Progettazione – parte III

Leggere sez. 4.2.5 - 4.8

Ghezzi et al

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

Progettazione per il cambiamento

- Tecniche di implementazione che la supportano
- Identificare gli aspetti che possono cambiare e rendere il sistema parametrico rispetto a essi
  - Costanti di configurazione
  - Compilazione condizionale
  - Generazione di codice

Costanti di configurazione

- a: array (1..10) of integer

```
...
for i = 1 to 10
 print a(i)
```

- E se la dimensione cambiasse?
- Quando le modifiche da fare sono sparse per il codice il processo è complesso e porta a errori

Costanti di configurazione

- Individuare valori passibili di cambiamento e definirli per mezzo di *costanti di configurazione*
- Supportato da molti linguaggi.
- Es: #define in C

```
#define N 10
int a[N];
for (i=0; i<N; i++)
    ...
```

Compilazione condizionale

- Tutti i software di una famiglia in un solo codice sorgente.
- Macro comandi specificano il codice rilevante per ciascuna versione
- La selezione della versione avviene per mezzo di parametri di configurazione

Compilazione condizionale

- Frammento di sorgente comune a tutte le versioni

```
#ifdef hardware-1
...frammento di codice per l'hardware 1 ...
#endif
#ifdef hardware-2
...frammento di codice per l'hardware 2 ...
#endif
```

- La scelta avviene per mezzo di una direttiva #define

Generazione del software

- Il codice sorgente in un linguaggio di programmazione è generato a partire da un testo in un altro linguaggio
- Es. generazione di compilatori (yacc in ambiente Unix)
- Se il linguaggio target cambia, non occorre riscrivere il codice, ma solo rigenerarlo

Raffinamento per passi successivi

- Processo iterativo di sviluppo
- Dato un problema da risolvere:
  1. si scompone in sottoproblemi
  2. si risolvono i sottoproblemi (eventualmente scomponendoli!)
  3. si compongono le soluzioni per mezzo di strutture semplici di controllo

Raffinamento per passi successivi

- Un problema P si può risolvere come:

1. P_1; P_2; ... ; P_n
2. if C then P_1
   else P_2
   end if
1. while C loop
   P_1
   end loop;

Raffinamento per passi successivi

- Notazione compatta per if annidati:

case
  C_1: P_1;
  C_2: P_2;
  ...
  C_N: P_N;
  otherwise P_0;
end case

Esempio: selection sort

- Passo 1

sia n la lunghezza dell'array da ordinare;
  i := 1 ;
  while i < n loop
   trova il minimo di a(i), ..., a(n) e scambialo con l'elemento in posizione i;
   i := i + 1;
  end loop;

Esempio: selection sort

- Passo 2

sia n la lunghezza dell'array da ordinare;
  i := 1 ;
  while i < n loop
    j := n; while j > i loop
      if a(i) > a(j) then
        scambia gli elementi nelle posizioni i e j ;
      end if;
      j := j - 1;
    end loop;
    i := i + 1;
  end loop;

Esempio: selection sort

- Passo 3

sia n la lunghezza dell'array da ordinare;

```
i := 1 ;
while i < n loop
  j := n; while j > i loop
          if a(i) > a(j) then
               x := a(i); a(i) := a(j);  a(j) := x;
          end if;
          j := j - 1;
      end loop;
 i := i + 1;
end loop;
```


Albero di scomposizione

- Il raffinamento per passi successivi si può rappresentare con *un albero di scomposizione* (DT) in cui
  - la radice rappresenta il problema principale
  - i nodi figli di un nodo rappresentano i sottoproblemi in cui è scomposto
  - l'ordine sinistra-destra dei nodi figli rappresenta la sequenzialità

Albero di scomposizione

- L'alternativa fra sottoproblemi è rappresentata con una linea tratteggiata che interseca gli archi che connettono i nodi corrispondenti al padre, etichettati con la condizione;
- L'iterazione è rappresentata con una linea continua etichettata con la condizione

Esempio

Passo 1

P; — *P problema da risolvere*

Passo 2

P_1; P_2; P_3; — *P scomposto in una sequenza*

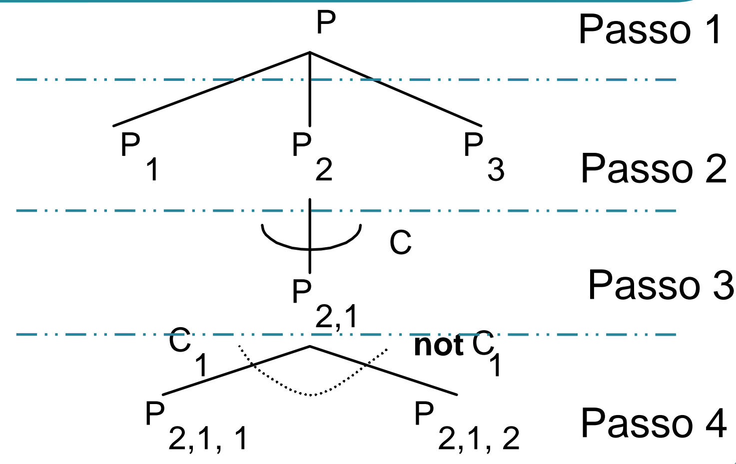
Passo 3

P_1;
while C loop
    $P_{2,1}$; — *P2 scomposto in un'iterazione*
end loop;
P_3;

Step 4

P_1;
while C loop
    if C_1 then — *$P_{2,1}$ scomposto in un'alternativa*
        $P_{2,1,1}$;
    else
        $P_{2,1,2}$;
    end if;
end loop;
P_3;

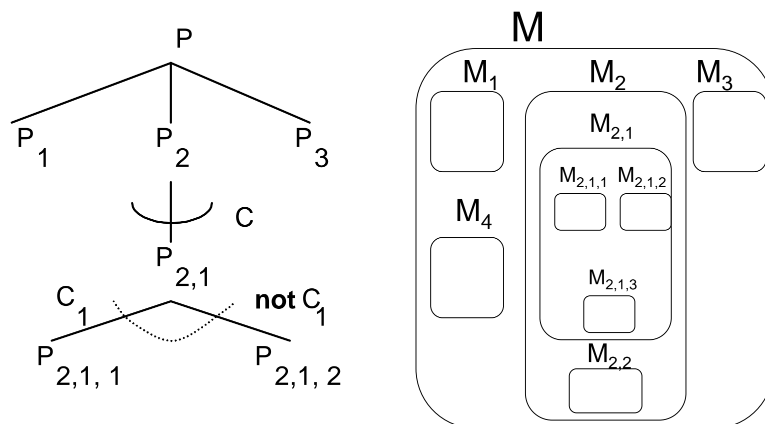
Esempio



Relazione con IS_COMPOSED_OF

- Dato un albero di scomposizione, si può ottenere una gerarchia IS_COMPOSED_OF che lo implementa:
  - Ogni nodo è implementato da un modulo
  - Il modulo che implementa un nodo è scomposto nei moduli che implementano i nodi figli, più un modulo che implementa il flusso di controllo fra i figli.

Esempio



M

M_1 M_2 M_3

$M_{2,1}$

$M_{2,1,1}$ $M_{2,1,2}$

M_4

$M_{2,1,3}$

$M_{2,2}$

Valutazione del raffinamento per passi successivi

- Accettabile come tecnica di programmazione, non di modularizzazione
- Porta a un'analisi dei singoli problemi che trascura i punti comuni
- Trascura l'information hiding

Valutazione del raffinamento per passi successivi

- Trascura la strutturazione dei dati
- Presuppone l'esistenza di una funzione che risolve il problema
  - il problema può non ammettere una funzione come soluzione
- Fissa prematuramente il flusso di controllo fra moduli

Esempio: analizzatore sintattico

- Passo 1:

riconosci un programma immagazzinato in un file f;

- Passo 2:

```
correct := true;
 analizza f secondo la definizione del linguaggio
if correct then
 print message "programma corretto";
else
 print message "programma non corretto";
end if;
```


Esempio: analizzatore sintattico

- Passo 3:

```
correct := true;
   esegui analisi lessicale:
    immagazzina il programma come sequenza di token in un file f_t e la
   tabella dei simboli nel file f_s e assegna  error_in_lexical_phase
   if error_in_lexical_phase then
    correct := false;
   else
    esegui analisi sintattica e assegna error_in_syntactic_phase:
    if error_in_syntactic_phase then
        correct := false;
    end if;
   end if;
   if correct then
    print message "programma corretto";
   else
    print message "programma non corretto";
   end if;
```


Decisioni premature

- L'analizzatore lessicale opera su tutto il file, producendo come risultato due file su cui in seguito opererà l'analizzatore sintattico
- Se si volesse che l'analizzatore sintattico invocasse l'analizzatore lessicale quando necessario, bisognerebbe rivedere l'intera soluzione

Progetto alternativo

- Moduli basati su information hiding
- CHAR_HOLDER:
  - nasconde la rappresentazione fisica del sorgente
  - esporta operazioni per accedere al sorgente un carattere alla volta
- SCANNER:
  - nasconde la struttura lessicale del linguaggio
  - esporta operazioni per fornire il successivo token
- PARSER:
  - nasconde la struttura dati utilizzata per memorizzare il programma

Top-down e bottom-up

- Il raffinamento per passi successivi è top-down.
- L'information hiding è bottom-up.
- I due approcci si possono combinare.
- Indipendentemente dall'approccio alla progettazione, la documentazione è più comprensibile se strutturata top-down.

Gestione delle anomalie

- Qualsiasi sistema può fallire
- Non per questo è necessariamente difettoso
  - Utilizzo scorretto
  - Mancata erogazione di servizi da cui il sistema dipende
- Necessario prevedere i fallimenti e gestirli

Gestione delle anomalie

- Un modulo M è in uno stato *anomalo* se non fornisce il servizio richiesto secondo la sua interfaccia
- Può accadere se
  - Il client di M non rispetta il protocollo
  - M non rispetta il protocollo di un suo server
  - Condizione non prevista in M

Gestione delle anomalie

- Rilevata l'anomalia, il modulo può passare il controllo al suo *exception handler*, parte di codice che può
  - cercare di gestire l'anomalia, in modo da evitare il fallimento
  - riportare il modulo in uno stato normale e segnalare l'anomalia al client (*sollevare un'eccezione*)

Esempio

```
module M
exports . . .
        procedure P (X: INTEGER; . . .)
              raises X_NON_NEGATIVE_EXPECTED,
                 INTEGER_OVERFLOW;
                  X is to be positive; if not, exception
                  X_NON_NEGATIVE_EXPECTED is raised;
                  INTEGER_OVERFLOW is raised if internal
                  computation of P generates an overflow
        .
        .
        .
end M
```


Esempio

- Propagazione delle eccezioni

```
module L
uses M imports P (X: INTEGER; . .)
exports . . .;
        procedure R ( . . .)
              raises INTEGER_OVERFLOW;
        .
        .
implementation
        If INTEGER_OVERFLOW is raised when P is invoked, the
        exception is propagated
        .
        .
end L
```


Esempio: compilatore MIDI

- Linguaggio strutturato a blocchi
- Necessita di modulo di gestione della tabella dei simboli in grado di funzionare con l'annidamento dei blocchi
- Consideriamo il modulo SYMBOL_TABLE

Es. (sintassi C)

```
int main (void)
{ int i = 1;
...
{  int i;
   i = 0;
   printf("%d\n",i);}     /* stampa 0 */
printf("%d\n",i);         /* stampa 1 */
...}
```

Ipotesi

- SYMBOL_TABLE funziona solo con programmi corretti:
  - Blocchi correttamente delimitati da simboli di begin e end;
  - Non esistono due identificatori con lo stesso nome nello stesso blocco
  - Le variabili vengono dichiarate prima dell'utilizzo
  - La profondità di annidamento non supera un valore prefissato

SYMBOL_TABLE (versione 1)

```
module  SYMBOL_TABLE
        Supports up to MAX_DEPTH block nesting levels
uses ... imports (IDENTIFIER, DESCRIPTOR)
exports procedure  INSERT (ID: in IDENTIFIER;
                           DESCR: in DESCRIPTOR);
        procedure  RETRIEVE (ID:in IDENTIFIER;
                           DESCR: out DESCRIPTOR);
        procedure  LEVEL (ID: in IDENTIFIER; L:out INTEGER);
        procedure  ENTER_SCOPE;
        procedure  EXIT_SCOPE;
        procedure  INIT (MAX_DEPTH: in INTEGER);
end SYMBOL_TABLE
```

La versione 1 non è robusta

- I client potrebbero violare il protocollo
- Vanno sollevate eccezioni in questi casi:
  - INSERT cerca di aggiungere un identificatore già presente
  - RETRIEVE e LEVEL cercano di accedere a un identificatore non specificato
  - ENTER_SCOPE supera il livello massimo di annidamento
  - EXIT_SCOPE viene invocata non all'interno di un blocco

SYMBOL_TABLE – Versione 2

```
module SYMBOL_TABLE
uses ... imports (IDENTIFIER, DESCRIPTOR)
exports
        Supports up to MAX_DEPTH block nesting levels; INIT
        must be called before any other operation is invoked
        procedure INSERT (ID: in IDENTIFIER;
                          DESCR: in DESCRIPTOR)
                          raises MULTIPLE_DEF,
        procedure RETRIEVE (ID:in IDENTIFIER;
                          DESCR: out DESCRIPTOR)
                          raises NOT_VISIBLE;
        procedure LEVEL (ID: in IDENTIFIER;
                          L: out INTEGER)
                          raises NOT_VISIBLE;
        procedure ENTER_SCOPE raises EXTRA_LEVELS;
        procedure EXIT_SCOPE raises EXTRA_END;
        procedure INIT (MAX_DEPTH: in INTEGER);
end SYMBOL_TABLE
```


Modulo generico di gestione di liste

```
generic module LIST(T) with MATCH (EL_1,EL_2: in T)
exports
        type LINKED_LIST:?;
        procedure IS_EMPTY (L: in LINKED_LIST): BOOLEAN;
        Tells whether the list is empty.
        procedure SET_EMPTY (L: in out LINKED_LIST);
        Sets a list to empty.
        procedure INSERT (L: in out LINKED_LIST; EL: in T);
        Inserts the element into the list
        procedure SEARCH (L: in LINKED_LIST; EL_1: in T;
                          EL_2: out T; FOUND: out boolean);
                Searches L to find an element EL_2 that
                matches EL_1 and returns the result in FOUND.
end LIST(T)
```


Sistemi concorrenti

- Finora, sistemi sequenziali, cioè con un solo flusso di esecuzione (*control thread*)
- I sistemi concorrenti hanno più flussi di esecuzione indipendenti
- Nuovi problemi per l’accesso ai dati condivisi
- Necessità di progettazione apposita

Esempio

- Oggetto astratto BUFFER
  - module QUEUE_OF_CHAR is GENERIC_FIFO_QUEUE (CHAR)
  - BUFFER : QUEUE_OF_CHAR.QUEUE
- Con le operazioni
  - PUT: inserisce un carattere in BUFFER
  - GET: estrae un carattere da BUFFER
  - NOT_FULL: ritorna true se BUFFER non è pieno
  - NOT_EMPTY: ritorna true se BUFFER non è vuoto

Accesso al buffer

- Il client può eseguire un controllo prima dell'accesso ai dati: es. scrittura

```
if QUEUE_OF_CHAR.NOT_FULL (BUFFER) then
   QUEUE_OF_CHAR.PUT (X, BUFFER);
end if;
```

- ma questo controllo può non essere sufficiente, se il client non è l'unico ad accedere al buffer

Accesso concorrente

- C_1 e C_2 vogliono accedere al buffer (che ha una sola cella libera) in scrittura
- Questa sequenza causa un fallimento:
  - C1 controlla (NOT_FULL(BUFFER) = true)
  - C2 controlla (NOT_FULL(BUFFER) = true)
  - C1 scrive (occupando l'ultima cella)
  - C2 scrive (fallimento)

Sincronizzazione

- Le operazioni sul buffer devono essere eseguite in mutua esclusione
- Operazioni come

```
if QUEUE_OF_CHAR.NOT_FULL
   (BUFFER) then
   QUEUE_OF_CHAR.PUT (X, BUFFER);
end if;
```

devono essere eseguite come operazioni atomiche non interrompibili

Monitor

- Oggetti astratti disponibili in ambiente concorrente
- Definiti da:
  - Dati privati
  - Blocco di inizializzazione
  - Procedure accessibili ai client
- E' garantito l'accesso esclusivo al monitor

Estensione a TDN

- Alle procedure si può associare una clausola **requires** che, alla chiamata della procedura da parte di un client, viene controllata.
  - Se il risultato è true, il client continua l'esecuzione della procedura (sempre in mutua esclusione)
  - Se il risultato è false, il client viene sospeso finchè la condizione non ritorna true

Esempio

```
concurrent module   CHAR_BUFFER
        This is a monitor, i.e., an abstract object module in a
        concurrent environment
uses  . . .
exports
        procedure  PUT (C : in CHAR) requires  NOT_FULL;
        procedure  GET (C: out CHAR) requires  NOT_EMPTY;
        NOT_EMPTY and NOT_FULL are hidden Boolean
        functions yielding TRUE if the buffer is not empty and not
        full, respectively. They are not exported as operations,
        because their purpose is only to delay the calls to PUT and
        GET if they are issued when the buffer is in a state where it
        cannot accept them
        .
        .
        .
end  CHAR_BUFFER
```


Monitor generico

```
generic concurrent module    GENERIC_FIFO_QUEUE (EL)
        This is a generic monitor type, i.e., an abstract data type
        accessed in a concurrent environment
uses  . . .
exports
        type  QUEUE: ?;
        procedure  PUT (Q1: in out  QUEUE; E1: in  EL)
              requires  NOT_FULL (Q1: QUEUE);
        procedure  GET (Q2: in out  QUEUE; E2: out  EL)
              requires  NOT_EMPTY(Q2: QUEUE);
        .
        .
        .
end  GENERIC_FIFO_QUEUE (EL)
```


Guardiani e rendezvous

- A differenza dei monitor, meccanismo attivo
- Un guardiano è un task, sempre in esecuzione, che riceve dai client le richieste di accesso alle risorse condivise.
- Il client viene sospeso finchè il guardiano non accetta la richiesta (rendezvous), e la esegue in mutua esclusione
- Stessa notazione usata per i monitor (ma semantica diversa)

Esempio: implementazione di un guardiano

```
loop
        select
            when NOT_FULL
                accept PUT (C: in CHAR);
                This is the body of PUT; the client calls it as if it
                were a normal procedure
                end ;
            or
            when NOT_EMPTY
                accept GET (C: out CHAR);
                This is the body of GET; the client calls it as if it
                were a normal procedure
            end ;
        end select ;
end loop ;
```


Deadlock

- La mutua esclusione può portare al blocco del sistema (*deadlock*).
- Due processi:
  - A sospeso su require X
  - B sospeso su require Y

  Se Y può essere resa vera solo da A e X solo da B, allora sia A che B saranno bloccati indefinitamente.
- L'analisi deve individuare e risolvere queste situazioni.

Software real-time

- Non è sempre possibile sospendere indefinitamente un processo
- Es. prelievo di un valore da un sensore
- Sistema real time: sistema per il quale la correttezza della risposta dipende anche dal tempo impiegato per produrla.

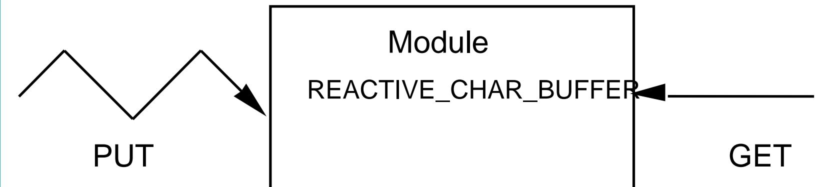
Notazione

- Commenti per esprimere vincoli temporali
- Procedure reattive: rispondono a stimoli esterni e non possono essere ritardate
- Parola chiave **reactive** in TDN
- Freccia a zig-zag in GDN

Esempio TDN

```
concurrent module   REACTIVE_CHAR_BUFFER
This is a monitorlike object working in a real-time environment.
uses . . .
exports
        reactive procedure   PUT (C: in CHAR);
        PUT is used by external processes, and two consecutive
        PUT requests must arrive more than 5 msec apart;
        otherwise, some characters may be lost
        procedure  GET (C: out CHAR);
        .
        .
        .
end REACTIVE_CHAR_BUFFER
```


Esempio GDN



Software distribuito

- Nuovi problemi:
  - Vincoli modulo-macchina
  - Comunicazione fra moduli
    - Non aree condivise
    - Chiamate a procedura remota
    - Messaggi
  - Accesso efficiente agli oggetti astratti
    - Replicazione, distribuzione

Modello client-server

- Il modello distribuito più diffuso
- I moduli server forniscono servizi ai moduli client
- Server e client possono essere eseguiti su macchine diverse

Vincoli modulo-macchina

- Possono essere
  - Rigidi: es. server di stampa
  - Dettati da prestazioni o riduzione dei costi: moduli server “vicini” ai client
- Possono essere
  - Statici: gestione più semplice
  - Dinamici: più complessi da gestire, ma permettono bilanciamento di carico e tolleranza ai guasti

Comunicazione fra moduli

- Chiamata a procedura remota: si possono progettare le applicazioni senza distinguere fra servizi locali e remoti
- Differenze:
  - Prestazioni
  - Passaggio parametri: puntatori non consentiti
- Messaggi: comunicazione asincrona

Replicazione e distribuzione

- Costo elevato dell’accesso a dati remoti
- Possibilità per rendere efficiente l’accesso a oggetti astratti:

1. Replicazione
   - Oggetto presente su più macchine
   - Copie dell’oggetto vanno mantenute coerenti in caso di modifiche
2. Distribuzione
   - L’oggetto, logicamente unico, viene fisicamente partizionato e le parti posizionate vicine ai client che più probabilmente vi accederanno.

Middleware

- Strato intermedio fra le funzionalità di rete del sistema operativo e le applicazioni
- Fornisce funzionalità normalmente richieste dalle applicazioni basate su reti:
  - Servizi basati su nomi: astrazione dalla locazione fisica di processi e risorse
  - Servizi di comunicazione: messaggi, procedure remote

Progettazione orientata agli oggetti

- Un solo tipo di modulo: il tipo di dato astratto (*classe*).
- Una classe esporta le operazioni (*metodi*) con cui è possibile manipolare le sue istanze (*oggetti*).
- Le istanze sono accessibili per mezzo di riferimenti.

Cambiamenti sintattici in TDN

- Non si distingue più tra nome del modulo e nome del tipo esportato: entrambi coincidono con la classe
- Se **a** è un riferimento a una classe e **op** è un'operazione che ha **a** come parametro, si scrive

  a.op(<altri_parametri>)

  anziché

  op(a, <altri_parametri>)

Ereditarietà

- Gerarchia fra classi data dalla relazione di *specializzazione* (inv. *generalizzazione*)
- La classe B *specializza* la classe A (*eredita* da A) se ha tutti i membri di A, più altri
- B è *sottoclasse* di A, A è *superclasse* di B.

Esempio

```
class EMPLOYEE
exports
        function FIRST_NAME(): string_of_char;
        function LAST_NAME(): string_of_char;
        function AGE(): natural;
        function WHERE(): SITE;
        function SALARY: MONEY;
        procedure HIRE (FIRST_N: string_of_char;
                        LAST_N: string_of_char;
                        INIT_SALARY: MONEY);
        Initializes a new EMPLOYEE, assigning a new identifier.
        procedure FIRE();
        procedure ASSIGN (S: SITE);
        An employee cannot be assigned to a SITE if already assigned to it (i.e., WHERE
        must be different from S). It is the client's responsibility to  ensure this. The effect is to
        delete the employee from those in WHERE, add the employee to those in S, generate
        a new id card with security code to access the site overnight, and update WHERE.
end EMPLOYEE
```

Esempio

```
class ADMINISTRATIVE_STAFF inherits EMPLOYEE
exports
        procedure DO_THIS (F: FOLDER);
        This is an additional operation that is specific to
        administrators; other operations may also be added.
end ADMINISTRATIVE_STAFF

class TECHNICAL_STAFF inherits EMPLOYEE
exports
        function GET_SKILL(): SKILL;
        procedure DEF_SKILL (SK: SKILL);
        These are additional operations that are specific to
        technicians; other operations may also be added.
end TECHNICAL_STAFF
```

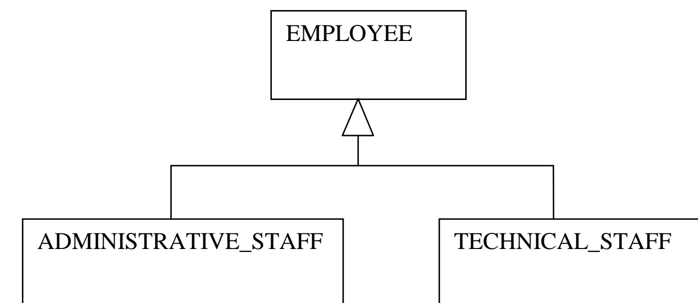

Ereditarietà

- Una sottoclasse è un sottotipo
  - Sostituibilità
- Polimorfismo: se B eredita da A, una variabile di tipo riferimento ad A può riferirsi anche a un'istanza di B
- Dynamic binding: il metodo invocato attraverso un riferimento dipende dal tipo dell'oggetto associato al riferimento a run-time

Rappresentazione grafica

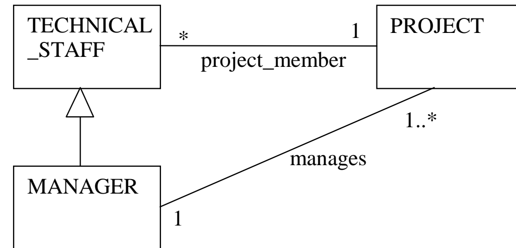
- Unified Modeling Language
- Insieme di formalismi grafici standard, ampiamente utilizzato nella progettazione orientata agli oggetti
- Gerarchie di classi rappresentate con un *diagramma delle classi*.

Ereditarietà in UML



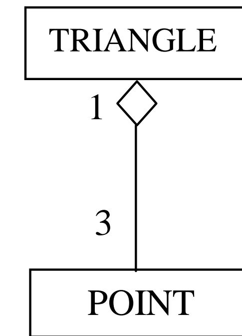
Associazioni

- Relazioni fra istanze di classi che l'implementazione deve rispettare
- Possono avere vincoli di molteplicità



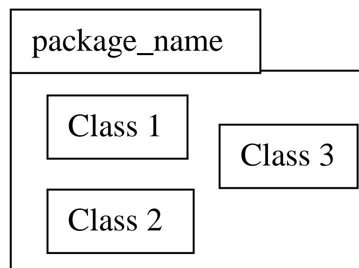
Aggregazione

- Relazione PART_OF
- Differisce da IS_COMPOSED_OF in quanto TRIANGLE ha metodi propri non forniti da POINT



Package

- Rappresentano la relazione IS_COMPOSED_OF



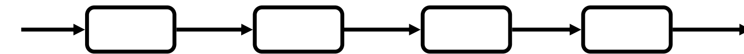
Architettura

- Definisce l'organizzazione generale del sistema
- Influenza molte delle qualità del sistema
- Va scelta tenendo conto dei vincoli di progetto
  - costi
  - compatibilità

Utilizzo delle architetture standard

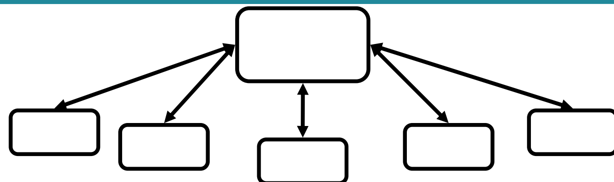
- Le architetture standard rappresentano le esperienze precedenti dei progettisti nei vari campi applicativi.
- Fungono da specifiche delle interfacce per i componenti, favorendo lo sviluppo di componenti standard e il loro riuso.

Pipeline



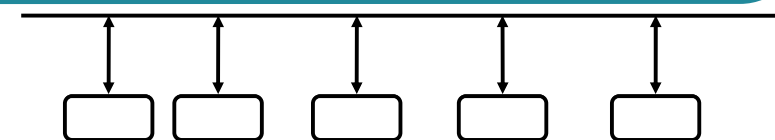
- Ogni sistema
  - riceve input dal precedente
  - lo elabora
  - passa l'output al successivo
- Elaborazione di dati
- Detta anche *pipe-and-filter* (UNIX)

Blackboard



- Necessità di comunicazioni non locali
- Uno dei sottosistemi (blackboard) funge da mezzo di comunicazione per gli altri
- Gli altri sottosistemi possono chiedere di scrivere o leggere informazioni.

Architettura basata su eventi

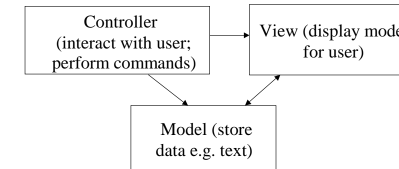


- I componenti possono creare *eventi* o reagire a essi
- Modello *publish-subscribe*.
- Adatti a sistemi che devono attendere input o in cui le relazioni client-server non sono chiare

Architetture specifiche al dominio

- Le architetture viste (e altre simili) astraggono le proprietà strutturali dei sistemi e trascurano i dettagli dei domini.
- Altre (specifiche al dominio) fanno ipotesi derivate dalla conoscenza su particolari domini applicativi.
- Rendono più veloce lo sviluppo e facilitano il riuso di componenti

Model-View-Controller



- Per software interattivi
- Tiene separati
  - Rappresentazione dei dati (modello)
  - Presentazione per l'utente (vista)
  - Gestione dei comandi (controllore)
- Es. Swing, Ruby on Rails

Componenti software

- Scopo: costruire le applicazioni a partire da componenti pronti
- Primo esempio: librerie di funzioni matematiche
- Successo dovuto a:
  - Interfaccia chiara (facilità d'uso)
  - Servizio ben definito
  - Dominio di applicabilità evidente

Componenti software

- Dagli anni Novanta, diffusione sempre maggiore per tutti i tipi di servizi. Ad esempio:
  - STL C++: algoritmi e strutture dati facilmente combinabili
  - JavaBeans: si possono comporre in modo visuale
  - Librerie per GUI

Architetture per integrazione di componenti

- Progettazione in due fasi:

1. Scelta dell'organizzazione generale del sistema (specifica dei componenti)
2. Reperimento dei componenti

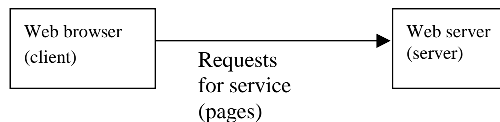
- Le due fasi si influenzano a vicenda
- La progettazione architetturale può ridursi all'assemblaggio di componenti per ottenere le funzionalità richieste.

CORBA

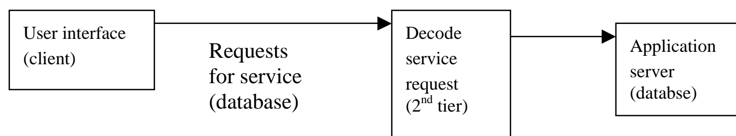
- Common Object Request Broker Architecture, architettura generale client-server in ambiente distribuito
- Object Request Broker:
  - I server comunicano la disponibilità a ORB
  - I client formulano le richieste a ORB
  - ORB mette in comunicazione client e server
- Interfacce definite in un Interface Definiton Language

Architetture per sistemi distribuiti

Due livelli



Tre livelli



Architetture a tre livelli

1. Livello client
2. Livello di business logic:
   - riceve le richieste del client, le decodifica e determina l'azione da eseguire
   - riceve le risposte dall'application server, la elabora e gira il risultato al client
3. Livello di application server (spesso DBMS): risponde alle richieste formulate dal livello

Architetture: linee guida

- Non è sempre possibile rifarsi a un modello architetturale pronto
- Dovendo produrre un'architettura originale, è opportuno seguire alcune linee guida.
- Scelta dell'architettura determinata da requisiti non funzionali

Architettura e requisiti non funzionali

- Prestazioni
  - Operazioni critiche confinate in un piccolo numero di sottosistemi
  - Minimizzazione della comunicazione fra sottosistemi (componenti grandi)
- Sicurezza
  - Architettura a strati, con le risorse più critiche all'interno

Architettura e requisiti non funzionali

- Safety
  - Operazioni pericolose in un piccolo numero di sottosistemi per facilitare la verifica
- Disponibilità
  - Ridondanza per facilitare la sostituzione "a caldo"
- Mantenibilità
  - componenti piccoli, autonomi, facili da modificare, no memoria condivisa