UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria Gestionale, dell'Informazione e
Della Produzione

Corso di laurea Magistrale in
Ingegneria Informatica

Corso di Programmazione Avanzata

Documentazione Progetti

Candidato:
Lorenzo Ferrari
Matricola n.1053161

Docente:
Prof. Angelo Gargantini

Anno Accademico
2022/2023

Indice

Introduzione

L’obbiettivo dei tre progetti consiste nella realizzazione di tre programmi che mostrino ed utilizzino i diversi costrutti studiati e mostrati a lezione: il primo sviluppato basandosi sul linguaggio C++, sfruttando il paradigma della programmazione ad oggetti, il secondo utilizzando il linguaggio Java, basato anche esso sul paradigma della programmazione ad oggetti ed infine il terzo in Haskell, che segue il paradigma della programmazione funzionale. Di seguito vengono mostrati e descritti i programmi e le loro funzionalità.

Capitolo 1

C++

Il primo progetto, sviluppato in C++, ha come obbiettivo la gestione di un negozio di biciclette di diverse tipologie (da Montagna, da Corsa ed Elettriche).
Il programma permette all'utente l'inserimento delle singole biciclette, definite attraverso una serie di campi, e la loro conseguente memorizzazione.
Questo progetto ha lo scopo di sfruttare al più tutti i costrutti del linguaggio C++ che sono stati mostrati ed illustrati a lezione. Di seguito viene innanzitutto presentato il diagramma delle classi relativo al progetto mentre successivamente vengono illustrate le funzionalità del linguaggio C++ implementate e sfruttate all'interno del progetto

1.1 Diagramma delle classi

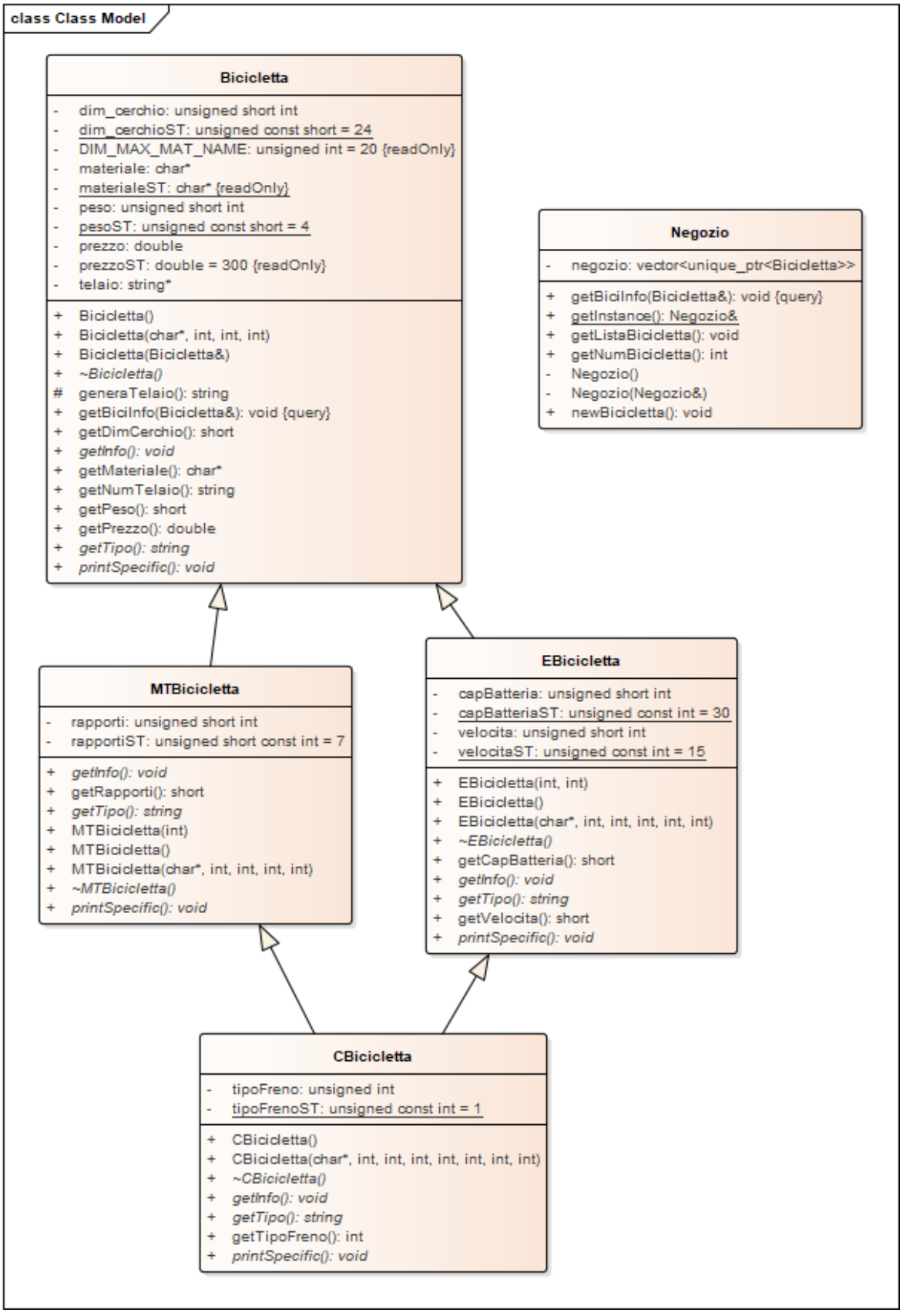

Figura 1: Diagramma delle classi C++

1.2 Analisi delle funzioni

- **Definizione dei file .h e .cpp**

  Le implementazioni delle classi sono state realizzate suddividendo il file header (.h), in cui vengono rese disponibili le funzionalità relativa alla classe e il file in cui è definita la loro implementazione (.cpp).

- **Overloading dei costruttori, valori di default**

  Si definiscono più costruttori per la stessa classe allo scopo di garantire la creazione di istanze dello stesso oggetto in modi diversi, ad esempio basandosi su parametri forniti dall'utente oppure parametri di default

```
//Bicicletta inizializzata a valori generici (esempio definiti dal venditore)
Bicicletta::Bicicletta() {
    materiale = strcpy((char*)malloc(sizeof(char)*strlen(Bicicletta::materialeST)),Bicicletta::materialeST);
    peso = Bicicletta::pesoST;
    dim_cerchio = Bicicletta::dim_cerchioST;
    prezzo = Bicicletta::prezzoST;
    telaio = new string();
    *telaio = generaTelaio();
}
//Bicicletta inizializzata basandosi sui parametri forniti in ingresso
Bicicletta::Bicicletta(char *materialeIN, int pesoIN, int dim_cerchioIN, int prezzoIN):
        peso(pesoIN), dim_cerchio(dim_cerchioIN), prezzo(prezzoIN) {

    materiale = (char*)malloc(sizeof(char)*Bicicletta::DIM_MAX_MAT_NAME);
    strcpy(materiale,materialeIN);
    telaio = new string();
    *telaio = Bicicletta::generaTelaio();

}
```

- **Copy constructor**

  Definizione di copy constructor per la classe Bicicletta per garantire la realizzazione di un'istanza di quest'ultima mediante una copia di un'istanza già creata.

```
Bicicletta::Bicicletta(const Bicicletta &NuovaBicicletta) {
    strcpy(materiale, NuovaBicicletta.materiale);
    Bicicletta::dim_cerchio = NuovaBicicletta.dim_cerchio;
    Bicicletta::peso = NuovaBicicletta.peso;
    Bicicletta::prezzo = NuovaBicicletta.prezzo;
    Bicicletta::telaio = NuovaBicicletta.telaio;
}
```

- **Definizioni di distruttori virtual**

  I distruttori vengono definiti per mezzo della clausola *virtual* allo scopo di garantire la chiamata del metodo distruttore della superclasse rispetto alla classe considerata.

  Da notare come il materiale sia allocato in modo esplicito, mediante *malloc()* e quindi necessita di una deallocazione tramite la clausola *free()* (C Style) mentre il numero è allocato mediante *new* e pertanto necessita di essere deallocato tramite la clausola *delete* (C++ Style).

```
Bicicletta::~Bicicletta() {
 free(materiale);
 delete telaio;
 cout << "Bicicletta rimossa" << endl;
}
```

```
// Distruttore della classe
virtual ~Bicicletta();
```

- **Definizione di funzioni e relativi passaggi dei parametri**

  Per ciascuna classe sono state definite diverse classi con diversi metodi di passaggio dei parametri. Da notare come l'utilizzo del termine *const* non consenta che il metodo modifichi i membri della classe.

```
void getBiciInfo(Bicicletta &b) const;
void getBiciInfo(Bicicletta &b)const {
    b.getInfo();
}
```

- **Funzioni inline**

  Sono funzioni la cui chiamata è sostituita dal corpo della funzione.

```
inline int getNumBicicletta(){
    return negozio.size();
}
```

- **Eccezioni**

  Si è voluto gestire le eccezioni implementandole in due diverse modalità: uso della clausola *throw* per lanciare un codice d'errore oppure mediante il costrutto *try-catch* in cui si intercetta il codice di errore e si procede alla visualizzazione di un messaggio.

- **Classe astratta**

  La classe Bicicletta è una classe astratta in quanto vi è la presenza di almeno un metodo pure *virtual* e questo fa capire come tale classe sia solo definita ma non venga mai istanziata. I metodi pure virtual verranno implementati esclusivamente nelle sole classi derivate.

```
virtual void getInfo() = 0; //Informazioni sulla bicicletta
virtual void printSpecific() = 0; // Informazioni specifiche della classe derivata
virtual string getTipo() = 0; //Tipologia della bicicletta
```

- **Incapsulamento e livelli di visibilità**

  Utilizzo dei diversi modificatori di visibilità: *public*, *protected*, *private*.

- **Ereditarietà, definizioni di classi base e derivate**

  Definizione di una classe base (**Bicicletta**) e tre classi derivate (**MTBicicletta, EBicicletta e CBicicletta**) con conseguente riutilizzo del codice.

- **Polimorfismo run-time**

  È stata utilizzata la clausola *virtual* per garantire l'implementazione del polimorfismo run-time dei vari metodi nelle sottoclassi.

- **STL Containers**

  Utilizzo di contenitori non associativi come *vector*<> per garantire la memorizzazione delle biciclette presenti nel Negozio inserite dall'utente, associando le relative funzioni per inserimento e rimozioni di elementi nel contenitore.

```
vector<unique_ptr<Bicicletta>> negozio;
```

- **Utilizzo degli smart pointers**

  Oltre all'utilizzo dei classici puntatori si è voluto presentare anche l'utilizzo degli smart pointers, i quali garantiscono una maggiore sicurezza nella gestione automatica di allocazione e conseguente deallocazione della memoria. Definendo poi un vettore (*vector*) di smart pointers unici (*unique_ptr*) è stata necessaria l'introduzione e l'utilizzo della primitiva *move()* in quanto essendo il puntatore unico, esso deve essere spostato manualmente per non perdere il riferimento.

```
unique_ptr<Bicicletta> MTB_pt(new MTBicicletta(materiale_pt,peso,dim_cerchio,prezzo,rapporti));
negozio.push_back(move(MTB_pt));
```

- **Variabili statiche e costanti**

  Utilizzo della primitiva *static* per indicare variabili che vengono condivise tra tutte le istanze della classe e della primitiva *const* per indicare variabili che non possono essere modificate. Nel progetto esse sono pari alle variabili che assegnano valori di default durante la creazione di istanze di oggetti di tipo MTBicicletta, EBicicletta e CBicicletta.

```
//Variabili statiche
static unsigned const short dim_cerchioST = 24;
static unsigned const short pesoST = 4 ;

//Variabili costanti
const double prezzoST = 300;
static const char* materialeST;
const unsigned int DIM_MAX_MAT_NAME = 20;
```

- **Utilizzo del tipo Bool**

  C++, rispetto a C, introduce la possibilità di utilizzare il tipo booleano (*bool*). In particolare, nel progetto l'utilizzo di tale tipo è riservato a verificare che l'utente voglia o meno inserire una nuova bicicletta.

```
bool continua = true;
```

- **Design Pattern**

  Per la classe Negozio, è stato implementato il design pattern *Singleton* in modo da garantire un'istanziazione univoca della classe, impendendo quindi all'utente la possibilità di crearne nuove copie.

```
class Negozio {
public:
    static Negozio& getIstance(){
        static Negozio istance;
        return istance;
    }
private:
    Negozio();

    Negozio(Negozio const&);
}
```

- **Ereditarietà multipla: Member duplication o problema del diamante e name clashing**

  Viene utilizzata la clausola *virtual* per risolvere il problema del **Member duplication** nell'ambito della gestione dell'ereditarietà multipla. Si faccia riferimento al diagramma delle classi per osservare quella che è la tipica struttura a "diamante" che si realizza in presenza di ereditarietà multipla. Viene inoltre risolto il problema del *name clashing* che si verifica quando una classe eredita da più classi base.

Capitolo 2

Java

Il secondo progetto, sviluppato in Java, vuole anche esso implementare la gestione di un negozio di biciclette di diverse tipologie (da Montagna, da Corsa ed Elettriche). Il programma permette all'utente l'inserimento delle singole biciclette, definite attraverso una serie di campi, e la loro conseguente memorizzazione.
Questo progetto ha lo scopo di cercare sfruttare al più tutti i costrutti del linguaggio Java che sono stati mostrati ed illustrati a lezione mettendo in risalto le differenze di implementazione rispetto alla soluzione proposta in C++. Di seguito viene innanzitutto presentato il diagramma delle classi relativo al progetto mentre successivamente vengono illustrate le funzionalità del linguaggio C++ implementate e sfruttate all'interno del progetto

2.1 Diagramma delle classi

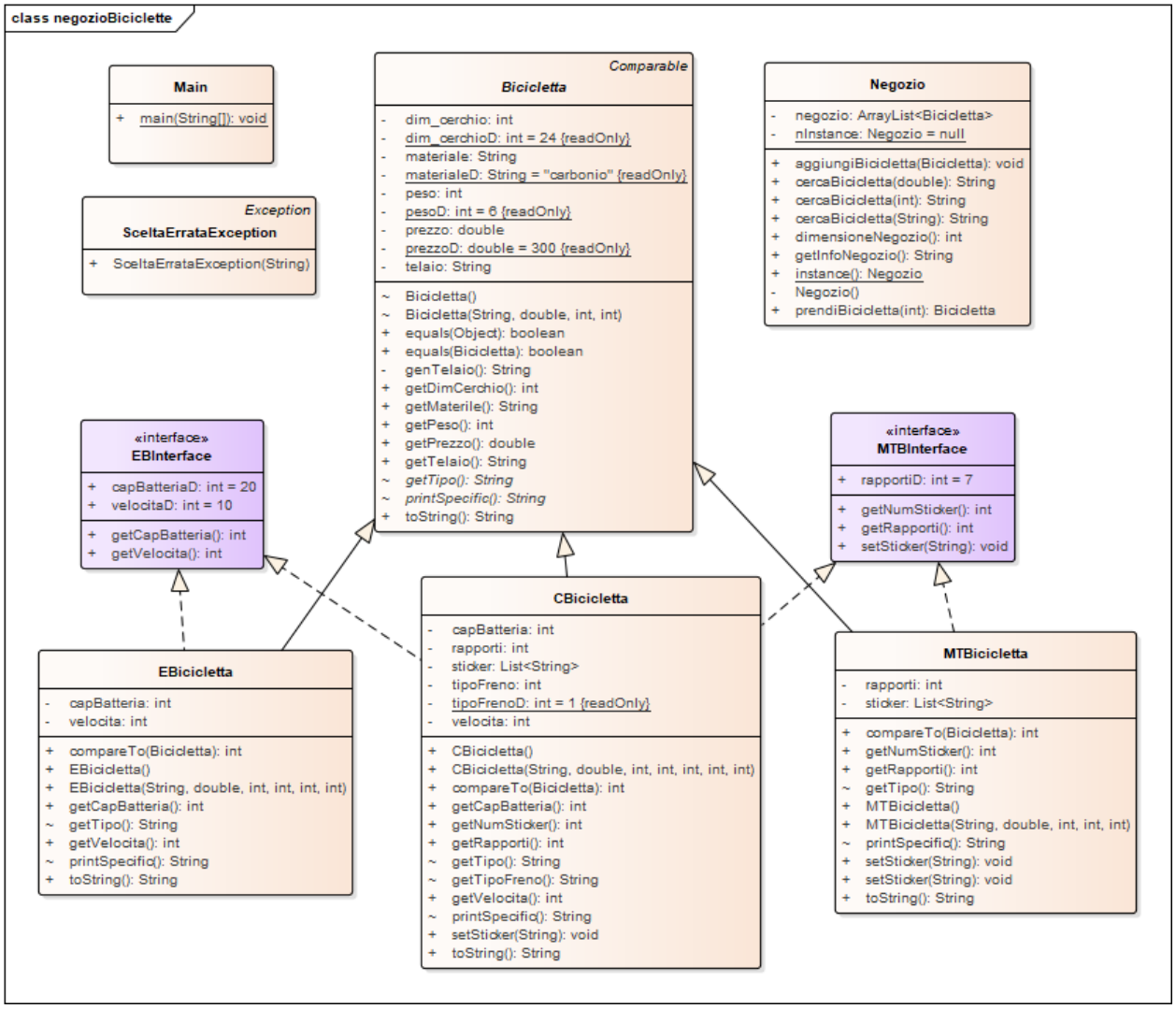

Figura 2: Diagramma delle classi JAVA

2.2 Analisi delle funzioni

- **Definizione dei file .java**

  La definizione e l'implementazione delle classi è avvenuta in un file (.java) per ciascuna classe.

- **Overloading dei costruttori, valori di default**

  Si definiscono più costruttori per la stessa classe allo scopo di garantire la creazione di istanze dello stesso oggetto in modi diversi, ad esempio basandosi su parametri forniti dall'utente oppure parametri di default.

```
//costruttore vuoto di default
Bicicletta(){
    this.dim_cerchio = dim_cerchioD;
    this.peso = pesoD;
    this.prezzo = prezzoD;
    this.materiale = materialeD;
    this.telaio = genTelaio();
}

//overloading del costruttore
Bicicletta(String materiale, double prezzo, int peso, int dim_cerchio){
    this.dim_cerchio = dim_cerchio;
    this.peso = peso;
    this.prezzo = prezzo;
    this.materiale = materiale;
    this.telaio = genTelaio();
}
```

- **Classe astratta**

  La classe Bicicletta è una classe astratta in quanto vi è la presenza di almeno un metodo che non risulta implementato e questo fa capire come tale classe sia solo definita ma non venga mai istanziata. Nel caso del linguaggio Java la clausola *abstract* determina la definizione di classi e metodi astratti.

```
public abstract class Bicicletta implements Comparable<Bicicletta> {
        //Metodi astratti implementati solo nelle classi derivate
        abstract String getTipo();
        abstract String printSpecific();
}
```

- **Ereditarietà, definizioni di classi base e derivate**

  Definizione di una classe base (**Bicicletta**) e tre classi derivate (**MTBicicletta, EBicicletta e CBicicletta**) con conseguente riutilizzo del codice. Questo nel codice è possibile realizzarlo mediante l'utilizzo della clausola *extends* che indica come la classe derivata estenda metodi e campi provenienti dalla classe base.

  Struttura di ereditarietà: Classe derivata *extends* Classe base.

```
public class MTBicicletta extends Bicicletta
```

- **Costruttori di classi derivate**

  Nella definizione del costruttore della sottoclasse, deve essere inserita la chiamata al costruttore della superclasse, mediante l'utilizzo della parola chiave *super*. Nel caso la prima istruzione di un costruttore della sottoclasse non è una chiamata a *super*, allora la chiamata a *super* viene inserita automaticamente dal compilatore.

```
public MTBicicletta(String materiale, int prezzo, int peso, int dim_cerchio, int rapporti) {
    super(materiale,prezzo,peso,dim_cerchio);
    this.rapporti = rapporti;
}
```

- **Incapsulamento e livelli di visibilità**
  Utilizzo dei diversi modificatori di visibilità: *public*, *private*.

- **Metodi final**

  L'utilizzo di metodi final garantisce che il metodo non possa essere sostituito in nessuna delle sottoclassi.

```
public final String getTelaio() { return this.telaio; }
```

- **Overloading e overriding dei metodi**

  Nel progetto sono presenti esempi di applicazione del polimorfismo dei metodi. In particolare, è presente:

  l'**overload**, ovvero la presenza di metodi nella stessa classe, che risultano avere lo stesso nome ma presentano una segnatura differente.

```
// -- Ricerca per prezzo
public String cercaBicicletta(double _prezzo) {
    String s = "";
    for(Bicicletta b: negozio) {
        if(b.getPrezzo() == _prezzo)
            s = s + b.toString();
    }
    return s;
}

// -- Ricerca per peso
public String cercaBicicletta(int _peso) {
    String s = "";
    for(Bicicletta b: negozio) {
        if(b.getPeso() == _peso)
            s = s + b.toString();
    }
    return s;
}
```

  l'**override** in cui avviene invece la riscrittura del metodo, lasciando invariata la sua segnatura, che è stato ereditata dalla superclasse.

```
@Override
String getTipo() {
    return "EB";
}
```

- **Interfaccia Comparable**

  Interfaccia (*interface*) che presenta una struttura simile ad una classe ma può contenere solo metodi astratti; di conseguenza sarà la classe stessa ad implementare (uso della parola chiave *implements*) quest'ultima, ovvero realizzerà i suoi metodi astratti rispettando le specifiche implementative. Un esempio di interfaccia è Comparable<T>, la quale per mezzo del suo metodo astratto *compareTo()* permette di definire un ordinamento totale sugli oggetti della classe e quindi dando la possibilità di effettuare confronti tra le diverse istanze della classe stessa. Nel progetto, l'implementazione del metodo *compareTo*() si è basato sull'effettuare un confronto tra i prezzi delle istanze della classe di tipo base Bicicletta, che invoca il metodo, e l'istanza dell'oggetto dello stesso tipo passato come parametro.

- **Variabili statiche e final**

  Utilizzo della primitiva *static* per indicare variabili che vengono condivise tra tutte le istanze della classe ed utilizzo della primitiva *final* per indicare che la variabile risulta essere immutabile e quindi non modificabile. Nel progetto esse sono pari alle variabili che assegnano valori di default durante la creazione di istanze di oggetti di tipo MTBicicletta, EBicicletta e CBicicletta.

```
private static final double prezzoD = 300;
private static final int dim_cerchioD = 24;
private static final int pesoD = 6;
private static final String materialeD = "carbonio";
```

- **Varargs**

  I Variabile Arguements permettono ad un metodo di accettare o zero o più argomenti. Nel progetto questa funzionalità è stata implementata mediante una fuzione denominata *sticker()* che accetta zero o più sticker che sono incollati alla bicicletta.

```
private List<String> sticker = new ArrayList<String>();
public void setSticker(String ... stickers) {
    for(String s: stickers)
            sticker.add(s);
}
```

- **Iteratori**

  L'utilizzo di un iteratore permette di atttraversare o scorrere una raccolta garantendone quindi l'accesso ai suoi elementi con la possibilità di aggiungerne di nuovi, rimuoverli dalla raccolta oppure elaborarli. In Java per poter sfruttare gli iteratori bisogna importarli da *java.util.Iterator*; la classe presenta i seguenti metodi:

  - *Next()* che ritorna il prossimo dato della raccolta.
  - *hasNext()* che ritorna true se il prossimo next() avrà successo; è usato quindi per determinare quando la scansione della raccolta terminerà.

```
public String getInfoNegozio() {
    String s = "";
    Iterator<Bicicletta> n = negozio.iterator();
    while(n.hasNext()) {
        s += n.next().toString();
    }
    return s;
}
```

- **Design Pattern**

  Per la classe Negozio, è stato implementato il design pattern *Singleton* in modo da garantire un'istanziazione univoca della classe, impendendo quindi all'utente la possibilità di crearne.

```
public class Negozio{

    //Implementazione del pattern Singleton per classe Negozio
    private static Negozio nInstance = null;

    //Attributo e costruttore della classe privati
    private ArrayList<Bicicletta> negozio;
    private Negozio(){ negozio = new ArrayList<Bicicletta>(); }

    //Gestione elaborazione pattern Singleton
    public static Negozio instance () {
        if ( nInstance == null )
            nInstance = new Negozio();
        return nInstance ;
    }
}
```

- **Gestione delle collezioni**

  Si utilizza il metodo *Collections.sort(negozio)*, messo a disposizione della classe *Collections*, per realizzare in modo effettivo l'ordinamento, inizialmente definito dall'implementazione dell'interfaccia Comparable<T>, delle istanze delle classi MTBicicletta, EBicicletta e CBicicletta memorizzate nella struttura dati presente nella classe Negozio.

- **Utilizzo delle collezioni**

  Per garantire la memorizzazione di più istanze delle diverse classi derivate dalla classe base Bicicletta, si è reso necessario sfruttare le strutture dati quali *ArrayLists<T>* messe a disposizione dall'ambiente java.

```
negozio = new ArrayList<Bicicletta>();
```

- **Implementazione e overriding del metodo *equals***

  Overriding del metodo *equals* della classe *Object* per permettere di realizzare in modo efficacie un confronto tra i campi delle istanze delle classi derivate dalla classe base *Bicicletta*. Questo, quindi, garantisce che il metodo sia in grado di trattare un qualunque parametro *Object* fornito in ingresso al metodo.

```java
//Implementazione metodo equals
public boolean equals (Object o) {
    if(o instanceof Bicicletta)
            return equals ((Bicicletta)o);
    return false ;
}
public boolean equals(Bicicletta b) {
    return this.getTipo().equals(b.getTipo());
}
```

- **Implementazione 'ereditarietà multipla' (problema del Diamante)**

  Vista l'impossibilità del linguaggio Java di poter realizzare l'ereditarietà multipla, (Ogni classe può essere estesa da più classi ma può estendere una e una sola classe), rispetto a C++, si è voluto quindi simularne la sua implementazione, in quanto la classe *CBicicletta*, in C++ ereditava da *EBicicletta* e *MTBicicletta*.

  Questa simulazione è stata resa possibile facendo implementare alla classe *CBicicletta*, così come alle altre classi derivate da Bicicletta, due interfacce chiamate *EBInterface* e *MTBInterface* che contengono i metodi propri e specifici rispettivamente delle classi *MTBicicletta* e *EBicicletta*. Lo svantaggio derivante dalla volontà di realizzare questa simulazione di ereditarietà multipla è dato dal fatto che la classe *CBicicletta* presenta attributi duplicati delle classi *MTBicicletta* e *EBicicletta*, rispetto alla sua equivalente in C++.

- **Eccezioni**

  Utilizzo di eccezione custom mediante la realizzazione della classe *SceltaErrataException*, per gestire il caso in cui l'utente inserisca un'opzione non valida nel menu a scelta utilizzato per l'elaborazione.

Capitolo 3

HASKELL

Il terzo progetto, sviluppato in Haskell, ha come obbiettivo quello di implementare una piccola libreria per rappresentare e manipolare alberi binari di ricerca.
Ogni albero binario può essere rappresentato con una delle seguenti forme:

- Una foglia, che non contiene elementi e non ha figli
- Una diramazione, che contiene un elemento e presenta due figli, ovvero due sotto-alberi.

Si ricorda inoltre di come un albero di ricerca è identificato dal fatto che, fissato un qualsiasi suo sotto-albero **T** avente per radice un elemento **x**, tutti gli elementi nel sotto-albero sinistro di **T** sono più piccoli di **x** e tutti gli elementi nel sotto-albero destro di **T** sono più grandi di **x**.

Di seguito vengono presentate le funzioni, le quali sfruttano un approccio ricorsivo per essere eseguite, che compongono la libreria ed alcune delle funzionalità del linguaggio utilizzate per svilupparle.

3.1 Analisi delle funzioni

Tipo di dato

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Eq,Show)
```

L'albero può essere Empty oppure un Node. Un nodo è composto da un elemento, che presenta un proprio valore e due lati, uno destro e uno sinistro, anche essi alberi.
Rappresentiamo quindi l'albero con un tipo polimorfo, ricorsivo e parametrico rispetto al tipo degli elementi contenuti nell'albero stesso.
La parola chiave deriving permette di implementare automaticamente funzioni per alcune classi di Haskell come Show ed Eq. In particolare, Eq permette che alberi dello stesso tipo possano essere confrontati tra loro mentre Show definisce le funzioni su come rappresentare i tipi di dati come una stringa.

Inserimento e rimozione di elementi nell'albero

insert :: Ord a => a -> Tree a -> Tree a	Inserisce un nuovo elemento all'interno dell'albero di ricerca
delete :: Ord a => a -> Tree a -> Tree a	Elimina un elemento, fornito come input alla funzione, presente nell'albero di ricerca
treeFromList :: Ord a => [a] -> Tree a	Genera un albero di ricerca a partire da una lista di elementi fornita in input

Informazioni di base sull'albero

depth :: Tree a -> Int	calcola la profondità di un albero, ovvero la lunghezza del percorso più lungo dalla radice a una delle sue foglie.
size :: Tree a -> Int	Restituisce il numero di nodi presenti all'interno dell'albero
isEmpty :: Tree a -> Bool	Verifica se un albero è vuoto
leafNumb :: Tree a -> Int	Restituisce il numero di foglie presenti nell'albero

Visita e ricerca nell'albero

anticipatedVisit :: Tree a -> [a]	Restituisce una lista di tutti gli elementi dell'albero visitando prima la radice, il sotto-albero sinistro ed infine quello destro
symmetricalVisit :: Tree a -> [a]	Restituisce una lista di tutti gli elementi dell'albero visitando in ordine il sotto-albero sinistro, la radice ed infine il sotto-albero destro
searchElem :: Ord a => a -> Tree a -> Bool	Ricerca se un elemento fornito in ingresso è presente all'interno dell'albero
maxElem :: Tree a -> Maybe a	Individua l'elemento più grande presente nell'albero
minElem :: Tree a -> Maybe a	Individua l'elemento più piccolo presente nell'albero

Altre operazioni

isBinaryTree :: Ord a => Tree a -> Bool	Verifica che l'albero fornito in input sia binario
printTree :: Show a => Tree a -> IO ()	Stampa l'albero di ricerca mettendo in risalto le sue diramazioni

3.2 Illustrazione funzionalità linguaggio

Si presentano ora un elenco di funzionalità e modi di programmare del linguaggio Haskell, utilizzate all'interno del progetto:

- **Definizione del tipo delle funzioni**: per alcune delle funzioni definite è utile definirne anche il tipo, come ad esempio la funzione `treeFromList :: Ord a => [a] -> Tree a`, che prende in ingresso una lista di elementi di tipo a, dove a è deve essere di tipo ordinabile, e restituisce un albero sempre di tipo a.

- **Uso della ricorsione**: tutte le funzioni implementate nella libreria basano il loro funzionamento sull'utilizzo della ricorsione, la quale è una delle caratteristiche fondamentali del linguaggio Haskell.

- **Pattern matching**: in base al tipo di input fornito in ingresso è possibile determinare i possibili comportamenti che permettono di definire la funzione. Quindi, quando la funzione verrà invocata, verrà utilizzata la prima definizione che determinerà un match tra quelle definite e l'input della funzione stessa.

- **Wildcard**: nel pattern matching può succedere che un caso di input è definito solo da uno dei parametri mentre gli altri possono assumere qualsiasi valore. Per indicare ciò si utilizza la wildcard "_".

- **Curried functions**: per gestire funzioni che presentano più parametri di ingresso senza ricorrere all'uso di tuple, Haskell permette di definire il tipo di una funzione in modo che questa restituisca a sua volta una funzione.

- **Guardie (|)**: è possibile indicare comportamenti diversi in funzione dell'input o di una particolare condizione di pattern matching mediante l'uso delle guardie (|) che equivale all'uso del costrutto `if … then … else`.

- **Costruttore di liste**: in Haskell le liste vengono definite per mezzo del costruttore ":", che aggiunge un elemento in testa alla lista. Ad esempio si può costruire la lista [a,b,c] come a:[b,c]. Inoltre, il costruttore può essere sfruttato nel pattern matching per definire che una lista è suddivisa tra testa e coda. Ad esempio definendo (x:xs) otteniamo già una suddivisione della lista in testa (x) e coda (xs).

- **`where`**: all'interno di una funzione è possibile definire valori intermedi attraverso l'utilizzo della parola chiave `where`.

- **Uso di `MapM_`**: consente di applicare un'azione (ovvero una funzione) su una lista, non raccogliendone i risultati. Tipicamente viene usata con funzioni come `print` oppure `putStrLn` per permettere di stampare una lista di elementi senza doverli memorizzare. Quindi in questo caso si è interessati agli effetti collaterali prodotti dalla funzione (ovvero la stampa) e non agli elementi modificati (la lista di elementi).

- **Uso di `map`**: funzione che restituisce una lista costruita applicando una funzione, fornita come primo argomento, a tutti gli elementi della lista passata come secondo argomento.

- **Uso di `filter`**: funzione che ritorna una lista, costruita utilizzando gli elementi della lista passata in input (secondo argomento) che soddisfano una particolare condizione (primo argomento).

- **Input/Output e main**: l'interazione con l'utente è determinata dalla definizione di una funzione main, mentre la gestione dell'interazione con l'utente è affidata a funzioni come putStrLn, print, read.