

Produzione del software

Evoluzione del processo:

- da arte (prevale il lavoro individuale e creativo)
- ad artigianato (piccoli gruppi di lavoro di esperti)
- a industria:
 - l'attività di sviluppo coinvolge gruppi anche grandi
 - il lavoro deve essere coordinato, pianificato, standardizzato e documentato (per permettere il ricambio di personale)
 - lo sviluppo dei sistemi deve permettere il riuso di componenti
 - la qualità deve essere certificata in modo convincente

1

Ciclo di vita dei sistemi informatici

- Insieme e sequenzializzazione delle attività svolte da analisti, progettisti, utenti, nello sviluppo e nell'utilizzazione dei sistemi informatici.
- Si tratta di una attività iterativa (perciò "ciclo")

Il tutto con l'obiettivo di migliorare la qualità (riducendo i costi)

2

Attori

- committente
- progettista (analista o programmatore)
- utente finale
- manutentore

3

Metodologia

Disciplina delle attività da svolgere per conseguire un risultato. Caratteristiche:

- articolazione in fasi
- definizione degli input e dei prodotti di ciascuna fase
- standardizzazione (più o meno spinta) di ciascuna fase

4

Fasi del ciclo di vita
una possibile articolazione

- Attività preliminari
- Raccolta e analisi dei requisiti
- Progettazione
- Realizzazione
- Operazione

5

- nella storia (ciclo di vita) di un sistema le fasi si ripetono ciclicamente (per varie ragioni)
- sono frequenti (almeno al termine delle varie fasi) attività di verifica (validazione) della qualità dei prodotti
- il ciclo di vita coinvolge componenti:
 - organizzativa
 - applicativa
 - tecnologica

6

Attività preliminari
(Variano molto a seconda del contesto e dei progetti)

Possono comprendere

- Definizione del problema
- Studio di fattibilità e valutazione dell'impatto
- Decisione e/o definizione della porzione da realizzare
- Formulazione del piano delle attività successive

7

Una decisione fondamentale:
“Make or buy?”

- le attività preliminari debbono includere valutazioni sulle modalità organizzative di sviluppo
- è spesso indispensabile affidare all'esterno le attività (di progettazione e realizzazione, ma anche lo studio di fattibilità oppure la gestione vera e propria)
- il committente non deve però “perdere il controllo” dell'iniziativa
- le problematiche relative all'affidamento all'esterno sono molto articolate (in particolare per gli enti pubblici, ma non solo)

8

- anche in caso di affidamento all'esterno, le fasi del ciclo di vita rimangono di interesse
 - perché il processo va coordinato e controllato
 - perché l'affidamento riguarda spesso solo alcune fasi

9

Raccolta e analisi dei requisiti

Attività volta alla individuazione dei requisiti dell'applicazione, in tutti i dettagli significativi. Le informazioni vengono acquisite attraverso

- interazione con gli utenti (a diversi livelli)
- studio delle realizzazioni esistenti
- studio della normativa

10

Progettazione

Attività volta alla individuazione delle modalità secondo cui l'applicazione risponderà ai requisiti. In questa fase vengono definiti

- i dati di ingresso e uscita e la loro organizzazione
- l'architettura hardware e software
- la organizzazione dei moduli software

11

Realizzazione

Sviluppo dei programmi (codifica e test) con la relativa documentazione

12

Operazione

- installazione (messa in funzione del nuovo ambiente hardware e software)
- addestramento degli utenti
- conversione dei dati
- transizione
- controllo delle prestazioni (con riferimento al sistema, al suo impatto, all'attività di sviluppo)
- manutenzione (correttiva, adattativa, migliorativa)

13

Qualità del software

- qualità esterne: percepibili osservando il comportamento del prodotto (visto come “scatola nera”); sono l'obiettivo finale;
- qualità interne: richiedono un esame interno del prodotto o del processo che lo ha generato (“scatola trasparente”); permettono di raggiungere le qualità esterne

Molte delle qualità sono definibili solo in modo intuitivo e non misurabile.

14

Qualità esterne

- correttezza (rispetto alle specifiche; ma le specifiche sono corrette?)
- efficienza: capacità di sfruttare al meglio le risorse (spazio e tempo)

15

Qualità esterne

- robustezza: comportamento “accettabile” in situazioni non previste
- usabilità: l'utente è messo a proprio agio:
 - flessibilità;
 - interfaccia amichevole (evoluzione dei dispositivi);
 - capacità di aiuto e spiegazione;
 - verifiche e richieste di conferma

16

Qualità interne

- riusabilità (dei componenti)
- manutenibilità; rispetto a tre tipi di manutenzione:
 - correttiva
 - adattativa (rispetto a modifiche del contesto o delle specifiche — portabilità)
 - perfettiva (aggiunta di funzionalità — estendibilità)

17

Qualità interne

- portabilità
- interoperabilità
- modularità
- leggibilità
- completezza ed efficacia della documentazione

18

Produzione del software

principi generali

tecniche: metodi

strumenti di supporto

19

Produzione del software: principi

- astrazione
- decomposizione
- distinzione fra progettazione e realizzazione

20

Astrazione

un principio fondamentale dell'ingegneria

- procedimento mentale che porta a considerare solo gli aspetti essenziali (di interesse dal punto di vista del contesto in cui si opera) di una realtà, trascurando i dettagli secondari

L'astrazione porta a costruire un modello (più o meno semplificato) della realtà che si analizza o dell'oggetto da costruire

21

- in fase di progettazione si opera ad un livello più astratto rispetto alla realizzazione finale
- possiamo supporre che non vi sia una netta distinzione, ma diversi livelli, con dettaglio sempre maggiore (e quindi astrazione minore), fino ad arrivare al programma concreto

22

due tipi importanti di astrazione nella progettazione del software:

- astrazione sulle funzioni: porzioni di una funzione complessa vengono solo parzialmente specificate (“che cosa” e non “come”)
- astrazione sui dati: il comportamento esterno dei dati (le operazioni su di essi) viene descritto indipendentemente dalla effettiva realizzazione

23

Altri processi di astrazione

- classificazione
- aggregazione
- generalizzazione
- decomposizione

24

Decomposizione di un problema e della sua soluzione

È spesso conseguenza di precedenti astrazioni

Criteri per la decomposizione:

- livello di dettaglio e di complessità dei sottoproblemi
- caratterizzabilità e indipendenza dei sottoproblemi
- combinabilità delle soluzioni dei sottoproblemi

25

Top-Down o Bottom-Up

- il concetto di decomposizione fa pensare ad un procedimento top-down, attraverso raffinamenti basati su:
 - sequenza
 - iterazione
 - alternativa
 - ricorsione
- in effetti, si procede spesso in modo bottom-up, riutilizzando porzioni di progetti e programmi sviluppati in precedenza

26

Applicando i principi di astrazione, nel processo di sviluppo del software si distinguono due fasi:

progettazione (o concettualizzazione): produce una descrizione **astratta** della soluzione del problema, in linea di principio indipendente dal linguaggio poi utilizzato

realizzazione: produce il programma di interesse

Nello sviluppo del software non sempre è facile trovare il confine tra progettazione e realizzazione (e tra analisi e progettazione)

27

Progettazione

- concepimento della soluzione informatica del problema;
- specifica (astratta) dei dati da manipolare;
- specifica (astratta) delle operazioni sui dati (software).

28

Avvertenza

- il prodotto della progettazione di software (detta anche concettualizzazione) è diverso dal prodotto della progettazione di basi di dati
 - nella progettazione dei dati interessa la modellazione della realtà di interesse;
 - nella progettazione dei processi (software) interessano aspetti più concreti orientati alla realizzazione del software.

29

Progettazione di software: come si procede

- non ci sono regole precise: sono necessari esperienza, buon senso, creatività, conoscenza delle tecniche

30

Realizzazione

- definizione dell'architettura del programma, con individuazione dei **moduli** (applicazione del principio di decomposizione)
- realizzazione dei singoli moduli in specifici linguaggi di programmazione

la programmazione orientata agli oggetti permette di seguire in modo naturale una attività articolata in progettazione e realizzazione

31

Tecniche adottate nella progettazione e realizzazione

- modularizzazione, ovvero articolazione in componenti con interazione ben definita
- tecniche di specifica, basate su formalismi di vario tipo, per la descrizione dei risultati della progettazione (UML)
- definizione di algoritmi e strutture di dati
- tecniche di programmazione
 - tecniche generali (programmazione strutturata, ricorsione, etc.)
 - paradigmi di programmazione

32

Strumenti di progettazione e realizzazione

Strumenti software

- strumenti CASE (Computer Aided Software Engineering) per la progettazione
- per la realizzazione:
 - editor
 - compilatori, linker, interpreti
 - esecutori/debugger

33

Paradigmi di programmazione

imperativo: un programma è composto di istruzioni che specificano operazioni (comandi) da eseguire;
esistono variabili cui si assegnano valori

funzionale: un programma è la specifica di una funzione (che a sua volta può contenere la specifica di altre funzioni)

logico: un programma è la specifica di una relazione (anche complessa) fra insiemi di dati, basata sulla logica matematica;

orientato agli oggetti: un programma è la specifica di un insieme di classi di oggetti, ognuna definita per mezzo di struttura e operazioni

Nota: non si tratta di una classificazione netta;
ad esempio C++ è un linguaggio imperativo orientato agli oggetti.

34

Progettazione e programmazione orientata a ...

orientamento alle funzioni: il centro dell'attenzione sono le funzioni da realizzare, il procedimento risolutivo; i dati sono rilevanti solo in quanto utilizzati dalle funzioni

orientamento agli oggetti: il centro dell'attenzione sono le entità (oggetti, dati, ...) della realtà di interesse, ciascuno con le operazioni associate

35

Programmazione orientata agli oggetti aspetti fondamentali

Astrazione sui dati

incapsulamento : i dati sono descritti e organizzati con le relative operazioni;
non esiste altro modo di accedere ai dati

oggetto = dato + operazioni;

classificazione : dati (oggetti) omogenei vengono organizzati in classi con proprietà comuni

“information hiding” : per ogni oggetto (classe) esistono un'interfaccia e una implementazione; per utilizzare l'oggetto è sufficiente far riferimento all'interfaccia; gli oggetti hanno una parte “privata” e una “pubblica”

ereditarietà : è possibile definire nuove classi sulla base di classi definite in precedenza, aggiungendo o modificando caratteristiche

N.B.: i termini vengono talvolta usati con accezioni diverse, ma i concetti sono condivisi.

36

una semplice classe

```
class NumeroComplesso {
public
    float Re; // parte reale
    float Im; // coeff dell'immaginario
    float Modulo() { return sqrt(Re * Re + Im * Im); }
};
```

le operazioni che possiamo fare su oggetti della classe NumeroComplesso sono relative alle due variabili e alla funzione

37

utilizziamo la classe NumeroComplesso

```
NumeroComplesso c;
c.Re = 4.;
c.Im = 3.;

cout << c.Modulo(); // stampa il valore 5

cout << Modulo(); // errore (rilevato dal compilatore):
// Modulo() e' incapsulata
// puo' essere richiamata solo con riferimento
// ad un oggetto della classe
```

38

Information hiding

```
class NumeroComplesso {
public
    float Modulo() { return sqrt(Re * Re + Im * Im); }
    float Fase() {
        if (Im >= 0) return acos(Re/Modulo());
        else return -acos(Re/Modulo());
    }
    float Reale() { return Re; }
    float Immaginaria() { return Im; }
private
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

```
NumeroComplesso c;
cout << c.Reale(); // corretto
cout << c.Modulo(); // corretto
cout << c.Re; // scorretto
```

39

Un'altra implementazione della stessa classe

```
class NumeroComplesso {
public
    float Modulo() { return m; }
    float Fase() { return phi; }
    float Reale() { return m * cos(phi); }
    float Immaginaria() { return m * sin(phi); }
private
    float m; // modulo
    float phi; // fase
};
```

```
NumeroComplesso c;
cout << c.Reale(); // corretto
cout << c.Modulo(); // corretto
cout << c.Re; // scorretto
cout << c.m; // scorretto
```

40

utilizziamo la classe NumeroComplesso

```
int Quadrante(NumeroComplesso c) {
    if (c.Reale()==0 || c.Immaginaria()==0)
        return 0; // nell'origine o su uno degli assi
    if (c.Reale() > 0)
        if (c.Immaginaria() > 0)
            return 1;
        else
            return 4;
    else if (c.Immaginaria() > 0)
        return 2;
    else
        return 3;
}
```

41

Ereditarietà

```
class NumeroComplessoVisualizzabile : public NumeroComplesso {
public
    void visualizza() {
        ...
    }
};

void VisualizzaMassimo( NumeroComplessoVisualizzabile : c1 ,
                        NumeroComplessoVisualizzabile : c2 ) {
    if c1.Modulo() > c2.Modulo()
        c1.Visualizza();
    else
        c2.Visualizza();
}
```

Utilizziamo anche la funzione Modulo definita sulla classe NumeroComplesso ed ereditata dalla classe NumeroComplessoVisualizzabile

42

Oggetti e messaggi

- il paradigma orientato agli oggetti viene spesso definito attraverso il modello “oggetto-messaggio”;
- nel paradigma imperativo tradizionale, le funzioni si chiamano l'un l'altra con parametri di ingresso e uscita (ed ev. utilizzano variabili globali);
- nel paradigma O-O, ogni dato (oggetto appartenente ad una classe) sa eseguire “su se stesso” le operazioni ammissibili, secondo le richieste che gli provengono dall'esterno;
- ad esempio, nell'interfaccia del MacIntosh o di Windows, ogni documento ha una operazione “apri” che viene attivata dall'utente con il “clic” del mouse (che corrisponde all'invio di un messaggio: “apriti”)

43

Modularizzazione con procedure e con oggetti

- la decomposizione è una tecnica essenziale per affrontare problemi complessi
- la modularizzazione (“strutturazione dei programmi”) è il contraltare progettuale/realizzativo della decomposizione dei problemi
- nei linguaggi imperativi tradizionali, la modularizzazione è centrata sulle funzioni
- nel paradigma O-O, la modularizzazione è centrata sulle classi (o meglio, sui tipi astratti di dato)

Torneremo su questi aspetti fra qualche lezione

44

Classi e record

- le classi presentano due differenze fondamentali rispetto ai record del Pascal e alle strutture del C:
 - i campi possono essere funzioni
 - i campi non sono tutti **pubblici**
- una funzione **propria** di una classe, cioè definita in essa, può accedere anche ai campi privati di **tutti** gli oggetti della classe;
- una funzione **esterna** può accedere solo ai campi pubblici
- nota: in C++ le strutture (struct) hanno le stesse caratteristiche delle classi, con l'unica differenza che gli elementi sono pubblici, salvo diversa esplicita indicazione, mentre nelle classi sono privati, salvo ...

45

46

Dichiarazioni e definizioni

```
class NumeroComplesso {
public
    float Modulo();      // dichiarazione
    float Fase();        // dichiarazione
    float Reale();        // dichiarazione
    float Immaginaria(); // dichiarazione
private
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};

// definizioni delle funzioni
float NumeroComplesso::Modulo() { return sqrt(Re * Re + Im * Im); }
float NumeroComplesso::Fase() {
    if Im > 0 return acos(Re/Modulo());
    else return -acos(Re/Modulo());
}
float NumeroComplesso::Reale() { return Re; }
float NumeroComplesso::Immaginaria() { return Im; }
```

la distinzione è utile dal punto di vista metodologico (per esempio, in file diversi)

47

Assegnazioni e inizializzazioni

- Come si assegnano i valori agli oggetti?
- Come si inizializzano?

```
class NumeroComplesso {
public
    float Modulo() { return sqrt(Re * Re + Im * Im); }
    float Fase() {
        if Im > 0 return acos(Re/Modulo());
        else return -acos(Re/Modulo());
    }
    float Reale() { return Re; }
    float Immaginaria() { return Im; }
private
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

...

```
NumeroComplesso c;
c.Re = 2; // scorretto: Re e' privato!
```

48

- i linguaggi O-O prevedono specifiche funzioni (“costruttori”) per (la creazione e) l’inizializzazione di nuovi oggetti
- in C++, si possono definire funzioni con lo stesso nome della classe (e senza tipo per il risultato)
- possono essere più di una, purché le relative liste di argomenti siano diverse (in lunghezza o tipo) — un primo esempio di **overloading** o sovrapposizione (sovraccarico): più funzioni con lo stesso nome; quella da utilizzare viene individuata dal contesto
- debbono essere pubblici
- i costruttori monadici (o unari, con un solo argomento) sono anche **convertitori** di tipo

49

```
class NumeroComplesso {
public:
/* nuove funzioni nella parte pubblica */
    NumeroComplesso(float r, float i) {
        /* costruttore reale-immaginario */
        Re = r;
        Im = i;
    }
    NumeroComplesso(float r) {
        /* costruttore reale */
        Re = r;
        Im = 0.;
    }
    NumeroComplesso() {
        /* costruttore senza argomenti */
        Re = 0.;
        Im = 0.;
    }
    ...
private:
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

50

Utilizziamo i costruttori

```
int i = 5; // definizione e inizializzazione di variabile intera
NumeroComplesso c1 = NumeroComplesso(-3.2, 2.0);
NumeroComplesso c1(-3.2, 2.0); // equivale al precedente
NumeroComplesso c2 = NumeroComplesso(5.45);
NumeroComplesso c2(5.45); // equivale al precedente
NumeroComplesso c2 = 5.45; // equivale al precedente: converte
NumeroComplesso c3 = NumeroComplesso();
NumeroComplesso c3; // equivale al precedente
NumeroComplesso c3(); // non equivale al precedente
```

51

- costruttore di copia:

```
NumeroComplesso c2 = c1;
```

- costruttore senza argomenti e assegnazione:

```
NumeroComplesso c2;
c2 = c1;
```

- costruttore e assegnazione:

```
NumeroComplesso c2; // costruttore senza argomenti
c2 = NumeroComplesso(4.2); // costruttore e poi assegnazione
```

esiste per ogni classe un costruttore **standard**, senza argomenti, che lascia indefinite tutte le variabili; viene inibito dalla definizione dei costruttori espliciti

52

Costruttori e liste

```
#include <iostream.h>
class cell {
friend class list;
    cell(int i) { info = i; next = NULL; }
    cell(int i, cell* n) { info = i; next = n; }
    int    info;
    cell* next;
};
class list {
    cell* first;
    void destroy(cell *p) {
        p = first;
        while (p != NULL) {
            cell* tmp = p; p = p->next; delete tmp; cout << "cancello un elemento " << endl;
        }
    }
public:
    list() { first = NULL; } // costruttore
    ~list() { destroy(first); } // distruttore
    void insert(int i) { cell* p; p = new cell(i); p->next = first; first = p; }
    void print() {
        cell* p = first;
        while (p != NULL) { cout << p->info << " "; p = p->next; } cout << endl;
    }
};
main() {
    list L;
    L.insert(3);L.insert(4);L.insert(5);
    L.print();
}
```

53

Liste e distruttori

```
#include <iostream.h>
class cell {
friend class list;
    cell(int i) { info = i; next = NULL; }
    cell(int i, cell* n) { info = i; next = n; }
    int    info;
    cell* next;
};
class list {
    cell* first;
    void destroy(cell *p) {
        while (p != NULL) {
            cell* tmp = p; p = p->next; delete tmp; cout << "cancello un elemento " << endl;
        }
    }
public:
    list() { first = NULL; } // costruttore
    ~list() { destroy(first); first=NULL; } // distruttore
    void insert(int i) { cell* p; p = new cell(i); p->next = first; first = p; }
    void print() {
        cell* p = first;
        while (p != NULL) { cout << p->info << " "; p = p->next; } cout << endl;
    }
};
main() {
    list L;
    L.insert(3);L.insert(4);L.insert(5);
    L.print();
}
```

54

Funzioni pure e funzioni con “side-effect”

```
#include <iostream.h>

class NumeroComplesso {
public:
    NumeroComplesso PiuUno();
    NumeroComplesso AggiungiUno();
    void stampa() { cout << Re << '+' << Im << "i\n"; }
    NumeroComplesso(float r, float i) {
        /* costruttore reale-immaginario */
        Re = r;
        Im = i;
    }
    NumeroComplesso(float r) {
        /* costruttore reale */
        Re = r;
        Im = 0.;
    }
    NumeroComplesso() {
        /* costruttore senza argomenti */
        Re = 0.;
        Im = 0.;
    }
private:
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

55

Funzioni pure e con side effect

```
NumeroComplesso NumeroComplesso::PiuUno() {
    NumeroComplesso aux;
    aux.Re = Re + 1;
    aux.Im = Im;
    return aux;
};

NumeroComplesso NumeroComplesso::AggiungiUno() {
    NumeroComplesso aux;
    Re = Re + 1;
    aux.Im = Im;
    aux.Re = Re;
    return aux;
};

void main() {
    NumeroComplesso c1 (-3.2,2.0);
    c1.stampa();
    (c1.AggiungiUno()).stampa();
    c1.stampa();
    (c1.PiuUno()).stampa ();
    c1.stampa ();
}
```

56

Con solo side-effect

```
void NumeroComplesso::AggiungiUno() {
    Re = Re + 1;
};

NumeroComplesso NumeroComplesso::AggiungiUno() {
    Re = Re + 1;
    return *this;
};
```

57

Funzioni proprie costanti Oggetti costanti

```
NumeroComplesso NumeroComplesso::PiuUno() const
```

La definizione viene accettata dal compilatore solo se nel corpo non vi sono assegnazioni ai campi dell'oggetto di invocazione.

- Gli oggetti costanti possono essere utilizzati da funzioni costanti.
- Gli oggetti costanti non possono essere utilizzati da funzioni non costanti.
- Le funzioni costanti sono utili dal punto di vista metodologico

58

Elementi propri delle classi (non di ciascun oggetto) Variabili static

```
#include <iostream.h>

class Persona {
public:
    Persona (char* , int);
    ~Persona ();
    static int NPersone;
private:
    char Nome[10];
    int Eta;
};

Persona::Persona (char* UnNome, int UnEta) {
    strcpy(Nome,UnNome);
    Eta = UnEta;
    NPersone++;
};

Persona::~Persona () { NPersone--; }
int Persona::NPersone = 0;

void main() {
    Persona a ("Nico",23);
    Persona* b = new Persona("Marco",34);
    cout << Persona::NPersone << endl; // 2
    delete b;
    cout << Persona::NPersone << endl; // 1
}
```

59

Funzioni static

```
#include <iostream.h>
class Persona {
public:
    Persona (char* , int);
    ~Persona ();
    static int quantePersone();
private:
    char Nome[10];
    int Eta;
    static int NPersone;
};

Persona::Persona (char* UnNome, int UnEta) {
    strcpy(Nome,UnNome);
    Eta = UnEta;
    NPersone++;
}

Persona::~Persona () { NPersone--; }
int Persona::NPersone = 0;
int Persona::quantePersone(){ return NPersone; }

void main() {
    Persona a ("Nico",23);
    Persona* b = new Persona("Marco",34);
    cout << Persona::quantePersone() << endl; // 2
    delete b;
    cout << Persona::quantePersone() << endl; // 1
}
```

60