

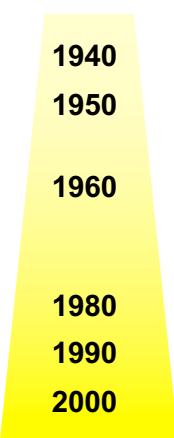
# Ingegneria del Software

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

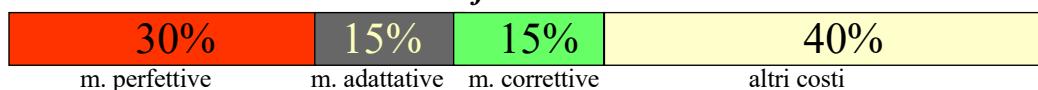
**E P Portabilità:** un sw è portatile se può funzionare su più piattaforme (es. Java)

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)

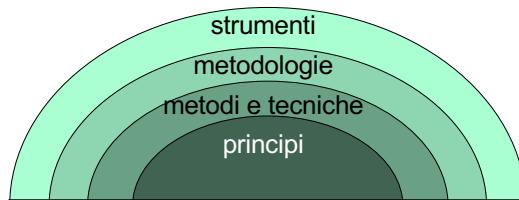
**PC Produttività:** misura l'efficienza del processo di produzione del software in termini di velocità di consegna del sw.

**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

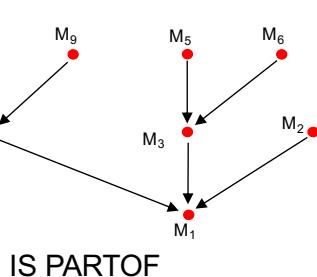
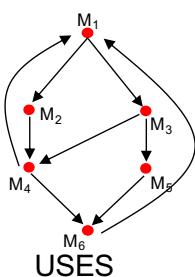
# Principi di progettazione: modularità

- 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

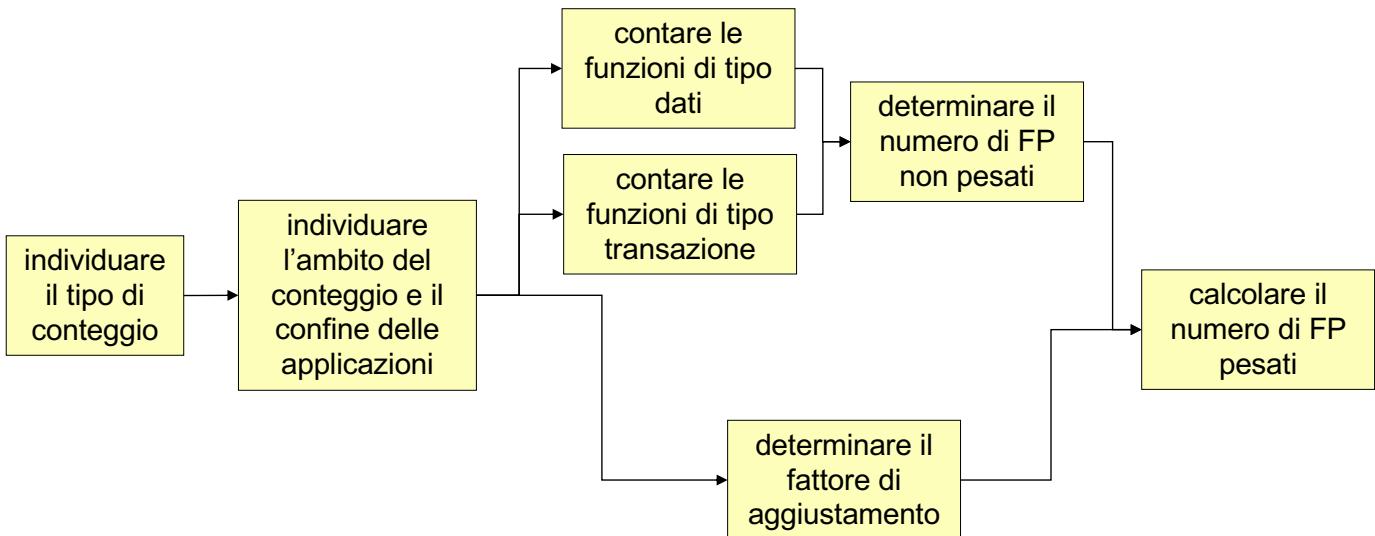
## Conteggio dei function point

- 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

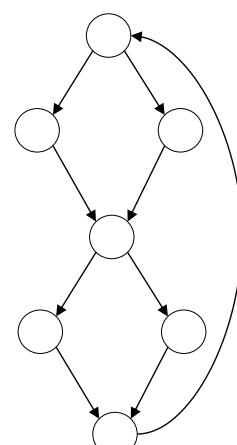
- Il numero ciclomatico di un grafo fortemente connesso è il numero minimo di archi che occorre eliminare per trasformarlo in un albero

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

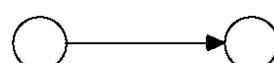
## Il numero ciclomatico

- 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*:

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

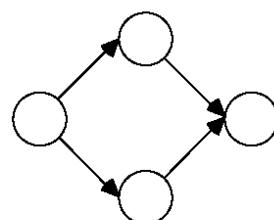
33

## Il numero ciclomatico

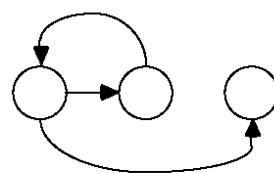


$$v(G)=1$$

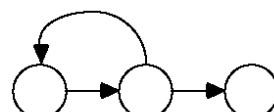
ESEMPI:



$$v(G)=2$$



$$v(G)=2$$



$$v(G)=2$$

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 <sub>Lang</sub>	#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
<b>Proprietà del prodotto</b>						
- 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
<b>Caratteristiche dell'hardware</b>						
- 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		
<b>Caratteristiche del team</b>						
- 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		
<b>Caratteristiche del progetto</b>						
- 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

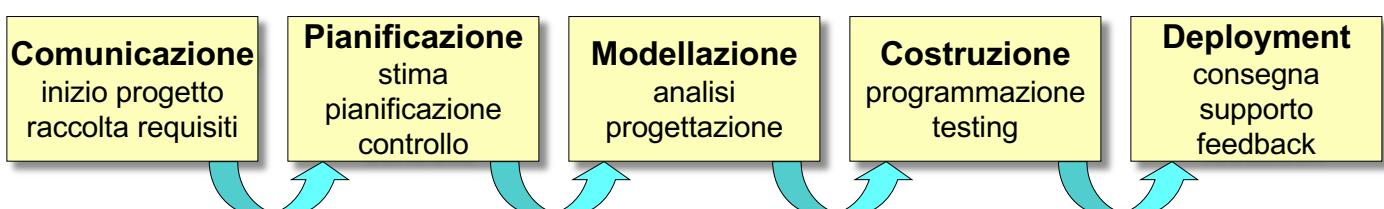
# I modelli prescrittivi

- 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)

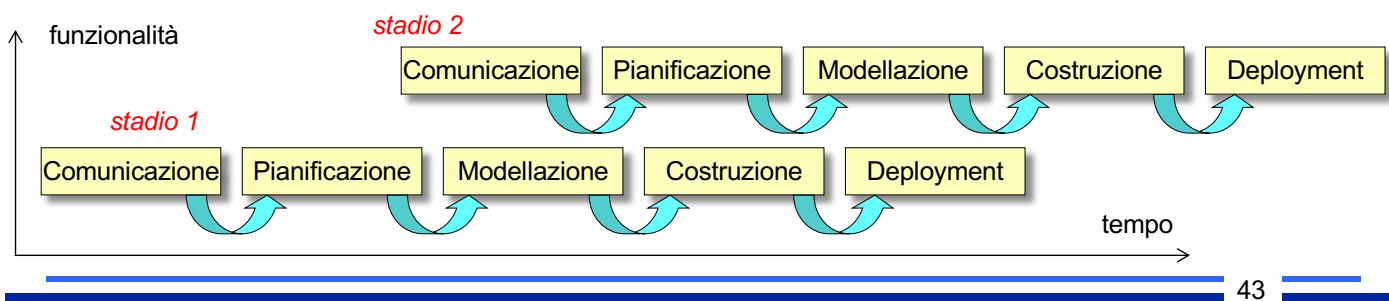
- Introdotto nel 1970, suggerisce un **approccio sistematico e sequenziale lineare**, in cui l'**output di ogni fase rappresenta l'input della successiva**
  - ⇒ E' inadeguato quando (come spesso accade) i requisiti sono incerti o non noti durante le fasi iniziali del progetto
  - ⇒ Non permette di modificare 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

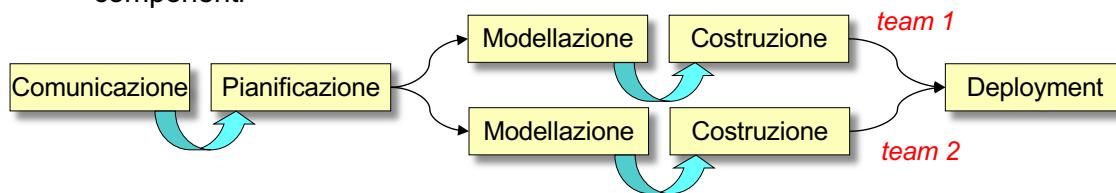
- E' un modello *iterativo* che combina aspetti del modello a cascata applicati a sottosistemi del prodotto finale, producendo il software a *incrementi*
- Consiste nell'applicare più sequenze lineari, scalate nel tempo, ognuna delle quali produce uno **stadio** operativo del software
  - ⇒ Il primo 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**

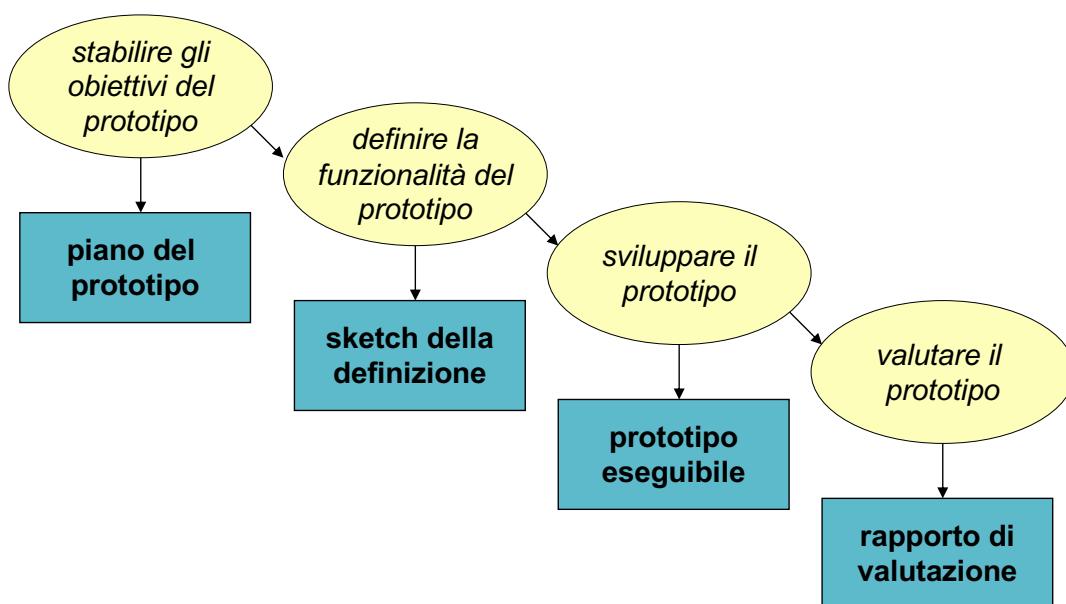
46

# 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

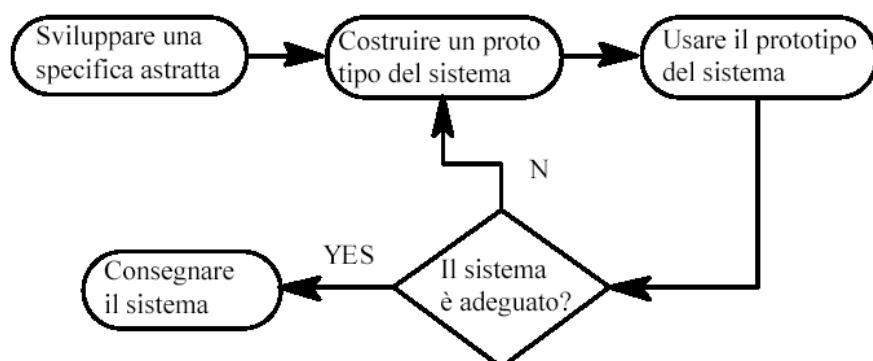
## □ Due tecniche:

- ⇒ 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



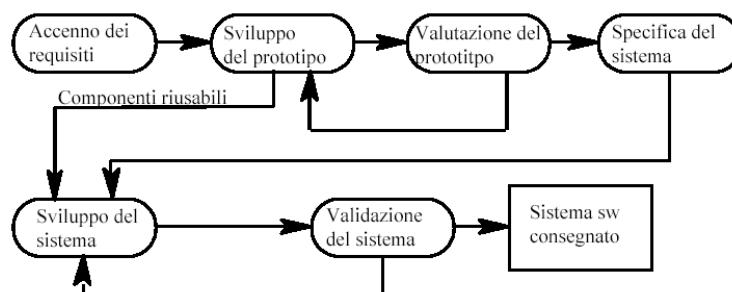
## □ Ma...

- ⇒ 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

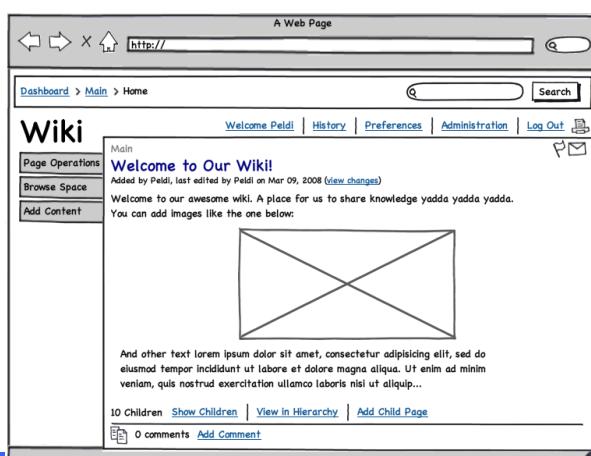
- Usata per ridurre il rischio dei requisiti incerti
- Il prototipo è sviluppato da una specifica iniziale, consegnato per sperimentazione e quindi gettato
- Il prototipo NON deve essere considerato un sistema finale perché
  - ⇒ alcune caratteristiche del sistema possono non essere state considerate
  - ⇒ 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

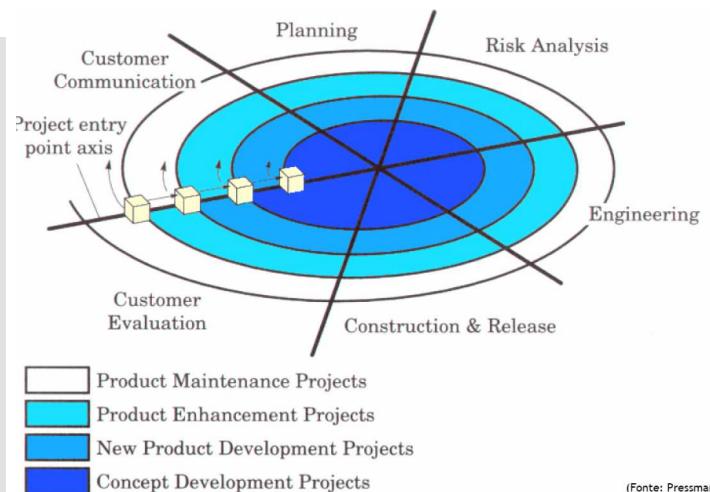


52

# 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

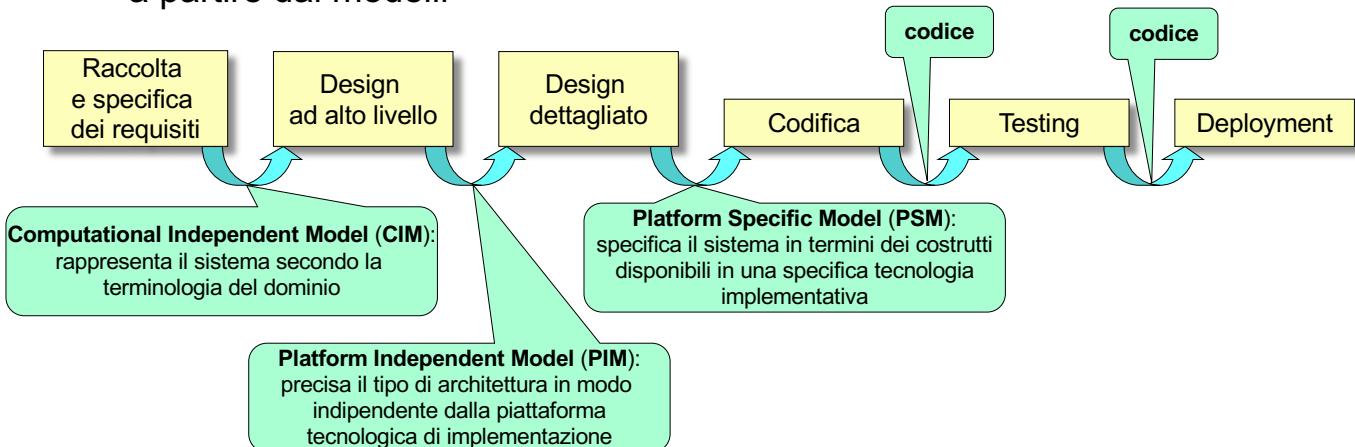


(Fonte: Pressman)

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

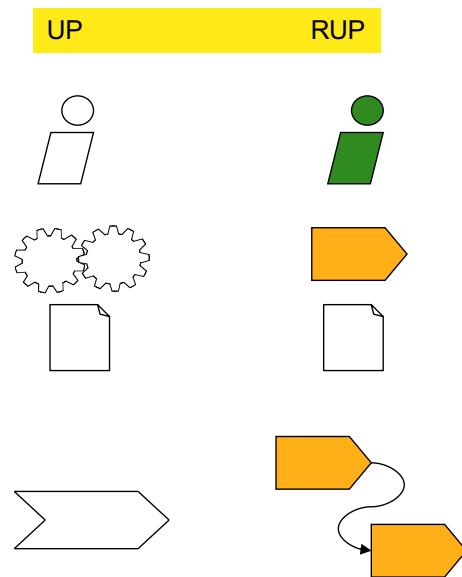
## Unified Process

- 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

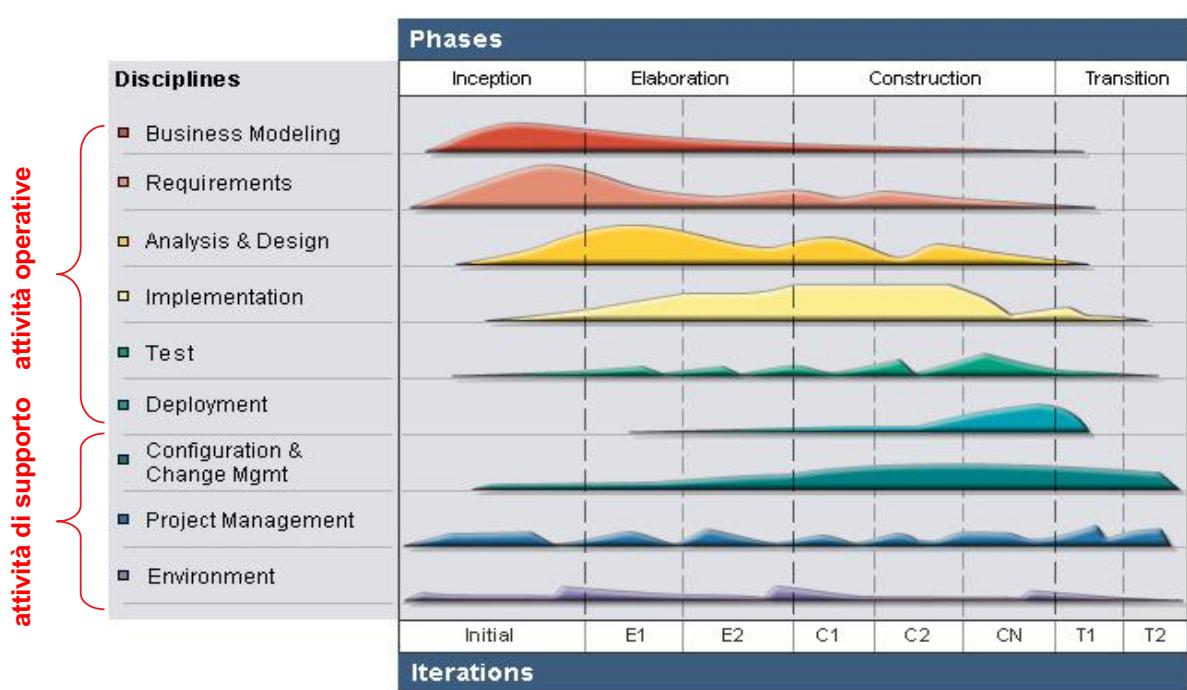
64

# 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:

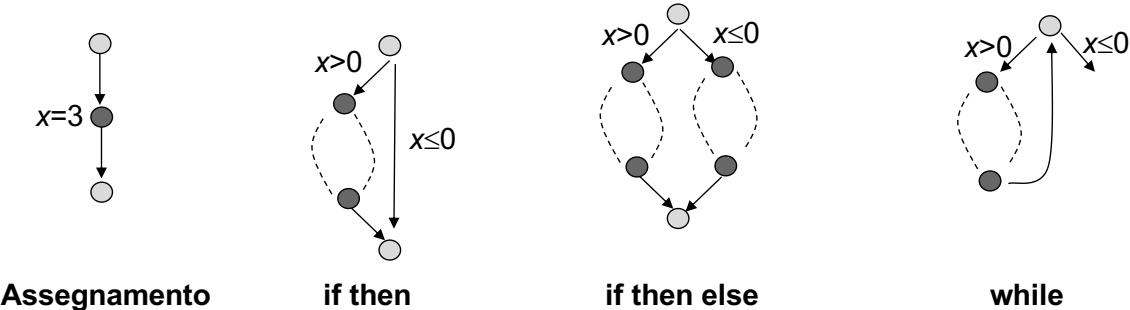
- **Testing in the small**: riguardano moduli singoli e porzioni specifiche del codice che rivestono una particolare importanza o che hanno una particolare complessità
- **Testing in the large**: riguardano il sistema nella sua globalità

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*)

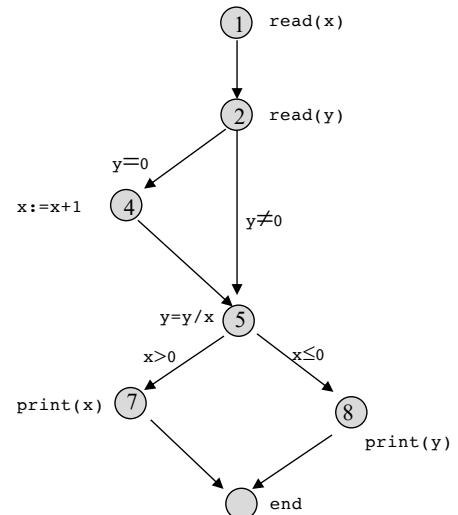
72

## 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

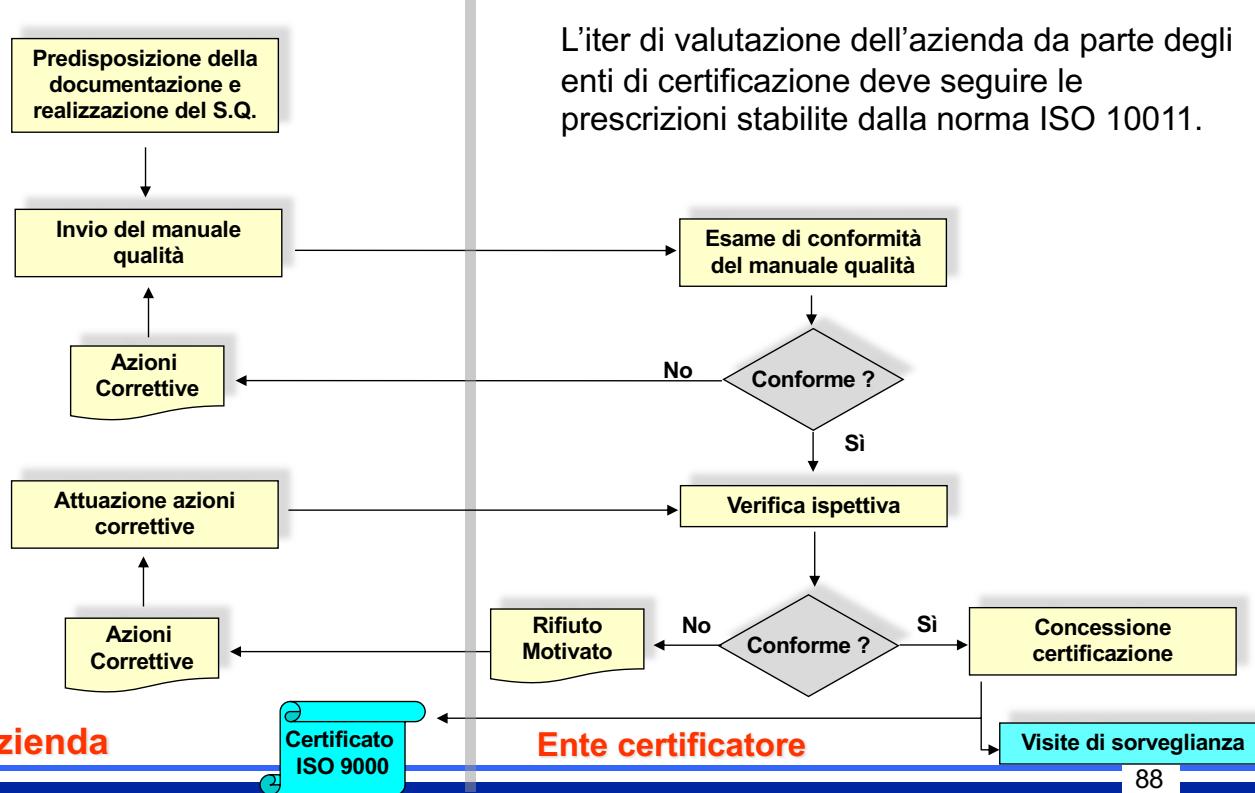
# La certificazione ISO 9000

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.

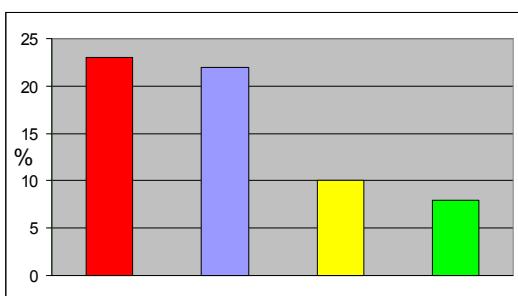
89

# 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

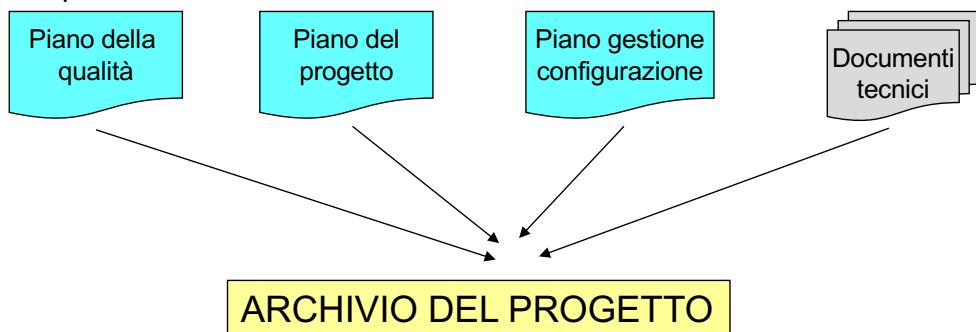
90

# 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*