Implementazione

La scrittura del codice può essere complessa a causa dei seguenti fattori:

- i progettisti non hanno considerato compiutamente le idiosincrasie della piattaforma hw e dell'ambiente di programmazione sw
- tabelle e diagrammi di progetto non si traducono in codice in maniera diretta
- il codice deve essere comprensibile non solo da parte di chi lo ha scritto e dovrà testarlo ma anche a coloro che ne seguiranno l'evoluzione
- si deve creare codice riusabile e nel contempo trarre vantaggio dalle caratteristiche di organizzazione, strutture dati e costrutti del linguaggio di programmazione

Procedure e standard

I progetti sw sono condotti in team, quindi agli altri deve essere possibile capire non solo che cosa si è scritto nel codice ma anche perché lo si è scritto e come si colloca nel quadro complessivo del progetto

↓

Molte aziende (Microsoft in primis) si dotano di standard e procedure da seguire durante la codifica, con lo scopo di adottare uno stile, un formato e un contenuto uniformi

Standard adottati dall'autore per l'autore stesso

Servono per

- organizzare i pensieri
- evitare errori

Comprendono:

a) Metodi di documentazione che consentono di
   - abbandonare per un po' il lavoro e poi ritornarci senza perderne traccia
   - chiarire quali sezioni del programma svolgono quali funzioni (così da potere localizzare errori e compiere cambiamenti)

b) Metodi di strutturazione del codice che mantengono la corrispondenza fra
   - componenti del design e del codice (cosicché le modifiche del design possano tradursi facilmente nel codice)
   - componenti hw e/o specifiche di interfaccia e componenti del codice (cosicché le modifiche esterne possano tradursi facilmente nel codice, minimizzando gli errori)

Standard adottati dall'autore per gli altri

Il commento introduttivo come quello del lucido seguente può notevolmente aiutare gli altri a capire le funzionalità del componente sw

Tale commento consente

- A chi cerca un componente da riusare, di decidere se l'ha trovato o meno
- A chi cerca la sorgente di un malfunzionamento, di stabilire se il componente è un indiziato o meno
- Al manutentore, di trovare facilmente il componente da cambiare

```
/**
 * Component to find intersection of two lines.
 *
 * Component name: findpt
 *
 * Programmer: J. Smith
 *
 * Version: 1.0 (February 2, 2002)
 *
 * Procedure invocation:
 *
 *      findpt(a1, b1, c1, a2, b2, c2, xs, ys, flag)
 *
 * Input parameters:
 *
 *      input lines are of the form:
 *
 *         a1 * x + b1 * y + c1 = 0  and
 *         a2 * x + b2 * y + c2 = 0
 *
 *      so input is coefficients a1, b1, c1, and a2, b2, c2
 *
 * Output parameters:
 *
 *      if lines are parallel, flag is set to 1
 *      else flag = 0 and point of intersection is (xs, ys)
**/
```

Standard adottati dall'autore per gli altri (cont.)

Strumenti automatici di analisi del codice possono elencare le procedure invocate dal componente e che invocano lo stesso → questa informazione è utile quando si apportano modifiche al componente

Gli standard devono

- Estendere la modularità e i suoi principi (massima coesione e minimo accoppiamento, interfacce ben definite, information hiding, ecc.) dal progetto al codice; ad es. l'accoppiamento fra componenti può essere sottolineato usando nomi di parametri e commenti adeguati agli stessi
- Registrare la corrispondenza fra progetto e codice (testing, manutenzione, gestione delle configurazioni sono altrimenti impossibili)

Linee guida e pratiche generali dell'ingegneria del sw

Aspetti principali di ogni componente sw:

- strutture di controllo
- algoritmi
- strutture dati

Strutture di controllo

- Molte sono suggerite dall'architettura e dal progetto e vanno preservate nel codice
- La struttura di controllo del programma deve riflettere quella del progetto
- Il flusso di controllo nel programma dovrebbe procedere linearmente dall'altro al basso, senza salti, in modo che il lettore possa concentrarsi su ciò che il programma fa (e non sulla struttura di controllo)

Difficile da seguire	Facile da seguire

`benefit = minimum;`
`if (age < 75) goto A;`
`benefit = maximum;`
`goto C;`
`A:  if (age < 65) goto B;`
`benefit = benefit * 1.5 + bonus;`
`goto C;`
`B:  if (age < 55) goto C;`
`benefit = benefit * 1.5;`
`goto C;`
`C: …`
	`if (age < 55) benefit = minimum;`
	`else if (age < 65) benefit = minimum * 1.5;`
	`else if (age < 75) benefit = minimum * 1.5 + bonus;`
	`else benefit = maximum;`

Algoritmi

Il design lascia una certa discrezionalità al programmatore nell'implementazione di algoritmi, in particolare per quanto concerne prestazioni ed efficienza

Costi nascosti dell'efficienza:

- codice più efficiente è anche più complesso da scrivere
- tale complessità rende più difficile il testing
- la sua comprensibilità è ridotta
- la sua modificabilità è ridotta

Se la velocità di esecuzione è un requisito importante, è necessario che il programmatore sappia come il compilatore ottimizza il codice (es. il tempo di accesso a un array tridimensionale effettivo può essere inferiore rispetto a quello a un array monodimensionale che lo simula)

Strutture dati

La scelta delle strutture dati in supporto alla computazione può rendere il programma notevolmente più chiaro/oscuro (un programma chiaro è più facile da comprendere, testare e modificare)

Esempio: tabella delle tasse

I primi Euro 10000 sono tassati al 10%
I successivi Euro 10000 sono tassati al 12%
I successivi Euro 10000 sono tassati al 15%
L'ammontare che eccede Euro 30000 è tassato al 20%

Scaglione	Base	Percentuale
0	0	10
10000	1000	12
20000	2200	15
30000	3700	20

Strutture dati (cont.)

Oscuro	Chiaro

`tax = 0;`
`if (income <= 10000)`
`  tax = tax + 0.10*income;`
`else`
`  {`
`  tax = tax + 1000;`
`  if (income <= 20000)`
`  tax = tax + 0.12*(income –`
`  10000);`
`  else`
`  {`
`  tax = tax + 1200;`
`  if (income <= 30000)`
`  tax = tax + 0.15*(income –`
`  20000);`
`  else`
`  tax = tax+1500+(income –`
`  30000)*0.20;`
`  }`
`  }`
	`for (int i = 2,level = 1;i <= 4; i++)`
	`  if (income > scaglione[i])`
	`  level++;`
	
	`tax = base[level]+`
	`percentuale[level]/100 *`
	`(income – scaglione[level]);`

Strutture dati (cont.)

Le strutture dati possono influenzare:

- il modo in cui si effettuano i calcoli
- l'organizzazione e il flusso di un programma
- la scelta del linguaggio (ad es. strutture dati ricorsive sono elaborate meglio da procedure ricorsive)

Manipolazione degli errori

Programmazione OO
Eccezione = condizione che, quando si verifica, determina il passaggio del controllo a una parte speciale del codice, detta manipolatore (handler) dell'eccezione, che ripara il guasto o almeno sposta il sistema in uno stato più accettabile

Progettazione per contratto
Il contratto spesso include specifici comportamenti di manipolazione delle eccezioni

La disponibilità di costrutti espliciti nel linguaggio per la gestione delle eccezioni rende più chiara la parte del codice dedicata alle condizioni di errore e la disaccoppia dal resto del programma

È necessario adottare nel sistema una strategia uniforme di manipolazione degli errori

Localizzazione di input e output

Le parti del programma che leggono i dati di ingresso o scrivono i dati di uscita sono altamente specializzate, dipendono da hw e sw (e per questo sono talvolta difficili da testare) e tendono a cambiare notevolmente nel tempo.

Confinare le funzioni di input e di output rispettivamente in due componenti specializzate, separate dal resto del codice, dove magari la componente di input effettua anche compiti di formattazione dei dati o di controllo dei tipi degli ingressi, migliora l'organizzazione funzionale del sistema e rende più generale il servizio che esso offre

Inclusione di pseudocodice

- Il design è solo il profilo di ciò che deve essere fatto in ciascun componente del programma → nella traduzione del design in codice si procede per stadi, passando attraverso lo pseudocodice (e più possibili versioni successive dello stesso) → lo pseudocodice può aiutare ad adattare progressivamente il design al linguaggio di programmazione; tipicamente, lo pseudocodice può essere risistemato e ricostruito con sforzo minimo, consentendo così di sperimentare soluzioni implementative diverse prima di decidere
- Se l'organizzazione del design viene modificata, tali cambiamenti devono essere approvati dai progettisti e riportati nel design
- Lo pseudocodice può essere incluso come uno o più commenti

Revisioni e riscritture

Mentre si scrive il programma spesso si rivedono e riscrivono varie parti, fino a quando si è soddisfatti del risultato.

Se il flusso di controllo è convoluto, le strutture dati non sono intuitive o ci sono molte dipendenze tra le parti è bene riconsiderare il design e rivedere il codice anziché aggiungere patch

Riuso

- Riuso del produttore (ovvero produzione di elementi riusabili)
- Riuso del consumatore (ovvero utilizzo di componenti sviluppati per altri progetti o di COTS)

Alcune aziende adottano standard per il riuso

Riuso del consumatore

È necessario valutare:

- le funzionalità del componente
- la documentazione di progetto allegata al componente, cosicché non sia necessario verificare l'implementazione linea per linea,
- la documentazione relativa al testing e alla storia delle revisioni, che certifichino l'assenza di guasti
- le difficoltà di interfacciare il proprio codice con il componente e di effettuare eventuali modifiche minori

Depositi internazionali per il riuso:

- ASSET
- CARDS
- COSMIC
- DSRS
- Software Technology Support Center at Hill Air Force Base

Riuso del produttore

È necessario:

- assicurare la generalità del componente, che dovrà anticipare le future condizioni di utilizzo
- ridurre e isolare le dipendenze da altri componenti
- definire un'interfaccia generale del componente
- adottare convenzioni per i nomi che ne garantiscano la significatività
- documentare strutture dati, algoritmi di base e interfaccia
- documentare e separare le modalità di comunicazione con altri componenti
- separare la gestione degli errori

Documentazione di programma

Spiega cosa fa il programma e come lo fa

Documentazione	Dove è scritta	A chi è diretta
interna → concisa → prospettiva = singolo componente	entro il codice, tipicamente ne cosiddetto *header comment block* (le voci contenute e il loro ordine sono specificate dagli standard aziendali)	A chi leggerà il codice sorgente (programmatori, manutentori, squadra preposta al testing, ecc.)
esterna → prospettiva = intero sistema	altrove	A chi leggerà il codice sorgente come pure a chi non lo leggerà mai (es. progettisti che devono introdurre modifiche o avanzamenti di progetto)

Header comment block

Contenuto tipico:

- nome del componente
- nome e indirizzo dell'autore e suoi puntatori (utili per testing e manutenzione)
- collocazione logica e fisica (gerarchia dei componenti) entro il progetto
- date di stesura e revisioni e indicazione degli autori di queste ultime
- ragion d'essere
- modalità di utilizzo di strutture dati, algoritmi e controllo e modalità di invocazione
- nome, tipo e scopo delle strutture dati
- flussi logici, algoritmi e gestione degli errori
- ingressi attesi e uscite prodotte
- istruzioni per l'uso
- indicazioni per il testing
- estensioni e revisioni

Header comment block: esempio

```
PROGRAM SCAN: Program to scan a line of text for a given
              Character

PROGRAMMER: Beatrice Clarman (718) 345-6789/bc@power.com

CALLING SEQUENCE: CALL SCAN(LENGTH, CHAR, NTEXT)
        where LENGTH is the length of the line to be scanned;
        CHAR is the character sought. Line of text is passed
        as array NTEXT.

VERSION 1: written 3 November 1997 by B. Clarman

REVISION 1.1: 5 December 1997 by B. Clarman to improve
              searching algorithm.

PURPOSE: General-purpose scanning module to be used for each
    new line of text, no matter the length. One of several
    text utilities designed to add a character to a line of
    text, read a character, change a character, or delete a
    character.

DATA STRUCTURES: Variable LENGTH - integer
     Variable CHAR - character
     Array NTEXT - character array of length LENGTH

ALGORITHM: Reads array NTEXT one character at a time; if
     CHAR is found, position in NTEXT returned in variable
     LENGTH; else variable LENGTH set to 0.
```

Identificatori

I nomi che identificano classi, metodi, funzioni, attributi, etichette, tipi e variabili devono essere significativi

↓

- Il bisogno di commenti sparsi si riduce
- Si riduce la probabilità di errore

Esempio:

```
weekwage = hrrate * hours + 1.5 * hrrate * (hours - 40);

z = a * b + 1.5 * a * (b - 40);
```

Formattazione, indentazione e spaziatura

- L'indentazione aiuta a comprendere il flusso di controllo e le condizioni gestite dal programma
- La spaziatura rende più leggibili le espressioni

↓

si riduce il bisogno di commenti

- L'allineamento dei commenti a destra del codice
  - ✓ne aumenta la leggibilità
  - ✓non costringe chi analizza il codice (ad es. chi effettua il testing) a leggere anche i commenti → evita la polarizzazione

Altri commenti: necessità

Un codice è ben strutturato, che riflette il design, che usa nomi auto-descrittivi è autoesplicativo → bastano pochi commenti addizionali sparsi per il codice che suddividano la computazione in fasi maggiori e via via in attività minori (è così che si impiega lo pseudocodice)

Altri commenti: regole

- I commenti devono riflettere il comportamento corrente effettivo del codice
- I commenti devono essere scritti mentre si scrive il codice, non dopo
- Se il codice cambia, anche i commenti devono cambiare (manutenzione)
- I commenti devono essere funzionali all'evoluzione futura del sistema

Commento inutile	Commento utile	Nome significativo che elimina la necessità di commenti

`// incrementa i`
 	
`i = i + 1;`
	`// inizializza il contatore per`
	`// leggere il caso successivo`
	`i = i + 1;`
		`contatore = contatore + 1;`

Documentazione delle strutture dati

Una mappa dei dati serve a:

- spiegare l'uso e la struttura dei dati e come i valori dei dati vengono modificati
- mettere in relazione i dati con il dizionario dei dati appartenente alla documentazione esterna

Documentazione esterna: motivazioni e rassegna

Spiegazione

- dell'organizzazione del codice
- dei dati
- di funzioni e controllo
- delle decisioni implementative e delle loro ragioni

Rassegne a livello di sistema:

- Elenco (testuale e/o grafico) dei componenti del sistema o dei raggruppamenti degli stessi
- Passaggio delle informazioni fra componenti
- Classi ed ereditarietà, ragioni per cui sono stati definiti certi tipi di dati

Documentazione esterna: sezioni

Per ogni singolo componente

1) Descrizione del problema risolto, delle alternative esplorate, della ragione della scelta di soluzione adottata
2) Descrizione degli algoritmi e delle loro condizioni limite; riferimenti a libri e articoli scientifici da cui gli algoritmi sono stati tratti. Descrizione dei casi gestiti, delle assunzioni fatte sugli input ($\rightarrow$ casi non gestiti) e della trattazione degli errori
3) Descrizione dei dati: flussi dati e interazioni dinamiche tra oggetti

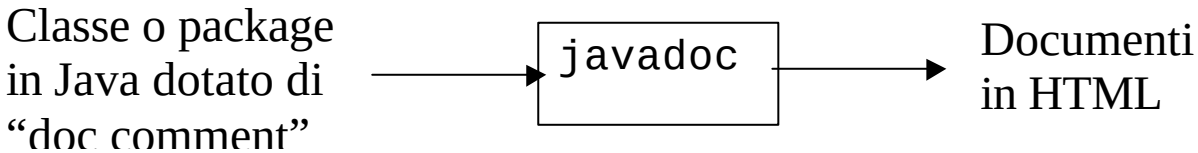
Conclusione: standard di programmazione in un progetto

- Regole sui nomi
- Stile del codice e dei commenti
- Organizzazione dei sottosistemi in file e direttori
- Documentazione

Auto-documentazione: il programma `javadoc`

È scritto in Java e fornito con JDK; è teso a

- Minimizzare il tempo necessario alla produzione della documentazione
- Minimizzare il rischio di avere disallineamenti tra codice e documentazione
- Produrre documentazione con un aspetto uniforme
- Produrre documentazione facilmente consultabile in formato sia elettronico, sia cartaceo
- Produrre documentazione in modo indipendente dalla piattaforma e da strumenti proprietari




```
/**
* questo è un doc comment; ogni doc comment può contenere dei tag
*/
```

Tag

È un comando per la creazione della documentazione

- @see *argomento*: richiesta di creazione di un'etichetta "See also" seguita da hyperlink verso *argomento*, dove ques'ultimo può essere una classe, un metodo (*Nome_classe#nome_metodo*) o un attributo (*Nome_classe#nome_attributo*)
- @version *versione*: richiesta di aggiunta di un'etichetta "Version *versione*", dove *versione* è un testo
- @author *autore*: richiesta di aggiunta di un'etichetta "Author *autore*", dove *autore* è un testo
- @param *nome_parametro descrizione* … : richiesta di aggiunta di un'etichetta "Parameters … elenco di coppie (*nome_parametro descrizione*)"
- @return *descrizione*: richiesta di aggiunta di un'etichetta "Returns *descrizione*", dove *descrizione* è un testo
- @exception *Nome_classe_eccezione descrizione* … : richiesta di aggiunta di un'etichetta "Throws" con hyperlink verso *Nome_classe_eccezione* e la *descrizione*

Documenti in uscita a `javadoc`

- Indice di metodi e variabili
- Gerarchia delle classi
- Se un metodo ne sovrascrive uno di una superclasse, viene riportato il commento “Overrides”
- Etichette richieste dal programmatore