Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

La Programmazione Funzionale

Anno Accademico 2020\2021

Candidato:

Emilio Palermo

matr. N46003229

Dedico la tesi ai miei genitori che mi hanno dato la possibilità di studiare e ai miei due nonni che mi hanno sempre sostenuto

Indice

Indice....4
Introduzione....5
Capitolo 1: La Programmazione Funzionale....7
1. Accenni di storia....8
2. Perché la programmazione funzionale è così importante oggi....8
3. I linguaggi di programmazione funzionale più importanti oggi....9
1.4 Considerazioni sull'efficienza....10
Capitolo 2: Le Lambda Expression....11
1. Introduzione alle espressioni Lambda e esempi.....14
Capitolo 3: Confronto con object oriented....21
Conclusioni....26
Bibliografia....27

Introduzione

Java da quando è nata, parliamo degli inizi degli anni novanta, ha sempre supportato uno stile di programmazione orientata agli oggetti e alla tipizzazione statica. Con l'introduzione sul mercato della versione Java 8, avviene un grande cambiamento, il nostro linguaggio di programmazione ad alto livello inizia a supportare anche la programmazione funzionale.

La domanda è lecita, cosa si intende precisamente per programmazione funzionale?

Il paradigma della programmazione funzionale (FP) si evolve dall'invenzione del Lambda Calculus di Alonzo Church nel 1936, che è un modello di computazione che fornisce un modo molto formale di descrivere la valutazione di funzioni.

Con l'aiuto di uno stile di programmazione dichiarativo, FP pone maggior accento sulla definizione di funzioni, rispetto ai paradigmi procedurali e imperativi, che invece prediligono la specifica di una sequenza di comandi da eseguire. In questi ultimi, i valori

vengono calcolati cambiando lo stato del programma attraverso delle assegnazioni; un programma funzionale, invece, è immutabile: i valori non vengono trovati cambiando lo stato del programma, ma costruendo nuovi stati a partire dai precedenti. Nel primo capitolo di questo elaborato, verrà spiegato più dettagliatamente quella che è la programmazione funzionale.

Facciamo una breve panoramica dei pilastri su cui si basa la programmazione funzionale:

-Funzioni pure.

-Trasparenza Referenziale.

-Immutabilità.

-Ricorsione.

-Funzione di ordine superiore (funzione higher-order).

-Funzioni composte.

Ritornando adesso alla nostra versione di Java, l'aggiunta di lambda rende possibile utilizzare uno stile di programmazione più funzionale. Soprattutto gli stream e gli Optionals ne fanno un largo uso. Nel corso di questo elaborato ci chiederemo, ma le nuove funzionalità reggono contro l'attuale paradigma di FP?

Capitolo 1: La Programmazione Funzionale

Prima di addentrarci nel mondo di Java, cercheremo di dare una spiegazione dettagliata di quella che è la programmazione funzionale, come è nata, quali sono i vantaggi e gli svantaggi.

Cerchiamo prima di tutto di definire bene il concetto di programmazione funzionale, come viene suggerito dallo stesso nome, è un approccio funzionale nella programmazione che si concentra sulle funzioni. In un programma funzionale, si possono intendere tutti gli elementi come funzioni, e il codice viene eseguito tramite richieste di funzioni sequenziali. È molto importante chiarire che non ci sono assegnazioni autonome di valori. Una funzione può essere considerata come una versione speciale di subroutine, cioè è riutilizzabile, e a differenza di una procedura, il risultato è restituito direttamente.

In molti linguaggi di programmazione specializzati, vengono definite ed utilizzate diverse funzioni. È importante capire che non è questo il tratto distintivo della nostra programmazione funzionale. Le funzioni nella FP possono assumere strutture diverse, e questo garantisce una certa versatilità, ed è proprio questa versatilità che rende tale approccio così importante per l'Informatica. Le funzioni possono essere collegate tra di loro

come dati, utilizzate come parametri e anche come risultati funzionali. Grazie a tutto questo, i programmatori possono implementare ed elaborare compiti di calcolo di ampia portata.

Ma quando nasce precisamente la nostra programmazione funzionale? Diamo un accenno di storia.

1. Accenni di storia

Il Lambda calcolo può essere considerato il primo linguaggio di programmazione funzionale, anche se inizialmente non fu progettato per essere eseguito da una macchina. Che cos'è quindi il Lambda calcolo? Il Lambda calcolo è un modello di computazione progettato da Alonzo Church, e serve a descrivere in un modo formale la valutazione di funzioni. Il primo linguaggio di programmazione funzionale, progettato da un computer, è sicuramente l'Information Processing Language (IPL) nato a metà degli anni cinquanta. Successivamente si hanno delle migliorie con il Lisp, nato alla fine degli anni cinquanta, che però non è puramente un linguaggio funzionale, ma ha introdotto delle caratteristiche oggi trovate nei moderni linguaggi funzionali. Negli anni settanta, nell'università di Edimburgo nasce il linguaggio ML, mentre alla fine di questo decennio nascerà un dialetto del Lisp, basato sul Lamba calcolo, denominato Scheme. La prima versione del linguaggio Haskell viene introdotta nel 1990, e nasce con lo scopo di mettere insieme molte delle idee della ricerca sulla programmazione funzionale per creare un linguaggio puro.

2. Perché la programmazione funzionale è così importante oggi

Dopo questo breve accenno di storia, ci domandiamo perché la programmazione funzionale, nonostante le radici risalgono agli anni 30, gode ancora di grande popolarità, soprattutto in campo tecnico e matematico. Le ragioni sono molteplici, grazie al FP, abbiamo ampie possibilità di trasformare e sintetizzare il programma algebrico, in più grazie all'eliminazione degli stati interni nel processo di calcolo (a differenza della programmazione imperativa non stono necessari stati interni di un processo di calcolo) e "degli effetti collaterali", abbiamo semplici opzioni di analisi semantica.

Con la programmazione funzionale abbiamo un alto grado di astrazione perché ci basiamo sul concetto matematico e sul principio di funzionamento. Se usiamo adeguatamente questo tipo di programmazione possiamo ottenere un codice molto preciso. Grazie a queste unità piccole (funzioni) , ripetutamente utilizzabili e molto specializzate, viene fornito un programma per la risoluzione di un compito sostanzialmente più complesso.

3. I linguaggi di programmazione funzionale più importanti oggi

Tra i linguaggi di programmazione più importanti basati sull'approccio funzionale vi sono:

-LISP

-ML

-Haskell

-OCaml

-F#

-Erlang

-Clojure

-Scala

Inoltre, ci sono altri noti linguaggi di programmazione che consentono di scegliere la programmazione funzionale tra i diversi paradigmi possibili:

-Perl

-Ruby

-Visual Basic .NET

-Dylan

-ECMAScript

1.4 Considerazioni sull'efficienza

Concludendo questo capitolo, è bene fare alcuni importanti considerazioni sulla programmazione funzionale. Inizialmente con i primi linguaggi funzionali non si teneva conto dell'efficienza, e quindi per lungo tempo sono stati etichettati come linguaggi mangia-risorse, in termini di CPU e in termini di memoria, questo ovviamente andava a favore dei linguaggi non funzionali che guadagnavano in velocità, lasciando al programmatore alcuni compiti di livello più basso.

L'aumento delle prestazioni dei calcolatori ha trasferito l'attenzione sullo sviluppo rapido del software, sulla sua correttezza e manutenibilità. Oggi abbiamo l'introduzione di tecniche del tipo suddetto in linguaggi imperativi. Grazie a ciò oggi le performance dei linguaggi funzionali e dei linguaggi imperativi possono convergere. Per programmi che passano la maggior parte del tempo facendo computazioni numeriche, alcuni linguaggi funzionali possono avvicinarsi alla velocità del C, che è un linguaggio operativo. I linguaggi funzionali però, possono diventare considerevolmente più lenti dei linguaggi imperativi quando si tratta di manipolare grandi strutture dati, per via dell'utilizzo meno ottimizzato della memoria.

Utilizzando strutture dati persistenti, possiamo ottimizzare l'utilizzo della memoria nella programmazione funzionale. Queste strutture dati rendono la copia e la modifica relativamente economici, permettendo ad una parte o alla totalità di dati di essere condivisi con altri valori. Queste operazioni vengono svolte in modo sicuro dato che queste strutture dati sono immutabili e di conseguenza non sorgono quelli che sono gli errori dovuti alla gestione dei puntatori, come invece avviene nelle strutture dati imperative. Tra le strutture dati persistenti comunemente usate vi sono le liste concatenate e gli alberi binari.

Grazie allo sviluppo di questi linguaggi, che li ha resi più moderni, grazie alle loro sofisticate strutture dati e algoritmi, la programmazione funzionale oggi viene utilizzata in vaste aree scientifiche, come l'analisi numerica.

Capitolo 2: Le Lambda Expression

Dopo aver capito storicamente e teoricamente cos'è la programmazione funzionale, non ci resta altro che addentrarci nel mondo Java e capire nella pratica come possiamo utilizzare questo tipo di paradigma di programmazione.

Come abbiamo già accennato, Java, prima dell'introduzione dell'ottava versione, ha sempre supportato uno stile di programmazione imperativo o orientata agli oggetti, oggi però con Java 8 siamo in grado di utilizzare anche la programmazione funzionale.

Ricapitolando la FP è uno stile di programmazione dichiarativo piuttosto che imperativo. L'obiettivo fondamentale di questo stile di programmazione è rendere il codice più conciso, meno complesso, più prevedibile e più facile da testare. La programmazione funzionale si occupa di alcuni concetti chiave come la funzione pura, lo stato immutabile, la programmazione senza assegnazioni.

Prima di fare alcuni esempi, è importante capire la differenza tra programmazione funzionale e programmazione funzionale pura. I linguaggi di programmazione funzionali puri non consentono alcuna mutabilità, mentre un semplice linguaggio di programmazione funzionale fornisce funzioni di ordine superiore, spesso consentendo la mutabilità, e questo può essere un rischio perché potremmo andare in contro a degli errori. Quindi, in generale,

possiamo dire che se un linguaggio permette l'utilizzo di funzioni di ordine superiore è uno stile di programmazione funzionale, mentre se un linguaggio limita la mutabilità, siamo parlando di uno stile di programmazione puramente funzionale. Java è un linguaggio di stile funzionale, mentre Haskell è un linguaggio di programmazione puramente funzionale.

É utile comprendere alcuni concetti nella programmazione funzionale:

Funzione di ordine superiore: è una funzione che può prendere altre funzioni come parametri e/o restituire funzioni come risultato.

In Java, la funzione più vicina a una funzione di ordine superiore è quella del (metodo) che accetta una o più espressioni lambda come parametri e restituisce un'altra espressione lambda. Ecco un esempio di una funzione di ordine superiore in Java:

```
public class HigherOrderFunctionClass {



    public   <T>   IFactory<T>   createFactory(IProducer<T>   producer,
IConfigurator<T> configurator) {

        return () -> {

           T instance = producer.produce();

           configurator.configure(instance);

           return instance;

        }

    }

}
```

Si noti come il metodo createFactory () restituisce un'espressione lambda come risultato. Questa è la prima condizione di una funzione di ordine superiore. Si noti inoltre che il metodo createFactory () accetta due istanze come parametri che sono entrambe implementazioni di interfacce (IProducer e IConfigurator).

Funzione Pura: Una funzione è chiamata funzione pura se restituisce sempre lo stesso risultato per gli stessi valori di argomento e non ha effetti collaterali come la modifica di un argomento (o variabile globale) o l'output di qualcosa. In parole più povere**,** una funzione si dice pura quando il suo output dipende esclusivamente dal suo input.

Ecco un esempio di funzione pura in Java:

```
public class ObjectWithPureFunction{



    public int sum(int a, int b) {

        return a + b;

    }

}
```

Sum () non ha effetti collaterali, il che significa che non modifica nessuno stato (variabile), al di fuori della funzione, da nessuna parte.

Funzione Lamba (o funzione anonima**):** è una funzione definita, e possibilmente chiamata, senza essere legata ad un identificatore. Le funzioni anonime sono utili per passare come argomento una funzione di ordine superiore. Ha solo un elenco di parametri e un corpo. La sintassi di un'espressione lambda è:

```
(parametro) -> corpo
```

Nel prossimo paragrafo verranno fatti diversi esempi.

Nessuno Stato: una regola del paradigma di programmazione funzionale è di non avere stato. Per "nessuno stato" si intende tipicamente che non ci sia nessuno stato esterno alla

funzione. La funzione non può fare riferimento a variabili membro della classe o dell'oggetto a cui appartiene.

Interfaccia Funzionale:

Le interfacce come ActionListener, la cui definizione è presente sotto, vengono chiamate in Java 8, interfacce funzionali, caratterizzate dalla presenza di un solo metodo. L'utilizzo di interfacce funzionali con le classi anonime interne sono un modello comune in Java. Oltre alle classi EventListener, interfacce come Runnable, Comparator o FileFilter sono da considerarsi in modo simile. Le interfacce funzionali sono sfruttate per l'utilizzo con le espressioni lambda.

```
1  package java.awt.event;
2  import java.util.EventListener;
3
4     public interface ActionListener extends EventListener {
5         public void actionPerformed(ActionEvent e);
6      }
```

1. Introduzione alle espressioni Lambda e esempi.

Riportiamo degli esempi pratici di espressioni lambda:

```
// espressione che prende in input due interi e restituisce la somma
(int x, int y) -> x + y

// espressione che prende in input una stringa e restituisce la sua lunghezza
s -> s.length()

// espressione senza argomenti che restituisce il valore 50
() -> 50

// espressione che prende in input una stringa e non restituisce nulla
(String s) -> { System.out.println("Benvenuto ");
                System.out.println(s); }
```

Le istruzioni di *break* e *continue* non si possono usare all'interno del blocco anche se sono permessi all'interno dei cicli. É necessario che ogni possibile ramo del flusso lanci un'eccezione o restituisca qualcosa nel caso il corpo produca un risultato.

Ora che abbiamo definito la sintassi e visto qualche semplice esempio vediamo di utilizzare le lambda nel caso delle interfacce funzionali viste sopra:

```
public class RunnableTest {
public static void main(String[] args) {

     System.out.println("=== RunnableTest ===");

    // Anonymous Runnable
    Runnable r1 = new Runnable(){

      @Override
      public void run(){
        System.out.println("Hello world old style!");
      }
    };

   // Lambda Runnable
   Runnable r2 = () -> System.out.println("Hello world with Lambda!");

   r1.run();
   r2.run();
 }
}
```

É evidente come l'utilizzo delle lambda, abbia ridotto un pezzo di codice da cinque linee a una soltanto.

Un altro esempio può essere:

```
public class ListenerTest {

   public static void main(String[] args) {

   // Anonymous ActionListener
   JButton testButton = new JButton("Test Button");
    testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anon Class");
      }
    });

    // Lambda ActionListener
    testButton.addActionListener(e -> System.out.println("Click Detected by

    // Swing stuff
    JFrame frame = new JFrame("Listener Test");
    . . .
  }
 }
```

In questo ultimo caso la lambda expression è passata come argomento di un metodo. Proseguiamo con gli esempi per evidenziare come le lambda expression riescono a semplificare il codice, rendendolo più leggibile.

Un tipo caso d'uso è quello di filtrare gli elementi di una *Collection* in base a dei criteri. Consideriamo il seguente scenario:

Consideriamo un social network italiano dove vengono inviati agli iscritti dei messaggi pubblicitari diversi in base alle caratteristiche dei soggetti e supponiamo di voler inviare tre messaggi differenti: uno indirizzato alle giovani donne (persone di sesso femminile di età compresa tra i 18 e i 29 anni), uno indirizzato a tutti gli iscritti di sesso maschile e uno indirizzato ai soli utenti di nazionalità estera. Vediamo come potremmo procedere per implementare questo tipo di scenario.

Vediamo la soluzione mediante l'uso delle "vecchie" classi anonime, utilizzandole in maniera congiunta ad un'interfaccia funzionale (*ITest.java*) con un unico metodo test che restituisce un boolean.

L'interfaccia ITest è una cosa simile:

```java
public interface ITest<T> {
    public boolean test(T t);
}
```

L'implementazione è la seguente:

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MessageSender3 {

 List<Persona> iscritti = Arrays.asList(
 new Persona("Mario", "Rossi", "M", 35, "italiana"),
 new Persona("Lucy", "Parker", "F", 22, "inglese"),
 new Persona("Gianni", "Bianchi", "M", 20, "italiana"),
 new Persona("Fabio", "Marchi", "M", 40, "italiana"),
 new Persona("John", "Simpson", "M", 18, "USA"),
 new Persona("Adele", "Fabi", "F", 20, "italiana")
 );

 public List<Persona> getIscrittiFiltratiPer(ITest<Persona> aTest){
 List<Persona> persone = new ArrayList<Persona>();
 for (Persona p:iscritti)
 if (aTest.test(p))
 persone.add(p);
 return persone;
 }

 public void sendMessage(String msg, List<Persona> persone){
 // Logica di Invio messaggio
 // .........

 System.out.println("Inviato messaggio a "+persone.size()+" iscritti");
 }
```

```
 public static void main(String[] args) {
    MessageSender3 ms = new MessageSender3();

// ------------ invio messaggio per giovani donne ---------------
 ms.sendMessage("messaggioX", ms.getIscrittiFiltratiPer(
 new ITest<Persona>(){
 @Override
 public boolean test(Persona p){
 return p.getSesso().equals("F") &amp;&amp; p.getEta() &gt; 17 &amp;&amp; p.
 }
 }
 ));

 // ------------ invio messaggio per iscritti maschi ---------------
 ms.sendMessage("messaggioY", ms.getIscrittiFiltratiPer(
 new ITest<Persona>(){
 @Override
 public boolean test(Persona p){
 return p.getSesso().equals("M");
 }
}
 ));
// ------------ invio messaggio per iscritti stranieri ---------------
 ms.sendMessage("messaggioZ", ms.getIscrittiFiltratiPer(
 new ITest<Persona>(){
 @Override
 public boolean test(Persona p){
 return !p.getNazionalita().equals("italiana");  
 }
 }
 ));
 }
}
```

Adesso riscriviamo il seguente codice utilizzando le lambda expression, utilizziamo l'interfaccia pradicate:

```
public interface Predicate<T> {
   public boolean test(T t);
}
```

Il metodo test prende in input un tipo generico e restituisce un risultato booleano che è proprio quello che fa al caso nostro e che ci permette di effettuare le selezioni. La versione finale del nostro esempio sarà:

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MessageSender4 {

    List<Persona> iscritti = Arrays.asList(
       new Persona("Mario", "Rossi", "M", 35, "italiana"),
       new Persona("Lucy", "Parker", "F", 22, "inglese"),
       new Persona("Gianni", "Bianchi", "M", 20, "italiana"),
       new Persona("Fabio", "Marchi", "M", 40, "italiana"),
       new Persona("John", "Simpson", "M", 18, "USA"),
       new Persona("Adele", "Fabi", "F", 20, "italiana")
    );

     public List<Persona> getIscrittiFiltratiPer(Predicate<Persona> pred){
      List<Persona> persone = new ArrayList<Persona>();
      for (Persona p:iscritti)
          if (pred.test(p))
              persone.add(p);

      return persone;
    }
```

```java
public void sendMessage(String msg, List<Persona> persone){
   // Logica di Invio messaggio
   // .........

   System.out.println("Inviato messaggio a "+persone.size()+" iscritti");
 }

public static void main(String[] args) {
  MessageSender4 ms = new MessageSender4();

  // Predicates
  Predicate<Persona> allGiovaniDonne = p -> p.getSesso().equals("F") && p.getEta() > 17 && p.getEta() < 30;
  Predicate<Persona> allMaschi = p -> p.getSesso().equals("M");
  Predicate<Persona> allStranieri = p -> !p.getNazionalita().equals("italiana");

  // ------------ invio messaggio per giovani donne ---------------
  ms.sendMessage("messaggioX", ms.getIscrittiFiltratiPer(allGiovaniDonne));

  // ------------ invio messaggio per iscritti maschi ---------------
  ms.sendMessage("messaggioY", ms.getIscrittiFiltratiPer(allMaschi));

  // ------------ invio messaggio per iscritti stranieri ---------------
  ms.sendMessage("messaggioZ", ms.getIscrittiFiltratiPer(allStranieri));
```

Notate che viene creato un predicate per ciascun gruppo di iscritti: allGiovaniDonne, allMaschi *e* allStranieri, inoltre si possono creare semplicemente dei nuovi predicati da passare al metodo di ricerca. Qui per questioni di chiarezza si è preferito usare i predicate ma un ulteriore semplificazione sarebbe stata possibile passando

le definizioni dei predicati, cioè le lambda expressions, direttamente al metodo di ricerca.

Giusto un caso per esempio:

```
// ------------ invio messaggio per iscritti stranieri ---------------
ms.sendMessage("messaggioZ", ms.getIscrittiFiltratiPer(p -> !p.getNazionalita().equals("italiana")));
```

Concludendo possiamo affermare che in questo nuovo approccio con l'uso delle lambda expression, il codice risulta compatto, facile da leggere, e non è ripetitivo.

Capitolo 3: Confronto con object oriented

Capiamo bene adesso, le differenze che intercorrono tra programmazione funzionale e la programmazione orientata agli oggetti. Ormai abbiamo ben definito la programmazione funzionale, prima di entrare a pieno nell'argomento, diamo una breve spiegazione di cos'è la programmazione a oggetti.

La programmazione orientata agli oggetti è un paradigma di programmazione che utilizza l'astrazione per creare modelli basati sul mondo reale. Utilizza diverse di programmazione, come modularità, polimorfismo o incapsulamento. Molti tra i linguaggi di programmazione più popolari come Java, C ++, Python e PHP supportano la programmazione orientata agli oggetti.

Nella OOP, ogni oggetto può ricevere messaggi, elaborare dati e inviare messaggi ad altri oggetti. Ogni oggetto può essere inteso come una piccola macchina indipendente con un ruolo o responsabilità ben definito. La programmazione a oggetti ha lo scopo garantisce un'ottima flessibilità e facilità di sviluppo nella programmazione ed è molto popolare nello sviluppo di software su larga scala.

Differenze Fondamentali:

Per rendere le cose più semplici, si può dire che tutti i programmi hanno due componenti di base, i dati (ciò che il programma sa) e il comportamento (ciò che il programma può fare con quei dati). Quindi la programmazione orientata agli oggetti afferma che la combinazione di dati e comportamenti convergano in un unico punto, chiamato appunto "oggetto", il che renderebbe più facile capire come funziona un particolare programma. D'altra parte, la programmazione funzionale sostiene che i dati e il comportamento sono cose diverse e dovrebbero essere separati per garantire la loro chiarezza.

Ogni paradigma di programmazione ha i suoi vantaggi e svantaggi, e bisogna conoscerli prima di utilizzarli effettivamente.

I linguaggi orientati agli oggetti funzionano bene quando si dispone di un insieme fisso di operazioni sulle “cose” e, man mano che il codice si sviluppa, si aggiungono principalmente cose nuove. Ciò può essere ottenuto aggiungendo semplicemente nuove classi che implementano metodi esistenti. D’altra parte la programmazione funzionale funziona meglio quando si dispone di un insieme fisso di cose e, man mano che il codice si sviluppa, si aggiungono principalmente nuove operazioni su cose già esistenti. Ciò può essere ottenuto aggiungendo nuove funzioni che eseguono calcoli utilizzando tipi di dati, e le funzioni già esistenti sono lasciate a loro stesse.

Differenze sui criteri chiave:

È molto importante suddividere questi due stili di programmazione in criteri chiave, marcare le differenze tra dati, modelli, supporti e altro:

Definizione:

Programmazione funzionale si concentra sulla valutazione delle funzioni, come spiegato precedentemente.

Programmazione a oggetti si concentra sul concetto di oggetto, come spiegato precedentemente

Dati:

Programmazione funzionale usa dati immutabili, cioè, i valori non vengono trovati cambiando lo stato del programma, ma costruendo nuovi stati partendo dai precedenti. Le strutture dati sono immutabili e di conseguenza non sorgono quelli che sono gli errori dovuti alla gestione dei puntatori, come invece avviene nelle strutture dati imperative.

Programmazione a oggetti usa dati mutabili, gli oggetti sono mutabili, possiamo cambiare lo stato di un oggetto con un'assegnazione ad uno dei suoi attributi.

Modello:

Programmazione funzionale segue un modello di programmazione dichiarativa. La programmazione funzionale pone maggiore attenzione sulla definizione di funzioni, rispetto ai paradigmi procedurali e imperative, che invece prediligono la specifica di una sequenza di comandi da seguire.

Programmazione a oggetti segue un modello di programmazione imperativa.

Supporto:

Programmazione funzionale supporta la programmazione parallela. Un uso particolare del paradigma, per l'ottimizzazione dei programmi, è quello di trasformare gli stessi per utilizzarli nella programmazione parallela. Grazie alla programmazione parallela si può verificare che un insieme di processi o sottoprocessi (threads) sia in esecuzione nello stesso istante.

Programmazione a oggetti non supporta la programmazione parallela.

Iterazioni:

Programmazione funzionale usa le ricorsioni. La programmazione funzionale fa largo uso della ricorsione, le funzioni ricorsive invocano se stesse permettendo di eseguire più e più volte una stessa ricorsione.

Programmazione a oggetti usa i loops.

Elementi Basici:

Programmazione funzionale usa funzioni e variabili.

Programmazione a oggetti usa oggetti e metodi.

Programmazione funzionale vantaggi e svantaggi:

Verranno presentati di seguito i vantaggi e svantaggi della programmazione funzionale, con il simbolo + tratteremo un vantaggio, col simbolo - invece uno svantaggio:

+ L'uso di funzioni pulite e trasparenti porta a risultati affidabili senza effetti collaterali che forniscono e restituiscono esattamente ciò che ti aspetti.

+ Utilizza uno stile più dichiarativo che si concentra maggiormente su ciò che deve essere fatto e meno su come farlo, con un'enfasi sull'efficienza e l'ottimizzazione.

- É un paradigma nuovo e talvolta non è così facile trovare documentazione o informazioni rispetto alla programmazione orientata agli oggetti.

-A volte potrebbe non diventare illeggibile a causa di un numero molto elevato di funzioni rispetto alla programmazione orientata agli oggetti.

Programmazione a oggetti vantaggi e svantaggi.

Come prima, il simbolo "+" per i vantaggi, il simbolo - per gli svantaggi:

+ Oggetti e metodi sono molto chiari e comprensibili.

+Usa uno stile imperativo, in cui il codice viene letto come un semplice insieme di istruzioni, proprio come lo leggerebbe un computer.

-Può portare a risultati non specificati e inaspettati, nel senso che un codice parallelo che avrebbe accesso a una risorsa comune potrebbe farlo in un modo che produce risultati non specificati.

-I metodi ts possono avere effetti collaterali e possono avere un impatto sui processori.

Conclusioni

Concludendo, possiamo dire che non è molto facile capire quale delle due sarebbe l'opzione migliore. Non si può negare che forse, attualmente, la programmazione a oggetti risulta più vantaggiosa da utilizzare, esistendo da più tempo, ci si può basare su molta più documentazione, mentre d'altra parte non si può negare che la programmazione funzionale può risultare più piacevole da utilizzare, essendo più pulita e compatta. Importante sottolineare che comunque i concetti dei due stili di programmazione sono molto diversi, e spetta sempre allo sviluppatore decidere quale tra le due conviene utilizzare per rendere il lavoro produttivo e più semplice.

Bibliografia

[1] Programmazione Funzionale, Wikipedia

[2] Programmazione Orientata agli oggetti, Wikipedia

[3] Lo sviluppatore, http://losviluppatore.it/java-8-introduzione-alle-lambda-expressions/

[4] Better Programming, https://medium.com/better-programming/functional-programming-with-java-an-introduction-daa783355731

[5] GeeksforGeeks, https://www.geeksforgeeks.org/functional-programming-in-java-with-examples/