UNIVERSITÀ DEGLI STUDI DI PARMA

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Una Library per la Manipolazione di Polinomi in Forma di Bernstein in Java

Relatore:
Chiar.mo Prof. Federico Bergenti

Candidato:
Paolo Grossi

Anno Accademico 2013/2014

UNIVERSITY OF PARMA

Department of Mathematics and Computer Science

Bachelor Degree in Computer Science

Thesis

A Java Library for the Manipulation of Bernstein Polynomials

Supervisor:
Chiar.mo Prof. Federico Bergenti

Candidate:
Paolo Grossi

Academic Year 2013/2014

Indice

Prefazione

L'obiettivo della tesi proposta è quello di fornire una library scritta in linguaggio Java per una manipolazione efficace dei polinomi in forma di Bernstein. Inizialmente il lavoro ha previsto una fase di studio ed analisi dei requisiti richiesti, in particolar modo per individuare la modalità con cui realizzare alcune strutture algebriche di base, seguita da una fase di implementazione del codice effettivo e da un'ultima fase di verifica, per quanto riguarda le proprietà da rispettare.

La struttura di questa tesi prevede quindi la presenza di quattro principali macrosezioni:

- Introduzione dei principali concetti esposti da un punto di vista algebrico;
- Soluzione proposta con l'implementazione in Java e descrizione delle principali decisioni prese;
- Verifica delle soluzioni intraprese sfruttando un tool per un testing accurato;
- Implementazione di casi d'uso realistici.

Nella prima parte si parlerà delle principali strutture utilizzate, come anelli, monomi, polinomi, per poi concentrarsi sulla definizione dei polinomi di Bernstein e delle proprietà di cui questi godono.

All'interno della seconda parte verranno riprese le classi implementate, a partire dalle interfacce per gli anelli e per i suoi elementi, passando dalla realizzazione delle classi per i monomi ed i polinomi arrivando alle concretizzazioni delle classi per gli interi e i razionali. Sono state realizzate due versioni per la gestione dei polinomi (in forma sparsa e in forma densa) prevedendo una gestione simultanea dei due aspetti.

Con la terza parte si propone una semplice verifica delle varie proprietà di cui gli oggetti in discussione possono godere, in particolar modo concentrandosi su tutti gli aspetti considerati durante lo sviluppo della library. Ad esempio verificheremo che un numero razionale non possa mai avere un denominatore nullo oppure che uno dei termini del monomio non sia mai di grado negativo.

In conclusione vengono illustrati alcuni esempi per l'effettiva manipolazione di tutte le strutture implementate ed in particolar modo dei polinomi in forma di Bernstein.

Alla mia famiglia
che rimarrà per sempre
la parte migliore di me.

Capitolo 1

Introduzione

1.1 Gruppi, Anelli e Campi

Definizione 1.1 (Gruppo)**.** Un *gruppo* è un insieme G munito di un'operazione binaria $\star$ che ad ogni coppia di elementi a, b di G associa un elemento, che indichiamo con $a \star b$, appartenente a G, rispettando i seguenti assiomi:

1. proprietà associativa: $\forall a, b, c \in G\colon (a \star b) \star c = a \star (b \star c)$;
2. esistenza dell'elemento neutro e: $\forall a \in G\colon \exists e \in G.\, a \star e = e \star a = a$;
3. esistenza dell'elemento inverso a': $\forall a \in G\colon \exists a' \in G.\, a \star a' = a' \star a = e$.

Definizione 1.2 (Gruppo abeliano)**.** Un *gruppo abeliano* è un gruppo $(G, \star)$ la cui operazione binaria $\star$ gode di un'ulteriore proprietà:

4. proprietà commutativa: $\forall a, b \in G\colon a \star b = b \star a$.

I numeri interi $\mathbb{Z}$ con l'operazione di somma $+$ formano un gruppo abeliano, mentre i numeri interi $\mathbb{Z}$ non formano un gruppo con l'operazione di moltiplicazione $*$: la moltiplicazione è associativa e ha un elemento neutro 1, ma la maggior parte degli elementi non ha una inversa.

Definizione 1.3 (Anello)**.** Un *anello* è un insieme A dotato di due operazioni binarie, $+$ e $*$, per cui valgono le seguenti proprietà:

1. $(A, +)$ è un gruppo abeliano, in cui ogni elemento a ha elemento neutro 0 e elemento inverso $-a$;
2. $(A, *)$ è un gruppo associativo (o semigruppo).

L'esempio più basilare della struttura ad anello è l'insieme $\mathbb{Z}$ dei numeri interi, dotato delle usuali operazioni di somma e prodotto. Tale anello è commutativo ed è un dominio d'integrità. L'insieme $\mathbb{N}$ dei numeri naturali non è invece un anello, perché non esistono gli inversi rispetto all'addizione.

Definizione 1.4 (Campo)**.** Un *campo* è un insieme K dotato di due operazioni binarie, $+$ e $*$, per cui valgono le seguenti proprietà:

1. $(K, +)$ è un gruppo abeliano, in cui ogni elemento a ha elemento neutro 0 e elemento inverso $-a$;
2. $(K \setminus \{0\}, *)$ è un gruppo abeliano, in cui ogni elemento a ha elemento neutro 1 e elemento inverso a^{-1}.

Un pratico esempio di campo è l'insieme $\mathbb{Q}$ dei numeri razionali con le operazioni di addizione $+$ e moltiplicazione $*$ tra numeri; mentre l'insieme $\mathbb{Z}$ con le operazioni di addizione $+$ e moltiplicazione $*$ non è un campo in quanto i soli elementi ad avere un inverso moltiplicativo sono $+1$ e -1.

1.2 Domini d'integrità e chiusure algebriche

Definizione 1.5 (Dominio d'integrità)**.** Un *dominio d'integrità* è un anello $(A, +, *)$ commutativo privo di divisori dello zero, con unità tale che $0 \neq 1$ in cui il prodotto di due qualsiasi elementi non nulli è un elemento non nullo, per cui valgono le seguenti proprietà:

1. $\forall a, b \in A \colon a * b = b * a$;
2. $a * b = 0 \Rightarrow (a = 0 \vee b = 0)$.

La seconda legge viene detta legge di annullamento del prodotto. Equivalentemente, un dominio di integrità può essere definito come un anello commutativo in cui l'ideale nullo $\{0\}$ è primo, o come sottoanello di un qualche campo. La condizione che $0 \neq 1$ serve all'unico scopo di escludere l'anello banale $\{0\}$ con un solo elemento.

Definizione 1.6 (Chiusura algebrica)**.** La *chiusura algebrica* di un campo K è la più piccola estensione algebrica di K che è algebricamente chiusa; ovvero, la chiusura algebrica di K è quel campo che si ottiene aggiungendo a K le radici di tutti i polinomi a coefficienti in K.

Ogni campo ha una chiusura algebrica, e questa è unica a meno di isomorfismi: questo permette di parlare della chiusura algebrica di K, invece che di una chiusura algebrica di K.

Il teorema fondamentale dell'algebra afferma che il campo $\mathbb{C}$ dei numeri complessi è algebricamente chiuso, e di conseguenza è la chiusura algebrica del campo dei numeri $\mathbb{R}$. Tuttavia, $\mathbb{C}$ non è la chiusura algebrica del campo $\mathbb{Q}$, corrispondente al campo dei numeri algebrici.

1.3 Insiemi dei numeri $\mathbb{Z}$ e $\mathbb{Q}$

Definizione 1.7 (Insieme $\mathbb{Z}$)**.** L'*insieme* $\mathbb{Z}$ dei numeri interi è formato dall'unione dei numeri naturali [0, 1, 2, ...) e dei numeri interi negativi negativi (..., -3, -2, -1]. L'insieme $\mathbb{Z}$ gode di alcune proprietà algebriche:

1. è chiuso rispetto alle operazioni di addizione $+$, sottrazione $-$ e moltiplicazione $*$;
2. $(\mathbb{Z}, +)$ costituisce un gruppo abeliano;
3. $(\mathbb{Z}, +, *)$ costituisce un anello commutativo ed un dominio d'integrità;
4. la chiusura algebrica dei numeri $\mathbb{Z}$ è formata dai numeri razionali.

La divisione ordinaria non è definita su $\mathbb{Z}$, ma è possibile usare l'*algoritmo di Euclide* per effettuare una divisione con resto:

$$\forall a, b \in \mathbb{Z}\colon \exists q, r \in \mathbb{N}.\, a = q * b + r, \text{ con } 0 \leq r < |b|$$

L'algoritmo di Euclide mostra come due numeri interi abbiano sempre un massimo comune divisore ed un minimo comune multiplo. Inoltre, per il teorema fondamentale dell'aritmetica ogni numero intero ha un'unica decomposizione come prodotto di numeri primi. Ciò fa di $\mathbb{Z}$ un anello euclideo.

Definizione 1.8 (Insieme $\mathbb{Q}$)**.** L'*insieme* $\mathbb{Q}$ dei numeri razionali è formato dai numeri esprimibili come rapporto tra due numeri interi, numeratore e denominatore, il secondo dei quali diverso da 0. L'insieme $\mathbb{Q}$ gode di alcune proprietà algebriche:

1. è chiuso rispetto alle operazioni di addizione $+$, sottrazione $-$, moltiplicazione $*$ e, escludendo lo 0, divisione $/$;
2. $(\mathbb{Q} \setminus \{0\}, *)$ costituisce un gruppo abeliano;
3. $(\mathbb{Q}, +, *)$ costituisce un campo (caso particolare di anello commutativo) ed un dominio d'integrità;
4. la chiusura algebrica dei numeri $\mathbb{Q}$ è formata dai numeri algebrici.

Possiamo vedere i numeri razionali come particolari numeri reali $\mathbb{R}$, in quanto esiste un isomorfismo tra i numeri reali dotati di parte decimale finita o periodica e i numeri razionali; i numeri reali che non sono razionali sono detti irrazionali (ad esempio $\sqrt{2}, e, \pi$).

1.4 Monomi e Polinomi

Definizione 1.9 (Monomio)**.** Un *monomio* è un prodotto di potenze di variabili con esponenti interi non negativi, definite parte letterale, moltiplicato per una costante diversa da 0, definita coefficiente. Possiamo trovare un monomio sotto diverse forme:

1. d, con $d \neq 0$: ogni variabile ha esponente 0;
2. $d * x^n$ con $d \neq 0, n \geq 0$: si considerano le potenze di una sola variabile;
3. $d * x^a y^b z^c$ con $d \neq 0, a, b$ e $c \geq 0$: si considerano più di una variabile.

Si definisce *grado* (o *ordine*) di un monomio la somma degli esponenti di tutte le variabili, includendo l'implicito esponente 1 quando omesso (ad esempio il monomio xyz^2 ha grado 4). Il numero dei monomi di grado d in n variabili è dato dal numero di combinazioni con ripetizione di d elementi scelti tra n variabili.

Definizione 1.10 (Polinomio)**.** Un *polinomio* è un'espressione costituita da variabili e coefficienti, utilizzando operazioni di somma $+$, sottrazione $-$, moltiplicazione $*$ ed esponenti interi non negativi.

Formalmente definiamo un polinomio in singola variabile come una combinazione lineare di $n+1$ coefficienti $(a_n, ..., a_0)$ e n variabili $(x^n, ..., x)$:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i$$

La valutazione di un polinomio consiste nel sostituire un valore numerico al posto di ogni variabile e svolgere le operazioni indicate. Per polinomi in singola variabile la valutazione è più efficiente sfruttando l'*algoritmo di Horner*:

$$((...((a_n x + a_{n-1})x + a_{n-2})x + ... + a_2)x + a_1)x + a_0$$

L'aritmetica dei polinomi prevede che:

1. la somma o il prodotto di due polinomi sia un polinomio;
2. la composizione di due polinomi sia un polinomio;
3. la derivata o la primitiva di un polinomio sia un polinomio.

1.5 Polinomi nella base di Bernstein

Definizione 1.11 (Polinomio nella base di Bernstein)**.** Un *polinomio nella base di Bernstein* $P(x)$ di grado n costituisce una particolare classe di polinomi sul campo reale, dato dalla formula:

$$P(x) = \sum_{k=0}^{n} c_k B_k^n(x)$$

Gli $n+1$ $B_k^n(\cdot)$ sono elementi della base dei polinomi di Bernstein di grado n definiti dalla formula:

$$B_k^n(x) = \binom{n}{k} \frac{(b-x)^{n-k}(x-a)^k}{(b-a)^n} \text{ se } x \in [a,b]$$

I primi elementi della base dei polinomi di Bernstein sono:

$$\begin{array}{llll} B_0^0(x) = 1 & & & \\ B_0^1(x) = 1-x & B_1^1(x) = x & & \\ B_0^2(x) = (1-x)^2 & B_1^2(x) = 2x(1-x) & B_2^2(x) = x^2 & \\ B_0^3(x) = (1-x)^3 & B_1^3(x) = 3x(1-x)^2 & B_2^3(x) = 3x^2(1-x) & B_3^3(x) = x^3 \end{array}$$

Alcune proprietà di cui godono gli elementi della base dei polinomi di Bernstein di grado n:

1. per convenzione poniamo $B_i^n(t) \equiv 0$ se $i < 0 \vee i < n$;
2. i polinomi di Bernstein partizionano l'unità: $\sum_{i=0}^{n} B_i^n(t) = 1$;
3. possono essere definiti per ricorsione: $B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)$;
4. formano una base di Π_n.

Capitolo 2

Una Library Java per Strutture Algebriche

La library è stata realizzata integrando un duplice aspetto, ovvero due differenti package contenuti nel principale *it.unipr.informatica*.

Vedremo dapprima una spiegazione delle varie funzionalità proposte nella sezione *structures* e successivamente quelle situate nella sezione *executors*. Per chiarezza espositiva si è scelto di non indicare nel codice proposto i vari package a cui è possibile far riferimento:

- *structures*, intesa come oggetti che possono essere paragonabili ad elementi di un anello:
  - *RingElement*;
  - *FieldElement*;
  - *VariableDomain*;
  - *Integer*;
  - *Rational*;
  - *Monomial*;
  - *Polynomial*;
  - *SparsePolynomial*;
  - *DensePolynomial*;
  - *BernsteinBasis*;
  - *BersnteinPolynomial*;

- *executors*, intesa come oggetti che possono costituire un anello degli elementi proposti:
  - *Ring*;
  - *Field*;
  - *IntegerRing*;
  - *RationalRing*;
  - *MonomialExecutor*;
  - *PolynomialRing*;
  - *SparsePolynomialRing*;
  - *DensePolynomialRing*;
  - *BernsteinPolynomialRing*.

2.1 Struttura delle classi e interfacce base

2.1.1 RingElement

```
public interface RingElement {
        public boolean equals(Object other);

        public boolean isZero();

        public boolean isOne();
}
```

L'interfaccia *RingElement* è il nucleo centrale del progetto realizzato: ad essa infatti si rifaranno la quasi totalità delle altre classi, in quanto tramite questa è possibile istanziare degli oggetti facenti parte di un anello.

La sua struttura è molto semplice: i tre metodi elencati verranno ridefiniti all'interno degli altri segmenti di codice; si occupano rispettivamente di confrontare se due elementi di un anello sono uguali oppure se l'elemento in questione corrisponde all'elemento 0 o 1, i termini neutri per quanto riguarda la somma e il prodotto, per esempio, di due numeri interi.

2.1.2 FieldElement

```
public interface FieldElement extends RingElement {
}
```

L'interfaccia *FieldElement* è una componente molto semplice appartenente al progetto: come notiamo non contiene nulla nella sua definizione, mentre verrà specificata più avanti quando parleremo di strutture algebriche ad anello.

2.1.3 VariableDomain

```
public class VariableDomain<T extends RingElement> {
        private final T min;
        private final T max;

        public VariableDomain(T min, T max) {
                this.min = min;
                this.max = max;
        }

        public T getMin() {
                return min;
        }

        public T getMax() {
                return max;
        }

        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + min.hashCode();
                result = prime * result + max.hashCode();

                return result;
        }
```

```
        @Override
        public boolean equals(Object obj) {
                if (!(obj instanceof VariableDomain<?>))
                return false;

                VariableDomain<?> other =
                (VariableDomain<?>) obj;

                return min.equals(other.min) &&
                max.equals(other.max);
        }
}
```

La classe *VariableDomain* si occupa di poter definire un dominio all'interno del quale considereremo le variabili utilizzate all'interno degli altri componenti.

Il dominio specificato ha due campi private corrispondenti al valore minimo e al valore massimo del tipo T specificato inizialmente, unitamente al costruttore con i due parametri e alle due funzioni getter per ritornare gli estremi desiderati.

Inoltre sono stati ridefiniti tramite override due metodi, uno dalla superclasse, *equals(Object obj)*, ed uno ereditato implicitamente, *hashCode()*.

2.1.4 Integer

```
public class Integer implements RingElement {
        private BigInteger value;

        public Integer(BigInteger value) {
                this.value = value;
        }

        public BigInteger getValue() {
                return value;
        }

        @Override
        public boolean equals(Object other) {
                return other instanceof Integer
                && value.equals(((Integer) other).getValue());
        }
```

```
        @Override
        public boolean isZero() {
                return value.equals(BigInteger.ZERO);
        }

        @Override
        public boolean isOne() {
                return value.equals(BigInteger.ONE);
        }

        @Override
        public int hashCode() {
                return value.hashCode();
        }

        @Override
        public String toString() {
                return value.toString();
        }
}
```

La classe *Integer* si occupa di poter definire un numero intero facente parte di un anello, implementando infatti l'interfaccia *RingElement*.

Il numero intero specificato ha un metodo private corrispondente al valore numerico, realizzato come *BigInteger* tramite l'apposita libreria standard; unitamente ad un costruttore con parametro e ad una funzione getter per ottenere questo valore.

Sono stati implementati tramite override dalla superclasse i tre metodi ereditati:

- *equals(Object other)*, funzione booleana per confrontare due elementi dove prima di paragonare i due valori si verifica che l'elemento other sia effettivamente un'istanza dei numeri interi;
- *isZero()*, funzione booleana per verificare se l'elemento corrisponde allo ZERO dei *BigInteger*;
- *isOne()*, funzione booleana per verificare se l'elemento corrisponde allo ONE dei *BigInteger*.

Infine sono stati realizzati tramite override implicito due metodi molto importanti, richiamando semplicemente le funzionalità specificate nella libreria *BigInteger*:

- *hashCode()*, funzione intera che restituisce il valore hash del numero intero, ovvero un termine che riassume l'istanziazione della classe in un singolo valore intero con segno a 32 bit;
- *toString()*, funzione che restituisce sotto forma di stringa il valore del numero intero, molto comoda per poter visualizzare il valore effettivo.

2.1.5 Rational

```
public class Rational implements RingElement {
        private BigInteger numerator;
        private BigInteger denominator;

        public Rational(BigInteger numerator,
                        BigInteger denominator) {
                if (denominator.equals(BigInteger.ZERO)) {
                        throw new IllegalArgumentException
                        ("Denominator can't be equal to zero");
                }

                BigInteger gcd = numerator.gcd(denominator);
                this.numerator = numerator.divide(gcd);
                this.denominator = denominator.divide(gcd);
        }

        public BigInteger getNumerator() {
                return numerator;
        }

        public BigInteger getDenominator() {
                return denominator;
        }

        @Override
        public boolean equals(Object other) {
                if (!(other instanceof Rational))
                        return false;
                Rational otherRational = (Rational) other;

                return numerator.equals(otherRational.numerator)
                && denominator.equals(otherRational.denominator);
        }
```

```
        @Override
        public boolean isZero() {
                return numerator.equals(BigInteger.ZERO);
        }

        @Override
        public boolean isOne() {
                return numerator.equals(denominator);
        }

        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + numerator.hashCode();
                result = prime * result + denominator.hashCode();
                return result;
        }

        @Override
        public String toString() {
                return numerator.toString() + "/"
                        + denominator.toString();
        }
}
```

La classe *Rational* si occupa di poter definire un numero razionale facente parte di un anello, implementando infatti l'interfaccia *RingElement*. Stiamo definendo concettualmente un aspetto molto simile a quello presentatosi con la classe precedente: essendo infatti $\mathbb{Q} \supseteq \mathbb{Z}$, ritroveremo tutte le caratteristiche precedente sotto un aspetto leggermente diverso.

Infatti a differenza di prima il numero razionale è definito tramite due campi private, corrispondenti al numeratore e al denominatore visti come *BigInteger*; il costruttore prevede un controllo sull'eventuale illegalità del denominatore ($= 0$) e anche la riduzione del numero razionale ai minimi termini tramite la funzione di MCD ($16/10 \rightarrow 8/5$). Le funzioni getter sono specifiche per numeratore e denominatore.

I tre metodi ereditati dalla superclasse sono stati implementati in maniera del tutto simile al precedente, sfruttando il controllo sul numeratore per *isZero()* e l'uguaglianza tra numeratore e denominatore per *isOne()*.

I due metodi ereditati implicitamente prevedono di richiamare le funzionalità di *BigInteger*:

- *hashCode()*, dove il valore hash viene generato identificando come termini chiave il valore hash del numeratore e del denominatore;
- *toString()*, una concatenazione tra la stringa del numeratore e quella del denominatore.

2.1.6 Monomial

```
public class Monomial<T extends RingElement> {
        private final T coefficient;
        private final List<java.lang.Integer> grades;

        public Monomial(T coefficient,
                        List<java.lang.Integer> grades) {
                for (int i : grades) {
                        if (i < 0)
                        throw new IllegalArgumentException
                        ("Grade can't be negative");
                }

                this.coefficient = coefficient;
                this.grades = new ArrayList<>(grades);
        }

        public Monomial(T coefficient,
                        java.lang.Integer... grades) {
                this(coefficient, Arrays.asList(grades));
        }

        public List<java.lang.Integer> getGrades() {
                return Collections.unmodifiableList(grades);
        }

        public T getCoefficient() {
                return coefficient;
        }

        public int variablesCount() {
                return grades.size();
        }
```

```
@Override
public boolean equals(Object obj) {
        if (!(obj instanceof Monomial<?>))
        return false;

        Monomial<?> objAsMonomial = (Monomial<?>) obj;

        return coefficient.equals(objAsMonomial
        .coefficient) && gradesEquals(objAsMonomial);
}

public boolean gradesEquals(Monomial<?> other) {
        if (variablesCount() != other.variablesCount())
        return false;

        for (int i = 0; i < variablesCount(); i++) {
                if (!grades.get(i)
                .equals(other.grades.get(i)))
                return false;
        }
        return true;
}

@Override
public boolean isZero() {
        return coefficient.isZero();
}

@Override
public boolean isOne() {
        if (!coefficient.isOne()) return false;
        for (java.lang.Integer grade : grades) {
                if (grade != 0) return false;
        }
        return true;
}

@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + coefficient.hashCode();
        for (int grade : grades) {
                result = prime * result + grade;
        }
        return result;
}
```

```
        @Override
        public String toString() {
                StringBuilder result =
                new StringBuilder(coefficient.toString());
                for (int i = 0; i < variablesCount(); i++) {
                        result.append("(x");
                        result.append(i + 1);
                        if (grades.get(i) > 1) {
                                result.append("^");
                                result.append(grades.get(i));
                        }
                        result.append(")");
                }

                return result.toString();
        }
}
```

La classe *Monomial* si occupa di poter definire un monomio di un tipo T facente parte di un anello, in quanto notiamo come T implementa infatti l'interfaccia *RingElement*. Ricordiamo che un monomio può essere espresso sotto forma del suo coefficiente, della sua variabile, dell'insieme delle sue variabili oppure combinando queste soluzioni ricordando tuttavia come l'unica operazione ammessa sia la moltiplicazione.

Per i monomi sono stati realizzati due campi private contenenti rispettivamente il coefficiente e gli esponenti delle variabili; sono stati inoltre previsti tre metodi getter per ottenere il grado, il coefficiente e il numero di variabili del monomio. Vale spendere qualche parola in più per l'implementazione dei due costruttori:

- *Monomial(T coefficient, List<java.lang.Integer> grades)*, per il quale specifichiamo un controllo sull'eventuale illegalità del grado proposto (< 0);
- *Monomial(T coefficient, java.lang.Integer... grades)*, che sfrutta il costruttore precedente e quindi delega i vari controlli. Troviamo inoltre la dichiarazione multipla con "..." la quale indica che verrà passato come parametro un oggetto di tipo array con 0 o più elementi.

Sono stati implementati tramite override dalla superclasse i tre metodi ereditati con l'aggiunta di un quarto metodo specifico:

- *equals(Object obj)*, funzione booleana che controlla che l'oggetto sia effettivamente un monomio di un tipo deciso a tempo di compilazione per poi verificare che sia uguale al termine di paragone;
- *gradesEquals(Monomial<?> other)*, funziona booleana per verificare che variabili corrispondenti tra due monomi abbiano uguali esponenti;
- *isZero()*, funzione booleana che controlla che il coefficiente sia uguale a 0;
- *isOne()*, funzione booleana per verificare che il coefficiente sia uguale a 1 e che tutti gli esponenti delle variabili siano di grado 0.

I due metodi ereditati implicitamente sono simili a quanto visto per le classi precedenti; *toString()* cataloga le variabili sotto forma di $x_1, x_2, ..., x_n$.

2.1.7 Polynomial

```
public interface Polynomial<T extends RingElement>
                                  extends RingElement {
}
```

L'interfaccia *Polynomial* è il nucleo di base per la ridefinizione delle classi che si occuperanno delle varie strutture polinomiali.

La sua concezione è basilare: i polinomi saranno riferiti ad un tipo T che rimanda agli elementi di un anello, ed a loro volta saranno a tutti gli effetti elementi di un anello.

2.1.8 SparsePolynomial

```
public class SparsePolynomial<T extends RingElement>
                                  implements Polynomial<T> {

        private List<Monomial<T>> monomials;
```

```
        public SparsePolynomial(
        Iterable<Monomial<T>> monomials) {
                this.monomials = new ArrayList<>();
                int variablesCount = 0;

                for (Monomial<T> monomial : monomials) {
                        if (!monomial.isZero()) {
                                if (variablesCount == 0) {
                                        variablesCount =
                                        monomial
                                        .variablesCount();
                                } else {
                                if (variablesCount !=
                                monomial.variablesCount()) {
                                        throw new
                                        IllegalArgumentException(
                                        "Monomials should have
                                        same number
                                        of variables");
                                }
                        }

                        this.monomials.add(monomial);
                        }
                }
        }

        @SafeVarargs
        public SparsePolynomial(Monomial<T>... monomials) {
                this(Arrays.asList(monomials));
        }

        public List<Monomial<T>> getMonomials() {
                return Collections.unmodifiableList(monomials);
        }

        @Override
        public boolean isZero() {
                return monomials.isEmpty();
        }

        @Override
        public boolean isOne() {
                if (monomials.size() != 1) return false;
                Monomial<T> monomial = monomials.get(0);
                return monomial.isOne();
        }
```

```
        @Override
        public String toString() {
                if (isOne()) return "1";
                if (isZero()) return "0";

                StringBuilder builder = new StringBuilder();
                for (int i = 0; i < monomials.size(); i++) {
                        Monomial<T> monomial = monomials.get(i);
                        if (i > 0)
                        builder.append(" + ");

                        builder.append("(");
                        builder.append(monomial);
                        builder.append(")");
                }

                return builder.toString();
        }

        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                for (Monomial<T> monomial : monomials) {
                        result = prime * result
                        + monomial.hashCode();
                }
                return result;
        }

        @Override
        public boolean equals(Object obj) {
                if (!(obj instanceof SparsePolynomial<?>))
                return false;

                SparsePolynomial<?> polynomial =
                (SparsePolynomial<?>) obj;

                if (monomials.size() !=
                polynomial.monomials.size()) return false;

                for (Monomial<?> monomial : monomials) {
                        if (!polynomial.monomials
                        .contains(monomial)) return false;
                }

                for (Monomial<?> monomial : polynomial
                                        .monomials) {
```

```
                        if (!monomials.contains(monomial))
                        return false;
                }

                return true;
        }
}
```

La classe *SparsePolynomial* si occupa di poter definire un polinomio in forma sparsa di un tipo T facente parte di un anello, in quanto T implementa infatti l'interfaccia *RingElement*.

Per polinomio sparso intendiamo una combinazione lineare di monomi con coefficiente non nullo, come tuple di termini coefficiente-esponenti: ciò fornisce una rappresentazione potenzialmente più compatta.

Il polinomio in forma sparsa prevede un campo private corrispondente ad una lista di monomi di tipo T: essendo una combinazione lineare di elementi possono essere visti come una somma algebrica dove l'operazione viene definita dal segno del coefficiente del monomio considerato (somma + per coefficienti > 0, differenza $-$ per coefficienti < 0).

I due costruttori presentano una struttura simile a quella vista in precedenza, con il primo che effettua un controllo sul numero delle variabili indicate e con il secondo che richiama quello soprastante, specificando inoltre un *@SafeVarargs*, in quanto ci stiamo riferendo ad un parametro con un numero non ben specificato di operandi ma non stiamo operando in maniera pericolosa, dato che l'obiettivo è quello di iterare su questi elementi. Il metodo getter si occupa di restituire l'insieme dei vari monomi.

I tre metodi ereditati dalla superclasse operano in maniera similare a quanto già fatto ma è utile specificare il comportamento delle funzioni:

- *isZero()*, funzione booleana che controlla se l'insieme dei monomi è vuoto;
- *isOne()*, funzione booleana che verifica che sia presente un solo monomio con coefficiente 1.

I due metodi ereditati implicitamente sono simili a quanto visto per le classi precedenti; *toString()* cataloga i monomi apponendo un simbolo "+" per separarli.

2.1.9 DensePolynomial

```
public class DensePolynomial<T extends RingElement>
                                implements Polynomial<T> {
        private List<T> coefficients;
        private List<java.lang.Integer> maxGrades;
        private int variablesCount;
        private List<java.lang.Integer> gradeWeights;

        public DensePolynomial(int variablesCount,
        List<Integer> maxGrades, List<T> coefficients) {
                this.coefficients =
                new ArrayList<>(coefficients);
                this.maxGrades = new ArrayList<>(maxGrades);
                this.variablesCount = variablesCount;

                gradeWeights =
                new ArrayList<>(Collections
                .nCopies(variablesCount, 1));
                int weight = 1;
                for (int i = 1; i < variablesCount; i++) {
                        weight *= (maxGrades.get(i - 1) + 1);
                        gradeWeights.set(i, weight);
                }
        }

        public List<T> getCoefficients() {
                return
                Collections.unmodifiableList(coefficients);
        }

        public List<Integer> getMaxGrades() {
                return
                Collections.unmodifiableList(maxGrades);
        }

        public int getVariablesCount() {
                return variablesCount;
        }

        @Override
        public boolean isZero() {
                for (T coef : coefficients) {
                        if (!coef.isZero())
                        return false;
                }
                return true;
        }
```

```
        @Override
        public boolean isOne() {
                if (coefficients.isEmpty())
                return false;

                if (!coefficients.get(0).isOne())
                return false;

                for (int i = 1; i < coefficients.size(); i++)
                        if (!coefficients.get(i).isZero())
                        return false;
                }

                return true;
        }

        @Override
        public String toString() {
                if (isOne()) return "1";
                if (isZero()) return "0";

                StringBuilder builder = new StringBuilder();
                int[] currentGrades = new int[variablesCount];
                int coefficientPointer = 0;
                boolean isFirst = true;
                while
                (coefficientPointer < coefficients.size()) {
                        if (!coefficients
                        .get(coefficientPointer).isZero()) {
                                if (!isFirst)
                                        builder.append(" + ");
                                isFirst = false;
                                builder.append("(");
                                builder.append(coefficients
                                .get(coefficientPointer)
                                .toString());
                                builder
                                .append(monomialString(
                                currentGrades));
                                builder.append(")");
                        }
                        ArrayHelper
                        .increaseGrade(currentGrades,
                        maxGrades, 0);
                        coefficientPointer++;
                }
                return builder.toString();
        }
```

```
        private String monomialString(int[] grades) {
                StringBuilder result = new StringBuilder();
                for (int i = 0; i < grades.length; i++) {
                        result.append("(");
                        result.append("x");
                        result.append(i + 1);
                        if (grades[i] > 1) {
                                result.append("^");
                                result.append(grades[i]);
                        }
                        result.append(")");
                }
                return result.toString();
        }

        @Override
        public boolean equals(Object obj) {
                if (!(obj instanceof DensePolynomial<?>))
                return false;

                DensePolynomial<?> polynomial
                = (DensePolynomial<?>) obj;
                if (variablesCount != polynomial.variablesCount)
                return false;

                for (int i = 0; i < variablesCount; i++) {
                        if (!maxGrades.get(i)
                        .equals(polynomial.maxGrades.get(i)))
                        return false;
                }

                for (int i = 0; i < coefficients.size(); i++) {
                        if (!coefficients.get(i)
                        .equals(polynomial.coefficients.get(i)))
                        return false;
                }

                return true;
        }
```

```
        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + variablesCount;
                for (int grade : maxGrades) {
                        result = prime * result + grade;
                }

                for (T coef : coefficients) {
                        result = prime * result
                        + coef.hashCode();
                }

                return result;
        }

        public T getCoefficientByGrades(int... grades) {
                if (grades.length != variablesCount)
                throw new IllegalArgumentException(
                "Wrong number of variables");

                int coefIndex = 0;
                for (int i = 0; i < variablesCount; i++) {
                        if (grades[i] > maxGrades.get(i))
                        return null;
                        coefIndex +=
                        gradeWeights.get(i) * grades[i];
                }

                return coefficients.get(coefIndex);
        }

        public List<java.lang.Integer>
        getGradesByCoefIndex(int coefI) {
                List<java.lang.Integer> result =
                new ArrayList<>(Collections
                .nCopies(variablesCount, 0));
                for (int i = variablesCount - 1; i >= 0; i--) {
                        result.set(i,
                        coefI / gradeWeights.get(i));
                        coefI = coefI % gradeWeights.get(i);
                }

                return result;
        }
}
```

La classe *DensePolynomial* si occupa di poter definire un polinomio in forma densa di un tipo T facente parte di un anello, in quanto T implementa infatti l'interfaccia *RingElement*.

Per polinomio denso intendiamo una combinazione lineare di monomi comprendente i coefficienti anche nulli di tutti i possibili termini: è una versione estesa dei polinomi.

Il polinomio in forma densa prevede quattro campi private che localizzano: una lista dei coefficienti dei monomi, una lista comprendente il grado massimo dei termini del polinomio, un valore che memorizza il numero di variabili e una lista che setta il peso di ogni termine.

Il costruttore prevede di inizializzare un oggetto della classe in questione fissando i primi tre campi private e ricavando di conseguenza il peso di ogni termine: di conseguenza i metodi getter sono specificati per i primi tre campi.

I metodi ereditati dalla superclasse e quelli ereditati implicitamente sono simili a quanto visto per le classi precedenti; sono state implementati due metodi che si occupano di due aspetti reciproci:

- *getCoefficientByGrades(int... grades)*, ritorna il coefficiente specifico per il grado passato come argomento;
- *getGradesByCoefIndex(int coefI)*, ritorna la lista dei gradi dei termini specifici per il coefficiente passato come argomento.

2.1.10 BernsteinBasis

```
public class BernsteinBasis {
        private final int n;
        private final int k;
        private final long coef;

        public BernsteinBasis(int n, int k) {
                this.n = n;
                this.k = k;
                this.coef = binomialCoefficient(n, k);
        }

        private long binomialCoefficient(int n, int k) {
                BigInteger numerator = BigInteger.ONE;
                BigInteger denominator = BigInteger.ONE;
                for (int i = 1; i <= k; i++) {
                        numerator = numerator.multiply(BigInteger
                        .valueOf(n + 1 - i));
                        denominator = denominator
```

```
                        .multiply(BigInteger.valueOf(i));
                }
                return numerator.divide(denominator).longValue();
        }

        public int getN() {
                return n;
        }

        public int getK() {
                return k;
        }

        public long getCoef() {
                return coef;
        }
}
```

La classe *BernsteinBasis* si occupa di poter definire la base di un polinomio in forma di Bernstein secondo la definizione fornita nel capitolo introduttivo.

La base di un polinomio in forma di Bernstein prevede due campi private corrispondenti ai due termini del coefficiente binomiale e ad un ulteriore campo private dove localizziamo il risultato dell'operazione binomiale: i metodi getter corrispondono alle funzioni per ottenere ognuno dei campi private specificati.

Il metodo private *binomialCoefficient(int n, int k)* ha la funzione di calcolare il coefficiente binomiale tra i due numeri interi passati come argomenti della funzione.

2.1.11 BernsteinPolynomial

```
public class BernsteinPolynomial<T extends RingElement>
implements Polynomial<T> {
        private int variablesCount;
        private int grade;
        private List<T> coefficients;
        private List<VariableDomain<T>> variableDomains;

        public BernsteinPolynomial(int grade,
        int variablesCount, List<T> coefficients,
        List<VariableDomain<T>> variableDomains) {
                long coefficientsCount = (long) Math
                .pow(grade + 1, variablesCount);
                if (coefficientsCount != coefficients.size())
```

```
                throw new IllegalArgumentException("
                Number of coefficients must be equal
                to (grade+1)^variablesCount");

                if (variableDomains.size() != variablesCount)
                throw new IllegalArgumentException("
                Number of variables domains must
                equal count of variables");

                this.grade = grade;
                this.variablesCount = variablesCount;
                this.coefficients = Collections
                .unmodifiableList(
                new ArrayList<>(coefficients));
                this.variableDomains = Collections
                .unmodifiableList(
                new ArrayList<>(variableDomains));
        }

        public int getGrade() {
                return grade;
        }

        public int getVariablesCount() {
                return variablesCount;
        }

        public List<T> getCoefficients() {
                return coefficients;
        }

        public List<VariableDomain<T>> getVariableDomains() {
                return variableDomains;
        }

        @Override
        public boolean isZero() {
                for (T i : coefficients) {
                        if (!i.isZero()) return false;
                }

                return true;
        }
```

```
        @Override
        public boolean isOne() {
                if (!coefficients.get(0).isOne()) return false;

                for (int i = 1; i < coefficients.size(); i++) {
                        if (!coefficients.get(i).isZero())
                        return false;
                }

                return true;
        }

        @Override
        public String toString() {
                StringBuilder builder = new StringBuilder();

                List<java.lang.Integer> maxGrades =
                Collections.nCopies(variablesCount, grade);
                int[] grades = new int[variablesCount];
                boolean isNotFirst = false;

                for (T coef : coefficients) {
                        if (!coef.isZero()) {
                                if (isNotFirst)
                                        builder.append(" + ");
                                isNotFirst = true;
                                builder.append("(");
                                builder.append(coef);
                                builder.append(")");
                                for (int i = 0;
                                i < variablesCount; i++) {
                                        builder.append("b(");
                                        builder
                                        .append(grades[i]);
                                        builder.append(",");
                                        builder.append(grade);
                                        builder.append(")");
                                        builder.append("(x");
                                        builder.append(i + 1);
                                        builder.append(")");
                                }
                        }
                        ArrayHelper.increaseGrade(grades,
                        maxGrades, 0);
                }

                return builder.toString();
        }
```

```
        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + variablesCount;
                result = prime * result + grade;

                for (T coef : coefficients) {
                        result = prime * result
                        + coef.hashCode();
                }

                for (VariableDomain<T> domain
                : variableDomains) {
                        result = prime * result
                        + domain.hashCode();
                }

                return result;
        }

        @Override
        public boolean equals(Object obj) {
                if (!(obj instanceof BernsteinPolynomial<?>))
                return false;

                BernsteinPolynomial<?> other =
                (BernsteinPolynomial<?>) obj;

                if (variablesCount != other.variablesCount)
                return false;
                if (grade != other.grade) return false;
                for (int i = 0; i < coefficients.size(); i++) {
                        if (!coefficients.get(i).equals(other
                        .coefficients.get(i))) return false;
                }

                for (int i = 0; i < variablesCount; i++) {
                        if (!variableDomains.get(i).equals(other
                        .variableDomains.get(i))) return false;
                }

                return true;
        }
}
```

La classe *BernsteinPolynomial* si occupa di poter definire un polinomio in forma di Bernstein per elementi appartenenti ad un anello, implementando di fatto l'interfaccia *Polynomial<T>*.

Il polinomio in forma di Bernstein prevede quattro campi private corrispondenti al numero delle variabili presenti e al grado del polinomio, insieme alla lista dei coefficienti utilizzati e a quelle con i domini delle variabili utilizzate.

Il costruttore prevede di inizializzare un oggetto della classe in questione fissando i campi private secondo gli argomenti passati come parametro, effettuando un controllo sul numero dei coefficienti e delle variabili. Le funzioni getter non fanno altro che ritornare i valori fondamentali della classe.

I metodi ereditati dalla superclasse e quelli ereditati implicitamente sono simili a quanto visto per le classi precedenti.

2.2 Classi del package *executors*

2.2.1 Ring

```
public interface Ring<T extends RingElement> {
        T sum(T a, T b) throws UnsupportedOperationException;

        T product(T a, T b)
        throws UnsupportedOperationException;

        T pow(T a, int exponent)
        throws UnsupportedOperationException;

        T zero();

        T one();
}
```

L'interfaccia *Ring* permette la realizzazione della struttura algebrica di anello, delineata sotto varie forme a seconda del tipo T che viene utilizzato come *RingElement*.

Contiene cinque funzioni che verranno spesso ridefinite all'interno delle classi executors che realizzano le operazioni di base per la struttura ad anello:

- *sum(T a, T b)*, restituisce un elemento di tipo T dato dalla somma di due elementi di tipo T;
- *product(T a, T b)*, restituisce un elemento di tipo T dato dal prodotto di due elementi di tipo T;
- *pow(T a, int exponent)*, restituisce un elemento di tipo T dato dalla potenza tra un elemento di tipo T e un esponente intero;
- *zero()*, restituisce l'elemento di tipo T che funge da elemento neutro per la somma;
- *one()*, restituisce l'elemento di tipo T che funge da elemento neutro per il prodotto.

2.2.2 Field

```
public interface Field<T extends FieldElement> extends Ring<T> {
        T product(T a, int b);

        T negate(T a);

        default T subtraction(T a, T b)
        throws UnsupportedOperationException {
                return sum(a, negate(b));
        }

        T invert(T a);

        default T divide(T a, T b)
        throws UnsupportedOperationException {
                return product(a, invert(b));
        }

        T divide(T a, int b);
}
```

L'interfaccia *Field* permette la realizzazione della struttura algebrica di campo, delineata sotto varie forme a seconda del tipo T che viene utilizzato come *FieldElement*, visto come un'estensione del concetto di anello.

Contiene sei funzioni che verranno spesso ridefinite all'interno delle classi executors che realizzano le operazioni di base per la struttura a campo:

- *product(T a, int b)*, restituisce un elemento di tipo T dato dal prodotto di un elemento di tipo T e uno intero;
- *negate(T a)*, restituisce un elemento di tipo T corrispondente all'opposto di quello passato come parametro;
- *subtraction(T a, T b)*, restituisce un elemento di tipo T dato dalla sottrazione di due elementi di tipo T;
- *invert(T a)*, restituisce un elemento di tipo T corrispondente all'inverso di quello passato come parametro;
- *divide(T a, T b)*, restituisce un elemento di tipo T dato dalla divisione di due elementi di tipo T;
- *divide(T a, int b)*, restituisce un elemento di tipo T dato dalla divisione di un elemento di tipo T e uno intero.

2.2.3 IntegerRing

```
public class IntegerRing implements Ring<Integer> {
        @Override
        public Integer sum(Integer a, Integer b) {
                return new Integer(a.getValue()
                        .add(b.getValue()));
        }

        @Override
        public Integer product(Integer a, Integer b) {
                return new Integer(a.getValue()
                        .multiply(b.getValue()));
        }

        @Override
        public Integer pow(Integer a, int exponent) {
                return new Integer(a.getValue().pow(exponent));
        }

        @Override
        public Integer zero() {
                return new Integer(BigInteger.ZERO);
        }
```

```
        @Override
        public Integer one() {
                return new Integer(BigInteger.ONE);
        }

        public Integer valueOf(long value) {
                return new Integer(BigInteger.valueOf(value));
        }
}
```

La classe *IntegerRing* si occupa di poter definire una struttura algebrica ad anello nel campo dei numeri interi $\mathbb{Z}$, implementando di fatto l'interfaccia *Ring<Integer>*.

I cinque metodi ereditati dalla superclasse vengono ridefiniti secondo le specifiche richieste dall'insieme dei numeri interi, inoltre viene aggiunto un ulteriore metodo *valueOf(long value)* per indicare il valore di un elemento dell'anello:

- *sum(Integer a, Integer b)*, restituisce un *Integer* dato dalla somma di due *Integer*;
- *product(Integer a, Integer b)*, restituisce un *Integer* dato dal prodotto di due *Integer*;
- *pow(Integer a, int exponent)*, restituisce un *Integer* dato dalla potenza tra un *Integer* e un esponente intero;
- *zero()*, restituisce un *Integer* corrispondente all'elemento ZERO dei *BigInteger*;
- *one()*, restituisce un *Integer* che corrisponde all'elemento ONE dei *BigInteger*.

2.2.4 RationalRing

```
public class RationalRing implements Ring<Rational> {
        @Override
        public Rational sum(Rational a, Rational b) {
                return new Rational(
                a.getNumerator().multiply(b.getDenominator())
                .add(b.getNumerator()
                .multiply(a.getDenominator())),
                a.getDenominator().multiply(b.getDenominator())
                );
        }

        @Override
        public Rational product(Rational a, Rational b) {
                return new Rational(
                a.getNumerator().multiply(b.getNumerator()),
                a.getDenominator().multiply(b.getDenominator())
                );
        }

        @Override
        public Rational pow(Rational a, int exponent) {
                return new Rational(
                a.getNumerator().pow(exponent),
                a.getDenominator().pow(exponent)
                );
        }

        @Override
        public Rational zero() {
                return new Rational(BigInteger.ZERO,
                        BigInteger.ONE);
        }

        @Override
        public Rational one() {
                return new Rational(BigInteger.ONE,
                        BigInteger.ONE);
        }
}
```

La classe *RationalRing* si occupa di poter definire una struttura algebrica ad anello nel campo dei numeri interi $\mathbb{Q}$, implementando di fatto l'interfaccia *Ring<Rational>*.

La ridefinizione dei cinque metodi ereditati dalla superclasse avviene in maniera del tutto analoga a quanto determinato in precedenza: per convenzione fissiamo l'elemento restituito da:

- *zero()*, come *Rational(BigInteger.ZERO/BigInteger.ONE)*;
- *one()*, come *Rational(BigInteger.ONE/BigInteger.ONE)*.

2.2.5 MonomialExecutor

```
public class MonomialExecutor<T extends RingElement> {
        private Ring<T> ring;

        public MonomialExecutor(Ring<T> ring) {
                this.ring = ring;
        }

        public Monomial<T> sum(Monomial<T> a, Monomial<T> b)
        throws it.unipr.informatica
        .UnsupportedOperationException {
                if (!a.gradesEquals(b)) throw new
                UnsupportedOperationException("Only monomials
                with equlas grades can be summed");

                return new Monomial<>(ring.sum(a
                .getCoefficient(), b.getCoefficient()),
                a.getGrades());
        }

        public Monomial<T> product(Monomial<T> a, Monomial<T> b)
        throws it.unipr.informatica
        .UnsupportedOperationException {
                if (a.variablesCount() != b.variablesCount())
                throw new UnsupportedOperationException(
                "Can't multiply monomials with different
                count of variables");
                T newCoefficient =
                ring.product(a.getCoefficient(),
                b.getCoefficient());
                List<Integer> newGrades =
                new ArrayList<>(a.variablesCount());
                for (int i = 0; i < a.variablesCount(); i++) {
                        newGrades.add(a.getGrades().get(i)
                        + b.getGrades().get(i));
                }
                return new Monomial<>(newCoefficient, newGrades);
        }
```

```
        public T evaluate(Monomial<T> monomial, List<T> values)
        throws UnsupportedOperationException {
                if (values.size() < monomial.variablesCount())
                throw new
                IllegalArgumentException("Not enough values");

                T result = monomial.getCoefficient();
                for (int i = 0; i < monomial.variablesCount();
                                                        i++) {
                        T poweredVariable = ring
                        .pow(values.get(i),
                        monomial.getGrades().get(i));
                        result = ring.product(result,
                        poweredVariable);
                }
                return result;
        }

        @SafeVarargs
        public final T evaluate(Monomial<T> monomial,
                                        T... values)
        throws UnsupportedOperationException {
                return evaluate(monomial, Arrays.asList(values));
        }

        public Monomial<T> one() {
                return new Monomial<>(ring.one(), 0);
        }
}
```

La classe *MonomialExecutor* si occupa di poter definire una struttura algebrica simile all'anello per i monomi di tipo T, dove T è un tipo compatibile con un elemento di un anello.

Un campo private contiene un anello di tipo T che conterrà i termini del monomio creato come istanza della classe. I metodi implementati permettono di operare per quanto riguarda: somma, moltiplicazione, valutazione del monomio ed elemento neutro rispetto alla moltiplicazione.

2.2.6 PolynomialRing

```
public interface PolynomialRing<T extends RingElement>
                        extends Ring<Polynomial<T>> {
        T evaluate(Polynomial<T> polynomial, List<T> values)
        throws it.unipr.informatica
        .UnsupportedOperationException;

        T evaluate(Polynomial<T> polynomial, T... values)
        throws it.unipr.informatica
        .UnsupportedOperationException;
}
```

L'interfaccia *PolynomialRing* permette la realizzazione della struttura algebrica ad anello per i polinomi, delineata sotto varie forme a seconda del tipo T che viene utilizzato come *RingElement*, estendo concretamente l'interfaccia *Ring<Polynomial<T>>*.

Contiene due funzioni che verranno spesso ridefinite all'interno delle classi executors che realizzano le operazioni di base per la struttura ad anello per i polinomi: si occupano della valutazione di un polinomio di tipo T, tramite il passaggio di una lista di valori di tipo T predefinita oppure di un insieme qualsiasi.

2.2.7 SparsePolynomialRing

```
public class SparsePolynomialRing<T extends RingElement>
                        implements PolynomialRing<T> {
        private Ring<T> baseRing;
        private MonomialExecutor<T> monomialExecutor;

        public SparsePolynomialRing(Ring<T> baseRing) {
                this.baseRing = baseRing;
                this.monomialExecutor =
                new MonomialExecutor<>(baseRing);
        }
```

```
        @Override
        public Polynomial<T> sum(Polynomial<T> a,
                                 Polynomial<T> b)
        throws UnsupportedOperationException {
                if (a.isZero())
                return b;

                if (b.isZero())
                return a;

                List<Monomial<T>> newMonomials =
                new ArrayList<>(toSparsePolynomial(a)
                .getMonomials());

                newMonomials.addAll(toSparsePolynomial(b)
                .getMonomials());

                return simplify(new
                SparsePolynomial<>(newMonomials));
        }

        private SparsePolynomial<T> simplify(
        SparsePolynomial<T> before)
        throws UnsupportedOperationException {
                List<Monomial<T>> newMonomials =
                new ArrayList<>(before
                .getMonomials().size());
                for (Monomial<T> forInserting : before
                                        .getMonomials()) {
                        boolean insert = true;
                        for (int i = 0; i < newMonomials.size();
                                                        i++) {
                                Monomial<T> existed =
                                newMonomials.get(i);
                                if (existed
                                .gradesEquals(forInserting)) {
                                        newMonomials.set(i,
                                         monomialExecutor.sum(
                                         existed, forInserting));
                                        insert = false;
                                        break;
                                }
                        }
                        if (insert)
                        newMonomials.add(forInserting);
                }
                return new SparsePolynomial<>(newMonomials);
        }
```

```
@Override
public Polynomial<T> product(Polynomial<T> a,
Polynomial<T> b)
throws UnsupportedOperationException {
        if (a.isZero() || b.isZero())
        return zero();

        if (a.isOne())
        return b;

        if (b.isOne())
        return a;

        SparsePolynomial<T> sparseA =
        toSparsePolynomial(a);
        SparsePolynomial<T> sparseB =
        toSparsePolynomial(b);

        List<Monomial<T>> newMonomials =
        new ArrayList<>(sparseA.getMonomials().size());

        try {
                for (Monomial<T> am : sparseA
                .getMonomials()) {
                        for (Monomial<T> bm : sparseB
                        .getMonomials()) {
                                newMonomials
                                .add(monomialExecutor
                                .product(am, bm));
                        }

                }
        } catch (it.unipr.informatica
        .UnsupportedOperationException ex) {
                throw new UnsupportedOperationException(
                "Can't multiply polynomials with
                different number of variables");
        }

return simplify(new SparsePolynomial<>(newMonomials));
}
```

```
@Override
public Polynomial<T> pow(Polynomial<T> a, int exponent)
throws UnsupportedOperationException {
        if (exponent < 0)
        throw new ArithmeticException("
        Exponent can't be negative");

        if (exponent == 0)
        return one();

        if (exponent == 1)
        return a;

        SparsePolynomial<T> sparseA =
        toSparsePolynomial(a);

        Polynomial<T> result;
        Polynomial<T> square = product(sparseA, sparseA);
        Polynomial<T> poweredSquare =
        pow(square, exponent / 2);

        if (exponent % 2 == 1) {
                result = product(poweredSquare, a);
        } else {
                result = poweredSquare;
        }

        return simplify(toSparsePolynomial(result));
}

@Override
public SparsePolynomial<T> zero() {
        return new
        SparsePolynomial<>(Collections.emptyList());
}

@Override
public SparsePolynomial<T> one() {
        return new
        SparsePolynomial<>(monomialExecutor.one());
}
```

```
@Override
public T evaluate(Polynomial<T> polynomial,
List<T> values)
throws UnsupportedOperationException {
        T result = baseRing.zero();
        SparsePolynomial<T> sparsePolynomial =
        toSparsePolynomial(polynomial);
        for (Monomial<T> monomial :
        sparsePolynomial.getMonomials()) {
                result = baseRing.sum(result,
                monomialExecutor.evaluate(monomial,
                values));
        }

        return result;
}

@SafeVarargs
@Override
public final T evaluate(Polynomial<T> polynomial,
T... values)
throws UnsupportedOperationException {
        return evaluate(polynomial,
        Arrays.asList(values));
}

public SparsePolynomial<T>
toSparsePolynomial(Polynomial<T> polynomial) {
        if (polynomial instanceof DensePolynomial<?>)
        return
        denseToSparse((DensePolynomial<T>) polynomial);

        if (polynomial instanceof SparsePolynomial<?>)
        return (SparsePolynomial<T>) polynomial;

        throw new IllegalArgumentException(
        "Unknown polynomial type");
}
```

```
        private SparsePolynomial<T>
        denseToSparse(DensePolynomial<T> dense) {
                List<Monomial<T>> monomials =
                new LinkedList<>();

                for (int i = 0; i < dense.getCoefficients()
                                .size(); i++) {
                        T coef = dense.getCoefficients().get(i);
                        if (!coef.isZero()) {
                                List<java.lang.Integer>
                                grades =
                                dense.getGradesByCoefIndex(i);
                                monomials
                                .add(new
                                Monomial<>(coef, grades));
                        }
                }

                return new SparsePolynomial<>(monomials);
        }
}
```

La classe *SparsePolynomialRing* si occupa di poter definire una struttura algebrica ad anello per i polinomi in forma sparsa, implementando di fatto l'interfaccia *PolynomialRing<T>*.

La classe prevede due metodi private per la gestione di due anelli, uno contenente la base dell'anello, ovvero i coefficienti, ed uno contenente le variabili dei vari monomi. Il costruttore prevede di inizializzare un elemento tramite l'anello fornito come base.

La ridefinizione dei cinque metodi ereditati dalla superclasse rispetta le considerazioni formulate nel capitolo precedente; è utile illustrare la presenza di una funzione *simplify(SparsePolynomial<T> before)* che agisce sull'anello fornito come argomento per una rielaborazione che permette di ottenere una forma semplificata. Verrà infatti utilizzata all'interno dei tre metodi che gestiscono le operazioni di questo anello (somma, moltiplicazione ed elevamento a potenza).

Sono state scritte anche due funzioni che permettono la valutazione dell'anello generato da questi polinomi seconda una lista di valori, che può essere fornita in input oppure può essere specificata in maniera generica utilizzando la dichiarazione multipla con "..." come visto nel capitolo precedente.

Inoltre sono stati implementati due metodi che, insieme a quelli scritti per i polinomi in forma densa, permettono una una conversione da anelli di polinomi in forma sparsa a forma densa e viceversa:

- *toSparsePolynomial(Polynomial<T> polynomial)*, converte un generico polinomio a polinomio in forma sparsa;
- *densetoSparse(DensePolynomial<T> dense)*, converte un polinomio in forma densa in un polinomio in forma sparsa.

2.2.8 DensePolynomialRing

```
public class DensePolynomialRing<T extends RingElement>
implements PolynomialRing<T> {
        private Ring<T> baseRing;
        private MonomialExecutor<T> monomialExecutor;

        public DensePolynomialRing(Ring<T> baseRing) {
                this.baseRing = baseRing;
                this.monomialExecutor =
                new MonomialExecutor<>(baseRing);
        }

        public DensePolynomial<T> create(
                        List<Monomial<T>> monomials) {
                if (monomials.isEmpty()) {
                        return new
                        DensePolynomial<>(0,
                        Collections.emptyList(),
                        Collections.emptyList());
                }

                int variablesCount = monomials.get(0)
                .variablesCount();
                List<java.lang.Integer> maxGrades =
                new ArrayList<>(Collections
                .nCopies(variablesCount, 0));
                for (Monomial<T> monomial : monomials)
                        for (int i = 0; i < variablesCount;
                                                    i++)
                                if (maxGrades.get(i) < monomial
                                        .getGrades().get(i))
                                        maxGrades.set(i,
                                        monomial.getGrades()
                                        .get(i));
```

```
                int coefficientsCount = 1;
                int[] variablesGradesWeights =
                new int[variablesCount];
                for (int i = 0; i < variablesCount; i++) {
                        int grade = maxGrades.get(i);
                        variablesGradesWeights[i] =
                        coefficientsCount;
                        coefficientsCount *= (grade + 1);
                }

                List<T> coefficients =
                new ArrayList<>(Collections
                .nCopies(coefficientsCount, baseRing.zero()));

                for (Monomial<T> monomial : monomials) {
                        int coefficientIndex = 0;
                        for (int i = 0; i < variablesCount;
                                                        i++) {
                                coefficientIndex +=
                                variablesGradesWeights[i] *
                                monomial.getGrades().get(i);
                        }
                        coefficients.set(coefficientIndex,
                        monomial.getCoefficient());

                }

                return new DensePolynomial<>(
                variablesCount, maxGrades, coefficients);
        }

        @SafeVarargs
        public final DensePolynomial<T> create(
                        Monomial<T>... monomials) {
                return create(Arrays.asList(monomials));
        }
```

```
        @Override
        public Polynomial<T> sum(Polynomial<T> a,
        Polynomial<T> b)
        throws UnsupportedOperationException {
                if (a.isZero())
                return b;

                if (b.isZero())
                return a;

                DensePolynomial<T> denseA = toDensePolynomial(a);
                DensePolynomial<T> denseB = toDensePolynomial(b);

                if (denseA.getVariablesCount() !=
                denseB.getVariablesCount())
                throw new UnsupportedOperationException(
                "Can't sum polynomials with
                different count of variables");

                int variablesCount = denseA.getVariablesCount();
                List<java.lang.Integer> maxGrades =
                new ArrayList<>(Collections
                .nCopies(variablesCount, 0));
                for (int i = 0; i < variablesCount; i++) {
                        maxGrades.set(i, Math.max(
                        denseA.getMaxGrades().get(i),
                        denseB.getMaxGrades().get(i)));
                }

                int coefficientsCount = 1;
                for (int i = 0; i < variablesCount; i++) {
                        int grade = maxGrades.get(i);
                        coefficientsCount *= (grade + 1);
                }

                List<T> coefficients =
                new ArrayList<>(Collections
                .nCopies(coefficientsCount, baseRing.zero()));

                int[] grades = new int[variablesCount];
                for (int i = 0; i < coefficientsCount; i++) {
                        T coefA = denseA
                        .getCoefficientByGrades(grades);
                        if (coefA == null)
                        coefA = baseRing.zero();

                        T coefB = denseB
                        .getCoefficientByGrades(grades);
```

```
                    if (coefB == null)
                    coefB = baseRing.zero();

                    coefficients.set(i,
                    baseRing.sum(coefA, coefB));

                    ArrayHelper
                    .increaseGrade(grades, maxGrades, 0);
            }

            return new DensePolynomial<>(variablesCount,
            maxGrades, coefficients);
    }

    @Override
    public Polynomial<T> product(Polynomial<T> a,
    Polynomial<T> b)
    throws UnsupportedOperationException {
            if (a.isZero() || b.isZero())
            return zero();

            if (a.isOne())
            return b;

            if (b.isOne())
            return a;

            DensePolynomial<T> denseA = toDensePolynomial(a);
            DensePolynomial<T> denseB = toDensePolynomial(b);

            if (denseA.getVariablesCount() !=
            denseB.getVariablesCount())
            throw new UnsupportedOperationException("
            Can't multiply polynomials with
            different count of variables");

            int variablesCount = denseA.getVariablesCount();
            List<java.lang.Integer> maxGrades =
            new ArrayList<>(Collections
            .nCopies(variablesCount, 0));
            for (int i = 0; i < variablesCount; i++) {
                    maxGrades.set(i,
                    denseA.getMaxGrades().get(i) +
                    denseB.getMaxGrades().get(i));
            }

            int coefficientsCount = 1;
            int[] variablesGradesWeights =
            new int[variablesCount];
```

```
            for (int i = 0; i < variablesCount; i++) {
                    int grade = maxGrades.get(i);
                    variablesGradesWeights[i] =
                    coefficientsCount;
                    coefficientsCount *= (grade + 1);
            }

            List<T> coefficients =
            new ArrayList<>(Collections
            .nCopies(coefficientsCount, baseRing.zero()));
            for (int i = 0; i < denseA
            .getCoefficients().size(); i++) {
                    List<java.lang.Integer>
                    multiplicandGrades =
                    denseA.getGradesByCoefIndex(i);
                    for (int j = 0; j <
                    denseB.getCoefficients().size(); j++) {
                            List<java.lang.Integer>
                            multiplierGrades =
                            denseB.getGradesByCoefIndex(j);
                            int coefficientIndex = 0;
                            for (int k = 0; k <
                            variablesCount; k++) {
                                    coefficientIndex +=
                                    variablesGradesWeights[k]
                                    * (multiplicandGrades
                                    .get(k) +
                                    multiplierGrades.get(k));
                            }

                            T multiplyingResult =
                            baseRing.product(denseA
                            .getCoefficients().get(i),
                            denseB.getCoefficients().get(j));

                            coefficients
                            .set(coefficientIndex,
                            baseRing.sum(coefficients
                            .get(coefficientIndex),
                            multiplyingResult));
                    }
            }

            return new DensePolynomial<>(
            variablesCount, maxGrades, coefficients);
    }
```

```
@Override
public Polynomial<T> pow(Polynomial<T> a, int exponent)
throws UnsupportedOperationException {
        if (exponent < 0)
        throw new ArithmeticException("
        Exponent can't be negative");

        if (exponent == 0)
        return one();

        if (exponent == 1)
        return a;

        DensePolynomial<T> denseA = toDensePolynomial(a);

        Polynomial<T> result;
        Polynomial<T> square = product(denseA, denseA);
        Polynomial<T> poweredSquare =
        pow(square, exponent / 2);

        if (exponent % 2 == 1) {
                result = product(poweredSquare, a);
        } else {
                result = poweredSquare;
        }

        return result;
}

@Override
public DensePolynomial<T> zero() {
        return create(Collections.emptyList());
}

@Override
public DensePolynomial<T> one() {
        return create(monomialExecutor.one());
}
```

```
        @Override
        public T evaluate(Polynomial<T> polynomial,
        List<T> values)
        throws UnsupportedOperationException {
                DensePolynomial<T> densePolynomial =
                toDensePolynomial(polynomial);
                T result = baseRing.zero();

                for (int i = 0; i < densePolynomial
                .getCoefficients().size(); i++) {
                        T coef = densePolynomial
                        .getCoefficients().get(i);
                        if (!coef.isZero()) {
                                T subres = coef;
                                List<java.lang.Integer>
                                grades = densePolynomial
                                .getGradesByCoefIndex(i);
                                for (int j = 0; j <
                                densePolynomial
                                .getVariablesCount(); j++) {
                                        subres =
                                        baseRing.product(subres,
                                        baseRing.pow(values
                                        .get(j),
                                        grades.get(j)));
                                }

                                result = baseRing
                                .sum(result, subres);
                        }
                }

                return result;
        }

        @Override
        @SafeVarargs
        public final T evaluate(Polynomial<T> polynomial,
        T... values)
        throws UnsupportedOperationException {
                return evaluate(polynomial,
                Arrays.asList(values));
        }
```

```
        public DensePolynomial<T>
        toDensePolynomial(Polynomial<T> polynomial) {
                if (polynomial instanceof DensePolynomial<?>)
                return (DensePolynomial<T>) polynomial;

                if (polynomial instanceof SparsePolynomial<?>)
                return
                sparseToDense((SparsePolynomial<T>) polynomial);

                throw new IllegalArgumentException("
                Unknown polynomial type");
        }

        private DensePolynomial<T>
        sparseToDense(SparsePolynomial<T> sparse) {
                return create(sparse.getMonomials());
        }
}
```

La classe *DensePolynomialRing* si occupa di poter definire una struttura algebrica ad anello per i polinomi in forma densa, implementando di fatto l'interfaccia *PolynomialRing<T>*.

La classe prevede due metodi private per la gestione di due anelli, uno contenente la base dell'anello, ovvero i coefficienti, ed uno contenente le variabili dei vari monomi. Il costruttore prevede di inizializzare un elemento tramite l'anello fornito come base.

La ridefinizione dei cinque metodi ereditati dalla superclasse rispetta le considerazioni formulate nel capitolo precedente; sono stati inoltre implementati due metodi per poter maneggiare agevolmente la creazione di un polinomio in forma densa:

- *create(List<Monomial<T>> monomials)*, consente di poter creare un polinomio in forma densa a partire da una lista di termini monomiali;
- *create(Monomial<T>... monomials)*, consente di poter creare un polinomio in forma densa a partire da una lista generica di monomi.

Sono state scritte anche due funzioni che permettono la valutazione dell'anello generato da questi polinomi seconda una lista di valori, che può essere fornita in input oppure può essere specificata in maniera generica utilizzando la dichiarazione multipla con "..." come visto nel capitolo precedente.

Inoltre sono stati implementati due metodi che, insieme a quelli scritti per i polinomi in forma sparsa, permettono una una conversione da anelli di polinomi in forma densa a forma sparsa e viceversa:

- *toDensePolynomial(Polynomial<T> polynomial)*, converte un generico polinomio a polinomio in forma densa;
- *sparseToDense(SparsePolynomial<T> dense)*, converte un polinomio in forma sparsa in un polinomio in forma densa.

2.2.9 BernsteinPolynomialRing

```
public class BernsteinPolynomialRing<T extends FieldElement>
implements Ring<BernsteinPolynomial<T>> {
        private Field<T> baseField;

        public BernsteinPolynomialRing(Field<T> baseField) {
                this.baseField = baseField;
        }

        @Override
        public BernsteinPolynomial<T> sum(
        BernsteinPolynomial<T> a, BernsteinPolynomial<T> b)
        throws UnsupportedOperationException {
                if (a.isZero())
                return b;

                if (b.isZero())
                return a;

                if (a.getVariablesCount() !=
                b.getVariablesCount())
                throw new UnsupportedOperationException(
                "Can't sum polynomials with
                different count of variables");
                for (int i = 0; i < a.getVariablesCount(); i++) {
                        if (!a.getVariableDomains().get(i)
                        .equals(b.getVariableDomains().get(i)))
                        throw new UnsupportedOperationException("
                        Can't sum polynomials with
                        different variable domains");
                }

                int variablesCount = a.getVariablesCount();
                int grade = Math.max(a.getGrade(), b.getGrade());
                a = increaseGrade(a, grade);
```

```
                b = increaseGrade(b, grade);

                int coefficientsCount =
                (int) Math.pow(grade + 1, variablesCount);

                List<T> coefficients =
                new ArrayList<>(a.getCoefficients());

                for (int i = 0; i < coefficientsCount; i++) {
                        coefficients.set(i,
                        baseField.sum(coefficients.get(i),
                        b.getCoefficients().get(i)));
                }

                return new BernsteinPolynomial<>(grade,
                variablesCount, coefficients,
                a.getVariableDomains());
        }

        private BernsteinPolynomial<T>
        increaseGrade(BernsteinPolynomial<T> polynomial,
        int goalGrade) throws UnsupportedOperationException {
                if (polynomial.getGrade() >= goalGrade)
                return polynomial;

                int variablesCount =
                polynomial.getVariablesCount();
                int grade = polynomial.getGrade() + 1;
                int coefficientsCount =
                (int) Math.pow(grade + 1, variablesCount);
                List<T> coefficients =
                new ArrayList<>(Collections
                .nCopies(coefficientsCount, baseField.zero()));

                List<java.lang.Integer> baseMaxGrades =
                Collections.nCopies(variablesCount, grade);
                int[] baseGrades = new int[variablesCount];

                for (int i = 0;
                i < polynomial.getCoefficients().size(); i++) {
                        T baseCoef = polynomial
                        .getCoefficients().get(i);

                        List<java.lang.Integer> maxGrades =
                        Collections.nCopies(variablesCount, 1);
                        int[] grades = new int[variablesCount];

                        do {
                                int[] summedGrades = ArrayHelper
```

```
                                .sumArray(baseGrades, grades);
                                int coefIndex = 0;
                                int coef = 1;
                                for (int j = 0;
                                j < variablesCount; j++) {
                                        if (grades[j] == 0) {
                                                coef *=
                                                (grade -
                                                baseGrades[j])
                                                / grade;
                                        } else {
                                        coef *=
                                        (baseGrades[j] + 1)
                                        / grade;
                                }

                                coefIndex += summedGrades[j]
                                * Math.pow(grade + 1, j);
                        }

                        T oldValue = coefficients.get(coefIndex);
                        result polynomial
                        coefficients.set(coefIndex,
                        baseField.sum(oldValue,
                        baseField.product(baseCoef, coef)));

                } while (ArrayHelper
                .increaseGrade(grades, maxGrades, 0));

                ArrayHelper
                .increaseGrade(baseGrades, baseMaxGrades, 0);
        }

        return
        increaseGrade(new BernsteinPolynomial<>
        (grade, variablesCount, coefficients,
        polynomial.getVariableDomains()), goalGrade);
}
```

```
@Override
public BernsteinPolynomial<T>
product(BernsteinPolynomial<T> a,
BernsteinPolynomial<T> b)
throws UnsupportedOperationException {
        if (a.isZero() || b.isZero())
        return zero();

        if (a.isOne())
        return b;

        if (b.isOne())
        return a;

        if (a.getVariablesCount() != b.getVariablesCount())
        throw new UnsupportedOperationException(
        "Can't multiply polynomials with
        different count of variables");

        for (int i = 0; i < a.getVariablesCount(); i++) {
                if (!a.getVariableDomains().get(i)
                .equals(b.getVariableDomains().get(i)))
                throw new UnsupportedOperationException(
                "Can't sum polynomials with
                different variable domains");
        }

        int variablesCount = a.getVariablesCount();
        int grade = a.getGrade() + b.getGrade();
        int coefficientsCount =
        (int) Math.pow(grade + 1, variablesCount);
        List<T> coefficients =
        new ArrayList<>(Collections
        .nCopies(coefficientsCount, baseField.zero()));

        List<java.lang.Integer> aMaxGrades =
        Collections.nCopies(variablesCount, a.getGrade());
        int[] aGrades = new int[variablesCount];

        int n = a.getGrade();
        int m = b.getGrade();

        for (T aCoef : a.getCoefficients()) {
                List<java.lang.Integer> bMaxGrades =
                Collections
                .nCopies(variablesCount,
                b.getGrade());
                int[] bGrades = new int[variablesCount];
```

```
                for (T bCoef : b.getCoefficients()) {
                        T baseCoef =
                        baseField.product(aCoef, bCoef);
                        if (!baseCoef.isZero()) {
                                T resultCoef = baseField.one();
                                for (int g = 0;
                                g < variablesCount; g++) {
                                        int i = aGrades[g];
                                        int j = bGrades[g];

                                        T numerator =
                                        baseField
                                        .product(baseField.one(),
                                        binomialCoefficient(n, i)
                                        * binomialCoefficient(
                                        m, j));
                                        T denominator =
                                        baseField
                                        .product(baseField.one(),
                                        binomialCoefficient(
                                        n + m, i + j));
                                        T coef =
                                        baseField
                                        .divide(numerator,
                                        denominator);
                                        resultCoef =
                                        baseField
                                        .product(resultCoef,
                                        coef);
                                }
                                int[] resultGrade =
                                ArrayHelper
                                .sumArray(aGrades, bGrades);
                                int coefIndex =
                                getCoefficientIndexByGrades(
                                grade, resultGrade);
                                T oldValue =
                                coefficients.get(coefIndex);
                                coefficients.set(coefIndex,
                                baseField.sum(oldValue,
                                baseField
                                .product(baseCoef, resultCoef)));
                        }

                        ArrayHelper
                        .increaseGrade(bGrades, bMaxGrades, 0);
                }
                ArrayHelper
                .increaseGrade(aGrades, aMaxGrades, 0);
```

```
        }
        return new
        BernsteinPolynomial<>(grade, variablesCount,
        coefficients, a.getVariableDomains());
}

@Override
public BernsteinPolynomial<T>
pow(BernsteinPolynomial<T> a, int exponent)
throws UnsupportedOperationException {
        if (exponent < 0)
        throw new ArithmeticException("Exponent
        can't be negative");

        if (exponent == 0)
        return one();

        if (exponent == 1)
        return a;

        BernsteinPolynomial<T> result;
        BernsteinPolynomial<T> square = product(a, a);
        BernsteinPolynomial<T> poweredSquare =
        pow(square, exponent / 2);

        if (exponent % 2 == 1) {
        result = product(poweredSquare, a);
} else {
result = poweredSquare;
}

return result;
}

@Override
public BernsteinPolynomial<T> zero() {
        return new BernsteinPolynomial<>(0, 0,
        Arrays.asList(baseField.zero()),
        Collections.emptyList());
}

@Override
public BernsteinPolynomial<T> one() {
        return new BernsteinPolynomial<>(0, 0,
        Arrays.asList(baseField.one()), Collections.emptyList());
}
```

```
public T evaluate(BernsteinPolynomial<T> polynomial,
List<T> values) throws UnsupportedOperationException {
        int variablesCount = polynomial.getVariablesCount();
        List<java.lang.Integer> maxGrades =
        Collections.nCopies(variablesCount,
        polynomial.getGrade());
        int[] grades = new int[variablesCount];
        List<VariableDomain<T>> domains =
        polynomial.getVariableDomains();

        T result = baseField.zero();
        for (T coef : polynomial.getCoefficients()) {
                T basisResult = coef;
                for (int i = 0; i < variablesCount; i++) {
                        T firstMultiplier = baseField
                        .pow(baseField
                        .subtraction(values.get(i),
                        domains.get(i).getMin()), grades[i]);
                        T secondMultiplier = baseField
                        .pow(baseField
                        .subtraction(domains.get(i).getMax(),
                        values.get(i)),
                        polynomial.getGrade() - grades[i]);
                        int thirdMultiplier =
                        binomialCoefficient(polynomial
                        .getGrade(), grades[i]);
                        T denominator = baseField
                        .pow(baseField
                        .subtraction(domains.get(i).getMax(),
                        domains.get(i).getMin()), grades[i]);

                        T variableResult = baseField.divide(
                        baseField.product(
                        baseField.product(firstMultiplier,
                        secondMultiplier),
                        thirdMultiplier),
                        denominator);

                        basisResult = baseField
                        .product(basisResult, variableResult);
                }
                result = baseField.sum(result, basisResult);
                ArrayHelper.increaseGrade(grades, maxGrades, 0);
        }

        return result;
}
```

```
        public final
        T evaluate(BernsteinPolynomial<T> polynomial,
        T... values)
        throws UnsupportedOperationException {
                return
                evaluate(polynomial, Arrays.asList(values));
        }
}
```

La classe *BernsteinPolynomialRing* si occupa di poter definire una struttura algebrica ad anello per i polinomi in forma di Bernstein, implementando di fatto l'interfaccia *Ring<BernsteinPolynomial<T>>*.

La classe prevede un campo private per la gestione del campo costituito dai polinomi in forma di Bernstein; il costruttore prevede di inizializzare un elemento tramite il campo fornito come base.

La ridefinizione dei cinque metodi ereditati dalla superclasse rispetta le considerazioni formulate nel capitolo precedente; inoltre sono state scritte anche due funzioni che permettono la valutazione dell'anello generato da questi polinomi seconda una lista di valori, che può essere fornita in input oppure può essere specificata in maniera generica utilizzando la dichiarazione multipla con "..." come visto nel capitolo precedente.

Il metodo private *increaseGrade(BernsteinPolynomial<T> polynomial, int goalGrade)* permette di poter estendere la definizione del polinomio in forma di Bernstein passato come argomento ad un grado superiore rispetto a quello secondo cui è stato creato.

2.3 Gestione delle eccezioni

2.3.1 UnsupportedOperationException

```
public class UnsupportedOperationException extends Exception {
        public UnsupportedOperationException(String operation) {
                super("Unsupported operation: " + operation);
        }
}
```

Una classe molto semplice per la visualizzazione dell'operazione non supportata tramite una stringa testuale.

Capitolo 3

Unit Test della Library

Per realizzare la struttura di testing sono state create delle classi apposite contenute nel package *it.unipr.informatica*:

- *structures*, in riferimento alle classi del relativo package descritto nel capitolo precedente, contenente le classi:
  - *IntegerRingElementTest*;
  - *VariableDomainTest*;
  - *RationalTest*;
  - *MonomialTest*;
  - *SparsePolynomialTest*;
  - *DensePolynomialTest*;
  - *BernsteinBasisTest*;
  - *BernsteinPolynomialTest*;
- *executors*, in riferimento alle classi del relativo package descritto nel capitolo precedente, contenente le classi:
  - *IntegerRingTest*;
  - *RationalRingTest*;
  - *MonomialExecutorTest*;
  - *SparsePolynomialRingTest*;
  - *DensePolynomialRingTest*;
  - *BernsteinPolynomialRingTest*.

3.1 Test delle classi base

3.1.1 IntegerRingElementTest

```
public class IntegerRingElementTest {
        IntegerRing ring = new IntegerRing();
        Integer a = ring.valueOf(42);

        @Test
        public void testGetValue() {
                BigInteger expected = bi(42);

                assertEquals(expected, a.getValue());
        }

        @Test
        public void testEqualsWithOtherInteger() {
                Integer a = new Integer(bi(42));
                Object b = new Integer(bi(83));

                assertFalse(a.equals(b));
        }

        @Test
        public void testEqualsWithSameInteger();

        @Test
        public void testEqualsWithNotInteger();

        @Test
        public void testIsZeroForZero() {
                Integer a = ring.zero();
                assertTrue(a.isZero());
        }

        @Test
        public void testIsZeroForNonZero();

        @Test
        public void testIsOneForOne();

        @Test
        public void testIsOneForNotOne() {
                Integer a = ring.zero();
                assertFalse(a.isOne());
        }
```

```
        @Test
        public void testToString() {
                BigInteger value = bi(42);
                Integer integer = new Integer(value);

                assertEquals(value.toString(),
                        integer.toString());
        }

        @Test
        public void testHashCode() {
                BigInteger value = bi(42);
                Integer integer = new Integer(value);

                assertEquals(value.hashCode(),
                        integer.hashCode());
        }
}
```

La classe *IntegerRingElementTest* prevede due elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di interi e un elemento inizializzato appartenente all'anello.

I metodi proposti si occupano di verificare: l'assegnamento, l'uguaglianza e la disuguaglianza tra elementi dell'anello di interi, la presenza o meno dell'elemento neutro per la somma e per la moltiplicazione, il funzionamento dei metodi *toString()* e *hashCode()*.

3.1.2 VariableDomainTest

```
public class VariableDomainTest {
        @Test
        public void testGetMin() {
                Integer expected = integer(5);
                VariableDomain<Integer> domain =
                new VariableDomain<>(expected, integer(7));

                assertEquals(expected, domain.getMin());
        }
```

```
        @Test
        public void testGetMax() {
                Integer expected = integer(5);
                VariableDomain<Integer> domain =
                new VariableDomain<>(integer(3), expected);

                assertEquals(expected, domain.getMax());
        }
}
```

La classe *VariableDomainTest* prevede due metodi che si occupano di verificare ed ottenere l'elemento minimo e massimo di un dominio di interi specificato.

3.1.3 RationalTest

```
public class RationalTest {
        RationalRing ring = new RationalRing();
        Rational a = new Rational(bi(2), bi(13));
        Rational b = new Rational(bi(57), bi(17));

        @Test(expected = IllegalArgumentException.class)
        public void testThatDenominatorCannotBeNull() {
                new Rational(bi(0), bi(0));
        }

        @Test(expected = IllegalArgumentException.class)
        public void testGetNumerator();

        @Test
        public void testGetDenominator();

        @Test
        public void testEqualsWithOtherRational();

        @Test
        public void testEqualsWithSameRational() {
                Object sameAsA = new Rational(bi(2), bi(13));

                assertTrue(a.equals(sameAsA));
        }
```

```
        @Test
        public void testEqualAfterSimplicity() {
                Rational first = new Rational(bi(1), bi(2));
                Rational second = new Rational(bi(2), bi(4));

                assertTrue(first.equals(second));
        }

        @Test
        public void testEqualsWithNotRational();

        @Test
        public void testIsZeroForZero();

        @Test
        public void testIsZeroForNonZero() {
                Rational a = ring.one();
                assertFalse(a.isZero());
        }


        @Test
        public void testIsOneForOne() {
                Rational a = ring.one();
                assertTrue(a.isOne());
        }

        @Test
        public void testIsOneForNotOne();

        @Test
        public void testToString();

        @Test
        public void testHashCode();
}
```

La classe *RationalTest* prevede due elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di razionali e un elemento inizializzato appartenente all'anello.

I metodi proposti si occupano di verificare: il denominatore mai nullo, l'uguaglianza e la disuguaglianza tra elementi dell'anello di razionali, l'efficacia della riduzione ai minimi termini, la presenza o meno dell'elemento neutro per la somma e per la moltiplicazione, il funzionamento dei metodi *toString()* e *hashCode()*.

3.1.4 MonomialTest

```
public class MonomialTest {
        Integer coef = integer(2);
        Monomial<Integer> a = new Monomial<>(coef, 1, 2, 3);

        @Test(expected = IllegalArgumentException.class)
        public void testCreateMonomialWithNegativeGrades() {
                new Monomial<>(integer(1), -1);
        }

        @Test
        public void testCreateMonomialWithZeroGrades();

        @Test
        public void testGetCoefficient();

        @Test
        public void testVariablesCount();

        @Test
        public void testToString();

        @Test
        public void
                testEqualsWithMonomialWithOtherCoefficient() {
                Monomial<Integer> other =
                new Monomial<>(integer(258), 1, 2, 3);
                assertFalse(a.equals(other));
        }

        @Test
        public void testEqualsWithMonomialWithOtherGrades();

        @Test
        public void
                testEqualsWithMonomial
                WithOtherCountOfVariables();

        @Test
        public void testEqualsWithNotMonomial() {
                assertFalse(a.equals(new Object()));
        }

        @Test
        public void testEqualsWithSameMonomial();
```

```
        @Test
        public void testEqualsWithMonomialOfOtherType() {
                Object other =
                new Monomial<>(new RationalRing()
                .one(), 1, 2, 3);

                assertFalse(a.equals(other));
        }

        @Test
        public void testHashCode() {
                int expected = 984129;

                assertEquals(expected, a.hashCode());
        }
}
```

La classe *MonomialTest* prevede due elementi di riferimento che verranno sfruttati per le varie verifiche: un coefficiente intero e un elemento inizializzato sotto forma di monomio.

I metodi proposti si occupano di verificare: il grado di un termine sempre positivo, il funzionamento dei costruttori e delle funzioni getter, l'uguaglianza e la disuguaglianza tra monomi con numero di variabili o grado differenti, il funzionamento dei metodi *toString()* e *hashCode()*.

3.1.5 SparsePolynomialTest

```
public class SparsePolynomialTest {
        SparsePolynomialRing<Integer> ring =
        new SparsePolynomialRing<>(new IntegerRing());
        SparsePolynomial<Integer> a =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6, 8)
        ));

        SparsePolynomial<Integer> b =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(5), 1, 2, 3),
        new Monomial<>(integer(4), 3, 3, 7)
        ));
```

```
@Test(expected = IllegalArgumentException.class)
public void testCreatePolynomial
FromMonomialsWIthDifferentSetOfVariables() {
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6)
        ));
}

@Test
public void testEqualsWithOtherPolynomial();

@Test
public void testEqualsWithSamePolynomial() {
        Object sameAsA =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6, 8)
        ));

        assertTrue(a.equals(sameAsA));
}

@Test
public void testEqualsWithNotPolynomial();

@Test
public void testIsZeroForZero();

@Test
public void testIsZeroForNonZero() {
        SparsePolynomial<Integer> a = ring.one();
        assertFalse(a.isZero());
}


@Test
public void testIsOneForOne();
}

@Test
public void testIsOneForNotOne() {
        SparsePolynomial<Integer> a = ring.zero();
        assertFalse(a.isOne());
}
```

```
        @Test
        public void testToString() {
                String expected = "(2(x1)(x2^2)(x3^3))
                + (8(x1^4)(x2^6)(x3^8))";

                assertEquals(expected, a.toString());
        }

        @Test
        public void testHashCode();
}
```

La classe *SparsePolynomialTest* prevede tre elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di polinomi in forma sparsa e due elementi inizializzati appartenenti ai polinomi in forma sparsa.

I metodi proposti si occupano di verificare: l'inizializzazione corretta di un polinomio con elementi aventi le stesse variabili, l'uguaglianza e la disuguaglianza tra elementi dell'anello dei polinomi in forma sparsa, la presenza o meno dell'elemento neutro per la somma e per la moltiplicazione, il funzionamento dei metodi *toString()* e *hashCode()*.

3.1.6 DensePolynomialTest

```
public class DensePolynomialTest {
        MonomialExecutor<Integer> monomialExecutor =
        new MonomialExecutor<>(new IntegerRing());
        DensePolynomialRing<Integer> ring =
        new DensePolynomialRing<>(new IntegerRing());
        DensePolynomial<Integer> a =
        ring.create(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6, 8)
        ));

        DensePolynomial<Integer> b =
        ring.create(Arrays.asList(
        new Monomial<>(integer(5), 1, 2, 3),
        new Monomial<>(integer(4), 3, 3, 7)
        ));

        @Test
        public void testEqualsWithOtherPolynomial();
```

```
        @Test
        public void testEqualsWithPolynomial
        WithOtherVariablesCount() {
                DensePolynomial<Integer> c =
                ring.create(Arrays.asList(
                new Monomial<>(integer(5), 1, 2, 3, 5),
                new Monomial<>(integer(4), 3, 3, 7, 5)
                ));

                assertFalse(a.equals(c));
        }

        @Test
        public void testEqualsWithPolynomialWith
        SameMonomialsButDifferentCoeffs() {
                DensePolynomial<Integer> c =
                ring.create(Arrays.asList(
                new Monomial<>(integer(3), 1, 2, 3),
                new Monomial<>(integer(7), 4, 6, 8)
                ));

                assertFalse(a.equals(c));
        }

        @Test
        public void testEqualsWithSamePolynomial();

        @Test
        public void testEqualsWithNotPolynomial();
        }

        @Test
        public void testIsZeroForZero() {
                DensePolynomial<Integer> a = ring.zero();
                assertTrue(a.isZero());
        }

        @Test
        public void testIsZeroForNonZero();

        @Test
        public void testIsOneForOne();
```

```
        @Test
        public void testIsOneForZero() {
                DensePolynomial<Integer> zero = ring.zero();
                assertFalse(zero.isOne());
        }

        @Test
        public void testIsOneForPolynomial
        WithOneMonomial() {
                DensePolynomial<Integer> a =
                ring.create(Arrays.asList(
                monomialExecutor.one(),
                new Monomial<>(integer(2), 1, 2, 3),
                new Monomial<>(integer(8), 4, 6, 8)
                ));
                assertFalse(a.isOne());
        }

        @Test
        public void testToString();

        @Test
        public void testHashCode();
}
```

La classe *DensePolynomialTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di polinomi in forma sparsa, un anello di monomi e due elementi inizializzati appartenenti all'anello di polinomi in forma sparsa.

I metodi proposti si occupano di verificare: l'inizializzazione corretta di un polinomio con elementi aventi le stesse variabili, l'uguaglianza e la disuguaglianza tra elementi dell'anello dei polinomi in forma sparsa, la presenza o meno dell'elemento neutro per la somma e per la moltiplicazione, il funzionamento dei metodi *toString()* e *hashCode()*.

3.1.7 BernsteinBasisTest

```
public class BernsteinBasisTest {
        int n = 7;
        int k = 5;
        BernsteinBasis basis = new BernsteinBasis(n, k);
```

```
        @Test
        public void testGetN() throws Exception {
                assertEquals(n, basis.getN());
        }

        @Test
        public void testGetK() throws Exception {
                assertEquals(k, basis.getK());
        }

        @Test
        public void testGetCoef() throws Exception {
                long expected = 21;
                assertEquals(expected, basis.getCoef());
        }
}
```

La classe *BernsteinBasisTest* prevede tre elementi di riferimento che verranno sfruttati per le varie verifiche: due interi che corrispondono ai termini del coefficiente binomiale e una base del polinomio in forma di Bernstein inizializzata secondo questi termini.

I metodi proposti si occupano di verificare che gli elementi dalla base corrispondano a quelli sfruttati dalla classe di testing.

3.1.8 BernsteinPolynomialTest

```
public class BernsteinPolynomialTest {
        BernsteinPolynomialRing<Rational> ring =
        new BernsteinPolynomialRing<>(new RationalRing());

        List<VariableDomain<Rational>> domains = Arrays.asList(
        new VariableDomain<>(rational(2), rational(5)),
        new VariableDomain<>(rational(3), rational(6))
        );

        List<Rational> aCoefs = Arrays.asList(
        rational(1), rational(2), rational(3),
        rational(4), rational(5), rational(6),
        rational(7), rational(8), rational(9)
        );

        BernsteinPolynomial<Rational> a =
        new BernsteinPolynomial<>(2, 2, aCoefs, domains);
```

```
List<Rational> bCoefs = Arrays.asList(
rational(4), rational(5), rational(6),
rational(7), rational(8), rational(9),
rational(1), rational(2), rational(3)
);

BernsteinPolynomial<Rational> b =
new BernsteinPolynomial<>(2, 2, bCoefs, domains);

@Test
public void testEqualsWithOtherPolynomial() {
        assertFalse(a.equals(b));
}

@Test
public void testEqualsWithPolynomial
WithOtherVariablesCount();

@Test
public void testEqualsWithPolynomial
WithDifferentCoeffs();

@Test
public void testEqualsWithSamePolynomial() {
        BernsteinPolynomial<Rational> sameAsA =
        new BernsteinPolynomial<>(2, 2, aCoefs, domains);

        assertTrue(a.equals(sameAsA));
}

@Test
public void testEqualsWithNotPolynomial();

@Test
public void testIsZeroForZero();

@Test
public void testIsZeroForNonZero() {
        BernsteinPolynomial<Rational> a = ring.one();
        assertFalse(a.isZero());
}

@Test
public void testIsOneForOne() {
        BernsteinPolynomial<Rational> one = ring.one();
        assertTrue(one.isOne());
}
```

```
        @Test
        public void testIsOneForZero();

        @Test
        public void testToString();

        @Test
        public void testHashCode();
}
```

La classe *BernsteinPolynomialTest* prevede tre elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di polinomi in forma di Bernstein e due elementi appartenenti all'anello inizializzati, comprendenti di dominio delle variabii.

I metodi proposti si occupano di verificare: l'inizializzazione corretta di un polinomio con elementi aventi le stesse variabili, l'uguaglianza e la disuguaglianza tra elementi dell'anello dei polinomi in forma di Bernstein, la presenza o meno dell'elemento neutro per la somma e per la moltiplicazione, il funzionamento dei metodi *toString()* e *hashCode()*.

3.2 Test delle classi del package *executors*

3.2.1 IntegerRingTest

```
public class IntegerRingTest {
        IntegerRing ring = new IntegerRing();
        Integer a = new Integer(bi(42));
        Integer b = new Integer(bi(83));
        Integer c = new Integer(bi(118));

        @Test
        public void testAddOtherInteger() {
                Integer expected = new Integer(bi(42 + 83));
                Integer actual = ring.sum(a, b);

                assertEquals(expected, actual);
        }

        @Test
        public void testAddingCommutativity() {
                assertEquals(ring.sum(a, b), ring.sum(b, a));
        }
```

```
        @Test
        public void testAddingAssociativity();

        @Test
        public void testAddingWithZero();

        @Test
        public void testMultiplyWithOtherInteger() {
                Integer expected = new Integer(bi(42 * 83));
                Integer actual = ring.product(a, b);

                assertEquals(expected, actual);
        }

        @Test
        public void testMultiplyingCommutativity();

        @Test
        public void testMultiplyingAssociativity() {
                assertEquals(ring.product(ring
                .product(a, b), c),
                ring.product(a, ring.product(b, c)));
        }

        @Test
        public void testDistributivity();

        @Test
        public void testPowWithPositiveExponent();

        @Test
        public void testPowWithZeroExponent();

        @Test(expected = ArithmeticException.class)
        public void testPowWithNegativeExponent() {
                Integer base = new Integer(bi(2));

                ring.pow(base, -10);
        }

        @Test
        public void testONE();

        @Test
        public void testZERO();
}
```

La classe *IntegerRingTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di interi e tre elementi inizializzati appartenenti agli interi.

I metodi proposti si occupano di verificare: le operazioni di somma, moltiplicazione ed elevamento a potenza per numeri interi, le proprietà commutative, associative e distributive riferite alle operazioni, l'elemento neutro per la somma e per la moltiplicazione, l'illegalità di un elevamento a potenza con esponente negativo.

3.2.2 RationalRingTest

```
public class RationalRingTest {
        RationalRing ring = new RationalRing();
        Rational a = new Rational(bi(2), bi(13));
        Rational b = new Rational(bi(57), bi(17));
        Rational c = new Rational(bi(12), bi(7));

        @Test
        public void testAddOtherRational()
        throws UnsupportedOperationException;

        @Test
        public void testAddingCommutativity()
        throws UnsupportedOperationException;

        @Test
        public void testAddingAssociativity()
        throws UnsupportedOperationException {
                assertEquals(ring.sum(ring.sum(a, b), c),
                ring.sum(a, ring.sum(b, c)));
        }

        @Test
        public void testAddingWithZero()
        throws UnsupportedOperationException;

        @Test
        public void testMultiplyWithOtherRational()
        throws UnsupportedOperationException {
                Rational expected =
                new Rational(bi(2 * 57), bi(13 * 17));
                RingElement actual = ring.product(a, b);

                assertEquals(expected, actual);
        }
```

```
        @Test
        public void testMultiplyingCommutativity()
        throws UnsupportedOperationException;

        @Test
        public void testMultiplyingAssociativity()
        throws UnsupportedOperationException;

        @Test
        public void testDistributivity()
        throws UnsupportedOperationException {
                assertEquals(ring.product(a,
                ring.sum(b, c)), ring.sum(ring.product(a, b),
                ring.product(a, c)));
        }

        @Test
        public void testPowWithPositiveExponent();

        @Test
        public void testPowWithZeroExponent() {
                Rational expected = ring.one();
                RingElement actual = ring.pow(a, 0);

                assertEquals(expected, actual);
        }

        @Test(expected = ArithmeticException.class)
        public void testPowWithNegativeExponent();

        @Test
        public void testONE() {
                assertEquals(BigInteger.ONE,
                ring.one().getNumerator());
                assertEquals(BigInteger.ONE,
                ring.one().getDenominator());
        }

        @Test
        public void testZERO();
}
```

La classe *RationalRingTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di razionali e tre elementi inizializzati appartenenti ai razionali.

I metodi proposti si occupano di verificare: le operazioni di somma, moltiplicazione ed elevamento a potenza per numeri razionali, le proprietà commutative, associative e distributive riferite alle operazioni, l'elemento neutro per la somma e per la moltiplicazione, l'illegalità di un elevamento a potenza con esponente negativo.

3.2.3 MonomialExecutorTest

```
public class MonomialExecutorTest {
        MonomialExecutor<Integer> executor =
        new MonomialExecutor<>(new IntegerRing());
        Monomial<Integer> a =
        new Monomial<>(integer(2), 1, 2, 3);
        Monomial<Integer> b =
        new Monomial<>(integer(3), 3, 2, 2);
        Monomial<Integer> c =
        new Monomial<>(integer(3), 3, 2, 2, 0);

        @Test
        public void testMultiply
        WithSameNumberOfVariables()
        throws it.unipr.informatica
        .UnsupportedOperationException {
                Monomial<Integer> ab =
                new Monomial<>(integer(6), 4, 4, 5);

                assertEquals(executor.product(a, b), ab);
        }

        @Test(expected =
        UnsupportedOperationException.class)
        public void testMultiply
        WithDifferentNumberOfVariables()
        throws UnsupportedOperationException;

        @Test
        public void testEvaluateNormal()
        throws UnsupportedOperationException {
                List<Integer> values
                = Arrays.asList(integer(5),
                integer(6), integer(8));
                Integer expected =
                integer(2 * 5 * 6 * 6 * 8 * 8 * 8);
                assertEquals(expected,
                executor.evaluate(a, values));
        }
```

```
        @Test(expected =
        IllegalArgumentException.class)
        public void testEvaluate
        WithoutRequiredNumberOfValues()
        throws UnsupportedOperationException;
        }

        @Test
        public void testEvaluate
        WithExcessNumberOfValues()
        throws UnsupportedOperationException {
                List<Integer> values =
                Arrays.asList(integer(5), integer(6),
                integer(8), integer(2048));
                Integer expected =
                integer(2 * 5 * 6 * 6 * 8 * 8 * 8);

                assertEquals(expected,
                executor.evaluate(a, values));
        }
}
```

La classe *MonomialExecutorTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di monomi e tre elementi inizializzati appartenenti ai monomi.

I metodi proposti si occupano di verificare: la moltiplicazione tra monomi con un numero di variabili uguale e la relativa impossibilità a svolgerla altrimenti, l'operazione di valutazione tra elementi diversi.

3.2.4 SparsePolynomialRingTest

```
public class SparsePolynomialRingTest {
        SparsePolynomialRing<Integer> ring =
        new SparsePolynomialRing<>(new IntegerRing());

        SparsePolynomial<Integer> a =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6, 8)
        ));
```

```
        SparsePolynomial<Integer> b =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(5), 1, 2, 3),
        new Monomial<>(integer(4), 3, 3, 7)
        ));
        SparsePolynomial<Integer> c =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(2), 4, 6, 8),
        new Monomial<>(integer(1), 3, 3, 7)
        ));

        @Test
        public void testAddOtherSparsePolynomial()
        throws UnsupportedOperationException {
                SparsePolynomial<Integer> expected =
                new SparsePolynomial<>(Arrays.asList(
                new Monomial<>(integer(7), 1, 2, 3),
                new Monomial<>(integer(8), 4, 6, 8),
                new Monomial<>(integer(4), 3, 3, 7)
                ));
                assertEquals(expected, ring.sum(a, b));
        }

        @Test
        public void testAddingCommutativity()
        throws UnsupportedOperationException;

        @Test
        public void testAddingAssociativity()
        throws UnsupportedOperationException {
                assertEquals(ring.sum(ring.sum(a, b), c),
                ring.sum(a, ring.sum(b, c)));
        }

        @Test
        public void testAddingWithZero()
        throws UnsupportedOperationException;

        @Test(expected = UnsupportedOperationException.class)
        public void testMultiplyWithOtherSparsePolynomial()
        throws it.unipr.informatica
        .UnsupportedOperationException {
                SparsePolynomial<Integer> other =
                new SparsePolynomial<>(Arrays.asList(
                new Monomial<>(integer(2), 1, 2, 3, 8),
                new Monomial<>(integer(8), 4, 6, 8, 8)
                ));
                ring.product(a, other); }
```

```
@Test
public void testMultiplyWithSparsePolynomial
WithOtherVariablesCount()
throws it.unipr.informatica
.UnsupportedOperationException {
        SparsePolynomial<Integer> expected =
        new SparsePolynomial<>(Arrays.asList(
        new Monomial<>(integer(10), 2, 4, 6),
        new Monomial<>(integer(8), 4, 5, 10),
        new Monomial<>(integer(40), 5, 8, 11),
        new Monomial<>(integer(32), 7, 9, 15)
        ));

        assertEquals(expected, ring.product(a, b));
}

@Test
public void testMultiplyWithZero()
throws it.unipr.informatica
.UnsupportedOperationException;

@Test
public void testMultiplyWithOne()
throws it.unipr.informatica
.UnsupportedOperationException;

@Test
public void testMultiplyingCommutativity()
throws UnsupportedOperationException;

@Test
public void testMultiplyingAssociativity()
throws UnsupportedOperationException {
        assertEquals(ring.product(ring
        .product(a, b), c), ring
        .product(a, ring.product(b, c)));
}

@Test
public void testDistributivity()
throws UnsupportedOperationException;
```

```
        @Test
        public void testPowWithPositiveExponent()
        throws UnsupportedOperationException {
                SparsePolynomial<Integer> expected =
                new SparsePolynomial<>(Arrays.asList(
                new Monomial<>(integer(32), 5, 10, 15),
                new Monomial<>(integer(640), 8, 14, 20),
                new Monomial<>(integer(5120), 11, 18, 25),
                new Monomial<>(integer(20480), 14, 22, 30),
                new Monomial<>(integer(40960), 17, 26, 35),
                new Monomial<>(integer(32768), 20, 30, 40)
                ));
                Polynomial<Integer> actual = ring.pow(a, 5);

                assertEquals(expected, actual);
        }

        @Test
        public void testPowWithZeroExponent()
        throws UnsupportedOperationException;

        @Test(expected = ArithmeticException.class)
        public void testPowWithNegativeExponent()
        throws UnsupportedOperationException {
                ring.pow(a, -5);
        }

        @Test
        public void testEvaluate()
        throws UnsupportedOperationException;


        @Test
        public void testToSparsePolynomialForSparsePolynomial()
        throws UnsupportedOperationException;

        @Test
        public void testToSparsePolynomialForDensePolynomial()
        throws UnsupportedOperationException;

        @Test(expected = IllegalArgumentException.class)
        public void testToSparsePolynomialForUnknownPolynomial()
        throws UnsupportedOperationException;
}
```

La classe *SparsePolynomialRingTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di polinomi in forma sparsa e tre elementi inizializzati appartenenti ai polinomi in forma sparsa.

I metodi proposti si occupano di verificare: le operazioni di somma, moltiplicazione ed elevamento a potenza per polinomi in forma sparsa, le proprietà commutative, associative e distributive riferite alle operazioni, l'elemento neutro per la somma e per la moltiplicazione, l'illegalità di un elevamento a potenza con esponente negativo, l'efficacia delle funzione di conversione da un polinomio in forma sparsa a densa e viceversa.

3.2.5 DensePolynomialRingTest

```
public class DensePolynomialRingTest {
        DensePolynomialRing<Integer> ring =
        new DensePolynomialRing<>(new IntegerRing());
        DensePolynomial<Integer> a =
        ring.create(Arrays.asList(
        new Monomial<>(integer(2), 1, 2, 3),
        new Monomial<>(integer(8), 4, 6, 8)
        ));

        DensePolynomial<Integer> b =
        ring.create(Arrays.asList(
        new Monomial<>(integer(5), 1, 2, 3),
        new Monomial<>(integer(4), 3, 3, 7)
        ));

        DensePolynomial<Integer> c =
        ring.create(Arrays.asList(
        new Monomial<>(integer(2), 4, 6, 8),
        new Monomial<>(integer(1), 3, 3, 7)
        ));

        @Test
        public void testAddOtherDensePolynomial()
        throws UnsupportedOperationException {
                DensePolynomial<Integer> expected =
                ring.create(Arrays.asList(
                new Monomial<>(integer(7), 1, 2, 3),
                new Monomial<>(integer(8), 4, 6, 8),
                new Monomial<>(integer(4), 3, 3, 7)
                ));

                assertEquals(expected, ring.sum(a, b));
        }
```

```
@Test
public void testAddingCommutativity()
throws UnsupportedOperationException;

@Test
public void testAddingAssociativity()
throws UnsupportedOperationException;

@Test
public void testAddingWithZero()
throws UnsupportedOperationException;

@Test(expected =
UnsupportedOperationException.class)
public void testAddingWithPolynomial
WithDifferentCountOfVariables()
throws UnsupportedOperationException {
        DensePolynomial<Integer> d =
        ring.create(Arrays.asList(
        new Monomial<>(integer(2), 4, 6, 8, 9),
        new Monomial<>(integer(1), 3, 3, 7, 1)
        ));

        ring.sum(a, d);
}

@Test(expected =
UnsupportedOperationException.class)
public void testMultiply
WithOtherDensePolynomial()
throws UnsupportedOperationException;

@Test
public void testMultiplyWithDensePolynomial
WithOtherVariablesCount()
throws UnsupportedOperationException {
        DensePolynomial<Integer> expected =
        ring.create(Arrays.asList(
        new Monomial<>(integer(10), 2, 4, 6),
        new Monomial<>(integer(8), 4, 5, 10),
        new Monomial<>(integer(40), 5, 8, 11),
        new Monomial<>(integer(32), 7, 9, 15)
        ));

        assertEquals(expected,
        ring.product(a, b));
}
```

```
@Test
public void testMultiplyWithZero()
throws UnsupportedOperationException;

@Test
public void testMultiplyWithOne()
throws UnsupportedOperationException;

@Test
public void testMultiplyingCommutativity()
throws UnsupportedOperationException;

@Test
public void testMultiplyingAssociativity()
throws UnsupportedOperationException;

@Test
public void testDistributivity()
throws UnsupportedOperationException;

@Test
public void testPowWithPositiveExponent()
throws UnsupportedOperationException {
        DensePolynomial<Integer> expected =
        ring.create(Arrays.asList(
        new Monomial<>(integer(32), 5, 10, 15),
        new Monomial<>(integer(640), 8, 14, 20),
        new Monomial<>(
        integer(5120), 11, 18, 25),
        new Monomial<>(
        integer(20480), 14, 22, 30),
        new Monomial<>(
        integer(40960), 17, 26, 35),
        new Monomial<>(
        integer(32768), 20, 30, 40)
        ));
        Polynomial<Integer> actual =
        ring.pow(a, 5);
        assertEquals(expected, actual);
}

@Test
public void testPowWithZeroExponent()
throws UnsupportedOperationException;

@Test(expected = ArithmeticException.class)
public void testPowWithNegativeExponent()
throws UnsupportedOperationException;
```

```
        @Test
        public void testEvaluate()
        throws UnsupportedOperationException;

        @Test
        public void testToDensePolynomial
        ForDensePolynomial()
        throws UnsupportedOperationException;

        @Test
        public void testToDensePolynomial
        ForSparsePolynomial()
        throws UnsupportedOperationException {
                List<Monomial<Integer>> monomials =
                Arrays.asList(
                new Monomial<>(integer(2), 1, 2, 3),
                new Monomial<>(integer(8), 4, 6, 8)
                );

                Polynomial<Integer> input =
                new SparsePolynomial<>(monomials);
                Polynomial<Integer> expected =
                ring.create(monomials);

                Polynomial<Integer> actual =
                ring.toDensePolynomial(input);
                assertEquals(expected, actual);
        }

        @Test(expected = IllegalArgumentException.class)
        public void testToDensePolynomial
        ForUnknownPolynomial()
        throws UnsupportedOperationException;
    }
```

La classe *DensePolynomialRingTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie verifiche: un anello di polinomi in forma densa e tre elementi inizializzati appartenenti ai polinomi in forma densa.

I metodi proposti si occupano di verificare: le operazioni di somma, moltiplicazione ed elevamento a potenza per polinomi in forma sparsa, le proprietà commutative, associative e distributive riferite alle operazioni, l'elemento neutro per la somma e per la moltiplicazione, l'illegalità di un elevamento a potenza con esponente negativo, l'efficacia delle funzione di conversione da un polinomio in forma densa a sparsa e viceversa.

3.2.6 BernsteinPolynomialRingTest

```
public class BernsteinPolynomialRingTest {
        BernsteinPolynomialRing<Rational> ring =
        new BernsteinPolynomialRing<>(new RationalRing());
        List<VariableDomain<Rational>> domains = Arrays.asList(
        new VariableDomain<>(rational(2), rational(5)),
        new VariableDomain<>(rational(3), rational(6))
        );
        List<Rational> aCoefs = Arrays.asList(
        rational(1), rational(2), rational(3),
        rational(4), rational(5), rational(6),
        rational(7), rational(8), rational(9)
        );
        BernsteinPolynomial<Rational> a =
        new BernsteinPolynomial<>(2, 2, aCoefs, domains);
        List<Rational> bCoefs = Arrays.asList(
        rational(4), rational(5), rational(6),
        rational(7), rational(8), rational(9),
        rational(1), rational(2), rational(3)
        );
        BernsteinPolynomial<Rational> b =
        new BernsteinPolynomial<>(2, 2, bCoefs, domains);
        List<Rational> cCoefs = Arrays.asList(
        rational(7), rational(8), rational(9),
        rational(4), rational(5), rational(6),
        rational(1), rational(2), rational(3)
        );
        BernsteinPolynomial<Rational> c =
        new BernsteinPolynomial<>(2, 2, cCoefs, domains);

        @Test
        public void testAddOtherBernsteinPolynomial()
        throws UnsupportedOperationException {
                List<Rational> coefs = Arrays.asList(
                rational(5), rational(7), rational(9),
                rational(11), rational(13), rational(15),
                rational(8), rational(10), rational(12)
                );
                BernsteinPolynomial<Rational> expected =
                new BernsteinPolynomial<>(2, 2, coefs, domains);
                assertEquals(expected, ring.sum(a, b));
        }

        @Test
        public void testAddOtherBernstein
        PolynomialWithLowerGrade()
        throws UnsupportedOperationException;
```

```
@Test
public void testAddingCommutativity()
throws UnsupportedOperationException {
        assertEquals(ring.sum(a, b), ring.sum(b, a));
}

@Test
public void testAddingAssociativity();

@Test
public void testAddingWithZero()
throws UnsupportedOperationException;

@Test(expected = UnsupportedOperationException.class)
public void testAddingWithPolynomial
WithDifferentCountOfVariables()
throws UnsupportedOperationException;

@Test(expected = UnsupportedOperationException.class)
public void testMultiplyWithBernsteinPolynomial
WithOtherVariablesCount()
throws UnsupportedOperationException;

@Test
public void testMultiplyWith
OtherBernsteinPolynomial()
throws UnsupportedOperationException;

@Test
public void testMultiplyWithZero()
throws UnsupportedOperationException {
        assertEquals(ring.zero(),
        ring.product(a, ring.zero()));
        assertEquals(ring.zero(),
        ring.product(ring.zero(), a));
}

@Test
public void testMultiplyWithOne()
throws UnsupportedOperationException;

@Test
public void testMultiplyingCommutativity()
throws UnsupportedOperationException;

@Test
public void testMultiplyingAssociativity()
throws UnsupportedOperationException;
```

```
        @Test
        public void testDistributivity()
        throws UnsupportedOperationException;

        @Test
        public void testPowWithPositiveExponent()
        throws UnsupportedOperationException;

        @Test
        public void testPowWithZeroExponent()
        throws UnsupportedOperationException {
                BernsteinPolynomial<Rational>
                expected = ring.one();
                BernsteinPolynomial<Rational>
                actual = ring.pow(a, 0);

                assertEquals(expected, actual);
        }

        @Test(expected = ArithmeticException.class)
        public void testPowWithNegativeExponent()
        throws UnsupportedOperationException;

        @Test
        public void testEvaluate()
        throws UnsupportedOperationException {
                Rational expected = rational(65, 3);
                Rational actual =
                ring.evaluate(a, rational(4), rational(6));

                assertEquals(expected, actual);
        }
}
```

La classe *BernsteinPolynomialRingTest* prevede quattro elementi di riferimento che verranno sfruttati per le varie modifiche: un anello di polinomi in forma di Bernstein e tre elementi inizializzati appartenenti ai polinomi in forma di Bernstein.

I metodi proposti si occupano di verificare: le operazioni di somma, moltiplicazione ed elevamento a potenza per polinomi in forma sparsa, le proprietà commutative, associative e distributive riferite alle operazioni, l'elemento neutro per la somma e per la moltiplicazione, l'illegalità di un elevamento a potenza con esponente negativo.

Capitolo 4

Esempi e Casi d'Uso

Di seguito vengono presentate alcune delle possibili applicazione rese disponibili dall'implementazione della libreria. In particolare sono illustrati esempi su come maneggiare:

- *monomi*;
- *polinomi in forma sparsa*;
- *polinomi in forma densa*;
- *conversione tra le varie forme di polinomi*;
- *polinomi in forma di Bernstein.*

```
public class Main {
        public static void println(Object obj) {
                System.out.println(obj);
        }

        public static Integer integer(long value) {
                return new Integer(BigInteger.valueOf(value));
        }

        public static Rational rational(long value) {
                return new Rational(BigInteger.valueOf(value),
                BigInteger.ONE);
        }
```

```
        public static void main(String[] args)
        throws IOException, UnsupportedOperationException {
                monomials();
                sparsePolynomials();
                densePolynomials();
                converting();
                bernsteinPolynomials();
        }

        public static void monomials()
        throws UnsupportedOperationException {
                println("");
                println("Monomials");
                println("———————————————————
                ———————————————————————————");
                MonomialExecutor<Integer> monomialExecutor =
                new MonomialExecutor<>(new IntegerRing());
                Monomial<Integer> a =
                new Monomial<>(integer(2), 1, 2, 3);
                Monomial<Integer> b =
                new Monomial<>(integer(3), 1, 2, 3);

                println("  a = " + a.toString());
                println("  b = " + b.toString());
                println("a+b = " + monomialExecutor
                .sum(a, b).toString());
                println("a*b = " + monomialExecutor
                .product(a, b).toString());
                Integer x1 = integer(5);
                Integer x2 = integer(6);
                Integer x3 = integer(8);
                println("if x1 = " + x1 + ",
                x2 = " + x2 + ", x3 = " + x3 + ",
                then a = " + monomialExecutor
                .evaluate(a, x1, x2, x3));
        }

        public static void sparsePolynomials()
        throws UnsupportedOperationException {
                println("");
                println("Sparse polynomials");
                println("———————————————————
                ———————————————————————————");
                SparsePolynomialRing<Integer> ring =
                new SparsePolynomialRing<>(new IntegerRing());
```

```
            SparsePolynomial<Integer> a =
            new SparsePolynomial<>(Arrays.asList(
            new Monomial<>(integer(2), 1, 2, 3),
            new Monomial<>(integer(8), 4, 6, 8)
            ));

            SparsePolynomial<Integer> b =
            new SparsePolynomial<>(Arrays.asList(
            new Monomial<>(integer(5), 1, 2, 3),
            new Monomial<>(integer(4), 3, 3, 7)
            ));

            println("  a = " + a.toString());
            println("  b = " + b.toString());
            println("a+b = " + ring.sum(a, b).toString());
            println("a*b = "
            + ring.product(a, b).toString());
            println("a^5 = " + ring.pow(a, 5).toString());
            Integer x1 = integer(5);
            Integer x2 = integer(6);
            Integer x3 = integer(8);
            println("if x1 = " + x1 + ",
            x2 = " + x2 + ", x3 = " + x3 + ",
            then a = " + ring.evaluate(a, x1, x2, x3));
    }

    public static void densePolynomials()
    throws UnsupportedOperationException {
            println("");
            println("Dense polynomials");
            println("————————————————————
            ——————————————————————————");
            DensePolynomialRing<Integer> ring =
            new DensePolynomialRing<>(new IntegerRing());
            DensePolynomial<Integer> a =
            ring.create(Arrays.asList(
            new Monomial<>(integer(2), 1, 2, 3),
            new Monomial<>(integer(8), 2, 2, 3)
            ));

            DensePolynomial<Integer> b =
            ring.create(Arrays.asList(
            new Monomial<>(integer(5), 1, 2, 3),
            new Monomial<>(integer(4), 3, 3, 1)
            ));
```

```
                println("  a = " + a.toString());
                println("  b = " + b.toString());
                println("a+b = " + ring.sum(a, b).toString());
                println("a*b = "
                + ring.product(a, b).toString());
                println("a^3 = " + ring.pow(a, 3).toString());
                Integer x1 = integer(5);
                Integer x2 = integer(6);
                Integer x3 = integer(8);
                println("if x1 = " + x1 + ",
                x2 = " + x2 + ", x3 = " + x3 + ",
                then a = " + ring
                .evaluate(a, x1, x2, x3));
        }

        public static void converting() {
                println("");
                println("Converting polynomials");
                println("————————————————————————
                ————————————————————————");
                SparsePolynomialRing<Integer> sparseRing =
                new SparsePolynomialRing<>(new IntegerRing());
                DensePolynomialRing<Integer> denseRing =
                new DensePolynomialRing<>(new IntegerRing());
                SparsePolynomial<Integer> a =
                new SparsePolynomial<>(Arrays.asList(
                new Monomial<>(integer(2), 1, 2, 3),
                new Monomial<>(integer(8), 4, 6, 8)
                ));

                DensePolynomial<Integer> b =
                denseRing.toDensePolynomial(a);
                SparsePolynomial<Integer> c =
                sparseRing.toSparsePolynomial(b);


                println("initial sparse polynomial: "
                + a.toString());
                println("convert to dense form: "
                + b.toString());
                println("convert back to sparse form: "
                + c.toString());
        }
```

```
        public static void bernsteinPolynomials()
        throws UnsupportedOperationException {
                println("");
                println("Bernstein polynomials");
                println("————————————————————————
                ————————————————————————");
                BernsteinPolynomialRing<Rational> ring =
                new BernsteinPolynomialRing<>(
                new RationalRing());
                List<VariableDomain<Rational>> domains =
                Arrays.asList(
                new VariableDomain<>(rational(2), rational(5)),
                new VariableDomain<>(rational(3), rational(6))
                );

                List<Rational> aCoefs = Arrays.asList(
                rational(1), rational(2), rational(3),
                rational(4), rational(5), rational(6),
                rational(7), rational(8), rational(9)
                );
                BernsteinPolynomial<Rational> a =
                new BernsteinPolynomial<>(2, 2, aCoefs, domains);

                List<Rational> bCoefs = Arrays.asList(
                rational(4), rational(5), rational(6),
                rational(7), rational(8), rational(9),
                rational(1), rational(2), rational(3)
                );
                BernsteinPolynomial<Rational> b =
                new BernsteinPolynomial<>(2, 2, bCoefs, domains);

                println("  a = " + a.toString());
                println("  b = " + b.toString());
                println("a+b = " + ring.sum(a, b).toString());
                println("a*b = "
                + ring.product(a, b).toString());
                println("a^5 = " + ring.pow(a, 5).toString());
                Rational x1 = rational(5);
                Rational x2 = rational(6);
                println("if x1 = " + x1 + ",
                x2 = " + x2 + ", then a = "
                + ring.evaluate(a, x1, x2));
        }
}
```

Conclusioni

L'obiettivo della tesi proposta era quello di fornire una library scritta in linguaggio Java per una manipolazione efficace dei polinomi in forma di Bernstein. Lo scopo è stato raggiunto sviluppando un adeguato supporto ai programmatori che in futuro potrebbero manifestare interesse alle potenzialità di questi strumenti.

Infatti esistono e sono ampiamente documentate, sia in letteratura che in rete, molte librerie scritte per poter interagire con strutture algebriche quali gruppi, anelli, campi ed oggetti come monomi e polinomi; ma non era ancora reperibile un unico mezzo che fornisse una visione d'insieme della questione, partendo dagli oggetti primitivi sino ad arrivare al caso specifico, ovvero la possibilità di interagire con i polinomi in forma densa e sparsa e, soprattutto, in forma di Bernstein.

La library sviluppata può essere oggetto, un domani, di eventuali modifiche, in quanto sono state identificate in fase di sviluppo e di verifica alcune potenziali migliorie che possono essere interessanti per un lavoro di tesi.

Le classi sono state suddivise, all'interno del package principale di riferimento *it.unipr.informatica*, in due sotto-sezioni dai nomi *structures* ed *executors*, rispettivamente per oggetti paragonabili ad elementi di un anello e per oggetti che possono costituire un anello degli elementi proposti. Intuitivamente si potrebbe evidenziare il carattere delle due parti effettuando una loro ridenominazione.

All'interno dei principali metodi delle classi e delle interfacce della library non è stato considerato il concetto di ordinamento: ad esempio, possiamo pensare alla dichiarazione di un dominio delle variabili, dove si è ritenuto implicitamente che i valori fossero ordinati per identificare un minimo e un massimo. Uno sviluppo futuro può prevedere di implementare questa elaborazione.

Infine evidenziamo un piccolo inconveniente che è stato identificato ma non corretto: nella richiesta, ad esempio, di verifica di un oggetto razionale *if (!(other instanceof Rational))* non si è considerato il caso in cui l'oggetto *other* fosse *NULL*. Sarebbe opportuno eseguire quest'ultima verifica.

Ringraziamenti

Il primo gesto di riconoscenza va alla mia famiglia, ai miei genitori Norma e Giorgio ed ai miei fratelli Diego e Marco, i quali mi hanno concesso l'opportunità di crescere in una casa accogliente e mi hanno sopportato per tutto questo tempo.

Ringrazio in particolar modo il mio professore e relatore Federico Bergenti, per la costante disponibilità e per l'inesauribile entusiasmo che mette al servizio dell'università e di tutte le attività di cui è un promotore convinto.

Molti altri professori, sia per la formazione universitaria che per quella liceale, sono stati una fonte preziosa di apprendimento e impegno: ringrazio quindi in prima battuta Alessandra Aimi, Roberto Alfieri, Alessandro Dal Palù, Giulio Destri, Grazia Lotti, Gianfranco Rossi ed Enea Zaffanella; successivamente esprimo gratitudine anche a Piera Bagozzi, Alessandro Benigni, Cecilia Fava, Giulia Ghidini, Alfonso Mongardi e Luciana Teroni.

Dico grazie anche e soprattutto alla persona più importante della mia vita, a Viola che è il sostegno più solido e l'alleato più fedele, sempre paziente ma allo stesso tempo decisa nei miei confronti: senza di te sicuramente tutto questo non si sarebbe avverato.

Ringrazio tutti coloro con cui ho condiviso questo importante percorso durato oltre tre anni, in quanto hanno reso più leggere le difficoltà e più divertente lo studio: Alessio, Daniela, Eleonora, Federico, Francesco, Jacopo, Luca, Maxim, Sebastian, Victoria e Francesco.

Anche se indicati per ultimi non sono sicuramente stati meno rilevanti tutti quelli con cui ho condiviso le gioie e gli insuccessi della vita: Federico, Filippo, Francesco, Gerri, Graziano, Federico, Ivan, Lorenzo, Riccardo, Silvia, Dana, Alessandra, Mafo, Stefano, Francesca, Giovanni, Davide, Andrea, Nicola, Angelo, Michela, Giuseppe, Angela, Umberto, Bruno, Davide, Filippo, Davide, Simone, Tommaso, Jacopo, Giuseppe, Mauro, Felice, Daniela, Federica, Laila, Miguel, Arianna, Benedetta, Davide, Denise, Elisabetta, Federica, Lorenza, Lorenzo, Marco, Marika, Michele, Marco, Valentina, Valerio, Benedetta, Bianca, Marco, Sara, Anna, Omar, Andrea e Francesco; perché siete un'ancora solida e fonte sempre accesa della mia felicità.

Bibliografia

[1] Pavarani Alice, “Soddisfacimento di Vincoli Globali mediante Polinomi di Bernstein”, Università degli Studi di Parma, 2012

[2] Roche Daniel Steven, “Efficient Computation with Sparse and Dense Polynomials”, University of Waterloo, 2011

[3] Zini Francesco, “Progettazione e Realizzazione di uno Strumento per la Soluzione di Sistemi di Disequazioni Diofantine mediante la Forma di Bernstein”, Università degli Studi di Parma, 2013