Esercizio: la classe CashRegister

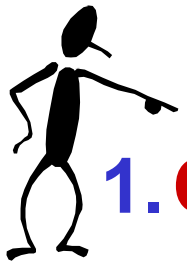
Si veda anche il libro di testo:

Consigli pratici 3.1	(pag. 98)
Sezione 4.1	(pag. 140)
Sezione 8.2	(pag. 395)

Problema

- Si chiede di **realizzare una classe** **CashRegister** che simuli il comportamento di un registratore di cassa.
- Per semplicità si considera un registratore di cassa molto semplice. Il compito assegnato specifica quali aspetti di un registratore di cassa devono essere simulati dalla classe.
- Elenco, in linguaggio naturale, delle operazioni realizzabili da un registratore di cassa semplificato:
  - **Registra** il prezzo di vendita per un articolo venduto.
  - **Consenti** l'inserimento della somma di denaro pagata.
  - **Calcola** il resto dovuto al cliente.

Riassunto: progettare una classe



1. **Capire** cosa deve fare un oggetto della classe
   - Elenco in linguaggio naturale delle operazioni possibili
2. **Specificare** l'interfaccia pubblica
   - Ovvero, definire i metodi tramite le loro intestazioni
3. **Documentare** l'interfaccia pubblica
4. **Identificare** i **campi di esemplare** a partire dalle intestazioni dei metodi
5. **Realizzare** costruttori e metodi
   - Se avete problemi a realizzare un metodo forse dovete riesaminare i passi precedenti
6. **Collaudare** la classe con un programma di collaudo

La classe CashRegister

- **Fase 1**: capire cosa **deve fare** la classe
  - Registrare prezzo di vendita per un articolo venduto
  - Consentire inserimento della somma pagata
  - Calcolare resto dovuto al cliente

- **Fase 2**: specificare **l'interfaccia pubblica**
  - Trasformare l'elenco precedente in un insieme di **metodi**
  - Specificando parametri e valori restituiti
  - E aggiungendo i costruttori della classe

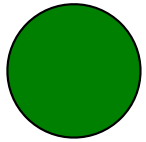
- **Fase 3**: scrivere la **documentazione**
  - Commenti di documentazione per metodi e classe
  - In questo esercizio saltiamo questa fase…

L'interfaccia pubblica di CashRegister

```
public class CashRegister
{
   public CashRegister()
   {
   }

   public void recordPurchase(double amount)
   {
   }

   public void enterPayment(double amount)
   {
   }
   public double giveChange()
   {
   }
}
```



Commenti di documentazione

Commentare l'interfaccia pubblica

- Java ha **delimitatori speciali** per commenti di documentazione

```
/**
  Registra la vendita di un articolo
  @param amount il prezzo dell'articolo
*/
public void recordPurchase(double amount)
{
    //corpo del metodo
}
```

- **`@param nomeparam`** descrive un parametro esplicito
- **`@return`** descrive il valore restituito dal metodo

```
/**
  Calcola il resto e azzera la
  macchina per la vendita successiva
  @return resto dovuto al cliente
*/
public double giveChange()
{
    //corpo del metodo
}
```

Commentare l'interfaccia pubblica

- Solitamente si inseriscono brevi commenti anche alla classe, per illustrarne lo scopo

```
/**
   Un registratore di cassa somma gli articoli venduti
   e calcola il resto dovuto al cliente
*/
public class CashRegister
{
   ...
}
```

- Usando commenti di documentazione in questo formato si può generare in **maniera automatica** documentazione in **html**

```
javadoc NomeClasse.java
```

  - Genera un documento **NomeClasse.html** ben formattato e con collegamenti ipertestuali, contenente i commenti a **NomeClasse**

La classe CashRegister

- **Fase 4**: identificare i **campi di esemplare**
  - Sono le informazioni che un oggetto deve conservare per svolgere il proprio compito
  - Vanno identificati a partire dalle definizioni dei metodi
  - In questo caso ci servono
    - la spesa totale
    - il pagamento effettuato
- **Fase 5**: realizzare costruttori e metodi

La classe CashRegister (v. 1)

```
public class CashRegister
{
   public CashRegister()
   {
      purchase = 0;
      payment = 0;
   }

   public void recordPurchase(double amount)
   {
      double newTotal = purchase + amount;
      purchase = newTotal;
   }
                                                    //(continua)
```

La classe CashRegister (v. 1)

```
                                                    //(continua)
public void enterPayment(double amount)
   {
     payment = amount;
   }

   public double giveChange()
   {
     double change = payment - purchase;
     payment = 0;
     purchase = 0;
     return change;
   }

   private double purchase;
   private double payment;

}
```

La classe CashRegisterTester (v. 1)

- **Fase 6:** scrivere una classe di collaudo

- Scriviamo una classe CashRegisterTester_v1 che simuli il lavoro di un cassiere
  - Il cassiere può registrare un numero a piacere di articoli acquistati dal cliente, ciascuno con il proprio prezzo
  - Al termine dell'inserimento, il cassiere registra in cassa la somma versata dal cliente per il pagamento
  - Infine, il registratore di cassa calcola il resto dovuto al cliente e lo stampa a standard output

La classe CashRegisterTester (v. 1)

```
import java.util.Scanner;
import java.util.Locale;

public class CashRegisterTester
{
  public static void main(String[] args)
  {
    CashRegister register = new CashRegister();
    Scanner in = new Scanner(System.in);
    in.useLocale(Locale.US);
    boolean done = false;
    while(!done)
      {
   System.out.println("Prezzo articolo (Q per uscire):");
   String input = in.next();
                                                    // (continua)
```

La classe CashRegisterTester (v. 1)

```
                                                        //(continua)
   if (input.equalsIgnoreCase("Q"))
     done = true;
   else
     {
       double amount = Double.parseDouble(input);
       register.recordPurchase(amount);
     }
      }
   System.out.println("Inserire pagamento");
   double amount = in.nextDouble();
   register.enterPayment(amount);
   double change = register.giveChange();
   System.out.printf(Locale.US,
                          "Resto dovuto: %.2f%n", change);
  }
}
```

Migliorare la classe CashRegister

- Vogliamo scrivere un metodo **enterPayment** più comodo da usare
  - I valori di ingresso sono **conteggi di monete**
  - Il metodo calcola la somma pagata sommando le quantità di monete versate, moltiplicate per i rispettivi valori

```
payment = dollars + quarters * 0.25 +
   dimes * 0.1 + nickels * 0.05 + pennies * 0.01;
```

- Non usiamo **numeri magici**
  - Definiamo delle **costanti** (QUARTER_VALUE, DIME_VALUE, …)
  - Saranno costanti **di classe** (**static**)

La classe CashRegister (v. 2)

```
public class CashRegister
{
   ...
   public void enterPayment(int dollars, int quarters,
                            int dimes, int nickels, int pennies)
   {
      payment = dollars + quarters * QUARTER_VALUE +
                dimes * DIME_VALUE + nickels * NICKEL_VALUE +
                pennies * PENNY_VALUE;

   }
   ...
   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   public static final double PENNY_VALUE = 0.01;
   ...
}
```

La classe CashRegisterTester (v. 2)

```
import java.util.Scanner;
import java.util.Locale;

public class CashRegisterTester
{
  public static void main(String[] args)
  {
    CashRegister register = new CashRegister();
    Scanner in = new Scanner(System.in);
    in.useLocale(Locale.US);
    boolean done = false;
    while(!done)
      {
   System.out.println("Prezzo articolo (Q per uscire):");
   String input = in.next();
       if (input.equalsIgnoreCase("Q"))
     done = true;
   else
     {
                                                  // (continua)
```

La classe CashRegisterTester (v. 2)

```
                                                    //(continua)
            double amount = Double.parseDouble(input);
          register.recordPurchase(amount);
        }
         }
      System.out.println("Inserire pagamento");
      System.out.println("Dollars:");
      int dollars = in.nextInt();
      System.out.println("Quarters:");
      int quarters = in.nextInt();
      System.out.println("Dimes:");
      int dimes = in.nextInt();
      System.out.println("Nickels:");
      int nickels = in.nextInt();
      System.out.println("Pennies:");
      int pennies = in.nextInt();
      register.enterPayment(dollars, quarters, dimes,
                            nickels, pennies);
      double change = register.giveChange();
      System.out.printf(Locale.US, "Resto dovuto: %.2f%n", change);
   }
}
```

Coesione e accoppiamento

- Una classe dovrebbe rappresentare un singolo concetto
  - Metodi pubblici e costanti elencati nell'interfaccia pubblica devono avere una buona **coesione**
  - Ovvero devono essere strettamente correlate al singolo concetto rappresentato dalla classe
- La nostra classe "registratore di cassa" ha **scarsa coesione**

```
public class CashRegister
{
  public void enterPayment(int dollars, int quarters, int dimes,
                           int nickels, int pennies)
    . . .
   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 e accoppiamento

- Una progettazione più attenta ci porta a definire **due classi**
  - Una per descrivere il concetto "**registratore di cassa**"
  - Una per descrivere il concetto "**moneta/valuta**"

```
public class Coin
{
   public Coin(double aValue, String aName){ . . . }
   public double getValue(){ . . . }
   . . .
}

public class CashRegister
{
   public void enterPayment(int coinCount,
         Coin coinType) { . . . }
   . . .
}
```

La classe Coin

```
public class Coin
{
  public Coin(double value, String name)
  {
    this.value = value;
    this.name = name;
  }

  public double getValue()
  { return value; }

  public String getName()
  { return name; }

  private double value;
  private String name;
}
```

La classe CashRegister (v. 3)

```
public class CashRegister
{
   ...
    public void enterPayment(int coinCount, Coin coinType)
   {
     double amount = coinCount * coinType.getValue();
     payment += amount;
   }

   ...
}
```

La classe CashRegisterTester (v. 3)

```
import java.util.Scanner;
import java.util.Locale;

public class CashRegisterTester
{
  public static void main(String[] args)
  { CashRegister register = new CashRegister();
    Coin euro = new Coin(1, "eur");
    Coin euroCent = new Coin(0.01, "euCent");

    Scanner in = new Scanner(System.in);
    in.useLocale(Locale.US);
    boolean done = false;
    while(!done)
      { System.out.println("Prezzo articolo (Q per uscire):");
    String input = in.next();
       if (input.equalsIgnoreCase("Q"))
      done = true;
    else
      { double amount = Double.parseDouble(input);
        register.recordPurchase(amount);
      }
      }                          // (continua)
```

La classe CashRegisterTester (v. 3)

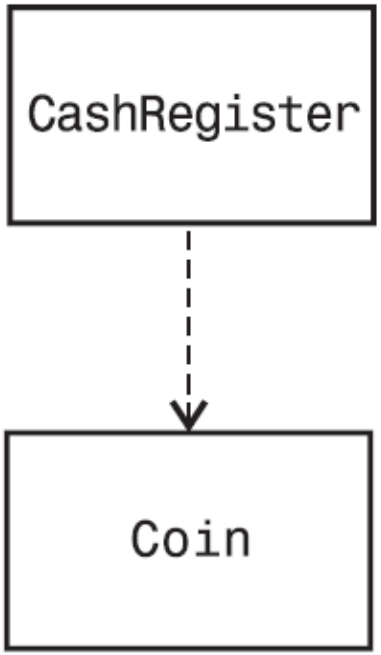
```
    done = false;                                                    //(continua)
    while (!done)
      {  System.out.println("Inserire pagamento");
      System.out.println("Quantita' taglio (Q per uscire):");
      String input = in.next();
      if (input.equalsIgnoreCase("Q"))
        done = true;
      else
        {  int coinCount = Integer.parseInt(input);
        String coinName = in.next();
        if (coinName.equals(euro.getName()))
          register.enterPayment(coinCount, euro);
        else if (coinName.equals(euroCent.getName()))
          register.enterPayment(coinCount, euroCent);
        else
          System.out.println("Moneta sconosciuta");
        }
      }
    double change = register.giveChange();
    System.out.printf(Locale.US, "Resto dovuto: %.2f%n", change);
  }
}
```

Progettare classi: accoppiamento, coesione, effetti collaterali (sezioni 8.2, 8.4)

Coesione e accoppiamento

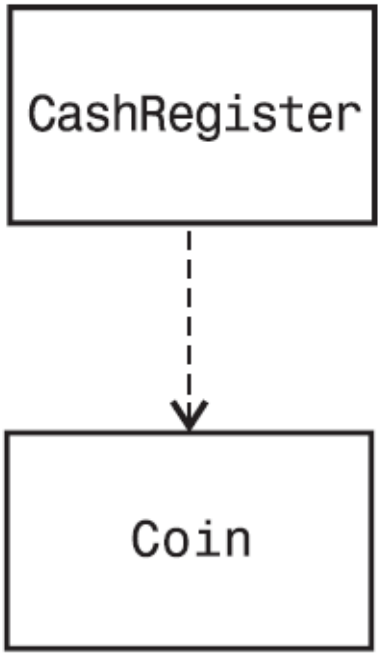
- Una classe dovrebbe rappresentare un singolo concetto
  - Metodi pubblici e costanti elencati nell'interfaccia pubblica devono avere una buona **coesione**
  - Ovvero devono essere strettamente correlate al singolo concetto rappresentato dalla classe

- Quando una classe usa un'altra classe all'interno del proprio codice si parla di **accoppiamento** tra le due classi
  - In particolare, la prima classe **dipende** dalla seconda classe perché non può essere usata senza di essa



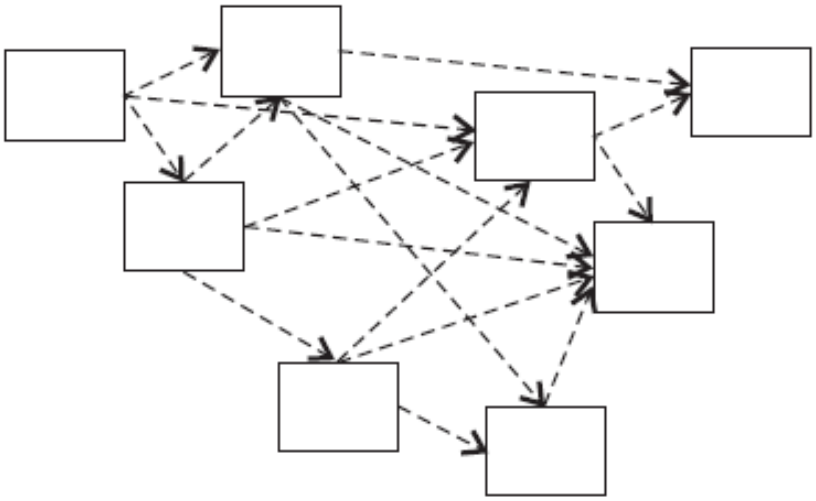
Relazione di dipendenza tra classi

- Visualizziamo accoppiamento con diagrammi ***UML*** *(Unified Modeling Language)*
  - formalismo per rappresentare il rapporto fra le classi
  - ogni classe viene rappresentata con un **rettangolo**.
  - una **linea tratteggiata** indica la **dipendenza** di una classe da un'altra classe
  - Sulla linea c'è una **freccia** rivolta verso la classe nei confronto della quale esiste la **dipendenza**

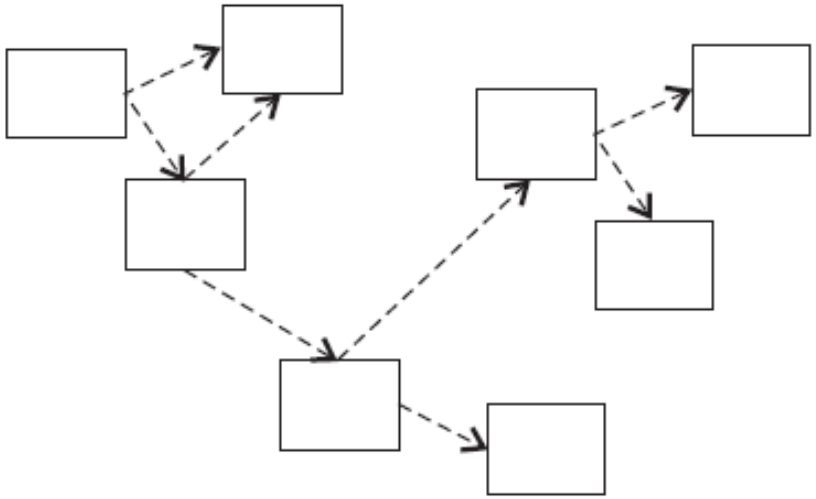


Accoppiamento

- **Problema**: se una classe viene modificata, tutte quelle che da essa dipendono possono richiedere **modifiche**
- **Problema**: per usare una classe in un programma, vanno **usate** anche tutte quelle da cui essa dipende
  - Bisogna eliminare tutti gli accoppiamenti non necessari



Elevato accoppiamento



Basso accoppiamento

Effetti collaterali

- Effetto collaterale di un metodo:
  - Qualsiasi tipo di comportamento **osservabile** al di fuori del metodo stesso
- Qualsiasi metodo modificatore ha effetti collaterali
  - Modificano il proprio **parametro implicito**
- Esistono altri tipi di effetto collaterale:
  - Modifica di un **parametro esplicito**
  - Esempio: un metodo **transfer** per **BankAccount**

```
public void transfer(double amount, BankAccount other)
{
   balance = balance - amount;
   other.balance = other.balance + amount;
      // modifica il parametro esplicito
}
```

Effetti collaterali

- Un altro tipo di effetto collaterale: visualizzazione di **dati in uscita**
  - Esempio: un metodo **printPurchase** per **CashRegister**

```
public void printPurchase() // Non consigliato
{
   System.out.println("The purchase is now $" + purchase);
}
```

- **Problema**: abbiamo ipotizzato che qualsiasi utente parli la lingua inglese
- **Problema**: abbiamo realizzato accoppiamento con le classi **System** e **PrintStream**
  - Evitiamo di inserire attività di input/output all'interno di metodi di una classe che ha altre finalità

È tutto chiaro? …

1. Se a è un riferimento a un oggetto **BankAccount**, allora **a.deposit(100)** modifica l'oggetto. È un effetto collaterale?
2. Immaginiamo di avere scritto il seguente metodo in una classe **Dataset**: produce effetti collaterali?

```
void read(Scanner in)
{  while (in.hasNextDouble())
     add(in.nextDouble());
}
```