UNIVERSITÀ DEGLI STUDI
DI MILANO

Ingegneria del software

Progettazione

Un programma in C

http://mdnahas.github.io/xmassong/
https://youtu.be/YpuNU3y1KAk

```
main(t,_,a ) char* a;{return!0<t?t<3?main(-79,-13,a+main(-87,
1-_,main(-86, 0,a+1 )+a)):1,t<_?main( t+1, _, a ):3,main(-94,
-27+t, a )&&t == 2 ?_<13 ? main ( 2, _+1,"%s %d %d\n" ):9:16:
t<0?t<-72?main( _, t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+\
,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'\
3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]\
'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){nl]!/n{n#'; r{#w'r\
 nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' iwk{KK{nl]!/w{\
%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;{nl'-{}rw]'/+,\
}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/"):t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main
((*a == '/')+t, _,a+1):0<t?main ( 2, 2 , "%s"):*a=='/'||main
(0,main(-61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m\
 .vpbks,fxntdCeghiry"),a+1);}
```

Cosa fa questo codice?

Altra versione

```
1  static String[] days = {"first", "second", ..., "twelfth"};
2  static String[] gifts = { "a partridge in a pear tree", "two turtle doves", ... };
3
4  static String firstLine(int day) {
5    return "On the " + days[day] +
6           " day of Christmas my true love gave to me:\n";
7  }
8
9  static String allGifts(int day) {
10   if( day == 0 ) { return "and " + gifts[0]; }
11   else { return gifts[day] + "\n" + allGifts(day-1); }
12 }
13
14 public static void main(String[] args) {
15   System.out.println(firstLine(0));
16   System.out.println(gifts[0]);
17   for (int day = 1; day < 12; day++) {
18     System.out.println(firstLine(day));
19     System.out.println(allGifts(day));
20   }
21 }
```

Refactoring

Improving the design of code without changing its functionality

1. Migliorare un design tenuto inizialmente "semplice" (o nato male)
2. Preparare il design per una funzionalità che non si integra bene in quello esistente
3. Eliminare debolezze (debito tecnico)

Design Knowledge

Dove la salviamo?

0. memoria
1. documenti di design (linguaggio naturale o diagrammi)
2. all'interno di piattaforme di discussione, issue management, version control
3. con modelli specializzati (UML). Può portare a approcci *generative programming*
4. nel codice, ma spesso difficile rappresentare le *ragioni*

Come condividiamo

Non un design specifico, ma il know how

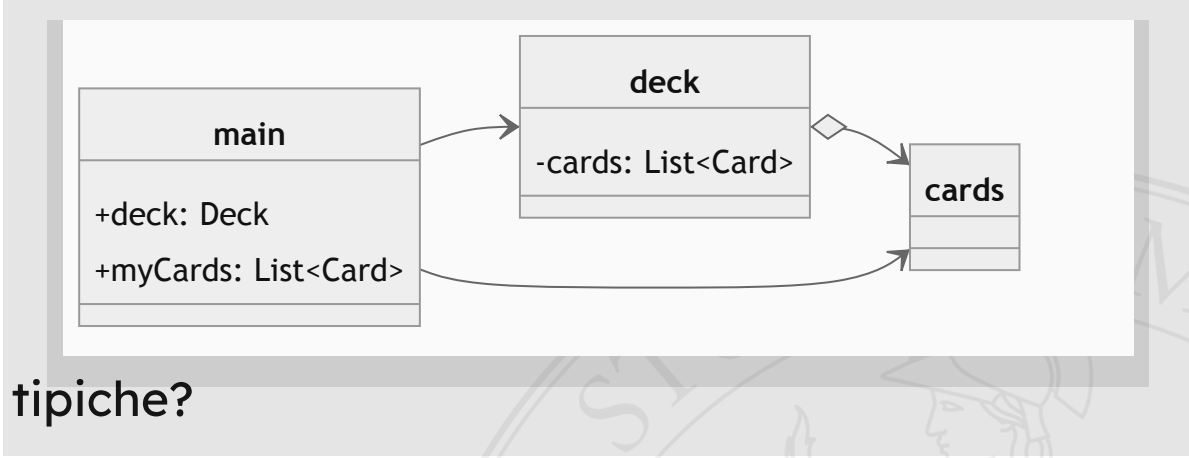
0. metodi
1. design pattern
2. principi

check conoscenza di alcuni termini e concetti

- Object Orientation
  - Ereditarietà
  - Polimorfismo
  - Collegamento Dinamico
- Principi SOLID
  - Single Responsability
  - Open Close Principle
  - Liskov Substitution Principle
  - Interface Segregation
  - Dependency Inversion

Reference escaping

- Cosa è? Violazione encapsulation Errore che non faceva passare Prog2 con me



- Situazioni tipiche?
  1. getter ritorna un riferimento a un segreto
  2. setter assegna al segreto un qualche riferimento che gli viene passato
  3. costruttore assegna al segreto un qualche riferimento che gli viene passato

Encapsulation and information hiding

Parnas [L8]
Solo ciò che è nascosto può essere cambiato liberamente e senza pericoli

I due scopi fondamentali sono:

- facilitare la comprensione del codice
  - vengono definite le responsabilità
- rendere più facile modificarne una parte senza fare danni

Immutabilità

- Cosa è una classe immutabil?
- Non c'è modo di cambiare lo stato dell'oggetto dopo la sua inizializzazione
  - non fornisce metodi che modificano lo stato
  - ha tutti attributi privati (non obbligatorio)
  - ha tutti gli attributi final (non obbligatorio dichiararlo)
  - assicura accesso esclusivo a tutte le parti non immutabili

Code smell

- Codice duplicato
- Metodo troppo lungo
- Troppi livelli di indentazione
- Troppi attributi/responsabilità per classe
- Lunghe sequenze di *if-else* o *switch*
- Classe troppo grande
- Lista di parametri troppo lunga
- Numeri *magici*
- Commenti
- Nomi oscuri o inconsistenti
- Codice morto
- *getter* e *setter*

Un mazzo di 52 carte

- Vogliamo *ideare* una **struttura dati** capace di rappresentare un mazzo o una mano (sequenza) di carte tratte dal mazzo
- Il mazzo è quello con:
  - 4 semi (cuori, quadri, fiori e picche)
  - 13 valori (asso, due, …, regina, re)

Prima idea

- Mappiamo le nostre 52 carte su 52 interi: [0-51] (0-12 Cuori, 13-25 Quadri…)
- Usiamo il contenitore di interi per la sequenza di carte

```
1  int[] deck = {15,10,1,8}; // mano con quattro carte
2
3  int card = 15;          // 15 = ??
4  int suit = card / 13; // 1 => Quadri
5  int rank = card % 13; // 2 => Tre (si comincia da 0)
```

Problemi:

- scomodo dover fare operazioni per capire seme e valore
- non inibisco operazioni non lecite (per il compilatore è un intero)

Encapsulation e astrazioni

- Diamo un nome ai concetti (type abstraction)

```java
class Deck {
  private Card[] cards;
}

class Card {
  private int value; // 0-51
}
```

```java
class Card {
  private int[] card = {1, 3};
  // 1 = Quadri, 3 = Tre (cominciando da 1)
}
```

Rappresentazione interna

```
1  enum Suit { CLUBS, DIAMONDS, SPADES, HEARTS };
2  enum Rank { ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
3              NINE, TEN, JACK, QUEEN, KING };
4
5  class Card {
6    private Suit suit;
7    private Rank rank;
8  }
9
10 class Deck {
11   private List<Card> cards = new ArrayList<>();
12 }
```

Getter e setter ?

```java
class Card {
  private Suit suit;
  private Rank rank;


  public Rank getRank() { return rank; }
  public Suit getSuit() { return suit; }
  public void setRank(Rank r) { rank = r; }
  public void setSuit(Suit s) { suit = s; }
}

class Deck {
  List<Card> cards = new ArrayList<>();
}
```

```java
// Volendo potevo mantenere int per
// struttura interna
class Card {
  private int suit;
  public Suit getSuit() {
    return switch (suit) {
      case 0 -> Suit.CLUBS;
      case 1 -> Suit.DIAMONDS;
      case 2 -> Suit.SPADES;
      case 3 -> Suit.HEARTS;
      default -> null;
    };
  }

  public void setSuit(Suit s) {
    suit = s.ordinal();
  }
}
```

Problemi

```
1  class Main {
2    public static void main(String[] args) {
3      Deck deck = new Deck();
4      Card card = new Card();
5
6      card.setSuit(Suit.DIAMONDS);
7      card.setRank(Rank.THREE);
8      deck.getCards().add(card);
9      deck.getCards().add( new Card() );
10
11     System.out.println("There are " + deck.getCards().size() + " cards:");
12
13     for (Card currentCard : deck.getCards())
14       System.out.println(currentCard.getRank() + " of " + currentCard.getSuit());
15   }
16 }
```

https://martinfowler.com/bliki/TellDontAsk.html

Tell-Don’t-Ask Principle

Non chiedere i dati, ma dì cosa vuoi che faccia sui dati

- Cercare di minimizzare i getter studiando cosa ci facciamo con il valore ritornato e definendo funzioni opportune

```
class Card {
  private Suit suit;
  private Rank rank;

  public Card(Suit s, Rank r) {
    suit = s;
    rank = r;
  }
  @Override
  public String toString() {
    return rank + " of " + suit;
  }
}
```

```
class Deck {
  private ArrayList<Card> cards = new ArrayList<>();
  @Override
  public String toString() {
    String ans = "There are " + cards.size() + " cards:\n";
    for (Card currentCard : cards)
      ans += currentCard.toString() + '\n';
    return ans;
  }
  public void add(Card card) {
    cards.add(card);
  }
}
```

Proviamo a fare insieme il laboratorio 03

```java
public class ForthInterpreter implements Interpreter {
  private final Deque<Integer> stack = new ArrayDeque<>();

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder("");
    for (Integer s : stack) sb.insert(0, s.toString() + ' ');
    sb.append("<- Top");
    return sb.toString();
  }

  @Override
  public void input(String program) {
    stack.clear();
    Scanner sc = new Scanner(program);
    while (sc.hasNext()) {
```

Necessario fare refactoring?

Quali sono gli indizi che spingono a ristrutturare il codice?

Uno dei principali è quando il cliente continua a chiederti cambiamenti di una parte. A questo punto pensate se sia opportuno separare quella parte dalla classe in modo da rendere possibile modificarla senza modificare la classe base (OCP).
Ad es. se vediamo che continua ad aggiungere comandi e l'impressione è che possano esserci diverse versioni con diversi set di comandi allora può avere senso trovare un design meno statico dell'`if`/`switch`

```
String item = sc.next();
if (item.equals("+"))
  stack.push(stack.pop() + stack.pop());
else if (item.equals("-"))
  stack.push(-stack.pop() + stack.pop());
else try {
    stack.push(Integer.valueOf(item));
  } catch (NumberFormatException e) {
    throw new IllegalArgumentException(
          "Token error '" + item + "'");
  }
```

```
private final Map<String, Runnable> op = new HashMap<>();

public ForthInterpreter() {
 op.put("+", () -> stack.push(stack.pop() + stack.pop()));
 op.put("-", () -> stack.push(-stack.pop() + stack.pop()));
}
[...]
String item = sc.next();
if (op.containsKey(item))
  op.get(item).run();
else try {
    stack.push(Integer.valueOf(item));
[...]
```