SAPIENZA Università di Roma, Facoltà di Ingegneria
Laurea in Ingegneria Informatica

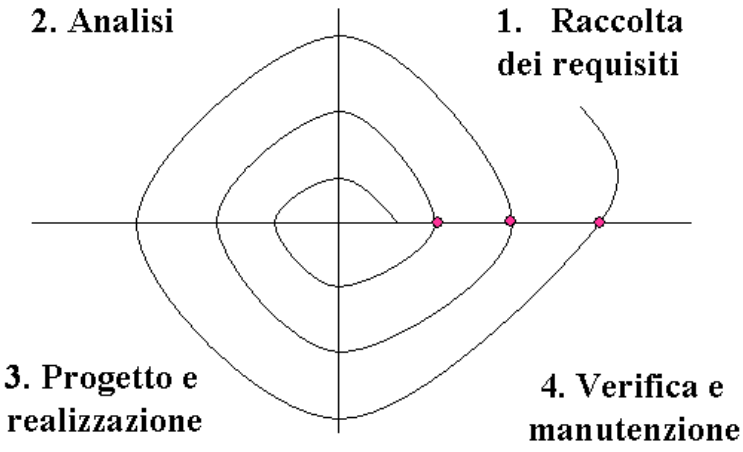
Corso di
PROGETTAZIONE DEL SOFTWARE

Prof. Giuseppe De Giacomo

LA FASE DI PROGETTO
(Diagrammi delle Classi)

Fasi del ciclo di vita del software (riassunto)

1. Raccolta dei requisiti
2. Analisi
3. Progetto e realizzazione
4. Verifica e manutenzione



Progetto e realizzazione (riassunto)

Si occupa del *come* l'applicazione dovrà realizzare le sue funzioni:

1. definire l'architettura del programma,
2. scegliere le strutture di rappresentazione,
3. produrre la documentazione,
4. scrivere il codice del programma.

Progetto: 1, 2 e 3; realizzazione: 3 e 4

Progetto: generalità

- In inglese: *design*.
- Alcune decisioni prese in questa fase dipendono dal linguaggio di programmazione scelto.
  - In questo corso, usiamo Java.
  - Le considerazioni che facciamo valgono in larga misura ($\geq$ 90%) per altri linguaggi orientati agli oggetti, ad es., C++, C#, VisualBasic.
  - È comunque possibile definire una metodologia anche per linguaggi non orientati agli oggetti, ad es., C, Pascal.

Input alla fase di progetto

È l'output della fase di analisi, ed è costituito da:

- lo **schema concettuale**, formato da:
  - diagramma delle classi e degli oggetti,;
  - diagramma delle attività;
  - diagramma degli stati e delle transizioni;
- la **specifica** (formale) delle operazioni.

Output della fase di progetto

E' l'input della fase di realizzazione, ed è costituito da:

1. scelta delle classi UML che hanno **responsabilità** sulle associazioni;
2. scelta/progetto delle **strutture di dati**;
3. scelta della corrispondenza fra **tipi** UML e Java;
4. scelta della **gestione delle proprietà** di una classe UML: immutabili, note alla nascita, ...
5. progetto della **API** delle principali classi Java,
6. . . .

Considereremo ognuno di questi elementi singolarmente.

Studio di caso

- Vedremo la fase di progetto attraverso lo studio di un caso.
- Per tale caso, considereremo già fatta la fase di analisi, per cui, oltre ai requisiti, assumeremo di avere come input lo schema concettuale.

Studio di caso: scuola elementare

Requisiti. L'applicazione da progettare riguarda le informazioni su provveditorati scolastici, scuole elementari e lavoratori scolastici. Di ogni scuola elementare interessa il nome, l'indirizzo e il provveditorato di appartenenza. Di ogni provveditorato interessa il nome e il codice attribuitogli dal Ministero. Dei lavoratori scolastici interessa la scuola elementare di cui sono dipendenti, il nome, il cognome e l'anno di vincita del concorso.

Esistono solamente tre categorie di lavoratori scolastici, che sono fra loro disgiunte: dirigenti, amministrativi e insegnanti. Dei primi interessa il tipo di laurea che hanno conseguito, dei secondi il livello (intero compreso fra 1 e 8), mentre dei terzi interessano le classi in cui insegnano, e, per ogni classe, da quale anno.

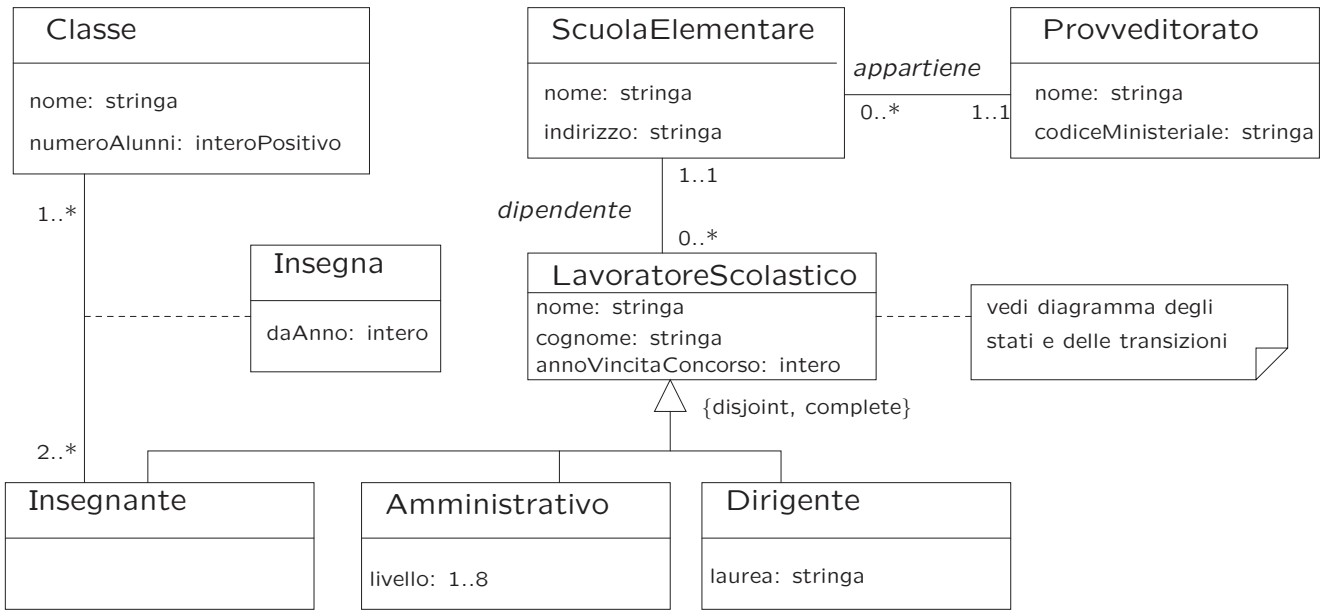
Ogni insegnante insegna in almeno una classe, e ogni classe ha almeno due insegnanti. Di ogni classe interessa il nome (ad es. "IV A") e il numero di alunni.

Studio di caso: scuola elementare

Il Ministero dell'Istruzione deve poter effettuare, come cliente della nostra applicazione, dei controlli sull'insegnamento. A questo scopo, si faccia riferimento alle seguenti operazioni:

- data una classe e l'anno corrente, calcolare quanti insegnanti della classe hanno vinto il concorso da più di 15 anni;
- dato un insegnante, calcolare il numero totale di alunni a cui insegna;
- dato un insieme di insegnanti, calcolare il numero medio di alunni a cui insegnano.

Studio di caso: diagramma delle classi UML



Progetto: responsabilità sulle associazioni

Prima di realizzare una classe UML che è coinvolta in un'associazione, ci dobbiamo chiedere se la classe ha **responsabilità** sull'associazione.

Diciamo che **una classe C ha responsabilità sull'associazione A**, quando, per ogni oggetto x che è istanza di C vogliamo poter eseguire opportune operazioni sulle istanze di A a cui x partecipa, che hanno lo scopo di:

- **conoscere** l'istanza (o le istanze) di A alle quali x partecipa,
- **aggiungere** una nuova istanza di A alla quale x partecipa,
- **cancellare** una istanza di A alla quale x partecipa,
- **aggiornare** il valore di un attributo di una istanza di A alla quale x partecipa.

Progetto: responsabilità sulle associazioni

Per ogni associazione A, **deve** esserci almeno una delle classi coinvolte che ha responsabilità su A.

I criteri per comprendere se una classe C ha responsabilità sull'associazione A sono i seguenti:

1. esiste una parte (ad es., una frase) nel documento dei requisiti, da cui si evince che per ogni oggetto x che è istanza di C vogliamo poter eseguire almeno una delle operazioni di "conoscere, aggiungere, cancellare, aggiornare";

2. esiste un'operazione per realizzare la quale è necessario che la classe C abbia tale responsabilità;

3. la responsabilità è logicamente implicata dai vincoli di molteplicità delle associazioni.

Studio di caso: responsabilità

Prendiamo in considerazione il **criterio 1**.

- *Di ogni scuola elementare interessa [. . .] il provveditorato di appartenenza.*
  → *ScuolaElementare* ha responsabilità su *appartiene*.

- *Dei lavoratori scolastici interessa la scuola elementare di cui sono dipendenti.*
  → *LavoratoreScolastico* ha responsabilità su *dipendente*.

- *[Degli insegnanti] interessano le classi in cui insegnano.*
  → *Insegnante* ha responsabilità su *insegna*.

Studio di caso: responsabilità

Prendiamo in considerazione il **criterio 2**.

- Prendiamo in considerazione l'operazione **NumeroInsegnantiDaAggiornare**. È evidente che per la sua realizzazione è necessario che, a partire da un oggetto *c* che è istanza di *Classe* possiamo conoscere le istanze di *insegna* alle quali *c* partecipa.
  → *Classe* ha responsabilità su *insegna*.

- Prendiamo in considerazione le operazioni **NumeroAlunniPerDocente** e **NumeroMedioAlunniPerDocente**. È evidente che per la loro realizzazione è necessario che, a partire da un oggetto *i* che è istanza di *Insegnante* possiamo conoscere le istanze di *insegna* alle quali *i* partecipa.
  → *Insegnante* ha responsabilità su *insegna*.

  Si noti che eravamo già a conoscenza di questa responsabilità.

Molteplicità e responsabilità

- L'esistenza di alcuni vincoli di molteplicità di associazione ha come conseguenza l'esistenza di responsabilità su tali associazioni.

- In particolare, quando una classe *C* partecipa ad un'associazione *A* con un vincolo di:

  1. molteplicità massima finita, oppure
  2. molteplicità minima diversa da zero,

  la classe *C* **ha necessariamente responsabilità su** *A*.

- Il motivo, che sarà ulteriormente chiarito nella fase di realizzazione, risiede nella necessità di poter interrogare gli oggetti della classe *C* sul numero di link di tipo *A* a cui partecipano, al fine di verificare il soddisfacimento del vincolo di molteplicità.

Studio di caso: responsabilità

Prendiamo in considerazione il **criterio 3**.

- Esiste un vincolo di molteplicità minima diversa da zero (1..*) nell'associazione *insegna*.
  $\rightarrow$ *Classe* ha responsabilità su *insegna*.

  Si noti che eravamo già a conoscenza di questa responsabilità.

- Anche tutte le altre responsabilità determinate in precedenza possono essere evinte utilizzando questo criterio.

Studio di caso: tabella delle responsabilità

Possiamo riassumere il risultato delle considerazioni precedenti nella seguente *tabella delle responsabilità*.

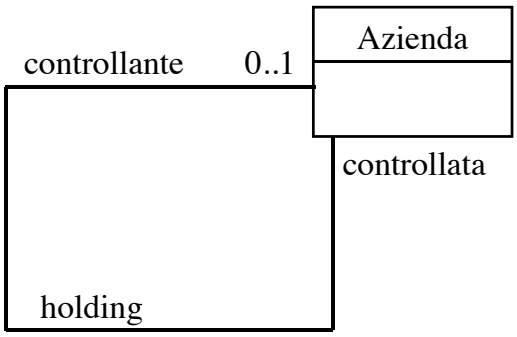
Associazione	Classe	ha resp.
insegna	*Classe*	SÌ[2,3]
	Insegnante	SÌ[1,2,3]
dipendente	*ScuolaElementare*	NO
	LavoratoreScolastico	SÌ[1,3]
appartiene	*ScuolaElementare*	SÌ[1,3]
	Provveditorato	NO

1. dai requisiti
2. dalle operazioni
3. dai vincoli di molteplicità

Criterio di **verifica**: per ogni associazione deve esserci almeno un "SÌ".

Responsabilità dei ruoli

Quanto detto vale anche per il caso in cui l'associazione coinvolga più volte la stessa classe. In questo caso il concetto di responsabilità si attribuisce **ai ruoli**, piuttosto che alle classi.



Ad esempio, la classe *Azienda* potrebbe avere la responsabilità sull'associazione *holding*, solo nel ruolo *controllata*. Questo significa che, dato un oggetto x della classe *Azienda*, vogliamo poter eseguire operazioni su x per conoscere l'eventuale azienda controllante, per aggiornare l'azienda controllante, ecc.

Scelta delle strutture di dati

- Prendendo in considerazione:
  - il diagramma delle classi,
  - la tabella delle responsabilità,
  - gli argomenti delle operazioni e i loro valori restituiti,
  - ...

  è possibile determinare se si avrà bisogno di **strutture** per la rappresentazione **dei dati** della nostra applicazione.

Scelta delle strutture di dati

- Facendo riferimento allo studio di caso, notiamo che:
  - poiché la classe *Insegnante* ha responsabilità sulla associazione *insegna*, la cui molteplicità è 1..*, per la realizzazione di quest'ultima avremo bisogno di rappresentare *insiemi di link*;
  - lo stesso si può dire prendendo in considerazione la classe *Classe*;
  - per rappresentare l'input dell'operazione *NumeroMedioAlunniPerDocente* avremo bisogno di un opportuno insieme di insegnanti.

Scelta delle strutture di dati

- In generale, emerge la necessità di rappresentare *collezioni omogenee* di oggetti.
- Per fare ciò, utilizzeremo il *collection framework* di Java 1.5 che, attraverso l'uso dei *generics* permette di realizzare collezioni omogenee, in particolare:
  - `Set<Elem>`, `HashSet<Elem>`, `...`,
  - `List<Elem>`, `LinkedList<Elem>`, `...`,
  - `...`

Esercizio 1

La funzione `Set.size()` è ridondante, in quanto può essere realizzata come **cliente** (esterna all'interfaccia `Set`) che calcola la cardinalità dell'insieme passato come argomento.

Realizzare funzioni cliente per calcolare:

- la cardinalità dell'insieme passato come argomento mediante un algoritmo **iterativo** ed uno **ricorsivo**;
- l'insieme **unione** dei due insiemi passati come argomento;
- l'insieme **intersezione** dei due insiemi passati come argomento;
- l'insieme **differenza simmetrica** dei due insiemi passati come argomento.

Corrispondenza fra tipi UML e Java

- Prendendo in considerazione:
  - il diagramma delle classi,
  - la specifica delle operazioni,
  - la scelta delle strutture di dati,

  è possibile compilare una lista dei **tipi** UML per i quali dobbiamo decidere la rappresentazione in Java.

- In generale, per la rappresentazione è opportuno scegliere un tipo base Java (`int`, `float`, ...) o una classe Java di libreria (`String`, ...) ogni volta ci sia una chiara corrispondenza con il tipo UML.

Studio di caso: tabella di corrispondenza dei tipi UML

Possiamo riassumere il risultato delle nostre scelte nella seguente *tabella di corrispondenza dei tipi* UML.

Tipo UML	**Rappresentazione in** Java
intero	`int`
interoPositivo	`int`
1..8	`int`
reale	`double`
stringa	`String`
Insieme	`Set`

Corrispondenza fra tipi UML e Java

Per due casi servono ulteriori considerazioni:

1. quando il tipo UML è necessario per un attributo di classe con una sua molteplicità (ad es., *NumeroTelefonico: Stringa* {0..∗});

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

Ulteriori considerazioni: caso 1

- In questo caso, coerentemente con la scelta precedente relativa alle strutture di dati, scegliamo di rappresentare l'attributo mediante una classe Java per *collezioni omogenee* di oggetti, come `Set`.

- Va notato che tali classi non consentono la rappresentazione di insiemi di valori di tipi base (`int`, `float`, etc.), ma **solamente di oggetti**.

- Di conseguenza, per la rappresentazione dei valori atomici dobbiamo utilizzare classi Java di libreria, quali `Integer`, `Float`, etc.
- Ad esempio, per rappresentare un ipotetico attributo *AnniVincitaConcorsi: intero* {0..∗}, useremo un oggetto costruito mediante un'espressione come:

```
HashSet<Integer> s = new HashSet<Integer>();
```

Ulteriori considerazioni: caso 2

- Per quanto riguarda la realizzazione di tipi UML tramite classi Java, notiamo che, dal punto di vista formale, il tipo UML andrebbe specificato ulteriormente tramite le *operazioni* previste per esso.

- In questo corso, affrontiamo questo aspetto in una maniera *intuitiva e non formale*, descrivendo in linguaggio naturale le operazioni.

- L'approccio seguito è simile a quello della realizzazione di *classi* UML (per maggiori dettagli rimandiamo quindi alla successiva parte del corso: "*Parte 4: La fase di realizzazione*"), con alcune alcune regole da seguire per le funzioni speciali:

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

Ulteriori considerazioni: caso 2

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

Infatti due valori sono uguali solo se sono lo stesso valore, e quindi il comportamento di default della funzione `equals()` non è corretto.

`hashCode()`: **è necessario** fare overriding della funzione `hashCode()` ereditata dalla classe `Object`.

Infatti in Java deve sempre valere il principio secondo il quale *se due oggetti sono uguali secondo* `equals()` *allora questi devono avere lo stesso codice di hash secondo* `hashCode()`. Quindi poiché ridefiniamo `equals()` dobbiamo anche ridefinire coerentemente a detto principio `hashCode()`.

`clone()`: ci sono due possibilità:

1. Nessuna funzione della classe Java effettua side-effect. In questo caso, `clone()` non si ridefinisce (gli oggetti sono *immutabili*).

2. Qualche funzione della classe Java effettua side-effect. In questo caso, poiché i moduli clienti hanno tipicamente la necessità di copiare valori (ad esempio, se sono argomenti di funzioni) **si mette a disposizione la possibilità di copiare un oggetto**, rendendo disponibile la funzione `clone()` (facendo overriding della funzione `protected` ereditata da `Object`).

Realizzazione di tipi UML

- A titolo di esempio, vediamo la realizzazione del tipo UML *Data*, inteso come aggregato di un giorno, un mese ed un anno validi secondo il calendario gregoriano, e per cui le operazioni previste sono:
  - selezione del giorno, del mese e dell'anno;
  - verifica se una data sia precedente ad un'altra;
  - avanzamento di un giorno.
- Realizziamo il tipo UML *Data* mediante la classe Java `Data`, rappresentando il giorno, il mese e l'anno come campi dati `private` di tipo `int`.
- Scegliamo di realizzare l'operazione di avanzamento di un giorno mediante una funzione Java che fa side-effect sull'oggetto di invocazione.

Esempio: la classe Java Data

```
// File Tipi/Data.java
public class Data implements Cloneable {
  // SERVE LA RIDEFINIZIONE DI clone(), in quanto una funzione fa side-effect
  public Data() {
    giorno = 1;
    mese = 1;
    anno = 2000;
  }
  public Data(int a, int me, int g) {
    giorno = g;
    mese = me;
    anno = a;
    if (!valida()) {
      giorno = 1;
      mese = 1;
      anno = 2000;
    }
  }
  public int giorno() {
    return giorno;
  }
  public int mese() {
    return mese;
  }
```

```
public int anno() {
  return anno;
}
public boolean prima(Data d) {
  return ((anno < d.anno)
          ||  (anno == d.anno && mese < d.mese)
          ||  (anno == d.anno && mese == d.mese && giorno < d.giorno));
}
public void avanzaUnGiorno() {
  // FA SIDE-EFFECT SULL'OGGETTO DI INVOCAZIONE
  if (giorno == giorniDelMese())
    if (mese == 12) {
      giorno = 1;
      mese = 1;
      anno++;
    }
    else {
      giorno = 1;
      mese++;
    }
  else
    giorno++;
}
public String toString() {
  return giorno + "/" + mese + "/" + anno;
}
```

```
public Object clone() {
  try {
    Data d = (Data)super.clone();
    return d;
  } catch (CloneNotSupportedException e) {
    // non può accadere, ma va comunque gestito
    throw new InternalError(e.toString());
  }
}
public boolean equals(Object o) {
  if (o != null && getClass().equals(o.getClass())) {
    Data d = (Data)o;
    return d.giorno == giorno && d.mese == mese && d.anno == anno;
  }
  else return false;
}
public int hashCode() {
  return giorno + mese + anno; //possiamo naturalmente realizzare una
                               //funzione di hash più sofisticata
}

// CAMPI DATI
private int giorno, mese, anno;

// FUNZIONI DI SERVIZIO
private int giorniDelMese() {
```

```
    switch (mese) {
    case 2:
      if (bisestile()) return 29;
      else return 28;
    case 4: case 6: case 9: case 11: return 30;
    default: return 31;
    }
  }
  private boolean bisestile() {
    return ((anno % 4 == 0) && (anno % 100 != 0))
      || (anno % 400 == 0);
  }
  private boolean valida() {
    return anno > 0 && anno < 3000
      && mese > 0 && mese < 13
      && giorno > 0 && giorno <= giorniDelMese();
  }
}
```

Esercizio 2

È possibile realizzare l'operazione di avanzamento di un giorno **senza fare side-effect** sull'oggetto di invocazione, ovvero nella cosiddetta maniera *funzionale*.

Ad esempio, ciò è possibile prevedendo un metodo Java pubblico con la seguente dichiarazione.

```
// File Tipi/DataFunzionale.java
public class DataFunzionale {
// ...
  public static DataFunzionale unGiornoDopo(DataFunzionale d) {
    // NON FA SIDE-EFFECT
    // ...
  }
```

Si noti che la funzione è statica e riceve l'input tramite il suo argomento, e non tramite l'oggetto di invocazione.

Realizzare la funzione Java `unGiornoDopo()` in maniera che restituisca il giorno successivo al suo argomento, senza modificare quest'ultimo.

Esercizio 3

Considerare la seguente versione modificata dei requisiti:

"Dei lavoratori scolastici interessa [...] e la residenza (indirizzo, CAP, città, provincia)."

Produrre la corrispondente versione modificata del diagramma delle classi e della tabella di corrispondenza dei tipi UML, progettando anche le eventuali classi Java che si dovessero rendere necessarie.

Gestione delle precondizioni

- Nella fase precedente è possibile che sia stato scelto un tipo o classe Java *semanticamente più esteso* del corrispondente tipo UML, ovvero che ha dei valori **non ammessi** per quest'ultimo.

  Ad esempio, abbiamo scelto il tipo Java `int` per la rappresentazione dell'attributo *livello: 1..8* della classe *Amministrativo*).

- Vedremo che una situazione simile si ha quando un'operazione ha **precondizioni**.

  Ad esempio, l'operazione *NumeroMedioAlunniPerDocente* ha la precondizione che il suo argomento non sia l'insieme vuoto.

Gestione delle precondizioni

- In tali casi si pone il problema di assicurare che i valori usati nei parametri attuali di varie funzioni Java siano coerenti con i valori ammessi per il tipo UML, ad esempio:
  - che il parametro attuale del costruttore della classe Java `Amministrativo` sia compreso fra 1 e 8),
  - che il parametro attuale della funzione Java che realizza l'operazione *NumeroMedioAlunniPerDocente* non sia l'insieme vuoto.
- 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.

Come esempio, facciamo riferimento alla classe Java `Amministrativo` per lo studio di caso (maggiori dettagli nella parte successiva del corso) e ad un suo potenziale cliente che ha la necessità di creare un oggetto.

```
// File Precondizioni/LatoClient/Amministrativo.java
public class Amministrativo {
  private int livello;
  public Amministrativo(int l) { livello = l; }
  public int getLivello() { return livello; }
  public void setLivello(int l) { livello = l; }
  public String toString() {
    return " (livello = " + livello + ")";
  }
}
```

```
// File Precondizioni/LatoClient/ClientAmministrativo.java
public class ClientAmministrativo {
  public static void main(String[] args) {
    Amministrativo giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci livello");
```

```
      int livello = InOut.readInt();
      if (livello >= 1 && livello <= 8) { // CONTROLLO PRECONDIZIONI
        giovanni = new Amministrativo(livello);
        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`) che rappresenta le eccezioni sulle precondizioni. La classe tipicamente farà overriding di `toString()`, per poter stampare un opportuno messaggio.
- 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 Precondizioni/LatoServer/EccezionePrecondizioni.java
public class EccezionePrecondizioni extends Exception {
  private String messaggio;
  public EccezionePrecondizioni(String m) {
    messaggio = m;
  }
  public EccezionePrecondizioni() {
    messaggio = "Si e' verificata una violazione delle precondizioni";
  }
  public String toString() {
    return messaggio;
  }
}

// File Precondizioni/LatoServer/Amministrativo.java
public class Amministrativo {
  private int livello;
  public Amministrativo(int l) throws EccezionePrecondizioni {
    if (l < 1 || l > 8)  // CONTROLLO PRECONDIZIONI
      throw new
        EccezionePrecondizioni("Il livello deve essere compreso fra 1 e 8");
    livello = l;
  }
  public int getLivello() { return livello; }
```

```
  public void setLivello(int l) throws EccezionePrecondizioni {
    if (l < 1 || l > 8)  // CONTROLLO PRECONDIZIONI
      throw new EccezionePrecondizioni();
    livello = l;
  }
  public String toString() {
    return " (livello = " + livello + ")";
  }
}

// File Precondizioni/LatoServer/ClientAmministrativo.java
public class ClientAmministrativo {
  public static void main(String[] args) {
    Amministrativo giovanni = null;
    boolean ok = false;
    while (!ok) {
      System.out.println("Inserisci livello");
      int livello = InOut.readInt();
      try {
        giovanni = new Amministrativo(livello);
        ok = true;
      }
      catch (EccezionePrecondizioni e) {
        System.out.println(e);
      }
    }









    System.out.println(giovanni);
  }
}
```

Verifica precondizioni

- Riassumendo, per la verifica della coerenza dei parametri attuali delle funzioni Java nel contesto di:
  - precondizioni delle operazioni,
  - tipi UML rappresentati con tipi Java che hanno valori non ammessi,

  in fase di progetto dobbiamo scegliere l'approccio lato client oppure quello lato server.

- Fatta salva la migliore qualità (come evidenziato in precedenza) dell'approccio lato server, **per pure esigenze di compattezza del codice mostrato**, nella quarta parte del corso adotteremo l'approccio lato client.

Gestione delle proprietà di classi UML

- In generale le proprietà (attributi e associazioni) di un oggetto UML evolvono in maniera arbitraria durante il suo ciclo di vita.

- Esistono però alcuni casi particolari che vanno presi in considerazione nella fase di progetto.

  Definiamo una proprietà:
  - **non nota alla nascita**, se non è completamente specificata nel momento in cui nasce l'oggetto;
  - **immutabile**, se, una volta che è stata specificata, rimane la stessa per tutto il ciclo di vita dell'oggetto.

Gestione delle proprietà di classi UML

Alcuni esempi relativi allo studio di caso:

prorietà immutabile: attributo *nome* della classe UML *LavoratoreScolastico*;

prorietà mutabile: associazione *insegna* nel ruolo di *Insegnante*;

prorietà non nota alla nascita: associazione *dipendente* nel ruolo di *ScuolaElementare*;

prorietà nota alla nascita: associazione *appartiene* nel ruolo di *ScuolaElementare*.

Assunzioni di default

- Distinguiamo innanzitutto fra:
  - proprietà **singole**, ovvero attributi (di classe o di associazione) e associazioni (o meglio, ruoli) con molteplicità 1..1;
  - proprietà **multiple**, tutte le altre.
- Le nostre assunzioni di default, ovvero che valgono in assenza di ulteriori elementi, sono le seguenti:
  - tutte le proprietà sono **mutabili**;
  - le proprietà singole sono **note alla nascita**;
  - le proprietà multiple **non sono note alla nascita**.

Tabelle di gestione delle proprietà di classi UML

Ovviamente, in presenza di motivi validi, possiamo operare scelte diverse da quelle di default.

Riassumeremo tutte le nostre scelte **differenti da quelle di default** mediante la *tabella delle proprietà immutabili* e la *tabella delle assunzioni sulla nascita*.

Mostriamo le tabelle per lo studio di caso.

Classe UML	**Proprietà immutabile**
Provveditorato	*nome*
LavoratoreScolastico	*nome*
	cognome
	annoVincitaConcorso
Dirigente	*laurea*
ScuolaElementare	*appartiene*

Classe UML	**Proprietà**	
	nota alla nascita	**non nota alla nascita**
LavoratoreScolastico	–	*dipendente*

Sequenza di nascita degli oggetti

Sono necessari alcuni commenti sulla seconda tabella appena mostrata.

- In generale, non possiamo dire nulla sull'*ordine* in cui gli oggetti nascono. Ad esempio, facendo riferimento allo studio di caso, non sappiamo se nasceranno prima gli oggetti di classe *Insegnante* o quelli di classe *Classe*.

- L'assunzione che quando nasce un oggetto Java corrispondente ad una scuola elementare sia noto il suo provveditorato di appartenenza è ragionevole poiché le responsabilità su *appartiene* è *singola*, e la molteplicità è 1..1. Questa assunzione implica che nascano prima i provveditorati delle scuole elementari.

- Viceversa, quando nasce un oggetto Java corrispondente ad un lavoratore scolastico non assumiamo che sia nota la scuola elementare di cui è dipendente.

Valori alla nascita

Per tutte le proprietà che sono note alla nascita potremmo chiederci se per esse esiste un valore di default (valido per tutti gli oggetti) oppure no.

Ad esempio:

- l'attributo *nome* della classe UML *LavoratoreScolastico* è noto alla nascita dell'oggetto, ed è in generale diverso per oggetti distinti;
- nell'ipotesi di aggiungere alla classe *LavoratoreScolastico* l'attributo intero *noteDiDemerito*, potremmo assumere che sia noto alla nascita, che il valore iniziale sia 0 per tutti gli oggetti, e che sia mutabile.

Queste informazioni potrebbero essere rappresentate mediante un'opportuna tabella.

API delle classi Java progettate

- Prendendo in considerazione:
  - il diagramma delle classi,
  - la tabella di corrispondenza fra tipi UML e rappresentazione in Java,
  - la tabella delle proprietà immutabili,
  - la tabella delle assunzioni sulla nascita,

  è possibile dare l'interfaccia pubblica (API) di molte delle classi Java che realizzeremo nella fase successiva.

API delle classi Java progettate

In particolare, possiamo dare le API per le classi Java corrispondenti alle classi UML. Per tali classi dobbiamo prevedere le seguenti categorie di funzioni Java:

- **costruttori**;
- funzioni per la **gestione degli attributi**;
- funzioni per la **gestione delle associazioni**;
- funzioni corrispondenti alle **operazioni di classe**;
- funzioni per la **gestione degli eventi** e altro;
- funzioni **di servizio** (ad es., per la stampa).

API delle classi Java progettate

Ad esempio, possiamo definire la API della classe Java `ScuolaElementare` nel seguente modo (per maggiori dettagli e per le classi corrispondenti alle associazioni rimandiamo alla parte successiva del corso, in quanto servono ulteriori considerazioni).

```
public class ScuolaElementare {
// COSTRUTTORI
    /** specificare, se opportuno, il significato degli argomenti,
        ad esempio in formato utile per javadoc */
    public ScuolaElementare
        (String nome, String indirizzo, Provveditorato appartiene) { };
// GESTIONE ATTRIBUTI
    public String getNome(){return null;};
    public String getIndirizzo(){return null;};
    public void setIndirizzo(String i){};
    public void setNome(String n){};
// GESTIONE ASSOCIAZIONI
    // - appartiene
    public Provveditorato getProvveditorato(){return null;};
    // - dipendente: non ha responsabilità
// OPERAZIONI DI CLASSE
    // assenti
// GESTIONE EVENTI
    // assenti
// STAMPA
    public String toString(){return null;};
}
```

Struttura dei file e dei package

- Prendendo in considerazione le regole di visibilità di Java e le esigenze di *information hiding*, possiamo definire la cosiddetta *struttura dei file, dei direttori e dei package*, che costituisce un'ulteriore aspetto progettuale relativo all'*architettura dei moduli software* della nostra applicazione.

- In generale, sceglieremo di disporre tutti i file dell'applicazione in un package Java `P`, nel direttorio `P`.

- Saranno presenti opportuni sottodirettori e sottopackage (maggiori dettagli nella parte successiva del corso).

UML e fase di progetto

- A titolo di completezza, si fa notare che in alcuni testi viene proposto UML come linguaggio grafico **anche per la fase di progetto**. Ad esempio:
  - è possibile rappresentare le responsabilità sulle associazioni con opportuni arricchimenti del diagramma delle classi;
  - è possibile rappresentare la struttura dei file e dei package mediante il cosiddetto *diagramma dei package*;
  - ...

- Per semplicità, in questo corso ci limitiamo ad usare UML per la sola fase di analisi. Con l'eccezione di quando illustreremo pattern realizzativi, dove useremo diagrammi delle classi UML per illustrare in modo grafico le API necessarie per la realizzazione.

Soluzioni degli esercizi della terza parte

Soluzione esercizio 1

```
// File Insieme/UtilSet.java
import java.util.*;
public class UtilSet {
    public static <T> int cardIter(Set<T> ins) {
        int card = 0;
        Iterator<T> it = ins.iterator();
        while(it.hasNext()) {
            card++;
            T elem = it.next();
        }
        return card;
    }
    public static <T> int cardRic(Set<T> ins) {
        if (ins.isEmpty())
            return 0;
        else {
            Iterator<T> it = ins.iterator();
            T elem = it.next();
            ins.remove(elem);
            int temp = cardRic(ins);
            ins.add(elem);
            return temp + 1;
        }
    }
}
```

Soluzione esercizio 2

```
// File Tipi/DataFunzionale.java
public class DataFunzionale {
  // NON SERVE LA RIDEFINIZIONE DI clone(), in quanto nessuna funzione
  // fa side-effect
  public DataFunzionale() {
    giorno = 1;
    mese = 1;
    anno = 2000;
  }
  public DataFunzionale(int a, int me, int g) {
    giorno = g;
    mese = me;
    anno = a;
    if (!valida()) {
      giorno = 1;
      mese = 1;
      anno = 2000;
    }
  }
  public int giorno() {
    return giorno;
  }
  public int mese() {
    return mese;









  }
  public int anno() {
    return anno;
  }
  public boolean prima(DataFunzionale d) {
    return ((anno < d.anno)
            ||  (anno == d.anno && mese < d.mese)
            ||  (anno == d.anno && mese == d.mese && giorno < d.giorno));
  }
  public static DataFunzionale unGiornoDopo(DataFunzionale d) {
    // NON FA SIDE-EFFECT
      DataFunzionale res = new DataFunzionale(d.anno,d.mese,d.giorno);
      if (d.giorno == d.giorniDelMese())
  if (d.mese == 12) {
      res.giorno = 1;
      res.mese = 1;
      res.anno++;
  }
  else {
      res.giorno = 1;
      res.mese++;
  }
      else
  res.giorno++;
      return res;
  }
```

```
  public String toString() {
    return giorno + "/" + mese + "/" + anno;
  }
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      DataFunzionale d = (DataFunzionale)o;
      return d.giorno == giorno && d.mese == mese && d.anno == anno;
    }
    else return false;
  }
  public int hashCode() {
    return giorno + mese + anno; //possiamo naturalmente realizzare una
                                 //funzione di hash più sofisticata
  }

  // CAMPI DATI
  private int giorno, mese, anno;

  // FUNZIONI DI SERVIZIO
  private int giorniDelMese() {
    switch (mese) {
    case 2:
      if (bisestile()) return 29;
      else return 28;
    case 4: case 6: case 9: case 11: return 30;
    default: return 31;









    }
  }
  private boolean bisestile() {
    return ((anno % 4 == 0) && (anno % 100 != 0))
      || (anno % 400 == 0);
  }
  private boolean valida() {
    return anno > 0 && anno < 3000
      && mese > 0 && mese < 13
      && giorno > 0 && giorno <= giorniDelMese();
  }
}
```

Soluzione esercizio 3: diagramma delle classi

L'unico aspetto che viene modificato è la classe UML *LavoratoreScolastico*, che ha ora un ulteriore attributo.

Soluzione esercizio 3: tabella di corrispondenza dei tipi UML

Si tiene conto del tipo del nuovo attributo mediante una ulteriore riga nela tabella di corrispondenza dei tipi UML.

Tipo UML	**Rappresentazione in** Java
intero	`int`
interoPositivo	`int`
1..8	`int`
reale	`double`
stringa	`String`
Insieme	`HashSet`
Indirizzo	`Indirizzo`

Soluzione esercizio 3: scelte realizzative

- Intendiamo il tipo UML *Indirizzo* come aggregato di via, cap, città e provincia. Le uniche operazioni che prevediamo sono quelle di selezione dei componenti.
- Realizziamo il tipo UML *Indirizzo* mediante la classe Java `Indirizzo`, rappresentando i componenti come campi dati `private` di tipo `String`.
- Nessuna funzione Java farà side-effect sull'oggetto di invocazione, quindi **non** prevediamo la realizzazione di `clone()`.
- Come sempre nella realizzazione di tipi, prevediamo la ridefinizione di `equals()` e `hashCode()`.

Soluzione esercizio 5: classe Java Indirizzo

```
// File Tipi/Indirizzo.java
public class Indirizzo {
    // CAMPI DATI
    private String via, cap, citta, provincia;
    // NON SERVE LA RIDEFINIZIONE DI clone(), in quanto nessuna funzione
    // fa side-effect
    public Indirizzo(String vi, String ca, String ci, String pr) {
        via = vi;
        cap = ca;
        citta = ci;
        provincia = pr;
    }
    public String via() {
        return via;
    }
    public String cap() {
        return cap;
    }
    public String citta() {
        return citta;
    }
    public String provincia() {
```

```
        return provincia;
    }
  public String toString() {
    return via + "/" + cap + "/" + citta + "/" + provincia;
  }
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      Indirizzo r = (Indirizzo)o;
      return r.via.equals(via) && r.cap.equals(cap) &&
          r.citta.equals(citta) && r.provincia.equals(provincia);
    }
    else return false;
  }
  public int hashCode() { //funzioni di hash più sofisticate sono possibili
    return via.hashCode() + cap.hashCode() +
      citta.hashCode() + provincia.hashCode();
  }

}
```