Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

PROGETTAZIONE DEL SOFTWARE

(Ing. Gestionale)

Prof. Giuseppe De Giacomo & Monica Scannapieco

Anno Accademico 2003/04

LA FASE DI PROGETTO E REALIZZAZIONE: DA UML A JAVA

Quarta parte

La fase di progetto e realizzazione

L'input di questa fase è costituito da:

- lo **schema concettuale**, formato da:
  - diagramma delle classi e degli oggetti
  - diagramma degli use case
  - diagramma degli stati e delle transizioni
- la **specifica**
  - una specifica per ogni classe
  - una specifica per ogni use case

Diagramma delle classi realizzativo

Il diagramma delle classi che si è costruito durante la fase di analisi, che chiameremo **diagramma delle classi concettuale**, è una concettualizzazione delle infomazioni di interesse per la nostra applicazione che però prescinde da considerazioni tecnologiche.

Ora per potere tradurre tale diagramma in in codice in modo efficace, dobbiamo innanzitutto prendere in esame alcuni aspetti su come la nostra applicazione utilizzarà le informazioni rappresentate dal diagramma.

In particolare dobbiamo decidere alcuni aspetti fondamentali per la realizzazione:

- Se gli attributi sono mutabili o immutabili

- Data una associazione, quali classi hanno responsabilità sulla stessa, nel senso che possono "navigare" (reperire le tuple) o aggiornare l'associazione (vedi dopo)

- Fornire ai clienti delle classi operazioni atte a verificare l'ammissibilità dello stato degli oggetti rispetto al diagramma delle classi concettuale (vedi dopo)

Tali decisioni sono formalizzate in un nuovo diagramma delle classi detto **diagramma delle classi realizzativo**.

Tale diagramma contiene tutte le informazioni necessarie alla realizzazione del codice, le uniche scelte che rimangono da fare sono scelte programmative.

Nota: il diagramma delle classi realizzativo è **peggiore** del diagramma cdelle classi concettuale, in quanto si sono effettuate delle scelte influenzate da tecnicalità realizzative, che servono per la realizzazione.

Traduzione in Java

Dobbiamo fare quanto segue:

1. **Traduzione dei tipi** (in genere, si usano tipi predefiniti o classi di libreria, ma in casi particolari si possono realizzare classi Java per rappresentare tipi particolari – vedi dopo).

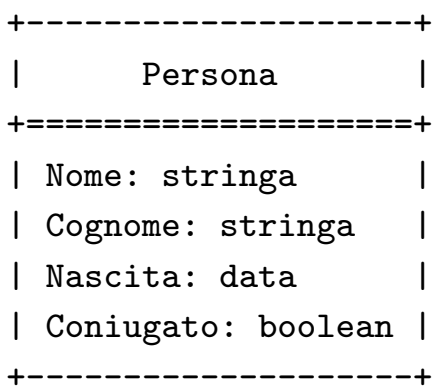
2. **Traduzione delle classi** (tipicamente, una classe Java per ogni classe UML).

3. Nel tradurre le classi, si procede alla:

   - **realizzazione delle associazioni**,
   - **realizzazione delle generalizzazioni**.

Singola classe UML con soli attributi

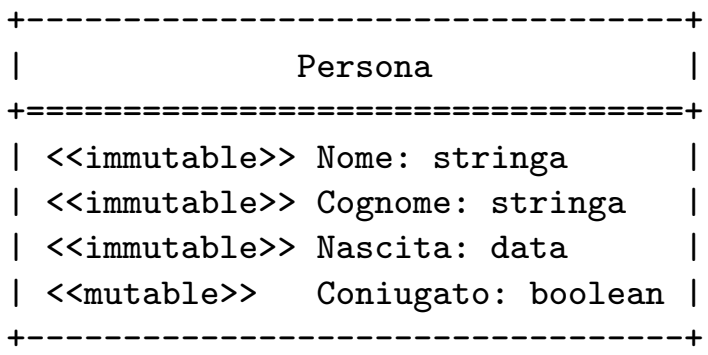
Iniziamo con considerare diagrammi delle classi concettuali semplici: formati da una sola classe



Supponiamo che il nome, il cognome, e la data di nascita di una persona **non cambiano** mente l'essere coniugato **cambia**...

Diagramma delle classi realizzativo

... allora il diagramma delle classi realizzativo è il seguente:



Dove abbiamo esplicitato quali attibuti non possono cambiare (**immutable**) e quali possono (**mutable**).

Una volta stabilito questo possiamo passare alla realizzazione del codice.

Realizzazione di classe UML con soli attributi

Per il momento, consideriamo il caso in cui la molteplicità di tutti gli attributi sia 1..1.

Per realizzare una classe UML *C* in termini di una classe Java `C`, le regole generali sono:

- La classe Java `C` è `public` e si trova in un file dal nome `C.java`.
- `C` è derivata da `Object` (no `extends`).
- La classe Java `C` avrà opportuni campi dati per gli attributi della classe UML *C*.
- La classe Java `C` avrà opportune funzioni (metodi) per gestire gli attributi e per costruire gli oggetti della classe.

Metodologia per la realizzazione: campi dati

I campi dati della classe Java `C` corrispondono agli attributi della classe UML *C*. Le regole principali sono le seguenti:

- I campi dati di `C` sono tutti `private` (o `protected`), per incrementare l'information hiding.

- Se i campi rappresentano un attributo **immutable** allora possono essere dichiarati `final` poiché non vengono più cambiati dopo la crezione dell'oggetto; altrimenti, non sono `final`.

- Si sceglie un opportuno valore iniziale per ogni attributo, o affidandosi al valore di default di Javao facendo in modo che il valore iniziale sia fissato, oggetto per oggetto, mediante un costruttore (vedi dopo).

Metodologia per la realizzazione: campi dati

- Ad un singolo attributo UML possono corrispondere uno o più campi dati (ad esempio, all'attributo *dataNascita*, possono corrispondere i tre campi `giorno`, `mese`, e `anno` della classe Java).

  I campi dati sono di un tipo base Java (`int`, `float`, ...) o `String`, ogni volta ci sia una chiara corrispondenza con il tipo dell'attributo della classe UML.

- Per due casi verranno dati maggiori dettagli in seguito:

  1. quando gli attributi UML hanno una loro molteplicità (ad es., *NumTel: int* {0..∗});

  2. quando non esiste in Java un tipo base o una classe predefinita che corrisponda chiaramente al tipo dell'attributo UML (ad es., *Valuta*).

Metodologia per la realizzazione: campi funzione

I campi funzione della classe `C` sono tutti `public`, e si classificano in:

Costruttori: devono inizializzare tutti i campi dati, esplicitamente o implicitamente.

Nel primo caso, le informazioni per l'inizializzazione vengono tipicamente acquisite tramite gli argomenti.

Funzioni `get`: per ogni campo dato (ad esempio `nome`), occorre definire una corrispondente funzione **get** (ad esempio `getNome()`), che serve a restituire al cliente, per ogni oggetto della classe, il valore dell'attributo.

Funzioni `set`: vanno previste solo per quei campi dati che possono mutare (cioè dichiarati **mutable**). Per ogni campo di questo tipo, la funzione **set** consente al cliente, per ogni oggetto della classe, di cambiare il valore dell'attributo. Ad esempio, `x.setEta(35)` fissa a 35 il valore del campo `eta` dell'oggetto `x`.

Metodologia: funzioni speciali

`equals()`: **non è opportuno** fare overriding della funzione `equals()` ereditata dalla classe `Object`.

Infatti due entità sono uguali solo se in realtà sono la stessa entità e quindi il comportamento di default della funzione `equals()` è corretto.

`clone()`: in molti casi, è ragionevole decidere di **non mettere a disposizione la possibilità di copiare un oggetto**, e non rendere disponibile la funzione `clone()` (non facendo overriding della funzione `protected` ereditata da `Object`).

Questa scelta deve essere fatta solo nel caso in cui si vuole che i moduli clienti utilizzino ogni oggetto della classe singolarmente e direttamente – *maggiori dettagli in seguito*.

`toString()`: si può prevedere di farne overriding, per avere una rappresentazione testuale dell'oggetto.

Realizzazione in Javadella classe Persona

```
// File ParteQuarta/SoloAttributi/Persona.java

public class Persona {
  private final String nome, cognome;
  private final int giorno_nascita, mese_nascita, anno_nascita;
  private boolean coniugato;
  public Persona(String n, String c, int g, int m, int a) {
    nome = n;
    cognome = c;
    giorno_nascita = g;
    mese_nascita = m;
    anno_nascita = a;
  }
  public String getNome() {
    return nome;
  }
```



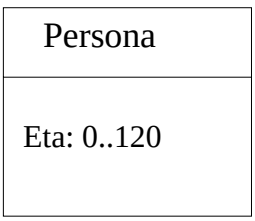
```
  public String getCognome() {
    return cognome;
  }
  public int getGiornoNascita() {
    return giorno_nascita;
  }
  public int getMeseNascita() {
    return mese_nascita;
  }
  public int getAnnoNascita() {
    return anno_nascita;
  }
  public void setConiugato(boolean c) {
    coniugato = c;
  }
  public boolean getConiugato() {
    return coniugato;
  }
```

```
  public String toString() {
    return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
        mese_nascita + "/" + anno_nascita + ", " +
        (coniugato?"coniugato":"celibe");
  }
}
```

Il problema dei valori non ammessi

In alcuni casi, il tipo base Java usato per rappresentare il tipo di un attributo ha dei valori **non ammessi** per quest'ultimo.

Ad esempio, nella classe UML *Persona* potrebbe essere presente un attributo età, con valori interi ammessi compresi fra 0 e 120.



In tali casi si pone il problema di assicurare che i valori usati nei parametri attuali del costruttore di `Persona` e della funzione `setEta()` siano coerenti con l'intervallo. Un problema simile si ha quando un'operazione di una classe o di uno use case ha **precondizioni**.

Vedremo due possibili approcci alla soluzione di questo problema.

Verifica nel lato client

Con il primo approccio è sempre **il cliente** a doversi preoccupare che siano verificate le condizioni di ammissibilità.

```
// File ParteQuarta/Precondizioni/LatoClient/Persona.java

public class Persona {
  private int eta;
  public Persona(int e) { eta = e; }
  public int getEta() { return eta; }
  public void setEta(int e) { eta = e; }
  public String toString() {
    return " (" + eta + " anni)";
  }
}

// File ParteQuarta/Precondizioni/LatoClient/Client.java
```



```
public class Client {
  public static void main(String[] args) {
    Persona giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci eta'");
      int eta = InOut.readInt();
      if (eta >= 0 && eta <= 120) { // CONTROLLO PRECONDIZIONI
        giovanni = new Persona(eta);
        ok = true;
      }
    }
    System.out.println(giovanni);
  }
}
```

Problemi dell'approccio lato client

Con tale approccio, il cliente ha bisogno di un certo grado di conoscenza dei meccanismi di funzionamento della classe, il che potrebbe causare un **aumento dell'accoppiamento**.

Inoltre, il controllo delle precondizioni verrà duplicato in ognuno dei clienti, con **indebolimento dell'estendibilità e della modularità**.

Per questo motivo, un altro approccio tipico prevede che sia la classe a doversi preoccupare della verifica delle condizioni di ammissibilità (si tratta, in altre parole, di un approccio **lato server**).

In tale approccio, le funzioni della classe lanceranno un'eccezione nel caso in cui le condizioni non siano rispettate. Il cliente intercetterà tali eccezioni, e intraprenderà le opportune azioni.

Verifica nel lato server

In questo approccio, quindi:

- Va definita un'opportuna classe (derivata da `Exception` – o `RuntimeException`, se vogliamo che l'eccezione sia "unchecked") che rappresenta le eccezioni sulle precondizioni.

- Nella classe server, le funzioni devono lanciare (mediante il costrutto `throw`) eccezioni nel caso in cui le condizioni di ammissibilità non siano verificate.

- La classe client deve intercettare mediante il costrutto `try catch` (o rilanciare) l'eccezione, e prendere gli opportuni provvedimenti.

Verifica nel lato server: esempio

```
// File ParteQuarta/Precondizioni/LatoServer/EccezionePrecondizioni.java

public class EccezionePrecondizioni extends Exception {
  public EccezionePrecondizioni(String m) {
    super(m);
  }
  public EccezionePrecondizioni() {
    super("Si e' verificata una violazione delle precondizioni");
  }
}

// File ParteQuarta/Precondizioni/LatoServer/Persona.java

public class Persona {
  private int eta;
  public Persona(int e) throws EccezionePrecondizioni {
```



```
    if (e < 0 || e > 120)  // CONTROLLO PRECONDIZIONI
      throw new
        EccezionePrecondizioni("L'eta' deve essere compresa fra 0 e 120");
    eta = e;
  }
  public int getEta() { return eta; }
  public void setEta(int e) throws EccezionePrecondizioni {
    if (e < 0 || e > 120)  // CONTROLLO PRECONDIZIONI
      throw new EccezionePrecondizioni();
    eta = e;
  }
  public String toString() {
    return " (" + eta + " anni)";
  }
}

// File ParteQuarta/Precondizioni/LatoServer/Client.java
```

```
public class Client {
  public static void main(String[] args) {
    Persona giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci eta'");
      int eta = InOut.readInt();
      try {
        giovanni = new Persona(eta);
        ok = true;
      }
      catch (EccezionePrecondizioni e) {
        System.out.println(e);
      }
    }
    System.out.println(giovanni);
  }
}
```

Classe UML con attributi e operazioni

A quanto detto per il caso di classe con soli attributi si aggiunge:

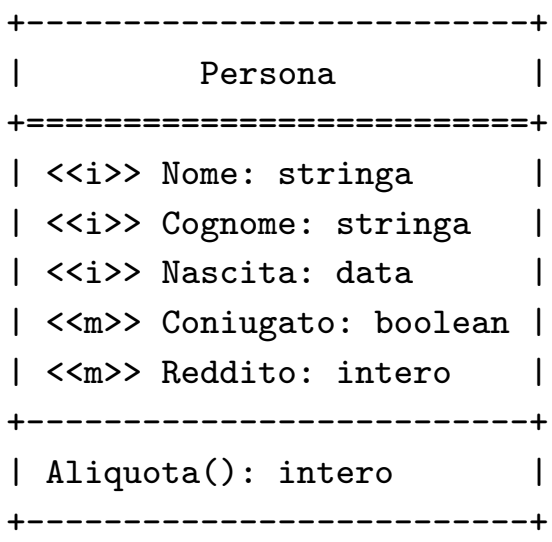
- Si analizza la specifica della classe UML *C*, che fornisce l'informazione sul significato di ogni operazione, cioè su quali sono le istruzioni che essa deve svolgere.

- Ogni operazione viene realizzata da una funzione `public` della classe Java.

  Sono possibili eventuali funzioni `private` o `protected` che dovessero servire per la realizzazione delle operazioni della classe UML *C*, ma che non vogliamo rendere disponibili ai clienti.

- Le precondizioni delle operazioni si trattano con l'approccio della verifica dal lato server, e quindi con opportune eccezioni.

Classe UML con attributi e operazioni: esempio



InizioSpecificaClasse Persona

Aliquota (): intero

pre: nessuna

post: *result* vale 0 se *this.Reddito* è inferiore a 5001, vale 20 se

this.Reddito è compreso fra 5001 e 10000, vale 30 se *this.Reddito* è compreso fra 10001 e 30000, vale 40 se *this.Reddito* è superiore a 30000.

FineSpecifica

Realizzazione in Java

```
// File ParteQuarta/AttributiEOperazioni/Persona.java

public class Persona {
  private final String nome, cognome;
  private final int giorno_nascita, mese_nascita, anno_nascita;
  private boolean coniugato;
  private int reddito;
  public Persona(String n, String c, int g, int m, int a) {
    nome = n;
    cognome = c;
    giorno_nascita = g;
    mese_nascita = m;
    anno_nascita = a;
  }
  public String getNome() {
    return nome;
```



```
  }
  public String getCognome() {
    return cognome;
  }
  public int getGiornoNascita() {
    return giorno_nascita;
  }
  public int getMeseNascita() {
    return mese_nascita;
  }
  public int getAnnoNascita() {
    return anno_nascita;
  }
  public void setConiugato(boolean c) {
    coniugato = c;
  }
  public boolean getConiugato() {
    return coniugato;
```

```
  }
  public void setReddito(int r) {
    reddito = r;
  }
  public int getReddito() {
    return reddito;
  }
  public int aliquota() {
    if (reddito < 5001)
      return 0;
    else if (reddito < 10001)
      return 20;
    else if (reddito < 30001)
      return 30;
    else return 40;
  }
  public String toString() {
    return nome + ' ' + cognome + ", " + giorno_nascita + "/" +
```

```
        mese_nascita + "/" + anno_nascita + ", " +
        (coniugato?"coniugato":"celibe") + ", aliquota fiscale: " +
        aliquota();
  }
}
```

Esempio di cliente

InizioSpecificaUseCase Analisi Statistica

QuantiConiugati (*i: Insieme(Persona)): intero*
pre: nessuna
post: *result* è il numero di coniugati nell'insieme di persone *i*

FineSpecifica

Assumiamo per il momento di rappresentare l'insieme in input semplicemente come un vettore.

```
public class AnalisiStatistica { // ...
    public static int quantiConiugati(Persona[] vett) {
        int quanti = 0;
        for (int i = 0; i < vett.length; i++)
            if (vett[i].getConiugato())
                quanti++;
        return quanti;
    }
```


Esercizio 1: classi UML con operazioni

Realizzare in Java la classe UML *Persona* che comprende anche la funzione *Età*, così specificata:

InizioSpecificaClasse Persona

Aliquota (): intero . . .

Età (*d*: data): intero
pre: nessuna
post: *result* è l'età (in mesi) della persona *this* alla data *d*.

FineSpecifica

Esercizio 2: cliente della classe

Realizzare in Java lo use case *Analisi Statistica* che comprende anche l'operazione *EtàMediaRicchi*:

InizioSpecificaUseCase Analisi Statistica

QuantiConiugati (*i: Insieme(Persona)): intero* . . .

EtàMediaRicchi (*i: Insieme(Persona), d: data): reale*
pre: *i* contiene almeno una persona
post: *result* è l'età media (in mesi) alla data *d* delle persone con aliquota massima tra le persone nell'insieme *i*

FineSpecifica

Rappresentare l'insieme in input semplicemente come un vettore.