

Aplicação Móvel para Registo de Hábitos de Saúde e Bem-Estar



Habittus

The logo consists of the word "Habittus" written in a large, flowing, black cursive script font. A horizontal black underline is positioned below the script, extending from approximately the middle of the first "t" to the end of the "us".

Licenciatura em Engenharia Informática – EaD

LEIF003ON1. Programação Orientada por Objetos

Grupo 11

20240458 | Rui Filipe da Silva Encarnação

20240609 | Cátia Alexandra Ferreira Macedo

20241639 | Carlos Manuel Ferreira Dias

20241693 | Rute Maria da Fonseca Gouveia Frutuoso

20240455 | Daniel Chambel da Cruz

20240964 | Carlos Daniel Pinto Vieira

20241915 | Marcelo Agostinho Lopes

Índice

1. Introdução	1
1.1 Enquadramento do Projeto Integrado	1
1.2 Objetivos da Unidade Curricular de POO	1
1.3 Relação entre a Base de Dados e a Lógica Orientada a Objetos	2
1.4 Metodologia Aplicada.....	2
2. Fundamentos da Programação Orientada por Objetos (POO).....	3
2.1 Conceito e Paradigma.....	3
2.2 Vantagens da POO	4
2.3 Princípios Fundamentais da POO.....	4
3. Modelação Conceptual e UML	8
3.1 Diagrama de Classes – Estrutura Global	8
3.2 Relações entre Classes e o Modelo de Dados (ER → Classes Flutter).....	9
3.3 Multiplicidades e Dependências.....	10
3.4 Justificação das Decisões de Modelação	10
3.5 Figura do Diagrama UML.....	11
4. Atributos, Construtores e Métodos	12
4.1 Declaração e Tipos de Atributos	12
4.2 Construtores e Inicialização de Objetos	14
4.3 Métodos e Operações do Sistema	16
4.4 Exemplo Prático de Implementação em Flutter.....	18
5.2 Ciclos de Repetição e Controlo de Fluxo	22
5.3 Aplicação em Registos de Hábitos.....	23
5.4 Benefícios do Uso de Estruturas Dinâmicas	25

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

6. Encapsulamento, Herança e Polimorfismo.....	25
6.1 Encapsulamento e Modificadores de Acesso	25
6.2 Herança Aplicada ao Sistema de Hábitos	27
6.3 Polimorfismo e Interfaces Comuns	29
6.5 Benefícios da Aplicação dos Três Princípios	32
7. Padrões de Design e Boas Práticas	32
7.1 Princípios SOLID.....	32
7.2 Padrões Utilizados (Singleton, Factory, Observer)	33
7.3 Modularidade e Reutilização	37
7.4 Boas Práticas Adotadas.....	37
8. Integração com a Base de Dados e Flutter	38
8.1 Camada DAO (Data Access Object)	38
8.2 Integração com Supabase	40
8.3 Comunicação entre Camadas (MVC)	41
8.4 Integração com o Flutter (Frontend).....	42
8.5 Benefícios da Integração Multicamadas	43
9. Conclusão	44
9.1 Síntese dos Resultados	44
9.2 Considerações Finais.....	44
10. Referências Bibliográficas	46
11. Anexos	47
11.1 Diagrama Entidade-Relacionamento (ER)	47
11.2 Diagrama de Classes UML	50

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

1. Introdução

1.1 Enquadramento do Projeto Integrado

O presente relatório é desenvolvido no âmbito da Unidade Curricular de Programação Orientada por Objetos (POO), integrando o projeto interdisciplinar das três Unidades Curriculares do terceiro semestre do curso de Engenharia Informática EaD: *Bases de Dados, Programação de Dispositivos Móveis e Programação Orientada por Objetos*.

Após a definição e normalização da Base de Dados relacional no relatório anterior, esta nova etapa visa a modelação e implementação da lógica do sistema através dos princípios da Programação Orientada por Objetos, servindo como base conceptual para a futura integração com a aplicação móvel desenvolvida em Flutter e o backend alojado em Supabase.

A aplicação em desenvolvimento tem como objetivo apoiar os utilizadores na monitorização de hábitos de saúde e bem-estar, permitindo o registo diário de informações como consumo de água, refeições, horas de sono, atividade física e estados de humor. Esta abordagem combina boas práticas de modelação de dados com a abstração e modularidade proporcionadas pela POO, resultando num sistema escalável, seguro e coerente.

1.2 Objetivos da Unidade Curricular de POO

A disciplina de Programação Orientada por Objetos tem como finalidade desenvolver competências de análise, modelação e implementação de soluções modulares, através da criação de classes, objetos e métodos que representem de forma estruturada os componentes de um sistema real.

No contexto deste projeto, os principais objetivos são:

- Aplicar os conceitos fundamentais da POO (classe, objeto, encapsulamento, herança, polimorfismo e abstração);
- Traduzir as entidades do modelo de dados para classes Flutter, estabelecendo uma correspondência direta com a base de dados relacional;
- Garantir a modularidade e reutilização do código, permitindo futuras expansões do sistema (ex.: registo de novos hábitos);

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

- Implementar padrões de design e boas práticas de desenvolvimento;
- Preparar a integração entre a lógica orientada a objetos e as camadas de interface móvel (Flutter) e persistência de dados (Supabase).

Desta forma, a unidade curricular reforça a ligação entre a teoria e a prática, aproximando os alunos de um cenário real de desenvolvimento de software integrado.

1.3 Relação entre a Base de Dados e a Lógica Orientada a Objetos

Enquanto o relatório de *Bases de Dados* definiu a estrutura relacional que suporta a aplicação, o presente documento dedica-se à implementação da camada lógica, responsável pela manipulação e interação com os dados.

Cada entidade da base de dados é agora representada por uma classe Flutter, cujos atributos correspondem aos campos das tabelas, e cujos métodos encapsulam a lógica de negócio associada.

Por exemplo:

- A entidade User transforma-se na classe User, que contém atributos como userId, email e name, e métodos como addHabit() ou addGoal().
- A entidade Meal converte-se na classe Meal, associada a uma lista de objetos MealItem, permitindo cálculos dinâmicos do valor energético total.

Assim, a lógica orientada por objetos atua como intermediária entre o utilizador e a base de dados, assegurando coerência estrutural e integridade funcional em toda a aplicação.

1.4 Metodologia Aplicada

A metodologia adotada neste relatório baseia-se na sequência natural de desenvolvimento de software orientado a objetos:

1. Identificação das entidades e relações a partir do modelo Entidade-Relação (ER) anteriormente criado;
2. Modelação UML, através de um Diagrama de Classes que traduz as entidades da BD em classes Flutter;

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

3. Definição de atributos, construtores e métodos, com base nas regras de negócio previamente definidas;
4. Implementação da hierarquia de classes, aplicando conceitos como herança, encapsulamento e polimorfismo;
5. Integração com a base de dados e camada móvel, segundo o modelo MVC (Model-View-Controller);
6. Validação e documentação segundo as boas práticas de engenharia de software e as normas APA.

O resultado pretende demonstrar a integração coerente entre modelação de dados, lógica de programação e interface móvel, reforçando o domínio técnico e conceptual dos princípios da POO no desenvolvimento de aplicações completas e escaláveis.

2. Fundamentos da Programação Orientada por Objetos (POO)

2.1 Conceito e Paradigma

A Programação Orientada por Objetos (POO) representa um dos paradigmas mais amplamente utilizados na engenharia de software moderna. Baseia-se na ideia de modelar o mundo real através de objetos, que possuem características (atributos) e comportamentos (métodos), permitindo uma estrutura de código mais modular, flexível e reutilizável.

Em vez de centrar o desenvolvimento em procedimentos e funções isoladas, a POO organiza o sistema em classes, que servem de modelo para a criação de objetos concretos. Estes objetos interagem entre si de forma controlada, simulando entidades reais do domínio da aplicação.

No contexto deste projeto, cada componente do sistema como o utilizador, o registo de refeições, o consumo de água ou as metas de bem-estar é representado como um objeto independente, mas integrado num ecossistema comum. Assim, o código reflete diretamente as relações e operações que já foram definidas no modelo de dados, promovendo uma forte coerência entre as camadas de persistência (Base de Dados) e lógica (POO).

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

2.2 Vantagens da POO

A adoção da Programação Orientada por Objetos traz diversas vantagens para o desenvolvimento de software de média e grande escala, destacando-se:

- **Modularidade:** o código é dividido em blocos lógicos independentes (classes), o que facilita a manutenção e a atualização do sistema.
- **Reutilização de código:** as classes podem ser reutilizadas em outros módulos ou projetos, evitando redundâncias e acelerando o desenvolvimento.
- **Facilidade de manutenção:** uma alteração feita numa classe não afeta outras partes do código, desde que as interfaces públicas sejam mantidas.
- **Abstração e clareza:** a POO permite esconder detalhes técnicos complexos, expondo apenas o essencial para a interação entre objetos.
- **Escalabilidade:** novas funcionalidades podem ser adicionadas através da criação de novas classes ou da extensão de classes existentes, sem comprometer o funcionamento global.
- **Modelagem realista:** o código passa a representar entidades reais do domínio da aplicação, facilitando o entendimento do sistema e a comunicação entre programadores e stakeholders.

No caso da aplicação de registo de hábitos de saúde e bem-estar, estas vantagens traduzem-se em:

- Facilidade em adicionar novos tipos de regtos (ex.: hidratação, sono, humor);
- Clareza na separação de responsabilidades (ex.: User, Goal, Reminder);
- Flexibilidade na implementação de futuras funcionalidades, como gráficos ou relatórios personalizados.

2.3 Princípios Fundamentais da POO

Os pilares da Programação Orientada por Objetos constituem a base sobre a qual o sistema é estruturado. Estes princípios **classe, objeto, encapsulamento, herança, polimorfismo e abstração** são aplicados de forma integrada ao longo do projeto, garantindo robustez e escalabilidade.

a) Classe

POO

Uma **classe** é o modelo ou estrutura que define as características e comportamentos de um conjunto de objetos semelhantes. Em Flutter, é composta por **atributos** (dados) e **métodos** (funções).

Exemplo prático no projeto:

```

public class WaterIntake {

    private int waterId;

    private double amountML;

    private LocalDateTime recordedAt;

    public WaterIntake(double amountML, LocalDateTime recordedAt) {

        this.amountML = amountML;

        this.recordedAt = recordedAt;

    }

    public double summary() { return amountML; }

    public void register(double amountML) { this.amountML = amountML; }

}

```

Esta classe representa a entidade WATER_INTAKE do modelo de dados, encapsulando os valores de quantidade e data de registo de forma segura.

b) Objeto

Um **objeto** é uma instância concreta de uma classe. Cada objeto possui os seus próprios valores de atributos, representando uma entidade individual.

No exemplo anterior, a criação de um novo registo de consumo de água seria feita assim:

```
WaterIntake registo = new WaterIntake(250, LocalDateTime.now());
```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Aqui, o objeto registo representa uma ocorrência específica do hábito de hidratação de um utilizador.

c) Encapsulamento

O **encapsulamento** é o princípio que garante o controlo de acesso aos dados internos de um objeto. Através de modificadores de acesso como `private`, `protected` e `public`, apenas métodos específicos podem ler ou alterar determinados atributos.

Isto promove a segurança da informação e evita alterações indevidas, algo essencial quando lidamos com dados pessoais de saúde.

No projeto, este princípio é aplicado para proteger dados sensíveis, como peso, altura e dados de perfil do utilizador.

d) Herança

A **herança** permite que uma classe (subclasse) herde atributos e métodos de outra (superclasse), promovendo a reutilização e especialização do código.

```
public class Habit {  
  
    protected int userId;  
  
    protected LocalDateTime recordedAt;  
  
}  
  
  
public class SleepSession extends Habit {  
  
    private LocalDateTime startTime;  
  
    private LocalDateTime endTime;  
  
}
```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

No sistema, podemos definir uma superclasse `Habit` para representar os atributos comuns a todos os registo (ex.: `userId`, `recordedAt`), e subclasses especializadas para cada tipo de hábito (`WaterIntake`, `SleepSession`, `MoodEntry`, etc.).

e) Polimorfismo

O **polimorfismo** permite que diferentes classes respondam de forma distinta ao mesmo método, de acordo com o seu tipo específico.

No contexto da aplicação, um método genérico como `register()` pode comportar-se de maneira diferente em `WaterIntake` ou `SleepSession`, ajustando a operação conforme o tipo de hábito.

```
public interface Trackable {  
  
    void register();  
  
}
```

Cada classe implementa o método `register()` de forma personalizada, mantendo coerência e extensibilidade.

f) Abstração

A abstração consiste em representar apenas as características essenciais de um objeto, ignorando detalhes irrelevantes para o seu funcionamento.

No projeto, a classe abstrata `Habit` define a estrutura básica para todos os registo, deixando a implementação específica para cada tipo de hábito.

Este princípio torna o código mais claro e reduz a complexidade do sistema.

Em conjunto, estes princípios sustentam o desenho do sistema e garantem que o software seja modular, seguro e facilmente extensível. A aplicação prática destes conceitos é apresentada nas secções seguintes, através da modelação UML e da correspondência direta entre as classes Flutter e as entidades da base de dados.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

3. Modelação Conceptual e UML

3.1 Diagrama de Classes – Estrutura Global

A modelação conceptual em Programação Orientada por Objetos (POO) tem como principal objetivo representar, de forma estruturada e abstrata, os elementos centrais do sistema e as suas inter-relações.

Nesta fase, o Diagrama de Classes UML serve como base para a implementação em código, traduzindo a lógica do modelo Entidade-Relação (ER) desenvolvido na Unidade Curricular de Bases de Dados para uma perspetiva orientada a objetos.

O Diagrama de Classes do projeto reflete a estrutura modular e coerente da aplicação móvel de registo de hábitos, tendo como elemento central a classe User. Esta é a raiz das relações com todas as entidades operacionais (consumos, atividades, metas, lembretes, entre outras), garantindo a integridade lógica e o encapsulamento de dados sensíveis.

De forma geral, a arquitetura segue o padrão entidade principal → entidades dependentes, garantindo uma hierarquia clara e o isolamento funcional entre módulos. Abaixo apresenta-se uma síntese das principais classes e suas responsabilidades:

Classe	Descrição
User	Representa o utilizador autenticado. Contém dados de identificação e gere o acesso às entidades dependentes.
Profile	Armazena informações pessoais e biométricas (altura, peso, data de nascimento).
Goal	Define metas configuráveis para água, sono, atividade ou outros hábitos.
Reminder	Gera lembretes personalizados associados às metas.
WaterIntake	Regista cada consumo de água com quantidade e hora.
Meal	Representa uma refeição (pequeno-almoço, almoço, jantar).
MealItem	Relaciona alimentos (Food) com as respetivas quantidades e valores calóricos.
Food	Catálogo de alimentos, com valores nutricionais padrão.
SleepSession	Regista ciclos de sono com início, fim e qualidade.
Activity	Regista atividades físicas, incluindo duração, intensidade e calorias gastas.
MoodEntry	Regista o estado de humor e notas associadas.

POO

CycleEntry	Monitoriza o ciclo menstrual.
HabitLog	Regista e acompanha o progresso relativamente aos hábitos/vícios.
Photo	Associa imagens a refeições, perfis ou progresso.

Esta estrutura modular garante que novas classes possam ser facilmente integradas, assegurando escalabilidade e compatibilidade futura com a camada móvel.

3.2 Relações entre Classes e o Modelo de Dados (ER → Classes Flutter)

A transposição do modelo Entidade-Relação (ER) para o modelo orientado a objetos foi efetuada respeitando os princípios da normalização e a coerência semântica entre as tabelas e as classes Flutter.

Cada entidade da base de dados originou uma classe, e cada relação foi traduzida em associação, composição ou agregação no diagrama UML, conforme o tipo e a cardinalidade da ligação.

Entidade BD	Classe Flutter	Tipo de Relação	Descrição
USER	User	Principal (1:N)	Classe raiz do sistema. Um utilizador possui múltiplos registos dependentes.
PROFILE	Profile	1:0..1	Cada utilizador tem no máximo um perfil associado.
GOAL	Goal	1:N	Um utilizador pode definir várias metas.
REMINDER	Reminder	1:N	Cada utilizador pode ter múltiplos lembretes ativos.
MEAL	Meal	1:N	Um utilizador pode registrar várias refeições.
MEAL_ITEM	MeallItem	N:1	Cada item pertence a uma refeição e referencia um alimento (Food).
FOOD	Food	Catálogo global	Fornece os dados nutricionais para os Meallitem.
WATER_INTAKE	WaterIntake	1:N	Registros independentes de consumo de água.
SLEEP_SESSION	SleepSession	1:N	Cada utilizador possui múltiplos registos de sono.
ACTIVITY	Activity	1:N	Atividades físicas diárias registadas.

POO

MOOD_ENTRY	MoodEntry	1:N	Registros de humor e observações.
PHOTO	Photo	1:N	Imagens associadas a outras entidades do utilizador.

Desta forma, o modelo orientado a objetos mantém uma correspondência direta e transparente com a base de dados, facilitando a implementação da camada **DAO (Data Access Object)** e a integração com o Supabase.

3.3 Multiplicidades e Dependências

O diagrama UML adota as multiplicidades padrão (1, 0..1, 0.., 1..), definindo as regras de associação entre classes.

As principais relações são as seguintes:

- User → Profile: **1 : 0..1**
Um utilizador pode ter um único perfil biométrico.
- User → Goal, Reminder, WaterIntake, Meal, SleepSession, MoodEntry, Activity, CycleEntry: **1 : N**
O utilizador é o elemento pai que centraliza todos os registos de hábitos.
- Meal → MeallItem: **1 : N**
Uma refeição pode incluir vários itens alimentares.
- MeallItem → Food: **N : 1**
Cada item refere-se a um alimento específico do catálogo.
- User → Photo: **1 : N**
Um utilizador pode ter várias fotos associadas a registos diversos.

Esta modelação favorece o encapsulamento e a reutilização, uma vez que cada classe pode ser manipulada independentemente, respeitando o princípio de baixo acoplamento e alta coesão.

3.4 Justificação das Decisões de Modelação

As opções de modelação adotadas baseiam-se em critérios técnicos e pedagógicos que asseguram a consistência, escalabilidade e clareza do sistema:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Decisão de Design	Justificação Técnica
Separação entre Meal e MealItem	Evita redundância de dados e permite cálculos automáticos de nutrientes.
Criação de Food como classe independente	Permite reutilização e integração com APIs nutricionais externas.
Definição de Profile fora da classe User	Facilita atualizações de dados biométricos sem alterar a lógica de autenticação.
Criação de Photo como classe genérica	Permite associação polimórfica a diferentes entidades (perfil, refeições, progresso).
Implementação modular das classes de hábitos (WaterIntake, SleepSession, MoodEntry)	Garante escalabilidade e independência lógica de cada tipo de registo.
Centralização de relações em User	Simplifica a segurança e o controlo de acesso (ex.: RLS em Supabase).

Assim, o sistema foi concebido de modo a refletir as boas práticas da engenharia de software, mantendo coerência entre o modelo de dados relacional e a estrutura orientada a objetos.

A abstração de elementos comuns (como datas e identificação do utilizador) prepara o código para a reutilização em múltiplas camadas — backend, frontend e integração com o Supabase.

3.5 Figura do Diagrama UML

A representação gráfica do **Diagrama de Classes UML** (Figura 1) encontra-se em anexo a este relatório.

O diagrama apresenta as classes principais, os seus atributos e métodos essenciais, bem como as relações de associação, agregação e herança.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Figura 1 Diagrama UML do Sistema de Registo de Hábitos de Saúde e Bem-Estar.
(Fonte: Elaboração própria com base no modelo ER desenvolvido na UC de Bases de Dados.)

4. Atributos, Construtores e Métodos

4.1 Declaração e Tipos de Atributos

Os atributos são os elementos que definem o estado de um objeto e correspondem, de forma direta, às colunas das tabelas do modelo de dados.

Na Programação Orientada por Objetos (POO), cada classe possui atributos próprios, definidos com modificadores de acesso (`private`, `protected`, `public`) e tipos de dados adequados ao conteúdo que representam.

A escolha dos tipos de dados segue os princípios de coerência e eficiência, assegurando compatibilidade entre o modelo lógico (Flutter) e o modelo físico (Base de Dados Supabase).

O quadro seguinte apresenta exemplos extraídos das principais classes do sistema:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Classe	Atributo	Tipo de Dado (Flutter)	Descrição
User	id, email, password, name, isVerified, createdAt, updatedAt	int, String, String, String, boolean, DateTime, DateTime	Identificação e dados de login do utilizador.
Profile	id, sex, height_cm, weight_kg, birth_date	Int, char(1), double, double, Date	Dados biométricos associados ao utilizador.
Goal	id, key, target_value, unit, period, progress	int, String, double, String, String, double, double	Objetivos configuráveis (ex.: litros de água, horas de sono).
WaterIntake	amountML, source	int, string	Quantidade e data do registo de consumo de água.
Meal	meal_type, notes	String, String	Estrutura de refeição e lista de itens associados.
Food	id, name, kcal_per_100g, protein_g, carbs_g, fat_g	int, String, double, double, double, double	Tabela nutricional de alimentos.
SleepSession	start_time, end_time, quality	DateTime, DateTime, int	Dados relativos a cada período de sono.
Reminder	id, type, time_spec, enabled, repeat_rule	Int, String, string, boolean, string	Mensagem e hora de ativação do lembrete.

Os tipos **primitivos** (int, double, boolean) e **objetos compostos** (String, DateTime) foram utilizados de acordo com a natureza dos dados. Esta combinação permite maior precisão no controlo de valores e maior flexibilidade na manipulação dos registo dentro da aplicação.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

4.2 Construtores e Inicialização de Objetos

Os **construtores** são métodos especiais invocados no momento em que um objeto é criado. A sua função é inicializar os atributos com valores predefinidos, garantindo que o objeto comece o seu ciclo de vida num estado válido e consistente.

Em Flutter, o construtor tem o mesmo nome da classe e **não possui tipo de retorno**. Podem existir múltiplos construtores (sobrecarga) para permitir diferentes formas de inicialização.

Exemplo prático da classe WaterIntake:

```

public class WaterIntake {

    private double amountML;

    private LocalDateTime recordedAt;

    // Construtor padrão

    public WaterIntake() {

        this.amountML = 0;

        this.recordedAt = LocalDateTime.now();

    }

    // Construtor parametrizado

    public WaterIntake(double amountML, LocalDateTime recordedAt) {

        this.amountML = amountML;

        this.recordedAt = recordedAt;

    }

}

```

Este exemplo demonstra como a sobrecarga de construtores permite flexibilidade na criação de objetos:

- O construtor padrão cria um registo com valores iniciais automáticos;
- O construtor parametrizado permite que o utilizador defina explicitamente a quantidade e a hora do registo.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

4.3 Métodos e Operações do Sistema

Os **métodos** representam os comportamentos de um objeto, ou seja, as ações que este pode executar.

São implementados para **ler, modificar, calcular ou comunicar** informações entre classes, mantendo o princípio de encapsulamento.

Os métodos dividem-se em três categorias principais:

1. **Métodos de acesso (getters/setters):** permitem a leitura e alteração controlada de atributos.
2. **Métodos de negócio:** executam operações relacionadas com a lógica funcional (ex.: cálculo de médias, validação de dados).
3. **Métodos utilitários:** realizam tarefas complementares, como formatação ou exportação de dados.

Exemplo simplificado da classe Meal:

```
public class Meal {  
  
    private List<MealItem> mealItems = new ArrayList<>();  
  
    public void addMealItem(MealItem item) {  
        mealItems.add(item);  
    }  
  
    public double calculateTotalCalories() {  
        double total = 0;  
        for (MealItem item : mealItems) {  
            total += item.getCalories();  
        }  
        return total;  
    }  
}
```

Neste caso:

- O método `addMealItem()` adiciona um novo alimento à refeição.
- O método `calculateTotalCalories()` percorre a lista e soma o valor energético total, demonstrando a aplicação de ciclos (`for`) e estruturas dinâmicas (`ArrayList`).

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

4.4 Exemplo Prático de Implementação em Flutter

```

public class Goal {

    private String type;

    private double targetValue;

    private String unit;

    private double progress;

    // Construtor

    public Goal(String type, double targetValue, String unit) {

        this.type = type;

        this.targetValue = targetValue;

        this.unit = unit;

        this.progress = 0;

    }

    // Método para atualizar o progresso

    public void updateProgress(double value) {

        this.progress += value;

    }

    // Método para verificar se a meta foi atingida

    public boolean isGoalReached() {

        return progress >= targetValue;

    }

    // Getters e Setters

    public double getProgress() { return progress; }

    public double getTargetValue() { return targetValue; }

}

```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Para ilustrar a integração entre construtores, atributos e métodos, apresenta-se a seguir um exemplo mais completo com base na classe `Goal`, que representa as metas de saúde do utilizador:

Neste exemplo, a classe `Goal`:

- Define atributos essenciais para caracterizar uma meta;
- Possui um construtor parametrizado para inicialização;
- Inclui métodos de negócio (`updateProgress()` e `isGoalReached()`) que representam comportamentos reais do sistema;
- Implementa o princípio do encapsulamento, evitando o acesso direto a atributos sensíveis como `progress`.

A combinação adequada entre atributos, construtores e métodos garante que cada classe funcione como uma unidade lógica independente, reforçando os princípios da modularidade e reutilização.

Esta estrutura torna o código mais legível, mais fácil de manter e perfeitamente integrável com a camada de persistência em Supabase.

5. Coleções, Arrays e Estruturas de Dados Dinâmicas

5.1 Uso de Arrays e Listas Dinâmicas

Em Programação Orientada por Objetos, os **Arrays** e as **Coleções** são estruturas fundamentais para armazenar e manipular conjuntos de dados relacionados.

Um **Array** é uma estrutura de tamanho fixo, ideal para conjuntos de dados estáticos, enquanto as **Coleções** (como `ArrayList`, `HashSet`, `LinkedList`) são estruturas dinâmicas que permitem adicionar, remover ou reorganizar elementos em tempo de execução.

No projeto de registo de hábitos de saúde e bem-estar, as listas dinâmicas são amplamente utilizadas para representar grupos de objetos criados a partir das classes principais, como:

- Lista de **consumos de água** (`List<WaterIntake>`);
- Lista de **refeições e alimentos** (`List<MealItem>`);
- Lista de **lembretes ativos** (`List<Reminder>`);

POO

- Lista de **sessões de sono** (List<SleepSession>).

A utilização de listas (ArrayList) proporciona flexibilidade para armazenar quantos registos o utilizador desejar, sem necessidade de definir previamente o seu tamanho.

Exemplo prático da classe User:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

```

public class User {

    private int userId;

    private String name;

    private List<WaterIntake> waterIntakes = new ArrayList<>();

    private List<Meal> meals = new ArrayList<>();

    // Adiciona um novo registo de consumo de água

    public void addWaterIntake(WaterIntake intake) {

        waterIntakes.add(intake);

    }

    // Lista todos os registas de consumo de água

    public void listWaterIntakes() {

        for (WaterIntake w : waterIntakes) {

            System.out.println("Data: " + w.getRecordedAt() + " | Quantidade: " +
w.getAmountML() + " ml");

        }

    }

}
  
```

Neste exemplo, o `ArrayList` é utilizado para armazenar dinamicamente todos os registos de consumo de água do utilizador.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Através dos métodos `addWaterIntake()` e `listWaterIntakes()`, é possível adicionar e visualizar os dados em tempo real, demonstrando a **interatividade e dinamismo** da aplicação.

5.2 Ciclos de Repetição e Controlo de Fluxo

Os **ciclos de repetição** (loops) e as **estruturas de controlo de fluxo** são indispensáveis para a execução lógica de operações repetitivas e tomadas de decisão.

Em Flutter, os principais tipos de ciclos utilizados são `for`, `while` e `foreach`, sendo escolhidos conforme o contexto e a complexidade da operação.

No contexto do projeto, estas estruturas permitem:

- Percorrer listas de registo (ArrayList);
- Calcular médias e totais (ex.: consumo de água, calorias ingeridas);
- Filtrar informações específicas (ex.: registo de um determinado dia);
- Avaliar condições (ex.: meta atingida ou não).

Exemplo de controlo de fluxo aplicado à classe Goal:

```

public void checkGoalStatus(List<WaterIntake> waterIntakes) {

    double total = 0;

    for (WaterIntake w : waterIntakes) {

        total += w.getAmountML();

    }

    if (total >= targetValue) {

        System.out.println("Meta diária atingida!");

    } else {

        System.out.println("Progresso atual: " + total + " ml / " + targetValue + " ml");

    }

}
  
```

Este método percorre a lista de registos de água e utiliza uma estrutura condicional (if/else) para verificar se a meta foi atingida, representando a aplicação prática dos conceitos de **ciclo** e **decisão lógica**.

5.3 Aplicação em Registos de Hábitos

A manipulação de coleções dinâmicas é particularmente útil na gestão dos **registos diários de hábitos**, pois estes variam de acordo com o comportamento do utilizador. Cada tipo de registo (água, sono, humor, alimentação) é armazenado em listas independentes, permitindo análises personalizadas e geração de relatórios.

Exemplo integrado:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

```

public class HabitTracker {

    private List<WaterIntake> waterList = new ArrayList<>();

    private List<SleepSession> sleepList = new ArrayList<>();

    public void addWater(double amount) {

        waterList.add(new WaterIntake(amount, LocalDateTime.now()));

    }

    public void addSleep(LocalDateTime start, LocalDateTime end) {

        sleepList.add(new SleepSession(start, end));

    }

    public void dailySummary() {

        System.out.println("Resumo diário de hábitos:");

        System.out.println("- Registros de água: " + waterList.size());

        System.out.println("- Sessões de sono: " + sleepList.size());

    }

}

```

Este exemplo demonstra a modularidade da estrutura orientada a objetos:

- Cada tipo de hábito é tratado como uma **coleção independente**;
- Os métodos são simples e diretos, mantendo o princípio de **alta coesão**;
- A classe HabitTracker funciona como um agregador central de registros, facilitando futuras integrações com o **frontend Flutter** e o **Supabase**.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

5.4 Benefícios do Uso de Estruturas Dinâmicas

O uso de coleções dinâmicas neste projeto apresenta múltiplas vantagens:

- **Eficiência:** permite armazenar e processar grandes volumes de dados sem limitação de tamanho pré-definido;
- **Escalabilidade:** facilita a introdução de novos tipos de hábitos ou registo sem alterações na estrutura base;
- **Simplicidade:** reduz a complexidade do código, tornando-o mais legível e intuitivo;
- **Integração direta com APIs e bases de dados:** a lista pode ser facilmente percorrida e convertida em formato JSON para comunicação com o Supabase.

Em suma, a adoção de listas e coleções dinâmicas assegura que o sistema seja flexível e evolutivo, acompanhando o crescimento natural da aplicação e a diversidade de hábitos registados pelos utilizadores.

6. Encapsulamento, Herança e Polimorfismo

6.1 Encapsulamento e Modificadores de Acesso

O encapsulamento é um dos princípios centrais da POO e refere-se à prática de proteger os dados internos de uma classe, permitindo o acesso apenas através de métodos controlados.

Em vez de expor diretamente os atributos, são utilizados métodos públicos de leitura e escrita (getters e setters) para manter a integridade e a segurança da informação.

No contexto do sistema, este conceito é fundamental, pois os dados tratados nomeadamente perfis de utilizador, dados biométricos e registo de saúde exigem confidencialidade e validação rigorosa antes de serem modificados.

Exemplo prático da classe Profile:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

```

public class Profile {

    private double height;

    private double weight;

    private LocalDate birthDate;

    // Encapsulamento através de métodos de acesso

    public double getHeight() { return height; }

    public void setHeight(double height) {

        if (height > 0) this.height = height;

    }

    public double getWeight() { return weight; }

    public void setWeight(double weight) {

        if (weight > 0) this.weight = weight;

    }

}

```

Neste exemplo:

- Os atributos são privados (`private`), impedindo o acesso direto;
- Os métodos `setHeight()` e `setWeight()` incluem validação, evitando valores incorretos;
- A integridade dos dados é garantida a cada operação de escrita.

Desta forma, o encapsulamento contribui para a segurança, legibilidade e manutenibilidade do código, permitindo que as alterações internas de uma classe não afetem o funcionamento das demais.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

6.2 Herança Aplicada ao Sistema de Hábitos

A herança é o mecanismo que permite que uma classe (subclasse) herde atributos e métodos de outra (superclasse), possibilitando reutilização de código e organização hierárquica.

No projeto, a herança foi aplicada para estruturar o conjunto de classes que representam os diversos hábitos de saúde registados pelo utilizador.

Foi criada uma superclasse genérica chamada `Habit`, da qual derivam as classes específicas (`WaterIntake`, `SleepSession`, `MoodEntry`, `Activity`, entre outras).

```
// Superclasse genérica

public class Habit {

    protected int userId;

    protected LocalDateTime recordedAt;

    public Habit(int userId) {

        this.userId = userId;

        this.recordedAt = LocalDateTime.now();

    }

    public LocalDateTime getRecordedAt() { return recordedAt; }

}
```

Exemplo de hierarquia:

Subclasses especializadas:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

```

public class WaterIntake extends Habit {

    private double amountMI;

    public WaterIntake(int userId, double amountMI) {

        super(userId);

        this.amountMI = amountMI;
    }
}

public class SleepSession extends Habit {

    private LocalDateTime startTime;

    private LocalDateTime endTime;

    public SleepSession(int userId, LocalDateTime startTime, LocalDateTime endTime) {

        super(userId);

        this.startTime = startTime;

        this.endTime = endTime;
    }
}
  
```

Nesta estrutura:

- A classe `Habit` centraliza os atributos comuns a todos os tipos de registo (ex.: `userId` e `recordedAt`);
- As subclasses especializam-se nos dados e métodos específicos de cada hábito;
- O uso do comando `super()` permite reutilizar o construtor da classe-pai, evitando redundância.

A herança torna o sistema mais limpo, modular e extensível, permitindo adicionar novos tipos de hábitos sem modificar a estrutura existente.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

6.3 Polimorfismo e Interfaces Comuns

O polimorfismo é o princípio que permite que métodos com o mesmo nome se comportem de forma diferente, conforme o tipo do objeto que os executa.

Este conceito é essencial quando se pretende que várias classes possam executar a mesma operação de modo independente, sem necessidade de duplicar código.

No projeto, o polimorfismo foi implementado através de uma interface genérica chamada Trackable, que define um conjunto de métodos comuns a todos os tipos de hábitos.

```
public interface Trackable {  
  
    void register();  
  
    String summary();  
  
}
```

Cada classe concreta (WaterIntake, SleepSession, MoodEntry, etc.) implementa a interface e fornece a sua própria versão dos métodos definidos.

Exemplo:

```

public class WaterIntake extends Habit implements Trackable {

  private double amountMl;

  public WaterIntake(int userId, double amountMl) {
    super(userId);
    this.amountMl = amountMl;
  }

  @Override
  public void register() {
    System.out.println("Registrado " + amountMl + " ml de água.");
  }

  @Override
  public String summary() {
    return "Consumo de água: " + amountMl + " ml (" + getRecordedAt() + ")";
  }
}

public class SleepSession extends Habit implements Trackable {

  private LocalDateTime startTime;
  private LocalDateTime endTime;

  public SleepSession(int userId, LocalDateTime startTime, LocalDateTime endTime) {
    super(userId);
    this.startTime = startTime;
    this.endTime = endTime;
  }

  @Override
  public void register() {
    System.out.println("Sessão de sono iniciada às " + startTime);
  }

  @Override
  public String summary() {
    return "Sono de " + startTime + " a " + endTime;
  }
}
  
```

Aqui, o método `register()` tem comportamentos distintos consoante a classe que o implementa, mas mantém uma **interface comum**.

O polimorfismo permite que o sistema execute operações genéricas sem precisar de saber o tipo exato de objeto.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Exemplo de aplicação prática:

```

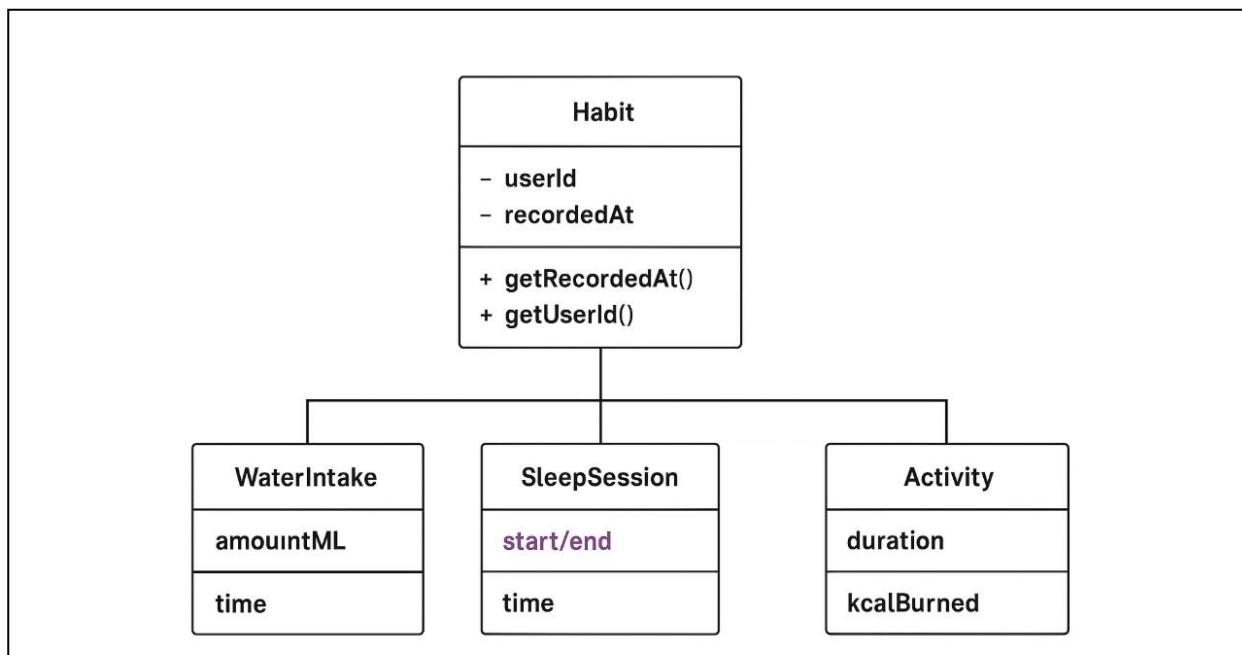
public void listAllHabits(List<Trackable> habits) {

    for (Trackable h : habits) {
        System.out.println(h.summary());
    }
}
    
```

Esta abordagem simplifica o código e aumenta a reutilização e abstração, já que a mesma função pode lidar com múltiplos tipos de hábitos de forma uniforme.

6.4 Exemplo de Hierarquia e Extensibilidade

A hierarquia completa do projeto pode ser resumida da seguinte forma:



Esta estrutura demonstra como:

- A superclasse **Habit** fornece uma base comum de atributos e métodos;
- As subclasses expandem a funcionalidade conforme o tipo de hábito;
- As interfaces (**Trackable**, **Analyzable**, etc.) permitem extensões futuras, como relatórios estatísticos ou sincronização com APIs externas.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

6.5 Benefícios da Aplicação dos Três Princípios

A integração equilibrada entre encapsulamento, herança e polimorfismo proporciona diversas vantagens ao projeto:

Princípio	Benefício Concreto no Sistema
Encapsulamento	Protege dados sensíveis e evita inconsistências. Garante validação e acesso controlado.
Herança	Reutiliza código e mantém uma estrutura lógica e hierárquica entre classes.
Polimorfismo	Permite tratamento genérico e flexível de diferentes tipos de hábitos, facilitando a expansão da aplicação.

Estes três princípios tornam o código mais organizado, reutilizável e preparado para evolução futura, pilares fundamentais da engenharia de software moderna.

7. Padrões de Design e Boas Práticas

7.1 Princípios SOLID

O conjunto de princípios **SOLID** define diretrizes fundamentais para o desenvolvimento de sistemas orientados a objetos, promovendo a legibilidade, escalabilidade e manutenção do código.

No contexto deste projeto, estes princípios foram aplicados de forma integrada, assegurando que cada componente cumpre uma função clara e independente.

Princípio	Descrição	Aplicação no Projeto
S – Single Responsibility Principle	Cada classe deve ter uma única responsabilidade.	As classes WaterIntake, Meal, SleepSession e Goal são independentes e focadas numa única tarefa: gerir registos específicos de hábitos.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

O – Open/Closed Principle	O código deve estar aberto à extensão, mas fechado à modificação.	Novos tipos de hábitos (ex.: MoodEntry) podem ser adicionados sem alterar as classes existentes, apenas estendendo Habit ou implementando Trackable.
L – Liskov Substitution Principle	Subclasses devem poder substituir as superclasses sem afetar o comportamento.	Todas as subclasses de Habit respeitam a estrutura e o comportamento da classe-pai, garantindo compatibilidade nas chamadas de métodos genéricos.
I – Interface Segregation Principle	Evita interfaces demasiado amplas; cada uma deve ser específica.	Interfaces como Trackable (para registo) e Analyzable (para estatísticas) foram criadas com responsabilidades separadas.
D – Dependency Inversion Principle	As classes de alto nível não devem depender de implementações concretas, mas de abstrações.	A comunicação entre a lógica Flutter e a base de dados é feita através de uma camada DAO, permitindo substituição do backend (Supabase) sem alterar o código principal.

Estes princípios asseguram que o projeto é **robusto, modular e preparado para evoluções futuras**, reduzindo o acoplamento entre classes e simplificando a integração com novas tecnologias.

7.2 Padrões Utilizados (Singleton, Factory, Observer)

Os **padrões de design** são soluções comprovadas para problemas recorrentes em arquitetura de software. No sistema desenvolvido, foram aplicados três dos padrões mais relevantes: **Singleton**, **Factory Method** e **Observer**, adaptados ao contexto da aplicação móvel e à integração com a base de dados.

a) Singleton Gestão Centralizada de Conexão à Base de Dados

O padrão **Singleton** garante que exista apenas **uma instância de uma determinada classe** durante toda a execução do programa.

No projeto, este padrão é utilizado para a gestão da **conexão com a base de dados Supabase**, evitando a criação desnecessária de múltiplas ligações.

POO

Exemplo:

```

public class DatabaseConnection {

  private static DatabaseConnection instance;

  private DatabaseConnection() {
    // Código de inicialização da ligação
    System.out.println("Ligaçāo ao Supabase inicializada.");
  }

  public static DatabaseConnection getInstance() {
    if (instance == null) {
      instance = new DatabaseConnection();
    }
    return instance;
  }
}
  
```

Com esta implementação, qualquer classe que necessite aceder ao Supabase invoca:

```
DatabaseConnection db = DatabaseConnection.getInstance();
```

Assim, garante-se eficiência e segurança, evitando múltiplas instâncias que poderiam comprometer o desempenho do sistema.

b) Factory Method Criação de Objetos de Forma Genérica

O padrão **Factory Method** permite **instanciar objetos de forma controlada**, centralizando a lógica de criação numa classe responsável por decidir qual tipo de objeto deve ser gerado.

No projeto, este padrão é utilizado para criar dinamicamente objetos das classes que herdam de Habit, de acordo com o tipo de registo que o utilizador deseja adicionar.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Exemplo:

```
public class HabitFactory {  
  
    public static Habit createHabit(String type, int userId, double value) {  
  
        switch (type) {  
  
            case "water":  
  
                return new WaterIntake(userId, value);  
  
            case "activity":  
  
                return new Activity(userId, value);  
  
            default:  
  
                throw new IllegalArgumentException("Tipo de hábito inválido: " + type);  
  
        }  
    }  
}
```

Uso prático:

```
Habit novoHabit = HabitFactory.createHabit("water", 1, 500);
```

Este padrão simplifica a criação de objetos e facilita a extensão futura do sistema, bastando adicionar novas subclasses sem modificar o código existente.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

c) Observer Notificações e Lembretes

O padrão Observer define uma relação de publicação-subscrição, onde um objeto (*sujeito*) notifica automaticamente outros objetos (*observadores*) sobre alterações de estado.

Na aplicação, este padrão foi aplicado na implementação de lembretes e notificações, garantindo que as metas e eventos de hábitos disparam alertas automáticos.

```

public interface Observer {
  void update(String message);
}

public class Reminder implements Observer {
  @Override
  public void update(String message) {
    System.out.println("⚠️ Lembrete: " + message);
  }
}

public class GoalTracker {
  private List<Observer> observers = new ArrayList<>();
  public void addObserver(Observer observer) {
    observers.add(observer);
  }
  public void notifyObservers(String message) {
    for (Observer o : observers) {
      o.update(message);
    }
  }
  public void checkGoal(double progress, double target) {
    if (progress >= target) {
      notifyObservers("Meta atingida!");
    }
  }
}

```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Esta abordagem torna o sistema **reativo e orientado a eventos**, ideal para integração com notificações móveis via Flutter.

7.3 Modularidade e Reutilização

A aplicação dos padrões acima permitiu estruturar o sistema segundo princípios de **modularidade, reutilização e independência funcional**.

Cada componente cumpre uma função específica e comunica com os restantes através de interfaces e métodos públicos, respeitando o encapsulamento.

Módulo	Função Principal	Padrão Aplicado
DatabaseConnection	Gerir a ligação com o Supabase.	Singleton
HabitFactory	Criar objetos das subclasses de Habit.	Factory Method
GoalTracker / Reminder	Notificar alterações e metas atingidas.	Observer
Habit + Subclasses	Estrutura modular dos hábitos.	Herança e Polimorfismo
DAO	Interagir com a base de dados.	Abstração e Dependency Inversion

Esta abordagem garante que cada módulo pode ser testado, substituído ou atualizado sem impacto direto nos restantes, cumprindo as boas práticas de **baixo acoplamento e alta coesão**.

7.4 Boas Práticas Adotadas

Durante o desenvolvimento da camada lógica do projeto, serão seguidas boas práticas de programação que contribuem para a qualidade e sustentabilidade do código:

- **Nomenclatura clara e sem ambiguidade**, seguindo convenções Flutter (camelCase para métodos e atributos, PascalCase para classes).
- **Comentário de código estruturado**, explicando o propósito de cada método e parâmetros relevantes.
- **Revisão e refatoração contínua**, garantindo simplicidade e legibilidade.
- **Validação de dados na origem**, antes da inserção em base de dados.

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

- **Separação de camadas**, respeitando a arquitetura MVC (Model-View-Controller).
- **Reutilização de classes utilitárias**, evitando duplicação de código.
- **Gestão de exceções**, prevenindo falhas inesperadas e garantindo robustez.

Estas práticas refletem um desenvolvimento profissional, orientado à manutenibilidade e alinhado com os padrões de engenharia de software recomendados.

A aplicação conjunta dos **princípios SOLID** e dos **padrões de design** reforça a solidez e a escalabilidade da aplicação, preparando-a para futuras integrações com novas funcionalidades e serviços externos.

8. Integração com a Base de Dados e Flutter

8.1 Camada DAO (Data Access Object)

A **camada DAO** é o elo de ligação entre o modelo orientado a objetos e a base de dados. O padrão **Data Access Object** permite isolar a lógica de acesso aos dados do restante código da aplicação, garantindo que as classes de negócio (User, WaterIntake, Goal, etc.) não precisam lidar diretamente com comandos SQL ou conexões de rede.

Cada entidade relevante da base de dados possui um **DAO correspondente**, responsável pelas operações CRUD (Create, Read, Update, Delete).

Exemplo de implementação simplificada do DAO para a classe WaterIntake:

```

public class WaterIntakeDAO {

    private DatabaseConnection db = DatabaseConnection.getInstance();

    // Inserir novo registo

    public void insert(WaterIntake water) {

        String query = "INSERT INTO water_intake (user_id, amount_ml, recorded_at) VALUES (?, ?, ?)";

        db.execute(query, water.getUserId(), water.getAmountML(), water.getRecordedAt());

    }

    // Listar registos

    public List<WaterIntake> getAll(int userId) {

        String query = "SELECT * FROM water_intake WHERE user_id = ?";

        ResultSet rs = db.query(query, userId);

        List<WaterIntake> list = new ArrayList<>();

        while (rs.next()) {

            list.add(new WaterIntake(
                rs.getInt("user_id"),
                rs.getDouble("amount_ml"),
                rs.getTimestamp("recorded_at").toLocalDateTime()
            ));

        }

        return list;
    }
}

```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Esta estrutura apresenta três benefícios principais:

1. **Isolamento da lógica de persistência** as classes de negócio não precisam conhecer a estrutura da base de dados;
2. **Reutilização** o mesmo DAO pode ser chamado por diferentes partes da aplicação;
3. **Escalabilidade** facilita a substituição do backend (por exemplo, Supabase → Firebase) sem alterar a lógica Flutter.

8.2 Integração com Supabase

O **Supabase** é o serviço utilizado como backend do projeto, fornecendo uma base de dados PostgreSQL, autenticação e API REST nativa.

A integração com o modelo POO é feita através de **requisições HTTP** ou de SDKs específicos que convertem as classes Flutter em formatos compatíveis com a comunicação web (JSON).

Cada objeto do sistema (ex.: User, Meal, Goal) pode ser serializado em JSON antes de ser enviado à base de dados, ou desserializado no momento da leitura.

Exemplo conceitual de integração:

```

import org.json.JSONObject;

public class SupabaseAPI {

    public static void sendWaterIntake(WaterIntake water) {

        JSONObject json = new JSONObject();

        json.put("user_id", water.getUserId());

        json.put("amount_ml", water.getAmountML());

        json.put("recorded_at", water.getRecordedAt().toString());

        HttpClient.post("https://supabase.io/api/water_intake", json);

    }

}

```

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

Com esta abordagem:

- A camada de lógica (WaterIntake, Goal, Meal) comunica-se diretamente com o **endpoint REST do Supabase**;
- As classes DAO funcionam como intermediárias entre o código Flutter e as tabelas PostgreSQL;
- A estrutura orientada a objetos mantém coerência total com o modelo relacional previamente definido no relatório de *Bases de Dados*.

A integração é segura, modular e escalável, permitindo que novas tabelas ou funcionalidades sejam incorporadas com facilidade.

8.3 Comunicação entre Camadas (MVC)

A arquitetura da aplicação segue o padrão **MVC (Model–View–Controller)**, uma das práticas mais consolidadas na engenharia de software moderna.

Este padrão divide o sistema em três camadas principais:

Camada	Descrição	Exemplo no Projeto
Model	Contém a lógica de negócio e a estrutura de dados. Representa as classes e objetos da POO.	User, WaterIntake, Meal, Goal, SleepSession, DAO, HabitFactory
View	Responsável pela interface gráfica e interação com o utilizador.	Ecrãs e componentes desenvolvidos em Flutter (ex.: WaterScreen, GoalsPage)
Controller	Faz a mediação entre o modelo e a vista. Recebe eventos do utilizador, processa-os e atualiza a interface.	HabitController, UserController, GoalTracker

A **camada Controller** é responsável por interpretar as ações do utilizador na interface Flutter, converter essas ações em chamadas às classes Flutter correspondentes e atualizar o ecrã conforme o resultado.

Fluxo geral de funcionamento:

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

[Flutter UI] → [Controller] → [Model (Flutter)] → [DAO / Supabase] → [Resposta ao UI]

Este fluxo assegura **independência entre as camadas**, permitindo que a interface ou o backend possam ser alterados sem comprometer a lógica principal do sistema.

8.4 Integração com o Flutter (Frontend)

Na Unidade Curricular de *Programação de Dispositivos Móveis*, a aplicação será desenvolvida em **Flutter**, utilizando **Dart** como linguagem base.

A camada móvel comunica com o backend em Supabase e com a lógica POO através de chamadas HTTP e pacotes de sincronização em JSON.

Cada ecrã (widget Flutter) representa uma vista específica da aplicação:

- **Ecrã de login:** autenticação de utilizadores e verificação de tokens.
- **Ecrã de registo de água:** comunicação com o endpoint /water_intake e atualização da lista em tempo real.
- **Ecrã de metas:** leitura da tabela goals e cálculo de progresso diário.
- **Dashboard:** consolidação de dados de múltiplas classes (WaterIntake, Meal, SleepSession) em gráficos interativos.

A integração Flutter ↔ Supabase garante:

- **Coerência entre dados e interface** (o mesmo modelo de objetos é refletido visualmente);
- **Atualização em tempo real** dos hábitos registados;
- **Extensibilidade** (qualquer nova entidade criada no Flutter pode ser exposta via API e consumida pelo Flutter).

POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

8.5 Benefícios da Integração Multicamadas

A abordagem integrada entre POO, Supabase e Flutter traz um conjunto de benefícios técnicos e funcionais que reforçam a qualidade do sistema:

Benefício	Descrição
Escalabilidade	Novas funcionalidades podem ser adicionadas sem comprometer as camadas existentes.
Manutenibilidade	Alterações na interface ou na base de dados não exigem reestruturação da lógica principal.
Reutilização	O mesmo modelo de objetos pode ser usado em múltiplas plataformas (web, mobile, desktop).
Segurança	O acesso à base de dados é controlado e abstraído através da camada DAO.
Coerência de dados	Todos os módulos seguem a mesma estrutura de classes e relações, garantindo consistência.

9. Conclusão

9.1 Síntese dos Resultados

O desenvolvimento do presente relatório, centrado na Programação Orientada por Objetos (POO), consolida o trabalho anteriormente realizado na Unidade Curricular de Bases de Dados, permitindo transformar o modelo relacional em uma arquitetura lógica e funcional, baseada em classes, objetos e métodos.

Com a implementação do paradigma orientado a objetos, o sistema de registo de hábitos de saúde e bem-estar evoluiu de uma simples estrutura de dados para um conjunto coeso, modular e escalável de componentes de software.

Cada entidade do modelo de dados foi convertida numa classe Flutter independente, respeitando os princípios de encapsulamento, herança e polimorfismo.

A estrutura resultante mostrou-se eficaz para representar o comportamento real de cada módulo da aplicação desde o registo de água e refeições, até ao controlo de metas e lembretes automáticos.

Com o apoio de padrões de design como Singleton, Factory Method e Observer, o sistema irá atingir um nível de organização e reutilização de código compatível com boas práticas profissionais.

A articulação entre a camada de lógica POO, a base de dados Supabase e a interface móvel Flutter assegurará uma comunicação fluida e uma gestão de dados eficiente, demonstrando a importância do trabalho interdisciplinar entre as três unidades curriculares.

O resultado é uma aplicação coerente, funcional e tecnicamente bem estruturada, capaz de evoluir para novas funcionalidades sem comprometer a estabilidade do sistema.

9.2 Considerações Finais

A elaboração deste relatório permitiu compreender, de forma prática e aplicada, a importância da **Programação Orientada por Objetos** como base da arquitetura de software moderna.

Ao transformar um modelo de dados em código estruturado e modular, será possível criar uma ponte sólida entre teoria e prática, refletindo a integração entre o pensamento analítico e a execução técnica.

Mais do que um exercício académico, este trabalho representa uma síntese realista do ciclo de desenvolvimento de software, em que o programador é capaz de conceber, modelar, implementar e integrar um sistema funcional, respeitando os princípios de clareza, reutilização e escalabilidade.

Assim, este projeto reforça não apenas o domínio técnico, mas também a visão global necessária para o desenvolvimento de soluções digitais completas e sustentáveis.

POO

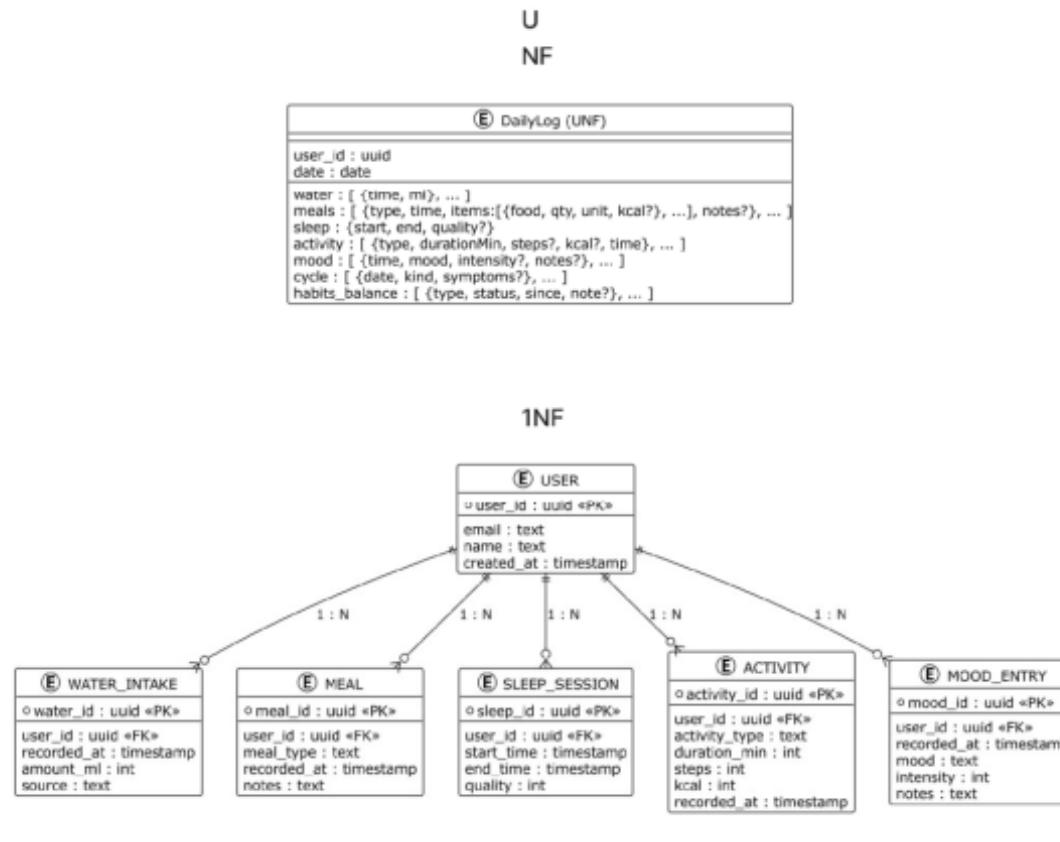
PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

10. Referências Bibliográficas

- Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, R. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Larman, C. (2004). *Applying UML and patterns* (3rd ed.). Prentice Hall.
- Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.
- Google. (2024). *Flutter documentation*. <https://docs.flutter.dev>
- Supabase. (2024). *Supabase documentation*. <https://supabase.com/docs>
- PostgreSQL Global Development Group. (2024). *PostgreSQL documentation*. <https://www.postgresql.org/docs/>
- Grupo 11. (2025). *Documentação interna do projeto – ER, UML e tabelas Supabase* [Documento interno]. Universidade Europeia.

11. Anexos

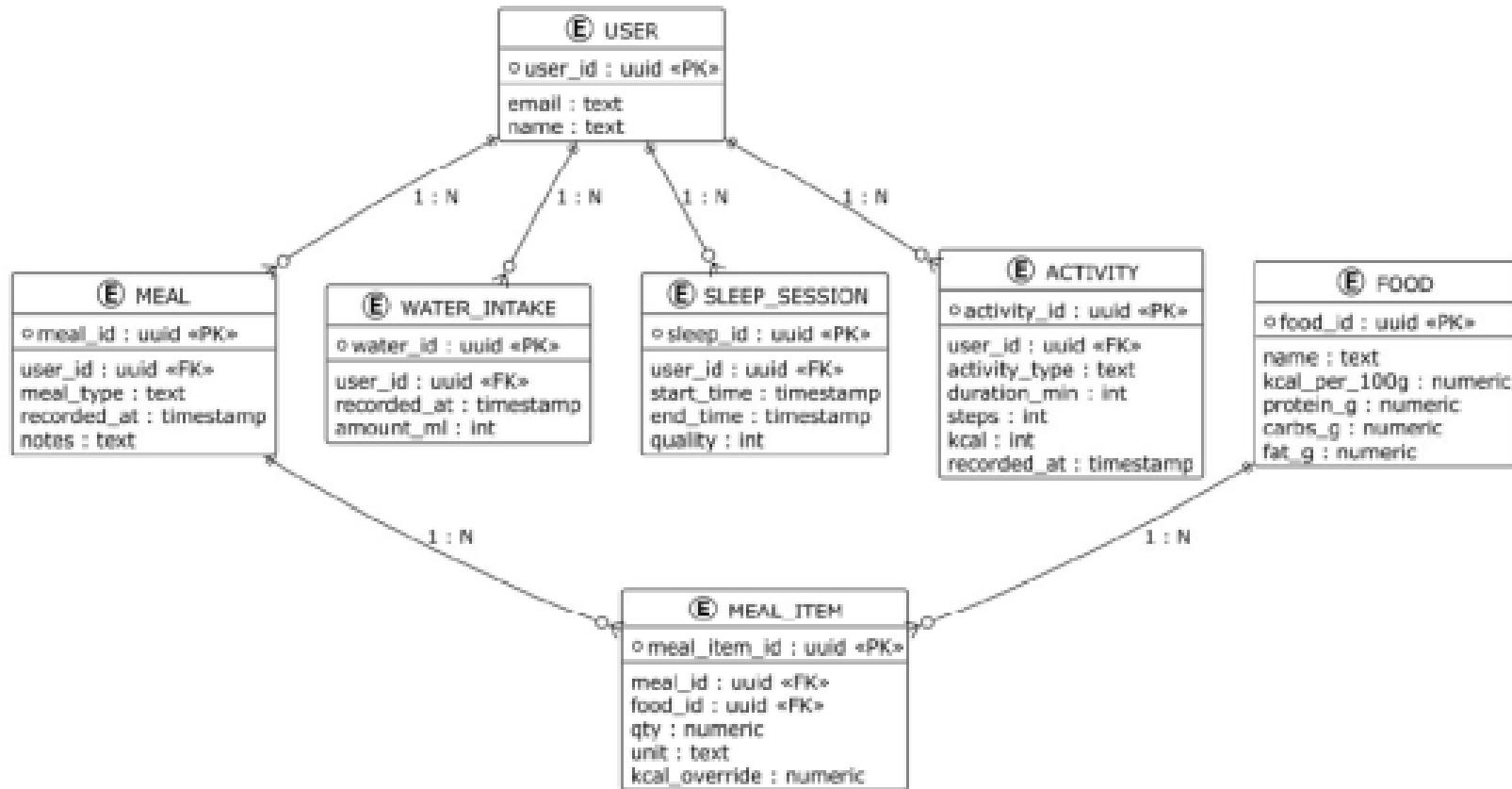
11.1 Diagrama Entidade-Relacionamento (ER)



POO

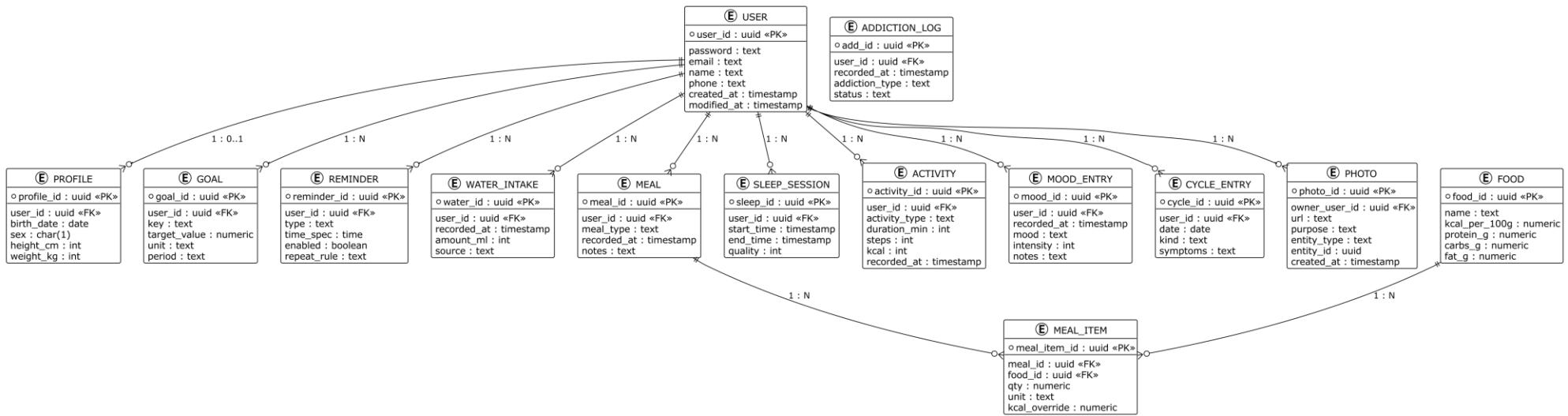
PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

2NF



POO

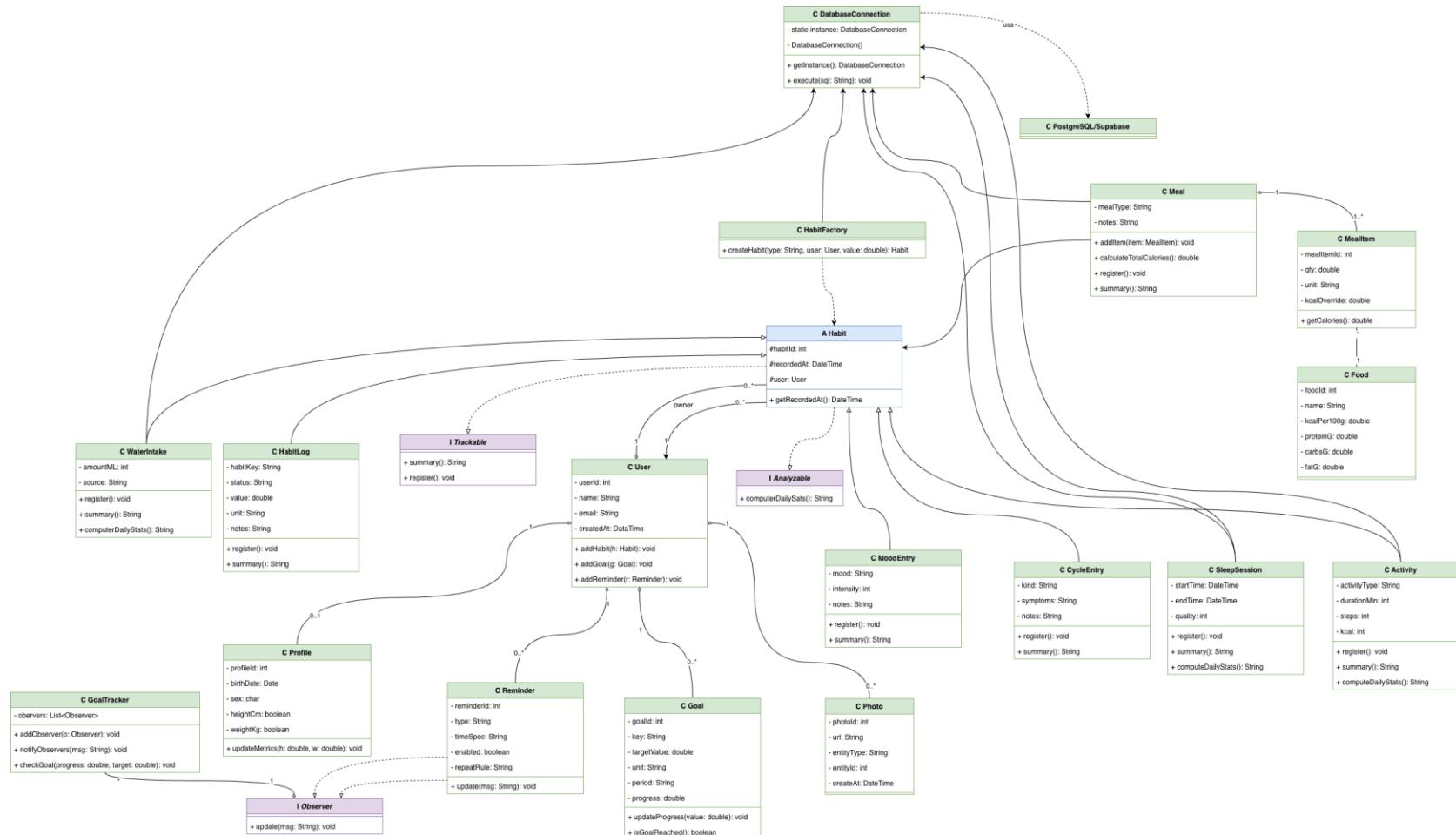
PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11



POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11

11.2 Diagrama de Classes UML



POO

PBL – Aplicação Móvel para Registo de Hábitos de Saúde e Bem-estar | Grupo 11