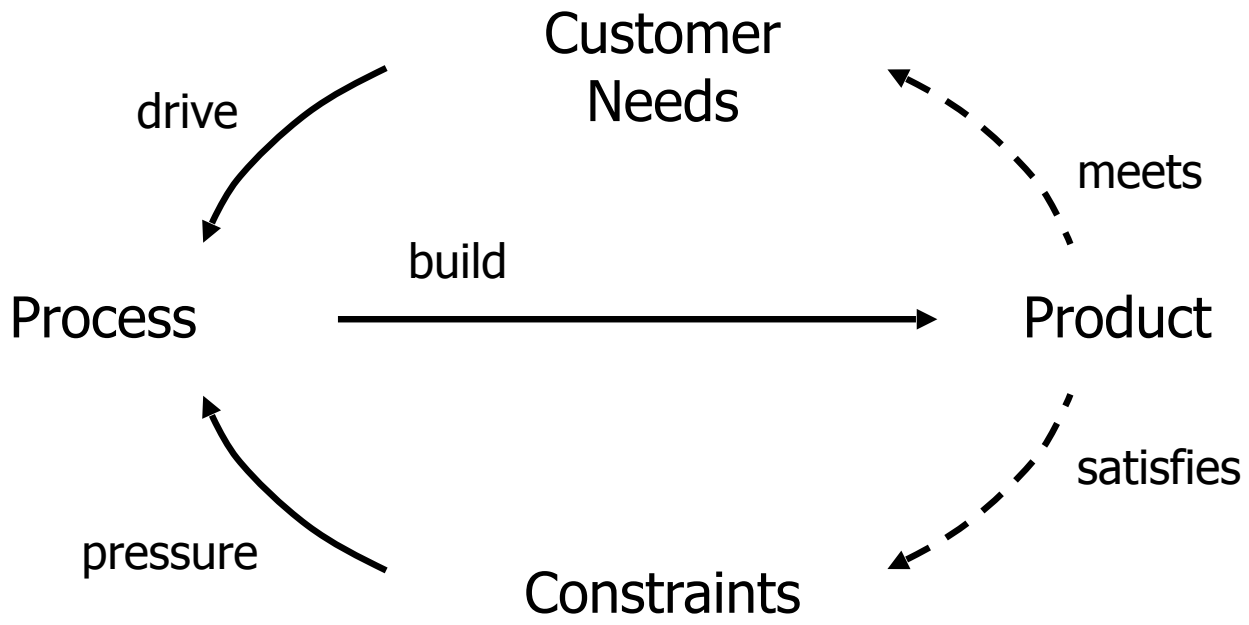
Programmare Object Oriented in Java

Prof. A.M. Calvagna

Software Engineering

Developers can control the Process (how they work) and Product (what they build)



What is a software process?

- A set of activities and their relationships to each other to support the development of a software system

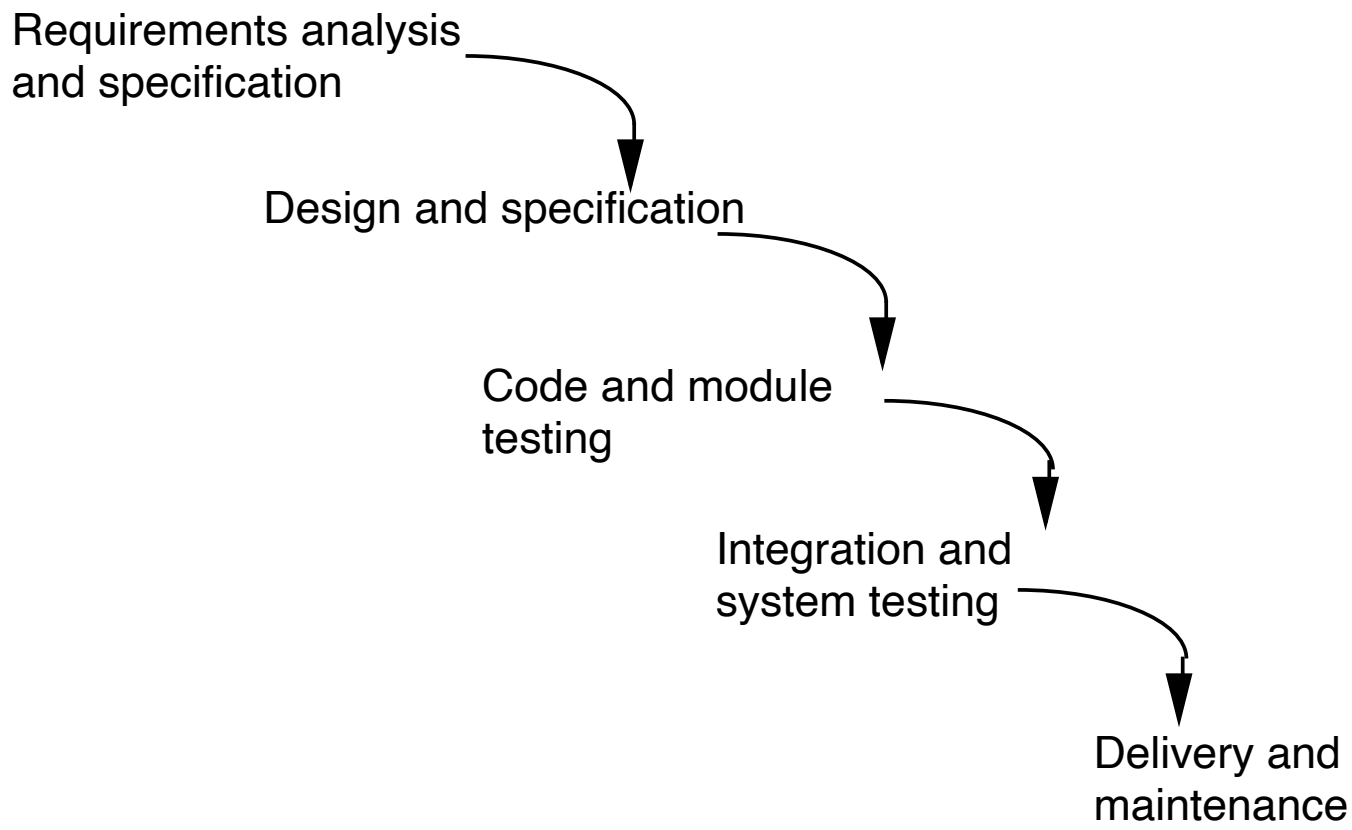
- Typical questions:
  - **Which activities** should I select for the software project?
  - What are the **dependencies** between activities?
  - How should I **schedule** the activities?

Software Process

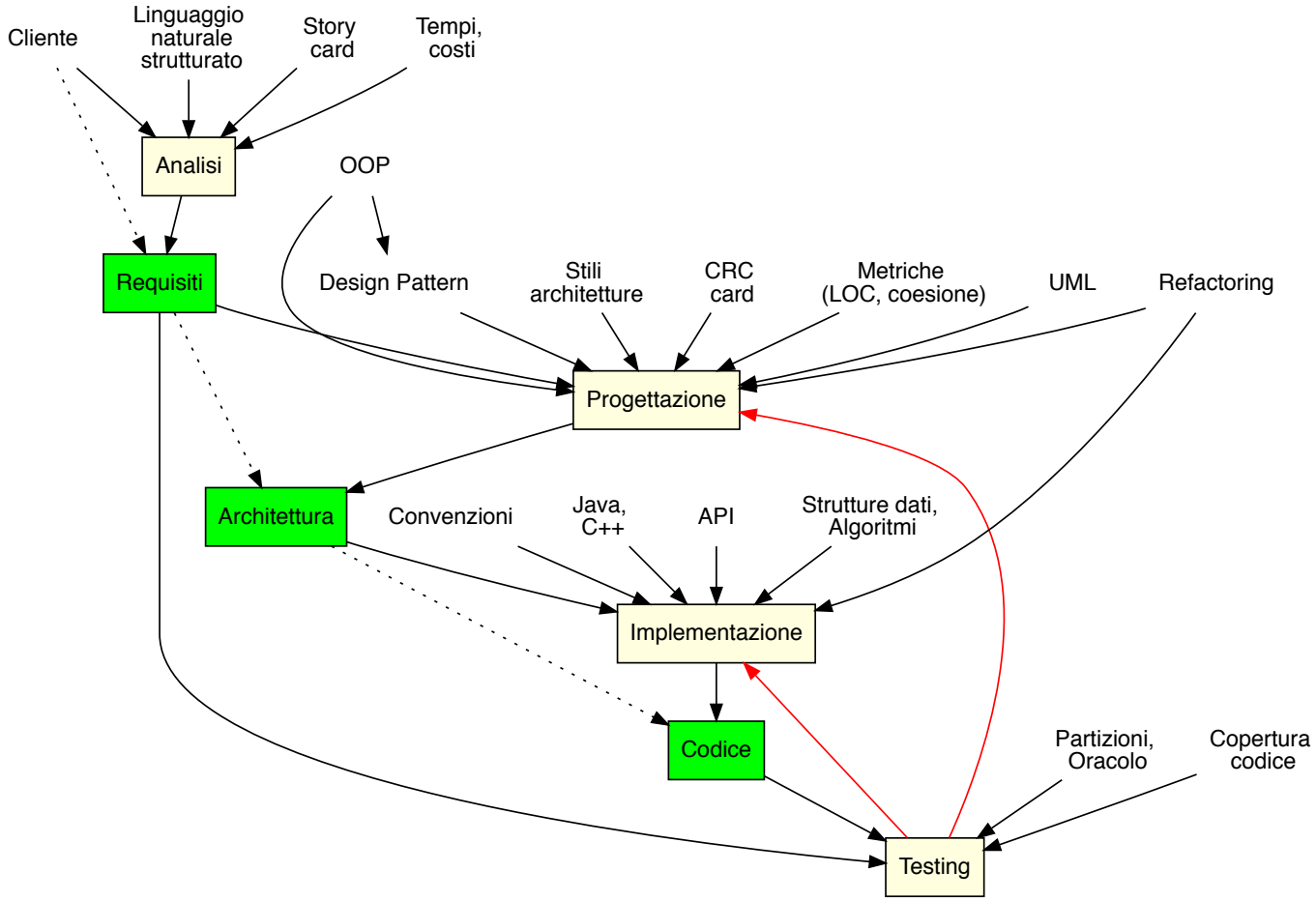
- Generic activities in all software processes are:
  - **Specification** - what the system should do and its development constraints
  - **Development** - production of the software system
  - **Validation** - checking that the software is what the customer wants
  - **Maintenance** - changing the software in response to changing demands.

Basic software lifecycle
(a preview)

Waterfall model

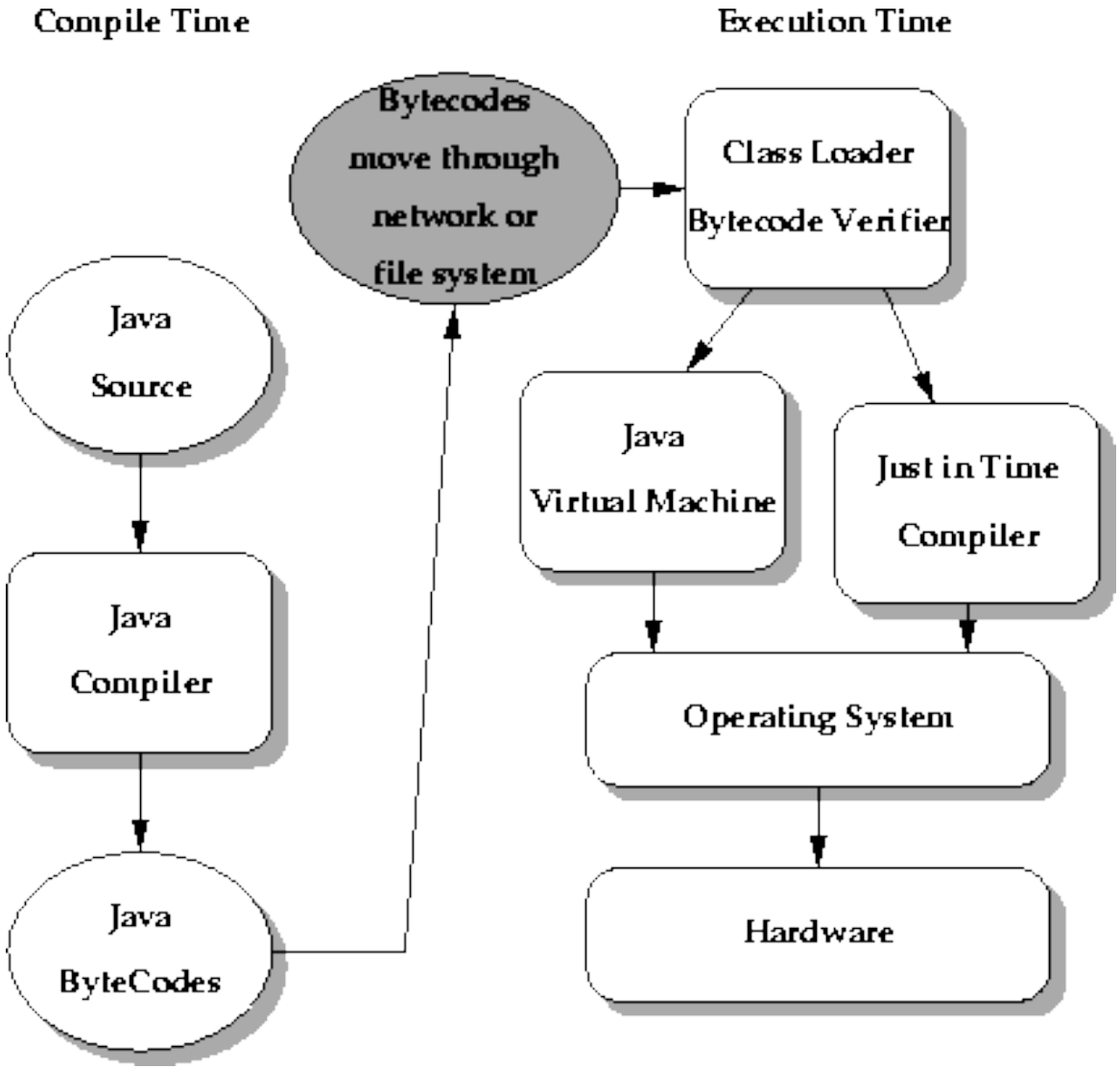


Basic software lifecycle (enhanced model)



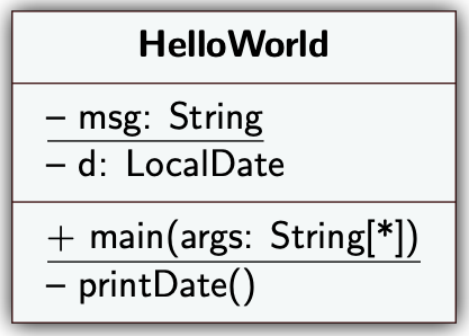
giallo = attività
verde = artefatto

Java Program Environment



Hello World in Java

```java
import java.time.LocalDate; // indica dove trovare la classe LocalDate

public class HelloWorld { // dichiara classe HelloWorld
   private static final String msg = "Lezione di Ingegneria del Software";
   private LocalDate d; // dichiara campo d di tipo LocalDate

   public static void main(String[] args) {
      System.out.println("Hello World”); // scrive su schermo
      System.out.println(msg);
      final HelloWorld world = new HelloWorld(); // crea oggetto
      world.printDate(); // chiama metodo
   }

   private void printDate() { // metodo
      d = LocalDate.now();    // chiama metodo static now
      System.out.println(d);
   }
}
```



- Il codice della classe HelloWorld deve essere salvato sul file `HelloWorld.java`, compilato con `javac HelloWorld.java` ed eseguito con `java HelloWorld`

Output
```
Hello World
Lezione di Ingegneria del Software
2024-03-07
```

Parole chiave di Java

- **`class`** permette di definire un tipo, e quindi le sue istanze
- **`final`** definisce un campo o una variabile che non può essere assegnata più di una volta (una costante). Una classe final non può essere ereditata, un metodo final non può essere ridefinito (override)
- **`import`** indica dove trovare la definizione di una classe che sarà usata nel seguito
- **`new`** permette di creare un'istanza di una classe
- **`private`** e **`public`** indicano l'accessibilità di classi, campi e metodi
- **`static`** è usata per dichiarare un campo o un metodo appartenente alla classe (e non all'istanza)
- **`void`** indica che il metodo non ritorna alcun valore
- Tipi usati: **`String`**, per rappresentare insiemi di caratteri; **`LocalDate`**, per accedere alla data attuale; **`System`** per scrivere sullo schermo

Differenze C++ vs Java

- Il significato del separatore punto . È lo stesso di —> in C++
  - Gli identificatori in Java sono tutti puntatori
- Non ci sono che oggetti, nulla può essere definito al di fuori di un costrutto di classe (o interfacce)
- Non serve il punto e virgola alla fine dei blocchi dichiarativi delle classi {}
- In una Classe metto contestualmente sia l'astrazione (l'ADT) che la sua implementazione
- È fortemente tipizzato (strong type checking)
  - I tipi ai lati di un assegnamento devono essere formalmente compatibili a tempo di compilazione
- Controlla anche il tipo dinamico
  - un identificatore punta a oggetti di tipo diverso in momenti diversi dell' esecuzione
  - Il tipo effettivo a runtime deve essere compatibile col tipo a cui lo assegni
-

Differenze C++ vs Java

- Non supporta l'ereditarietà multipla
  - Problema del diamante
  - Possibile con le interfacce
- La radice della gerarchia delle classi in Java è unica ed è **la classe Object**
- Libreria standard java è nel dominio java.*
- Una sola classe per file con lo stesso nome
- Il main definito per forza dentro una classe
- gli include non indicano interi file (header) ma ho visibilità a livello del mio package e delle classi che importo singole o a gruppi indicandone il nome composto per intero

un esempio pratico

- Riusciamo a sviluppare un componente software che risulti: riusabile, modificabile e corretto?
- Consideriamo un componente estremamente piccolo (potrebbe far parte di un sistema più grande)
- Descrizione dei requisiti
  - Dati vari file contenenti valori numerici, con un valore per ciascuna riga del file
    1. Leggere da ciascun file la lista di valori
    2. Tenere solo i valori non duplicati
    3. Calcolare la somma dei numeri letti dal file (non duplicati)
    4. Calcolare il massimo fra i numeri letti

```java
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

public class CalcolaImporti {  // classe Java vers 0.0.1
    private final List<String> importi = new ArrayList<>();
    // List e ArrayList sono tipi della libreria Java
   private float totale;

    public float calcola(String c, String n) throws IOException { // metodo
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      // lettura file tramite le API Java: File, FileReader, LineNumberReader

      totale = 0;
         while (true) {
             final String riga = f.readLine();   // legge una riga dal file
             if (null == riga) break;            // esce dal ciclo
             importi.add(riga);                  // aggiunge in lista
             totale += Float.parseFloat(riga);   // converte da String a float
         }
         f.close();      // chiude file
         return totale;  // restituisce totale al chiamante
    }
}
```

Linguaggio Java

- Parole chiave
  - **`float`** si usa per dichiarare una variabile che può tenere numeri in virgola mobile; si usa pure per dichiarare che un metodo restituisce un valore float
  - **`if`** si usa per creare un'istruzione condizionale
  - **`return`** si usa per concludere l'esecuzione di un metodo, se seguita da un valore quest'ultimo è restituito al chiamante
  - **`throws`** si usa nelle dichiarazioni di metodi per indicare quali eccezioni non sono gestite dal metodo ma passate
  - **`while`** si usa per creare un ciclo

JAVA Collections Framework

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
- Da fare
  - controlli su valori unici
  - estrazione del massimo

```
public class CalcolaImporti {  // classe Java vers 0.0.2
     private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

     public float calcola(String c, String n) throws IOException {
          LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
       // lettura file tramite le API Java: File, FileReader, LineNumberReader

       totale = massimo = 0;
          while (true) {
               final String riga = f.readLine();   // legge una riga dal file
               if (null == riga) break;            // esce dal ciclo
               importi.add(riga);                  // aggiunge in lista
               float x = Float.parseFloat(riga);   // converte da String a float
           totale += x;                          // aggiorna totale
               if (massimo < x) massimo = x;       // aggiorna massimo
          }
          f.close();      // chiude file
          return totale;  // restituisce il totale al chiamante
     }
}
```

Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
  - estrazione del massimo
- Da fare
  - controlli su valori unici

```java
public class CalcolaImporti {  // classe Java vers 0.1

     private final List<String> importi = new ArrayList<>();
   private float totale, massimo;

   public float calcola(String c, String n) throws IOException {
      LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      // lettura file tramite le API Java: File, FileReader, LineNumberReader

      totale = massimo = 0;
      while (true) {
         final String riga = f.readLine();   // legge una riga dal file
         if (null == riga) break;            // esce dal ciclo
         if (!importi.contains(riga)) {      // se non presente
            importi.add(riga);               // aggiunge in lista
            float x = Float.parseFloat(riga); // converte da String a float
            totale += x;                     // aggiorna totale
            if (massimo < x) massimo = x;    // aggiorna massimo
         }
      }
      f.close();      // chiude file
      return totale;  // restituisce il totale al chiamante
   }
}
```

Librerie Java

- Riepilogo di alcuni tipi e metodi di librerie Java utilizzati
- `List`, interfaccia utile a tenere una sequenza di elementi
- `ArrayList`, implementazione di `List`, la sua dimensione cresce automaticamente
- `add()`, metodo di `List`, aggiunge un elemento alla fine della lista
- `contains()`, metodo di `List`, ritorna true se la lista contiene l'elemento specificato
- `parseFloat(String s)`, metodo di `Float`, ritorna un nuovo float con il valore specificato nel parametro stringa s, o ritorna un'eccezione se la stringa non contiene un numero
- `readLine()`, metodo di `LineNumberReader`, ritorna una stringa contenente la linea del file, o `null` se si raggiunge la fine del file

Progressi

- Passi implementati
  - lettura da file
  - calcolo del totale
  - estrazione del massimo
  - controlli su valori unici
- Il codice è conforme alla programmazione OO?
- E se il codice prodotto fosse invece …

```java
public class CalcolaImporti {  // classe Java vers 0.2

     private final List<String> importi = new ArrayList<>();
   private float totale, massimo;

   public float calcola(String c, String n) throws IOException {
      LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      String riga;
      while (true) {
         riga = f.readLine();
         if (null == riga) break;
         if (!importi.contains(riga))
            importi.add(riga);
      }
      f.close();
        // calcola totale
      totale = 0;
      for (int i = 0; i < importi.size(); i++)
         totale += Float.parseFloat(importi.get(i));
        // calcola massimo
      massimo = Float.parseFloat(importi.get(0));
      for (int i = 1; i < importi.size(); i++)
         if (massimo < Float.parseFloat(importi.get(i)))
            massimo = Float.parseFloat(importi.get(i));
      return totale;
   }
}
```

Problemi?

- Il metodo `calcola` di entrambe le versioni è **spaghetti code** (un antipattern)
- Il codice è monolitico: fa troppe cose in un unico flusso. Non è un codice Object-Oriented. Conseguenze: non si può riusare, né verificarne la correttezza
    1. Come verificare che tutti i valori del file siano stati letti? Si dovrà modificare il metodo. Non è una soluzione, si dovrebbe poter verificare il comportamento del metodo dall'esterno
    2. Analogamente per verificare il calcolo di somma e totale, in più punti si dovrebbero aggiungere alcuni controlli
    3. Non si riesce a modificare facilmente o riusare il codice. Per es. se si volessero conservare tutti i valori letti, quali ulteriori effetti provoca la modifica?
- Quindi: difficoltà di comprensione e modifiche che coinvolgono varie operazioni

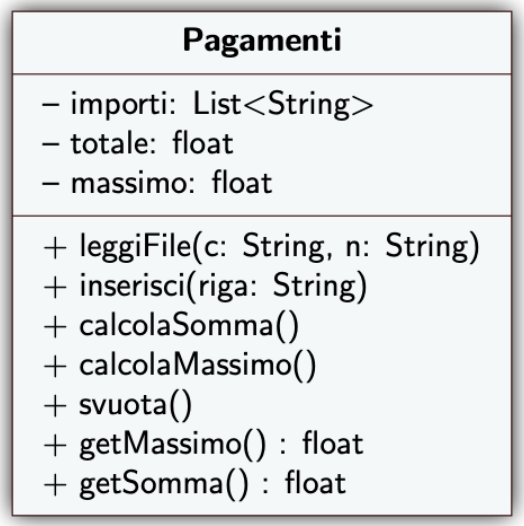
Spaghetti Code

- Metodi lunghi, senza parametri, e che usano variabili globali
- Flusso di esecuzione determinato dall'implementazione interna all'oggetto, non dai chiamanti
- Interazioni minime fra oggetti
- Nomi classi e metodi indicano la programmazione procedurale
- Ereditarietà e polimorfismo non usati, riuso impossibile
- Gli oggetti non mantengono lo stato fra le invocazioni
- Cause: inesperienza con OOP, nessuna progettazione

```
public class CalcolaImporti {  // classe Java vers 0.2

    private final List<String> importi = new ArrayList<>();
   private float totale, massimo;

   public float calcola(String c, String n) throws IOException {
      LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      String riga;
      while (true) {
         riga = f.readLine();                                          leggiFile()
         if (null == riga) break;
         if (!importi.contains(riga))                                inserisci()
            importi.add(riga);
      }
      f.close();
      totale = 0;                                                    calcolaSomma()
      for (int i = 0; i < importi.size(); i++) {
         totale += Float.parseFloat(importi.get(i));
      }
      massimo = Float.parseFloat(importi.get(0));                  calcolaMassimo()
      for (int i = 1; i < importi.size(); i++)
         if (massimo < Float.parseFloat(importi.get(i)))
            massimo = Float.parseFloat(importi.get(i));
      return totale;                                                   getSomma()
   }
}
```

```java
public class CalcolaImporti {  // classe Java vers 0.1

    private final List<String> importi = new ArrayList<>();
   private float totale, massimo;

   public float calcola(String c, String n) throws IOException {
      LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
      totale = 0;
      massimo = 0;
      while (true) {                                          leggiFile()
         String riga = f.readLine();
         if (null == riga) break;
         if (!importi.contains(riga)) {                       inserisci()
            importi.add(riga);
            float x = Float.parseFloat(riga);                 calcolaSomma()
            totale += x;
            if (massimo < x) massimo = x;                     calcolaMassimo()
         }
      }
      f.close();
      return totale;                                          getSomma()
   }
}
```

```java
public class Pagamenti { // Pagamenti vers 1.1
   private List<String> importi = new ArrayList<>();
   private float totale, massimo;
   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 (null == riga) break;
          inserisci(riga);
      }
      f.close();
   }
    public void inserisci(String riga) {
      if (!importi.contains(riga)) importi.add(riga);
   }
    public void calcolaSomma() {
         totale = 0;
      for (String v : importi)    // enhanced for
          totale += Float.parseFloat(v);
   }
    public void calcolaMassimo() {
         massimo = 0;
      for (String v : importi)
          if (massimo < Float.parseFloat(v))
              massimo = Float.parseFloat(v);
   }
    public void svuota() {
         importi = new ArrayList<>();
         totale = massimo = 0;
    }
    public float getMassimo() {
      return massimo;
    }
    public float getSomma() {
      return totale;
    }
}
```




```java
// chiamate da un'altra classe
public static void main(String[] args) {
    Pagamenti p = new Pagamenti();
     try {
        p.leggiFile("csv","importi");
     } catch (IOException e) {}
    p.calcolaSomma();
    p.calcolaMassimo();
}
```

Considerazioni sul codice

- Si sta usando bene il paradigma di programmazione ad Oggetti (OOP)
  - Ogni metodo ha una sola piccola responsabilità
  - Il flusso di chiamate ai metodi è indipendente dai singoli algoritmi
  - Posso riusare (richiamandoli) i servizi offerti dai metodi
- Inoltre, sto usando il **paradigma Command e Query**
  - I metodi Query restituiscono un risultato (si vede dal parametro di ritorno), e non modificano lo stato del sistema
  - I metodi Command (o modificatori) cambiano lo stato del sistema ma non restituiscono un valore
  - I metodi query si possono chiamare liberamente, senza preoccupazioni sulla modifica dello stato, mentre si deve stare più attenti quando si chiamano i metodi command
- Enhanced for indica che si vogliono gli elementi della lista, uno per ogni passata, si può usare con i tipi che implementano `Iterable`

Metriche

- Classe CalcolaImporti (vers. 0.1)
  - Metodi 1, LOC 26 (di cui 5 linee sono per i vari import)
- Classe CalcolaImporti (vers. 0.2)
  - Metodi 1, LOC 29
- Classe Pagamenti (vers 1.1)
  - Metodi 7, LOC 43 (media 6 LOC per metodo)
- Confronto con sistemi software open source (valori approssimativi) JUnit (JU), JHotDraw (JHD):
  - JU LOC 22K, Classi 231, Metodi 1200, Attributi 265, media 18
  - JHD LOC 28K, Classi 600, Metodi 4814, Attributi 1151, media 6

Conclusioni

- **Key points**
  - Correttezza del codice: test
  - Antipattern Spaghetti Code
  - Ciascun metodo ha un unico compito
- Esempi di domande d'esame
  - Implementare un frammento di codice che usa l'enhanced for
  - Dire come si può controllare se un codice è corretto
  - Implementare un metodo query
  - Dire qual è la differenza fra List ed ArrayList
  - Dire a cosa serve il metodo contains di List

Attività di gruppo in classe

- In un foglietto, carta e penna, scrivete il codice che implementa un generico stack
  - Struttura dati lineare e omogenea con protocollo LIFO
  - Generico? dimensione e tipo sono parametrici
- Usare il linguaggio C o C++, a vostra scelta
- non usare librerie predefinite
- Consegnate l'attività solo se volete un feedback indivuale
- Non è una valutazione
- Tempo 15 minuti

Stack versione nuda e cruda

- Molto efficiente e prestante in termini di velocità di esecuzione ma… per niente astratto!

- Tipo di elementi nello stack generico, ma un solo tipo!
    - Typedef crea un solo alias
- Un solo stack in tutto il sistema: è una variabile globale
- Lo stato dello stack è totalmente esposto

- **Problemi che avremmo con qualsiasi codice, non solo dello stack**

Stack versione migliorata

- Incapsulamento (data hiding)
  - Uso un modulo separato per definire lo stato e il protocollo

- **Lo stato dello stack non è più esposto**
- **Implementazione modulare e riusabile, ma…**
- Tipo di elementi nello stack generico, ma un solo tipo per tutti
- Un solo stack in tutto il sistema: è una variabile globale
- **Inquinamento dello spazio dei nomi**
- **Non rientrante**

- come avere più di uno stack?

Stack versione ADT

- Definizione di uno stack come **Abstract Data Type**
  - Uso un modulo separato per definire lo stato e il protocollo
    - l'interfaccia è accessibile: il protocollo d'uso
    - l'implementazione, col suo stato, è inaccessibile (incapsulata)

- **Molteplici istanze di Stack, non è più una variabile globale**
- **Conflitto nello spazio dei nomi… risolto (manualmente)**
- **Rientrante**
- Tipo di elementi nello stack generico, ma uno solo per tutti

Stack versione ADT: problemi residui

- Ancora comunque una soluzione imperfetta e incompleta

- Non c'è garanzia di inizializzazione e terminazione dell'ADT
- Sta al programmatore ricordarsi di invocare sempre
    - Create(), all'inizio
    - Destroy(), alla fine

- Conflitto dei nomi… evitato manualmente, **in uno spazio globale**
- Le chiamate a funzioni hanno introdotto un **overhead** in più
    - Uso "inline" per alleviare il problema

Stack versione ADT: problemi residui

- Ancora comunque una soluzione imperfetta e incompleta
- Tipo di elementi nello stack generico, ma sempre uno per tutti
- Non c'è una gestione degli errori generalizzata…
- Il compilatore C non supporta realmente l'incapsulamento: è ancora possibile…violare l'astrazione
- Come avere incapsulamento reale e molteplici stack?

Stack versione Object Oriented

- Implemento l'ADT con lo strumento object oriented detto **classe**

- **incapsulamento e astrazione sono supportati dal compilatore**
    - Le violazioni sono segnalate in fase di compilazione
- Supporto molteplici stack contemporanei: le **istanze** di classe
- **l'inizializzazione e terminazione sono invocate in automatico**

Problemi residui:

La gestione degli errori è invadente nel codice e non generica

Introduco la **gestione delle eccezioni**

C'è ancora un solo tipo di stack per tutte le istanze!

Introduco le classi **parametriche** (templates/generics)

Implementazione in JAVA dello stack

Conclusioni

- Caratteristiche peculiari del paradigma Object Oriented

- Incapsulamento
- Ereditarietà
- Polimorfismo

- Ma cosa è una classe?
- Un tipo ?
- Un Oggetto?
- Un ADT ?
- E' solo uno strumento di codifica: il suo significato concettuale (la semantica) varia col linguaggio di programmazione considerato

Glossary of OO Terms and definitions

an **ADT** is only a ***specification*** of properties, a rule that concrete objects may or not happen to obey. it is a model abstracting ***many*** real-world entities.

an **Object** is a ***model*** of a *unique* entity, and an instance is its *working* dual. Each instance, like its corresponding object, is unique and distinguishable from all the others, including those of same *type,* through its "identity" (i.e., the *this* reference).

a **Module** is a ***unique*** collection of ***programs***, usable to realize an object instance.

an **Instance** is last stage: a program already allocated in memory and all is ready for execution. they are also usually called objects, informally, since the realize an Object model in the context of software systems.

a **Class** is a programming language ***syntactical artifact*** whose semantics could be an ADT, an Object, a Module, or a combination of those. A template implementation for object instances creation.

Glossary of OO Terms and definitions

- ***Class-Attribute***: a compile-time allocated (static) identifier, stored once in a Class, and shared by all its instances. The referenced object instance (which must also be static) gets shared too

- ***Instance-Attribute:*** an identifier *declared* in a Class, but run-time allocated inside all its instances, as separate copies. Thus, they can reference private instances of objects.

- ***Class-Method:*** a method globally available for execution. The receiver of the method call is the defining Class, thus instance-attributes can't be used inside (they are not yet allocated).

- ***Instance-Method***: an object method which can be called only through an Instance of its defining class, so that it will be able to operate on that receiver's instance-attributes.

- ***call dispatch:*** determining which code to execute in response to a message sent to an identifier. This can be done statically (early binding) if identifier has an associated static type, or dynamically (late binding) at run-time, based on the type of the currently referenced instance.

- ***self/this:*** is a *pseudo variable* referencing to the *receiver* instance of a method call (as an additional, *hidden*, formal parameter to a function call): it is the context in which the function has to be executed (execution class instance).
- ***super*** : is a *dispatch qualifier*. it is not an identifier (doesn't actually references nothing). it just means to start searching for a corresponding method (to bind to an operation call) from the parent class of the current object. the current object is the one actually defining (not executing) the method where super is used. put simply, is always the parent context (class) with respect to method definition.