22
Programmazione Object-Oriented Efficace

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2018/2019

Outline

Goal della lezione

- Discutere alcuni principi e metodologie di progettazione
- Ripassare varie linee guida fornite durante il corso
- Illustrarne di nuove largamente accettate

Argomenti

- DRY, KISS and SOLID principles
- Il testo: J.Block "Effective Java"
- Linee guide di interesse per il corso
- Alcuni approfondimenti

Note generali su questa lezione

Note:

- ri-consolidano vari aspetti già analizzati, ed alcuni nuovi
- va vista in prospettiva di una elevata professionalità nello sviluppo del SW
- contiene suggerimenti auspicabilmente applicati al progetto d'esame

Outline

SW mal progetatto..

Difetti

- rigidità — piccoli cambiamenti richiedono in cascata molte modifiche
  - questo porta a riluttanza a modificare il sistema, incapacità di stimare tempi
- fragilità — cambiamenti ad un modulo/classe, causano il malfunzionamento di altri
  - i cambiamenti fanno diventare il software "peggiore"
- immobilità — un componente/funzionalità già scritto non è facilmente estraibile/riusabile
  - causato da troppe e inutili dipendenze, si preferisce riscrivere daccapo
- viscosità — deployment e testing sono difficili da realizzare
  - è difficile/costoso verificare il funzionamento corretto di una nuova versione
- complessità (KISS) — certe porzioni di codice sono più complicate del necessario
  - sono più difficili da capire, modificare, testare
- ripetitività (DRY) — certe porzioni di codice sono sostanzialmente duplicate
  - a vari livelli; cioè rende il sistema meno coerente e manutenibile
- opacità — il codice non è comprensibile
  - scarsa documentazione, scarsa leggibilità (nomi variabili, formattazione)

I SOLID principles

5 principi cardine

- SRP: Single responsibility principle
  - $\Rightarrow$ "A class should have one, and only one, reason to change"
- OCP: Open/closed principle
  - $\Rightarrow$ "You should be able to extend a class behavior, without modifying it"
- LSP: Liskov's substitutability principle
  - $\Rightarrow$ "Derived classes must be substitutable for their base classes"
- ISP: Interface segregation principle
  - $\Rightarrow$ "Make fine grained interfaces that are client specific"
- DIP: Dependency inversion principle
  - $\Rightarrow$ "Depend on abstractions, not on concretions"

SRP — Single responsibility principle

"A class should have one, and only one, reason to change"

- evitare di costruire classi che gestiscono responsabilità diverse
- altrimenti vi saranno più occasioni di doverle modificare
- le modifiche rischiano di essere "poco incapsulate", andando a toccare uno o più delle varie responsabilità
- meglio costruire più classi piccole invece che singole "enormi" classi (God class), ma senza nemmeno esagerare
- SRP si applica in realtà a metodi, classi e package, a granularità diverse

Esempi visti

- MVC: divide in parti diverse le responsabilità M,V,C
- Strategy pattern: sposta una strategia fuori dalla classe che la usa (e similmente, Iterator e Observer..)
- Esempio negativo: `Stream` (è una API con filosofia "rich interface"..)

OCP — Open/closed principle

"You should be able to extend a class behavior, without modifying it"

- costruita/testata una classe, deve essere facile aggiungere funzionalità per estensione, ma non andrebbe più modificata
- costruita una classe, la si deve cambiare solo per risolvere dei bug
- consente di limitare il bisogno di intaccare codice già testato

$\Rightarrow$ aderire allo schema Interfaccia - Classi Astratte - Classi Concrete

- $\Rightarrow$ classi importanti siano "nascoste" da interfacce
- $\Rightarrow$ astrarre dai nomi concreti delle classi per consentire subclassing
- $\Rightarrow$ uso di patterns quali T. Method e Strategy per gestire punti "caldi"

Esempi visti

- Gerarchia collections (`Set`, `AbstractSet`, `HashSet` vs `TreeSet`)
  - I clienti si riferiscono perloppiù a `Set`, senza dipendenze sul codice
  - `AbstractSet` consolida codice riusabile fra le varie implementazioni
  - `HashSet` e `TreeSet` specializzano opportunamente `AbstractSet`

LSP — Liskov's substitutability principle

"Objects of derived classes must be semantically substitutable for their base version"

- la classe che estende/implementa una classe/interfaccia deve ottemperare anche al suo contratto di comportamento informalmente specificato
- (javac si occupa solo della parte di "interfaccia" del contratto)
- questo evita di generare malfunzionamenti quando si passa da comportamenti a loro presunte specializzazioni

$\Rightarrow$ regole da seguire

- $\Rightarrow$ leggere/scrivere documentazione di interfacce e classi
- $\Rightarrow$ non lanciare eccezioni (unchecked) inaspettate
- $\Rightarrow$ non violare regole supposte valide dalla classe/interfaccia base
- $\Rightarrow$ non portare a side-effect inaspettati
- $\Rightarrow$ non stravolgere il funzionamento logico di un metodo in caso di override

Esempi visti

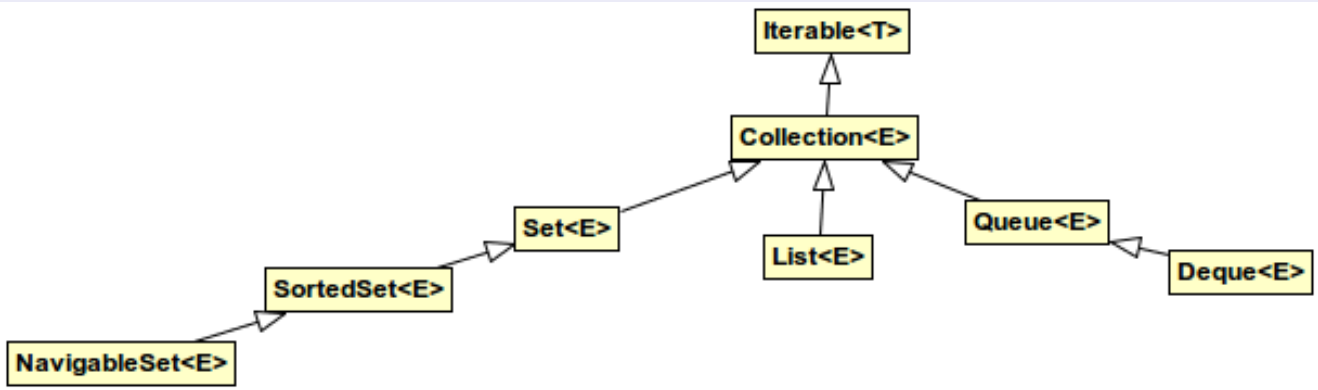
- `Iterator.next` deve lanciare una eccezione oltre i limiti
- `Object.equals` richiede vari controlli, ad esempio quello sul `null`
- `Collection.addAll` non dovrebbe modificare l'argomento

ISP — Interface segregation principle

"Make fine grained interfaces that are client specific"

- le classi non vanno forzate ad avere (o dipendere da) metodi che non servono
- ove possibile, si costruiscano interfacce specifiche per le richieste di ogni tipologia di cliente, con solo i metodi usati
  - $\Rightarrow$ non avere interfacce con molti metodi
  - $\Rightarrow$ ristrutturarle in gerarchie con le tipologie più comunemente richieste
  - $\Rightarrow$ se una classe ha due clienti diversi, forse dovrebbe avere due interfacce

Esempi visti: gerarchia interfacce `Collection`



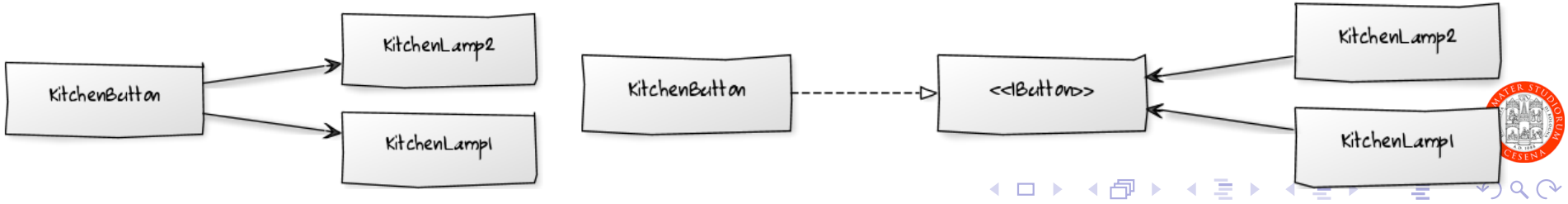
DIP — Dependency Inversion Principle

"Depend on abstractions, not on concretions"

- una variante di (o accoppiamento a) OCP
- una classe non sia dipendenti da un aspetto implementativo estraibile, bensì..
- tale aspetto sia astratto da una interfaccia da cui si dipende
- regole da seguire
  - ⇒ creare una barriera di astrazione fra classi che lavorano a livello diverso o che appartengono a componenti logiche diverse
- limitare al massimo i casi di classi che dipendono tra loro

Esempi visti

- gerarchia `Collection`, `Device` (domotica), `Comparable`,..



Riassunto: SOLID principles in Java

5 principi cardine

- SRP: Single responsibility principle
  - $\Rightarrow$ tenere le classi "piccole" e coerenti
- OPC: Open/closed principle
  - $\Rightarrow$ fattorizzare codice funzionante in classi estendibili (astratte o non)
- LSP: Liskov's substitutability principle
  - $\Rightarrow$ obbedire anche al contratto (sottointeso) di una classe/interfaccia base
- ISP: Interface segregation principle
  - $\Rightarrow$ costruire gerarchie fatte di interfacce "piccole", coerenti e utili
- DIP: Dependency inversion principle
  - $\Rightarrow$ dipendere da interfacce, non dalle loro implementazioni

Linee guida di programmazione efficace

Alcuni elementi

- Sono indicazioni operative su come comportarsi in certe situazioni di programmazione, con lo scopo di ottenere una programmazione più efficace e moderna (maggior prestazioni, flessibilità, leggibilità)
- Sono disgiunte (anche se non del tutto) da altri aspetti correlati: convenzioni sul codice, principi, pattern, regole di "clean coding"
- Spunto tratto da: J.Block "Effective Java"

In questa lezione

- Illustreremo varie linee guida
- Ne approfondiremo qualcuna
- Da usare ove possibile all'esame
- Da usare in futuro

Effective Java, 2nd edition

Struttura del testo

- 78 item, ossia 78 suggerimenti, numerati
- divisi in 10 capitoli (creazione/distruzione oggetti, metodi comuni a tutti gli oggetti, classi/interfacce, generici, enum/annotazioni, metodi, programmazione, eccezioni, concorrenza, serializzazione)
- tutti condivisibili, ne approfondiremo un sottoinsieme

Ogni linea guida descrive

- la situazione(i) in cui si usa
- cosa è opportuno fare in quei casi
- motivazioni e vantaggi/svantaggi

La parte di catalogo che ci concerne (1/3)

Creating and destroying objects

- Consider static factory methods instead of constructors
- Consider a builder when faced with many constructor parameters
- Enforce the singleton property with a private constructor or an enum type
- Enforce noninstantiability with a private constructor

Methods common to all objects

- Obey the general contract when overriding `equals`
- Always override `hashCode` when you override `equals`
- Always override `toString`

Classes/interfaces

- Minimize the accessibility of classes and members
- In public classes, use accessor methods, not public fields
- Minimize mutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it
- Prefer interfaces to abstract classes

La parte di catalogo che ci concerne (2/3)

Generics

- Don't use raw types in new code
- Eliminate unchecked warnings
- Prefer lists to arrays

Enum and annotations

- Use enums instead of int constants
- Use `EnumMap` instead of ordinal indexing
- Consistently use the `Override` annotation

Methods

- Check parameters for validity
- Make defensive copies when needed
- Design method signatures carefully
- Write doc comments for all exposed API elements

La parte di catalogo che ci concerne (3/3)

General Programming

- Minimize the scope of local variables
- Prefer for-each loops to traditional for loops
- Know and use the libraries
- Avoid float and double if exact answers are required
- Prefer primitive types to boxed primitives
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Adhere to generally accepted naming conventions

Exceptions

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
- Avoid unnecessary use of checked exceptions
- Favor the use of standard exceptions
- Throw exceptions appropriate to the abstraction
- Document all exceptions thrown by each method
- Include failure-capture information in detail messages

Outline

Creating and destroying objects

Creating and destroying objects

- Consider static factory methods instead of constructors
- Consider a builder when faced with many constructor parameters
- Enforce the singleton property with a private constructor or an enum type
- Enforce noninstantiability with a private constructor

Static factory, confronto con l'uso di costruttori

+ possiamo dargli un nome espressivo
+ possiamo avere più costruzioni diverse a parità di signature
+ possiamo istanziare una specializzazione
– problematico con strutture di ereditarietà
– difficile distinguere i factory dagli altri metodi

Consider a builder when faced with many constructor parameters

- Quando la costruzione è complessa, e favorisce un approccio step-by-step

Creating and destroying objects: sui singleton

Creating and destroying objects

- Consider static factory methods instead of constructors
- Consider a builder when faced with many constructor parameters
- Enforce the singleton property with a private constructor or an enum type
- Enforce noninstantiability with a private constructor

Enforce the singleton property with a private constructor or an enum type

- Come sappiamo i singleton abbiano costruttore privato
- Block suggerisce l'uso di una `enum` a singolo valore `SINGLETON` per realizzare un singleton
  - Gestisce bene serializzazione e thread-safety
  - Forza una visione, peraltro ragionevole, di una `enum` come collezione di un insieme finito di oggetti (nella fattispecie, 1)

Singleton con enum

```
1  public enum Log {
2
3      SINGLETON;
4
5      // Codice del singleton vero e proprio
6
7
8      private int counter = 0;
9
10     public void add(String s){
11         System.err.println((this.counter++)+" "+s);
12     }
13
14     // Un main di prova
15
16     public static void main(String[] args){
17         Log.SINGLETON.add("errore..");
18         Log.SINGLETON.add("warning..");
19         Log.SINGLETON.add("uscita..");
20     }
21 }
```

Creating and destroying objects: costruttori privati

Creating and destroying objects

- Consider static factory methods instead of constructors
- Consider a builder when faced with many constructor parameters
- Enforce the singleton property with a private constructor or an enum type
- Enforce noninstantiability with a private constructor

Enforce noninstantiability with a private constructor

- Ricordarsi di nascondere i costruttori in classi "helper"
- Vedi classi `java.lang.Math`, `java.util.Collections`

Esempio

```
1 public class Collections {
2     // Suppresses default constructor, ensuring non-instantiability.
3     private Collections() {
4     }
5     ..
6 }
```

Methods common to all objects

Methods common to all objects

- Obey the general contract when overriding `equals`
- Always override `hashCode` when you override `equals`
- Always override `toString`

Alcuni fatti riepilogativi di quanto già visto in parte (oop10)

- `equals` è usato in tutto il Collection framework (`Collection.contains/remove`, `Map.get`)
- `hashCode` è usato in `HashSet` e `HashMap`
- è opportuno definirli (bene!) in coppia e farli lavorare sullo stesso set di campi, perché è facile cambiare collezione così che serva l'hashcode
- è opportuno ri-definire *sempre* `toString`, anche a fini di debug, esponendo le info pubbliche rilevanti
- l'uso (informato) di Eclipse è molto utile
- attenzione alla semantica intesa quando servono implementazioni ad-hoc (`java.lang.Object`)

Classes/interfaces

Classes/interfaces

- Minimize the accessibility of classes and members
- In public classes, use accessor methods, not public fields
- Minimize mutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it
- Prefer interfaces to abstract classes

..prossime slide

Classi e interfacce: minimize accessibility

Minimize the accessibility of classes and members: filosofia

Per il principio dell'information-hiding, e per evitare di influenzare molte classi con delle modifiche, si renda ogni proprietà (classe, metodo, campo) visibile il meno possibile

Fatti principali

- Classi ad uso "interno" di una libreria o applicazione siano package-private, ossia non le si dichiari pubbliche
- Classi usate solo da un'altra classe `C`, vengano innestate private in `C` (oop12) – senza eccessi
- Limitare visibilità di campi/metodi (`private`, `protected`, `public`)
- L'uso del livello package-private è avanzato, da usare documentando
- L'esposizione "pubblica" di una proprietà è un fatto "rilevante"

Classi e interfacce: don't use public fields

In public classes, use accessor methods, not public fields: filosofia

La struttura di un oggetto è un aspetto implementativo, e come tale suscettibile di modifiche successive che non devono toccare i clienti

Fatti principali

- L'esistenza di un getter o setter non implica l'esistenza di un campo
  - Una classe che rappresenta uno slot temporale in una agenda appuntamenti avrebbe getter e setter per ora e minuto di inizio dello slot, ma quale campo usereste?
- Alcuni ritengono che in classi non pubbliche sia accettabile avere campi non privati, per avere codice più compatto, ma questo pone comunque vincoli a modifiche successive
  - È sconsigliato, ma non è un errore
- Anche i campi `protected` sono sconsigliati

Classi e interfacce: minimize mutability

Minimize mutability: filosofia

Cercare di rendere meno mutabile possibile il codice: ciò rende il codice più semplice, evita errori e interferenze fra il lavoro dei clienti

Fatti principali

- Campi e variabili siano finali se non serve esplicitamente il contrario
- Metodi e classi siano finali se si vuole prevenire la loro specializzazione
- Classi immutabili hanno campi finali che sono a loro volta immutabili
- Invece di metodi "mutators", si usi l'approccio funzionale
  - il metodo torni un nuovo oggetto, senza modificare `this`
- Esempi: `String`, `Integer`, `Date`.. perché?
- Vantaggi:
  - classi più semplici
  - classi thread-safe
  - oggetti facilmente "sharable" (p.e.: costanti)
  - ottimizzazioni (flighweight, caching)

Un esempio di design avanzato di una API: Complex

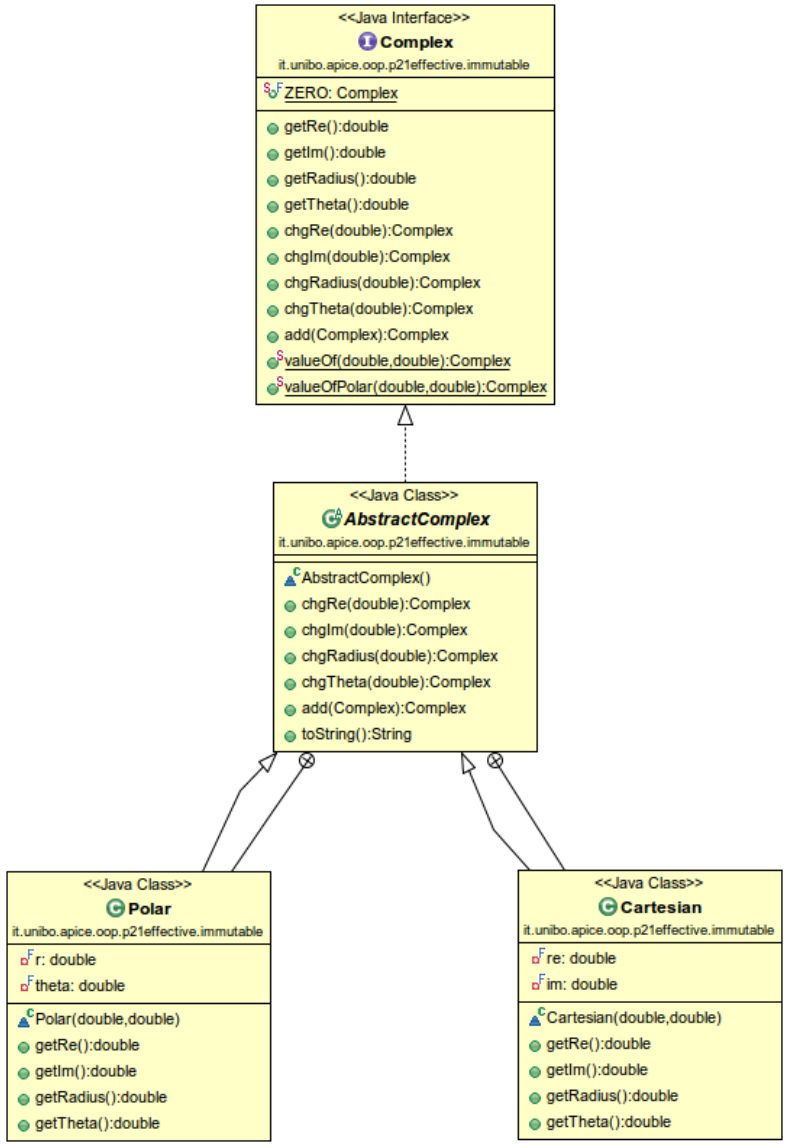
Astrazione numero Complesso: funzionalità

- costruibile sia in coordinate polari che cartesiane
- astrazione ovviamente immutabile
- metodi per cambiare coordinate (senza mutabilità)
- approccio completamente "funzionale"

Dettagli interni

- interfaccia senza metodi default
- interfaccia con static factory
- costruzione oggetti con riuso della costante "zero"
- classe astratta per fattorizzare codice comune
- codice implementativo package-protected
- implementazaione cartesiana-polare scelta "dinamicamente"

Complex: UML completo



Interfaccia Complex: 1/2

```
1  public interface Complex {
2
3      // costante pubblica senza rischio di modifica da fuori
4      public static final Complex ZERO = new AbstractComplex.Cartesian(0, 0);
5
6      double getRe();
7
8      double getIm();
9
10     double getRadius();
11
12     double getTheta();
13
14     Complex chgRe(double re);
15
16     Complex chgIm(double im);
17
18     Complex chgRadius(double r);
19
20     Complex chgTheta(double t);
21
22     Complex add(Complex c);
```

Interfaccia Complex: 2/2

```
1     public static Complex valueOf(final double re, final double im) {
2         if (re == 0.0 && im == 0.0) {
3             return ZERO;
4         }
5         return new AbstractComplex.Cartesian(re, im);
6     }
7
8     public static Complex valueOfPolar(double r, double theta) {
9         if (r == 0.0) {
10            return ZERO;
11        }
12        return new AbstractComplex.Polar(r, theta % (2 * Math.PI));
13    }
14 }
```

AbstractComplex

```
1  abstract class AbstractComplex implements Complex {
2      // Metodi con approccio funzionale
3      @Override
4      public Complex chgRe(final double re) {
5          return new Cartesian(re, this.getIm());
6      }
7
8      @Override
9      public Complex chgIm(final double im) {
10         return new Cartesian(this.getRe(), im);
11     }
12
13     @Override
14     public Complex chgRadius(double r) {
15         return new Polar(r, this.getTheta());
16     }
17
18     @Override
19     public Complex chgTheta(double t) {
20         return new Polar(this.getRadius(), t);
21     }
22
23     @Override
24     public Complex add(final Complex toAdd) {
25         return new Cartesian(this.getRe() + toAdd.getRe(), this.getIm()
26                 + toAdd.getIm());
27     }
28
29     // Si potrebbe fare caching del toString
30     @Override
31     public String toString() {
32         return "[" + getRe() + "," + getIm() + "]";
33     }
```

AbstractComplex.Cartesian

```
1   static final class Cartesian extends AbstractComplex {
2
3       private final double re;
4       private final double im;
5
6       Cartesian(final double re, final double im) {
7           this.re = re;
8           this.im = im;
9       }
10
11      @Override
12      public double getRe() {
13          return re;
14      }
15
16      @Override
17      public double getIm() {
18          return im;
19      }
20
21      @Override
22      public double getRadius() {
23          return Math.sqrt(this.re * this.re + this.im * this.im);
24      }
25
26      @Override
27      public double getTheta() {
28          return Math.atan2(this.im, this.re);
29      }
30  }
```

AbstractComplex.Polar

```
1   static final class Polar extends AbstractComplex {
2
3       private final double r;
4       private final double theta;
5
6       Polar(final double r, final double theta) {
7           if (r < 0.0) {
8               throw new IllegalArgumentException();
9           }
10          this.r = r;
11          this.theta = theta;
12      }
13
14      @Override
15      public double getRe() {
16          return this.getRadius() * Math.cos(this.getTheta());
17      }
18
19      @Override
20      public double getIm() {
21          return this.getRadius() * Math.sin(this.getTheta());
22      }
23
24      @Override
25      public double getRadius() {
26          return this.r;
27      }
28
29      @Override
30      public double getTheta() {
31          return this.theta;
32      }
33  }
```

UseComplex

```
1  public class UseComplex {
2
3      public static void main(String[] args){
4          final List<Complex> l = Arrays.asList(
5                  Complex.valueOf(10.0, 20.0),
6                  Complex.valueOf(-10.0, 20.0),
7                  Complex.valueOfPolar(10.0, 0),
8                  Complex.valueOfPolar(10.0, Math.PI/4),
9                  Complex.valueOfPolar(10.0, Math.PI),
10                 Complex.valueOfPolar(10.0, 3*Math.PI/4),
11                 Complex.ZERO
12         );
13
14         System.out.println(l.stream()
15                            .map(Complex::toString)
16                            .collect(Collectors.joining(";", "{", "}"))
17         );
18
19         System.out.println(l.stream()
20                            .map(c->c.chgRadius(1.0))
21                            .map(c->c.add(Complex.valueOf(10.0,10.0)))
22                            .map(Complex::toString)
23                            .collect(Collectors.joining(";", "{", "}"))
24         );
25     }
26 }
```

Classi e interfacce: favour composition over inheritance

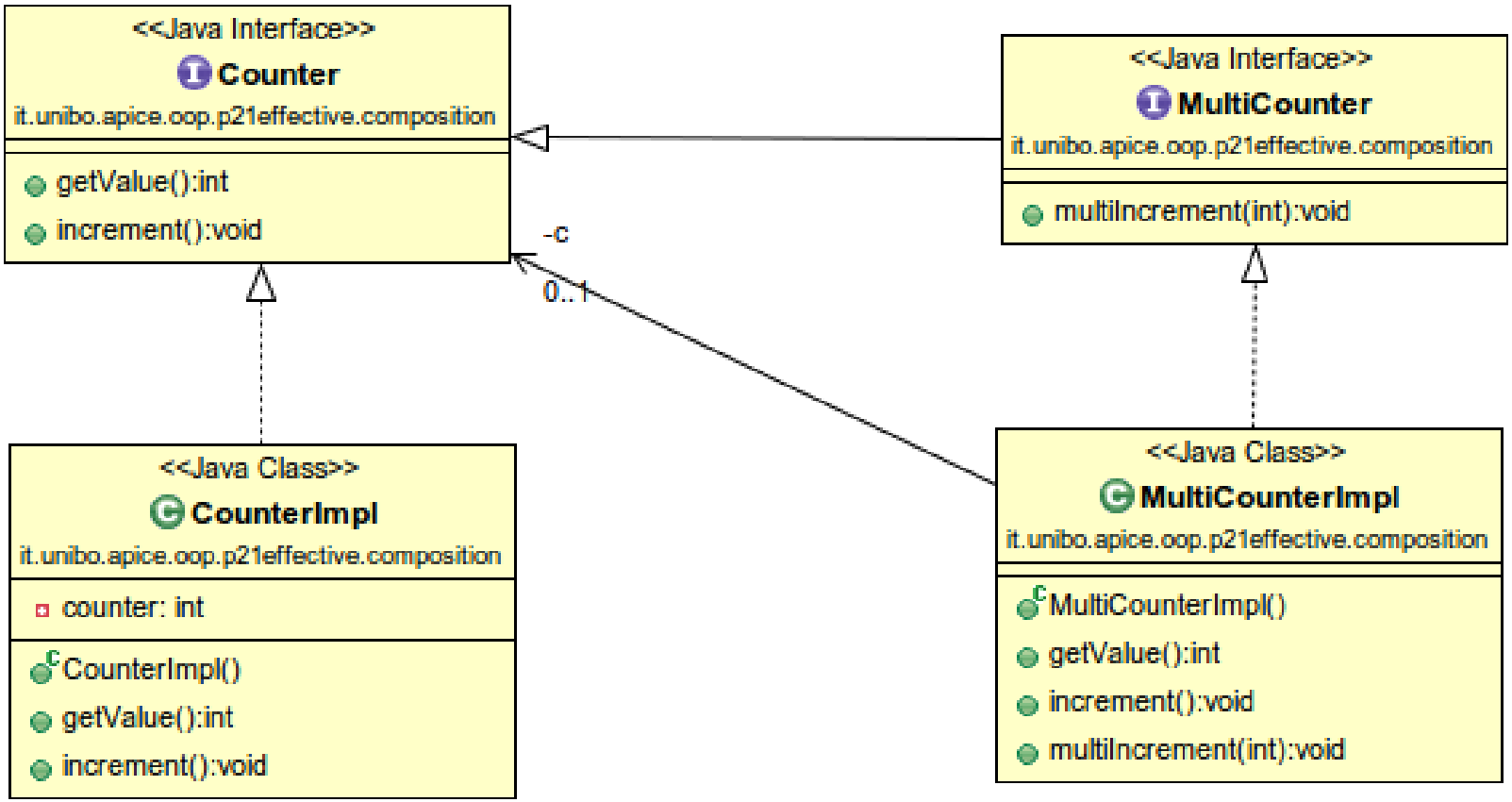
Favour composition over inheritance: filosofia

In molti contesti l'ereditarietà (fra classi) non è la migliore strategia di riuso, meglio usare delegazione, ossia con una decorazione

Fatti principali

- L'ereditarietà funziona bene solo se il programmatore ha il controllo di entrambe le classi
- Non c'è vero incapsulamento fra una classe e la sua sotto-classe
  - modifiche apparentemente innocue nella sopra-classe potrebbero compromettere il funzionamento della sotto-classe
- Spesso si vuole ereditare parte del codice, non tutto
- Spesso l'aspetto is-a della relazione di ereditarietà è secondario
- In molti linguaggi (Java), l'erediterietà è "singola"
- Valutare l'uso di un approccio a decorazione
- Eclipse fornisce anche un modo veloce di creare "delegazione"

L'esempio `Counter`: UML completo



Counter e MultiCounter

```java
public interface Counter {

    int getValue();

    void increment();
}
```

```java
public class CounterImpl implements Counter {

    private int counter = 0;

    @Override
    public int getValue() {
        return this.counter;
    }

    @Override
    public void increment() {
        counter++;
    }
}
```

```java
public interface MultiCounter extends Counter {

    void multiIncrement(int n);
}
```

Implementazione `MultiCounterImpl` via composizione

```
1  public class MultiCounterImpl{
2
3      private final Counter c = new CounterImpl();
4
5      private int getValue() {
6          return this.c.getValue();
7      }
8
9      public void inc() {
10         this.c.increment();
11     }
12
13     public void multiIncrement(int n) {
14         for (int i=0; i<n ; i++){
15             this.c.increment();
16         }
17     }
18 }
```

Classi e interfacce: Prefer interfaces to abstract classes

Prefer interfaces to abstract classes: filosofia

Quando va creato un tipo di dato relativamente ad un contratto è molto meglio usare una `interface` piuttosto che una classe astratta (con tutti metodi astratti).

Fatti principali

- È molto più facile implementare una interfaccia che estendere una classe, grazie alla possibilità di ereditarietà multipla
- In Java il pattern adapter si realizza facilmente per adattarsi ad una interfaccia (oop20)
  - Es.: è meglio adattarsi ad un `Runnable` o ad un `Thread`?
- I `default` di Java 8 rendono le interfacce più simili alle classi astratte, e quindi allargano la possibilità di usare le interfacce

Outline

Generics

Generics

- Don't use raw types in new code
- non portare a side-effect inaspettatiEliminate unchecked warnings
- Prefer lists to arrays

..prossime slide

Generics: Don't use raw types in new code

Don't use raw types in new code: filosofia

È possibile usare una classe generica come se non lo fosse, ma è sconsigliato in codice "nuovo": è solo tollerato in codice versione pre-Java 5.

Fatti principali

- È possibile scrivere: `List list = new ArrayList();`
- Più o meno è come lavorare con `List<Object>`.. non si hanno i vantaggi della genericità
- Il compilatore segnala un warning
- Lavorare con i raw type è severamente sconsigliato

Generics: Eliminate unchecked warnings

Eliminate unchecked warnings: filosofia

Non si devono tollerare i warning, perché ci si abitua ad averne, e quindi ad ignorare i warning segnalati dal compilatore.

Fatti principali

- Esempi: uso di raw type, conversioni di tipo con generici, etc.. (oop9)
- A volte non si possono evitare, in qual caso usiamo l'annotazione `SuppressWarning` (oop9)
- Se si possono evitare, evitiamoli
- Discorso analogo per ogni altro warning segnalato dal compilatore

Generics: Prefer lists to arrays

Prefer lists to arrays: filosofia

Gli array sono una specie di tipo generico, ma per motivi di legacy non si integrano molto bene con tutto il resto della genericità. Quindi, è in generale meglio usare il tipo generico più simile: le `java.util.List`

Fatti principali

- Gli array sono trattati "erroneamente" come covarianti (oop9,p58)
- Non si integrano bene: `List<Integer[]>` ok, `List<Integer>[]` no
- Agli array si può chiedere il tipo preciso (sono reificati), ai generici no
- Ok usare array di tipi primitivi per motivi di performance
- Ok usare array (`Integer[]`) se siamo certi di non incappare nei problemi di cui sopra
- Altrimenti conviene usare le liste, che forniscono anche più operazioni, oppure liste non modificabili (`Collections.unmodifiableList`)

Enum and annotations

Enum and annotations

- Use enums instead of int constants
- Consistently use the `Override` annotation
- Use `EnumMap` instead of ordinal indexing

Ricordiamo:

- Le enumerazioni andrebbero usate sempre quando una classe ha un numero finito e noto di oggetti, ossia per modellare insiemi finiti (e noti) di elementi
- J.Block addirittura suggerisce di usare le Enum anche al posto dei booleani (e addirittura per il Singleton)
- Una struttura frequentemente utile è la mappa da enumerazioni a valori, usare l'implementazione ottimizzata `EnumMap`

UseEnumMap

```java
public class UseEnumMap {

    static enum WeekDay {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
        FRIDAY, SATURDAY, SUNDAY
    }

    public static void main(String[] args) {

        Map<WeekDay,List<String>> lessons = new EnumMap<>(WeekDay.class);
        lessons.put(WeekDay.TUESDAY,Arrays.asList("LAB"));
        lessons.put(WeekDay.WEDNESDAY,Arrays.asList("OOP","FIS"));
        lessons.put(WeekDay.THURSDAY,Arrays.asList("SISOP","FIS"));
        lessons.put(WeekDay.FRIDAY,Arrays.asList("SISOP","OOP"));

        // In quali giorni ho OOP?
        System.out.println(
                lessons.entrySet()
                        .stream()
                        .filter(e->e.getValue().contains("OOP"))
                        .map(e->e.getKey())
                        .map(Enum::name)
                        .collect(Collectors.joining(",","(",")"))
        );

    }

}
```

Methods

Methods

- Make defensive copies when needed
- Check parameters for validity
- Design method signatures carefully
- Write doc comments for all exposed API elements

Getter di campi

- Se un campo ha un getter ma non un setter, bisogna accertarsi che modificando l'oggetto tornato dal getter non si modifichi il campo
- Se c'è questo pericolo (gli oggetti dei campi non sono immutabili) bisogna fare copie difensive
- Si clona l'oggetto e si restituisce la versione clonata
- Con le collezioni si può usare una versione non modificabile – nelle API di Java sono realizzate mediante "composizione" (decorator)

MyCollection

```java
public class MyCollection<X> {

    private final Set<X> set = new HashSet<>();

    public MyCollection<X> add(X x){
        set.add(x);
        return this;
    }

    public Set<X> getAll(){
        // return new HashSet<>(this.set)
        return Collections.unmodifiableSet(this.set);
    }

    public static void main(String[] args) {
        final MyCollection<Integer> myc = new MyCollection<>();
        myc.add(1).add(2).add(10);
        final Set<Integer> myc2 = myc.getAll();
        //myc2.add(11);
    }

}
```

Methods: Check parameters for validity

Check parameters for validity: filosofia

Non sempre si può assumere che un cliente sappia i vincoli che gli input di un metodo debbano soddisfare, quindi è opportuno che un metodo (specialmente se pubblico) effettui i controlli necessari, lanciando eccezioni se necessario.

Fatti principali

- Aggiungere i controlli necessari spesso porta a penalizzazione in performance, ed è noioso
- Nei metodi pubblici è però fondamentale
- Si lancino eccezioni "unchecked", specialmente se l'errore è dovuto ad una errata lettura della documentazione
- Si lancino eccezioni "checked" se invece l'errore non era facilmente intercettabile dal cliente

Outline

General Programming

- Minimize the scope of local variables
- Prefer for-each loops to traditional for loops
- Know and use the libraries
- Avoid float and double if exact answers are required
- Prefer primitive types to boxed primitives
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Adhere to generally accepted naming conventions

Discussione

- Definire una variabile quando serve, non ad inizio metodo!
  - minimizza errori e aumenta la leggibilità
- La dichiaratività rende il codice più comprensibile e chiaro, quindi:
  - meglio metodo `forEach`, poi costrutto for-each, poi il `for` classico

General programming: Know and use the libraries

Know and use the libraries: filosofia

Le librerie fornite incapsulano funzionalità note a tutti e ben funzionanti. Usarle è molto meglio che riscriversi pezzi di codice da soli.

Fatti principali

- Il codice che usa librerie è di migliore qualità: più leggibile, comprensibile, corretto, performante.
- Conoscere le librerie evita di "reinventare la ruota"
- Librerie importanti (in `java.lang`, `java.util`, `java.util.stream`, `java.io` e `javax.swing`), anche all'esame:
  - `Math`, `Collections`, `Arrays`, `Collection`, `Iterator`, `List`, `Set`, `Object`, `Integer`, `InputStream`, `JFrame`, `JPanel`..
- ..ma non cercate librerie per ogni cosa vi serva: assicuratevi di non diventarne dipendenti al punto di non saper più realizzare niente da soli

General Programming

General Programming

- Minimize the scope of local variables
- Prefer for-each loops to traditional for loops
- Know and use the libraries
- Avoid float and double if exact answers are required
- Prefer primitive types to boxed primitives
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Adhere to generally accepted naming conventions

Discussione

- Ricordarsi i problemi di precisione dei float e double
- I primitive sono più veloci e espressivi, i boxed servono per la visione "everything is an object"
- Non rappresentate come stringhe valori che modellano concetti diversi
- La creazione di stringhe complesse avvenga con `StringBuffer`
- Seguite le convenzioni Java.. sempre!

General programming: Refer to objects by their interfaces

Refer to objects by their interfaces: filosofia

I clienti di un oggetto percepiscono solo la sua interfaccia. Quindi, ove disponibile, è sempre meglio indicare tipi `interface`: si ha una visione più astratta, meno vincolata all'implementazione, più riusabile, e più aperta ai cambiamenti (si segue il principio DIP)

Fatti principali

- Nelle proprie gerarchie di oggetti, si cerchino sempre di costruire le interfacce più significative, che denotano i tipi fondamentali di oggetti. (ISP)
- Usare interfacce in parametri, tipi di ritorno e variabili/campi

Exceptions

Exceptions

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
- Avoid unnecessary use of checked exceptions
- Favor the use of standard exceptions
- Throw exceptions appropriate to the abstraction
- Document all exceptions thrown by each method
- Include failure-capture information in detail messages

Exceptions

Discussione

- Usare appropriatamente le eccezioni unchecked `IllegalArgumentException`, `IllegalStateException`, `NullPointerException`, `IndexOutOfBoundsException`, `UnsupportedOperationException`
- Lanciare eccezioni compatibili col livello d'astrazione della classe, catturando quelle di basso livello e rilanciando quelle di alto livello (vedi `AbstractSequentialList.get`)
- Documentare le eccezioni, dichiarare il `throws` solo per le checked

Conclusione

Costruire del buon codice è un mix di vari aspetti

- Uso delle convenzioni sulla formattazione del codice
- Uso di tecniche di programmazione efficace
- Uso di plugin per check di stile o di bug
- Uso di pattern di progettazione
- Uso efficiente delle risorse disponibili

$\Rightarrow$ Il tutto richiede esperienza

Dall'OO alla costruzione di software

- Quanto visto nel corso ha validità generale per l'Ingegneria Informatica
- L'OO è lo stile più usato
- Ma le tecniche ed esperienze viste sono cruciali in ogni contesto di progettazione/costruzione di sistemi informatici