Programmazione a oggetti 2018 parte 4

Giorgio Bruno

Dip. Automatica e Informatica
Politecnico di Torino
email: giorgio.bruno@polito.it

Quest'opera è stata rilasciata sotto la licenza Creative Commons
Attribuzione-Non commerciale-Non opere derivate 3.0 Unported.
Per leggere una copia della licenza visita il sito web
http://creativecommons.org/licenses/by-nc-nd/3.0/

Contenuto

3 esercizi d'esame

- Biblioteca
- Pubblicazioni
- GestioneOrdini

4 soluzioni di compiti

- Bank
- HandleApplications
- IssueManager
- ExamHandling

Biblioteca

Si scriva un programma per la gestione dei prestiti dei libri di una biblioteca.

I metodi principali sono offerti dalla classe **Biblioteca**. Questa classe e le classi applicative si trovano nel package **biblioteca**.

Il programma consente l'inserimento di libri e utenti, la richiesta e restituzione di libri e fornisce varie statistiche.

Nel seguito sono presentati i requisiti, le classi iniziali e il programma di test.

I metodi illustrati sono della classe Biblioteca.

nel main

Biblioteca b = **new Biblioteca();**

b.addLibro("rosso", 1, "falco"); b.addLibro("giallo", 2, "falco", "rondine");

Requisiti: R1

Inserimenti

La biblioteca può avere più copie (volumi) dello stesso libro. Il metodo **addLibro** (titolo, nVolumi, autori) inserisce un libro e i volumi corrispondenti.

Un libro ha il titolo e la lista dei nomi degli autori. Ogni volume ha un indice intero progressivo univoco che parte da 0.

Il metodo dà errore se il titolo è ripetuto.

Tutte le eccezioni sollevate sono di tipo BiblioEccezione.

nel main

Biblioteca b = **new Biblioteca();**

b.addLibro("rosso", 1, "falco"); b.addLibro("giallo", 2, "falco", "rondine");

In Biblioteca

public void addLibro (String titolo, int nVolumi, String... autori) throws BiblioEccezione {}

R1

Il metodo **addUtente** (nome, maxPrestiti, durata) inserisce un utente con il nome, il n. max di prestiti che può avere nello stesso periodo, la durata (massima) in giorni dei suoi prestiti.

Il metodo dà errore se l'utente è ripetuto.

nel main

b.addUtente("quercia", 2, 7); b.addUtente("faggio", 1, 10);

In Biblioteca

public void addUtente (String nome, int maxPrestiti, int durata) throws BiblioEccezione {}

R2

Date

La biblioteca gestisce una sua data corrente (LocalDate) che è inizializzata con quella di sistema.

La data corrente si può variare aggiungendo un n. di giorni con il metodo **addGiorni**.

Si può leggere la data corrente della biblioteca con il metodo LocalDate **getOggi**.

Nel main

b.addGiorni(1);

b.getOggi()

In Biblioteca

public void addGiorni(int nGiorni) {}

public LocalDate getOggi() {}

Prestiti **R3**

Il metodo

public PrestitoI addPrestito(String utente, String titolo) throws BiblioEccezione

genera un prestito per un utente e un volume. Il volume è quello con l'indice inferiore tra quelli disponibili corrispondenti al libro di cui è dato il titolo.

Il metodo dà errore se l'utente ha già in prestito il n. max di volumi o se non trova nessun volume disponibile.

Il prestito contiene la data di scadenza (ottenuta sommando alla data corrente la durata dei prestiti dell'utente) e la data di restituzione. Le date sono di tipo LocalDate.

Titolo, indice, nome utente e date sono leggibili con getters.

I volumi sono inizialmente disponibili; quando è dato in prestito, un volume diventa non disponibile e ritorna tale alla restituzione del volume.

Interfaccia PrestitoI

Un prestito è visibile attraverso l'interfaccia PrestitoI.

```
public interface PrestitoI {
int getIndice();
String getTitolo();
LocalDate getScadenza();
LocalDate getDataRestituzione();
String getNomeUtente ();

default String getInfo() {
return String.format("prestito: utente %s titolo %s volume % d scadenza %s",
getNomeUtente(), getTitolo(), getIndice(), getScadenza().toString());
}
static void stampaListaPrestiti(String info, List<PrestitoI> lista, LocalDate oggi) {
System.out.println(info + " " + oggi + " n:" + lista.size());
for (PrestitoI p: lista) System.out.println("  " + p.getInfo());
}
}
```

R4

Restituzioni

Il metodo

public void restituzione(String utente, int indice) throws BiblioEccezione

chiude il prestito (scrivendo la data di restituzione, cioè quella corrente) per il volume di cui è dato l'indice.

Il metodo dà errore se non c'è un prestito aperto per quell'indice e quell'utente.

Il metodo List<PrestitoI> **prestitiScaduti** dà l'elenco dei prestiti scaduti ordinati per scadenze crescenti.

Note:

Un prestito è aperto se la data di restituzione è nulla;

è chiuso se la data di restituzione non è nulla;

è scaduto se è aperto e la data di scadenza è prima di oggi (la data corrente della biblioteca).

R3, R4

Nel main

```
PrestitoI p = b.addPrestito("faggio", "giallo");

List<PrestitoI> prestitiScaduti = b.prestitiScaduti();

b.restituzione("quercia", 3);
```

R5: statistiche

SortedMap<String, Long> **nPrestitiXTitolo** ();

dà il n. dei prestiti chiusi per titolo (i titoli sono ordinati alfabeticamente).

SortedMap<Long, Set<String>> **titoliXnPrestiti** ();

dà l'elenco dei titoli ordinati alfabeticamente per il n. decrescente dei prestiti chiusi.

List<InfoI> lista **utentiNPrestiti**();

dà la lista nome utente-nPrestiti chiusi ordinata per n. decrescente e nomi crescenti.

List<InfoI> lista **autoriNPrestiti**();

dà la lista autore-nPrestiti chiusi ordinata per n. decrescente e nomi crescenti.

Interfaccia InfoI

Gli elementi delle ultime due liste sono visibili attraverso l'interfaccia InfoI.

```
public interface InfoI {
        String getNome();
        long getValore();
        default String getInfo() {
                return getNome() +": " + getValore() + " ";
        }
        static void stampaListaInfoI(String nomeLista, List<InfoI> lista) {
                System.out.println(nomeLista);
                for (InfoI i:lista) System.out.print(i.getInfo());
                System.out.println();
        }
}
```

Il main di prova

```
public static void main(String[] args) throws BiblioEccezione {
Biblioteca b = new Biblioteca();
b.addLibro("rosso", 1, "falco"); b.addLibro("giallo", 2, "falco", "rondine");
b.addLibro("verde", 3, "aquila"); b.addLibro("nero", 4, "corvo");
        try{b.addLibro("giallo", 2);}catch(BiblioEccezione e)
        {{System.out.println(e.getMessage());};}
b.addUtente("quercia", 2, 7); b.addUtente("faggio", 1, 10);
b.addUtente("acero", 3, 14);
System.out.println("oggi " + b.getOggi());  //oggi 2016-05-12
```

Il main di prova

```
PrestitoI p = b.addPrestito("faggio", "giallo");
System.out.println(p.getInfo());
//prestito: utente faggio titolo giallo volume  1 scadenza 2016-05-22
b.addPrestito("quercia", "rosso"); b.addPrestito("acero", "giallo");
        try{b.addPrestito("faggio", "nero");}catch(BiblioEccezione e)
        {{System.out.println(e.getMessage());};}
b.addGiorni(1); //domani
b.addPrestito("quercia", "verde"); b.addPrestito("acero", "verde");
b.restituzione("quercia", 0);
```

Il main di prova

```
b.addGiorni(20);
List<PrestitoI> prestitiScaduti = b.prestitiScaduti();
PrestitoI.stampaListaPrestiti("prestitiScaduti",
prestitiScaduti, b.getOggi());
```

```
prestitiScaduti 2016-06-02 n:4
  prestito: utente quercia titolo verde volume  3 scadenza 2016-05-20
  prestito: utente faggio titolo giallo volume  1 scadenza 2016-05-22
  prestito: utente acero titolo giallo volume  2 scadenza 2016-05-26
  prestito: utente acero titolo verde volume  4 scadenza 2016-05-27
```

```
b.restituzione("quercia", 3); b.restituzione("faggio", 1);
b.restituzione("acero", 2); b.restituzione("acero", 4);
```

Il main di prova

```
List<InfoI> lista = b.utentiNPrestiti();
InfoI.stampaListaInfoI("utentiNPrestiti", lista);
InfoI.stampaListaInfoI("autoriNPrestiti", b.autoriNPrestiti());
SortedMap<String, Long> mappa = b.nPrestitiXTitolo();
System.out.println("nPrestitiXTitolo " + mappa);
SortedMap<Long, Set<String>> mappa2 = b.titoliXnPrestiti();
System.out.println("titoliXnPrestiti " + mappa2);
b.addGiorni(20);
PrestitoI.stampaListaPrestiti("prestitiScaduti", b.prestitiScaduti(), b.getOggi());
}
```

utentiNPrestiti
acero: 2 quercia: 2 faggio: 1
autoriNPrestiti
falco: 3 aquila: 2 rondine: 2
nPrestitiXTitolo {giallo=2, rosso=1, verde=2}
titoliXnPrestiti {2=[giallo, verde], 1=[rosso]}
prestitiScaduti 2016-06-22 n:0

Analisi ad oggetti

Quali sono i tipi di oggetto necessari? E le loro classi?

Un oggetto è un contenitore di attributi e di collegamenti con altri oggetti.

Definire un class model.

Progetto ad oggetti

In quali collezioni vanno memorizzati gli oggetti?

Gli oggetti primari nelle collezioni dell'oggetto facciata, gli oggetti secondari nelle collezione di altri oggetti (primari o anche secondari).

Quali collezioni nell'oggetto facciata: mappe, liste o set?

Distribuzione dei metodi negli oggetti (principio di information hiding: ogni oggetto tratta le proprie informazioni).

Inserimento di ridondanze per questioni di efficienza.

Decidere le navigazioni tra gli oggetti.

Raffinamenti successivi in base all'analisi dei requisiti dei metodi principali.

Analisi dei metodi principali

Il metodo **addLibro** (titolo, nVolumi, autori) inserisce un libro e i volumi corrispondenti.

Attributi di Libro: titolo e lista nomi autori.

Attributi di Volume: indice, boolean disponibile = true; //stato

Che rapporto tra libro e volumi?

Un libro contiene la lista dei suoi volumi, generata dal costruttore di Libro.

Volume può essere una classe annidata in Libro.

Il metodo **addUtente** (nome, maxPrestiti, durata) inserisce un utente con il nome, il n. max di prestiti che può avere nello stesso periodo, la durata (massima) in giorni dei suoi prestiti. Serve la classe Utente con i tre attributi suddetti.

Il metodo **addPrestito** (String utente, String titolo) *genera un prestito per un utente e un volume*. Un prestito è un oggetto collegato ad un utente e ad un volume, con due attributi (date).

Libri, utenti e prestiti sono oggetti primari; i volumi sono secondari e sono inclusi nei libri.

Primo modello

String titolo;
List<String> autori;

String nome;
int maxPrestiti;
int durata;

1 1

n n

1

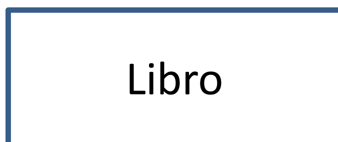
n

LocalDate scadenza;
LocalDate dataRestituzione;

int indice;
boolean disponibile = true;

Relazioni binarie navigabili in entrambe le direzioni: molteplicità 1: un riferimento; n: una lista di riferimenti (oppure un set o una mappa).

Per avere un modello normalizzato servirebbe la classe Autore.



n

n

String nome;

Analisi dei prestiti

Un prestito può essere aperto (anche scaduto) o chiuso. I prestiti aperti e quelli chiusi stanno nella stessa collezione?

Il metodo **restituzione** (String utente, int indice) come fa a rintracciare il prestito dato l'indice del volume in prestito?

Con una mappa<Integer, Prestito> si può rintracciare il prestito dato l'indice (univoco del volume). La mappa contiene i prestiti aperti, quelli chiusi si possono trasferire in una lista sulla quale operano i metodi di statistica.

Come si ottiene il n. dei prestiti chiusi per utente e per libro? Si possono contare i prestiti chiusi associati, oppure si possono definire dei contatori che sono incrementati dalle restituzioni.

Con la seconda soluzione si introducono delle *ridondanze*.

Secondo modello

String titolo;
List<String> autori;

1

n

1

n

String nome;
int maxPrestiti;
int durata;

Attributi essenziali

Le frecce indicano il senso di navigazione

1

n

LocalDate scadenza;
LocalDate dataRestituzione;

int indice;
boolean disponibile = true;

Collezioni principali
Map<String,Libro> libri;
Map<String,Utente> utenti;
Map<Integer, Prestito> prestitiAperti;
List<Prestito> prestitiChiusi;

Attributi di supporto (ridondanti)
Libro: int nPrestitiChiusi.
Utente: int nPrestiti; //quelli aperti
int nPrestitiChiusi;

Analisi statistiche

String titolo;
List<String> autori;

String nome; int maxPrestiti;
int durata;

1

n

1

n

1

n

Prestito

int indice;
boolean disponibile = true;

LocalDate scadenza;
LocalDate dataRestituzione;

Collezioni principali
Map<String,Libro> libri;
Map<String,Utente> utenti;
Map<Integer, Prestito>
prestitiAperti;
List<Prestito> prestitiChiusi;

Attributi di supporto (ridondanti)
Libro: int nPrestitiChiusi.
Utente: int nPrestiti; //quelli aperti
int nPrestitiChiusi;

SortedMap<String, Long> **nPrestitiXTitolo** (); counting dei prestiti chiusi per titolo.
SortedMap<Long, Set<String>> **titoliXnPrestiti** (); serve l'attributo di supporto
nPrestitiChiusi oppure si può riaprire la mappa precedente.
List<InfoI> lista **utentiNPrestiti**(); counting dei prestiti chiusi per utente.
List<InfoI> lista **autoriNPrestiti**(); flatMap da prestiti ad autori e poi mappa che si riapre per collocare key e value in lista.

Libro

```
class Libro {
private String titolo; String getTitolo() {return titolo;}
private List<String> autori; List<String> getAutori() {return autori;}
Libro (String titolo, int n, List<String> autori, int indice){
        this.titolo = titolo; this.autori = autori;
        for (int i = 0; i < n; i++) volumi.add(new Volume(indice++));
}
private List<Volume> volumi = new ArrayList<>();
Volume getVolume() {
        for (Volume v: volumi){
        if (v.disponibile) {v.disponibile = false; return v;}}
        return null;}

private int nPrestitiChiusi = 0; void incrNPrestitiChiusi() {nPrestitiChiusi ++;}
int getNPrestitiChiusi() {return nPrestitiChiusi;}  // dipende dal tipo di mappa
long getNPrestitiChiusiL() {return nPrestitiChiusi;} // nelle statistiche
private Libro libro = this;  // per la navigazione da volume a libro
```

Libro.Volume

```
class Volume {
private int indice; int getIndice() {return indice;}
Volume (Integer indice) {this.indice = indice;}
private boolean disponibile = true;
void restituzione() {disponibile = true;}
Libro getLibro() {return libro;}
}
}
```

Utente

```
class Utente implements InfoI {
private String nome; public String getNome(){return nome;}
private int maxPrestiti;
private int durata; int getDurata() {return durata;}
Utente(String nome, int maxPrestiti, int durata)
          {this.nome = nome; this.maxPrestiti = maxPrestiti;
                    this.durata = durata;}
private int nPrestiti = 0; int getNPrestiti() {return nPrestiti;}
void incrNPrestiti() {nPrestiti++;}
void decrNPrestiti() {nPrestiti--;}
boolean prestitoAmmissibile () {return nPrestiti < maxPrestiti;}
private int nPrestitiChiusi = 0; int getNPrestitiChiusi() {return nPrestitiChiusi;}
void incrNPrestitiChiusi() {nPrestitiChiusi++;}
public long getValore() {return nPrestitiChiusi;}
}

//getNome e getValore sono i metodi di InfoI
```

Prestito

```
public class Prestito implements PrestitoI {
private Utente utente; Utente getUtente() {return utente;}
private Libro.Volume volume; Libro.Volume getVolume() {return volume;}
private LocalDate scadenza; public LocalDate getScadenza() {return scadenza;}
Prestito(Utente utente, Libro.Volume volume, LocalDate scadenza) {
        this.utente = utente; this.volume = volume; this.scadenza = scadenza;
}
private LocalDate dataRestituzione = null;
public int getIndice() {return volume.getIndice();}
public String getTitolo (){return volume.getLibro().getTitolo();}
public LocalDate getDataRestituzione() {return dataRestituzione;}
void setDataRestituzione(LocalDate dataRestituzione) {
        this.dataRestituzione = dataRestituzione;
}
public String getNomeUtente () {return utente.getNome();}
}
```

In grassetto i metodi di PrestitoI

Biblioteca

```
public class Biblioteca {

private Map<String,Libro> libri = new HashMap<>();
private Map<String,Utente> utenti = new HashMap<>();
private Map<Integer, Prestito> prestitiAperti = new HashMap<>();
private List<Prestito> prestitiChiusi = new ArrayList<>();
private int indice = 0; // indice dei volumi
//gestione delle date
private LocalDate oggi = LocalDate.now();

public void addGiorni(int nGiorni) {this.oggi = oggi.plusDays(nGiorni);}
public LocalDate getOggi() {return oggi;}
```

Biblioteca

```
public void addLibro (String titolo, int nVolumi, String... autori)
        throws BiblioEccezione {
if (libri.containsKey(titolo))
        throw new BiblioEccezione ("titolo duplicato - " + titolo);
Libro l = new Libro(titolo, nVolumi, Arrays.asList(autori), indice);
libri.put(titolo, l); indice += nVolumi; }
// Il costruttore del libro riceve l'indice corrente e il n. di volumi da generare.

public void addUtente (String nome, int maxPrestiti, int durata)
        throws BiblioEccezione {
if (utenti.containsKey(nome))
        throw new BiblioEccezione("utente duplicato - " + nome);
Utente u = new Utente(nome, maxPrestiti, durata);utenti.put(nome, u);}
```

Biblioteca

```
public PrestitoI addPrestito (String utente, String titolo)
            throws BiblioEccezione {
Utente u = utenti.get(utente); Libro l = libri.get(titolo);
if (!u.prestitoAmmissibile())
            throw new BiblioEccezione ("max prestiti per " + utente);
Libro.Volume v = l.getVolume();
if (v == null)
            throw new BiblioEccezione ("volume non disponibile - " + titolo);
u.incrNPrestiti();
Prestito p = new Prestito(u, v, oggi.plusDays(u.getDurata()));
prestitiAperti.put(v.getIndice(), p);
return p;
}
```

Il metodo controlla che un nuovo prestito sia ammissibile per l'utente e chiede al libro di fornire l'indice del primo volume disponibile. In caso positivo aumenta nPrestiti dell'utente e aggiunge un nuovo prestito alla mappa prestitiAperti.

Biblioteca

```
public void restituzione(String utente, int indice)
        throws BiblioEccezione {
Prestito p = prestitiAperti.get(indice);
if (p == null) throw new BiblioEccezione ("volume non in prestito " + indice);
if (!p.getUtente().getNome().equals(utente))
        throw new BiblioEccezione (String.format("volume %d non in prestito a %s", indice, utente));
prestitiAperti.remove(indice);
p.setDataRestituzione(oggi); prestitiChiusi.add(p);
p.getUtente().decrNPrestiti(); p.getUtente().incrNPrestitiChiusi();
p.getVolume().restituzione();
p.getVolume().getLibro().incrNPrestitiChiusi();
}
```

Il metodo sposta il prestito dalla mappa alla lista e aggiorna i vari contatori (punto critico).

Biblioteca - statistiche

```
public List<PrestitoI> prestitiScaduti() {
return prestitiAperti.values().stream()
            .filter(p -> p.getScadenza().isBefore(oggi))
            .sorted(comparing(Prestito::getScadenza))
            .collect(toList());}

public SortedMap<String, Long> nPrestitiXTitolo () {
return prestitiChiusi.stream()
            .collect(groupingBy(Prestito::getTitolo,
                        TreeMap::new,
                        counting()));}
```

Biblioteca

```
public SortedMap<Long, Set<String>> titoliXnPrestiti () {
return prestitiChiusi.stream()
.collect(groupingBy(Prestito::getTitolo, counting()))
.entrySet().stream()
.collect(groupingBy(e -> e.getValue(),
            () -> new TreeMap<>(reverseOrder()) ,
            mapping(e -> e.getKey(), toCollection(TreeSet::new))));
}
```

non usa nPrestitiChiusi di Libro

OPPURE

```
public SortedMap<Long, Set<String>> titoliXnPrestiti2 () {
return libri.values().stream()
.filter(l -> l.getNPrestitiChiusi() > 0)
.collect(groupingBy(l -> (long) l.getNPrestitiChiusi(),
            () -> new TreeMap<>(reverseOrder()),
            mapping(Libro::getTitolo, toCollection(TreeSet::new))));
}
```

usa nPrestitiChiusi di Libro;

Statistiche con InfoI

I metodi utentiNPrestiti e autoriNPrestiti danno una lista di oggetti che implementano l'interfaccia InfoI.

Nel primo caso si possono usare gli oggetti di classeUtente in quanto la classe può implementare l'interfaccia.

Nel secondo caso mancando la classe Autore occorre definire una nuova classe, ad es. Info, che implementi l'interfaccia.

```
class Info implements InfoI{
        private String nome;
        private long valore;
        public String getNome() {return nome;}
        public long getValore() {return valore;}
        public Info(String nome, long valore)
                {this.nome = nome; this.valore = valore;}

}
```

Biblioteca

```
public List<InfoI> utentiNPrestiti() {
List<InfoI> elenco = utenti.values().stream()
.sorted(comparing(Utente::getNPrestitiChiusi,
            reverseOrder()).thenComparing(Utente::getNome))
.collect(toList());
return elenco;
}
```

Biblioteca

```
public List<InfoI> autoriNPrestiti() {
List<InfoI> elenco = prestitiChiusi.stream()
.flatMap(p -> p.getVolume().getLibro().getAutori().stream())
.collect(groupingBy(a -> a, counting()))
.entrySet().stream()
.map(e -> {return new Info(e.getKey(), e.getValue());})
.sorted(comparing(InfoI::getValore,
            reverseOrder()).thenComparing(InfoI::getNome))
.collect(toList());
return elenco;
}
```

Si parte dalla lista dei prestiti chiusi, si passa agli autori (flatMap) e si raggruppano contandone il numero; si riparte dall'entrySet e si passa ad uno stream di Info con una map, si ordina e si produce una lista.

BiblioEccezione

```
public class BiblioEccezione extends Exception {
BiblioEccezione(String s) {super(s);}}
```

GUI

Le operazioni principali si possono eseguire con l'interfaccia grafica seguente.

GuiTestBiblioteca

```
public class GuiTestBiblioteca extends JFrame implements ActionListener {
// attributi accessibili al metodo actionPerformed
private JTextArea ta;
private JTextField tf1; private JTextField tf2; private JTextField tf3;
private Biblioteca bib = new Biblioteca();

enum Etichetta {addLibro, addUtente, addPrestito, restituzione,
                        titoliXnPrestiti, utentiNPrestiti
               }
public static final int LINES = 10;
public static final int CHAR_PER_LINE = 40;
```

GuiTestBiblioteca

public static void *main* (String[] args)
{JFrame.setDefaultLookAndFeelDecorated(true);
    GuiTestBiblioteca gui = new GuiTestBiblioteca();
    gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    gui.pack();
gui.setVisible(true);}

private void **addButton** (String label, JPanel panel, ActionListener listener){
    JButton button = new JButton(label);
    button.addActionListener(listener);
    panel.add(button);}

private JTextField **addTF** (String label, int length, JPanel panel){
    JTextField tf = new JTextField(label,length);
tf.setBackground(Color.white); panel.add(tf); return tf;}

GuiTestBiblioteca

```
public GuiTestBiblioteca() {
setTitle("Prestiti");  Container contentPane = getContentPane();
JPanel buttonPanel = new JPanel(); buttonPanel.setBackground(Color.white);
addButton(Etichetta.addLibro.toString(),buttonPanel,this);
addButton(Etichetta.addUtente.toString(),buttonPanel,this);
addButton(Etichetta.addPrestito.toString(),buttonPanel,this);
addButton(Etichetta.restituzione.toString(),buttonPanel,this);
addButton(Etichetta.titoliXnPrestiti.toString(),buttonPanel,this);
addButton(Etichetta.utentiNPrestiti.toString(),buttonPanel,this);
contentPane.add(buttonPanel, BorderLayout.SOUTH);
JPanel textPanel = new JPanel();  textPanel.setBackground(Color.blue);
ta = new JTextArea(LINES, CHAR_PER_LINE);
ta.setBackground(Color.white); textPanel.add(ta);
contentPane.add(textPanel, BorderLayout.CENTER);
JPanel textFPanel = new JPanel(); textFPanel.setBackground(Color.white);
tf1 = addTF("", 10, textFPanel);  tf2 = addTF("", 10, textFPanel);
tf3 = addTF("", 10, textFPanel);
contentPane.add(textFPanel, BorderLayout.NORTH); }
```

GuiTestBiblioteca

```
public void actionPerformed(ActionEvent e) {
String actionCommand = e.getActionCommand();
ta.setText(""); String s1 = tf1.getText(); String s2 = tf2.getText();
String s3 = tf3.getText();
try{
if (actionCommand.equals(Etichetta.addLibro.toString())){
        bib.addLibro(s1, Integer.parseInt(s2), s3); ta.setText("done");
}
else if (actionCommand.equals(Etichetta.addUtente.toString())) {
        bib.addUtente(s1, Integer.parseInt(s2), Integer.parseInt(s3));
        ta.setText("done");
}
else if (actionCommand.equals(Etichetta.addPrestito.toString())) {
        ta.setText(bib.addPrestito(s1, s2).getInfo());
}
```

GuiTestBiblioteca

```
else if (actionCommand.equals(Etichetta.restituzione.toString())) {
            bib.restituzione(s1, Integer.parseInt(s2));
            ta.setText("done");}
else if (actionCommand.equals(Etichetta.titoliXnPrestiti.toString())) {
            ta.setText(bib.titoliXnPrestiti().toString());
}
else if (actionCommand.equals(Etichetta.utentiNPrestiti.toString())) {
            ta.setText(bib.utentiNPrestiti().stream()
                        .map(i -> {return i.getNome() + ": " + i.getValore();})
                                    .collect(joining(", ","[","]")));
}
}catch(BiblioEccezione ex){ta.setText(ex.getMessage());}
 catch(NumberFormatException ex){ta.setText(ex.getMessage());}
}
}
```

Nell'ultimo caso si costruisce la stringa da stampare con uno stream e un joining; oppure si può definire toString in Utente.

Pubblicazioni

Il programma fornisce varie statistiche sugli articoli pubblicati in riviste. La classe principale è **Pubblicazioni**; le classi applicative si trovano nel package pubblicazioni.

Requisiti: R1 - Riviste

Il metodo **Rivista addRivista** (String titolo, double impactFactor)
registra una rivista con titolo e impact factor; segnala la duplicazione del titolo.

Il metodo Rivista **getRivista** (String titolo)
cerca in base al titolo e segnala la mancanza.

Il metodo List<Rivista> **getRiviste** ()
dà la lista delle riviste ordinate per impact factor decr e titolo.

Le eccezioni sono di tipo Exception.

R2: articoli

Il metodo di Rivista

void **addArticolo** (String titolo, Year anno, String... nomiAutori)

registra un articolo con titolo, anno di pubbl., nomi autori.

Il titolo dell'articolo deve essere univoco per la rivista; segnala la duplicazione.

Il metodo di Rivista

List<Articolo> **getArticoli** ()

dà la lista degli articoli ordinati per anno crescente e titolo.

Getters per anno e titolo

R3: autori

Il metodo **Autore getAutore** (String nome)
dà un autore in base al nome e segnala la mancanza.

Il metodo List<Autore> **getAutori** ()
dà la lista degli autori ordinati per impact factor decr e nome.
L'impact factor si ottiene sommando gli impact factor degli
articoli dell'autore; l'IF di un articolo è pari a quello della rivista che lo contiene.

<u>Il metodo di Autore</u>
List<Articolo> **getArticoli** ()
dà la lista degli articoli ordinati per anno crescente e titolo.

Getter per nome.

Nota: gli autori compaiono in addArticolo di Rivista; le riviste devono ricevere dalla classe principale la mappa degli autori per poterla aggiornare.

R4: statistiche

SortedMap <Year, Long> **articoliPerAnno** ()
il n. di articoli per anni ordinati

Rivista **rivistaMaxArticoli**()
la rivista con il n. max di articoli

SortedMap <String, Long> **articoliPerAutore** ()
il n. di articoli per autore; i nomi degli autori sono ordinati

SortedMap <Long, TreeSet<String>> **autoriPerNarticoli** ()
gli autori per il n. di articoli; la mappa è ordinata per n. decrescenti, gli articoli sono ordinati per titolo

R5: Lettura da file

Si vuole leggere riviste e articoli da un file testuale come il seguente, con il metodo **void letturaFile (String fileName)**.

Si saltano le linee errate.
Un articolo è attribuito alla rivista precedente; se manca, è ignorato.

```
rivista, corriere scientifico, 1.5
          articolo, nanotecnologie, 2012, faggio
          articolo, alcuni problemi aperti, 2014, quercia
rivista, corriere scientifico, 3.5
          articolo, macrocosmo e microcosmo, acero
rivista, fisica e metafisica, 1.2
          articolo, materia oscura, 2012, quercia, faggio
rivista, universo in espansione, 2.4
          articolo, big bang aggiornato, 2010, quercia, acero
          articolo, alcuni problemi aperti, 2014, quercia
```

Attenzione al punto decimale

Test

```
Pubblicazioni p = new Pubblicazioni();
p.addRivista("corriere scientifico", 1.5);
Rivista cs = p.getRivista("corriere scientifico");
Rivista fm = p.addRivista("fisica e metafisica", 1.2);
Rivista uie = p.addRivista("universo in espansione", 2.4);
try {p.addRivista("corriere scientifico", 3.5);}catch(Exception e)
            {System.out.println(e.getMessage());}
uie.addArticolo("big bang aggiornato", Year.of(2014), "quercia", "acero");
fm.addArticolo("materia oscura", Year.of(2010), "quercia", "faggio");
cs.addArticolo("nanotecnologie", Year.of(2012), "faggio", "acero");
cs.addArticolo("alcuni problemi aperti", Year.of(2012), "quercia");
try {cs.addArticolo("alcuni problemi aperti", Year.of(2014),
    "quercia");}catch(Exception e) {System.out.println(e.getMessage());}
```

Test

```
List<Articolo> articoli = cs.getArticoli(); //ordinati per anno e titolo
for (Articolo a: articoli)
    System.out.println(a.getAnno() + " " + a.getTitolo());

Autore quercia = p.getAutore("quercia");
articoli = quercia.getArticoli();
for (Articolo a: articoli)
    System.out.println(a.getAnno() + " " +
            a.getTitolo());
List<Rivista> riviste = p.getRiviste();
for (Rivista r: riviste)
    System.out.println(r.getTitolo() + " "
            + r.getIF());
for (Autore a: p.getAutori())
    System.out.println(a.getNome() + " "
            + a.getIF());
```

2012 alcuni problemi aperti
2012 nanotecnologie

2010 materia oscura
2012 alcuni problemi aperti
2014 big bang aggiornato

universo in espansione 2.4
corriere scientifico 1.5
fisica e metafisica 1.2

quercia 5.1
acero 3.9
faggio 2.7

Test

```
sortedMap<Year, Long> mappaArticoliPerAnno = p.articoliPerAnno();
System.out.println("mappaArticoliPerAnno: " + mappaArticoliPerAnno);
Rivista r =  p.rivistaMaxArticoli();
System.out.println("rivistaMaxArticoli: " + r.getTitolo() + " " +
              r.getArticoli().size());
SortedMap<String, Long> mappaArticoliPerAutore =
p.articoliPerAutore();
System.out.println("mappaArticoliPerAutore: " + mappaArticoliPerAutore);
SortedMap<Long, Set<String>> mappaAutoriPerNarticoli =
       p.autoriPerNarticoli();
System.out.println("mappaAutoriPerNarticoli: " + mappaAutoriPerNarticoli);
```

n. articoli per anno
rivista con n. max di articoli
n. articoli per autore
autori per n. articoli

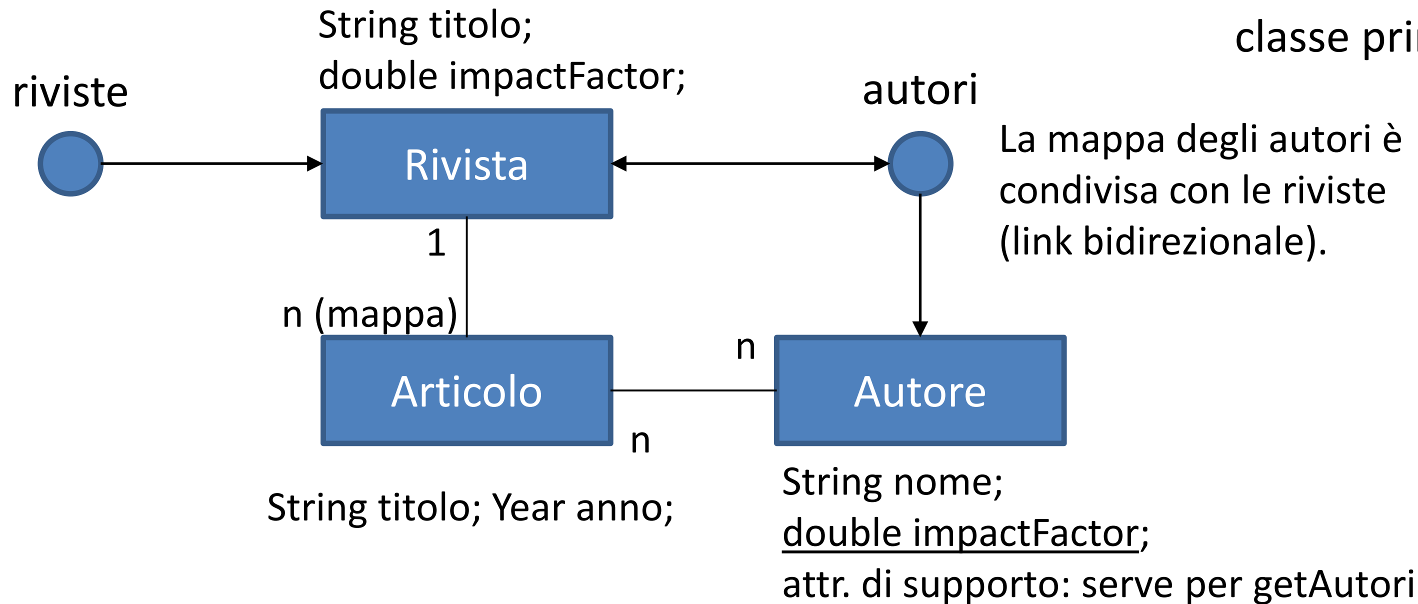
```
mappaArticoliPerAnno: {2010=1, 2012=2, 2014=1}
rivistaMaxArticoli: corriere scientifico 2
mappaArticoliPerAutore: {acero=2, faggio=2, quercia=3}
mappaAutoriPerNarticoli: {3=[quercia], 2=[acero, faggio]}
```

Lettura da file

```
p = new Pubblicazioni();
System.out.println("lettura file");
p.letturaFile("pubblicazioni.txt");
riviste = p.getRiviste();
for (Rivista r1: riviste)
    System.out.println(r1.getTitolo() + " " + r1.getIF());
for (Autore a:  p.getAutori())
    System.out.println(a.getNome() + " " + a.getIF());
```

universo in espansione 2.4
corriere scientifico 1.5
fisica e metafisica 1.2
quercia 7.5
acero 3.9
faggio 2.7

Progetto



Struttura dati di Pubblicazioni
Map<String, Autore> **autori**;
Map<String, Rivista> riviste;
Nota: gli articoli sono inseriti dalle riviste, che quindi devono inserire gli autori (se nuovi) nella mappa **autori**; una rivista riceve la mappa **autori** tramite il costruttore.

Rivista

```
public class Rivista {
private String titolo; private double impactFactor;
private Map<String, Articolo> articoli = new HashMap<>();
private Map<String, Autore> autori; //mappa globale per aggiungere nuovi autori
Rivista (String titolo, double impactFactor, Map<String, Autore> autori) {
    this.titolo = titolo; this.impactFactor = impactFactor; this.autori =
autori;}

public void addArticolo(String titolo, Year anno, String... nomiAutori)
throws Exception {
Articolo articolo = articoli.get(titolo);
if (articolo != null)  throw new Exception("articolo duplicato in: " + titolo
        + " " + this.titolo);
articolo = new Articolo(this, titolo, anno);
for (String nome:nomiAutori) {
        Autore autore = autori.get(nome);
        if (autore == null) {autore = new Autore(nome);
                autori.put(nome, autore);}
        autore.addArticolo(articolo); articolo.addAutore(autore);
}
articoli.put(titolo, articolo);}
```

Rivista

```
public List<Articolo> getArticoli() {
return articoli.values().stream()
.sorted(comparing(Articolo::getAnno)
        .thenComparing(Articolo::getTitolo))
.collect(toList());
}
```

Autore

```
public class Autore {
private String nome;
private List<Articolo> articoli = new ArrayList<>();
private double impactFactor = 0; // attributo di supporto, utile
                                    // per i confronti
public Autore(String nome) {this.nome = nome;}
void addArticolo(Articolo articolo) {
        articoli.add(articolo);
        impactFactor += articolo.getRivista().getIF();
}

public List<Articolo> getArticoli() {
return articoli.stream()
.sorted(comparing(Articolo::getAnno).thenComparing(Articolo::getTitolo))
.collect(toList());
}

public Double getIF() {return impactFactor;}
public String getNome() {return nome;}
long getNArticoli() {return articoli.size();} //per confronti
```

Articolo

```
public class Articolo {
private Rivista rivista;
private String titolo; private Year anno; // di pubblicazione
private ArrayList<Autore> autori = new ArrayList<>();
Articolo(Rivista rivista,String titolo, Year anno) {
        this.rivista = rivista;
        this.titolo = titolo; this.anno = anno;
}
void addAutore(Autore autore) {autori.add(autore);}
```

Pubblicazioni

```
import java.time.Year;
import java.util.*;
import java.io.*;
import static
java.util.Comparator.*;
import static
java.util.stream.Collectors.*;
```

```
public class Pubblicazioni {
private Map<String, Autore> autori = new HashMap<>();
private Map<String, Rivista> riviste = new HashMap<>();

public Rivista addRivista(String titolo,
double impactFactor) throws  Exception {
if (riviste.containsKey(titolo))
        throw new Exception ("rivista duplicata - " + titolo);
Rivista r = new Rivista(titolo, impactFactor, autori);
riviste.put(titolo, r); return r;
}
public Rivista getRivista(String titolo) throws  Exception {
Rivista r = riviste.get(titolo);
if (r == null) throw new Exception ("rivista mancante - " + titolo);
return r;}

public Autore getAutore(String nome) throws  Exception {
Autore a = autori.get(nome);
if (a == null) throw new Exception ("autore mancante - " + nome);
return a;}
```

Pubblicazioni

```
public List<Rivista>getRiviste() {
//riviste ordinate per impact factor decr e titolo
return riviste.values().stream()
.sorted(comparing(Rivista::getIF,
       reverseOrder()).thenComparing(Rivista::getTitolo))
.collect(toList());
}
public List<Autore>getAutori() {
//autori ordinati per impact factor decr e nome
return autori.values().stream()
.sorted(comparing(Autore::getIF,
       reverseOrder()).thenComparing(Autore::getNome))
.collect(toList());
}
```

Pubblicazioni

```
public SortedMap<Year, Long> articoliPerAnno () {
return riviste.values().stream().flatMap(r ->
r.getArticoli().stream())
       .collect(groupingBy(Articolo::getAnno,
       TreeMap::new, counting()));
}
//rivista con più articoli
public Rivista rivistaMaxArticoli() {
//prima di orElse si ha un Optional<Rivista>
return riviste.values().stream()
       .max(comparing(r -> r.getArticoli().size())).orElse(null);
}
```

Pubblicazioni

```
//n. articoli per autore
public SortedMap<String, Long> articoliPerAutore () {
return autori.values().stream()
.collect(toMap(Autore::getNome, Autore::getNArticoli,
            (l1,l2) -> l1, TreeMap::new));
}
```

Si usa toMap con 4 par; l1 e l2 sono i valori long che andrebbero combinati a parità di chiave (che però è univoca).

```
// autori per n. articoli
public SortedMap<Long, TreeSet<String>> autoriPerNarticoli () {
return autori.values().stream()
            .collect(groupingBy(Autore::getNArticoli,
                        () -> new TreeMap<>(reverseOrder()),
                        mapping(Autore::getNome, toCollection(TreeSet::new))));
}
```

Pubblicazioni

```
public void letturaFile(String fileName) {
try(BufferedReader in = new BufferedReader(new FileReader(fileName));)
{String line; Rivista rivista = null;
while ((line = in.readLine()) != null)
{ try {Scanner sc = new Scanner(line); sc.useDelimiter(",\\s*"); sc.useLocale(Locale.US);
    String l = sc.next(); l = l.trim();
    if (l.equalsIgnoreCase("rivista"))
    {String titolo = sc.next(); double impactFactor =  sc.nextDouble();
            rivista = addRivista(titolo, impactFactor);
    } else if (l.equalsIgnoreCase("articolo")) {
            String titolo = sc.next(); String anno = sc.next();
            ArrayList<String> array = new ArrayList<String>();
            while (sc.hasNext()) array.add(sc.next());
            String[] nomiAutori = new String[array.size()]; array.toArray(nomiAutori);
            if (rivista != null) rivista.addArticolo(titolo, Year.of(Integer.parseInt(anno)), nomiAutori);
    }sc.close();
    } catch (Exception e) {System.out.println(e.getMessage());}
}
} catch (IOException e) {System.out.println(e.getMessage());}}
```

per il . decimale

Gestione ordini

Il programma permette ad un distributore di registrare i prodotti trattati, di accettare ordini dai clienti, di preparare ordini per i fornitori e di gestire lo stato degli ordini.

Un ordine si compone di linee d'ordine e una linea si riferisce ad un prodotto.

I metodi principali sono offerti dalla classe GestioneOrdini.

R1: inserimenti

Il metodo void **addProdotti** (String s) throws Exception

inserisce i prodotti (nome, prezzo *int*), segnalando la duplicazione del nome.

Come si vede dall'esempio l'elenco dei prodotti è dato mediante una stringa.

g.**addProdotti**("divano2pT,200,divano2pP,400,poltronaT,150,poltronaP,250, libreria,300,scaffale,120");

Il metodo void **addFornitore** (String nomeF, String prodotti) throws Exception

inserisce un fornitore con i prodotti trattati, segnalando la mancanza di un prodotto.

Come si vede dall'esempio l'elenco dei nomi dei prodotti trattati è dato mediante una stringa.

g.**addFornitore**("AlfaMobili", "divano2pT,poltronaT,libreria");

Nota: le stringhe prodotti contengono nomi di prodotti separati da virgole.

R2: Ordini clienti

Il metodo

void **addOrdineCliente** (String codice, String cliente, String *prodotti*) throws Exception

inserisce un ordine cliente; il codice è univoco, quindi è segnalata la duplicazione.

Un ordine cliente si compone di linee, una per prodotto; i prodotti sono distinti (altrimenti è segnalata la duplicazione). Ciascuna linea punta al prodotto di cui è dato il nome nella stringa *prodotti*.

g.**addOrdineCliente**("quercia:1", "quercia", "divano2pT,poltronaT");

//importo = 350

Note:

quercia:1 è il codice dell'ordine;

Si segnala un'eccezione se il codice esiste già, i prodotti non sono distinti o non sono registrati nel sistema.

Il metodo int **getImportoOC** throws Exception

dà l'importo in base al codice, con segnalazione di errore se non esiste l'ordine.

R3: Ordini fornitore

Il metodo void **addOrdineFornitore** (String codice, String fornitore, String linee) throws Exception

inserisce un ordine fornitore; il codice è univoco, quindi è segnalata la duplicazione.

g.**addOrdineFornitore**("alfa:1", "AlfaMobili",

"quercia:1 divano2pT, faggio:1 poltronaP"); //importo = 450

Un ordine fornitore aggrega linee di ordini cliente: ogni linea è indicata dal codice dell'ordine cliente e dal nome del prodotto. L'ordine fornitore mostrato aggrega quindi la linea divano2pT dell'ordine cliente quercia:1 e la linea poltronaP dell'ordine cliente faggio:1.

Il metodo lancia un'eccezione se il fornitore non fornisce i prodotti corrispondenti alle linee e se le linee indicate sono inesistenti (errore nel codice ordine cliente o nel nome del prodotto) o sono già state aggregate in un altro ordine fornitore.

Il metodo int **getImportoOF** throws Exception

dà l'importo in base al codice, con segnalazione di errore.

R4: Consegna

Il metodo void **consegnaFornitore** (String codice) throws Exception

porta nello stato consegnato l'ordine fornitore e le linee associate.

Verifica lo stato degli ordini cliente che non sono ancora nello stato consegnato: se tutte le linee sono nello stato consegnata, porta l'ordine cliente nello stato consegnato.

Dà errore se il codice è errato o l'ordine fornitore è già nello stato consegnato.

Il metodo List<String> **getOrdiniConsegnati** dà l'elenco dei codici (ordinati alfabeticamente) degli ordini consegnati.

Stati

Gli stati di un ordine cliente sono i seguenti:
inserito; stato iniziale
consegnato

Gli stati di un ordine fornitore sono:
inserito
consegnato

Gli stati di una linea sono:
inserita
inOrdine
consegnata

R5: Statistiche

SortedMap <String, Long> **nOrdiniCliente** ()
dà il numero di ordini per cliente (nome) con nomi ordinati.

int **maxOrdineCliente** ()
dà l'importo dell'ordine cliente più elevato

SortedMap <Integer, TreeSet<String>> **clientiTotaleOrdini**();
raggruppa i clienti in base all'importo totale dei loro ordini; gli importi sono decrescenti e dei clienti si danno i nomi distinti e ordinati.

SortedMap <Long, TreeSet<String>> **prodottiNLinee** ()
raggruppa i prodotti per n. di linee (degli ordini cliente); i n. di linee sono decrescenti e dei prodotti si danno i nomi distinti e ordinati

Test

```
GestioneOrdini g = new GestioneOrdini();
g.addProdotti("divano2pT,200,divano2pP,400,poltronaT,150,poltronaP,250,
libreria,300,scaffale,120");
// registra i prodotti nome prezzo (intero), segnala duplicazione
g.addFornitore("AlfaMobili", "divano2pT,poltronaT,libreria");
// registra un fornitore con i prodotti trattati, segnala mancanza prodotto
g.addFornitore("BetaMobili ", "divano2pP,poltronaP");
g.addFornitore("TuttoMobili",
        "divano2pT,poltronaT,divano2pP,poltronaP,libreria,scaffale");
```

Test

g.**addOrdineCliente**("quercia:1", "quercia", "divano2pT,poltronaT"); //<u>importo = 350</u>

g.addOrdineCliente("faggio:1", "faggio", "poltronaT,poltronaP,libreria"); //<u>importo = 700</u>

g.**addOrdineFornitore**("alfa:1", "AlfaMobili", "quercia:1 divano2pT, faggio:1 poltronaP"); //<u>importo = 450</u>

g.addOrdineFornitore("tutto:1", "TuttoMobili", "quercia:1 poltronaT, faggio:1 libreria");

g.**consegnaFornitore**("alfa:1");

g.consegnaFornitore("tutto:1");

g.addOrdineCliente("quercia:2", "quercia", "libreria,poltronaT"); //<u>importo = 450</u>

// <u>letture</u>

System.out.println(g.getStatoOC("quercia:1"));

System.out.println(g.getStatoOC("faggio:1"));

System.out.println(g.getImportoOC("quercia:1"));

System.out.println(g.getImportoOF("alfa:1"));

```
stato di quercia:1 consegnato
stato di faggio:1 inserito
importo di quercia:1 350
importo di faggio:1 450
```

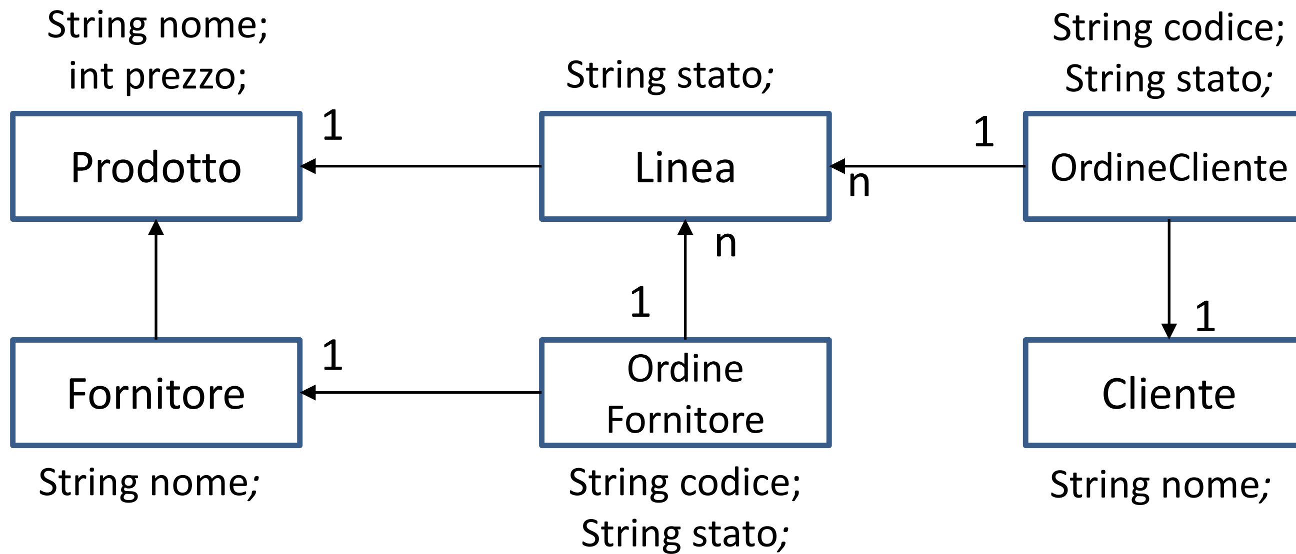
Test

```
// statistiche
SortedMap<String, Long>  mappaNOrdiniCliente = g.nOrdiniCliente();
System.out.println("mappaNOrdiniCliente " + mappaNOrdiniCliente);
int maxOrdineCliente = g.maxOrdineCliente();
System.out.println("maxOrdineCliente " + maxOrdineCliente);
SortedMap<Integer, SortedSet<String>> mappaClientiTotaleOrdini =
            g.clientiTotaleOrdini();
System.out.println("mappaClientiTotaleOrdini " + mappaClientiTotaleOrdini);
SortedMap<Long, SortedSet<String>> mappaProdottiNLinee = g.prodottiNLinee();
System.out.println("mappaProdottiNLinee " + mappaProdottiNLinee);
```

```
mappaNOrdiniCliente {faggio=1, quercia=2}
maxOrdineCliente 700
mappaClientiTotaleOrdini {800=[quercia], 700=[faggio]}
mappaProdottiNLinee {3=[poltronaT], 2=[libreria], 1=[divano2pT, poltronaP], 0=[divano2pP, scaffale]}
```

Analisi e Progetto

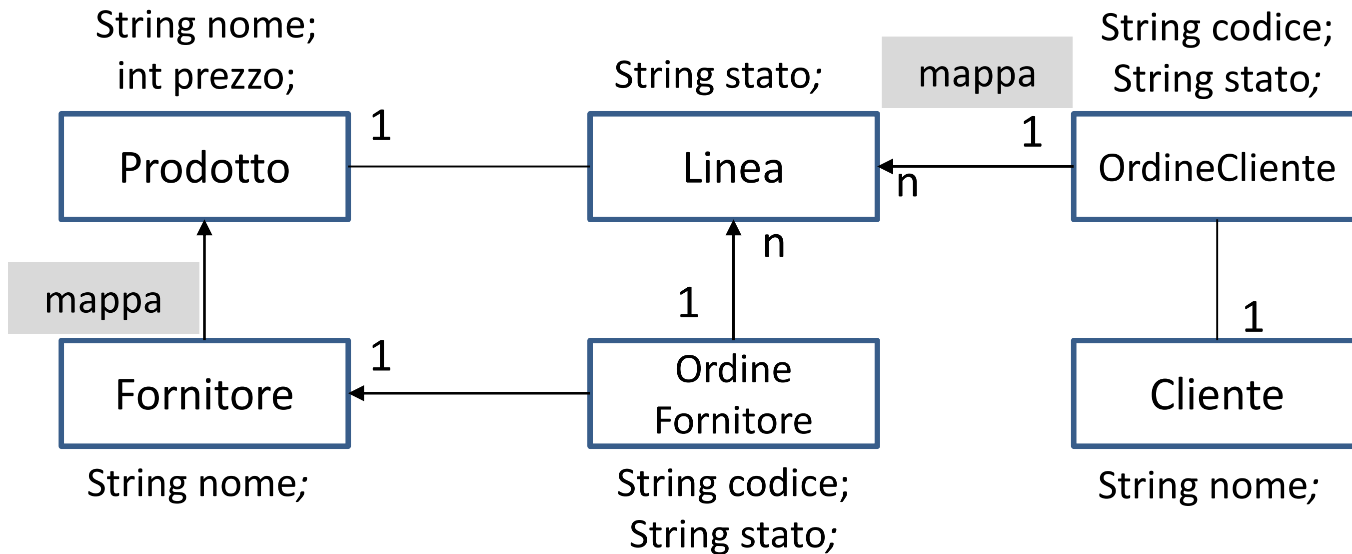
Attributi essenziali



Occorre stabilire gli attributi associativi e gli eventuali attributi di supporto (non essenziali).

Analisi e Progetto

Attributi essenziali



prodottiNLinee: è utile rendere bidirezionale la relazione Linea-Prodotto
maxOrdineCliente: conviene inserire l'attributo importo in OrdineCliente
getImportoOF: anche in OrdineFornitore
clientiTotaleOrdini: conviene inserire l'attributo importoTotale in Cliente insieme con la lista degli ordini cliente (il size dà nOrdiniCliente)

Prodotto

```
class Prodotto {
private String nome; String getNome() {return nome;}
private int prezzo; int getPrezzo() {return prezzo;}
Prodotto(String nome, int prezzo) {
        this.nome = nome; this.prezzo = prezzo;}
List<Linea> linee = new ArrayList<>();
        void addLinea (Linea l) {linee.add(l);}
        public int getNLinee() {return linee.size();}
}
```

Fornitore

```
class Fornitore {
private String nome; String getNome() {return nome;}
private Map<String, Prodotto> prodotti;
Prodotto getProdotto (String nome) {return prodotti.get(nome);}

Fornitore (String nome, Map<String, Prodotto> prodottiFornitore)
        {this.nome = nome; prodotti = prodottiFornitore;}
}
```

Cliente

```
class Cliente {
private String nome; String getNome() {return nome;}
Cliente(String nome) {this.nome = nome;}
int totaleOrdini = 0; int getTotaleOrdini() {return totaleOrdini;}
List<OrdineCliente> ordini = new ArrayList<>();
void addOrdine(OrdineCliente ordine) {
        //aggiorna totaleOrdini
        ordini.add(ordine);
        totaleOrdini += ordine.getImporto();
}
int getNOrdini() {return ordini.size();}
}
```

Linea

```
class Linea {
private Prodotto prodotto; Prodotto getProdotto() {return prodotto;}
Linea(Prodotto prodotto) {this.prodotto = prodotto;}
private String stato = "inserita"; String getStato() {return stato;}
            void inOrdine() {stato = "inOrdine";}
            void consegna() {stato = "consegnata";}
}

// lo stato è gestito internamente
```

OrdineCliente

```
class OrdineCliente {
private String codice; String getCodice() {return codice;}
private Cliente cliente; Cliente getCliente() {return cliente;}
private Map<String, Linea> linee;
        ArrayList<Linea> getLinee() {return new ArrayList<>(linee.values());}
OrdineCliente(String codice, Cliente cliente, Map<String, Linea> lineeOC) {
        //calcola l'importo e passa l'ordine al cliente
        this.codice = codice; this.cliente = cliente; linee = lineeOC;
        for (Linea l: linee.values()) importo += l.getProdotto().getPrezzo();
        cliente.addOrdine(this);}
private int importo = 0; int getImporto() {return importo;}
Linea getLinea(String nomeProdotto) {return linee.get(nomeProdotto);}
private String stato = "inserito"; String getStato() {return stato;}
void verificaStato() {
if (stato.equals("consegnato")) return;
for (Linea l:linee.values()) if(!l.getStato().equals("consegnata")) return;
stato = "consegnato";}
//per statistiche
String getNomeCliente() {return cliente.getNome();}
}
```

il costruttore riceve tutte le linee

OrdineFornitore

```
class OrdineFornitore {
private String codice; String getCodice() {return codice;}
private Fornitore fornitore; Fornitore getFornitore() {return fornitore;}
private List<Linea> linee;
OrdineFornitore (String codice, Fornitore fornitore, List<Linea> lineeOF) {
        //calcola l'importo e cambia stato alle linee
        this.codice = codice; this.fornitore = fornitore; linee = lineeOF;
        for (Linea l:linee) {l.inOrdine(); importo += l.getProdotto().getPrezzo();}
}
private int importo = 0; int getImporto() {return importo;}
private String stato = "inserito"; String getStato() {return stato;}
void consegna() {stato = "consegnato"; for (Linea l: linee) l.consegna();}
}
```

il costruttore riceve tutte le linee

class GestioneOrdini

```
public class GestioneOrdini {
private Map<String,Prodotto> prodotti = new HashMap<>();
private Map<String, OrdineCliente> ordiniCliente = new HashMap<>();
private Map<String, OrdineFornitore> ordiniFornitore = new HashMap<>();
private Map<String, Fornitore> fornitori = new HashMap<>();
private Map<String, Cliente> clienti = new HashMap<>();
```

addProdotti

```
public void addProdotti (String s) throws Exception {
// E: prodotto duplicato
// sequenza nome,prezzo separati da virgola
Scanner sc = new Scanner(s); sc.useDelimiter(",");
while (sc.hasNext()) {
        String nome = sc.next(); int prezzo = sc.nextInt();
        if(prodotti.containsKey(nome))
          {sc.close(); throw new Exception("prodotto duplicato " + nome); }
        prodotti.put(nome, new Prodotto(nome, prezzo));
}
sc.close();
}
```

aggiunge un prodotto fino ad un'eventuale eccezione

@SuppressWarnings("resource") evita il controllo sulla chiusura dello scanner

Oppure si può seguire un approccio transazionale: prima si controlla (con una mappa interna) e poi si aggiunge a prodotti la mappa interna con il metodo putAll.

addFornitore

```
public void addFornitore(String nomeF, String listaProdotti) throws Exception {
//E: fornitore duplicato, prodotto mancante, duplicati non controllati
Fornitore f = fornitori.get(nomeF);
if (f != null) throw new Exception("fornitore duplicato " + nomeF);
Map<String, Prodotto> prodottiFornitore = new HashMap<>();
Scanner sc = new Scanner(listaProdotti); sc.useDelimiter(",");
while (sc.hasNext()) {
        String nomeP = sc.next(); Prodotto p = prodotti.get(nomeP);
        if (p == null) throw new Exception("prodotto inesistente " + nomeP);
        prodottiFornitore.put(p.getNome(), p);
}
sc.close();
f = new Fornitore (nomeF, prodottiFornitore); //aggiunge se corretto
fornitori.put(nomeF, f);
}
```

il fornitore è istanziato dopo i controlli e riceve la mappa dei prodotti.

addOrdineCliente

```
public void addOrdineCliente (String codice, String cliente, String listaProdotti) throws
Exception {//E: codice duplicato, nome prodotto duplicato o inesistente
OrdineCliente ordine = ordiniCliente.get(codice);
if (ordine != null) throw new Exception("codice duplicato " + codice);
Map<String, Linea> lineeOC = new HashMap<>();
Scanner sc = new Scanner(listaProdotti); sc.useDelimiter(",");
while (sc.hasNext()) {
        String nomeProdotto = sc.next(); Prodotto p = this.prodotti.get(nomeProdotto);
        if(p == null) throw new Exception("prodotto inesistente " + nomeProdotto);
        if (lineeOC.containsKey(nomeProdotto))
                throw new Exception("prodotto duplicato in ordine " + nomeProdotto);
        lineeOC.put(nomeProdotto, new Linea(p));}
Cliente c = clienti.get(cliente);
if (c == null) {c = new Cliente (cliente); clienti.put(cliente, c);}
ordine = new OrdineCliente(codice, c, lineeOC);
//passa le linee ai prodotti
for (Linea l: lineeOC.values()) l.getProdotto().addLinea(l);
ordiniCliente.put(codice, ordine);
}
```

dopo i controlli sono istanziati il cliente, se nuovo, e l'ordine cliente con tutte le linee (passate poi ai prodotti)

addOrdineFornitore

```
public void addOrdineFornitore (String codice, String fornitore, String listaLinee) throws
Exception { //E: fornitore non esiste, codice duplicato, ordine cliente non esiste,
//linea non esiste, linea già in ordine
Fornitore f = fornitori.get(fornitore); if (f == null) throw new Exception("fornitore inesistente " + fornitore);
//"quercia:1 0, faggio:1 0" coppie (separate da ,spazio) di codice ordine e nome prodotto
if (ordiniFornitore.containsKey(codice)) throw new Exception("codice duplicato " + codice);
List<Linea> lineeOF = new ArrayList<>();
Scanner sc = new Scanner(listaLinee); sc.useDelimiter(",\\s*");
while (sc.hasNext()) {String linea = sc.next(); String[] elementi = linea.split("\\s+");
        String idOrdineCliente = elementi[0]; String nomeProdotto = elementi[1];
        OrdineCliente ordineCliente = ordiniCliente.get(idOrdineCliente);
        if(ordineCliente == null) throw new Exception("ordine inesistente " + idOrdineCliente);
        Linea l = ordineCliente.getLinea(nomeProdotto);
        if (l == null) throw new Exception("prodotto " + nomeProdotto + " mancante in ordine " + idOrdineCliente);
        if (l.getStato().equals("inOrdine")) throw new Exception("linea già in ordine " + linea);
        lineeOF.add(l);}
OrdineFornitore ordineF = new OrdineFornitore(codice, f, lineeOF);
ordiniFornitore.put(codice, ordineF);
}
```

l'ordine è istanziato con le linee

consegnaFornitore

```
public void consegnaFornitore (String codice) throws Exception{
//E: codice errato, ordine già consegnato
//cambia stato ordine fornitore e linee
//verifica cambio stato ordini cliente
OrdineFornitore ordineF = ordiniFornitore.get(codice);
if (ordineF == null) throw new Exception("ordine inesistente " + codice);
if (ordineF.getStato().equals("consegnato")) throw new Exception("stato errato " + codice);
ordineF.consegna();
for (OrdineCliente ordineCliente: ordiniCliente.values())
        ordineCliente.verificaStato();
}
```

letture stato e importo

```
public String getStatoOC(String codice) throws Exception{
OrdineCliente o = ordiniCliente.get(codice);
if (o == null) throw new Exception("ordine inesistente " + codice);
return o.getStato();
}
public int getImportoOC(String codice) throws Exception{
OrdineCliente o = ordiniCliente.get(codice);
if (o == null) throw new Exception("ordine inesistente " + codice);
return o.getImporto();
}
public int getImportoOF(String codice) throws Exception{
OrdineFornitore o = ordiniFornitore.get(codice);
if (o == null) throw new Exception("ordine inesistente " + codice);
return o.getImporto();
}
```

statistiche

```
public SortedMap<String, Integer>  nOrdiniCliente() {
return clienti.values().stream()
.collect(toMap(Cliente::getNome, Cliente::getNOrdini,
            (a, b) -> a, () -> new TreeMap<>()));
}

public int maxOrdineCliente() {
return ordiniCliente.values().stream()
.mapToInt(OrdineCliente::getImporto)
.max().orElse(0);
}
```

soluzione con l'uso del collettore toMap (getter per chiave, getter per valore, merge function e supplier della mappa).

statistiche

```
public SortedMap<Integer, SortedSet<String>> clientiTotaleOrdini () {
return clienti.values().stream()
.collect(groupingBy(Cliente::getTotaleOrdini,
            () -> new TreeMap<>(reverseOrder()),
            mapping(Cliente::getNome, toCollection(TreeSet::new))));
}

//la versione seguente non usa getTotaleOrdini di Cliente
public SortedMap<Integer, SortedSet<String>> clientiTotaleOrdini1 () {//senza supporti
return ordiniCliente.values().stream()
.collect(groupingBy(OrdineCliente::getNomeCliente,
            summingInt(OrdineCliente::getImporto)))
.entrySet().stream()
.collect(groupingBy(e -> e.getValue(),
            () -> new TreeMap<>(reverseOrder()),
            mapping(e -> e.getKey(), toCollection(TreeSet::new))));
}
```

servono due operazioni:

1. contare gli ordini per cliente (nome);
2. raggruppare i nomi dei clienti per n. di ordini

statistiche

```
public SortedMap<Integer, SortedSet<String>> prodottiNLinee () {
return prodotti.values().stream()
.collect(groupingBy(Prodotto::getNLinee,
            () -> new TreeMap<>(reverseOrder()),
            mapping(Prodotto::getNome, toCollection(TreeSet::new))));
}
```

Bank System

Si sviluppi un programma che fornisca servizi bancari.

Tutte le classi si trovano nel package BankServices.

La classe MainClass nel package Main presenta esempi di uso dei metodi principali.

La JDK documentation si trova sul server locale.

R1

Il costruttore della classe **Bank** riceve come argomento una stringa che rappresenta il nome di una banca; il metodo **getName()** restituisce questo nome.

Le date di tutte le operazioni sono rappresentate mediante un intero da 1 a 365 corrispondente al giorno dell'anno corrente.

Ogni conto corrente aperto presso una banca è rappresentato dalla classe **Account**.

Il metodo **createAccount()** permette di aprire un nuovo conto corrente, e riceve tre argomenti: il nome del correntista, la data, l'ammontare del versamento iniziale; il metodo restituisce un intero corrispondente al numero di conto corrente creato (i numeri partono da 1 e vengono incrementati di 1 ad ogni creazione); l'apertura di un conto costituisce la prima operazione su di esso.

Il metodo **getAccount()** riceve un numero di conto corrente e restituisce l'oggetto di tipo **Account** corrispondente; se il numero del conto non è tra quelli creati, solleva l'eccezione **InvalidCode**.

R1

Il metodo **deposit()** permette di effettuare un versamento su di un conto corrente; riceve come argomenti il numero del conto, la data del versamento e l'importo da versare; se il numero del conto non è tra quelli creati, solleva l'eccezione **InvalidCode** ; se la data indicata precede quella dell'ultima operazione sul conto, il versamento viene effettuato nella stessa data dell'ultima operazione.

Il metodo **withdraw()** permette di effettuare un prelievo da un conto corrente; riceve come argomenti il numero del conto, la data del prelievo e l'importo da prelevare; se il numero del conto non è tra quelli creati, solleva l'eccezione **InvalidCode**; se l'importo supera l'ammontare del saldo corrente, solleva l'eccezione **InvalidValue**; se la data indicata precede quella dell'ultima operazione sul conto, il prelievo viene effettuato nella stessa data dell'ultima operazione.

R1

Il metodo **transfer()** permette di effettuare un bonifico da un conto corrente verso un altro conto corrente della stessa banca; riceve come argomenti il numero del conto ordinante, il numero del conto beneficiario, la data e l'importo; se i numeri dei conti non sono tra quelli creati, solleva l'eccezione **InvalidCode**; se l'importo supera l'ammontare del saldo sul conto ordinante, solleva l'eccezione **InvalidValue**; le date dell'operazione sul conto beneficiario e sul conto ordinante vengono stabilite con gli stessi criteri dei prelievi e dei versamenti; in ogni caso la data dell'operazione sul conto beneficiario deve essere maggiore o uguale a quella sul conto ordinante.

Il metodo **deleteAccount()** permette di chiudere un conto corrente prelevando tutto il denaro depositato; riceve come argomenti il numero del conto e la data, restituendo l'**Account** chiuso; se il numero del conto non è tra quelli creati, solleva l'eccezione **InvalidCode**; se la data indicata precede quella dell'ultima operazione sul conto, la chiusura viene effettuato nella stessa data dell'ultima operazione.

R2

La classe astratta **Operation** rappresenta una generica operazione effettuata su di un conto corrente.

Le operazioni possibili sono di due tipi: versamento (rappresentato dalla classe **Deposit**) e prelievo (rappresentato dalla classe **Withdrawal**).

Entrambe le classi implementano il metodo **toString()**, che restituisce una stringa costituita da: la data dell'operazione, una virgola, l'importo seguito dal segno + o – a seconda che si tratti di un versamento o di un prelievo, senza spazi intermedi (Esempi: 5,500.5+ 41,158.0-).

L'apertura di un conto corrente comporta implicitamente un versamento, mentre la sua chiusura comporta implicitamente un prelievo.

Operation

```
public abstract class Operation {
int date; double value;

public Operation(int d, double v) {date=d; value=v;}

public int getDate() {return date;}

public double getValue() {return value;}

public abstract String toString();

}
```

R3

La classe **Account** implementa il metodo **toString()**, che restituisce una stringa costituita da: il numero di conto corrente, il nome del correntista, la data dell'ultima operazione, il saldo attuale, tutti separati da una virgola e senza spazi intermedi (Esempio: 4,Paul,35,522.3).

La classe fornisce inoltre i seguenti metodi:

getMovements() restituisce la lista di tutte le operazioni effettuate, ordinate per date decrescenti;

getDeposits() restituisce la lista di tutti i versamenti effettuati, ordinati per importi decrescenti;

getWithdrawals() restituisce la lista di tutti i prelievi effettuati, ordinati per importi decrescenti.

R4

I seguenti metodi della classe **Bank** forniscono informazioni sui conti correnti attualmente aperti presso la banca (esclusi quindi quelli che sono stati chiusi) :

getTotalDeposit() restituisce l'ammontare di tutto il denaro attualmente depositato presso la banca (somma dei saldi di ogni conto corrente);

getAccounts() restituisce la lista di tutti i conti correnti attualmente aperti, ordinati per numero di conto crescente;

getAccountsByBalance() riceve gli estremi di un intervallo di importi, e restituisce la lista dei conti correnti con un saldo attuale compreso nell'intervallo, ordinati per valori di saldo decrescenti;

getPerCentHigher() riceve un importo e restituisce la percentuale dei conti correnti con un saldo attuale non inferiore all'importo dato.

Main

```
public static void main(String[] args) {
Bank b1 = new Bank("Uncle-$crooge");
int c1 = b1.createAccount("John", 5, 500.5);
int c2 = b1.createAccount("Mary", 10,  1000.);
int c3 = b1.createAccount("John", 20,  800.);
 int c4 = b1.createAccount("Paul", 30, 252.4);
Account a1=null, a3=null;
try {
b1.deposit(c1, 7, 360.0);
b1.deposit(c4, 35, 270.0);
b1.withdraw(c3, 28, 350.0);
b1.withdraw(c2, 19, 350.0);
b1.withdraw(c3, 41, 158.0);
b1.transfer(c1, c3, 8, 400.0);
a1 = b1.getAccount(c1);
a3 = b1.deleteAccount(c3,50);
}
catch(InvalidCode ic) {ic.printStackTrace();}
catch(InvalidValue iv) {iv.printStackTrace();}
```

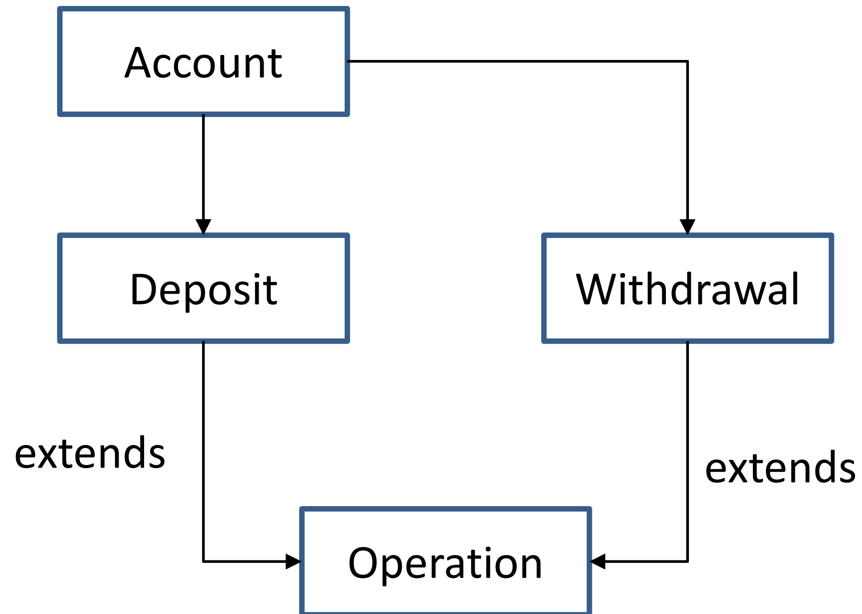
account: 1,John,8,460.5
movements:
8,400.0- trasferimento - l'8
7,360.0+ deposito il 7
5,500.5+ apertura il 5
account: 2,Mary,19,650.0
movements:
19,350.0- prelievo il 19
10,1000.0+ deposito il 10
account: 4,Paul,35,522.4
movements:
35,270.0+
30,252.4+
deleted account: 3,John,50,0.0
movements:
50,692.0- chiusura il 50
41,400.0+ trasferimento + 400 il 41(8)
41,158.0- prelievo il 41
28,350.0- prelievo il 28
20,800.0+ apertura il 20

Main

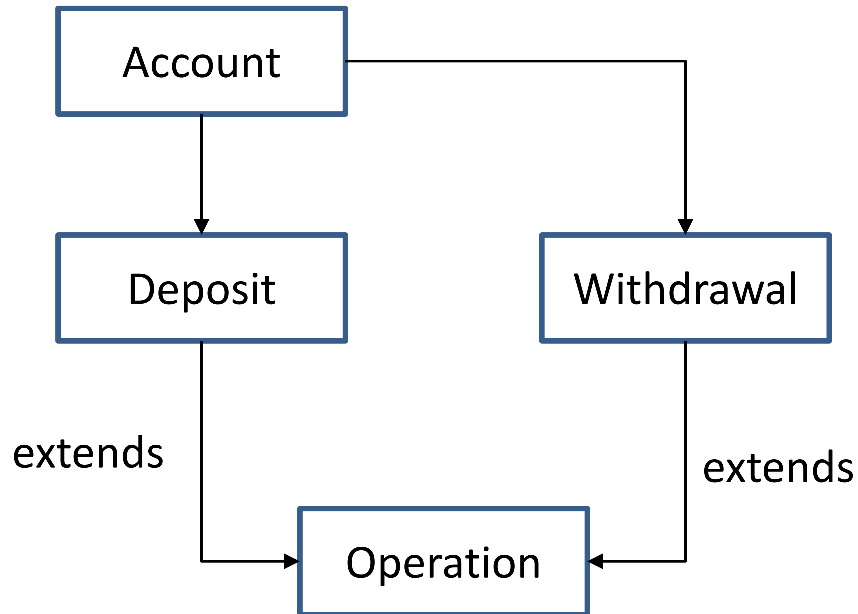
```
System.out.println("total deposit in the " + b1.getName() + " bank: " +
b1.getTotalDeposit());

System.out.println("accounts with balance higher than 500: " +
b1.getPerCentHigher(500) + " %");

System.out.println("accounts with balance in range 500..700 :");
for(Account a : b1.getAccountsByBalance(500, 700))
System.out.println(a);
```

```
total deposit in the Uncle-$crooge bank: 1632.9

accounts with balance higher than 500: 66.0 %

accounts with balance in range 500..700 :
2,Mary,19,650.0
4,Paul,35,522.4
```

Analisi e Progetto



Analisi e Progetto



Struttura dati nella classe principale
int code;
double totalDeposit;
Map<Integer,Account> accounts;

Account

```
public class Account {
private int code; private String customer; private double balance;
private int currentDate; private List<Deposit> deposits = new ArrayList<>();
private List<Withdrawal> withdrawals = new ArrayList<>();

public Account(int c, String name, int date, double initial) {
code=c; customer=name; balance=initial;currentDate=date;
Deposit d = new Deposit(date, initial); deposits.add(d);
}

public double getBalance() {return balance;}

public String toString() {
return code + "," + customer + "," + currentDate + "," + balance;
}
```

Account

```
public int addDeposit(int date, double value) {
if(date >= currentDate) currentDate = date;
deposits.add(new Deposit(currentDate, value));
balance += value; return currentDate;
}

public int addWithdraw(int date, double value) throws InvalidValue {
if(balance<value) throw new InvalidValue();
if(date >= currentDate) currentDate = date;
withdrawals.add(new Withdrawal(currentDate, value));
balance -= value; return currentDate;
}
```

Account

```
public List<Operation> getMovements() {
List<Operation> m = new ArrayList<>(deposits); m.addAll(withdrawals);
return m.stream()
        .sorted(comparing(Operation::getDate, reverseOrder()))
.collect(toList());}

public List<Deposit> getDeposits() {
return deposits.stream()
        .sorted(comparing(Deposit::getValue, reverseOrder()))
.collect(toList());}

public List<Withdrawal> getWithdrawals() {
return withdrawals.stream()
        .sorted(comparing(Withdrawal::getValue, reverseOrder()))
.collect(toList());}
```

Altre classi

```
public class Deposit extends Operation{
public Deposit(int d, double v) {super(d, v);}
public String toString() {return date + "," + value + "+" ;}}

public class Withdrawal extends Operation{
public Withdrawal(int d, double v) {super(d, v);}
public String toString() {return date + "," + value + "-" ;}}

public class InvalidCode extends Exception {}
```

Bank

```
public class Bank {
private String name; public String getName() {return name;}
int code = 0; double totalDeposit = 0.0;
Map<Integer,Account> accounts = new TreeMap<>();

public Bank(String n) {name=n;}

public int createAccount(String name, int date, double initial) {
code++; Account a = new Account(code, name, date, initial);
accounts.put(code, a); totalDeposit += initial; return code;}
public Account getAccount(int code) throws InvalidCode {
Account a = accounts.get(code); if(a==null) throw new InvalidCode(); return a;}

public Account deleteAccount(int code, int date) throws InvalidCode {
Account a = accounts.get(code); if(a==null) throw new InvalidCode();
double value = a.getBalance();
try {a.addWithdraw(date, value);}catch(InvalidValue iv) {}
accounts.remove(code); totalDeposit -= value; return a;}
```

Si può definire il metodo close in Account.

Bank

```
public void deposit(int code, int date, double value) throws InvalidCode {
Account a = accounts.get(code); if(a==null)throw new InvalidCode();
a.addDeposit(date, value); totalDeposit += value;}

public void withdraw(int code, int date, double value) throws InvalidCode, InvalidValue {
Account a = accounts.get(code); if(a==null)throw new InvalidCode();
a.addWithdraw(date, value); totalDeposit -= value;}

public void transfer(int fromCode, int toCode, int date, double value)
            throws InvalidCode, InvalidValue {
Account from = accounts.get(fromCode); if(from==null) throw new InvalidCode();
Account to = accounts.get(toCode); if(to==null) throw new InvalidCode();
int d = from.addWithdraw(date, value); to.addDeposit(d, value);}

public List<Account> getAccounts() {return new ArrayList<>(accounts.values());}

public double getTotalDeposit() {return totalDeposit;}
```

Bank

```
public List<Account> getAccountsByBalance(double low, double high) {
List<Account> acs = getAccounts();
return acs.stream()
            .filter(a -> a.getBalance() >= low && a.getBalance() <= high)
            .sorted(comparing(Account::getBalance, reverseOrder()))
.collect(toList());
}

public double getPerCentHigher(double min) {
List<Account> acs = getAccounts();
int s = acs.size();
long h = acs.stream()
            .filter(a -> a.getBalance() >= min)
.collect(counting());
return 100 * h/s;
}
```

Gestione Domande di Impiego

Il programma permette ad un'azienda di ricevere domande per posizioni scoperte. La classe principale è HandleApplications; le classi del programma si trovano nel package applications.

La classe di test (Test) contiene un esempio che illustra l'uso dei metodi principali; è utile leggere i requisiti guardando questa classe.

Le eccezioni lanciate dal programma sono di tipo ApplicationException.

R1: Inserimento di skills e positions

Il metodo **addSkills** inserisce un elenco di competenze dati i nomi. Lancia un'eccezione se trova una competenza duplicata.

Il metodo **addPosition** inserisce una posizione (dato il nome) con l'elenco delle competenze richieste (di cui sono dati i nomi). Lancia un'eccezione se la posizione è già stata inserita o se non trova una competenza richiesta tra quelle già inserite.

Il metodo **getSkill** fornisce un oggetto <u>Skill</u> dato il nome; se non lo trova tra quelli inseriti dà null.

Il metodo **getPositions** di Skill dà la lista delle posizioni che chiedono lo skill, ordinate alfabeticamente per nome. Il metodo **getPosition** fornisce un oggetto <u>Position</u> dato il nome; se non lo trova tra quelli inseriti dà null. Skill e Position hanno i getter **getName**.

Esempi

```
HandleApplications ha = new HandleApplications();
ha.addSkills("java", "c++", "javascript", "sql", "html", "uml", "bpmn", "sw design");
try {ha.addSkills("c#", "sql"); //sql duplicated
} catch (ApplicationException e) {System.out.println(e.getMessage());}
ha.addPosition("programmer", "java", "sql");
ha.addPosition("programmer1","java", "c++");
ha.addPosition("gui designer junior", "javascript", "html");
ha.addPosition("gui designer senior", "javascript", "java", "html");
ha.addPosition("sw team leader", "sw design", "uml", "bpmn", "sql");
try {ha.addPosition("programmer", "c++", "sql"); //programmer duplicated
        } catch (ApplicationException e) {System.out.println(e.getMessage());}
try {ha.addPosition("programmer2", "c", "sql"); //c undefined
        } catch (ApplicationException e) {System.out.println(e.getMessage());}
Skill javaSkill = ha.getSkill("java");
List<Position> positionsWithJava = javaSkill.getPositions();
System.out.println(positionsWithJava.get(0).getName()); //gui designer senior
Position programmerPos = ha.getPosition("programmer");
```

R2: Inserimento di richiedenti

Il metodo **addApplicant** inserisce un richiedente (dato il nome) con l'elenco delle sue capacità (capabilities). Un esempio di elenco è il seguente: "java:9,sql:7". Ogni capacità è costituita dal nome della competenza, dal separatore ':', e dal livello (compreso tra 1 e 10) posseduto dal richiedente. Le capacità sono separate da virgole. Il metodo lancia un'eccezione se il richiedente è già stato inserito, una competenza non è stata inserita, un livello non appartiene al range stabilito.

Il metodo **getCapabilities** fornisce l'elenco delle capacità di un richiedente (dato il nome) nel formato indicato sopra: le capacità sono però ordinate alfabeticamente. Lancia un'eccezione se non trova il richiedente tra quelli già inseriti. Se non ci sono capacità dà la stringa vuota.

Esempi

ha.addApplicant("john", "java:9,sql:7");
ha.addApplicant("david", "sql:8,java:7");
ha.addApplicant("mary", "javascript:6,html:9,java:7");
ha.addApplicant("jane", "javascript:7,html:4");
ha.addApplicant("james", "uml:7,bpmn:7,sw design:9");
try {ha.addApplicant("james", "sql:8,java:7");//james duplicated
} catch (ApplicationException e) {System.out.println(e.getMessage());}
try {ha.addApplicant("bob", "sql:11,java:7");//11 invalid level
} catch (ApplicationException e) {System.out.println(e.getMessage());}
try {ha.addApplicant("ted", "sql:9,xml:7");//xml undefined
} catch (ApplicationException e) {System.out.println(e.getMessage());}
String maryCpb = **ha.getCapabilities**("mary");
System.out.println(maryCpb); //html:9,java:7,javascript:6

R3: Candidature

Il metodo **enterApplication** permette ad un richiedente di candidarsi ad una posizione. Lancia un'eccezione se il richiedente o la posizione non sono stati inseriti, se il richiedente non ha una capacità per ciascuno skill richiesto dalla posizione o si è già candidato ad una posizione. Ad esempio, se la posizione chiede gli skill java e uml, il richiedente deve avere tra le sue capabilities quella per java e quella per uml. Il metodo **getApplicants** di Position fornisce la lista ordinata alfabeticamente dei nomi dei candidati alla posizione.

Esempi

```
ha.enterApplication("john", "programmer");
ha.enterApplication("david", "programmer");
ha.enterApplication("mary", "gui designer junior");
ha.enterApplication("jane", "gui designer junior");
try {ha.enterApplication("james", "sw team leader"); //james is not competent in sql
            } catch (ApplicationException e) {System.out.println(e.getMessage());}
try {ha.enterApplication("mary", "gui designer senior"); //mary has already applied for a position
            } catch (ApplicationException e) {System.out.println(e.getMessage());}
System.out.println(programmerPos.getApplicants()); //[david, john]
```

R4: Vincitori

Il metodo **setWinner** stabilisce il vincitore (richiedente) per una posizione. Lancia un'eccezione se il richiedente non si è candidato alla posizione, se c'è già un altro vincitore per la stessa posizione, o se la somma delle capacità relative agli skill della posizione non è maggiore del numero degli skill moltiplicati per 6 (ad es. per una posizione con 2 skill la soglia da superare è 12). Fornisce la somma delle capacità del vincitore.

Il metodo **getWinner** di Position dà il nome del vincitore o null se manca.

Esempi

```
int capSum = ha.setWinner("john", "programmer"); //16
System.out.println(programmerPos.getWinner()); //john
try {ha.setWinner("david", "programmer"); //winner already set
        } catch (ApplicationException e) {System.out.println(e.getMessage());}
try {ha.setWinner("jane", "gui designer junior"); //11 instead of 12+
        } catch (ApplicationException e) {System.out.println(e.getMessage());}
```

R5: Statistiche

Il metodo **skill_nApplicants** dà il numero di richiedenti per skill; gli skill sono ordinati alfabeticamente per nome.

Il metodo **maxPosition** dà il nome della posizione con il maggior numero di candidati.

Esempi

SortedMap<String, Long> skill_nApplicants = **ha.skill_nApplicants**();

System.out.println(skill_nApplicants);

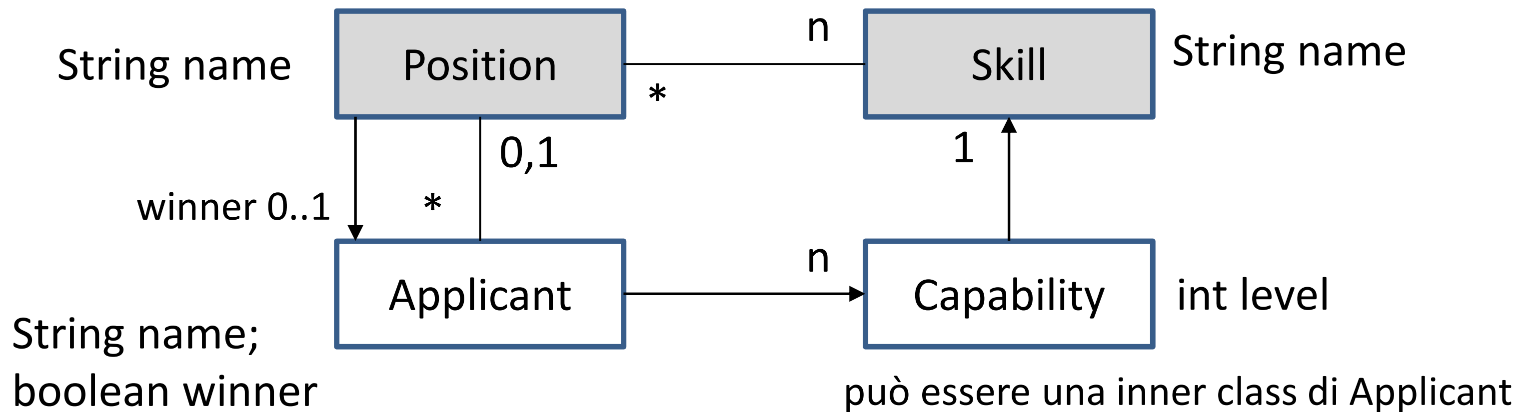
//{bpmn=1, html=2, java=3, javascript=2, sql=3, sw design=1, uml=1}

String maxPosition = **ha.maxPosition**();

System.out.println(maxPosition); //gui designer junior

Analisi e Progetto

Classi visibili: Position e Skill



Una posizione vede gli skill richiesti e uno skill vede le posizioni che lo richiedono (getPositions di Skill).

class Skill

```
public class Skill {
private String name; public String getName() {return name;}
Skill(String name) {this.name = name;}
private Map<String, Position> positions = new TreeMap<>();
void addPosition(Position position) {positions.put(position.getName(), position);}
public List<Position> getPositions() {return new ArrayList<>(positions.values());}
}

//il TreeMap è comodo per ordinare le posizioni per nome
```

class Position

```
public class Position {
private String name; public String getName() {return name;}
private List<Skill> skills; List<Skill> getSkills() {return skills;}
Position(String name, List<Skill> skills) {this.name = name; this.skills = skills;}
private List<Applicant> applicants = new ArrayList<>();
void addApplicant(Applicant applicant) {applicants.add(applicant);}
private Applicant winner;
public String getWinner() {if (winner == null) return null; else return
winner.getName();}
void setWinner(Applicant winner) {this.winner = winner;}
public List<String> getApplicants() {
        return applicants.stream().map(Applicant::getName).sorted()
        .collect(toList());
}
public int getApplicantN() {return applicants.size();}
public String toString() {return name;}
}
```

Si potrebbe usare una TreeMap per ordinare gli applicants per nome

class Applicant

```
class Applicant {
private String name; String getName() {return name;}
private Map<Skill,Capability> capabilities;
Collection<Capability> getCapabilities() {return capabilities.values();}
Applicant(String name, Map<Skill,Capability> capabilities) {
this.name = name; this.capabilities = capabilities;}
private Position position = null; Position getPosition() {return position;}
void setPosition(Position position) {this.position = position;}
private boolean winner = false; boolean isWinner() {return winner;}
void setWinner() {winner = true;}
Capability getCapability(Skill skill) {return capabilities.get(skill);}
Collection<Skill> getSkills() {return capabilities.keySet();}
}
```

class Capability

```
class Capability {
private int level; int getLevel() {return level;}
private Skill skill; Skill getSkill() {return skill;}
Capability(Skill skill, int level) {this.level = level;this.skill = skill;}
String getSkillName() {return skill.getName();}
}
```

public class HandleApplications

```
public class HandleApplications {
private Map<String, Skill> skills = new TreeMap<>();
private Map<String, Position> positions = new TreeMap<>();
private Map<String, Applicant> applicants = new TreeMap<>();

public void addSkills(String... names) throws ApplicationException {
for (String name: names) {
            if (skills.containsKey(name))
            throw new ApplicationException("duplicated skill " + name);
skills.put(name,  new Skill(name));
}
}
```

class HandleApplications

```
public void addPosition(String name, String... skillNames)
		throws ApplicationException {
if (positions.containsKey(name))
		throw new ApplicationException("duplicated position " + name);
List<Skill> positionSkills = new ArrayList<>();
for (String skillName: skillNames) {
		Skill skill = skills.get(skillName);
		if (skill == null)
			throw new ApplicationException("skill not found " + skillName);
		positionSkills.add(skill);
}
// gli inserimenti sono fatti se non ci sono errori
Position p = new Position(name, positionSkills);
positions.put(name, p);
for (Skill skill: positionSkills) skill.addPosition(p);
}
```

class HandleApplications

```
public Skill getSkill(String name) {return skills.get(name);}
public Position getPosition(String name) {return positions.get(name);}
```

HandleApplications

```
public void addApplicant(String name, String capabilities)
        throws ApplicationException {
if (applicants.containsKey(name))
        throw new ApplicationException("duplicated applicant " + name);
Map<Skill,Capability> capabilitiesMap = new TreeMap<>(comparing(Skill::getName));
Scanner scanner = new Scanner(capabilities);
scanner.useDelimiter("[,:]");
while (scanner.hasNext()) {
        String skillName = scanner.next(); int skillValue = scanner.nextInt();
        Skill skill = skills.get(skillName);
        if (skill == null) {scanner.close();
                throw new ApplicationException("skill not found " + skillName);}
        if (skillValue < 1 || skillValue > 10) {scanner.close();
        throw new ApplicationException(String.format("invalid level %d for skill %s", skillValue, skillName));}
        capabilitiesMap.put(skill, new Capability(skill, skillValue));}
scanner.close();
Applicant applicant = new Applicant(name, capabilitiesMap);
applicants.put(applicant.getName(), applicant);
}
```

HandleApplications

```
public String getCapabilities(String applicantName)
		throws ApplicationException {
Applicant applicant = applicants.get(applicantName);
if (applicant == null)
		throw new ApplicationException("applicant not found " + applicantName);
return applicant.getCapabilities().stream()
.map(c -> c.getSkillName() + ":" + c.getLevel())
.collect(joining(","));
}
```

HandleApplications

```
public void enterApplication(String applicantName, String positionName)
        throws ApplicationException {
Applicant applicant = applicants.get(applicantName);
if (applicant == null)
        throw new ApplicationException("applicant not found " + applicantName);
Position position = positions.get(positionName);
if (position == null)
        throw new ApplicationException("position not found " + positionName);
if (applicant.getPosition() != null)
        throw new ApplicationException("applicant already applied for a position " + applicant.getPosition().getName());
//si controlla che applicant abbia capabilities per tutti gli skill
for (Skill skill: position.getSkills()) {
        if (applicant.getCapability(skill) == null)
        throw new ApplicationException(String.format("capability %s not found in applicant %s ", skill.getName(),
        applicant.getName()));}
applicant.setPosition(position);
position.addApplicant(applicant);
}
```

HandleApplications

```
public int setWinner(String applicantName, String positionName)
        throws ApplicationException {
Applicant applicant = applicants.get(applicantName);
Position position = positions.get(positionName);
if(position.getWinner() != null) throw new ApplicationException(String.format("position %s already
        assigned to %s ", position.getName(),position.getWinner()));
int requiredCap = position.getSkills().size() * 6; // si può precalcolare nelle posizione
if (applicant.getPosition() != position) throw new ApplicationException(String.format("applicant %s didn't apply for
        position %s ", applicant.getName(), position.getName()));
int capSum = 0;
for (Skill skill: position.getSkills()) capSum += applicant.getCapability(skill).getLevel();
if (capSum <= requiredCap) throw new ApplicationException(String.format("applicant %s has not
the overall capability (%d, %d) for position %s ", applicant.getName(), capSum, requiredCap, position.getName()));
position.setWinner(applicant); applicant.setWinner();
System.out.println(String.format("applicant %s selected (%d, %d) for position %s ",
applicant.getName(), capSum, requiredCap, position.getName()));
return capSum;
}
```

HandleApplications

```
public SortedMap<String, Long> skill_nApplicants() {
return applicants.values().stream()
.flatMap(applicant -> applicant.getSkills().stream())
.collect(groupingBy(Skill::getName, TreeMap::new, counting()));
}
//con flatMap si ottiene uno stream di Skill (oppure di nomi di skill) per
//poi raggrupparli (per nome) e contarli

public String maxPosition() {
Position p = positions.values().stream()
.max(comparing(Position::getApplicantN)).orElse(null);
return p != null ? p.getName():null;
}
//si utilizza getApplicantN di Position per confrontare le posizioni
}
```

Issue Management

Si sviluppi un programma che consenta ad un'azienda di trattare le segnalazioni di anomalie relative ai componenti software che vende.

La classe principale si chiama IssueManager; tutte le classi si trovano nel package ticketing.

La classe Example presenta esempi di uso dei metodi principali.
Le eccezioni lanciate dal programma sono di tipo TicketException.

La classe IssueManager contiene

public static **enum UserClass** {Reporter, Maintainer}

R1: Users

Il sistema ammette due ruoli di utenti: *Reporter* and *Maintainer*. Un utente può svolgere un solo ruolo o entrambi.

Il metodo **createUser()** riceve uno username e il set dei ruoli (UserClasses) che l'utente svolge.
In alternativa al set si può usare una lista variabile di argomenti.
In entrambi i casi il metodo lancia un'eccezione se lo username è già stato inserito o se nessun ruolo è indicato.

Dato uno username si può ottenere il set dei ruoli dell'utente corrispondente con il metodo **getUserClasses().**

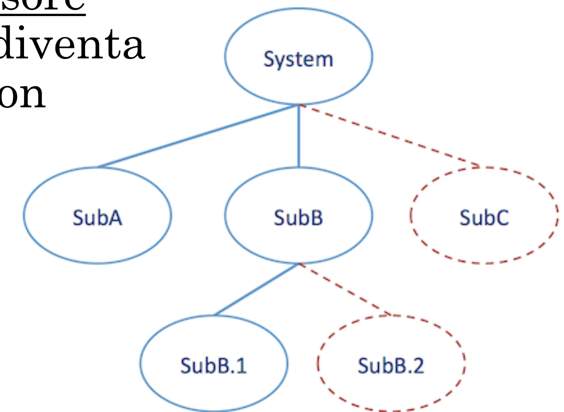
Esempio

```
import java.util.*;
import ticketing.*;
import static ticketing.IssueManager.*;

public class Example {
    public static void main(String[] args) throws TicketException {
        IssueManager ts = new IssueManager();
        HashSet<UserClass> both = new HashSet<>();
        both.add(UserClass.Reporter);
        both.add(UserClass.Maintainer);
        ts.createUser("alpha", UserClass.Reporter);
        ts.createUser("beta", UserClass.Reporter);
        ts.createUser("gamma", both);
        ts.createUser("delta", both);
        ts.createUser("epsilon", UserClass.Maintainer);
        System.out.println(ts.getUserClasses("gamma"));
```

R2: Components

I componenti forniti dall'azienda sono costituiti ricorsivamente da sotto-componenti. Il metodo **defineComponent()** genera un nuovo componente dato il nome e lancia un'eccezione se esiste già un componente con quel nome.

Il metodo **defineSubComponent()** genera un nuovo sotto-componente dati il nome e il <u>path che identifica il predecessore</u> (componente o sotto-componente) di cui il nuovo elemento diventa sotto-componente. Lancia un'eccezione se il predecessore non esiste o se ha già un sotto-componente con lo stesso nome. Esempio: dato il sistema in figura, per aggiungere *SubC* al componente System si scrive: *tm.defineSubComponent("SubC","/System");*, mentre per aggiungere *SubB.2* si scrive: *tm.defineSubComponent("SubB.2","/System/SubB");*.



Dato il path di un elemento (componente o sotto-componente) si può ottenere il set dei nomi dei sotto-componenti con il metodo **getSubComponents()** e il nome del predecessore con il metodo **getParentComponent()** (che dà null se l'elemento non ha un predecessore).

Esempio

```
ts.defineComponent("System");
ts.defineSubComponent("SubA", "/System");
ts.defineSubComponent("SubB", "/System");
ts.defineSubComponent("SubC", "/System");
ts.defineSubComponent("SubB.1", "/System/SubB");
ts.defineSubComponent("SubB.2", "/System/SubB");
ts.defineComponent("SystemBis");
ts.defineSubComponent("POS", "/SystemBis");

System.out.println("System has " +
    ts.getSubComponents("/System").size() + " children");
System.out.println("SubB.2 has parent " +
    ts.getParentComponent("/System/SubB/SubB.2"));
```

R3: Ticket opening

Un utente può aprire un ticket che contiene i dettagli di un'anomalia riguardante un dato elemento.

Un ticket è aperto con il metodo **openTicket()** che riceve lo username dell'utente, il path dell'elemento difettoso, la descrizione dell'anomalia e la severità (**Severity**) della stessa. Il metodo dà un id univoco (intero progressivo a partire da 1) per il ticket.
Lancia un'eccezione se lo username non è valido, il path non identifica alcun elemento, o se l'utente non svolge il ruolo *Reporter*.

Il metodo **getTicket()** riceve un id e dà l'oggetto **Ticket** corrispondente oppure null se l'id non è valido. Il metodo **getAllTickets()** dà la lista dei ticket ordinata naturalmente (si legga la nota).

La classe *Ticket* offre i metodi getter **getDescription()**, **getId()**, **getComponent()**, **getAuthor()** e **getSeverity()**.

Nota: Gli oggetti che compongono un tipo enumerativo implementano automaticamente l'interfaccia *Comparable*; l'ordinamento naturale corrisponde all'ordine con cui gli oggetti sono definiti nel tipo enumerativo. Quindi nella Severity *Blocking* precede *Major*.

La classe Ticket contiene gli enumerativi

public enum **Severity** {Blocking, Critical, Major, Minor, Cosmetic};

public static enum **State** {Open, Assigned, Closed};

Esempio

```
ts.openTicket("alpha", "/System/SubA", "Initial menu does not show 'open' item",
        Ticket.Severity.Major);
ts.openTicket("alpha", "/System/SubA", "Cannot save form XYZ",
        Ticket.Severity.Major);
ts.openTicket("alpha", "/System/SubB", "The colors in the diagram are hard to
        tell apart", Ticket.Severity.Minor);
ts.openTicket("beta", "/SystemBis/POS", "Credit cards not working",
        Ticket.Severity.Critical);


int id = ts.openTicket("alpha", "/System", "The system is not responding today",
        Ticket.Severity.Blocking);
Ticket t = ts.getTicket(id);
System.out.println("User " + t.getAuthor() + " created ticket " + t.getId() + " on
        component " + t.getComponent());
```

R4: Ticket lifecycle

I ticket hanno tre stati: *Open*, *Assigned*, *Closed*. Quando è aperto, un ticket è posto nello stato *Open*.

Il metodo **assignTicket()** riceve un ticket id e uno username: porta lo stato del ticket a *Assigned* e collega il ticket all'utente come assegnatario del ticket. Lancia un'eccezione se il ticket id o lo username non sono validi, o se l'utente non svolge il ruolo *Maintainer*.

Il metodo **closeTicket()** riceve un ticket id e la descrizione della soluzione, e porta lo stato del ticket a *Closed*. Lancia un'eccezione se il ticket non si trova nello stato *Assigned*.

La classe *Ticket* offre il metodo getter **getState()**, che dà lo stato corrente del ticket.

Il metodo **getSolutionDescription()** della classe *Ticket* dà la descrizione della soluzione; lancia un'eccezione se il ticket non si trova nello stato *Closed*.

Esempio

```
ts.assingTicket(id, "delta");
ts.closeTicket(id, "The user had the network cable unplugged...");

System.out.println("The ticket status is " + t.getState() + " solution:
    " + t.getDescription());
```

R5: Statistics

Il metodo **countBySeverityOfState()** dato uno stato dei ticket fornisce una mappa con il numero di ticket per Severity, considerando soltanto i ticket in quello stato oppure tutti i ticket se l'argomento è nullo. La mappa è ordinata in base alla Severity.

Il metodo **topMaintainers()** dà una lista di stringhe: ogni stringa ha il formato *"username:###"* dove *username* è il nome dell'utente e ### è il numero di ticket chiusi dall'utente come maintainer. La lista è ordinata per numero decrescente di ticket e poi per username (in ordine alfabetico).

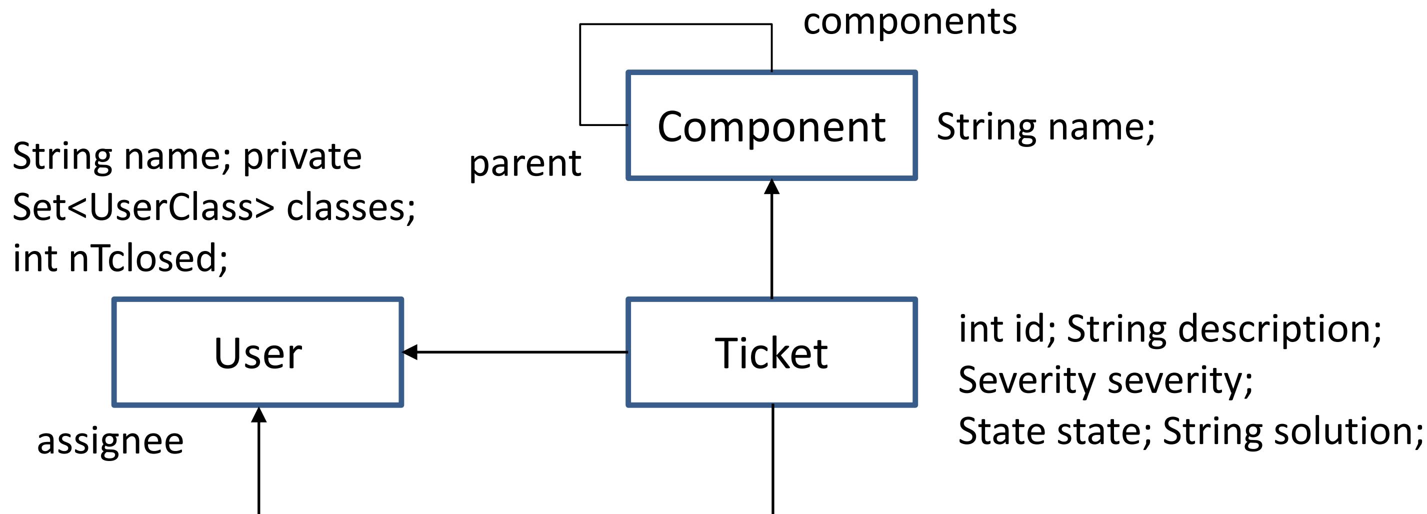
Esempio

```
System.out.println("Count open tickets:\n"
          +ts.countBySeverityOfState(Ticket.State.Open));

System.out.println("Count all tickets:\n"  +ts.countBySeverityOfState(null));

System.out.println(ts.topMaintainers());
```

Count open tickets:
{Critical=1, Major=2, Minor=1}
Count all tickets:
{Blocking=1, Critical=1, Major=2, Minor=1}

[delta:1, epsilon:0, gamma:0]

Analisi e Progetto



Ticket include
public enum Severity {*Blocking, Critical, Major, Minor, Cosmetic};*
public static enum State {*Open, Assigned, Closed*}

Nella classe principale:
private Map<String,User> users = new HashMap<>();
private Map<String, Component> components = new HashMap<>();
private int ticketId; private List<Ticket> tickets = new ArrayList<>();
public static enum UserClass { Reporter, Maintainer }

class IssueManager

```
private Map<String,User> users = new HashMap<>();
private Map<String,Component> components = new HashMap<>();
//components ha come chiave il path del componente
private int ticketId; private List<Ticket> tickets = new ArrayList<>();
//ticketId è il contatore dei ticket generati
public static enum UserClass { Reporter, Maintainer }

public void createUser (String username, UserClass... classes) throws TicketException {
   if(users.containsKey(username) || classes.length==0) throw new TicketException();
   users.put(username, new User(username,classes)); }

public void createUser (String username, Set<UserClass> classes) throws
         TicketException {
   if(users.containsKey(username) || classes.size()==0) throw new TicketException();
   users.put(username, new User(username,classes)); }

public Set<UserClass> getUserClasses (String username){
   if(!users.containsKey(username)) return null;
   return users.get(username).getClasses() ;}
```

class User

```
import java.util.*; import ticketing.IssueManager.UserClass;
public class User {
  private String name; private Set<UserClass> classes;
  private int nTclosed; // n. ticket chiusi

  public User(String name, Set<UserClass> classes) {
    this.name = name; this.classes = classes; }
  public User(String name, UserClass... classes) {
    this.name = name;  this.classes = new HashSet<>(Arrays.asList(classes)); }
  public String getName() {return name; }

  public Set<UserClass> getClasses() {return classes; }

  public int getNTclosed(){return nTclosed; }

  public void incrementNTclosed() {++nTclosed; }
```

class IssueManager

```
public void defineComponent (String name) throws TicketException {
    String path = "/" + name;
    if(components.containsKey(path))
        throw new TicketException("Duplicate component name");
    components.put(path, new Component(name));
}

public void defineSubComponent (String name, String parentPath) throws
        TicketException {
    String path = parentPath + "/" + name;
    if(components.containsKey(path))
        throw new TicketException("Duplicate component name");
    Component parent = components.get(parentPath);
    if(parent==null) throw new TicketException("No parent");
    components.put(path, new Component(name, parent));
}
```

class IssueManager

```
 public Set<String> getSubComponents(String path){
        Component c = components.get(path);
        if (c == null) return null;
        return c.getComponents().stream()
                .map(Component::getName)
                .collect(toCollection(() -> new TreeSet<>()));
    }
public String getParentComponent (String path){
        Component c = components.get(path);
        if (c==null) return null;
        Component p = c.getParent();
        return (p==null?null:p.getPath());
    }
```

Esempi:

System has these components [SubA, SubB, SubC]

SubB.2 has parent /System/SubB

class Component

```
private String name; public String getName() {return name;}
private Component parent; public Component getParent() {return parent;}
public Component(String name, Component parent) {
    this.name = name;  parent.addComponent(this);  this.parent = parent;}
public Component(String name) {this.name = name; this.parent = null;}
private ArrayList<Component> components = new ArrayList<>();
void addComponent(Component c) {components.add(c);}
ArrayList<Component> getComponents() {return components;}

public String getPath() {
    return (parent!=null?parent.getPath():"") + "/" + name;
}
```

Nota: si potrebbe registrare anche il path per evitare la recursione.

class IssueManager

```
public int openTicket(String username, String componentPath, String description,
          Ticket.Severity severity) throws TicketException {
    User user = users.get(username);
    Component comp = components.get(componentPath);
    if(user==null || comp==null) throw new TicketException();
    if(!user.getClasses().contains(IssueManager.UserClass.Reporter))
          throw new TicketException();
    Ticket t = new Ticket(++ticketId,user,comp,description,severity);
    tickets .add(t); return ticketId;}

  public Ticket getTicket(int ticketId){return tickets.get(ticketId-1); }

  public List<Ticket> getAllTickets(){
    ArrayList<Ticket> t = new ArrayList<>(tickets);
    Collections.sort(t, comparing(Ticket::getSeverity));
    return t;
  }
```

class IssueManager

```
public void assignTicket (int ticketId, String username) throws TicketException {
    if(ticketId<1 || ticketId>tickets.size()) throw new TicketException();
    Ticket t = tickets.get(ticketId-1);
    User user = users.get(username);
    if(user == null) throw new TicketException();
    t.assignTo(user);
  }

  public void closeTicket(int ticketId, String description) throws TicketException {
    Ticket t = tickets.get(ticketId-1);
    t.close(description);
  }
```

class Ticket

private int id; private User user; private Component comp;
private String description; private Severity severity;
private State state; private String solution;
private User assignee;
public enum Severity {*Blocking, Critical, Major, Minor, Cosmetic};*
public static enum State {*Open, Assigned, Closed}*

```
Ticket(int id, User user, Component comp, String description,
        Severity severity) {
    this.id=id; this.user = user; this.comp = comp; this.description = description;
    this.severity = severity; this.state = State.Open;
}
```

+ getters

class Ticket

```
public String getSolutionDescription() throws TicketException {
    if(state!=State.Closed)
        throw new TicketException("Cannot get solution for not closed ticket");
    return solution;}

void assignTo(User user) throws TicketException {
    if(this.state!=State.Open)
        throw new TicketException("assignTo requires an open ticket");
    if(!user.getClasses().contains(IssueManager.UserClass.Maintainer))
        throw new TicketException("Cannot assing ticket to non maintainer");
    this.assignee = user;
    this.state=State.Assigned;}

 void close(String description) throws TicketException {
    if(this.state!=State.Assigned)
        throw new TicketException("Cannot close a ticket that is not in assigne state");
    this.solution = description;
    this.state = State.Closed;
    assignee.incrementNTclosed(); }
```

class IssueManager

```
public List<String> topMaintainers (){
return users.values().stream()
.filter(u -> u.getClasses().contains(UserClass.Maintainer))
.sorted(comparing(User::getNTclosed, reverseOrder()).thenComparing(User::getName) )
.map( u -> u.getName() + ":" + u.getNTclosed())
.collect(toList());}

public SortedMap<Ticket.Severity,Long> countBySeverityOfState(Ticket.State state){
Predicate<Ticket> which = t -> (state==null?true:t.getState()==state);
return tickets.stream()
    .filter(which)
    .collect(groupingBy(Ticket::getSeverity,TreeMap::new,counting()));
}

//L'uso di getNTclosed semplifica il confronto tra gli utenti
```

ExamHandling

Gestione di esami

Si scriva un programma per la gestione di esami. Le classi si trovano nel package **exams**; la classe principale è **ExamHandling**. La classe **Example** nel package **main** presenta esempi di uso dei metodi principali ed esempi dei controlli richiesti. Le eccezioni sono di tipo **ExamException**.

R1: Piani di studio e studenti

Il metodo **addStudyPlan** (String name, String... courses) aggiunge un piano di studio; il primo parametro è il nome del piano di studio e i parametri successivi sono i nomi dei corsi che lo compongono. Un corso può far parte di più piani di studio. Lancia un'eccezione se il nome del piano è duplicato.

Il metodo **enrollStudent** (String studentId, String studentName, String studyPlan) registra uno studente con identificativo e nome, e associa allo studente un piano di studi. Lancia un'eccezione se l'identificativo è duplicato o se il piano non è definito.

Il metodo **numberOfStudentsForStudyPlan** dà il n. di studenti per ogni piano di studio; i piani sono ordinati alfabeticamente per nome.
Il metodo **studentsForStudyPlan** dà la lista ordinata degli id degli studenti per ogni piano di studio; i piani sono ordinati alfabeticamente per nome.

R2

R2: Esami

Il metodo **openExam** (String courseName) apre una sessione d’esame per il corso indicato. Lancia un'eccezione se il corso non è definito o se c’è già una sessione aperta.

Uno studente dà un esame con il metodo **takeExam** (String studentId, String course). Il metodo lancia un'eccezione se il corso non è definito o non è nel piano di studi dello studente, la sessione d’esame non è aperta, lo studente ha già eseguito takeExam nella sessione oppure ha già un voto sufficiente (>= 18) per il corso.

Il metodo **studentsWhoTookTheExam** (String courseName) dà l’elenco ordinato degli id degli studenti che hanno eseguito takeExam nella sessione corrente. Lancia un'eccezione se il corso non è definito.
Il metodo int **numberOfOpenExamSessions** dà il n. delle sessioni d’esame aperte nel momento della chiamata del metodo.

R3

R3: Valutazioni

Il metodo **giveGrade** (String courseName, String studentId, int grade) permette di dare un voto allo studente. Il metodo lancia un'eccezione se lo studente non ha dato l'esame (cioè non ha eseguito takeExam), ha già un voto sufficiente (>= 18) per l'esame, il voto non è compreso tra 12 e 30 (estremi inclusi). L'esame è superato se il voto è sufficiente, altrimenti è fallito. Un voto <18 è detto failing grade.

Il metodo **closeExam** (String courseName) chiude la sessione d'esame. Lancia un'eccezione se la sessione non è aperta per il corso o se non tutti gli studenti che hanno dato l'esame hanno ricevuto un voto. Dopo la chiusura di una sessione d'esame, la chiamata del metodo studentsWhoTookTheExam dà una lista vuota (cioè la lista degli studenti che hanno dato l'esame nella sessione ormai chiusa è vuota).

R4

R4: Statistiche

Il metodo **gradesOfStudent** (String studentId) per lo studente indicato raggruppa per voto i nomi dei corsi relativi agli esami; i voti sono ordinati in senso decrescente e i nomi dei corsi alfabeticamente.
Il metodo **gradesOfCourse** (String courseName) per il corso indicato raggruppa per voto gli id degli studenti che hanno superato l'esame; i voti sono ordinati in senso decrescente e gli id alfabeticamente.
Il metodo int **failingGradesOfStudent** (String studentId) dà il n. degli esami falliti (cioè il n. dei failing grades) per lo studente indicato.
Il metodo int **failingGradesOfCourse** (String courseName) dà il n. degli esami falliti (cioè il n. dei failing grades) per il corso indicato.
I 4 metodi lanciano un'eccezione se l'id dello studente o il corso sono indefiniti.

Programma di test

```
ExamHandling eh = new ExamHandling();
//R1
eh.addStudyPlan("cs1", "oop", "data structures", "operating systems");
eh.addStudyPlan("cs2", "oop", "graphical interfaces");
try {eh.addStudyPlan("cs1", "oop");} //duplicated study plan
        catch(ExamException ex) {System.out.println(ex.getMessage());}
eh.enrollStudent("jo","john", "cs1"); eh.enrollStudent("my", "mary", "cs1");
eh.enrollStudent("fk", "frank", "cs2"); eh.enrollStudent("an","anne", "cs2");
try {eh.enrollStudent("jo","jane", "cs1");} //duplicated student id
        catch(ExamException ex) {System.out.println(ex.getMessage());}
try {eh.enrollStudent("bo","bob", "cs3");} //undefined study plan
        catch(ExamException ex) {System.out.println(ex.getMessage());}
SortedMap<String, Long> numberOfStudentsForStudyPlan =
        eh.numberOfStudentsForStudyPlan();
System.out.println(numberOfStudentsForStudyPlan);
SortedMap<String, List<String>> studentsForStudyPlan =
        eh.studentsForStudyPlan();
        System.out.println(studentsForStudyPlan);
```

Programma di test

```
//R2
eh.openExam ("oop");
try {eh.openExam ("oop");} //session already open
        catch(ExamException ex) {System.out.println(ex.getMessage());}
eh.takeExam("jo", "oop"); eh.takeExam("fk", "oop");
try {eh.takeExam("my", "operating systems");} //no open session
        catch(ExamException ex) {System.out.println(ex.getMessage());}
eh.openExam ("operating systems");
try {eh.takeExam("an", "operating systems");} //course not in study plan
        catch(ExamException ex) {System.out.println(ex.getMessage());}
List<String> studentsWhoTookTheExam = eh.studentsWhoTookTheExam
        ("oop");
System.out.println(studentsWhoTookTheExam); // [fk, jo]
System.out.println (eh.numberOfOpenExamSessions ()); //2
eh.takeExam("my", "oop");
```

Programma di test

```
//R3
eh.giveGrade("oop", "jo", 27); eh.giveGrade("oop", "fk", 29);
eh.giveGrade("oop", "my", 29);
try {eh.giveGrade("oop", "an", 25);} //exam not taken
        catch(ExamException ex) {System.out.println(ex.getMessage());}
System.out.println(eh.studentsWhoTookTheExam ("oop")); //[fk, jo, my]
eh.closeExam("oop");
System.out.println(eh.studentsWhoTookTheExam ("oop")); //[]
eh.takeExam("my", "operating systems");
try {eh.closeExam("operating systems");} //missing grades
        catch(ExamException ex) {System.out.println(ex.getMessage());}
System.out.println (eh.numberOfOpenExamSessions ()); //1
```

Programma di test

```
//R4
eh.takeExam("jo", "operating systems");
eh.giveGrade("operating systems", "jo", 27);
eh.openExam ("data structures");
eh.takeExam("jo", "data structures");
eh.giveGrade("data structures", "jo", 30);
System.out.println (eh.gradesOfStudent("jo"));
        //{30=[data structures], 27=[oop, operating systems]}

System.out.println (eh.gradesOfCourse("oop")); //{29=[fk, my], 27=[jo]}
eh.giveGrade("operating systems", "my", 15);
System.out.println (eh.failingGradesOfStudent("my"));
System.out.println (eh.failingGradesOfCourse("operating systems"));
```

Analisi

Quali classi servono oltre alla classe ExamHandling?

Dai requisiti:

Il metodo **addStudyPlan** (String name, String… courses) aggiunge un piano di studio.

Il metodo **enrollStudent** (String studentId, String studentName, String studyPlan) registra uno studente con identificativo e nome, e associa allo studente un piano di studi.

Il metodo **openExam** (String courseName) apre una sessione d'esame per il corso indicato.

Uno studente dà un esame con il metodo **takeExam** (String studentId, String course).

Il metodo **giveGrade** (String courseName, String studentId, int grade) permette di dare un voto allo studente.

Termini principali: StudyPlan, Course, Student, Session, Grade.

Collegamenti: StudyPlan – Course, StudyPlan – Student,

La sessione (aperta o chiusa) è un attributo di un corso; da **openExam e closeExam.**

TakeExam: uno studente sostiene l'esame; è aggiunto alla lista degli studenti che hanno sostenuto l'esame.

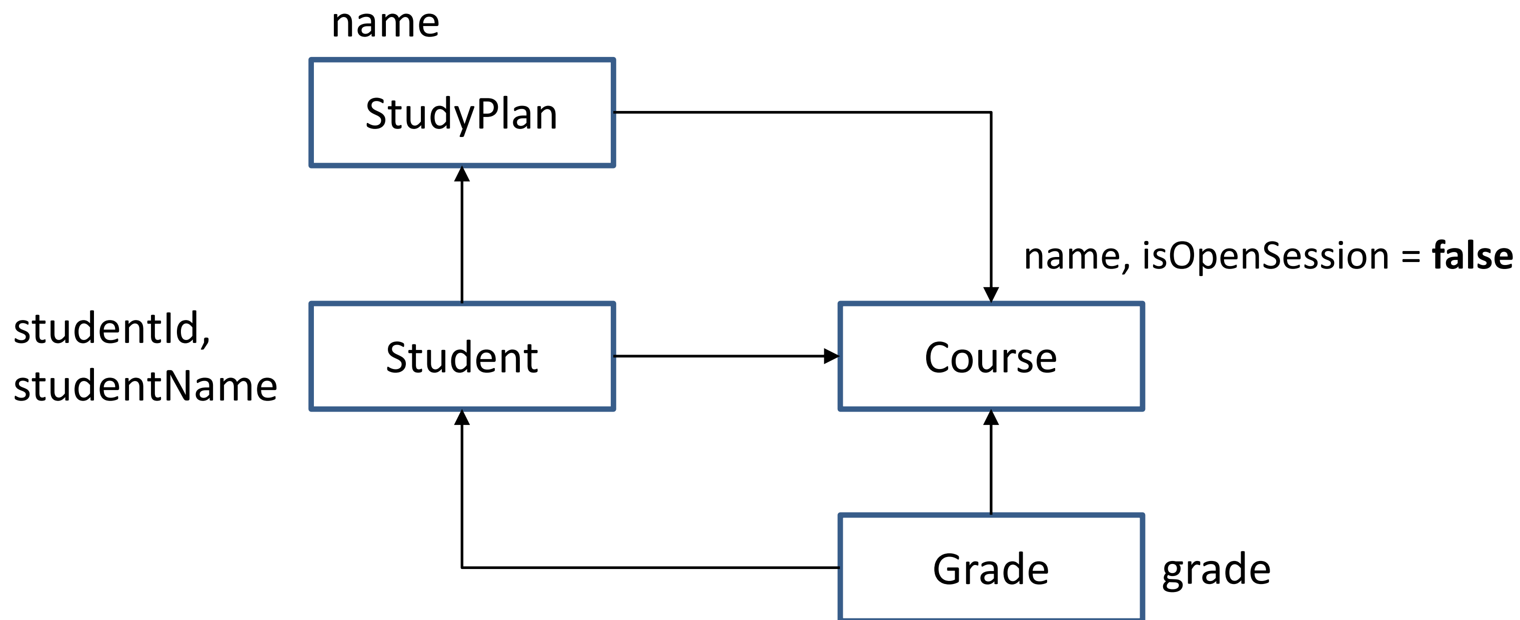
Collegamento: takeExam: Student – Course.

GiveGrade: uno studente riceve un voto per un corso.

Collegamenti: Grade – Course e Grade – Student.

Association class: Integer per contenere il voto

Progetto



Al posto della classe StudyPlan e della relazione StudyPlan – Course basta una mappa che associa ai nomi dei piani i nomi dei corsi.

ExamHandling

```
public class ExamHandling {
private Map<String, List<String>> studyPlanMap = new TreeMap<>();
private SortedMap<String, Student> studentMap = new TreeMap<>();
private SortedMap<String, Course> courseMap = new TreeMap<>();
```

R1

```
public void addStudyPlan (String name, String... courses) throws ExamException {
if (studyPlanMap.containsKey(name))
		throw new ExamException("duplicated studyPlan " + name);
List<String> studyPlanCourses = Arrays.asList(courses);
studyPlanMap.put(name, studyPlanCourses);
for (String s: studyPlanCourses)
		if (!courseMap.keySet().contains(s)) courseMap.put(s, new Course (s));
}

public void enrollStudent (String studentId, String studentName, String studyPlan)
		throws ExamException {
if (!studyPlanMap.containsKey(studyPlan))
		throw new ExamException("undefined studyPlan " + studyPlan);
if (studentMap.containsKey(studentId))
		throw new ExamException("duplicated student " + studentId);
studentMap.put(studentId, new Student(studentId, studentName, studyPlan));
}
```

R1

```
public SortedMap<String, Long> numberOfStudentsForStudyPlan() {
return studentMap.values().stream()
.collect(groupingBy(Student::getStudyPlan, TreeMap::new, counting()));
}
public SortedMap<String, List<String>> studentsForStudyPlan() {
        //dà null inizialmente
return studentMap.values().stream()
.sorted(comparing(Student::getId))  //sono già ordinati dalla studentMap
.collect(groupingBy(Student::getStudyPlan, TreeMap::new,
        mapping(Student::getId, toList())));
}
```

R2

```
// Utility methods

Course checkCourse (String courseName) throws ExamException {
Course course = courseMap.get(courseName);
if (course == null)
        throw new ExamException("undefined course " + courseName);
return course;
}

Student checkStudent (String studentId) throws ExamException {
Student student = studentMap.get(studentId);
if (student == null) throw new ExamException("undefined student " + studentId);
return student;
}
```

R2

```
public void openExam (String courseName) throws ExamException {
Course course = checkCourse(courseName);
if (course.isOpenSession())
		throw new ExamException("exam session already open for course " +
				courseName);
course.openSession();
}
```

R2

```
public void takeExam (String studentId, String courseName) throws
        ExamException {
Course course = checkCourse(courseName);
Student student = checkStudent(studentId);
List<String> courses = studyPlanMap.get(student.getStudyPlan());
if (!courses.contains(courseName))
        throw new ExamException(String.format("course %s not in study
        plan of %s ", courseName, studentId));
if (!course.isOpenSession())
        throw new ExamException("no open session for course " + courseName);
if (course.getGrade(studentId) != null)
        throw new ExamException(String.format("student %s already has a
        grade for course %s ", studentId, courseName));
if (!course.addStudentToExam(studentId)) // può aggiungere lo studente
        throw new ExamException(String.format("student %s already took
        the exam for course %s ", studentId, courseName));
}
```

R2

```
public List<String> studentsWhoTookTheExam (String courseName)
            throws ExamException {
Course course = checkCourse(courseName);
return course.getStudentsWhoTookTheExam();
}

public int numberOfOpenExamSessions () {
return (int) courseMap.values().stream()
.filter(c -> c.isOpenSession())
.count();
}
```

R3

```
public void giveGrade (String courseName, String studentId, int grade) throws
        ExamException {
Student student = checkStudent(studentId);
Course course = checkCourse(courseName);
Grade g = course.getGrade(studentId);
if (g != null)
        throw new ExamException(String.format("duplicated grade for student %s in
        course %s ", studentId, courseName));
if (!course.checkStudentForExam(studentId))
        throw new ExamException(String.format("student %s didn't take the exam for
        course %s ", studentId, courseName));
if (grade < 12 || grade > 30)
        throw new ExamException("wrong grade " + grade);
if (grade >= 18) {
g = new Grade (grade, studentId, courseName);
course.addGrade(studentId, g); // rel nn bidirectional
student.addGrade(courseName, g);
} else {
student.incrFailingGrades(); course.incrFailingGrades(); //utili per statistiche
}
}
```

R3

```
public void closeExam (String courseName) throws ExamException {
Course course = checkCourse(courseName);
if (!course.isOpenSession())
        throw new ExamException("exam session is not open for course " +
        courseName);
if (!course.closeSession())
        throw new ExamException("missing grades in session for course " +
        courseName);
}
```

R4

```
public SortedMap<Integer, List<String>> gradesOfStudent (String
        studentId) throws ExamException {
Student student = checkStudent(studentId);
return student.getGrades().stream()
.sorted(comparing(Grade::getCourseName))
.collect(groupingBy(Grade::getVal, () -> new TreeMap<>(reverseOrder()),
        mapping(Grade::getCourseName, toList())));
}

public SortedMap<Integer, List<String>> gradesOfCourse (String
        courseName) throws ExamException {
Course course = checkCourse(courseName);
return course.getGrades().stream()
.sorted(comparing(Grade::getStudentId))
.collect(groupingBy(Grade::getVal, () -> new TreeMap<>(reverseOrder()),
        mapping(Grade::getStudentId, toList())));
}
```

R4

```
public int failingGradesOfStudent(String studentId) throws ExamException {
Student student = checkStudent(studentId);
return student.getFailingGrades();
}

public int failingGradesOfCourse(String courseName) throws ExamException {
Course course = checkCourse(courseName);
return course.getFailingGrades();
}
```

Class Course

```
public class Course {
private String name; String getName() {return name;}
Course (String name) {this.name = name;}

private int failingGrades;
int getFailingGrades() {return failingGrades;}
void incrFailingGrades() {failingGrades++;}

private Map<String, Grade> grades = new HashMap<>();
void addGrade(String studentId, Grade grade) {
        grades.put(studentId, grade);
}
List<Grade> getGrades() {return new ArrayList<>(grades.values());}
Grade getGrade(String studentId) {return grades.get(studentId);}
```

Class Course

```
private SortedSet<String> studentsWhoTookTheExam = new TreeSet<>();
boolean addStudentToExam (String studentId) {
		return studentsWhoTookTheExam.add(studentId);}
boolean checkStudentForExam (String studentId) {
		return studentsWhoTookTheExam.contains(studentId);}
List<String> getStudentsWhoTookTheExam() {
		return new ArrayList<>(studentsWhoTookTheExam);}

private boolean isOpenSession = false;
boolean isOpenSession() {return isOpenSession;}
void openSession() {isOpenSession = true;}
boolean closeSession() { //checks that all the students have received a grade
		if (!grades.keySet().containsAll(studentsWhoTookTheExam)) return false;
		studentsWhoTookTheExam.clear(); // clears the list
		isOpenSession = false;
		return true;
}
```

Class Student

```
class Student {
private String id; String getId() {return id;}
private String name; String getName() {return name;}
private String studyPlan; String getStudyPlan() {return studyPlan;}
Student(String id, String name, String studyPlan) {
        this.id = id; this.name = name; this.studyPlan = studyPlan;}

private int failingGrades;
int getFailingGrades() {return failingGrades;}
void incrFailingGrades() {failingGrades++;}
private Map<String, Grade> grades = new HashMap<>();
boolean addGrade(String courseName, Grade grade) {
        if (grades.containsKey(courseName)) return false;
        grades.put(courseName, grade);return true;}
List<Grade> getGrades() {return new ArrayList<>(grades.values());}
Grade getGrade(String courseName) {return grades.get(courseName);}
```

class Grade

```
class Grade {
private int val;  int getVal() {return val;}
private String studentId;  String getStudentId() {return studentId;}
private String courseName; String getCourseName() {return courseName;}

Grade(int val, String studentId, String courseName) {
        this.val = val;this.studentId = studentId; this.courseName =
        courseName;
}
}
```

class ExamException

```
@SuppressWarnings("serial")
public class ExamException extends Exception {

public ExamException (String reason) {
		super(reason);
}
}
```