UNIVERSITÀ DELLA CALABRIA
il Campus per eccellenza

Domain primitives

Mario Alviano

Exercise

Implementare la classe Veicolo contenente:
- String targa
- String casaProduttrice
- String nome
- Double prezzo

Implementare inoltre Costruttore, get/set e toString.

Implementare due classi: Automobile e Moto,
entrambe estendono da Veicolo e ridefiniscono
il metodo getPrezzo() come segue:
- Per le automobili con un prezzo inferiore ai 10000 euro, il prezzo è ridotto del 5%
- Per le automobili con un prezzo inferiore ai 20000 euro, il prezzo è ridotto del 10%
- Per le moto con un prezzo inferiore ai 7000 euro, il prezzo è ridotto del 3%
- Per le moto con un prezzo inferiore ai 15000 euro, il prezzo è ridotto del 7.5%
- Negli altri casi il prezzo non è ridotto

Implementare una classe Concessionaria contenente
un ArrayList<Veicolo> e i seguenti metodi:
- void aggiungiVeicolo(Veicolo v)
- void rimuoviVeicolo(Veicolo v)
- void stampaVeicoli()
- void ordinaVeicoliPerPrezzo(), che ordina i veicoli in ordine crescente di prezzo
- void ordinaVeicoliPerNome(), che ordina i veicoli in ordine crescente di nome
- void calcolaValore(), che si occupa di sommare il prezzo di tutti i veicoli stampati
- void salvaSuFile(String filename), che si occupa di salvare su un file tutti i veicoli, ogni veicolo su una linea diversa, seguendo il formato “targa;casaProduttrice;nome;prezzo”
- void leggiDaFile(String filename), che legge i veicoli contenuti nel file e li aggiunge all’ArrayList

Creare infine un main di prova per testare la Concessionaria.

Commentiamo la soluzione “base”.
Ci sono problemi nel codice? E nelle richieste del committente?
Quali sono le entità e i valori oggetto di questo dominio? Aggregati?

What are primitive domains

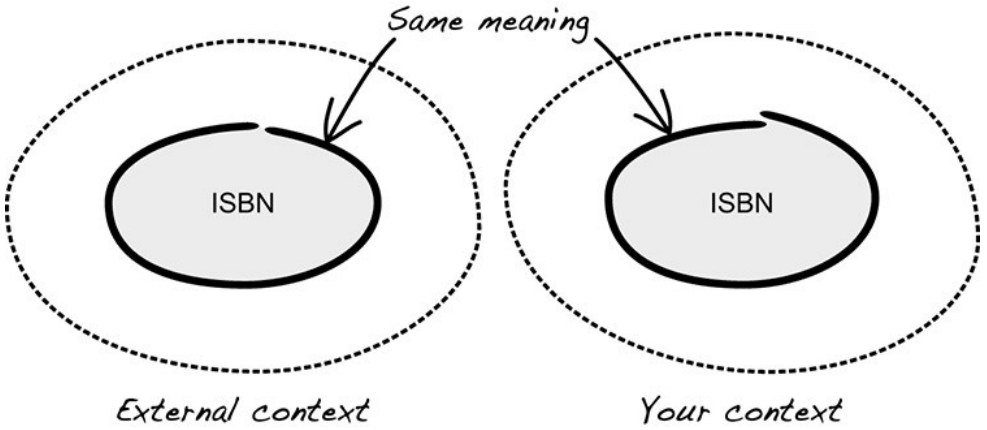
- Value objects of the domain
- Invariants checked on creation
- An object exists if and only if it is valid
- They don’t use generic types and values, like null
- Every concept of the domain must be well represented and encapsulated

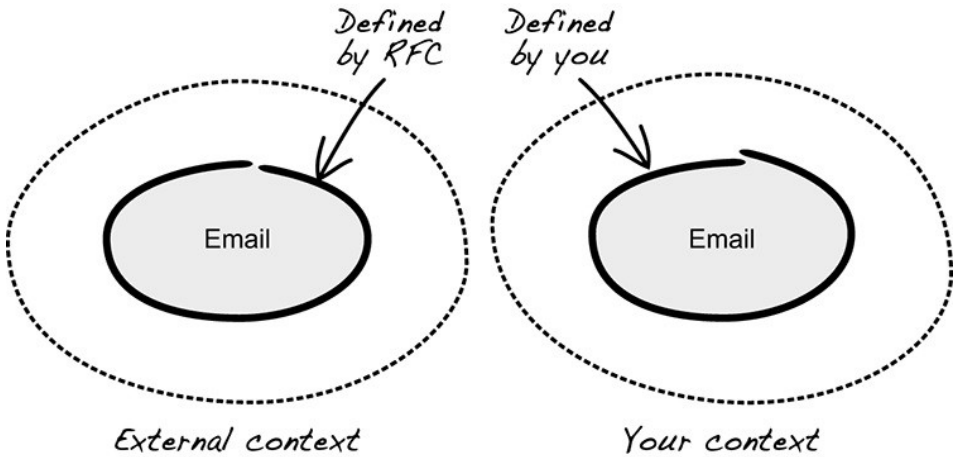
```java
import static
org.apache.commons.lang3.Validate.inclusiveBetween;
import static org.apache.commons.lang3.Validate.notNull;

public final class Quantity {

   private final int value;    ①

   public Quantity(final int value) {
      inclusiveBetween(1, 200, value);    ②
      this.value = value;
   }

   public int value() {
      return value;
   }

   public Quantity add(final Quantity addend) {    ③
      notNull(addend);
      return new Quantity(value + addend.value);
   }

   // equals() hashCode() etc...

}
```

A quantity is defined as an integer between 1 and 200

Not simply an integer

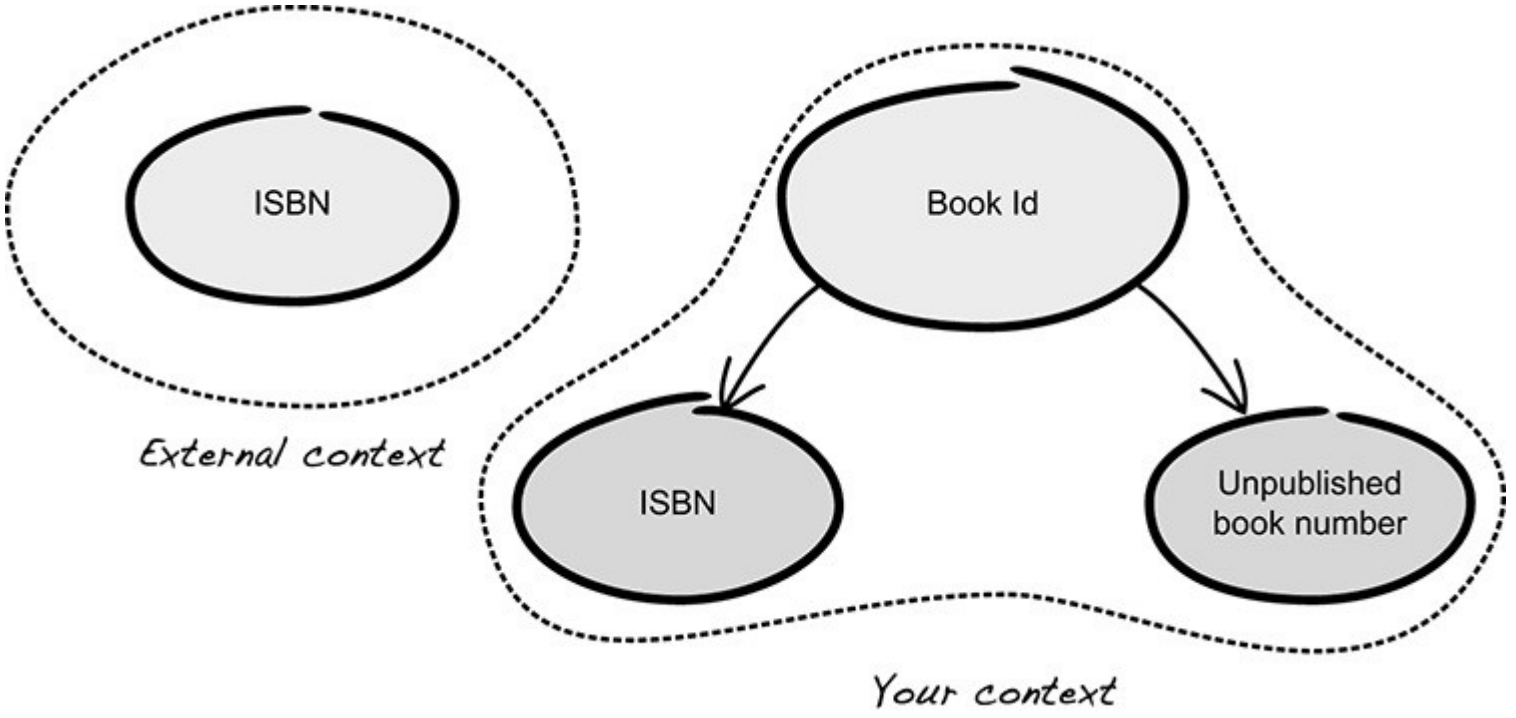
If an instance of Quantity exists, then it satisfies the required conditions





The meaning of a domain primitive is limited by a given context

It may make sense to redefine a concept to better adapt it to your goals
(usually by restricting its definition)



Making a definition more permissive is not recommended (it creates confusion)

Better to introduce a new term and use inheritance

Define your own library of domain primitives

- Define from the beginning domain primitives for all terms
- A method with arguments and returned value represented by domain primiteves is safer
  - Arguments are valid
  - The returned value is valid
- At the end of the day, you are writing less code
  - Validation is done once, in the correct place

Use case: hardening of API

Log repository on internal server

void sendAuditLogsToServerAt(java.net.InetAddress serverAddress);

A method like this does not prevent the log to be sent to an external IP

```
import static org.apache.commons.lang3.Validate.notNull;

void sendAuditLogsToServerAt(InternalAddress serverAddress)
{
   notNull(serverAddress);    ①

   // Retrieve logs and send them to server
}
```

Define a new type
to represent
interal addresses

It is not possible anymore
to inadvertently send logs
to the extern

Avoid to publicly expose the domain

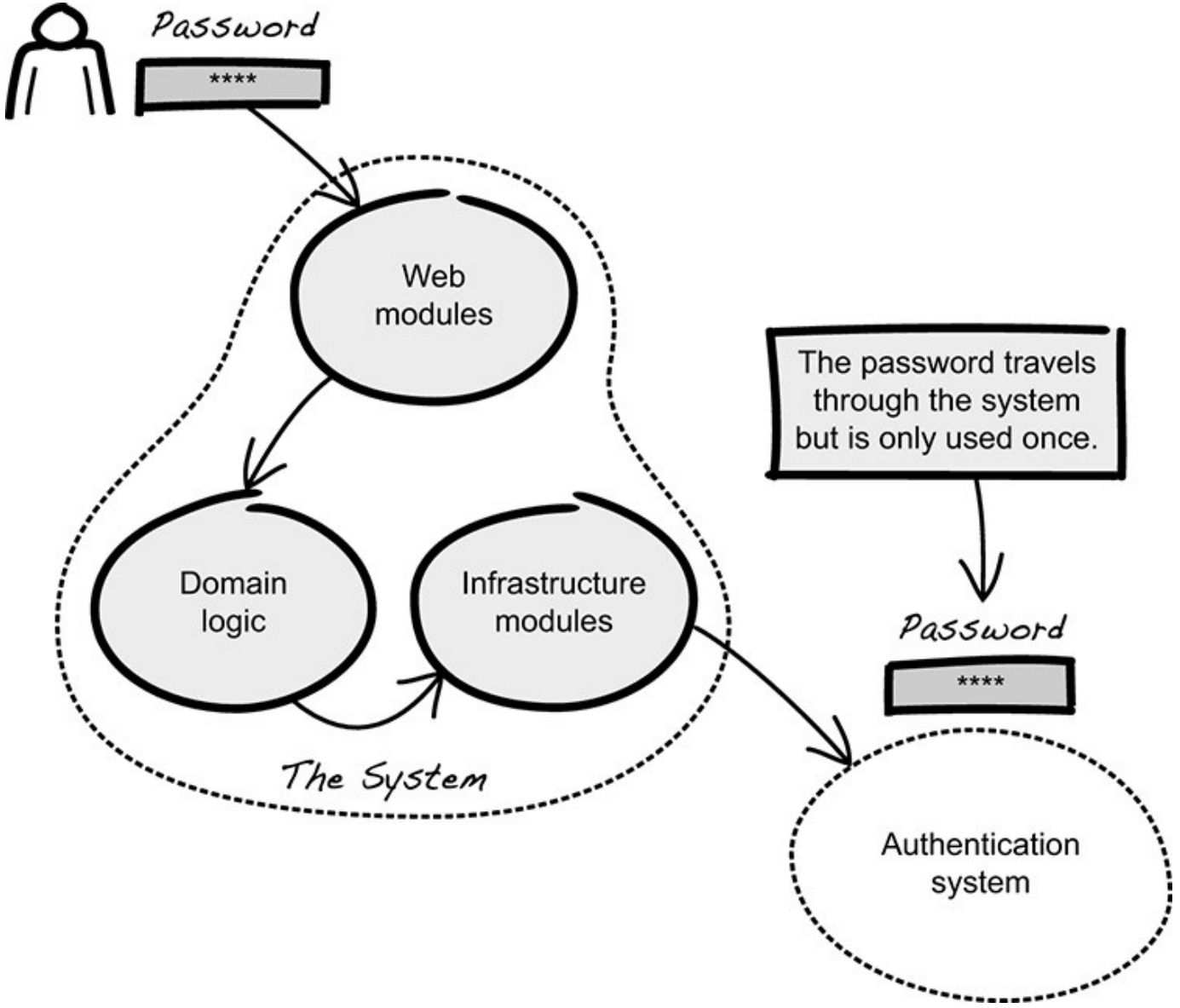
- If you are defining a REST API, avoid to expose your internal model
- Future changes will be expensive
  - All API users have to reflect such changes
- Define ad-hoc objects for transmission
  - Data Transfer Object (DTO)
  - Have their own invariants
  - The internal model can change more or less independently

Read-once objects

- Objects designed to be read only one time
- They identify unexpected usage
- Often primitive domains (but can be also entities or aggregates)
- Inhibit the serialization of sensible data
- Inhibit subclassing and extensions

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.concurrent.atomic.AtomicReference;
import static org.apache.commons.lang3.Validate.notNull;

public final class SensitiveValue                    ①
      implements Externalizable {                    ②

   private transient final
            AtomicReference<String> value;           ③

   public SensitiveValue(final String value) {
     validate(value);                                ④
     this.value = new AtomicReference<>(value);
   }

   public String value() {
      return notNull(value.getAndSet(null),
         "Sensitive value has already been
consumed");                                          ⑤
   }

   @Override
   public String toString() {
      return "SensitiveValue{value=*****}";          ⑥
   }

   @Override
   public void writeExternal(final ObjectOutput out) {
      deny();                                        ⑦
   }

   @Override
   public void readExternal(final ObjectInput in) {
      deny();                                        ⑦
   }

   private static String validate(final String value) {
      // Check domain-specific invariants
      return notBlank(value).trim();
   }

   private static void deny() {
      throw new UnsupportedOperationException(
            "Not allowed on sensitive value");
   }
}
```

1) Inhibit subclassing and extensions
2) Serialize only the identity of the class (action anyhow blocked by 7)
3) Value is transient, that is, it is not serialized – AtomicReference guarantees an atomic assignment of a reference (for multiple threads)
5) Value is read and forgotten
6) Redefine toString() to prevent value to be leaked in some log file



Password (o its hash) traverses several modules of the system

With a read-once object, the authentication system can detect unauthorized reads in preceeding modules

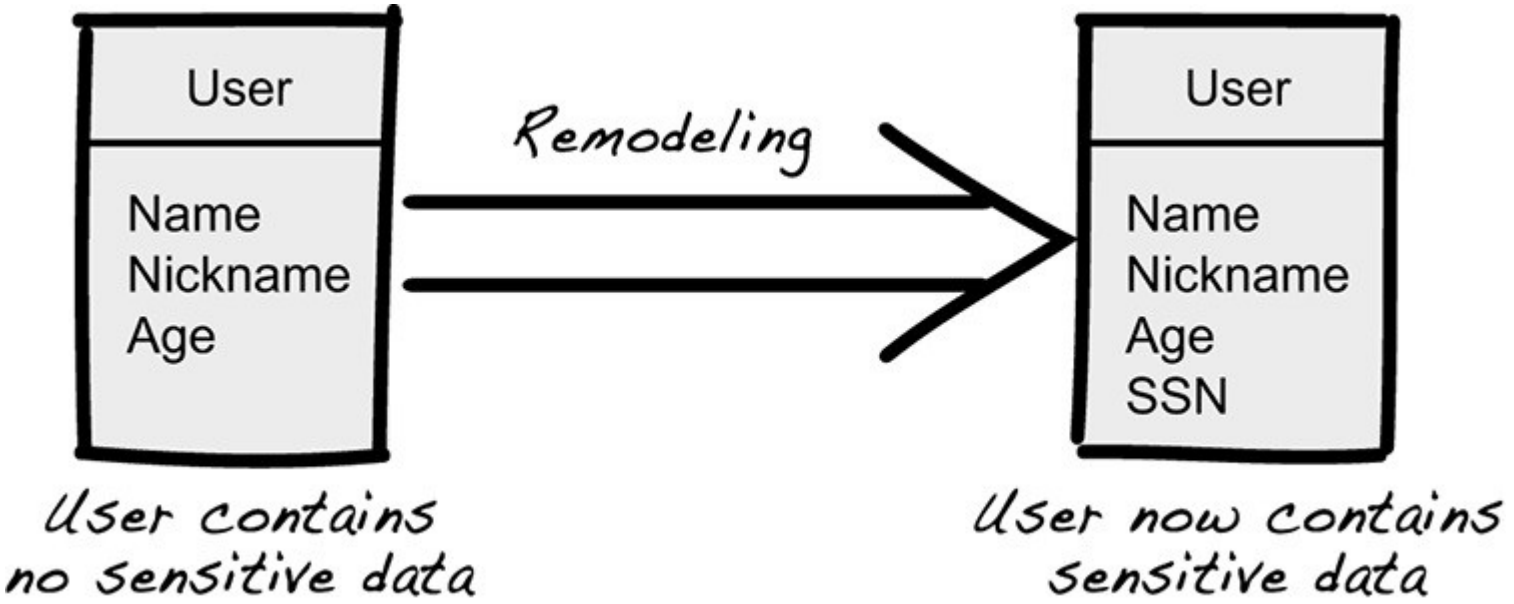
```
import static org.apache.commons.lang3.Validate.validState

public final class Password implements Externalizable {

   private final char[] value;
   private boolean consumed = false;

   public Password(final char[] value) {
      this.value = validate(value).clone();    ①
   }

   public synchronized char[] value() {    ②
      validState(!consumed, "Password value has already
been consumed");
      final char[] returnValue = value.clone();    ③
      Arrays.fill(value, '0');    ④
      consumed = true;    ⑤
      return returnValue;
   }

   @Override
   public String toString() {
      return "Password{value=*****}";
   }

   @Override
   public void writeExternal(final ObjectOutput out) {
      deny();
   }

   @Override
   public void readExternal(final ObjectInput in) {
      deny();
   }

   private static void deny() {
      throw new UnsupportedOperationException(
            "Serialization of passwords is not allowed");
   }

   private static char[] validate(final char[] value) {
      // Validate length, characters and so forth
      return value;
   }
}
```

Note that the password is actually deleted after the read (4)

This guarantees that the password in value is not in memory anymore

Removing the reference is not sufficient

The caller of value() must eliminate the returned array with a similar operation



SSN is modeled as a read-once object

Tests are passed and we go in production

After some time we observe error messages “Not allowed on sensitive value.” when Tomcat is terminated

We discover that Tomcat serializes the session on disk before termination

What would it happen if SSN was not a read-once object?

Primitive domains are the bricks of your system

- Entities represent long-living objects
  - Rooms of an hotel
  - The basket of an online shope
- Functionalities of the system change the state of these objects
  - Rooms in an hotel are booked
  - Products are added to and removed from the basket
- If everything is int or String, validation must be done in entities
  - Entity code grows in uncontrollable ways
  - Several for and if – some path will be missed

```java
import org.apache.commons.lang3.Validate;

class Order {
   private BookRepository bookCatalog;
   private ArrayList<Object> items;
   private boolean paid = false;
   Inventory inventory;

   public void addItem(String isbn, int qty) {
      if(this.paid == false) {    ①
         notNull(isbn);
         isTrue(isbn.length() == 10);    ②
         isTrue(isbn.matches("[0-9X]*"));    ②
         isTrue(isbn.matches("[0-9]{9}[0-9X]"));    ②
         Book book = bookCatalog.findByISBN(isbn);    ③
         if(inventory.avaliableBooks(isbn) >= qty)){    ④
            items.add(new OrderLine(book, qty));    ⑤
         }
      }
   }
   ...
}
```

```java
class ShoppingFlow {

    void handleOrderAdd() {    ⑥
        ...
        String isbnText = ...
        int qty = Integer.parseInt(qtyText);
        ...    ⑦
        order.addItem(isbn, qty);
       ...

    }
}
```

ISBN is validated in addItem()

Other methods that use ISBN must do the same. Be careful to not miss any check!

Did we just forget to validate quantity? Oops!

Remember the types of validation

- Origin
  - Are data coming from a legitimate sender?
- Size
  - Is the size reasonable?
- Lexical content
  - Does it contain only admissible characters?
- Syntax
  - Is the format correct?
- Semantic
  - Do data make sense?

Entities should validate the semantic of data, which depends on the context (known by entities)

All other types of validation should be done by domain primitives

```
import org.apache.commons.lang3.Validate.*;

public class ISBN {
    private final String isbn;

    public ISBN(final String isbn) {    ①
        notNull(isbn);
        isTrue(isbn.length() == 10);
        isTrue(isbn.matches("[0-9X]*"));
        isTrue(isbn.matches("[0-9]{9}[0-9X]"));
        isTrue(checksumValid(isbn));    ②
        this.isbn = isbn;
    }

    private boolean checksumValid(String isbn) {
        // ...
    }
    ...
}
```

```
class Order {

    public void addItem(ISBN isbn, Quantity qty) {    ①
        Validate.notNull(isbn);
        Validate.notNull(qty);

        if(this.paid == false) {    ②
            Book book =
                bookcatalogue.findByISBN(isbn);    ③
            if (inventory.avaliableBooks(isbn)
                .greaterOrEqualTo(qty)) {    ④
                addToItems(new OrderLine(book, qty));
            }
        }
    }

    private void addToItems(OrderLine bookQuantity) {
        ...    ⑤
        items.add(new OrderLine(book, qty));
        ...
    }
```

ISBN and Quantity are domain primitives

The Order entity should only validate the semantics: Does the catalog contain a book with the given ISBN? Is the quantity in the warehouse sufficient?


```
class ShoppingFlow {

    void handleOrderAdd() {
        ...
        String isbn = ...
        int qty = ...
        ...
        order.addItem(new ISBN(isbn),
                      new Quantity(qty));

...
```

Summing up

- Use domain primitives for method arguments, constructor arguments, return types and attributes of entities
  - Input is always valid
  - Validation is consistent
  - Entity code is more succinct (doesn’t have to care about limit cases, formats, etc.)
  - Entity code is more readable (it spokes the language of the domain)
- The overhead is irrelevant with respect to other operations of the system
  - regex evaluation vs database access

Google Form

- https://forms.gle/Zv6VQZRoNCMzeKQM7

Exercise

Implementare la classe Veicolo contenente:
- String targa
- String casaProduttrice
- String nome
- Double prezzo

Implementare inoltre Costruttore, get/set e toString.

Implementare due classi: Automobile e Moto,
entrambe estendono da Veicolo e ridefiniscono
il metodo getPrezzo() come segue:
- Per le automobili con un prezzo inferiore ai 10000 euro, il prezzo è ridotto del 5%
- Per le automobili con un prezzo inferiore ai 20000 euro, il prezzo è ridotto del 10%
- Per le moto con un prezzo inferiore ai 7000 euro, il prezzo è ridotto del 3%
- Per le moto con un prezzo inferiore ai 15000 euro, il prezzo è ridotto del 7.5%
- Negli altri casi il prezzo non è ridotto

Implementare una classe Concessionaria contenente
un ArrayList<Veicolo> e i seguenti metodi:
- void aggiungiVeicolo(Veicolo v)
- void rimuoviVeicolo(Veicolo v)
- void stampaVeicoli()
- void ordinaVeicoliPerPrezzo(), che ordina i veicoli in ordine crescente di prezzo
- void ordinaVeicoliPerNome(), che ordina i veicoli in ordine crescente di nome
- void calcolaValore(), che si occupa di sommare il prezzo di tutti i veicoli stampati
- void salvaSuFile(String filename), che si occupa di salvare su un file tutti i veicoli, ogni veicolo su una linea diversa, seguendo il formato “targa;casaProduttrice;nome;prezzo”
- void leggiDaFile(String filename), che legge i veicoli contenuti nel file e li aggiunge all’ArrayList

Creare infine un main di prova per testare la Concessionaria.

Implementiamo le primitive di dominio e scriviamo codice sicuro!

Fine della lezione