Programmazione a Oggetti
Modulo B

Lezione 18

Dott. Alessandro Roncato

08/04/2013

Riassunto

- `equals` e `==`
- Diagramma oggetti
- Serializzazione

Questa lezione

- Product Trader => Plug-in
- DriverManager

Product Trader

- Come gestire le estensioni in modo componibile.
- Nella progettazione dell'applicazione della segreteria studenti si vuole prevedere la gestione di future modalità di calcolo del punteggio per l'assegnazione delle borse di studio in base all'ente erogatore.

Non O.O.

Un approccio interamente procedurale alla soluzione è quello che prevede di gestire il calcolo del punteggio in dipendenza dell'ente erogatore utilizzando switch o if/then/else in cascata.

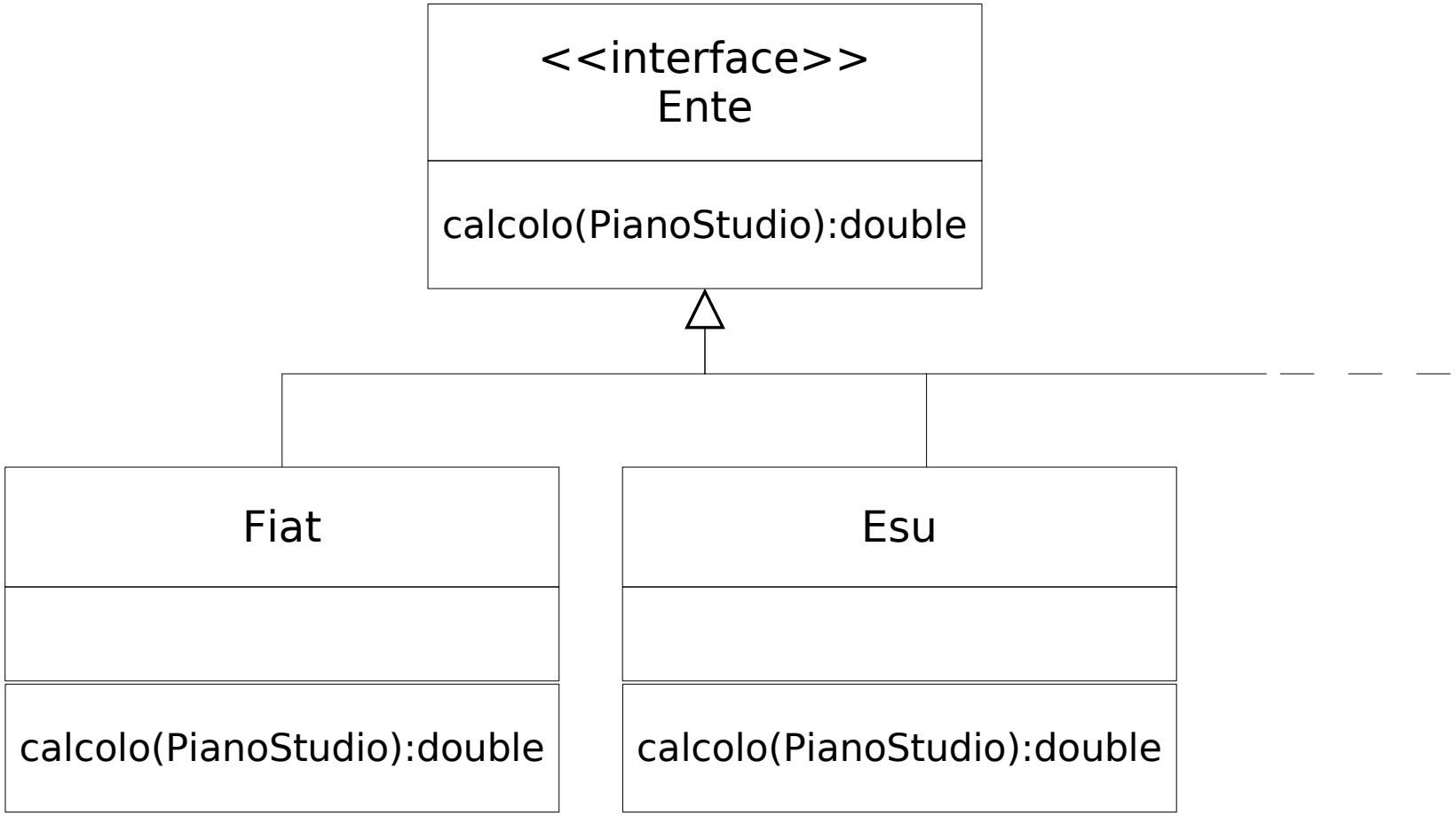
Esempio non O.O.

```
double punteggio(PianoStudio ps, Ente e){
   switch (ente.getTipo()){
      case FIAT:
         return ps.getEsamiSuperati().size();
      case ESU:
         double res = 0.0;
         for (Esame e: ps.getEsamiSuperati())
            res+=e.getVoto();
         return res;
      case TELECOM:
         …
```

O.O primo passo

Il primo passo per la gestione componibile è usare il polimorfismo. In questo caso, supponiamo di avere una interfaccia e varie implementazioni differenti per ogni ente.

Polimorfismo



Polimorfismo

```
interface Ente {
 double punteggio(PianoStudio ps);
}
class FIAT implements Ente {
 public double punteggio(PianoStudio ps) {
    return ps.getEsamiSuperati().size();}
}
class ESU implements Ente {
 public double punteggio(PianoStudio ps){
    double res = 0.0;
    for (Esame e: ps.getEsamiSuperati())
        res+=e.getVoto();
    return res;}
}
```

Polimorfismo

```
public class Ordina {
  Ente ente = EnteFactory.create(TIPO);
  public Studente[] ordina(Set<Studente> studenti){
    Studente[] res = new Studente[studenti.lenght];
    double[] punteggi = new double[studenti.lenght];
    int index = 0;
    for (Studente s: studenti){
      double p = ente.punteggio(s.getPiano());
      for (int i=0; i<index;i++)
        if (punteggi[i]<p) break;
      for (int j = index; j>i; j--)
        res[j]=res[j-1]; punteggi[j]=punteggi[j-1];
      res[i]= s; punteggi[i]=p; index++;
    }
  return res;
```

Polimorfismo

Rimane un problema: il recupero del riferimento all'oggetto Ente. Il problema consiste in particolare nella creazione dell'Ente. Finché non abbiamo l'oggetto stesso non possiamo sfruttare il polimorfismo.

Factory risolve in parte il problema (nasconde sotto il tappeto la polvere)

vediamo perché:

Factory

```
public class EnteFactory {
  public static Ente create(int tipo){
     switch (tipo){
      case FIAT:
         return new Fiat();
      case ESU:
         return new Esu();
      case TELECOM:
         return new Telecom();
      …
  }
}
```

Polimorfismo

Come vedete Factory da solo non risolve il problema: se un domani bisogna aggiungere un nuovo ente, tutto il codice esistente rimane invariato eccetto per il metodo create della Factory su cui dobbiamo mettere mano.

Non è un grosso problema se è possibile modificare liberamente tutto il codice. In alcuni casi non è possibile: codice in una libreria.

Librerie

Supponiamo il caso frequente di un'applicazione che usa librerie diverse per risolvere lo stesso problema (vedi Driver JDBC). Si vuole fare in modo che l'applicazione sia indipendente dalle librerie e che la scelta delle librerie sia libera.

Abbiamo quindi 2 parti (anche 3): chi scrive l'applicazione (e non può modificare il codice della libreria) e chi scrive la libreria (e non può modificare il codice dell'applicazione).

In pratica supponiamo che non sia possibile modificare il codice della Factory aggiungendo (o togliendo) nuovi sottotipi

Soluzione

Con il Pattern Product Trader è possibile:

1) rendere il codice pienamente componibile senza dover riscrivere il codice già scritto.

2) eliminare tutti gli switch (o if/then/else in cascata).

3) permettere la registrazione dinamica di nuovi estensioni/implementazioni

Product Trader

Dal punto di vista della scrittura del codice:

1) il client (la classe Ordina) può accedere all'interfaccia Prodotto (Ente) utilizzando una Specifica che permette al Trader di ritornare una opportuna implementazione/sottoclasse.

2) successivamente possono essere scritte da una terza parte le varie implementazioni/sottoclassi del Prodotto

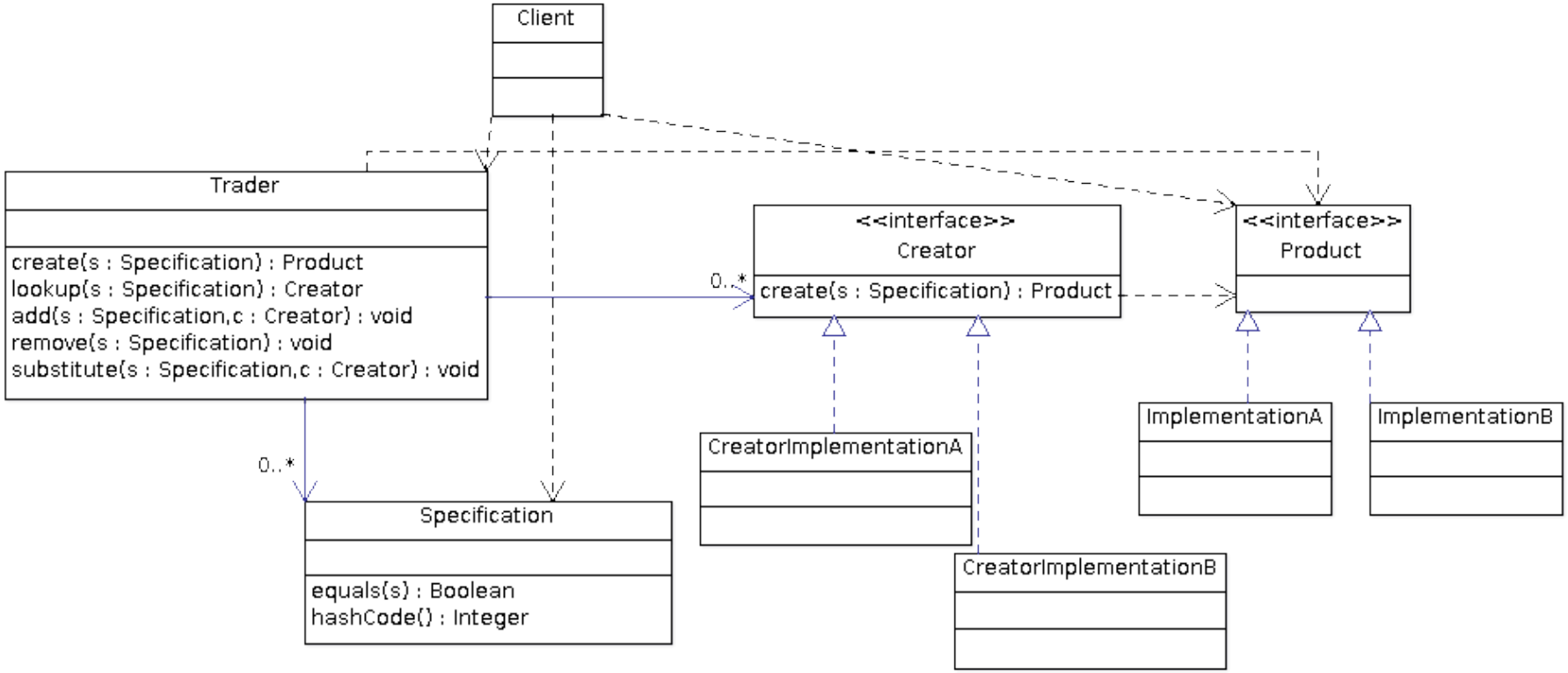
Product Trader

Dal punto di vista dell'esecuzione al run time

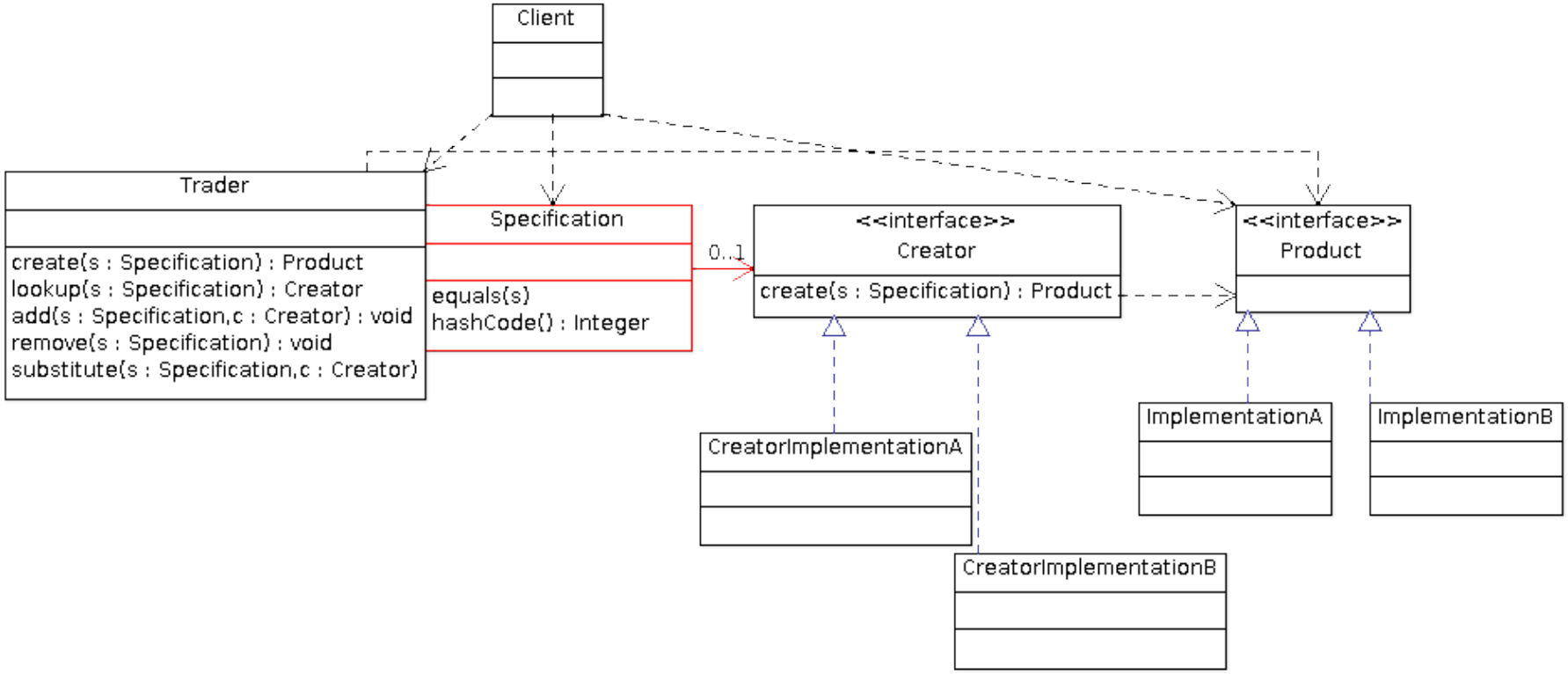
1) le implementazioni devono registrarsi al Trader

2) successivamente il client ottiene una nuova istanza dalla implementazione/estensione opportuna

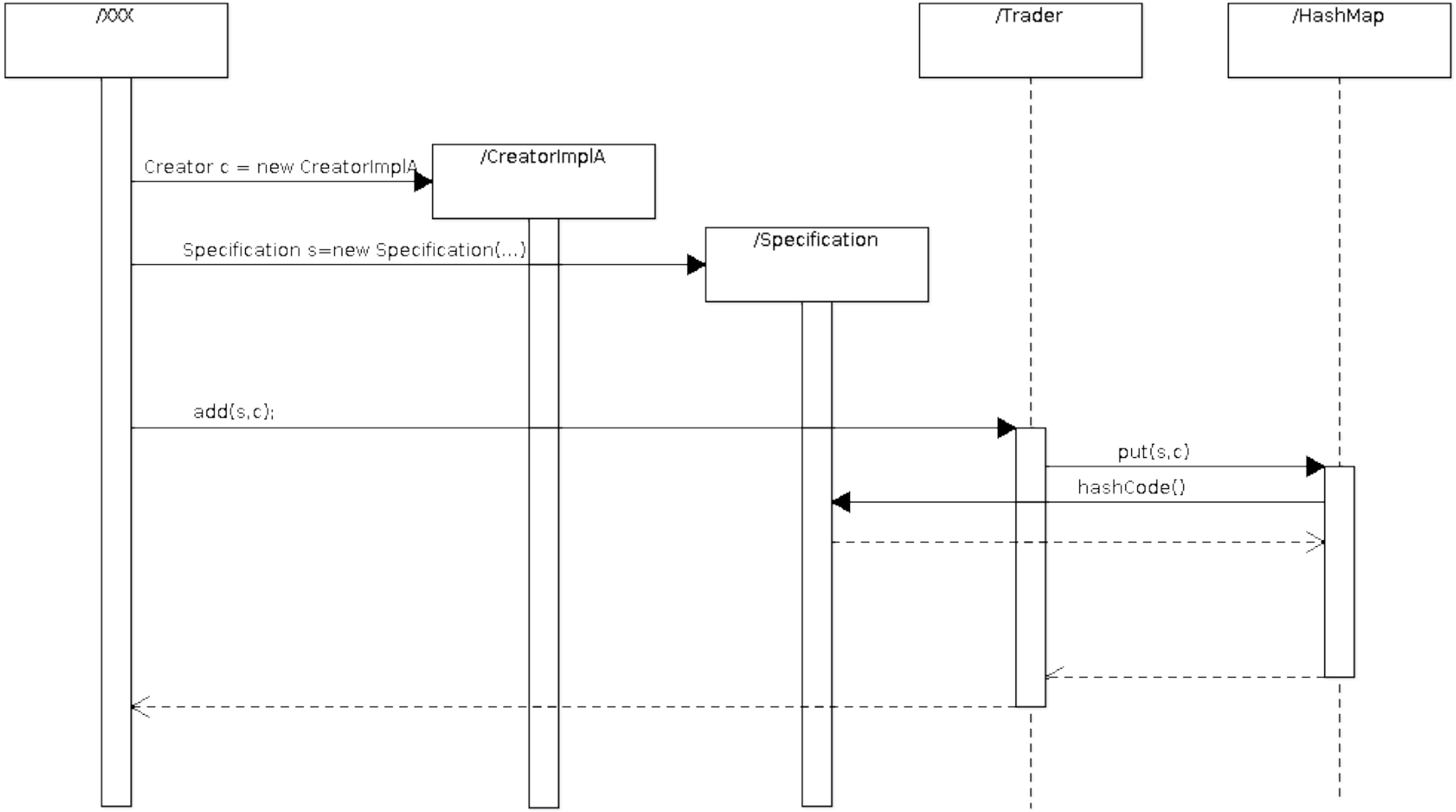
Product Trader



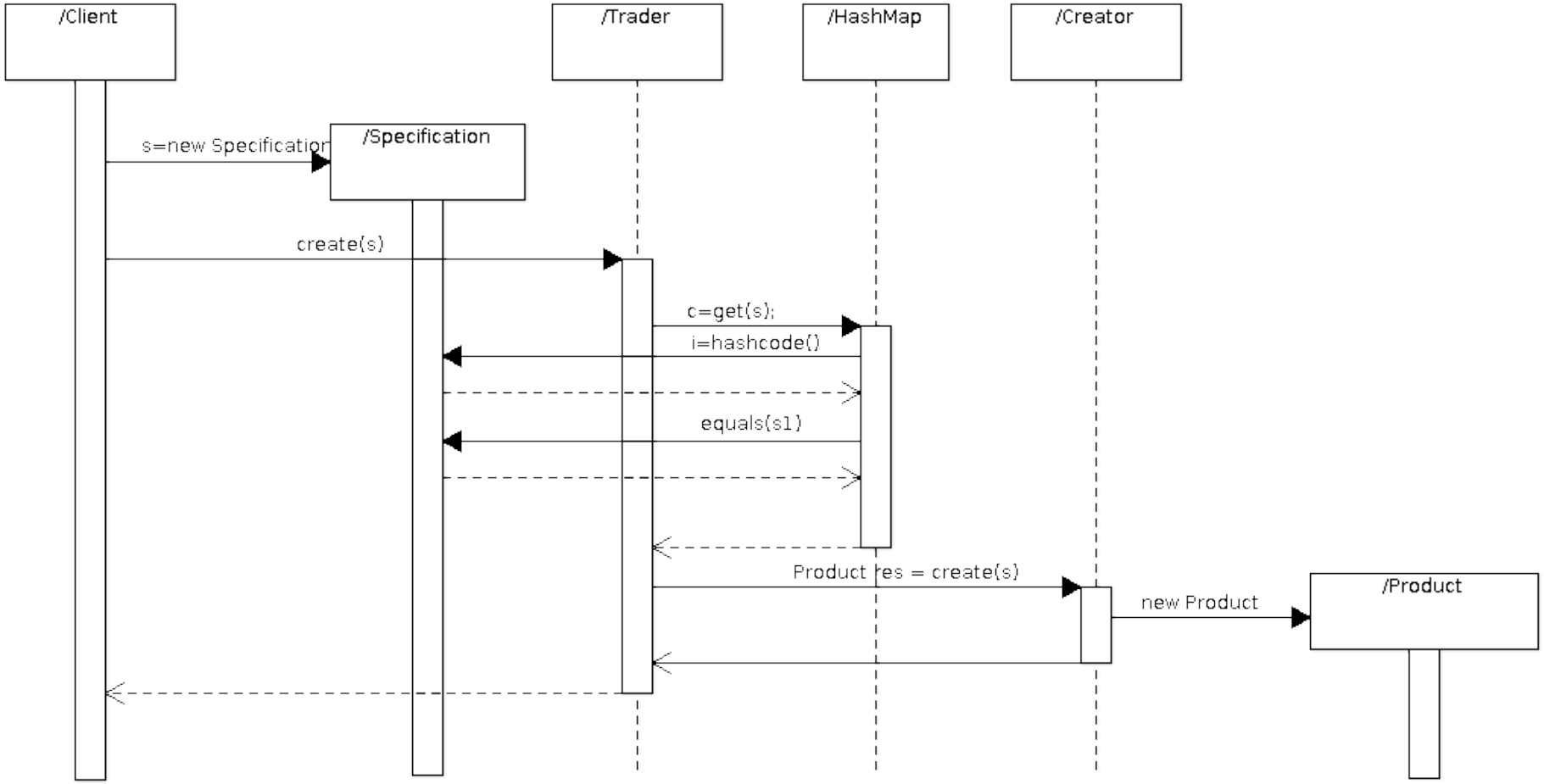
Associazione qualificata



D. sequenza registra



D. sequenza creazione



Specification

```
public class TipoEnte {
  int TIPO;
  public TipoEnte(int tipo){
    this.tipo = tipo;
  }
  public boolean equals(Object o){
    … //soliti controlli
    return tipo==te.tipo;
  }
  public int hashcode(){
   return tipo;
  }
}
```

Product

```
interface Ente {
 double punteggio(PianoStudio ps);
}
public class FIAT implements Ente {
 public double punteggio(PianoStudio ps) {
    return ps.getEsamiSuperati().size();}
}
public class ESU implements Ente {
 public double punteggio(PianoStudio ps){
    double res = 0.0;
    for (Esame e: ps.getEsamiSuperati())
        res+=e.getVoto();
    return res;}
}
```

Creator

```
interface Creator { //meglio EnteCreator

  Ente create(TipoEnte t);
}
public class FIATCreator() {
  public Ente create(TipoEnte t) {
    if (t==FIAT)
       return new Fiat();
    else return null;
  }
}
//implementazione degli altri Creator
...
```

Trader

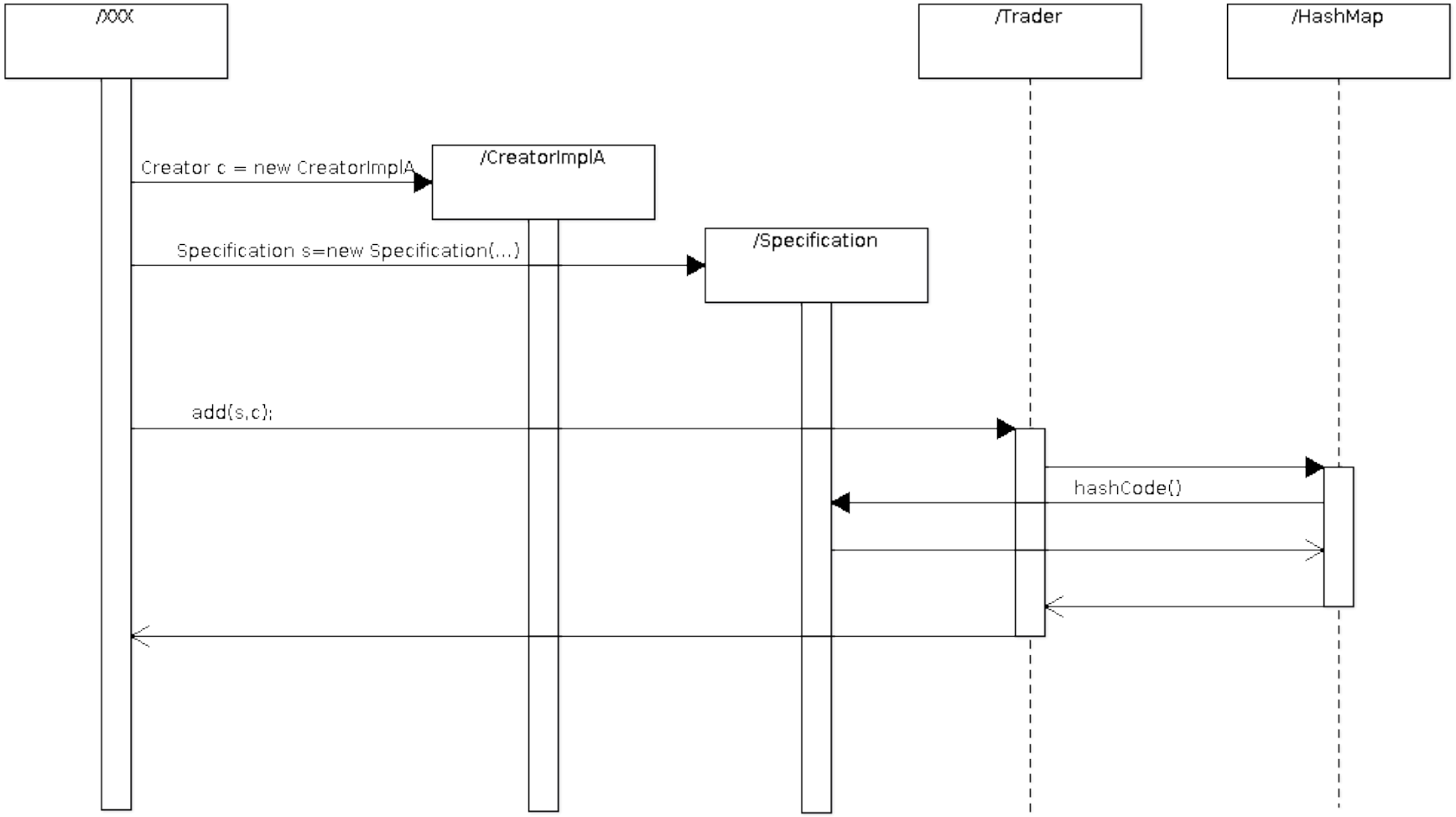
```
public class EnteTrader {//Singleton o static
  static Map<TipoEnte,Creator> map =
      new HashMap<TipoEnte,Creator>();

  static public Ente create(TipoEnte t){
     Creator c = map.get(t);//controllare null
     return c.create(t);
  }
  …
  static public void add(TipoEnte t, Creator c){
   map.put(t,c);
  }
```

Client

```
public class Ordina {
  Ente ente = EnteTrader.create(new TipoEnte(TIPO));
  public Studente[] ordina(Set<Studente> studenti){
    Studente[] res = new Studente[studenti.lenght];
    double[] punteggi = new double[studenti.lenght];
    int index = 0;
    for (Studente s: studenti){
      double p = ente.punteggio(s.getPiano());
      for (int i=0; i<index;i++)
        if (punteggi[i]<p) break;
      for (int j = index; j>i; j--)
        res[j]=res[j-1]; punteggi[j]=punteggi[j-1];
      res[i]= s; punteggi[i]=p; index++;
    }
  return res;
```

Chi registra i Creator?



Chi registra i Creator?

```
public class Main {
 …
 static public void main(String[] args){
  //read from file
  // number , Creator class
  while (!endOfFile){
   int number = … ;
   String class = … ;
   TipoEnte te = new TipoEnte(number);
   Object c = Class.forName(class).newInstance();
   EnteTrader.add(te, (Creator) c);
  }
 }
}
```

Pro e Contro

Pattern Product Trader:

+ Client indipendenti dalle implementazioni dei Prodotti

+ la scelta della implementazione del Prodotto al run-time

-/+ dipendenze spostate dal compile time al run-time

- richiede configurazioni complesse

- possibili ambiguità (+ Impl x 1 spec.)

- costruttore Product senza parametri

Altro esempio

Per gestire le operazioni bancarie supponiamo che una operazione sia rappresentata da una stringa del tipo:

`abbreviazione:descrizione` dove l'abbreviazione descrive il tipo di operazione mentre la descrizione, in dipendenza dal tipo, descrive gli attributi della operazione stessa.

```
bon:30.0;”affitto”;12/12/2012;IT242...
pre:250.0;13/12/2012;
```

Altro esempio

Grazie al polimorfismo un oggetto di tipo Bonifico è in grado di interpretare correttamente la stringa:

`bon:30.0;"affitto";12/12/2012;IT242...`

Mentre un oggetto di tipo Prelievo interpreta correttamente la stringa;

`pre:250.0;13/12/2012;`

Il problema quindi resta quello di creare un oggetto del giusto tipo sul quale invocare il metodo polimorfo

`loadFromString`

Specification

In generale la specifica può essere un oggetto qualsiasi.

Generalmente si usa una stringa con cui si possono esprimere specifiche dalle più semplici alle più complesse esprimibili tramite un linguaggio che verrà interpretato.

Molto spesso basta un stringa che sia interpretabile dall'utente e facilmente maneggiabile dall'applicazione stessa.

Specification

```
public class Specification {
  private String token;
  public Specification(String t) {
    token = t.substring(0,t.indexOf(':'));}
  public int hashCode() {
    return token.hashCode();}
  public boolean equals(Object o) {
    if (!(o instanceof Specification))
      return false;
    Specification s = (Specification) o;
    if (token.equals(s.token))
          return true;
    return false;    } }
```

Product

```
public interface Operazione {
    public boolean loadFromString(String s);
}
```

Creator

```
public interface Creator {
    Operazione create();
}
```

Trader

```
public class Trader {//static o singleton
  Map<Specification,Creator> map =
        new  HashMap<Specification, Creator>();

  public void add(Specification s, Creator c){
      map.put(s, c);
  }
  public Operazione create(Specification s){
        Creator c = map.get(s);
        if (c!=null) return c.create();
        else return null;
    }
}
```

Client

```
public class Cliente {
  public Operazione createFromString(String s) {
    Specification spec = new Specification(s);
    Operazione op = Trader.create(spec);
    if (op != null && op.loadFromString(s)) {
        return op;
     }
     return null;
    }
}
```

Service Manager

- In Java viene usata una versione più semplice: Service Manager
- Accesso ai DB tramite JDBC
- Un url specifica univocamente il database: “`jdbc:derby:sample`”
- La classe DriverManager fa da ServiceManager

DriverManager/Product Trader

- La classe Specification è la String
- Il Creator è il Driver
- Il Trader è il Manager
- Il Product è la Connection

Driver

```
Package java.sql;

interface Driver {//Service

  boolean acceptsURL(String url);
  Connection connect(String url) throws SQLException
  …

}
```

DriverManager

```
public class DriverManager {//Service Manager
  static set<Driver> drivers = new …;

  static public void registrDriver(Driver d){
    drivers.add(d);
  }
  static public Connection getConnection(String url)
  throws SQLException {
    for (Driver d: drivers)
      if (d.acceptUrl(url)
        return d.connect(url);

    throw new SQLException(...);
  }
}
```

Driver Implementation

```
public class DerbyDriver implements Driver {
  …
  public boolean acceptURL(String s){
    return s.startWith("jdbc:derby:");
  }

  public Connection connect(String url)
  throws SQLException {
  …
  }
  static {
    Driver d = new DerbyDriver();
    DriverManager.registerDriver(d);
  }
```

JDBC DriverManager

```
Java -Djdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.taste.ourDriver
```

```
public class DriverManager {
  …
static {
  String drivers=System.getProperty("jdbc.drivers");

  String[] driverClass = jdbcDrivers.split(":");

  for (String className : driverClass) {
        Class.forName(className);
  }
}
```

Commenti

- Il codice static
-

Domande