Relazione su Applicativo Java per la Gestione del Diabete

Federico Marra - Alberto del Buono Paolini

Ottobre - Dicembre 2023

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Dipartimento di Ingegneria dell'Informazione

Ingegneria del Software

Repository Github

Indice

1 Motivazione e Descrizione

Vogliamo modellare un sistema che controlli una *patch-pump* (detto anche microinfusore), quindi per la gestione insulinica, questo svolge due funzioni principali:

1. **Bolo**: È un bolo calcolato da una glicemia e una massa di carboidrati data in input dall'utente e divisi per fattori che dipendono dall'orario corrente, ci sono diverse opzioni tenendo per esempio conto dei vecchi boli e del decadimento lineare dell'insulina ancora attiva.

2. **Basale**: Fornisce una base continua di insulina erogata con una certa frequenza.

Modellizzando, queste funzioni vengono controllate tramite un palmare (Handheld Tracker) connesso via bluetooth alla patch installata sull'utente (Glucose Delivery System). Dal palmare è possibile creare un nuovo bolo in quattro modalità o modificare uno dei tre profili orari che viene salvata in un database locale, di cui può essere eseguito un backup su un database cloud (gestito da Cloud Interface). Quando si crea un bolo viene anche inviato al Glucose Delivery System che provvederà a controllare l'autenticità della richiesta ed erogarlo. L'utente può anche modificare il valore dell'insulina basale per certe ore, questo viene salvato nel profilo basale (Basal Profile) e conservato nel database locale. Il sistema a patch che consideriamo comprende anche una modalità di utilizzo standalone (senza palmare), infatti il Glucose Delivery System ha la possibilità di erogare automaticamente un bolo in base ai valori di glicemia letti dal suo stesso sensore, senza comunicare col palmare.

1.1 Possibili aggiunte

Nonostante siano state implementate solo in parte, riportiamo di seguito dei possibili sviluppi ulteriori per il nostro applicativo:

1. **Dashboard Web**
2. **Routine Insulina Basale**
3. **Routine Bolo Automatico**

1.1.1 Dashboard Web

Un'interfaccia grafica accessibile dal web (analoga alla GUI già implementata per i vari client) che include grafici e tabelle aggiuntivi. L'implementazione di questa funzionalità richiede un secondo server per ospitare la dashboard che dovrebbe includere un layer di autenticazione (permettendo ad un utente di controllare solo la sua pompa di insulina).

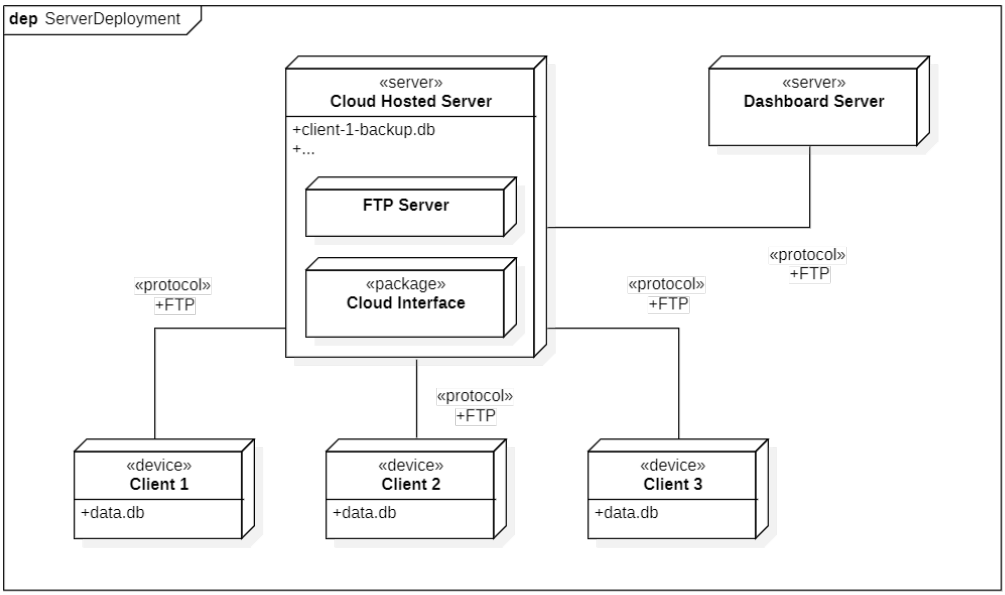

Figura 1: Deployment Diagram

1.1.2 Routine Insulina Basale

Una routine che attraverso un thread ogni 10/15 minuti inietti il quantitativo di insulina basale in percentuale alla frequenza (ad esempio: 10 min $\rightarrow \frac{1}{6}$), 15 min $\rightarrow \frac{1}{4}$) che è contenuto nel profilo orario.

1.1.3 Routine Bolo Automatico

Una routine che attraverso un thread ogni 10/15 minuti, prenda la glicemia dal sensore, controlli che non sia sopra una certa soglia (di solito 180mg/dL) e che se lo è esegue un bolo correttivo sottraendoci però l'insulina che è attiva.

2 Requisiti

2.1 Use Case

Dalla descrizione del nostro modello di dominio abbiamo identificato gli use cases che può incontrare l'utente. Di seguito viene riportato l'Use Case Diagram risultante:

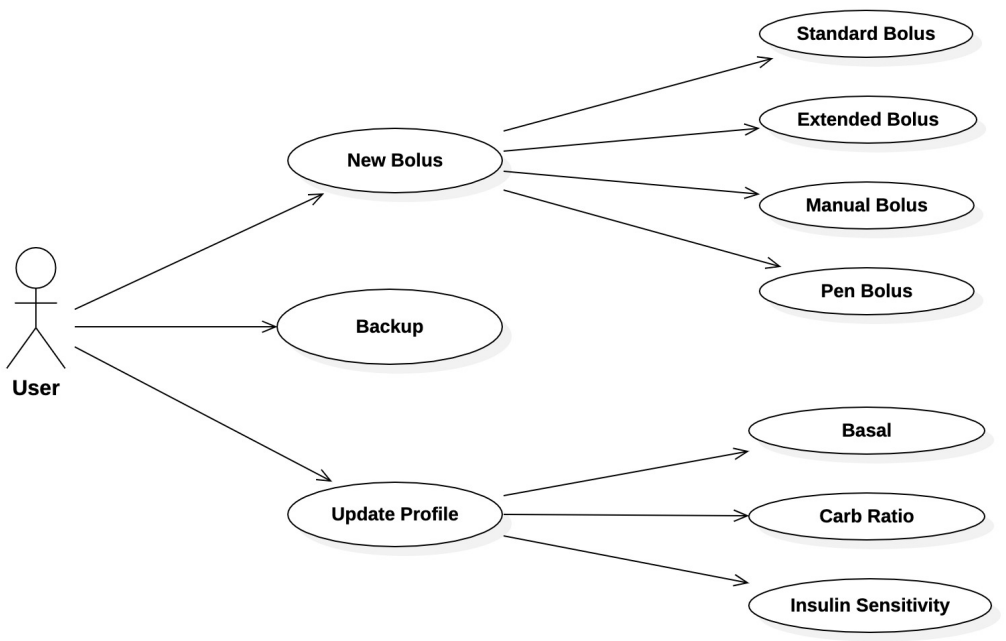

Figura 2: Use Case Diagram

2.2 Use Case Template

Riportiamo di seguito i template relativi a tutti i casi d'uso individuati nel nostro progetto.

Use Case 1	Bolo Standard - Standard Bolus
Livello	User Goal
Descrizione	L'utente immette la quantità di carboidrati e il sistema esegue il bolo d'insulina.
Attori	User
Pre-condizioni	Nessuna
Post-condizioni	Viene inviata la quantità di unità calcolata alla Pump se tale quantità è > 0.
Normale svolgimento	Viene letta l'ultima glicemia, se è di più di 10 minuti fa viene effettuata una nuova misurazione, viene preso il valore dei carboidrati e dell'insulina attiva. Con tutto ciò si calcola l'insulina che il sistema inietterà.
Svolgimenti alternativi	Se i carboidrati sono 0, verranno calcolatate solo le unità per correggere se la glicemia è oltre una certa soglia

Tabella 1: Use Case Template 1 che consiste nel richiedere al sistema il calcolo delle unità e l'iniezione immediata di insulina in modalità standard.

Use Case 2	Bolo Esteso - Extended Bolus
Livello	User Goal
Descrizione	L'utente immette la quantità di carboidrati e quanti minuti aspettare, e il sistema esegue il bolo d'insulina.
Attori	User
Pre-condizioni	Nessuna
Post-condizioni	Viene inviata la quantità di unità dopo un numero di minuti corrispondenti al `delay`
Normale svolgimento	Simile all'UC1 ma con in aggiunta in input di quanti minuti viene ritardata l'iniezione.
Svolgimenti alternativi	Come UC1, se i carboidrati sono 0, verranno calcolatate solo le unità per correggere se la glicemia è oltre una certa soglia

Tabella 2: Use Case Template 2 che consiste nel richiedere al sistema il calcolo delle unità e l'iniezione di insulina in modalità estesa, Mockup in figura 4

Use Case 3	Quante unità? - How Many Units?
Livello	User Goal
Descrizione	L'utente immette i carboidrati e riceve in output quante unità di insulina deve eseguire manualmente
Attori	User
Pre-condizioni	Nessuna
Post-condizioni	Viene salvata la quantità di unità calcolata
Normale svolgimento	Fornisce solo il risultato del calcolo delle unità ma non inietta, indicherà invece quante unità eseguire manualmente.
Svolgimenti alternativi	Se i carboidrati sono 0, verranno calcolate solo le unità per correggere se la glicemia è oltre una certa soglia

Tabella 3: Use Case Template 3 che consiste nel richiedere al sistema il calcolo delle unità e l'iniezione di insulina in modalità manuale.

Use Case 4	Bolo Penna - Pen Bolus
Livello	User Goal
Descrizione	Si inserisce nel sistema quante unità sono state fatte manualmente
Attori	User
Pre-condizioni	Si deve aver già eseguito manualmente un'iniezione di insulina
Post-condizioni	È utile per altri boli eseguiti nelle 3 ore successivi
Normale svolgimento	Viene chiesto il numero di unità iniettate, di conseguenza viene salvato nel database il bolo con le unità prese in input e con orario l'ora di immissione
Svolgimenti alternativi	Se i carboidrati sono 0, verranno calcolate solo le unità per correggere se la glicemia è oltre una certa soglia

Tabella 4: Use Case Template 4 che consiste nel richiedere al sistema il calcolo delle unità da poi iniettare personalmente mediante una penna di insulina.

Use Case 5	Aggiornamento del profilo di Basale - Update basal profile
Livello	User Goal
Descrizione	Si aggiorna il profilo `Basal`, ovvero delle basali (unità di insulina basale, o di fondo o di base necessarie per un singolo orario)
Attori	User
Pre-condizioni	Dev'esserci già un profilo da aggiornare (questo viene creato quando si fa partire il sistema)
Post-condizioni	Viene modificato l'`hourlyFactor` relativo al singolo orario
Normale svolgimento	In input si passa un orario e il numero di unità con cui sovrascrivere quelle precedentemente nell'`hourlyFactor` relativo all'orario
Svolgimenti alternativi	Se viene immesso come orario 0 o 24, il funzionamento è il medesimo

Tabella 5: Use Case Template 5 che consiste nel richiedere al sistema l'aggiornamento del profilo orario della Basale, Mockup in figura 5.

Use Case 6	Aggiornamento del profilo del rapporto insulina carboidrati - Update carb ratio profile
Livello	User Goal
Descrizione	Si aggiorna il profilo `IC`, ovvero dei rapporti insulina carboidrati (peso in grammi dei carboidrati coperto da un'unità di insulina)
Attori	User
Pre-condizioni	Dev'esserci già un profilo da aggiornare (questo viene creato quando si fa partire il sistema)
Post-condizioni	Viene modificato l'`hourlyFactor` relativo al singolo orario
Normale svolgimento	In input si passa un orario e il numero di unità con cui sovrascrivere quelle precedentemente nell'`hourlyFactor` relativo all'orario
Svolgimenti alternativi	Se viene immesso come orario 0 o 24, il funzionamento è il medesimo

Tabella 6: Use Case Template 6 che consiste nel richiedere al sistema l'aggiornamento del profilo orario del rapporto insulina carboidrati.

Use Case 7	Aggiornamento del profilo di sensitività di insulina - Update insulin sensitivity profile
Livello	User Goal
Descrizione	Si aggiorna il profilo `IS`, ovvero delle sensitività insuliniche (di che valore glicemico scende con un'unità di insulina)
Attori	User
Pre-condizioni	Dev'esserci già un profilo da aggiornare (questo viene creato quando si fa partire il sistema)
Post-condizioni	Viene modificato l'`hourlyFactor` relativo al singolo orario
Normale svolgimento	In input si passa un orario e il numero di unità con cui sovrascrivere quelle precedentemente nell'`hourlyFactor` relativo all'orario
Svolgimenti alternativi	Se viene immesso come orario 0 o 24, il funzionamento è il medesimo

Tabella 7: Use Case Template 7 che consiste nel richiedere al sistema l'aggiornamento del profilo orario della sensitività insulinica, Mockup in figura 6.

Use Case 8	Backup
Livello	User Goal
Descrizione	Viene fatto un backup del database `db/data.db` in locale nel file `db/backup.db`
Attori	User
Pre-condizioni	Deve esistere un file `db/data.db` di cui fare il backup
Post-condizioni	Viene creato o sovrascritto il database `backup.db`
Normale svolgimento	Il backup viene fatto in locale nella cartella `db`
Svolgimenti alternativi	Se viene impostato un file `.env` si può uploadare il file di backup su un server sqlite.

Tabella 8: Use Case Template 8 che consiste nell'eseguire una copia del database con il nome di `backup.db`, Mockup in figura 3.

2.3 Mockups

La GUI (Interfaccia Utente Grafica) è stata implementata usando il framework *Swing* ed è compilata in un `.jar` reperibile pubblicamente nella *Release Github*; nell'ultimo capitolo discuteremo come il processo di packaging e distribuzione è stato automatizzato.

Riportiamo di seguito i mockups realizzati relativi alla GUI per l'interazione dell'utente con il gestore del diabete. Oltre ad agire come mockup, la GUI è funzionante in tutti i casi d'uso precedentemente elencati.

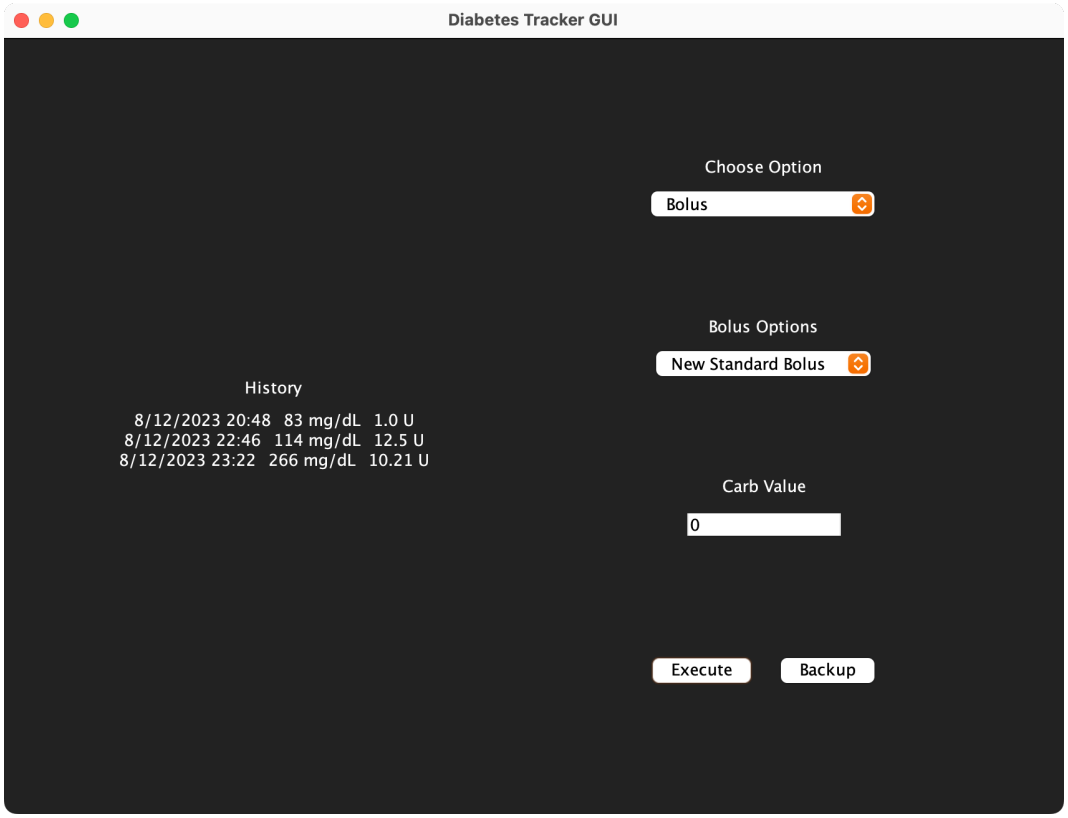

Figura 3: Storico dei boli sulla sinistra, bottone per il backup in basso a destra, UC8

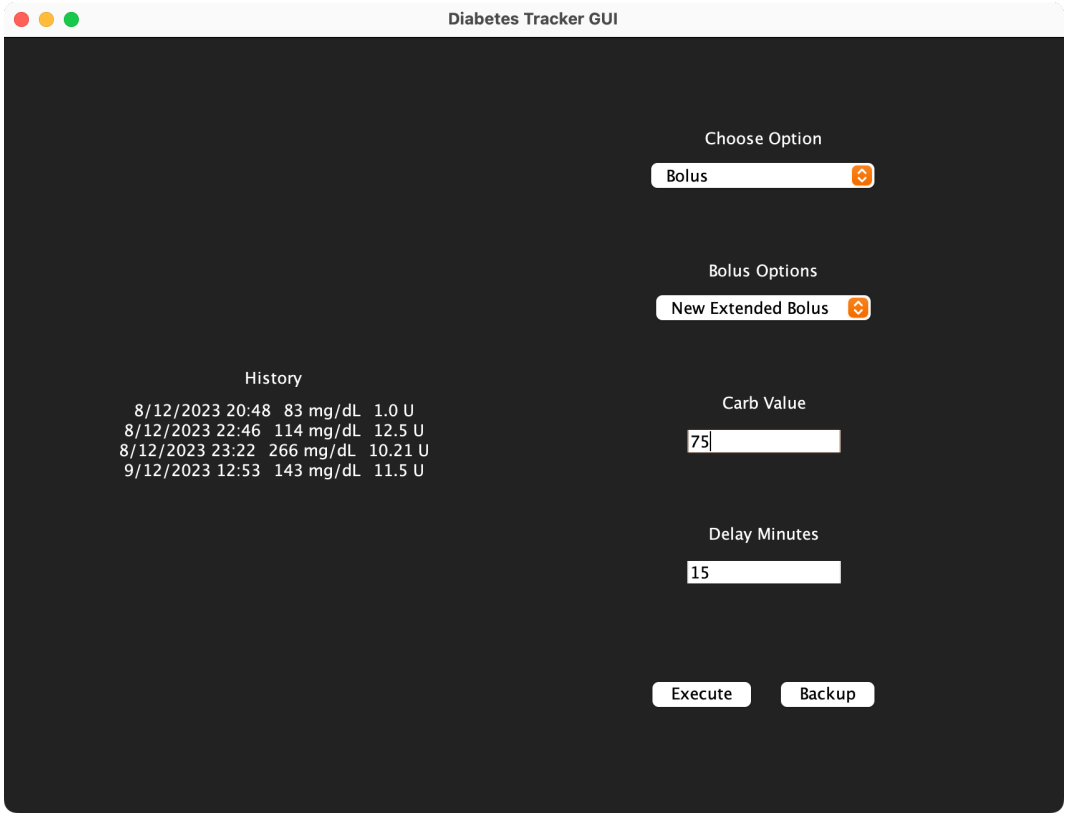

Figura 4: Creazione bolo in modalità estesa, UC2

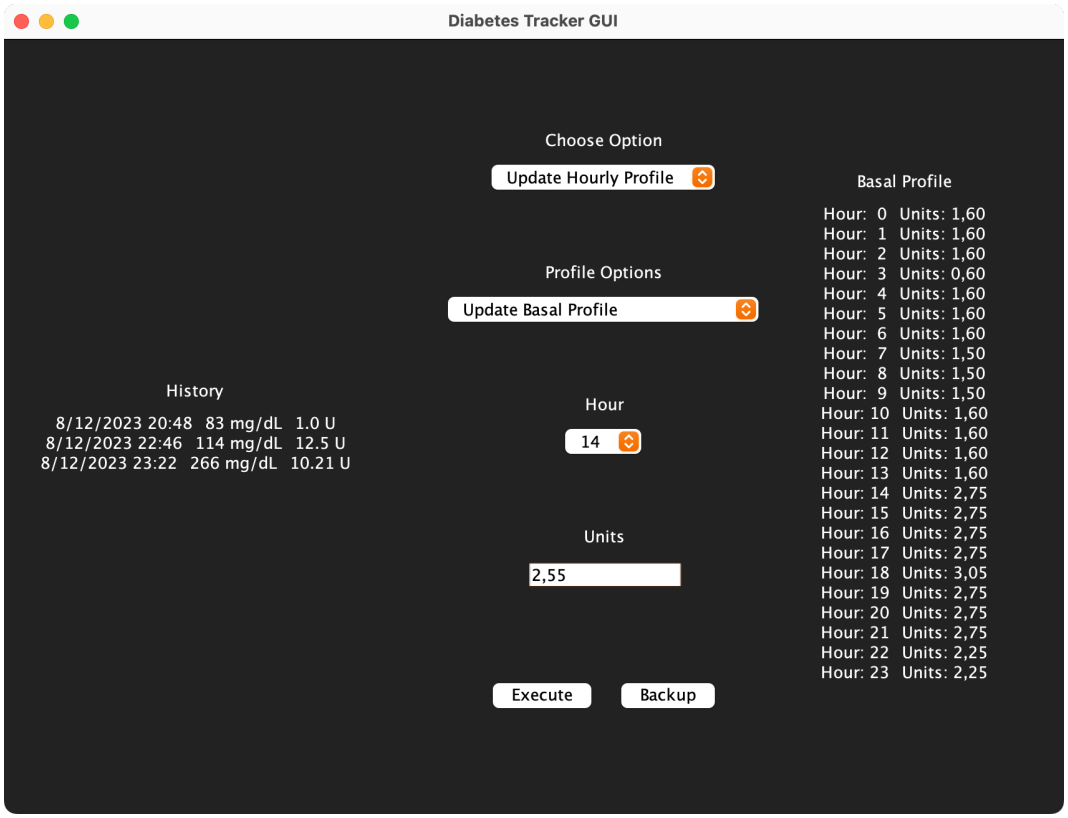

Figura 5: Aggiornamento profilo orario della basale, UC5

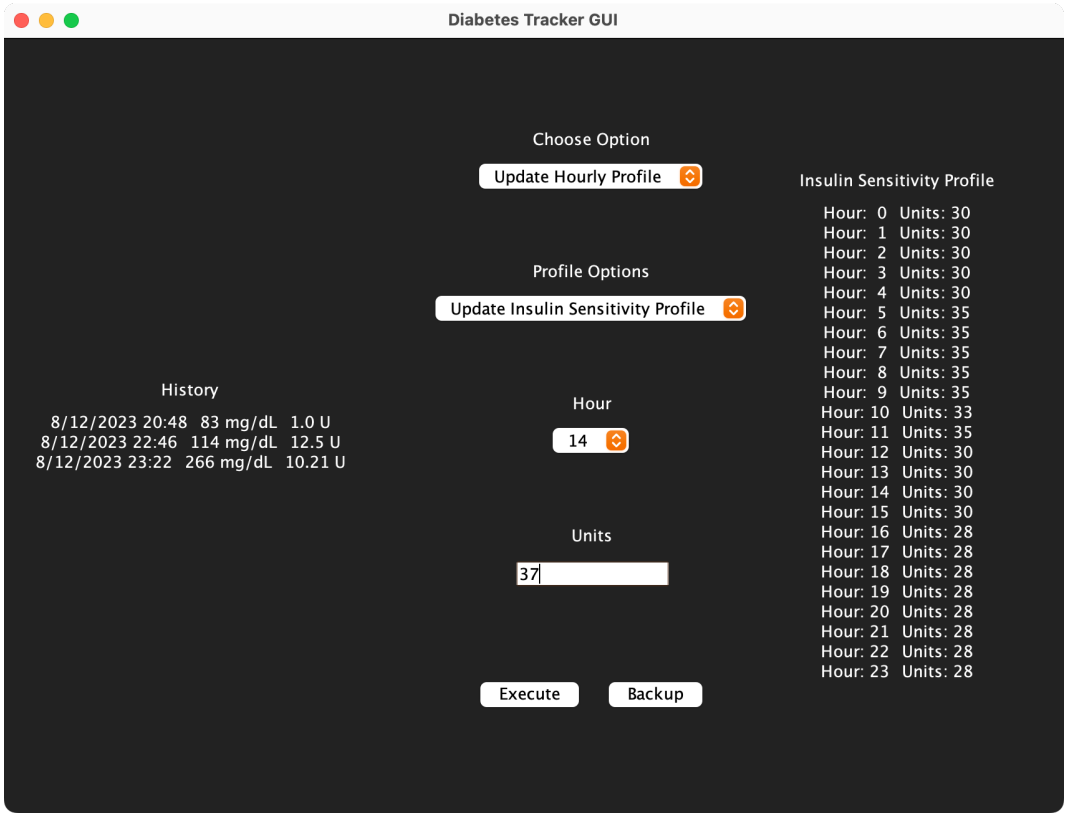

Figura 6: Aggiornamento profilo orario della sensitività dell'insulina, UC7

3 Progettazione e Implementazione

3.1 Scelte implementative e considerazioni

Per implementare il nostro progetto in linguaggio Java ed eseguire i test ci siamo serviti delle IDE *IntelliJIdea* e *Visual Studio Code*. Al fine di semplificare la collaborazione abbiamo utilizzato GitHub come stumento di versionamento del codice. Per quanto riguarda il Class Diagram, l'Use Case Diagram e lo schema architetturale in packages ci siamo serviti del software *StarUML*.

3.2 Class Diagram

Qui di seguito riportiamo la realizzazione del class diagram che descrive la nostra logica di dominio in prospettiva di implementazione:

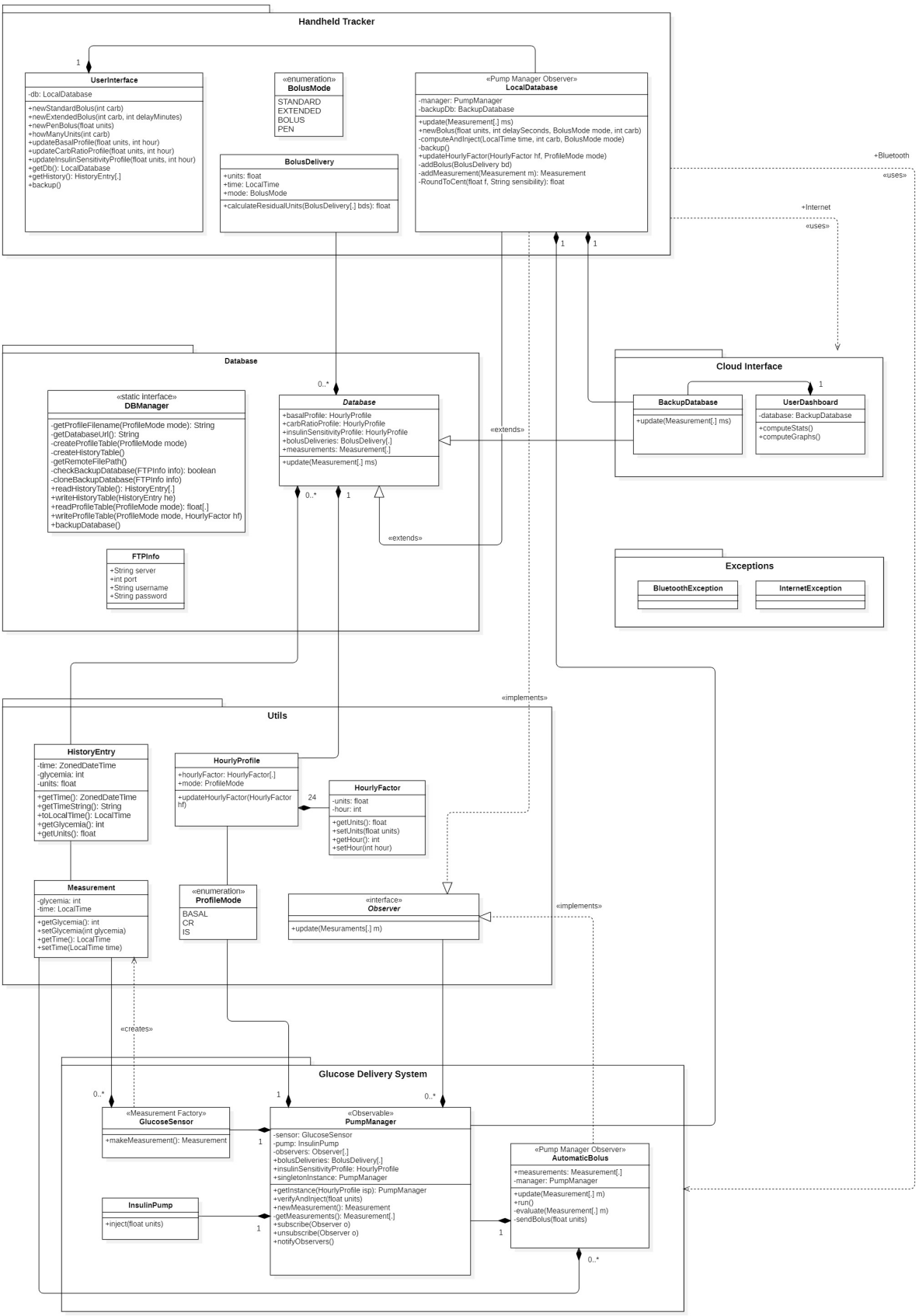

Figura 7: Class Diagram

3.3 Classi ed Interfacce

Per l'implementazione del nostro applicativo abbiamo sia definito nuove classi ed interfacce specifiche, sia utilizzato alcune di quelle già presenti nelle librerie standard di Java.

3.3.1 Package Handheld Tracker

è il package che rappresenta il palmare da cui comandare la pompa di insulina.

User Interface

è l'interfaccia che con tutti i suoi metodi pubblici permette di eseguire tutto ciò che può richiedere l'utente: boli, aggiornamenti dei profili orari e backup.

UserInterface.java

```
public class UserInterface {
    private final LocalDatabase db;
    public UserInterface() { db = new LocalDatabase(); }
    public LocalDatabase getDb() { return db; }
    public HistoryEntry[] getHistory() { return DBManager.readHistoryTable(); }
    public void backup() { DBManager.backupDatabase(); }
    public void newStandardBolus(int carb) { db.newBolus(0, 0, BolusMode.STANDARD, carb);
      . }
    public void newExtendedBolus(int carb, int delayMinutes) { // delay in minutes
        db.newBolus(0, delayMinutes * 60, BolusMode.EXTENDED, carb); // delay in seconds
    }
    public void howManyUnits(int carb) { db.newBolus(0, 0, BolusMode.MANUAL, carb); }
    public void newPenBolus(float units) { db.newBolus(units, 0, BolusMode.PEN, 0); }
    public void updateBasalProfile(float units, int hour) {
        db.updateHourlyFactor(new HourlyFactor(units, hour), ProfileMode.BASAL);
    }
    public void updateCarbRatioProfile(float units, int hour) {
        db.updateHourlyFactor(new HourlyFactor(units, hour), ProfileMode.CR);
    }
    public void updateInsulinSensitivityProfile(float units, int hour) {
        db.updateHourlyFactor(new HourlyFactor(units, hour), ProfileMode.IS);
    }
}
```

Local Database

è il centro nevralgico del sistema, estende la classe `Database` del package `Database` e si occupa di calcolare i boli raccogliendo le risorse necessarie, salva

nel database i dati dei boli e dei profili di gestione e manda al `GlucoseDeliverySystem` (che rappresenta la pump) il calcolo delle unità di insulina da iniettare. Qui di seguito la funzione più importante di questa classe:

LocalDatabase.java – *computeAndInject*

```
try {
    // Get the last measurement if it exists
    if (measurements.isEmpty())
        manager.newMeasurement();

    Measurement lm = measurements.get(measurements.size() - 1);
    // Difference between last measurement and bolus time
    Duration diff = Duration.between(lm.getTime(), time);

    // Make a new measurement if the last one is older than 10 minutes from now
    if (diff.toMinutes() > 10) {
        manager.newMeasurement();
        lm = this.measurements.get(measurements.size() - 1);
    }
    // Get the actual hourly factors from profiles
    HourlyFactor sensitivity =
        insulinSensitivityProfile.hourlyFactors[LocalTime.now().getHour()];
    HourlyFactor carbRatio = carbRatioProfile.hourlyFactors[LocalTime.now().getHour()];

    // Create the BolusDelivery object
    BolusDelivery bd = new BolusDelivery(0, time, mode);

    // Compute the units of correction, units of carbohydrates and total units
    float glycUnits = 0;
    if (lm.getGlycemia() > 160)
        glycUnits = ((float) (lm.getGlycemia() - GLYC_REFERENCE)) /
            sensitivity.getUnits();
    float activeUnits = bd.calculateResidualUnits(bolusDeliveries);
    // Units of correction = Units of glycemia - Units of active insulin
    float correctionUnits = glycUnits - activeUnits;
    float carbUnits = 0;
    if (carb > 0 && carb <= 150)
        carbUnits = carb / carbRatio.getUnits();

    // Round the results to 2 decimal places
    bd.units = roundTo(correctionUnits + carbUnits, 0.01);
    glycUnits = roundTo(glycUnits, 0.01);
    activeUnits = roundTo(activeUnits, 0.01);
    correctionUnits = roundTo(correctionUnits, 0.01);
    carbUnits = roundTo(carbUnits, 0.01);
```

```
if (bd.units > 0) {
    System.out.printf("%-16s%9s%14s%-18s%n", "Glycemia:", lm.getGlycemia() + " mg/dL",
        (glycUnits > 0 ? " " + glycUnits + " units" : ""), (correctionUnits != 0 ? "
        correction" : ""));
    System.out.printf("%-25s%14s%-18s%n", "Active insulin:", (activeUnits > 0 ? "-" :
        "") + activeUnits + " units", "    " + (correctionUnits != 0 ? correctionUnits
        + " units" : ""));
    System.out.printf("%-16s%9s%14s%n", "Carbohydrates:", carb + " g    ", (carbUnits
        > 0 ? " " + carbUnits + " units" : ""));
    System.out.printf("%-25s%14s%n%n", "Total insulin:", (bd.units > 0 ? " " +
        bd.units + " units" : ""));

    boolean result = false;
    switch (mode) {
        case STANDARD:
            result = manager.verifyAndInject(bd.units);
            break;
        case EXTENDED:
            Duration delay = Duration.between(LocalTime.now(), time).plusSeconds(1);
                System.out.print("Waiting ");
            int hours = (int) delay.toHours();
            int minutes = (int) delay.toMinutes() % 60;
            int seconds = (int) delay.toSeconds() % 60;
            if (hours > 0)
                System.out.print(hours + "h ");
            if (minutes > 0)
                System.out.print(minutes + "m ");
            if (seconds > 0)
                System.out.print(seconds + "s ");
            System.out.println("to inject " + bd.units + " units" + " at " +
                time.format(DateTimeFormatter.ofPattern("HH:mm")));
            Thread.sleep(delay.toMillis());

            result = manager.verifyAndInject(bd.units);

            break;
        case MANUAL:
            // Round the units to 0.5
            bd.units = roundTo(bd.units, 0.5);
            System.out.println("Manually inject: " + bd.units + " units");
        case PEN:
            result = true;
            break;
    }
    if (result) {
```

```
            DBManager.writeHistoryTable(new
                HistoryEntry(ZonedDateTime.now()lm.getGlycemia(), bd.units));
        } else {
            System.out.println("Injection failed");
        }
        addBolus(bd);
    } else if (bd.units == 0) {
        System.out.println("You don't need to inject insulin, you have a glycemia of: " +
            lm.getGlycemia() + " mg/dL");
    } else if (bd.units < 0) {
        System.out.println( "You don't need to inject insulin, you have " + activeUnits +
            " units of active insulin");
    }
} catch (Exception e) {
    e.printStackTrace();
    throw new BluetoothException();
}
```

Bolus Delivery

è la classe che ha come attributi `float units; LocalTime time; BolusMode mode` che permettono la rappresentazione di un oggetto bolo. Il suo unico metodo è `calculateResidualUnits` che prendendo in ingresso come parametro `List<BolusDelivery> bds` calcola attraverso una regressione linerare quante unità di insulina sono ancora attive. Qui di seguito la funzione più importante di questa classe:

BolusDelivery.java – calculateResidualUnits

```
public float calculateResidualUnits(List<BolusDelivery> bds) {
    int hours = 3;
    float residualUnits = 0;
    if (!bds.isEmpty()) {
        // Cycle init
        int i = 0;
        // Init the last bolus
        BolusDelivery lb = bds.get(bds.size() - 1);
        Duration diff = Duration.between(lb.time, LocalTime.now());

        // While last bolus is not older than 3 hours
        while (diff.toMinutes() <= hours * 60 && i < bds.size()) {
            // Linear decay
            residualUnits += lb.units * (1 - (float) diff.toMinutes() / (hours * 60));
```

```
            // Cycle increment
            i++;
            // Last bolus
            lb = bds.get(bds.size() - i);
            // Difference between last bolus time and now
            diff = Duration.between(lb.time, LocalTime.now());
        }
    }
    return residualUnits;
}
```

Bolus Mode

rappresenta i 4 diversi tipi di bolo che possono esserci: `STANDARD`, `EXTENDED`, `MANUAL`, `PEN`.

3.3.2 Package Database

DB Manager

si occupa della scrittura, lettura e backup del database e quindi del registro delle glicemie/boli, e dei tre `ProfileMode`: Basale, Rapporto Insulina Carboidrati e Rapporto Sensitività Insulinica.

Database

ha come attributi: `HourlyProfile carbRatioProfile; HourlyProfile insulinSensitivityProfile; HourlyProfile basalProfile; List<BolusDelivery> bolusDeliveries; List<Measurement> measurements;` e fa da classe base per `LocalDatabase` e `BackupDatabase` che implementano il metodo `update`, che aggiunge alla lista dei Measurements le ultime misurazioni.

FTP Info

ha come attributi: `String server; int port; String username; String password` che permettono di caricare il database su un server sqlite con le credenziale dettate da questa classe.

3.3.3 Package Utils

History Entry

è il tipo base che ha come attributi glicemia, unità (d'insulina) e orario

di somministrazione. Uno dei quattro registri nel database è composto da `HistoryEntry`. Come metodi sono implementati i getter e i setter.

Measurement

è il tipo base che rappresenta una misurazione di glicemia (di cui il creatore è `glucoseDeliverySystem.GlucoseSensor`) e ha come attributi un intero corrispondente alla densità di glucosio nel sangue, misurata in mg/dL e l'orario. Come metodi sono implementati i getter e i setter.

Hourly Factor

è il tipo base che va a comporre tre dei quattro registri nel database: `BasalProfile`, `InsulineSensitivityProfile` e `CarbRatioProfile`. Ha come attributi delle unità e un orario (compreso tra 0 e 23, 24 coincide con 0) Come metodi sono implementati i getter e i setter.

Hourly Profile

è composto da una lista di 24 `HourlyProfile` e da un attributo `ProfileMode mode`. Alla sua creazione se non è reperibile dal database la modalità corrispondente (attraverso il metodo `readProfileTable(mode)` della classe `DBManager` dal package `Database`), si andrà a creare un profilo di default. Qui di seguito la funzione più importante di questa classe che verifica che per ogni modalità i valori con cui aggiornare l'`HourlyFactor` siano in un certo range che permette sicurezza dell'utente:

HourlyProfile.java – updateHourlyFactor

```
public void updateHourlyFactor(HourlyFactor hf) {
    boolean success = false;
    String modeString = "";
    if (hf.getHour() < 0 || hf.getHour() > 23)
        hf.setHour(hf.getHour());
    switch (mode) {
        case BASAL:
            modeString = "basal profile";
            if (hf.getUnits() >= 0.1 && hf.getUnits() <= 5)
                success = true;
            break;
        case CR:
            modeString = "carb ratio";
            if (hf.getUnits() >= 1 && hf.getUnits() <= 15)
```

```
                success = true;
            break;
        case IS:
            modeString = "insulin sensitivity";
            if (hf.getUnits() >= 20 && hf.getUnits() <= 50)
                success = true;
            break;
    }
    if (success) {
        System.out.println("Updating " + modeString + " h=" + hf.getHour() + " from " +
          String.format("%.2f", hourlyFactors[hf.getHour()].getUnits()).replace(",",
          ".") + " to " + String.format("%.2f", hf.getUnits()).replace(",", ".") +
          "...");
        hourlyFactors[hf.getHour()] = hf;
    } else {
        System.out.println("Invalid " + modeString + " value");
    }
}
```

Profile Mode

rappresenta i 3 diversi tipi di HourlyProfile che possono esserci: `BASAL`, `CR`, `IS` corrispondenti rispettivamente ai profili di Basale, Rapporto insulina carboidrati e Rapporto Sensitività insulinica.

Observer

interfaccia per il design pattern Observer implementato poi nel `handheld-Tracker.LocalDatabase`.

3.3.4 Package Glucose Delivery System

Pump Manager

gestore principale del `GlucoseDeliverySystem`. Implementa i design pattern Observer, chiama il Factory e la creazione della sua istanza è un Singleton (esisterà solo un oggetto `PumpManager`). Permette di mandare le informazioni all'`InsulinPump` tramite il metodo:

PumpManager.java – verifyAndInject

```java
public boolean verifyAndInject(float units) {
    if (units > 0 && units <= 20) {
        bds.add(new BolusDelivery(units, LocalTime.now(), BolusMode.STANDARD));
        pump.inject(units);
        return true;
    } else if (units > 20) {
        System.out.println("Too many units");
    } else {
        System.out.println("Invalid units");
    }
    return false;
}
```

permette di chiamare una nuova misurazione e dunque la chiamata della factory `GlucoseSensor` tramite il metodo:

PumpManager.java – verifyAndInject

```java
public Measurement newMeasurement() {
    Measurement m = sensor.makeMeasurement();
    this.measurements.add(m);

    notifyObservers();
    return m;
}
```

Glucose Sensor

crea un oggetto `Measurement` e di fatto implementa il desing pattern Factory.

Insulin Pump

inietta il quantitativo di insulina che riceve dal `PumpManager` e stampa il risultato.

Automatic Bolus

fornisce una routine che ogni 10 o 15 minuti inietta rispettivamente $\frac{1}{6}$ o $\frac{1}{4}$ della basale relativa all'orario corrente, a cui è sommata una correzione per far tornare la glicemia in un range corretto, indicativamente di $80 - 150$ (non è sommata se la glicemia è già nel range), e manda il tutto al PumpManager.

3.3.5 Package Cloud Interface

Backup Database

tiene traccia del backup del database quando invocato.

User Dashboard

fornisce statistiche e grafici dei dati presenti nella History.

3.3.6 Package Exceptions

Bluetooth Exception

classe che rappresenta l'eccezione se non riesce il collegamento tra l'Handheld Tracker e il Glucose Delivery System.

Internet Exception

classe che rappresenta l'eccezione se non riesce il collegamento tra l'Handheld Tracker e la Cloud Interface.

3.4 Design Patterns

All'interno del progetto abbiamo avuto la necessità di introdurre alcuni design patterns noti per favorire la gestione di alcune dipendenze tra classi in modo agile ed elegante. I patterns utilizzati sono:

- **Observer**
- **Singleton**
- **Factory**

3.4.1 Observer

Implementazione di Observer - handheldTracker.LocalDatabase.java

```
public class LocalDatabase extends Database implements Observer {
    // ...
    @Override
    public void update(List<Measurement> ms) {
        super.update(ms);
        backup();
    }
}
```

```
class Database {
    public List<Measurement> measurements;

    protected void update(List<Measurement> ms) {
        for (Measurement m : ms) {
            measurements.add(m);
        }
    }
}
```

Implementazione di Observable – glucoseDeliverySystem.PumpManager

```
public class PumpManager {
    private List<Measurement> measurements = new ArrayList<>();
    private final List<Observer> observers = new ArrayList<>();
    // ...
    public void subscribe(Observer o) { observers.add(o); }

    public void unsubscribe(Observer o) { observers.remove(o); }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this.measurements);
        }
    }
}
```

3.4.2 Singleton

Implementazione di Singleton – glucoseDeliverySystem.PumpManager

```
public class PumpManager {
    // ...
    private static PumpManager SingletonInstance;

    private PumpManager(HourlyProfile isp) { ... }

    public static PumpManager getInstance(HourlyProfile isp) {
        if (SingletonInstance == null) {
            SingletonInstance = new PumpManager(isp);
        } else {
            SingletonInstance.insulinSensitivityProfile = isp;
        }
        return SingletonInstance;
    }
}
```

3.4.3 Factory

Implementazione di Factory – glucoseDeliverySystem.GlucoseSensor

```
public class GlucoseSensor {
    public Measurement makeMeasurement() {
        int min = 60;
        int max = 300;
        int glycemia = (int) Math.abs(Math.random() * (max - min) + min);
        return new Measurement(glycemia, LocalTime.now());
    }
}
```

3.5 Disposizione delle classi nei vari package

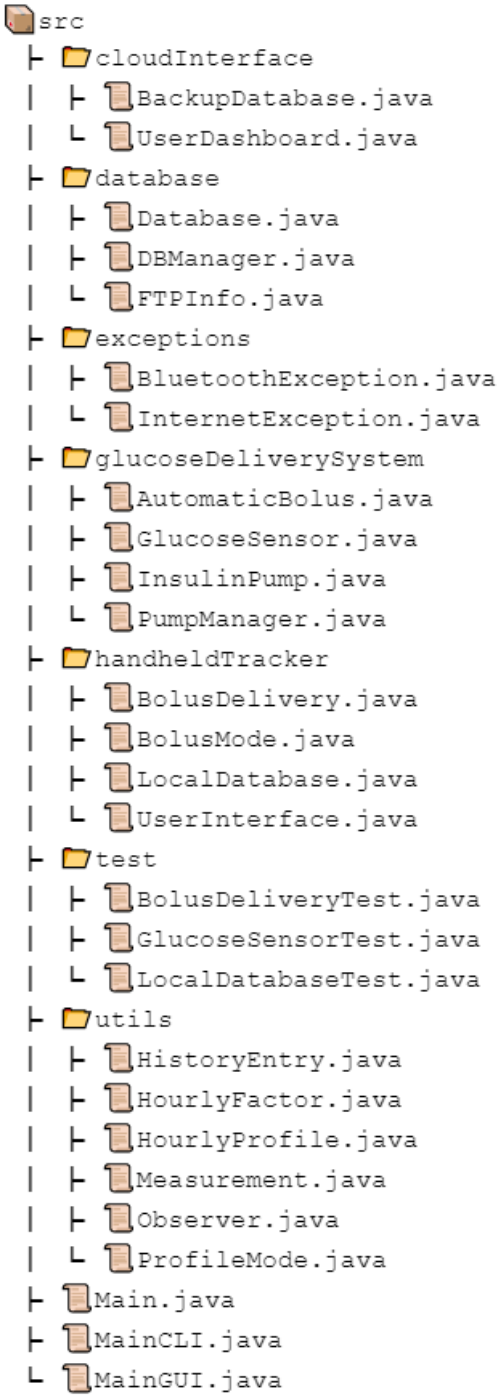

Figura 8: Raffigurazione della disposizione delle classi del progetto nei package

4 Unit Test

Per testare la corretta interazione e collaborazione fra le parti abbiamo realizzato i seguenti test cases per alcune delle classi principali dell'applicativo: Nel progetto è stato utilizzato il framework *JUnit 5*. In tutte e tre le classi di test faremo uso di

```
Import necessari per il testing

import org.junit.Test;
import static org.junit.Assert.*;
```

L'import statico permette di chiamare direttamente il metodo `assertEquals(expected,actual,delta)` che fa fallire il test se

$$expected \neq actual \pm delta$$

È possibile trovare tutta la parte di codice riguardante all'unit testing nella cartella *src/test*

4.1 Bolus Delivery Test

In questa suite di test viene controllato che gli oggetti creati appartenenti alla classe `BolusDelivery`

4.1.1 Constructor Test

Il metodo `ConstructorTest` nella classe `BolusDeliveryTest` è un test unitario che verifica che avvenga correttamente la creazione dell'oggetto `BolusDelivery` in tutte e 4 le modalità: standard, esteso, manuale e con penna. Inizialmente viene casualmente deciso un certo numero di unità in un range tra 1 e 15 con uno scalino di 0.01 per le prime due modalità, e con lo scalino di 0.5 per le restanti modalità; viene ulteriormente creato anche un `delay` tra 1 e 15 minuti. Si crea l'oggetto con come parametri passati in ingresso al costruttore i valori creati precedentemente, in seguito si testa che gli attributi dell'oggetto corrispondano a quelli che con cui è stato creato l'oggetto: `units` e orario te-

stando ora, minuti e secondi dell'attributo `time`. Tutto ciò viene fatto per tutte e quattro le modalità.

4.1.2 Calculate Residual Units Test

Il metodo `calculateResidualUnitsTest` nella classe `BolusDeliveryTest` è un test unitario che verifica il corretto funzionamento del metodo `calculateResidualUnits` della classe `BolusDelivery`. Inizializza alcune variabili, genera un oggetto `BolusDelivery` con attributi casuali, e successivamente esegue due test:

1. Il primo test verifica che venga restituito `0` quando chiamato su una lista vuota di oggetti `BolusDelivery`.

2. Il secondo test aggiunge l'oggetto `BolusDelivery` generato a una lista e verifica che il metodo restituisca un valore specifico, calcolato in base agli attributi dell'oggetto e agli elementi nella lista.

4.2 Glucose Sensor Test

In questa suite di test viene controllato che gli oggetti creati appartenenti alla classe `GlucoseSensor`

4.2.1 Constructor Test

Il metodo `ConstructorTest` nella classe `GlucoseSensorTest` è un test unitario che verifica il corretto funzionamento del metodo `newMeasurement` della classe `PumpManager`. In breve, il test crea un'istanza di `PumpManager` con un profilo orario, chiama il metodo `newMeasurement` per generare una nuova misurazione della glicemia, la memorizza in una variabile e verifica che questa misurazione sia correttamente aggiunta alla lista interna di misurazioni. Se l'asserzione passa, il test conferma che il metodo `newMeasurement` funziona come previsto; altrimenti, indica la presenza di un bug che richiede correzione.

4.3 Local Database Test

In questa suite di test viene controllato che gli oggetti creati appartenenti alla classe `LocalDatabase`

4.3.1 Constructor Test

nella classe `LocalDatabaseTest` è un test unitario che verifica il corretto funzionamento del costruttore nella classe `LocalDatabase`. Ecco una spiegazione passo dopo passo di come funziona:

1. Innanzitutto, crea un'istanza di `LocalDatabase` chiamando il suo costruttore.

2. Successivamente, verifica tramite un'asserzione che la lista `bolusDeliveries` nell'istanza di `LocalDatabase` sia vuota. Ciò verifica che il costruttore inizializzi correttamente la lista `bolusDeliveries`.

3. Verifica anche tramite un'asserzione che la lista `measurements` nell'istanza di `LocalDatabase` sia vuota. Ciò verifica che il costruttore inizializzi correttamente la lista `measurements`.

4. Verifica tramite un'asserzione che l'array `hourlyFactors` nei profili `carbRatioProfile`, `insulinSensitivityProfile` e `basalProfile` dell'istanza di `LocalDatabase` abbia ciascuno una lunghezza di 24. Ciò verifica che il costruttore inizializzi correttamente questi profili con 24 fattori orari.

Se tutte le asserzioni passano, significa che il costruttore nella classe `LocalDatabase` sta funzionando come previsto. Se una qualsiasi asserzione fallisce, l'intero test fallisce.

4.3.2 New Bolus Test

Il test `newBolusTest` nella classe `LocalDatabaseTest` verifica il corretto funzionamento del metodo `newBolus` nella classe `LocalDatabase`. Durante il test, viene creata un'istanza di `LocalDatabase`, generato un valore casuale rappresentante le unità di insulina, creato un oggetto `BolusDelivery` con questo valore e altre informazioni, chiamato il metodo `newBolus`, e infine verificato che la

lista di somministrazioni `bolusDeliveries` sia stata aggiornata correttamente con l'aggiunta di questo nuovo oggetto `BolusDelivery`. Se tutte le verifiche passano, il test conferma che il metodo `newBolus` funziona come previsto; in caso contrario, il test fallisce.

4.3.3 Compute And Inject Test

Il test `computeAndInjectTest` nella classe `LocalDatabaseTest` verifica il corretto funzionamento del metodo `computeAndInject` della classe `LocalDatabase`, indirettamente attraverso il metodo `newBolus`. Di seguito una spiegazione in breve del funzionamento:

1. Viene creata un'istanza di `LocalDatabase` chiamata `db`.
2. Si generano valori casuali per i carboidrati `c` e la glicemia `g`.
3. Viene aggiunto un nuovo oggetto `Measurement` alla lista `measurements` in `db` rappresentando una misurazione della glicemia.
4. Vengono recuperati i fattori orari per la sensibilità ai carboidrati `csens` e la sensibilità all'insulina `isens` da `db`.
5. Si calcolano le unità attese `u` di insulina considerando i fattori orari e la presenza di correzione.
6. Si randomizza la modalità del bolo `mode` e, se applicabile, il ritardo.
7. Viene creato un oggetto `BolusDelivery` con le informazioni calcolate.
8. Si chiama il metodo `newBolus` con le informazioni dell'oggetto `BolusDelivery`, il quale a sua volta richiama internamente `computeAndInject`.
9. Dopo la chiamata, il test verifica che la lista `bolusDeliveries` in `db` sia stata aggiornata correttamente, confrontando l'ultima voce con l'oggetto `BolusDelivery` creato.
10. Vengono verificati unità, orario e modalità dell'ultimo `BolusDelivery` nella lista. Se tutti questi valori corrispondono a quelli dell'oggetto `BolusDelivery`, il test è superato.

4.3.4 Update Hourly Factor Test

Il test updateHourlyFactorTest è un test unitario nella classe LocalDatabaseTest che verifica il corretto funzionamento del metodo updateHourlyFactor della classe LocalDatabase. Ecco una spiegazione dettagliata di come funziona:

1. Inizialmente, crea un'istanza di LocalDatabase denominata db.

2. Successivamente, inizializza due array hours e units di dimensione 3. L'array hours conterrà ore casuali tra 0 e 23, mentre l'array units conterrà unità casuali di insulina.

3. Il test esegue quindi tre sub-test, uno per ciascuna modalità di profilo: BASAL, CR (Rapporto Carboidrati), e IS (Sensibilità Insulinica).

4. Per il test del profilo BASAL, genera un numero casuale in virgola mobile units[0] tra 0.01 e 5 con una precisione di 0.05. Crea un oggetto HourlyFactor chiamato hf0 con units[0] e un'ora casuale hours[0]. Successivamente, chiama il metodo updateHourlyFactor dell'istanza di LocalDatabase chiamata db con hf0 e ProfileMode.BASAL. Dopo la chiamata del metodo, il test verifica se l'array list hourlyFactors nel profilo basalProfile dell'istanza di LocalDatabase db è stato aggiornato correttamente. Fa ciò verificando tramite un'asserzione che l'ora e le unità dell'HourlyFactor all'indice hours[0] corrispondano all'ora e alle unità di hf0.

5. Per il test del profilo CR, genera un numero intero casuale units[1] tra 1 e 15. Crea un oggetto HourlyFactor chiamato hf1 con units[1] e un'ora casuale hours[1]. Successivamente, chiama il metodo updateHourlyFactor dell'istanza di LocalDatabase db con hf1 e ProfileMode.CR. Dopo la chiamata del metodo, il test verifica se l'array list hourlyFactors nel profilo carbRatioProfile dell'istanza di LocalDatabase db è stato aggiornato correttamente. Fa ciò verificando tramite un'asserzione che l'ora e le unità dell'HourlyFactor all'indice hours[1] corrispondano all'ora e alle unità di hf1.

6. Per il test del profilo `IS`, genera un numero intero casuale `units[2]` tra 20 e 50. Crea un oggetto `HourlyFactor` chiamato `hf2` con `units[2]` e un'ora casuale `hours[2]`. Successivamente, chiama il metodo `updateHourlyFactor` dell'istanza di `LocalDatabase db` con `hf2` e `ProfileMode.IS`. Dopo la chiamata del metodo, il test verifica se l'array list `hourlyFactors` nel profilo `insulinSensitivityProfile` dell'istanza di `LocalDatabase db` è stato aggiornato correttamente. Fa ciò verificando tramite un'asserzione che l'ora e le unità dell'`HourlyFactor` all'indice `hours[2]` corrispondano all'ora e alle unità di `hf2`.

5 Integrazione Continua

Per rilasciare automaticamente il pacchetto GUI abbiamo sviluppato un *Workflow Github* che esegue lo script Maven di build e successivamente crea una *Release Github* contenente il file `.jar` aggiornato.

Questo workflow viene eseguito ogni volta che il codice è modificato sulla repository pubblica, cioè quando viene fatto un nuovo commit o pull request sul branch principale oppure manualmente tramite l'UI di Github cliccando il *bottone relativo a questa azione*.

maven.yml

```yaml
steps:
  - uses: actions/checkout@v3

  - name: "Set up JDK 11"
    uses: actions/setup-java@v3
    with:
      java-version: "11"
      cache: maven

  - name: "Build with maven"
    run: mvn clean -B package --file pom.xml

  - name: "Set up git"
    run: git config --global user.email "github-actions@github.com" && git config --global
    user.name "GitHub Actions"

  - name: "Change file name"
    run: mv ./target/swe-diab-1.0.0-jar-with-dependencies.jar ./target/swe-diab.jar

  - name: "Delete previous release"
    run: gh release delete v1.0.0 --yes

  - name: "Create new release"
    run: gh release create v1.0.0 ./target/swe-diab.jar -t "v1.0.0" -n "Version 1.0.0"

  - name: "Upload executable as release asset"
    run: gh release upload v1.0.0 ./target/swe-diab.jar --clobber
```

Elenco delle figure

Elenco delle tabelle