Miglioramenti Sul Codice

1. Separazione delle varie operazioni, ciascuna in un suo metodo. Ciò permette di **verificare la correttezza delle operazioni e di riusarle**
2. **Costruzione di astrazioni**, via via più utili, ovvero metodi di livello più basso e metodi di livello più alto, che richiamano i precedenti, lo stesso si fa per le classi
3. **Separazione delle operazioni** che aggiornano lo stato (command) da quelle che restituiscono valori (query), in metodi diversi
   - **Query** sono operazioni che ottengono informazioni (accedendo a dati o facendo calcoli sui dati)
   - **Command** sono operazioni che effettuano cambiamenti
4. Lo stato dell'oggetto diventa **osservabile**, attraverso metodi opportuni, in modo da poter fare i test
   - **Test: programma che esegue il codice sotto certe condizioni (valori in input) e valuta se il risultato fornito è corretto**


```java
public class TestPagamenti { // per classe Pagamenti vers 0.9
   private Pagamenti pgm = new Pagamenti();

   private void initLista() {
      pgm.svuota();
      pgm.inserisci("321.01");
      pgm.inserisci("531.7");
      pgm.inserisci("1234.5");
   }

   public void testSommaValori() {
      initLista();
      pgm.calcolaSomma();
      if (pgm.getSomma() == 2087.21)
         System.out.println("OK test somma val");
      else System.out.println("FAILED test somma val");
   }

   public void testMaxValore() {
      initLista();
      pgm.calcolaMassimo();
      if (Math.abs(pgm.getMassimo() - 1234.5) < 0.01)
         System.out.println("OK test massimo val");
      else System.out.println("FAILED test massimo val");
   }
   // continua ...
```



```java
   // continua
   public void testLeggiFile() {
      try {
         pgm.leggiFile("csv", "Importi.csv");
         System.out.println("OK test leggi file");
      } catch (IOException e) {
         System.out.println("FAILED test leggi file");
      }
   }

   public static void main(String[] args) {
      TestPagamenti tl = new TestPagamenti();
      tl.testLeggiFile();
      tl.testSommaValori();
      tl.testMaxValore();
   }
}
```

Output dell'esecuzione di TestPagamenti quando il file Importi.csv è presente nella cartella csv (dentro la cartella con gli eseguibili)

```
OK test leggi file
OK test somma val
OK test massimo val
```

Esecuzione quando il file non viene trovato

```
FAILED test leggi file
OK test somma val
OK test massimo val
```



```java
public class Pagamenti { // Pagamenti vers 1.2 (senza command and query)
   private List<Float> importi = new ArrayList<>();
   public void leggiFile(String c, String n) throws IOException {
      LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      String riga;
      while (true) {
         riga = f.readLine();
         if (riga == null) break;
         inserisci(Float.parseFloat(riga));
      }
      f.close();
   }
   public void inserisci(float x) {
      if (!importi.contains(x)) importi.add(x);
   }
   public float calcolaSomma() {
      float risultato = 0;
      for (float v : importi)  risultato += v;
      return risultato;
   }
   public float calcolaMassimo() {
      float risultato = 0;
      for (float v : importi)  if (risultato < v)  risultato = v;
      return risultato;
   }
   public void svuota() {
      importi = new ArrayList<>();   }
}
```

Pagamenti
− importi: List<Float>
+ leggiFile(c: String, n: String) + inserisci(x: float) + calcolaSomma(): float + calcolaMassimo(): float + svuota()

```java
public class TestPagamenti { // per classe Pagamenti vers 1.2
   private Pagamenti pgm = new Pagamenti();
   private void initLista() {
      pgm.svuota();
      pgm.inserisci(321.01);
      pgm.inserisci(531.7);
      pgm.inserisci(1234.5);
   }
   public void testSommaValori() {
      initLista();
      if (pgm.calcolaSomma() == 2087.21) System.out.println("OK test somma val");
      else System.out.println("FAILED test somma val");
   }

   public void testListaVuota() {
      pgm.svuota();
      if (pgm.calcolaSomma() == 0) System.out.println("OK test somma val empty");
      else System.out.println("FAILED test somma val empty");
      if (pgm.calcolaMassimo() == 0) System.out.println("OK test massimo val empty");
      else System.out.println("FAILED test massimo val empty");
   }
}
```


Qualità

- **Correttezza**: è stato possibile eseguire test che verificano il codice. Soddisfare (e verificare) i requisiti permette di valutare e ottenere la qualità del sistema software
  - **I test documentano le condizioni sotto le quali il codice funziona e proteggono il codice da modifiche indesiderate**. I test devono essere **indipendenti** fra loro e **auto valutarsi** (bisogna confrontare il risultato col valore atteso)
  - **Test Driven Development (TDD)**: implementare prima il test, e dopo il codice che risolve un requisito. Durante l'implementazione del test, si scelgono nomi di classi e metodi (si sta facendo progettazione)
    - **Motto: Red Green Refactor**
    - Red, test non superato: si possono implementare (e modificare) le classi
    - Green, test superato: si possono implementare test o si può fare refactoring
    - Refactor: migliorare la struttura del codice

Qualità

- **Responsabilità**: i compiti sono suddivisi su vari metodi (e quindi su più classi), questo permette di ottenere **coesione** del codice, ovvero non vi è un metodo (o una classe) che implementa tanti compiti
  - **Principio di Singola Responsabilità (SRP)**. La singola responsabilità è cruciale per la comprensione, il riuso, l'ereditarietà (OOP)
  - SRP: bisogna avere un solo motivo per cambiare il codice di quella parte, quindi il componente deve svolgere un solo piccolo compito
- **Astrazioni**: lo sviluppatore OOP costruisce astrazioni. Tipicamente, le astrazioni sono classi che nascondono dettagli di implementazione. Il nome delle astrazioni è estremamente importante: il **nome che si dà a classi e metodi ne descrive l'obiettivo**

Princìpi

- I princìpi fondamentali dell'ingegneria del software possono essere riassunti con i seguenti concetti
- **KISS** (Keep it Simple, Stupid): produrre componenti semplici, facili da capire e gestire
- **DRY** (Don't Repeat Yourself): evitare ripetizioni nel codice
- **YAGNI** (You Ain't Gonna Need It): non progettare funzionalità che non sono attualmente utili
- **LoD** (Law of Demeter): non parlare con gli estranei, un oggetto dovrebbe avere conoscenza solo di oggetti correlati (amici)

Princìpi Per I Test

- **FIRST**: Fast, Isolated/Independent, Repeatable, Self-validating, Timely
- **TDD**: Test-Driven Development
  - Prima implementare il test e poi il codice dell'applicazione
- **BDD**: Behaviour-Driven Development
  - Given (dato un contesto)
  - When (quando succede un evento)
  - Then (allora il sistema deve rispondere in un certo modo)

Ulteriori Risorse

- Per i tool di sviluppo e il runtime Java: www.java.com
- Per compilare una classe Java da shell dei comandi
  - javac NomeClasse.java, es. `javac Pagamenti.java`
- Se la compilazione va a buon fine non viene dato nessun messaggio e viene generato un file NomeClasse.class
- Per eseguire un programma Java
  - java NomeClasse, es. `java Pagamenti`
- La classe che si dà come argomento del comando java deve avere il metodo main (`public static void main(String[] args) { … }`), che è il metodo da cui comincia l'esecuzione del codice
- La classe Java specifica con `import` dove trovare il tipo che fa parte della libreria Java ed è usato all'interno della classe (per es. `List`, `File`), es. `import java.io.File`, inserito all'inizio del file NomeClasse.java

Conclusioni

- **Key points**
  - Test driven development
  - Single Responsibility Principle
- Esempi di domande d'esame
  - Implementare un test per un metodo che prende in ingresso un intero
  - Dire come si compila una classe in Java
  - Implementare una classe Java che può essere data alla JVM per essere eseguita