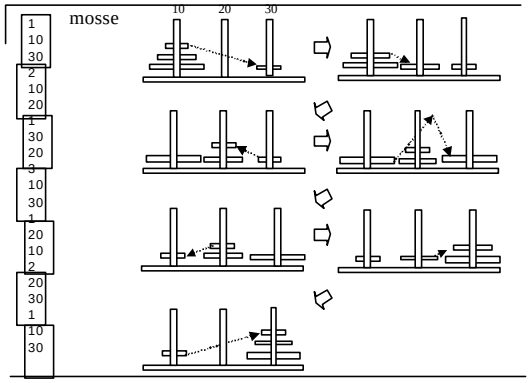
- Si basa sul ***Principio di induzione***
- Il principio di induzione dice che una proprietà è vera per qualsiasi valore di n se
  - è vera per $n = 1$, e
  - nell'ipotesi che sia vera per un dato valore di $n = n'$ siamo capaci di dimostrare che è vera per $n = n' + 1$

La "Torre di Hanoi"





Algoritmo Hanoi(*n, da, a, int*)
input: il numero di dichi *n*, il perno di partenza, di arrivo e intermedio
output: le mosse necessarie
if *n = 1* **then**
    muovi (*1, da, a*)
else
    hanoi (*n-1, da, int, a*)
    muovi (*n, da, a*)
    hanoi (*n-1, int, a, da*)



Algoritmo ricorsivo

- Si può dimostrare che il numero di mosse necessarie per risolvere il rompicapo con l'algoritmo proposto è pari a:

$$2^n - 1$$

- Il tempo necessario alla soluzione è proporzionale al numero di mosse (ogni mossa richiede un tempo costante) cioe':

$$T(n) = 2^n - 1$$

La torre di Brahama: la leggenda

- ***Torre di Hanoi con 64 dischi***
- Una leggenda narra che alcuni monaci buddisti in un tempio dell'Estremo Oriente siano da sempre impegnati nella soluzione del rompicapo, spostando fisicamente i loro **64** dischi da una torre all'altra, consapevoli che quando avranno terminato il mondo finirà

Quando finirà il mondo?

Dipende..

La torre di Brahama

- Sono necessarie $2^{64}-1$ mosse, che sono circa 16 miliardi di miliardi di mosse … cioè circa 1.6×10^{18}
- Supponendo che i monaci facciamo una mossa ogni minuto, essi fanno circa 500000 mosse all'anno, quindi il mondo finirà tra circa 30 mila miliardi di anni…
- Un processore ad 1GHz che fa una mossa ad ogni intervallo di clock (un miliardo di mosse al secondo…) impiega 16 miliardi di secondi, che sono circa 500 anni...

Linguaggi di programmazione

Linguaggi di Programmazione

- Un linguaggio di programmazione è una notazione formale per descrivere algoritmi
- Un programma è la descrizione di un algoritmo in un particolare linguaggio di programmazione
  - Quali "parole chiave"
  - Quali meccanismi di combinazione?

Linguaggi di Programmazione: Sintassi & Semantica

- Sintassi: l'insieme di regole formali per la scrittura di programmi in un linguaggio, che dettano le modalità per costruire frasi corrette nel linguaggio stesso
- Semantica: l'insieme dei significati da attribuire alle frasi (sintatticamente corrette) costruite nel linguaggio.
- **n.b.** Una frase può essere sintatticamente corretta e tuttavia non avere significato

Sintassi e Semantica

- Dopo la definizione di un linguaggio (con sintassi e semantica) serve un sistema che possa **eseguire** i programmi scritti in tale linguaggio
- Si parla di **implementazione** di un linguaggio
  - generalmente è un traduttore verso il linguaggio dell'elaboratore

Sintassi

- Per definire la sintassi di un linguaggio, è necessario definire le **grammatiche**
- La grammatica è un insieme di regole che servono per costruire una frase

Grammatica

- La grammatica è definita da:
  - un alfabeto di simboli *terminal* (l'alfabeto del linguaggio)
  - un alfabeto di simboli *non terminali*
  - un simbolo *iniziale S* (o *assioma*)
  - un insieme finito di regole sintattiche
- Data una grammatica, il linguaggio generato è definito come l'insieme delle frasi derivabili a partire dall'assioma S
  - Le frasi di un linguaggio di programma-zione sono dette **programmi**

Linguaggi di Programmazione

- Diversi tipi di linguaggi:
  - Imperativi
  - Funzionali
  - Dichiarativi
- Tutti basati sulla traduzione nell'unico linguaggio eseguibile dal calcolatore: il **Linguaggio Macchina**

Linguaggi di Programmazione

- I **linguaggi macchina** forniscono solo le operazioni che l'elaboratore può eseguire:
  - Operazioni molto elementari.
  - Diverse per ogni processore.
  - Scritte in linguaggio binario
- Sono più orientati alla macchina che ai problemi da trattare.

Linguaggi di Programmazione

- Una prima evoluzione sono i linguaggi assemblativi che sostituiscono i codici binari con codici simbolici.
  - ancora orientati alla macchina e non ai problemi
  - più immediati da utilizzare
  - definiscono variabili, simboli

```
START:  MOV   AX, BX
        CMP   AX, 12h
        JZ    EQUAL
        INT   21h
        RET
EQUAL:  MOV   BL, 82h
```

Linguaggi di Programmazione

- I linguaggi macchina e i linguaggi assemblativi sono detti linguaggi a basso livello.

Linguaggi di Programmazione

- I cosiddetti linguaggi ad alto livello permettono:
  - La descrizione del problema in modo intuitivo, dimenticandosi che verranno eseguiti da un calcolatore.
  - Una astrazione rispetto al calcolatore su cui verrà eseguito il programma.
- Ma devono essere tradotti in linguaggio macchina.

Linguaggi di Programmazione

- Esistono diversi tipi di linguaggi ad alto livello:
  - Linguaggi imperativi
  - Linguaggi logici
  - Linguaggi funzionali
  - Linguaggi orientati agli oggetti
- Ognuno provvede le forme espressive appropriate per alcuni problemi specifici.

Linguaggi Imperativi

- Un linguaggio imperativo è legato all'architettura della macchina di Von Neumann, ma ad un livello di astrazione vicino a quello dei diagrammi di flusso.
- Un programma è una sequenza di istruzioni in cui:
  - Le istruzioni vengono eseguite in sequenza.
  - La sequenza può essere alterata con istruzioni di salto.
- Eseguono 3 tipi di operazioni:
  - trasferimento dati
  - operazioni aritmetiche
  - alterazione del flusso del programma

Linguaggi Imperativi

- Le operazioni che effettuano le istruzioni si basano sul concetto di variabile.
- Una variabile è una astrazione del concetto di locazione di memoria in cui può essere assegnato, letto e modificato un valore.
- Il C, Fortran, Pascal, Cobol e Basic sono i linguaggi imperativi più diffusi.

Linguaggi Logici

- Basati sulla logica
  - obiettivo: formalizzare il ragionamento
  - caratterizzati da meccanismi deduttivi
- Programmare significa:
  - descrivere il problema con formule del linguaggio
  - interrogare il sistema, che effettua deduzioni sulla base delle definizioni

Linguaggi Logici

- In un linguaggio logico un programma è composto da una base di conoscenza contenente:
  - Una descrizione del dominio del problema composta da un insieme di fatti.
  - Un insieme di regole per manipolare l'insieme dei fatti.

Linguaggi Logici

- L'esecuzione di un programma corrisponde all'applicazione delle regole sui fatti della base di conoscenza:
  - L'esecuzione è guidata da un motore inferenziale.
  - L'esecuzione è attivata da una interrogazione dell'utente.
  - Lo scopo dell'esecuzione è verificare che l'interrogazione dell'utente è vera o falsa.
- Il linguaggio più conosciuto è il Prolog.

Linguaggi Funzionali

- Linguaggi basati sul concetto di funzione e di applicazione di una funzione ad argomenti; per questa ragione essi sono anche detti *linguaggi applicativi*
- In un linguaggio funzionale un programma è visto come una funzione matematica.
- L'esecuzione di un programma corrisponde alla valutazione della funzione rispetto ai suoi argomenti.

Linguaggi Funzionali

- La differenze maggiori con un linguaggio imperativo sono:
  - Non esiste il concetto di variabile: il calcolo è basato sul calcolo di valori e non sull'assegnamento di valori a variabili
  - il risultato è il risultato di una funzione, non l'effetto causato dalla esecuzione di una sequenza di operazioni
  - Sono adatti a elaborazioni simboliche non numeriche.
- Il Lisp (acronimo per LISt Processing) è il linguaggio più conosciuto, ma non è più molto usato.

Linguaggi Orientati agli Oggetti

- In un linguaggio di programmazione orientato agli oggetti (OOP) un programma consiste:
  - Definizione di un insieme di classi.
  - Creazione di un insieme di oggetti ed esecuzione dei suo metodi.
  - Il C++ e Java sono i due linguaggi OOP più diffusi

Cos'è un *oggetto*?

- Interagiamo con oggetti di uso quotidiano, conoscendone le *funzioni*, ma non il *funzionamento interno*
  - Gli oggetti sono ***scatole nere*** dotate di interfaccia che limita l'accesso ai meccanismi interni
  - Gli oggetti hanno uno ***stato***
    - L'insieme delle proprietà che lo caratterizzano in un dato istante
  - e un ***comportamento***
    - L'insieme delle azioni che un oggetto può compiere
- Un oggetto sw è un'***astrazione*** o un ***modello*** della realtà che limita il numero dei *dettagli* rappresentati *all'essenziale* per il contesto considerato

Astrazione

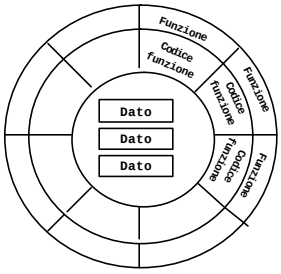
- L'astrazione nasconde o ignora dettagli inessenziali
- Un oggetto è un'astrazione
  - Non ci preoccupiamo dei suoi dettagli interni per usarlo
  - Non conosciamo come funziona il metodo "scrivi a monitor" quando l'invochiamo
- Effettuiamo astrazioni continuamente
  - Possiamo trattare solo poche informazioni contemporaneamente
  - Ma se raggruppiamo le informazioni (come gli oggetti) allora possiamo trattare informazioni più complicate
- Quindi, possiamo anche scrivere software complesso organizzandolo attentamente in classi e oggetti

Gli oggetti software

- Lo ***stato*** di un oggetto sw è descritto e rappresentato da una o più ***variabili***
  - Una variabile è un ***dato*** individuato da un ***identificatore***
- il ***comportamento*** è definito dai ***metodi***
- Un oggetto sw è costituito dall'insieme delle variabili e dei metodi

Gli oggetti software

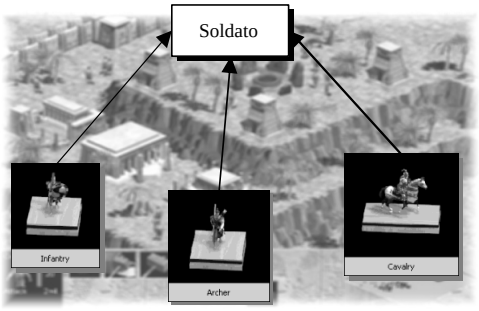
Un insieme di dati e funzioni:



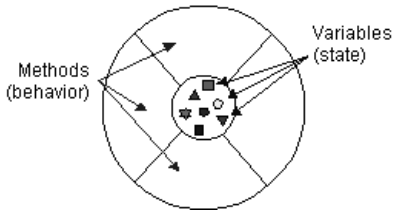
Oggetti e classi

- Gli ***oggetti*** sono generati da una ***classe***
  - Si dicono anche ***istanze*** della classe
- La ***classe*** è uno schema per produrre una categoria di oggetti strutturalmente identici
  - La classe costituisce il ***prototipo***
  - Una classe è una fabbrica di istanze: possiede lo schema e la tecnica di produzione

Classi ed ereditarieta'

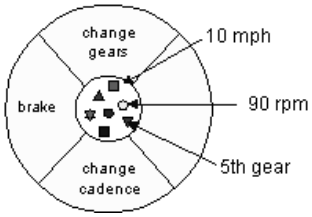


Gli oggetti come astrazione



- Un oggetto che modella una bicicletta
  - Una velocità (20 Km/h), il giro dei pedali (15 g/m) e la marcia (5°) sono ***variabili di istanza***
  - proprietà rappresentate in ciascuna bicicletta modellata

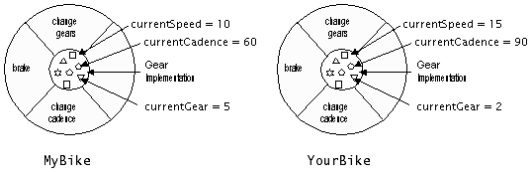
Gli oggetti come astrazione – 2



- Inoltre nel modello rappresentiamo funzioni come ***frenare*** o ***cambiare marcia***, che modificano le variabili d'istanza
- Si chiamano ***metodi d'istanza*** perché hanno accesso alle variabili d'istanza e le modificano

Le istanze

- Definita una classe, si possono creare un numero arbitrario di oggetti appartenenti alla classe

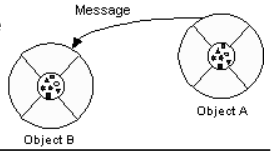


Incapsulamento dei dati

- Nascondere le informazioni fornendo *un'interfaccia*
- Netta divisione fra interfaccia e implementazione
- Le *variabili* di un oggetto, che ne rappresentano lo stato, sono *nascoste* all'interno dell'oggetto, *accessibili* solo ai metodi
- Idealmente i metodi proteggono le variabili
- Diversi livelli di accesso a metodi e variabili
  - **Private**: accessibili solo alla classe
  - **Public**: accessibili a chiunque e quindi anche alle sottoclassi
- Consente modularità e flessibilità nel codice, impedendo errori nel manipolare gli oggetti e semplifica la gestione di sistemi complessi

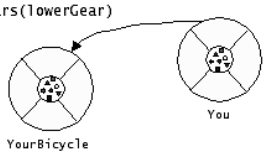
I messaggi

- Gli oggetti interagiscono tra loro per ottenere funzioni più complesse
  - La bicicletta appesa in garage è un oggetto e basta, ci vuole un ciclista che interagisca con lei perché sia interessante
- Gli oggetti sw per interagire si mandano messaggi
  - Chiedendo di eseguire un certo metodo (procedura)



I messaggi – 2

- Spesso i metodi necessitano di informazioni per poter essere eseguiti: ***i parametri***
- Tre componenti:
  - L'oggetto a cui il messaggio è rivolto
  - Il metodo da eseguire per ottenere un certo effetto
  - I parametri, se necessari al metodo

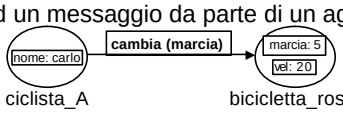


I messaggi – 3

- Un ***oggetto*** può essere visto come un insieme di servizi che possiamo chiedere di eseguire
- I servizi sono definiti dai ***metodi***
- Il comportamento degli oggetti è definito dai suoi metodi e il meccanismo di invio dei messaggi consente l'interazione tra gli oggetti
- Ogni oggetto ha in se tutto ciò che gli serve per rispondere alle chiamate
- Gli oggetti che si scambiano i ***messaggi*** possono anche essere '*distanti*' tra loro
  - Su macchine diverse
  - Non appartenenti allo stesso modello

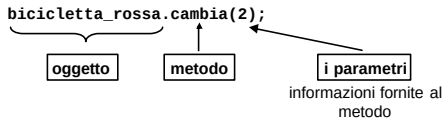
Interfaccia

- L'***interfaccia*** è l'insieme dei messaggi che un oggetto è in grado di interpretare
- Un oggetto deve soddisfare la richiesta di un messaggio
- Eseguendo il ***metodo*** si soddisfa la risposta ad un messaggio da parte di un agente



Inviare messaggi

- Il ciclista **`ciclista_A`** che vuole cambiare marcia invia il messaggio all'oggetto **`bicicletta_rossa`**



Approccio OO

- Sono le strutture di dati che svolgono le azioni, non le *subroutines*
- Il lavoro è svolto dal *server*, non dal *client*
- "Cos' è?" "Com' è fatto?"

  → Data Oriented
- "Cosa può fare per me?"

  → Object Oriented

Perché programmare *per oggetti*?

- Programmare *per oggetti* non velocizza l'esecuzione dei programmi...
- Programmare *per oggetti* non ottimizza l'uso della memoria…

  E allora perchè programmare *per oggetti*?
- Programmare *per oggetti* facilita la progettazione e il mantenimento di sistemi software molto complessi!

Caratteristiche del software *non mantenibile*

- Rigidità
  - non può essere cambiato con faciltà
  - non può essere stimato l'impatto di una modifica
- Fragilità
  - una modifica singola causa una cascata di modifiche successive
  - i bachi sorgono in aree concettualmente separate dalle aree dove sono avvenute le modifiche
- Non *riusabilità*
  - esistono molte interdipendenze, quindi non è possibile estrarre parti che potrebbero essere comuni

Programmazione ad oggetti

- La programmazione ad oggetti, attraverso l'incapsulazione, consente di:
  - ridurre la dipendenza del codice di alto livello dalla rappresentazione dei dati
  - riutilizzare del codice di alto livello
  - sviluppare moduli indipendenti l'uno dall'altro
  - avere codice utente che dipende dalle interfacce ma non dall'implementazione

Metodi di sviluppo del software

Metodi di sviluppo del software

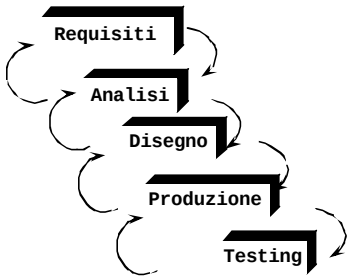
Un metodo comprende:

- Una notazione
  mezzo comune per esprimere strategie e decisioni
- Un processo
  specifica come deve avvenire lo sviluppo

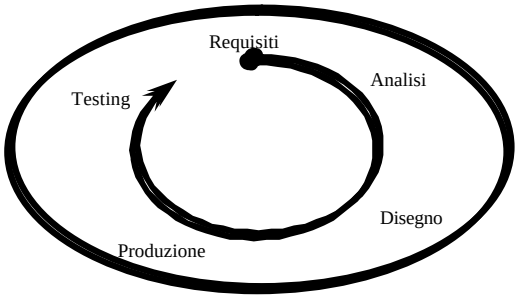
Strategie di sviluppo di un progetto software

- Requisiti: cosa l'utente vuole
- Analisi: la visione dell'informatico dei requisiti
- Disegno: l'aspetto del sistema software
- Produzione: codifica
- Testing: debugging e verifica dei requisiti
- Mantenimento: installazione del prodotto e controllo del funzionamento per il resto della sua vita

Modello *a cascata*



Modello *evoluzionario*



Confronto fra i modelli di sviluppo

A cascata

- Processo lineare (si torna al passo precedente solo in caso di problemi)
- Confinamento delle attività in ogni fase
- Facile da gestire (gestione delle scadenze)
- Difficile da modificare
- Prodotto utilizzabile solo alla fine del processo

Evoluzionario

- Processo ciclico (brevi processi completi)
- Attività distribuite su più fasi
- Difficile da gestire
- Facile da modificare e integrare
- Prototipo utilizzabile fin dal primo ciclo

Requisiti

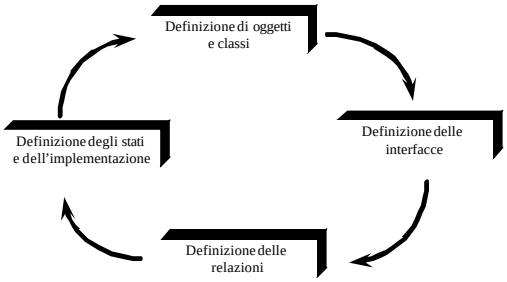
- Definizione delle richieste da parte dell'utente del programma (o di una sua parte) sul sistema
- Si parla di *programmazione per contratto* perchè l'utente richiede solamente la definizione del servizio richiesto NON la metodologia seguita per fornirglielo
  - è possibile delegare parte del lavoro richiesto ad altri
  - il sistema è indipendente da chi è il suo utente

INCAPSULAMENTO!

Analisi

- Comprensione e *razionalizzazione* delle richieste dell'utente
- Costruzione di un modello
  - astrazione (semplificazione delle relazioni)
  - rilevanza (identificazione degli oggetti chiave)
- Da non trascurare: analisi delle soluzioni esistenti. Può far risparmiare molto tempo!!!

Disegno

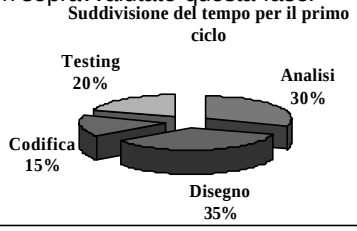


Disegno (2)

- Dopo ogni ciclo bisogna analizzare i rischi, la stabilità del disegno e la complessità delle classi
- Se una classe è troppo complessa conviene dividerla
- Ad ogni ciclo il numero di modifiche deve diminuire
- Architetture troppo complesse devono essere modularizzate

Codifica

- C'è poco da dire…
- Non sopravvalutate questa fase:



Testing

- Debugging: è ovvio… il codice non deve dare errori.
- Use cases: specificano il comportamento del sistema in una *regione*.
- Scenarios: sono esempi concreti di use cases. Per definizione se tutti gli scenari sono soddisfatti correttamente il test è positivo.