Esempi notevoli di problemi ed errori

Forse si impara di più a vedere gli errori che le cose giuste…
Quindi ho scelto sette progetti che presentano problemi notevoli, e li ho anonimizzati (gli autori li riconosceranno: ma non si deprimano per questo, cerchino piuttosto di cogliere in positivo l'indicazione di cosa devono sistemare).
Credo che guardarli, dopo aver letto i commenti che trovate nel seguito, possa essere di aiuto a chi deve ancora fare l'esame.

Progetto BAD01:

Il progetto non gira perché ha un problema con gli array.

1) Usare array quando non è necessario è un errore. Imparate a usare le collections invece! Ad esempio, per restituire i primi cinque elementi dopo il sort viene fatta una conversione di una lista in Array:

   ```
   stampa((Alloggio[])alloggi.subList(0, 5).toArray());
   ```

   Ma la toArray restituisce un Object[], e non si può fare un cast di un array su un altro! Quindi questa riga dà errore, e non se ne esce.
   Tutto questo usando le Collections sarebbe facile e immediato: sarebbe stato sufficiente scrivere il metodo stampa in modo che lavorasse su una lista invece che su un array…
2) Reimplementa il sort. Perchè mai? Le API di collection lo fanno probabilmente meglio di quanto lo scrivereste voi. Dovete imparare a riusare il codice esistente, è uno dei vantaggi di OOP. Solo quando ci sono esigenze particolari si riscrive, altrimenti si riusa!

Progetto BAD02:

Il progetto gira, e sembra funzionare. Ma davvero fa quel che deve nel modo giusto? No, siamo davvero molto lontani da una soluzione accettabile…

1) Nella classe di avvio, le viste ordinate sono preparate a mano (!) invece che calcolate!
2) Per gestire i click che mostrano le immagini si mettono dei bottoni. Ma chi ha mai detto che gli unici gestori di eventi sono i bottoni? Basta far ascoltare i click del mouse (con tipo di evento giusto e listener giusto) alla componente che mostra il nome dell'alloggio: non serve che sia un bottone!
3) Invece che usare ereditarietà viene creata una variabile che mantiene il tipo di alloggio: sbagliatissimo!

```
public class Alloggio {
   Alloggio(String t, String n, int p, float v, String d) {
      t = tipoalloggio;
      n = nome;
      p = prezzo;
      v = valutazione;
      d = descrizione;
   }
   String tipoalloggio;
   String nome;
   int prezzo;
   float valutazione;
   String descrizione;
}
```

   A parte poi che nel costruttore l'ordine delle variabili è invertito: n = nome; invece di nome = n;
4) Sostanzialmente è senza struttura: ci sono 4 classi, ma ListaAlloggi e Textfield non sono usate (la seconda poi è vuota…), di Alloggio abbiamo già detto, il resto è tutto in una sola classe!

Progetto BAD03:

Il progetto non è completo. La ragione per cui lo includiamo in questa "rassegna" è per la una curiosa creazione di classi: definisce la classe Stella, che in realtà non crea una Stella, ma solo un contenitore, e poi le classi da Stella1 a Stella5, ciascuna delle quali aggiunge delle stelle al contenitore.

```
public class Stella extends HBox{
   Rectangle sfondo = new Rectangle( 60, 30);
   Stella() {
      this.getChildren().add(sfondo);
   }
}
…
public class Stella3 extends Stella {
   Stella3() {
      Circle c1 = new Circle(10);
      Circle c2 = new Circle(10);
      Circle c3 = new Circle(10);
      this.getChildren().add(c1);
      this.getChildren().add(c2);
       this.getChildren().add(c3);
   }
}
…
```

Non è un buon uso del concetto di classe… Sarebbe stato assai meglio, e molto più semplice, scrivere ad esempio un'unica classe di questo tipo:

```
public class Stelle {
  Rectangle sfondo = new Rectangle( 60, 30);
   Stelle(int n) {
      this.getChildren().add(sfondo);
      for (int k=0; k<n; k++) {
         Circle c = new Circle(10);
         this.getChildren().add(c);
     }
  }
}
```

Progetto BAD04:

Un altro esempio di progetto assolutamente insufficiente anche se “gira”, pur non essendo completo.
Non c’è, nel progetto, alcun oggetto di business! Quando leggiamo un testo, dobbiamo vedere quali entità si prestano a diventare classi, quali sono le relazioni tra di esse, se sussiste dell’ereditarietà… Niente di tutto ciò è stato fatto in questo caso.
Abbiamo una classe “Campo” che viene istanziata con sei valori a seconda di uno switch su un indice.

```
public class Campo {

   int p;
   double v;
   Text nomeCampo, prezzoCampo, valutazioneCampo;
   HBox extraCampo;

   public Campo(int index) {
    switch (index) {
          case 1:
             nomeCampo.setText("Alpenhof");
             p = 60;
             v = 8.5;
             prezzoCampo.setText("" + p);
             valutazioneCampo.setText("" + v);
             extraCampo.getChildren().add(new Text("Mezza pensione"));
             break;
  …
```

Ma se la nostra applicazione dovesse diventare Booking.com, dovremmo prendere questa classe e mettere nella sua pancia milioni di alloggi, invece di istanziarli! Non ha proprio senso.
A leggere il codice, di cosa si occupa la classe Campo? Non lo sappiamo finché non ne esaminiamo il contenuto. In un progetto ben fatto, avremmo le classi Alloggio, Albergo, Appartamento, Pensione, che servono a istanziare business objects, ovvero le cose di cui parliamo e che avrebbero relazioni di ereditarità. Per inciso, l’uso di ereditarietà ove possibili era anche esplicitato come requisito progettuale.

Progetto BAD05:

Questo progetto fa tutto quel che è stato richiesto, ma mostra una grave confusione tra il concetto di classe e quello di oggetto.
Viene definita una classe astratat Alloggio, ma poi questa, invece che essere sottoclassata **per tipologia**, viene sottoclassata per ciascuna istanza.
Quindi ad esempio troviamo la seguente classe:

```
public class Alpenhof extends Alloggio{
   Alpenhof(){
      nome="Alpenhof";
      prezzo=60;
      valutazione=8.5;
      extra="MEZZA_PENSIONE";
      location="img/"+nome+".jpg";
   }
}
```

Ma una classe è uno stampino per produrre oggetti dello stesso tipo! Quanti Hotel Alpenhof vorremo creare? Avremmo dovuto fare una classe “Hotel”, e poi istanziarla nei vari casi (uno per Alpenhof, uno per Majestic ecc, e poi analogamente una classe per “Pensione” con le sue istanze ecc.
Questa confusione tra classe e istanza è stata considerata gravissima.

Altri suggerimenti

1) L'ereditarietà dovrebbe in genere accompagnarsi con il polimorfismo.
   Ad esempio, nella classe Alloggio prevediamo un comportamento definito come addExtra che agisce su un parametro che, in modo assai generale, prevediamo essere un Parent.

```
public abstract class Alloggio {
   String nomeAlloggio;
   int prezzoAlloggio;
   double valutazioneAlloggio;
   …
   abstract void addExtra(Parent p)
}
```

Le sottoclassi implementeranno il comportamento, ciascuna a modo suo. Ad esempio:

```
public class Albergo extends Alloggio {
  Stelle stelle;
   …
  void addExtra(Parent p) {
        p.getChildren().add(stelle);
  }
}

public class Appartament extends Alloggio {
  int  maxPersone;
   …
  void addExtra(Parent p) {
        TextField tf=new TextField("Max persone: "+maxPersone);
        p.getChildren().add(tf);
}
```

In questo modo, non servirà nel main preoccuparsi di quale particolare sottoclasse di alloggio abbiamo:

```
…
Alloggio a1=new Albergo("nomeAlb",100,8.5,3)
Alloggio a2=new Appartamento(("nomeApp",50,8.2,3)
HBox h1=new HBox();
HBox h2=new HBox();
a1.addExtra(h1);
a2.addExtra(h2);
```

Non servono if, switch, instanceof, tipo o altro!

2) E’ sbagliato usare variabili statiche come canale di comunicazione tra classi. Static sono SOLO le costanti, e le variabili e i metodi DI CLASSE.
3) Il nome del package deve essere tale da non creare potenziali conflitti nemmeno in futuro, quindi dovrebbe essere *univoco al mondo!*
4) Le consegna vanno rispettate: se ad esempio si dice che una finestra deve avere una dimensione data, così deve essere.
5) A far bene, le classi dovrebbero sempre avere la equals e la hashcode (ve le genera gratis Netbeans…)
6) Anche la documentazione (javadoc) può essere generata automaticamente da Netbeans (almeno come scheletro). @param e @ return, generati automaticamente, vanno però completati con la semantica – ovvero va aggiunta qualche parola che spieghi il ruolo della variabile o del valore di ritorno.
   Es.: il commento
   @param width
   generato automaticamente va completato in
   @param width Larghezza della finestra