Progettazione

- É il ponte fra la specifica dei requisiti e la codifica
- É la fase in cui si decide come passare da “che cosa” deve essere fatto a “come” deve essere fatto
- La sua uscita principale si chiama architettura (o progetto) del sw

Architettura = definizione della struttura statica del sistema sw in termini di componenti e loro reciproche interconnessioni

Componente = parte di un sistema che fornisce risorse e servizi computazionali; può essere costituita a sua volta da componenti

Nella progettazione OO una classe è il più piccolo componente implementabile

Architettura

- Interna: si riflette nella suddivisione delle funzioni tra i vari componenti sw
- Esterna: si traduce nella suddivisione dei componenti tra le risorse hw interessate e nelle interfacce verso l'hw

Architettura esterna

- Monolitica (o stand-alone): il sistema viene eseguito su un solo computer, senza comunicare via rete con altre parti del sistema
- Distribuita: il sistema è costituito da più applicazioni, operanti su macchine distinte

Client/server: le applicazioni sono collaboranti e ciascuna di esse implementa o un client o un server. I server funzionano su macchine a prestazioni elevate che spesso gestiscono una base di dati o un file system oppure sono collegati a una periferica (es. stampante). I client funzionano su macchine a prestazioni limitate (es. PC) e sfruttano i servizi dei server → la soluzione è vantaggiosa (anche) dal punto di vista dell'efficienza

Peer-to-peer: tutte le applicazioni sono dotate di funzioni e responsabilità simili, ciascuna di esse offre servizi e usa quelli delle altre

Master/slave (primario/secondario): l'applicazione master (o primaria) mantiene il controllo su tutte le applicazioni slave (o secondarie). Uno slave può chiedere al master un servizio, il master decide se concederglielo→ pericolo di starvation. Si adotta per coordinare la gestione di risorse scarse

Obiettivi della modularizzazione

o, meglio, dei principi che la guidano:

- dominare la complessità (la somma delle complessità delle singole parti deve essere minore della complessità originaria)
- ridurre i tempi di produzione (grazie alla distribuzione e parallelizzazione del lavoro)
- favorire il riuso sistematico
- migliorare i fattori di qualità del sw

Progettazione architetturale

Moduli

- Modulo = parte di un sistema sw che fornisce un insieme di servizi ad altri moduli
- Servizio = elemento computazionale che gli altri moduli possono usare

Interfacce

Un modulo consiste di:

- Corpo = implementazione e suoi segreti
- Interfaccia = insieme dei servizi esportabili, definisce un contratto fra il modulo e i moduli suoi utenti; gli utenti conoscono solo l'interfaccia di un modulo

Relazioni (tutte irriflessive)

- USA: un modulo usa i servizi esportati da un altro
- È-UN-COMPONENTE-DI: descrive l'aggregazione di moduli in moduli di livello più alto
- EREDITA-DA: per sistemi OO

Principi di modularizzazione

Progetto per il cambiamento

- Anticipare i cambiamenti
- Non concentrarsi sulle esigenze attuali ma prevedere la loro evoluzione (prototipazione evolutiva)
- Pensare al programma come al membro di una famiglia

Cambiamenti: verosimiglianza

- Degli algoritmi (ad es. per ragioni di efficienza)
  - Es. da ordinamento per affioramento a ordinamento per selezione
- Delle strutture dati (sia per ragioni di efficienza, sia per cambiamenti dei requisiti)
  - Es. dati dell'utente: 17% dei costi di manutenzione
- Funzionali (cambiamenti dei requisiti)
  - Es. l'introduzione dell'applicazione genera nuovi bisogni e modifica quelli esistenti, oppure si scoprono errori nei requisiti
- Della macchina astratta sottostante
  - Es. periferiche hw, OS, DBMS, ecc. ($\rightarrow$ nuovi rilasci, problemi di portabilità, ecc.)
- Dell'ambiente
  - Es. anno 2000, euro

Cambiamenti: rischi

Passare da un progetto ordinato, ben documentato, riusabile, con codice pulito a un progetto contorto, non riusabile, con documentazione inconsistente, con codice “spaghetti”

Cambiamenti: dove e come

- L’interfaccia di un modulo è un contratto con i clienti e, come tale, deve essere stabile
- Se le parti modificabili sono quelle segrete del modulo, il loro cambiamento non deve avere effetti per i clienti
- Minimizzare il flusso di informazioni di un modulo verso i clienti
- Minimizzare la presenza della relazione USA

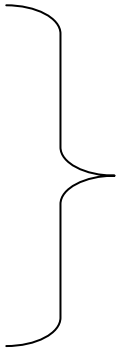
Cambiamenti: un esempio

Una TABELLA in cui è possibile inserire, cancellare, modificare e stampare voci (secondo un ordine prestabilito)

Interfaccia: INSERT, DELETE, MODIFY e PRINT

Si possono cambiare liberamente:

- Le strutture dati
- La politica di gestione dei dati (mantenere le voci ordinate oppure ordinarle solo prima di stamparle)
- Gli algoritmi (ad es. l'algoritmo di ricerca)



Questi sono i segreti di un modulo

Famiglia di programmi

Obiettivo del fatto di pensare al programma come al membro di una famiglia: progettare l'intera famiglia anziché ogni membro separatamente

Ad es. sistema di biglietteria

- Ferroviaria: formulazione interattiva del piano del viaggio (magari suddiviso in più tratte), scelta della classe e del posto da parte dell'utente, calcolo del prezzo, eventuale acquisto on-line, eventuale prenotazione, scelta fra modalità di pagamento diverse, invio messaggio di posta elettronica
- Marittima: molte funzionalità sono simili ma alcune diverse (ad es. possibilità di portare l'auto al seguito)

Principi di modularizzazione

Occultamento delle informazioni (Parnas 1974): “define what you wish to hide and design a module around it”

- Un modulo è un’unità logica autocontenuta
- Entro questa unità è necessario distinguere fra ciò che un modulo fa per gli altri e come lo fa (i suoi segreti)
- Il modulo deve essere un firewall intorno ai suoi segreti
- I segreti devono essere incapsulati e protetti, ovvero l’accesso agli stessi deve essere filtrato dall’interfaccia

Altri principi e concetti chiave della modularizzazione

- Massima coesione e minimo accoppiamento
- Progetto per il riuso
- Astrazione
- Estensibilità
- Strutturazione a macchine virtuali

Astrazione

Tipo di modulo	Equivalente
Operazione astratta	Procedure della programmazione procedurale
Oggetto astratto (ad es. il modulo TABELLA) – Un modulo che incapsula una struttura dati – Esporta un insieme di operazioni – L'applicazione delle operazioni modifica lo stato dell'oggetto astratto	
Tipo di dato astratto – Un modulo che consente l'istanziazione di oggetti astratti	Classe della programmazione OO (+ classi astratte + interface)

Circa la relazione USA

Utilizzare solo una gerarchia di USA, non un grafo ciclico, perché è più facile

- da comprendere
- da sviluppare e verificare (se non è una gerarchia, si sviluppa un sistema in cui non funziona niente fino a quando non funziona tutto)

I livelli della gerarchia diventano i livelli di astrazione del sistema

Classi progettuali

Derivano da tre fonti:

- dominio dell'applicazione
- infrastruttura
- interfaccia utente

Sono:

- Classi dell'interfaccia utente (classi di presentazione)
- Classi del dominio (logica del dominio): sono frequentemente raffinamenti delle classi del modello concettuale del dominio
- Classi dei processi: rappresentano le astrazioni operative di basso livello necessarie per gestire al meglio le classi del dominio
- Classi persistenti: rappresentano archivi di dati
- Classi di sistema: consentono al sistema di comunicare con l'ambiente di calcolo e il mondo esterno

Progettazione per contratto

- Tecnica sviluppata da Bertrand Meyer nel 1992 come caratteristica del linguaggio Eiffel
- Rifinisce un principio di progettazione noto, particolarmente adatto per la progettazione OO

Contratto = accordo fra un cliente e un contraente

Progettazione per contratto	Progettazione OO
Modulo contraente	Classe contraente
Contratto	Interfaccia della classe contraente
Modulo cliente	Classe cliente (che USA la classe contraente)

Un esempio di contratto nel mondo reale

	Obblighi	**Benefici**
Cliente	♦ Fornire un terreno di dimensioni minime prefissate ♦ Mettere a disposizione un punto di prelievo dall'impianto idraulico ♦ Pagare una certa somma secondo modalità di versamento prefissate	Disporre di una piscina funzionante, di dimensioni e caratteristiche note, allacciata all'impianto idraulico
Contraente	♦ Costruire una piscina funzionante, di dimensioni e caratteristiche note, effettuando l'allacciamento all'impianto idraulico ♦ Nessun obbligo se gli impegni assunti dal cliente non sono rispettati	Ricevere, secondo le modalità prefissate, i versamenti della somma di denaro pattuita

Il contratto per un metodo OO

	Obblighi	**Benefici**
Metodo cliente	Precondizioni = ciò che è richiesto dal metodo contraente	♦ Il servizio computazionale reso dal modulo contraente ♦ Conoscere le condizioni soddisfatte dai risultati, senza bisogno di controllarle
Metodo contraente (o servente)	Postcondizioni = ciò che deve essere fornito dal metodo contraente	Non dovere considerare tutte le possibili configurazioni d'ingresso (principio di progettazione)

Progettazione per contratto (cont.)

Usa tre tipi di asserzioni (cioè espressioni booleane che possono risultare false solo in presenza di errori di programmazione):

- precondizioni
- postcondizioni

} Ciascuna precondizione o postcondizione è associata specificamente a un singolo metodo e, in generale, è diversa per ogni metodo

- invarianti

Ciascun invariante è associato specificamente a una singola classe e, in generale, è diverso per ogni classe

Precondizioni

Se `p` è la precondizione del metodo servente `m`, o scriviamo nel codice del metodo cliente (per un qualsiasi oggetto `x`)

```
if x.p
   then x.m
   else …{trattamento speciale}
```

o, ragionando sul programma, assicuriamo che `p` sia vera (per ogni oggetto `x`) prima della chiamata

Attenzione: per consentire ai clienti del metodo `m` contenuto nella classe `K` di verificare la sua precondizione `p`, è necessario che `p` sia espressa in termini di operazioni esportabili di `K`

Precondizioni (cont.)

- Scelta delle precondizioni = decisione di progetto per cui non esistono regole guida
- Precondizioni deboli implicano che tutte le complicazioni sono delegate al metodo ($\rightarrow$ difficoltà di sviluppo)

Postcondizioni

Servono a specificare cosa fa un'operazione senza dire come lo fa, cioè a separare l'interfaccia dall'implementazione

Invariante

Invariante = proprietà che ogni istanza di una classe deve soddisfare

- dopo la sua creazione
- sia prima, sia dopo ogni operazione compiuta sull'istanza stessa (ma non necessariamente durante)

Rappresenta un ulteriore obbligo che la classe deve soddisfare con la sua implementazione

Può essere aggiunto alle precondizioni e postcondizioni di ciascuna operazione di una classe

Proprietà interne di una classe

Esempio: classe tabella

- Invariante: n_elementi ≤ capacità
- Operazione *inserisci(elemento)*
  Precondizione: (n_elementi < capacità) AND (*elemento* non è in tabella)
  Postcondizioni:
  ✓*elemento* è presente nella tabella
  ✓n_elementi' = n_elementi + 1

Le asserzioni

- esplicitano le responsabilità dei moduli, evitando così ridondanza di controlli o controlli insufficienti
- risolvono i conflitti di responsabilità in presenza di errori

Eccezione

Si verifica a run time quando un'operazione (viene invocata nel rispetto della sua precondizione ma) non è in grado di terminare la propria esecuzione nel rispetto della postcondizione

N.B. in questa definizione sia precondizione che postcondizione sono comprensive dell'invariante della classe

Progettazione difensiva

Si anticipano i rischi di fallimento di un modulo e si decide se evitarli o tollerarli

Eccezione = evento attraverso cui il modulo servente, prima di terminare la sua esecuzione, segnala al modulo cliente che non è riuscito a completare correttamente il servizio richiesto ($\rightarrow$ distinzione fra terminazione normale e terminazioni anomale). Il modulo cliente risponde gestendo l'eccezione in modo appropriato

I tipi di eccezioni sollevate da un modulo fanno parte della sua interfaccia

Le modalità di gestione delle eccezioni ricevute da un modulo fanno parte dei segreti di tale modulo

Asserzioni ed ereditarietà

Le sottoclassi possono

- Aggiungere attributi e metodi, rispettando però il vincolo semantico
  $inv_{sottoclasse} \rightarrow inv_{classe}$
- Ridefinire i metodi, soddisfando però i seguenti vincoli semantici:
  $post_{sottoclasse} \rightarrow post_{classe}$
  $pre_{classe} \rightarrow pre_{sottoclasse}$

In questo modo le asserzioni impediscono che le operazioni ridefinite o aggiunte nella sottoclasse siano inconsistenti con quelle della classe padre

Uso ideale ed effettivo delle asserzioni

Ideale

Le asserzioni dovrebbero essere incluse sia nel progetto, sia nel codice come parte della definizione di interfaccia

Effettivo

- Le asserzioni vengono incluse nel codice per il testing e il debugging e vengono rimosse quando si compila il codice finale
- Il programma deve ignorare la presenza del controllo delle asserzioni
- L'uso delle sole precondizioni spesso consente di rilevare il massimo numero di errori, con il minimo spreco di risorse computazionali
- Le asserzioni sono parte integrante di Eiffel e recentemente anche di altri linguaggi, fra cui Java

Asserzioni in Java

Sintassi:

`assert` *condizione*

`assert` *condizione : spiegazione*

Esempio.

```
assert count > 0 : "violated precondition size() > 0"
```

Semantica:

se *condizione* è vera l'esecuzione procede, altrimenti viene lanciata un'eccezione di tipo `AssertionError` la cui gestione visualizza un msg d'errore (nome_file, #linea, stringa *spiegazione*)

Asserzioni in Java (cont.)

Introdotte nella versione 1.4 di Java SDK

Il controllo delle asserzioni può essere abilitato e disabilitato (il meccanismo dipende dall'ambiente di esecuzione), anche in forma selettiva (ovvero limitatamente ad alcune classi/package)

Pre/postcondizioni e invarianti non esprimibili come asserzioni Java

{n > 0}

procedure search (table: **in** integer_array; n: **in** integer;
element: **in** integer; found: **out** Boolean);

{found ≡ (**exists** i (1 ≤ i ≤ n **and** table(i) = element))}

{n > 0 }

procedure reverse (a: **in out** integer_array; n: **in** integer);

{**for all** i (1≤ i ≤ n) **implies** (a' (i) = a (n – i + 1))}

Per il controllo di queste postcondizioni è necessaria l'esecuzione di programmi ciclici

Java Modeling Language (JML) www.eecs.ucf.edu/~leavens/JML

Es. L'algoritmo di ricerca binaria effettua correttamente la ricerca solo se l'array di ingresso è ordinato in ordine non decrescente

```
/*@ requires vet != null
  @          && (\forall int i;
  @              0 < i && i < vet.length;
  @              vet[i - 1] <= vet[i])
  @*/

int binarySearch(int[] vet, int x) { … }
```

Nota: l'uso di queste specifiche evita controlli difensivi inefficienti all'interno del codice Java

Programma Java contenente specifiche JML

- È traducibile da `jmlc`, estensione del compilatore Java SDK 1.4. Il byte code risultante include i controlli run-time di verifica delle condizioni espresse in JML (disabilitabili quando il programma va in produzione). L'esecuzione dei controlli delle asserzioni è trasparente, fatta eccezione per le prestazioni
- È interpretabile direttamente da `jmlrac`
- È verificabile da `jmlunit`, strumento di testing che estende `Junit` combinandolo col compilatore JML, che controlla se il comportamento del codice in corrispondenza dei casi di test soddisfa le condizioni espresse in JML (ovvero solleva i programmatori dalla scrittura di oracoli)
- È utilizzabile da `jmldoc`, strumento che genera documentazione esterna HTML comprendente sia i commenti Javadoc, sia le specifiche formali espresse in JML
- È analizzabile da `escjava2` (extended static checker), che può trovare difetti quali puntatori potenzialmente nulli o violazioni dei limiti di array. Esso sfrutta e propaga le specifiche JML

Istruzioni JML

Sono controllabili da `jml`, che sostituisce `jmlc` nel caso non sia necessario compilare il codice Java

Tutti i tool citati sono disponibili gratuitamente

È disponibile anche un plugin per l'IDE Eclipse che supporta la sintassi JML e si interfaccia a vari tool che fanno uso di annotazioni JML

JML: Clausole relative a un metodo

Precedono l'intestazione del metodo a cui si riferiscono

`requires`: precondizione

`ensures`: postcondizione (normale, cioè senza lancio di eccezioni)

`signals_only`: elenco delle eccezioni lanciabili dal metodo (per default, tutte e sole quelle contemplate nella clausola Java `throws` del metodo)

`signals`: condizioni particolari di terminazione (ad es. relative al lancio di un'eccezione), dette anche postcondizioni eccezionali. Es.

```
/*@ signals (IllegalArgumentException e) x < 0;
  @*/
```

`assignable`: elenca tutti e soli gli attributi a cui è possibile assegnare un valore all'interno del metodo

JML: Specifica di un metodo

Sintassi	Semantica
`\old(E)`	vecchio valore dell'espressione E (cioè valore della stessa appena dopo il passaggio dei parametri)
`\result`	risultato dell'invocazione di un metodo
`pure`	inserito nell'intestazione di un metodo, asserisce che tale metodo non modifica lo stato del programma. È la forma concisa di `assignable \nothing`. N.B. solo i metodi pure possono essere richiamati nelle asserzioni e nelle specifiche JML

JML: Clausole relative a una classe

È contenuta entro il codice Java della classe

`invariant`: invariante

JML: Operatori e altro

Sintassi	Semantica
`//@`	introduce codice JML presente sul resto della riga
`a ==> b`	a implies b
`a <== b`	b implies a
`a <==> b`	a iff b
`a <=!=> b`	`!(a <==> b)`
`nullable`	asserisce che un attributo può essere nullo
`non_null`	asserisce che un attributo o un parametro di un metodo non può essere nullo
`spec_public`	rende pubblico nelle specifiche JML un attributo privato del codice Java
`public`	rende pubblico nelle specifiche JML l'invariante di una classe

Per gli operatori logici (and, or, not) si adotta la sintassi Java. Nelle specifiche JML si possono includere istruzioni `assert` secondo la sintassi Java

JML: Visibilità

La visibilità della specifica JML di un metodo è la stessa del metodo specificato

La visibilità della specifica JML di un invariante di classe è per default quella di package ma può essere cambiata attraverso il modificatore `public`

Specifiche pubbliche possono menzionare solo nomi pubblici. Un attributo privato secondo il codice Java diventa pubblicamente visibile nelle specifiche JML solo se preceduto dall'annotazione `spec_public`

JML: Quantificatori

Hanno la forma generale

(`\quantifier_name` *declaration; [range predicate;] body predicate of the quantifier*)

dove *range predicate*, che è opzionale, restringe il dominio delle variabili quantificate definito da *declaration* per cui vale *body predicate*

Sintassi	**Quantificatore/i**
`\forall`	universale
`\exists`	esistenziale
`\min`, `\max`, `\product`, `\sum`	generalizzati (ritornano rispettivamente il minimo, il massimo, il prodotto e la somma dei valori dell'espressione *body predicate* quando le variabili quantificate soddisfano *range predicate*)
`\num_of`	numerico (ritorna il numero di valori delle variabili quantificate per cui *range predicate* e *body predicate* sono entrambi veri)

JML: Esempi con quantificatori

Le seguenti due condizioni hanno lo stesso significato

```
(\forall Student s; juniors.contains(s); s.getAdvisor() != null)
```

```
(\forall Student s; juniors.contains(s) ==> s.getAdvisor() != null)
```

(in questo secondo caso il *range predicate* è omesso)

```
(\sum int x; 1 <= x && x <= 5; x)
```

ritorna il valore 15

Specifiche informali

```
public class Person {
  private String name;
  private int weight;

  /*@ also
    @ ensures \result != null &&
    @    (* \result is a displayable
    @       form of this person *);
  public String toString() {
     return "Person(\" + name +
       "\", " + weight + ")";
  }

  public int getWeight() {
     return weight;
  }
```

```
  public void addKgs(int kgs) {
     if (kgs >= 0) {
        weight += kgs;
     } else {
        throw new IllegalArgumentException();
     }
  }

  public Person(String n) {
     name = n; weight = 0;
  }
}
```

Attualmente JML non supporta la specifica formale dell'interfaccia utente e dell'I/O in generale (lettura e scrittura su file). Pertanto tale specifica può essere solo informale

Specifiche formali

```
public class Person {
   private /*@ spec_public non_null @*/ String name;
   private /*@ spec_public @*/ int weight;

   //@ public invariant !name.equals("") && weight >= 0;
   /*@ also
     @ ensures \result != null;
     @*/
   public String toString();

   //@ also ensures \result == weight;
   public /*@ pure @*/ int getWeight();
```

Specifiche formali (cont.)

```
    /*@ also
      @ ensures kgs >= 0 && weight == \old(kgs + weight);
      @ signals (Exception e) kgs < 0 &&
      @ (e instanceof IllegalArgumentException);
      @*/
    public void addKgs(int kgs);

    /*@ also
      @ requires !n.equals(“”);
      @ ensures n.equals(name) && weight == 0;
      @*/
    public Person(/*@ non_null @*/ String n);
}
```