Ancora sulla progettazione/Pacchetti

Concetti di coesione/accoppiamento/coerenza

Uso dei package

Scelta delle classi

- Abbiamo già visto che per scrivere una buona applicazione usando un linguaggio ad oggetti come Java è bene fare un'adeguata progettazione iniziale
- Il punto focale su cui concentrarsi sono le classi
- Nella programmazione funzionale classica ci si concentra sulle funzioni: sul flusso che il codice dovrebbe seguire
- Nella programmazione ad oggetti invece l'accento è sulle entità, cioè gli oggetti appartenenti alle varie classi individuate
- I metodi, cioè la parte funzionale, devono essere pensati come associati alle entità

Scelta delle classi

- Uno degli aspetti fondamentali che sono indice di una buona progettazione è il seguente:
- **Ogni classe dovrebbe rappresentare un singolo concetto**

Scelta delle classi

- Abbiamo visto alcune classi che rappresentano concetti matematici o elementi della vita di tutti i giorni:
  - Rectangle
  - BankAccount
  - Purse
- Le proprietà degli oggetti di queste classi (variabili istanza) sono facili da capire, così come le operazioni che si possono eseguire su di essi (i metodi)

Scelta delle classi

- In generale i concetti che appartengono all'ambito dell'applicazione e che vengono identificati da sostantivi specifici sono ottimi candidati per essere classi
- Un'altra utile categoria di classi può essere descritta come quella degli **attori**:
- Gli oggetti di queste classi svolgono una serie di compiti. Esempi:
  - StringTokenizer
  - RandomNumberGenerator
  - GestoreNuoviConti

Scelta delle classi

- Abbiamo anche visto che in casi limitati è utile definire delle classi (o solo metodi, o solo variabili istanza) statiche quando vogliamo rappresentare qualcosa che si riferisce a tutti gli oggetti di una data classe (costanti pubbliche, variabili statiche) o a funzionalità correlate (metodi statici come ad esempio tutti quelli raggruppati nella classe **`Math`**) di utilità.
- Infine abbiamo visto le classi di Test che hanno come scopo quello di contenere un metodo main per testare funzionalità di classi definite precedentemente ed indipendentemente.

Scelta delle classi

- Quale potrebbe essere una classe poco valida?
- In generale sono sintomi di errori di progettazione:
  - Se dal nome di una classe non si capisce cosa dovrebbero fare gli oggetti della classe stessa
  - Se il nome di una classe non rappresenta un gruppo di entità, ma una specifica funzione
- Es: classi come **`CalcolaBustaPaga`** oppure **`PogrammaPerIlPagamento`**

Coesione e accoppiamento

- Vediamo due criteri utili per analizzare la qualità di una interfaccia pubblica di una classe:
  - Coesione
  - Accoppiamento

Coesione

- Un classe dovrebbe rappresentare, abbiamo detto, un singolo concetto
- I metodi e le costanti pubbliche che sono elencati nell'interfaccia dovrebbero avere una buona **coesione**, cioè tutte le caratteristiche dell'interfaccia dovrebbero essere strettamente correlate al singolo concetto rappresentato dalla classe
- Se così non è forse è meglio usare **classi separate**

Coesione

- Consideriamo ad esempio l'interfaccia della classe **`Purse`**:

```
public class Purse {
   public Purse() {...}
   public void addNickels(int count) {...}
   public void addDimes(int count) {...}
   public void addQuarters(int count) {...}
   public double getTotal() {...}
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE =
                                                0.25;
}
```

Coesione

- Se ci pensiamo bene in realtà sono persenti due concetti diversi in questa classe:
  1. Borsellino che calcola il valore totale delle monetine che contiente
  2. Valore delle singole monetine
- Potrebbe avere più senso definire una classe separata **`Coin`** i cui oggetti rappresentano singole monete ognuna con il proprio valore.
- La classe **`Purse`** dovrebbe allora cambiare interfaccia e permettere di inserire nel borsellino oggetti della classe **`Coin`**

Coesione

```
public class Coin {
   public Coin (double aValue, String
                           aName) {...}
   public double getValue() {...}
   ...
}
public class Purse {
   public void add(Coin aCoin) {...}
   public double getTotal() {...}
...
}
```

Coesione

- È evidente che questa è una soluzione migliore dal punto di vista della progettazione
- Abbiamo usato la prima soluzione negli esempi precedenti solo per avere un esempio semplice

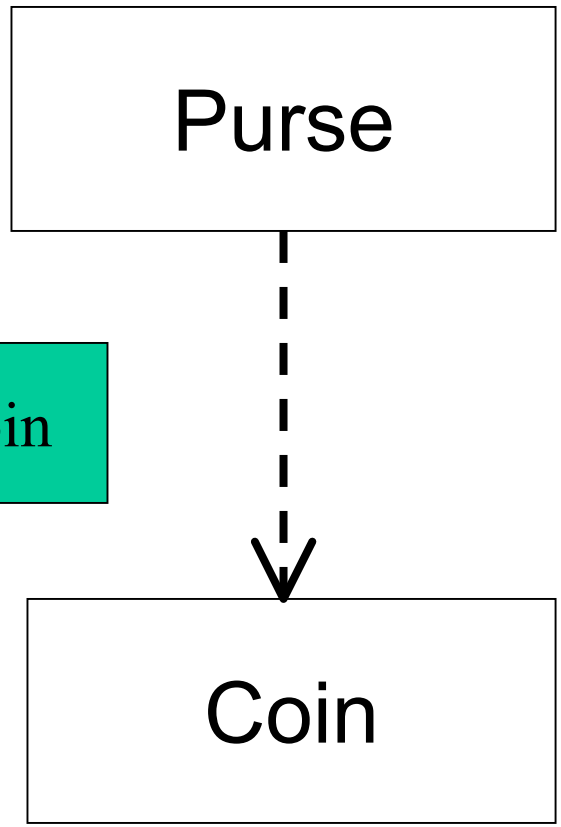
Dipendenza fra classi

- Molte classi hanno bisogno di altre classi per svolgere il loro compito
- Per esempio la classe **`Purse`** appena vista **dipende** dalla classe **`Coin`** per determinare il valore totale delle monete
- In generale una classe A dipende da una classe B se A usa istanze della classe B

Dipendenza fra classi

- UML -“Unified Modeling Language”- è un linguaggio grafico standardizzato per l’analisi e la progettazione orientata agli oggetti
- UML rappresenta, nei diagrammi di classi, la dipendenza tra classi con una linea tratteggiata che termina con una freccia aperta da una certa classe A a un’altra classe B dove A dipende da B

Dipendenza fra classi



Purse dipende da Coin

È la stessa notazione grafica che usa Bluej

Accoppiamento

- Se in un'applicazione molte classi dipendono una dall'altra diciamo che **c'è un elevato accoppiamento tra le classi**
- Perché l'accoppiamento è importante?
- Se la classe **Coin** viene modificata in una versione successiva del programma allora **tutte** le classi che dipendono da lei possono richiedere una modifica!
- Se la modifica è drastica tutte le classi accoppiate devono essere aggiornate

Accoppiamento

- Inoltre, se vogliamo usare una classe A in un altro programma, siamo costretti ad usare anche tutte le classi da cui A dipende
- Quindi, in generale, è bene **ridurre al minimo l'accoppiamento** tra le classi della propria applicazione
- Ovviamente ci sono alcuni casi in cui l'accoppiamento è necessario e non si può eliminare!

Coerenza

- La coesione e l'accoppiamento sono buoni criteri da seguire per analizzare una progettazione
- In aggiunta un altro criterio è quello di guardare la coerenza nella definizione dei metodi per quanto riguarda i nomi e i parametri
- La presenza di schemi coerenti è sempre segno di buona fattura

Coerenza

- Brutti esempi di incoerenza si trovano anche nelle librerie standard di Java!
- Ad esempio abbiamo visto che per far aprire una finestra di dialogo per prendere un input basta chiamare

```
JOptionPane.showInputDialog(
    promptString);
```

- Tuttavia per far apparire una finestra per visualizzare solo un messaggio siamo costretti ad usare

```
JOptionPane.showMessageDialog(null,
        messageString);
```

Coerenza

- A cosa serve **`null`**?
- Se guardiamo le API vediamo che il metodo ha bisogno di un parametro che gli indichi la finestra di appartenenza oppure **`null`** se non ha una finestra di appartenenza
- Perché questa incoerenza? Non c'è nessun motivo...
- Bastava fornire due metodi **`showMessageDialog`** di cui uno con un solo parametro stringa, come è stato fatto per **`showInputDialog`**

Coerenza

- Le incoerenze non sono errori gravissimi, ma perché non evitarle soprattutto quando si può farlo facilmente?

Metodi accessori/modificatori

- Abbiamo detto che è sempre meglio incapsulare tutto lo stato degli oggetti ed eventualmente, poi, mettere a disposizione dei metodi get/set per accedere a certe variabili istanza in maniera controllata

Effetti collaterali

- In generale un metodo qualsiasi (non statico) di una classe può cambiare il valore di una variabile istanza qualsiasi dell'oggetto su cui è chiamato, ma anche su altri oggetti della stessa classe!
- Abbiamo visto che il meccanismo per la chiamata dei metodi congela le attivazioni precedenti a quelle del metodo in esecuzione
- Pertanto la visibilità dello stato da parte di un metodo è limitata

Effetti collaterali

- In particolare **non** sono visibili le variabili di frame dichiarate in tutte le attivazioni precedenti
- Come variabili di frame sono visibili solo il parametro implicito **`this`** e i parametri del metodo
- I parametri che **non** sono di tipo riferimento ad oggetto sono variabili locali al metodo: i valori che contengono vengono passati dall'ambiente chiamante, ma eventuali loro modifiche non si riflettono all'esterno (passaggio dei parametri per valore)

Effetti collaterali

- Tuttavia l'accesso allo heap, e quindi agli oggetti, non è ristretto
- Se un oggetto
  - riceve come parametri variabili riferimento ad altri oggetti

  oppure
  - ha nel suo stato riferimenti ad altri oggetti

  può tranquillamente accedere ai loro campi (rispettando comunque i vincoli espressi da **`private`**) e chiamare su di loro metodi

Effetti collaterali: esempio

```
public class BankAccount {
  /** Trasferisce denaro da questo conto a un
    altro conto
    @param amount la somma da trasferire
    @param other il conto su cui trasferire
  public void transfer(double amount,
                       BankAccount other) {
    balance = balance - amount;
    other.balance = other.balance + amount;
  }
...
}
```

Effetti collaterali: esempio

- **`other`** è un parametro di tipo riferimento ad oggetti della classe **BankAccount**
- **`balance`** è una variabile privata della classe **`BankAccount`** e **`transfer`** è un metodo della classe **`BankAccount`**: quindi, per le regole di visibilità, **`transfer`** può accedere al **`balance`** dell'oggetto puntato da **`other`**

Effetti collaterali: esempio

- L'esecuzione del metodo **`transfer`** fa avvenire una modifica al di fuori dello stato dell'oggetto su cui il metodo è stato chiamato
- In particolare viene modificato lo stato di un altro oggetto della stessa classe (ma in generale può essere modificato lo stato anche di oggetti di altre classi)
- In questi casi si dice che il metodo in questione ha **effetti collaterali**

Effetti collaterali

- Anche operazioni di visualizzazione di messaggi sullo standard output all'interno di un metodo vengono considerate un effetto collaterale
- E se la vostra classe un giorno fosse usata su un hardware che non è dotato di un dispositivo di ouput simile a una console?
- Se succede qualcosa di "sbagliato" dentro un metodo la cosa migliore da fare è segnalarlo al chiamante con una **eccezione** o con la restituzione di un valore in uscita particolare (es –1 del metodo **`read()`** delle classi **`FileInputStream`** o **`FileReader`**)

Effetti collaterali

- La conclusione è:
- **È buona norma evitare di scrivere metodi con effetti collaterali**

Pacchetti

- Abbiamo spesso importato nei nostri programmi classi della libreria standard
- Abbiamo visto, nelle API, che queste sono raggruppate in pacchetti (**`package`**)
- Anche noi possiamo definire uno o più pacchetti personali che contengono le nostre classi!

Nomi di pacchetti

- Un nome di pacchetto, abbiamo visto, è una serie di stringhe separate da punti:
  - **`java.lang`**
  - **`javax.swing`**
  - `...`
- Per indicare una classe che si trova all'interno di un pacchetto va aggiunto al nome del pacchetto un punto e il nome della classe:
  - java.lang.String
  - javax.swing.JOptionPane

Nomi di pacchetti

- Se vogliamo creare un nostro pacchetto è bene seguire, per scegliere il nome, la procedura seguente:
- Indicare come prefisso del nome del pacchetto il nome **rovesciato** del dominio internet della propria azienda/organizzazione/università
- Questo perché esiste un organismo mondiale che controlla che non ci siano conflitti fra i nomi dei domini

Nomi di pacchetti

- Di riflesso avremo che non ci saranno mai conflitti fra i nomi delle nostre classi e quelle scritte da altri programmatori in tutto il mondo
- Ad esempio un nome per il pacchetto che contiente tutti gli esempi che abbiamo visto in questo corso potrebbe essere:

```
it.unicam.informatica.LabDiProgr20042005
```

Nomi di pacchetti

- Se volete pubblicare vostre classi personali potete usare anche il vostro indirizzo email:
- Ad esempio pippo@hotmail.com può usare come prefisso dei suoi pacchetti

```
com.hotmail.pippo
```

Creare un pacchetto

- Per creare un pacchetto dobbiamo innanzitutto raggruppare i sorgenti delle classi in una cartella e inserire all'inizio di ogni file sorgente la riga:

`package` *nomePacchetto*`;`

- Poi bisogna inserire questa cartella all'interno di una gerarchia di sottocartelle che riflettono il nome del pacchetto

Creare un pacchetto

- Nel nostro caso, ad esempio, dovremmo mettere le nostre classi all'interno di una cartella di nome

  `LabDiProgr20042005`

  inserita in una cartella di nome

  `informatica`

  a sua volta inserita in

  `unicam`

  a sua volta inserita in

  `it`

Creare un pacchetto

- Dopodiché compilare tutte le classi dal livello più alto:

```
#My Documents> javac
  it\unicam\informatica\LabDiProgr20
  042005\*.java
```

Usare le classi di un pacchetto

- Se volessimo a questo punto usare la classe **`BankAccount`** definita nel nostro pacchetto dovremmo importarla come

```
import
  it.unicam.informatica.LabDiProgr20
  042005.BankAccount;
```

- All'interno del file sorgente possiamo evitare di scrivere tutto il nome ed usare solo **`BankAccount`**

Usare le classi di un pacchetto

- Ovviamente possiamo anche importarle tutte:

```
import
  it.unicam.informatica.LabDiProgr2004200
  5.*;
```

- Se ci dovessero essere conflitti (ad esempio se abbiamo definito una classe **`String`** anche nel nostro pacchetto) possiamo risolverli specificando tutto il nome (unico) della classe nel pacchetto:

`java.lang.String` per quella della distribuzione java e

`it.unicam.informatica.LabDiProgr20042005.String` per la nostra

Usare le classi di un pacchetto

- Infine, per permettere ad un programma di usare le nostre classi, dobbiamo porre il percorso con cui arrivare alla cartella **`it`** all'interno della variabile di ambiente CLASSPATH oppure inserire tale percorso nella chiamata delle compilazioni/esecuzioni tramite l'opzione – cp:

#>**`javac -cp`** *percorsoPerIt nomeClasse.java*

#>**`java -cp`** *percorsoPerIt nomeClasse*