IPV-Instituto Politécnico de Viseu

ESTGV-Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Projeto Biblioteca

Engenharia de Software I

Trabalho realizador por:

David Almeida Costa nº16804

Diana Ferreira Lourenço do Souto nº16811

Gonçalo Filipe dos Santos Neves nº16812

Supervisores/Orientadores:

Carlos Cunha

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Licenciatura de Engenharia Informática

Projeto Biblioteca

Engenharia de Software I

Ano Letivo:

2018/2019

Período de realização do Projeto:

18/02/2019 – 20/06/2019

Trabalho realizador por:

David Almeida Costa nº16804

Diana Ferreira Lourenço do Souto nº16811

Gonçalo Filipe dos Santos Neves nº16812

Supervisores/Orientadores:

Carlos Cunha

Índice:

Índice de Figuras:

1.Introdução

Este relatório foi realizado no âmbito da Unidade Curricular de Engenharia de Software I, lecionada durante a Licenciatura em Engenharia Informática na Escola Superior de Tecnologias e Gestão de Viseu, pertencente ao Instituto Politécnico de Viseu.

Este projeto teve como base a gestão de uma biblioteca, que tem como objetivo o empréstimo de livros a utilizadores. Estando fixados limites do período de empréstimo e do número de livros que podem ser emprestados em simultâneo, o utilizador, caso não cumpra estas regras (verificadas, periodicamente, pelo funcionário da biblioteca), estava sujeito a pagar uma coima ou até ser cancelado o seu registo.

Os utilizadores, também, tinham a possibilidade de apresentar propostas de aquisição de livros à biblioteca. A biblioteca para proceder à compra de livros (novos ou outras cópias), o responsável da biblioteca tinha de apresentar uma requisição de compra à direção da biblioteca, acompanhada por um ofício que fundamenta a necessidade da compra.

Nesta base do projeto, criou-se uma classe *Repositorio* para gerir a biblioteca e uma interface, denominada *Repositorio_MEM*, com a finalidade de resolução do desacoplamento e de modularização.

A modularização é um conceito onde o sistema ou software é dividido em partes distintas. Compõe uma ferramenta fundamental para tornar o código mais legível, melhorando a sua manutenção, e para melhorar o seu desempenho por meio da programação estruturada. A utilização da modularização e do desacoplamento é vantajoso para tornar o código reutilizável.

Foi adicionado, ao projeto base inicial, a criação de salas de estudo como forma os alunos adquirirem mais conhecimentos sobre determinadas disciplinas. A criação das salas de estudo foi composta por quatro funcionalidades.

A primeira funcionalidade era a definição de horários das salas de estudo, por parte dos funcionários da biblioteca. Para esta definição o professor, que irá dar auxílio aos alunos, terá de fornecer as suas informações pessoais à biblioteca, tais como, as disciplinas que leciona, a sua disponibilidade horária e as suas habilitações literárias.

A segunda funcionalidade era o registo de entrada/saída dos alunos na sala de estudo, de modo, que a biblioteca fique a conhecer a frequência de cada aluno.

A terceira funcionalidade consiste na realização e cópias de fichas de trabalho pelo professor. Estas fichas irão dar auxílio na melhor aprendizagem de cada aluno.

Mais tarde, as fichas de trabalho (última funcionalidade) irão ser atribuídas pelos professores a cada aluno com base no seu nível de dificuldade e nos seus feedbacks anteriores.

Para a realização deste trabalho prático recorreu-se à prática de UML. UML (*Unified Modeling Language*) é uma linguagem que define uma série de artefactos que ajuda na tarefa de modular e documentar os sistemas orientados a objetos que se desenvolvem.

Dito isto, o presente relatório irá dividir-se:

-Primeiramente, explicar-se-á o projeto base que foi desenvolvido durante o semestre.

-No segundo capítulo, irá apresentar-se os USER STORYS das funcionalidades adicionadas.

-No terceiro capítulo, analisar-se-á o digrama de atividades da criação das salas de estudo, bem como a explicação da criação.

-No quarto e quarto capítulos, apresentar-se-á o digrama de classes e de casos de uso das funcionalidades que foram adicionadas ao projeto base.

-No capítulo seguinte, vai-se explicar os diagramas de sequência, um para cada funcionalidade.

-No antepenúltimo capítulo, irá apresentar-se o último diagrama do trabalho, o diagrama de empacotamento.

-Seguidamente, apresentar-se-á a explicação de todo o código realizado no *Intellij* para a criação destas quatro funcionalidades adicionais.

-Por fim, vai-se refletir e concluir sobre todo o trabalho realizado e sobre a dinâmica de grupo.

2.Projeto Base

Neste capítulo, irá falar-se do projeto de gestão de uma biblioteca que foi desenvolvido durante o semestre corrente. Este projeto visa o empréstimo de livros pelos utilizadores, tendo em conta um período limite de devolução e o número de livros emprestados em simultâneo. Caso o período limite de devolução seja excedido, o utilizador estava sujeito a pagar uma coima ou ser-lhe cancelado o registo. Neste caso, o utilizador receber uma notificação (vinda do funcionário da biblioteca) a referir a situação em que se encontra.

Os utilizadores tinham também a oportunidade de apresentar propostas de aquisição de livros à biblioteca. O responsável da biblioteca, para proceder à compra de livros (novos e/ou cópias), apresentava uma requisição de compra à direção da biblioteca, acompanhada por um ofício que fundamenta a necessidade da compra.

Primeiramente, foram realizados os doze UserStorys correspondentes ao que era pedido. Um dos conceitos em que foi baseado era na requisição de um livro de um dado utilizador, tendo em conta as regras que que lhe foram impostas, e, em caso de não comprimento, receber uma notificação. O outro conceito referia-se à compra de mais livros (novos ou cópias), tendo como base as propostas dadas pelos utilizadores, as consultas de aquisição e a requisição da compra de livros realizadas pelo responsável da biblioteca, a autorização da compra dos livros dada pelo diretor da biblioteca, e por fim, a encomenda e a entrada dos livros requeridos pelo responsável da biblioteca.

Depois dos UserStorys feitos, foi realizado os diagramas de casos de uso correspondes a todas as funcionalidades que o programa teria de ter.

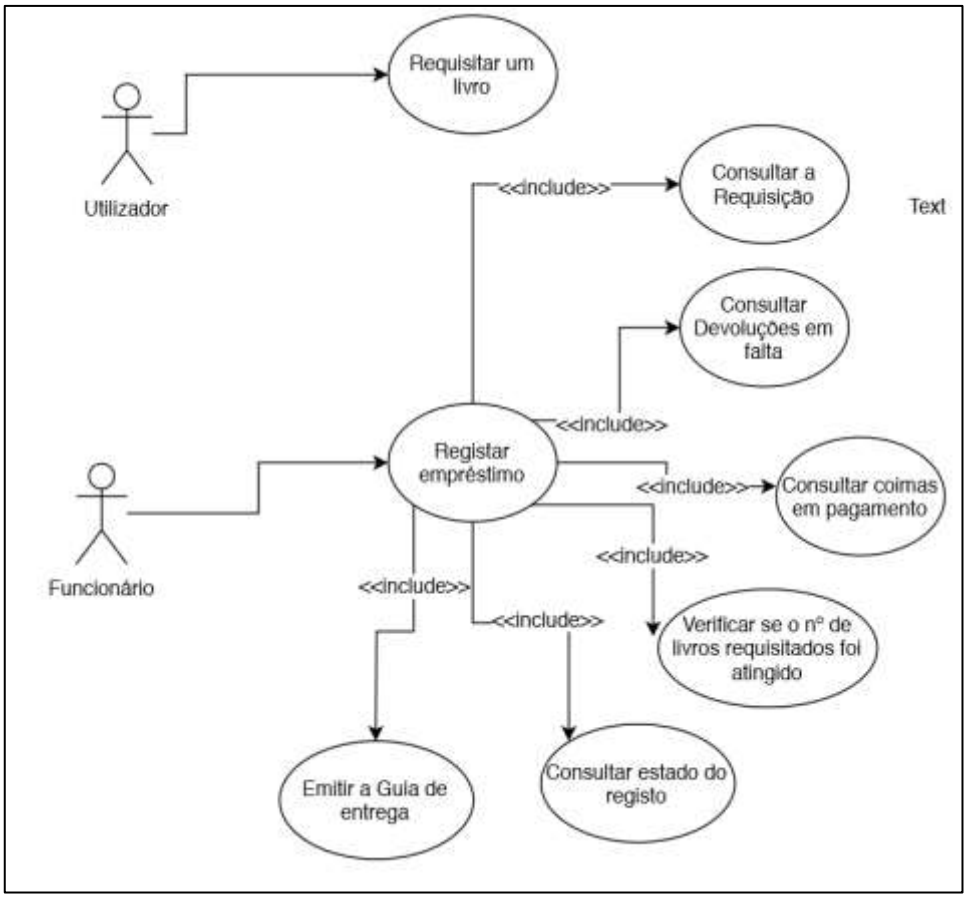


Figura 1: Diagrama de Casos de Uso do Empréstimo de livros.

No Diagrama de Casos de Uso, presente na Figura 1, verifica-se que o funcionário para registar o empréstimo da biblioteca (para o utilizador poder requisitar o livro) tinha de consultar a requisição, emitir guia de entrega e consultar se o utilizador: (1) tinha devoluções em falta; (2) se havia coimas para pagar; (3) se o limite do numero de livros requisitados tinha sido atingido; (4) qual era o estado do registo – se estava registado ou não.

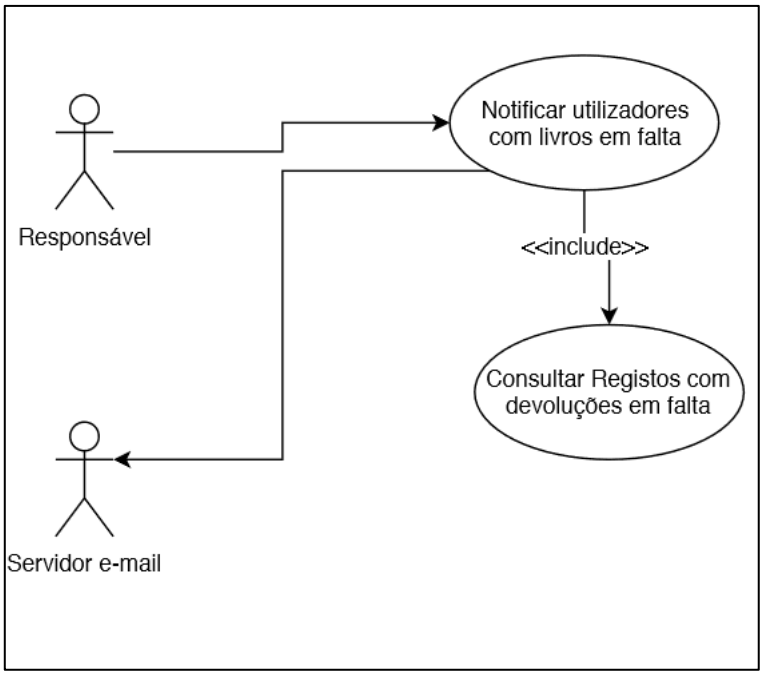


Figura 2: Diagrama de Casos de Uso de Notificação ao Utilizador.

O utilizador só era notificado se o prazo limite da devolução de um dado livro for ultrapassado. Assim, como demostra a Figura 2, o responsável para notificar os utilizadores com livros em falta, terá de consultar os registos de devoluções em falta. Sendo que, se este caso ocorrer o utilizador ia ser notificado através do seu e-mail.

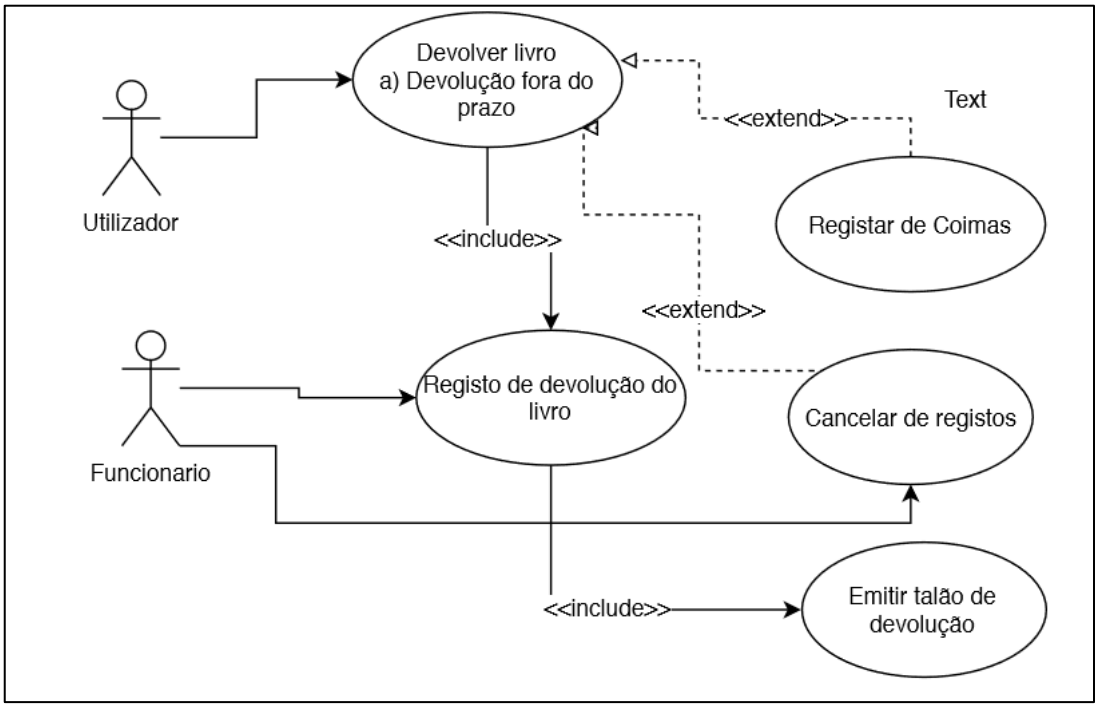


Figura 3: Diagrama de Casos de Uso da Devolução de Livro.

Como se verificou na Figura 3, o utilizador ao fazer a devolução do livro verificava-se se estava dentro ou fora do prazo, se estivesse fora sujeitava-se a pagar uma coima ou que lhe cancelassem o registo. Em qualquer uma das situações o funcionário da biblioteca ira registar a devolução do livro e emitir o talão.

Posteriormente, foram realizados dois diagramas de atividades que visava perceber melhor qual a ordem dos processos. Estes dois diagramas eram referentes às funcionalidades da proposta de aquisição de um determinado livro e a requisição de um livro.

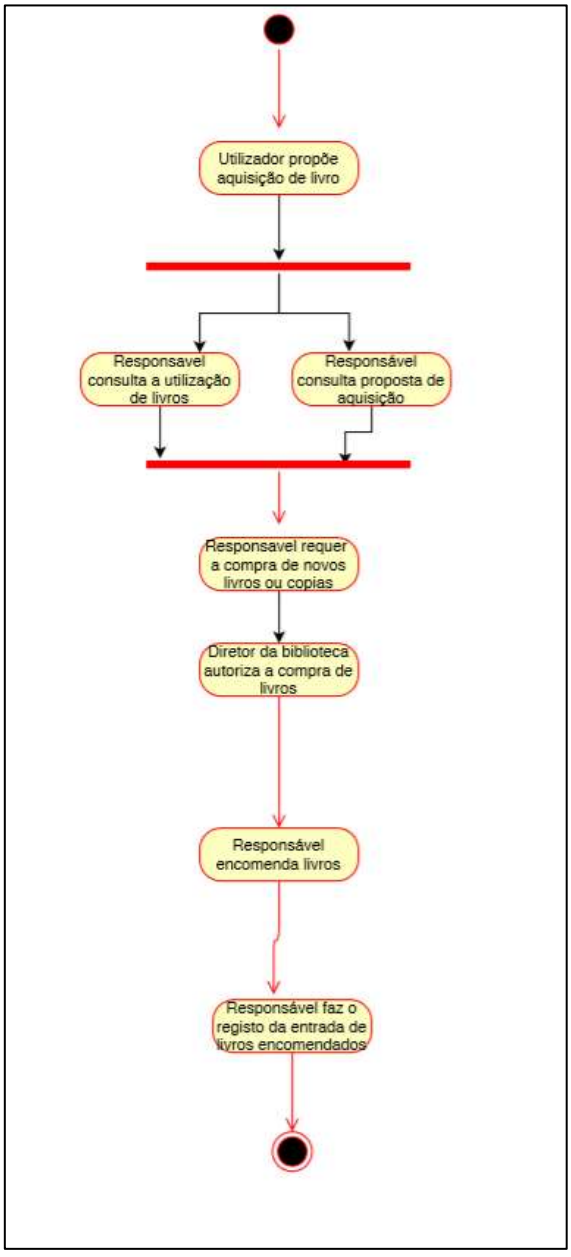


Figura 4: Diagrama de atividades da proposta de aquisição de um dado livro.

Na Figura 4 encontra-se um dos diagramas de atividades realizados. Este diagrama era referente à funcionalidade da proposta de aquisição de um livro (novo ou cópia) por parte do utilizador. Esta funcionalidade começava pelo utilizador propor à biblioteca um novo livro, sendo posteriormente, consultado a frequência de utilização do livro e a proposta do utilizador (estas duas atividades podiam ser realizadas em simultâneo). Depois do responsável requerer o livro e do diretor autorizar, faz-se a encomenda e regista-se a sua entrada na biblioteca.

Após o diagrama de atividades foi executado o diagrama de classes, com o objetivo de este conter todas as classes e métodos que iriam ser desenvolvidas no programa. Assim, realizou-se dezasseis classes pertencentes ao programa em si e mais duas classes gestores e moduladoras (‘Respositorio’ e ‘RepositorMem’).

Depois do código do programa feito, efetuou-se os diagramas de sequência e os diagramas de empacotamento.

3.User Storys

User Story é uma descrição concisa de uma necessidade do utilizador do produto, ou seja, de um requisito sob o ponto de vista desse utilizador. Um User Story procura descrever essa necessidade de forma simples e clara.

Um dos princípios por de trás das *User Story's* é a de que um produto poderia ser, integralmente, representado por meio de necessidades dos utilizadores. O produto desenvolvido com o *Scrum* é descrito por meio de itens do *Product Backlog* e, assim, cada um desses itens, de acordo com esse princípio, deve ser representado no formato User Story, ou seja, um User Story representa um e apenas um item do *Product Backlog*.

Foram descritos, e adicionados ao projeto base, os seguintes UserStorys:

Story ID: 1

Título: Fazer o registo de entrada

Enquanto: Alunos

quero registar a entrada

para registar a sua frequência das salas de estudo.

Confirmação Funcional

- O registo de entrada feito pelo aluno é realizado com sucesso.
- O aluno obtém uma mensagem de erro caso não haja lugares disponíveis.
- O aluno obtém uma mensagem de erro caso não esteja no horário estabelecido.

Story ID: 2

Título: Fazer o registo de saída

Enquanto: Alunos

quero registar a entrada

para registar a sua frequência das salas de estudo.

Confirmação Funcional

- O registo de saída feito pelo aluno é realizado com sucesso.
- O aluno obtém uma mensagem de erro caso já tenha registado a sua saída.
- O aluno obtém uma mensagem de erro caso não tenha registado a sua entrada.

Story ID: 3

Título: Criar Fichas de Trabalho

Enquanto: Professor

quero criar fichas de trabalho

para auxiliar o estudo dos alunos

Confirmação Funcional

- A criação das Fichas de Trabalho seja realizada com sucesso.
- O professor obtém uma mensagem de informação caso tenha fichas criadas.
- O professor obtém uma mensagem de alerta caso não tenha definido os seus horários disponíveis ou as disciplinas que leciona.

Story ID: 4

Título: Atribuir Fichas de trabalho aos Alunos

Enquanto: Professor

quero atribuir fichas de trabalho ao aluno

para o aluno adquirir melhores conhecimentos.

Confirmação Funcional

- A atribuição da ficha de trabalho ao aluno é realizada com sucesso.
- O professor obtém uma mensagem de erro caso já tenha atribuído a ficha de trabalho.
- O professor obtém uma mensagem de erro caso não tenha fichas de trabalho para atribuir.

Story ID: 5

Título: Definir os horários das salas de estudo

Enquanto: Funcionário

quero definir os horários das salas de estudo

para o aluno poder frequentar.

Confirmação Funcional

- A definição dos horários das salas de estudo é realizada com sucesso.
- O fornecedor obtém uma mensagem de erro caso não haja professores para auxiliar as salas de estudo.
- O fornecedor obtém uma mensagem de erro caso haja horários sobrepostos.

Story ID: 6

Título: Registar as informações profissionais

Enquanto: Professor

quero registar as minhas informações profissionais

para o funcionário poder fazer o horário das salas de estudo.

Confirmação Funcional

- A registo das informações profissionais é realizado com sucesso.
- O professor obtém uma mensagem de erro caso não tenha algum parâmetro dessas informações.

Story ID: 7

Título: Atribuir nível de dificuldade às fichas de trabalho

Enquanto: Professor

quero atribuir o nível de dificuldade às fichas de trabalho

para perceber a que nível o aluno está

Confirmação Funcional

- A atribuição do nível de dificuldade às fichas de trabalho é realizada com sucesso.
- O professor obtém uma mensagem de erro caso não tenha fichas de trabalho.

Story ID: 8

Título: Registar o feedback do nível de dificuldade da ficha de trabalho

Enquanto: Aluno

quero registar o feedback do nível de dificuldade da ficha de trabalho

para o professor conhecer quais as dificuldades do aluno

Confirmação Funcional

- O registo do feedback do nível de dificuldade da ficha de trabalho é realizado com sucesso.
- O aluno obtém uma mensagem de erro caso não tenha nenhuma ficha de trabalho.

Assim, estes userstorys deram à criação dos quatro casos de uso (funcionalidades) que foram adicionados ao projeto base. Nomeadamente, os userstorys com ID's: 1 e 2 pertenceram ao casos de uso do registo de entrada e saída de um aluno na Sala de estudo; 3 pertenceu ao caso de uso da criação de fichas de trabalho (do lado do professor) de forma a auxiliar o estudo dos alunos; 5 e 6, ao caso de uso definição dos horários das salas de estudo pelo funcionário com base nas informações profissionais de cada professor; por fim, 4, 7 e 8, ao caso de uso de atribuir, por parte do professor, as fichas de trabalho aos alunos dentro da sala de estudo com base no seu nível de dificuldade.

4.Diagramas de Atividades

Em UML, os diagramas de atividades, junto com os diagramas de caso de uso e de máquina de estados, são considerados diagramas de comportamento uma vez que descrevem o que é necessário acontecer no sistema sendo modelado. Estes diagramas são importantes a nível da comunicação com concisão e clareza.

Diagramas de atividades auxiliam a unir as pessoas das áreas de negócios e de desenvolvimento de uma organização para entender o mesmo processo de comportamento. Para criar um diagrama de atividade, é necessário um conjunto de símbolos especiais, incluindo aqueles para dar partida, encerrar, fundir ou receber etapas no fluxo.

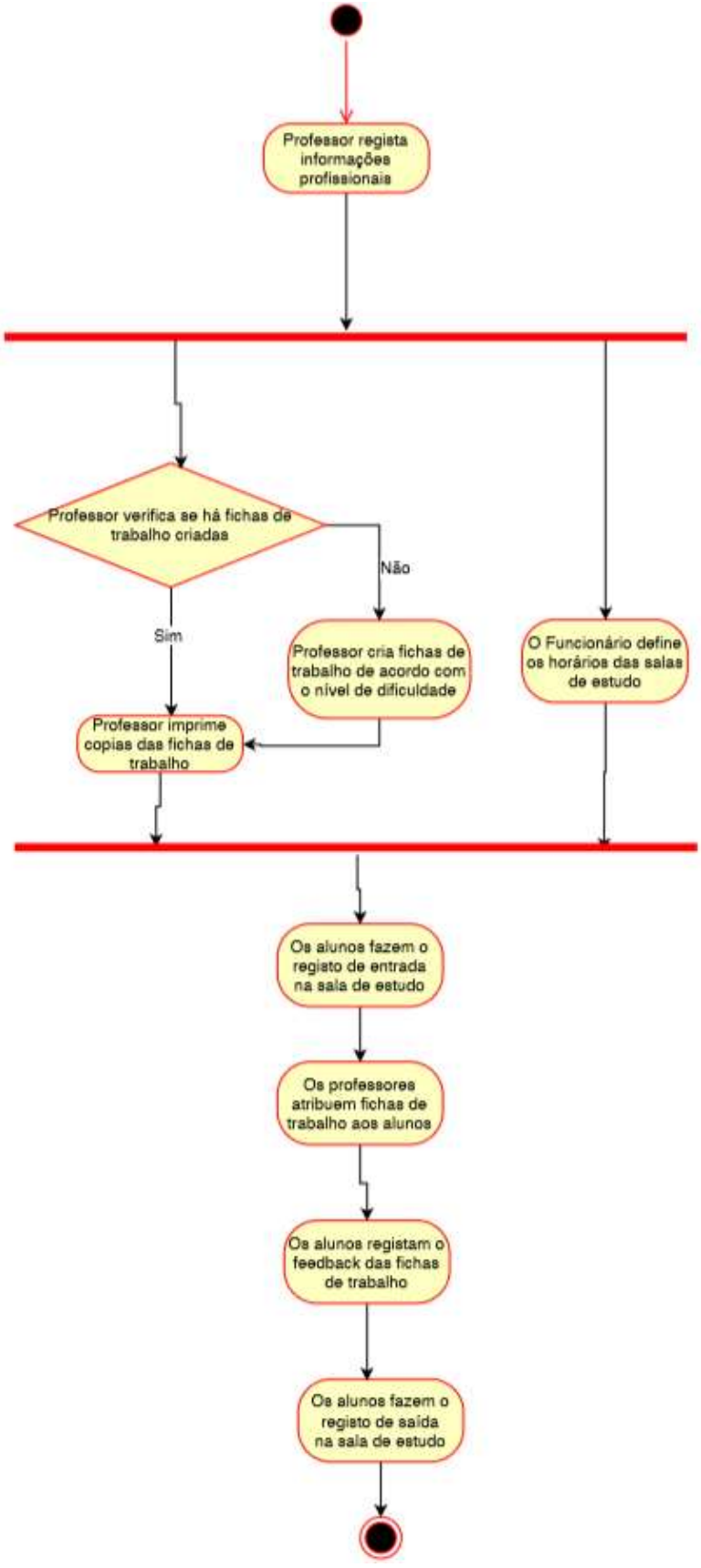


Figura 5: Diagrama de Atividades das Funcionalidades adicionais.

Analisando-se a Figura 5, primeiramente, os professores registam as suas informações profissionais, tais como, a disciplina que leciona, os horários que tem disponível, etc. Mais tarde, existem duas tarefas que se podem fazer em simultâneo, nomeadamente, enquanto o funcionário define qual o horário para as salas de estudo criadas na biblioteca, o professor faz a impressão das fichas de trabalho dos alunos (caso já forem criadas vai imprimi-las, caso contrário primeiro cria-as). Posteriormente, os alunos fazem o seu registo de entrada na sala de estudo e é lhe atribuído a ficha de trabalho por parte dos professores. Depois de realizada a ficha de trabalho, os alunos deixam o seu feedback (se era difícil, acessível, fácil, etc.) e fazem o seu registo de saída.

5.Diagramas de Classes

O diagrama de classes, em programação, é uma representação da estrutura e relações das classes que servem de modelo para objetos. Afirma-se, simplesmente, que seria um conjunto de objetos com as mesmas características, assim sabe-se identificar objetos e agrupá-los, de forma a encontrar as suas respetivas classes. No diagrama de classes (em UML), uma classe é representada por um retângulo com três divisões, entre as quais, o nome da classe, os atributos da mesma classe e, por fim, os métodos.

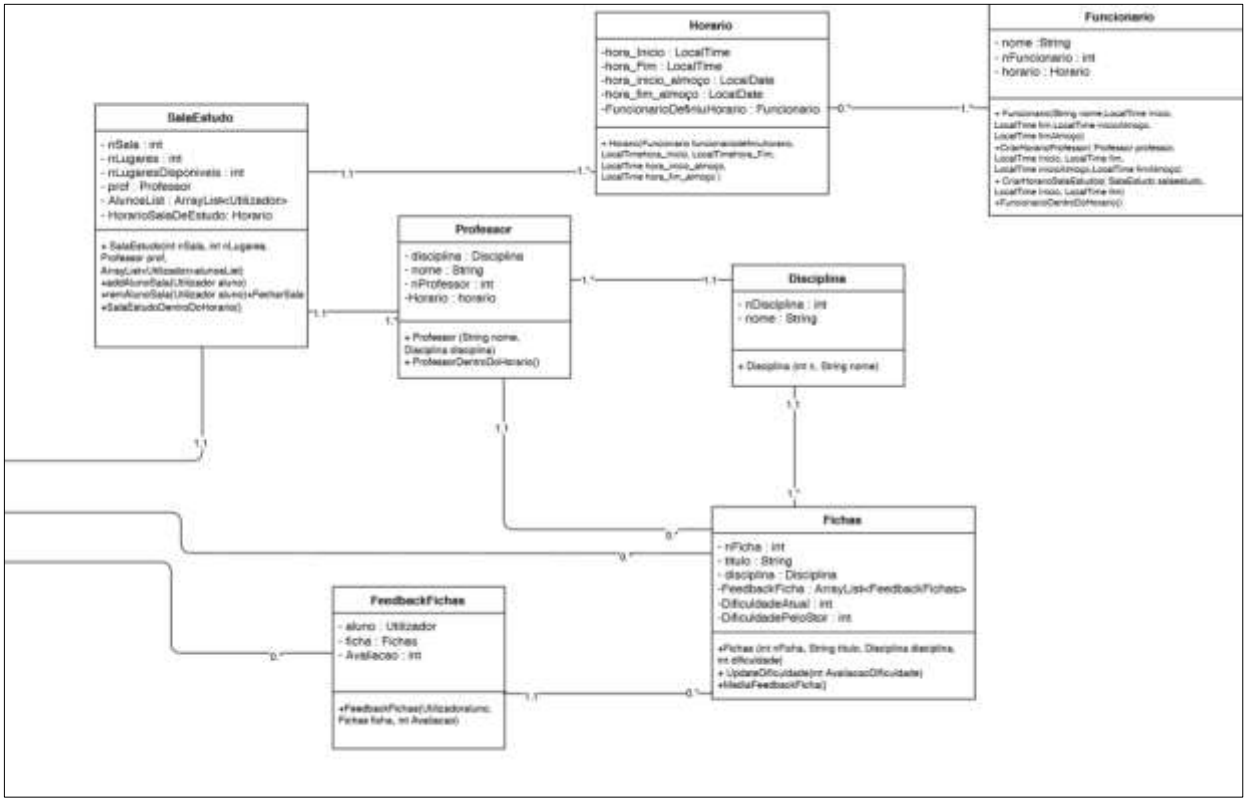


Figura 6: Diagrama de Classes das funcionalidades adicionais.

Observando-se a Figura 6, foram adicionadas sete classes ao projeto base. A classe ‘Disciplina’ armazena os dados da disciplina em questão, que é útil para a criação de ‘Ficha’ de trabalho de um ou mais ‘Professor’, de uma dada disciplina. A classe ‘Utilizador’ avaliar uma dada ‘Ficha’ que lhe é atribuída pelo ‘Professor’, através do ‘FeedbackFichas’. O ‘Funcionario’ define o ‘Horario’ para cada ‘Sala de Estudo’, com base das informações profissionais de cada ‘Professor’. A classe ‘Sala de Estudo’ gere toda a atividade que ocorre.

6.Diagramas de Casos de Uso

O diagrama de casos de uso é um diagrama que documenta o que o sistema faz do ponto de vista do utilizador. Em outras palavras, este descreve as principais funcionalidades do sistema e a interação dessas funcionalidades com os utilizadores do mesmo sistema. Este artefacto é derivado da especificação de requisitos, que por sua vez não faz parte da UML. Pode ser utilizado para criar o documento de requisitos.

Os diagramas de casos de uso são compostos por quatro partes: (1) cenário que corresponde à sequencia de eventos que acontecem quando o utilizador interage com o sistema, (2) ator que é o utilizador do sistema, (3) caso de uso, ou seja, a tarefa ou funcionalidade desempenhada pelo ator, e por fim, (4) comunicação que é a ligação entre o ator e o caso de uso.

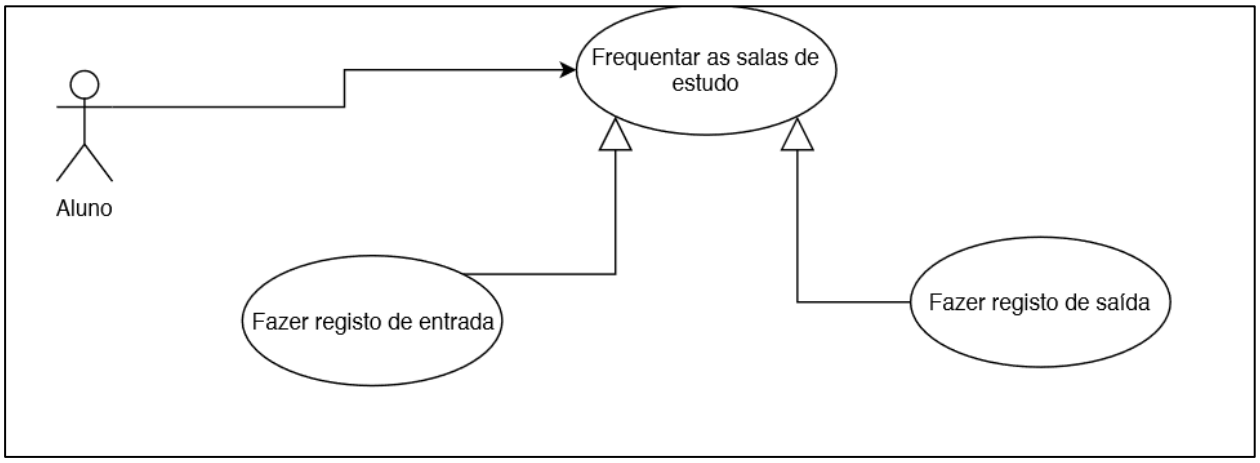


Figura 7: Caso de uso- Registo de entrada e saída da sala de estudo.

Na Figura 7, definiu-se o caso de uso registo de entrada e saída de um aluno da sala de estudo. Assim, criou-se uma relação de herança sobre a frequentação do aluno (Ator) as salas de estudo, que visa registar quando entra na sala e quando saí.

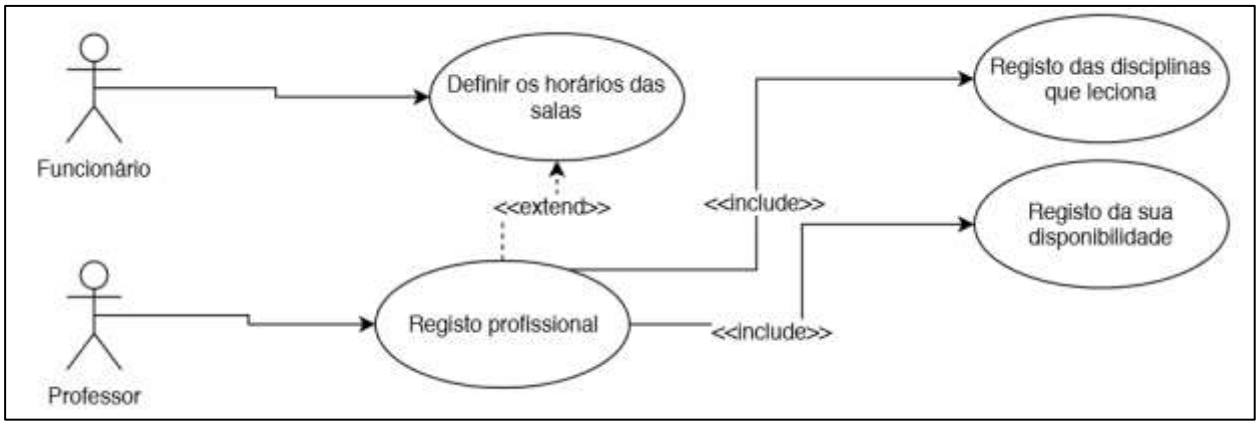


Figura 8: Caso de uso- Definição dos horários das salas de estudo.

Analogamente à Figura 8, desenvolveu-se o caso de uso para a definição de horários da sala de estudo. Assim, o funcionário define os horários com base nas informações profissionais que lhe são fornecidas pelo professor. Essas informações incluem as disciplinas lecionadas, a disponibilidade do horário e as habilitações literárias.

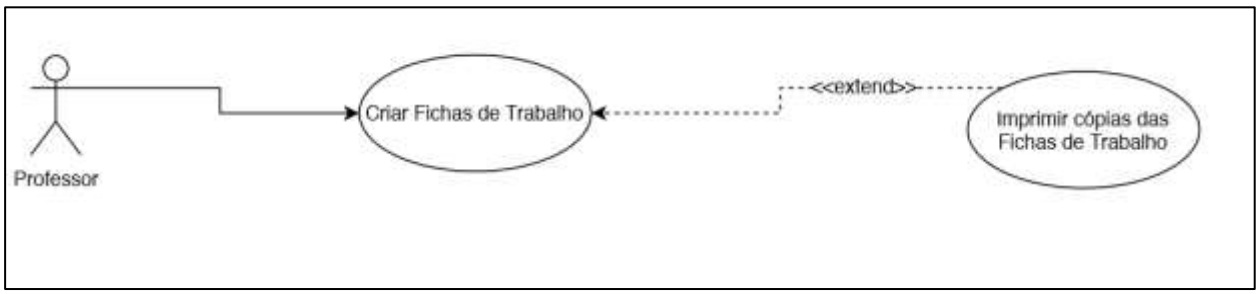


Figura 9: Caso de uso- Criação de fichas de trabalho.

Conforme a Figura 9, definiu-se o caso de uso para a criação de fichas de trabalho pelo professor. Após esta criação, o professor pode ou não imprimir cópias das fichas para, posteriormente, entregar aos alunos.

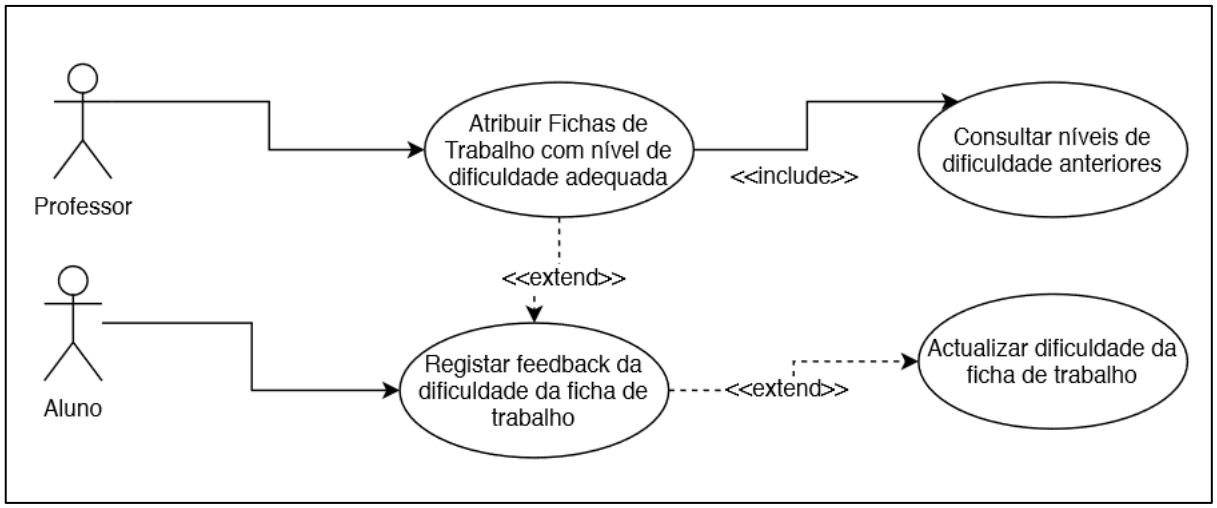


Figura 10: Caso de uso- Atribuição de fichas de trabalho aos alunos.

Notando a Figura 10, criou-se o caso de uso que atribui (o professor) a ficha de trabalho de acordo com o nível de dificuldade. Para a atribuição da ficha de trabalho, o professor deve antes consultar o nível de dificuldade anterior. Depois de atribuída a ficha ao aluno, este pode deixar o seu feedback em relação à dificuldade da ficha.

7.Diagramas de Sequências

Um diagrama de sequência é, praticamente, um diagrama de interação, pois descreve como, e em qual ordem, um grupo de objetos trabalha em conjunto. Estes diagramas são utilizados por engenheiros de softwares e profissionais de negócio para entender as necessidades de um novo sistema para documentar um processo já existente.

```
Utilizador aluno = new Utilizador("Kayle",null,null);
Utilizador aluno2 = new Utilizador("Lucius",null,null);
Utilizador aluno3 = new Utilizador("Barney",null,null);
Utilizador aluno4 = new Utilizador("Ashley",null,null);

rep.entradaAlunoNaSala(salaEstudo,aluno);
rep.entradaAlunoNaSala(salaEstudo,aluno2);
rep.entradaAlunoNaSala(salaEstudo,aluno3);
//rep.saidaAlunoNaSala(salaEstudo,aluno3);
//rep.FecharSalaDeAula(salaEstudo);
```

Figura 11: Código para a funcionalidade de registar a entrada e saída do utilizador.

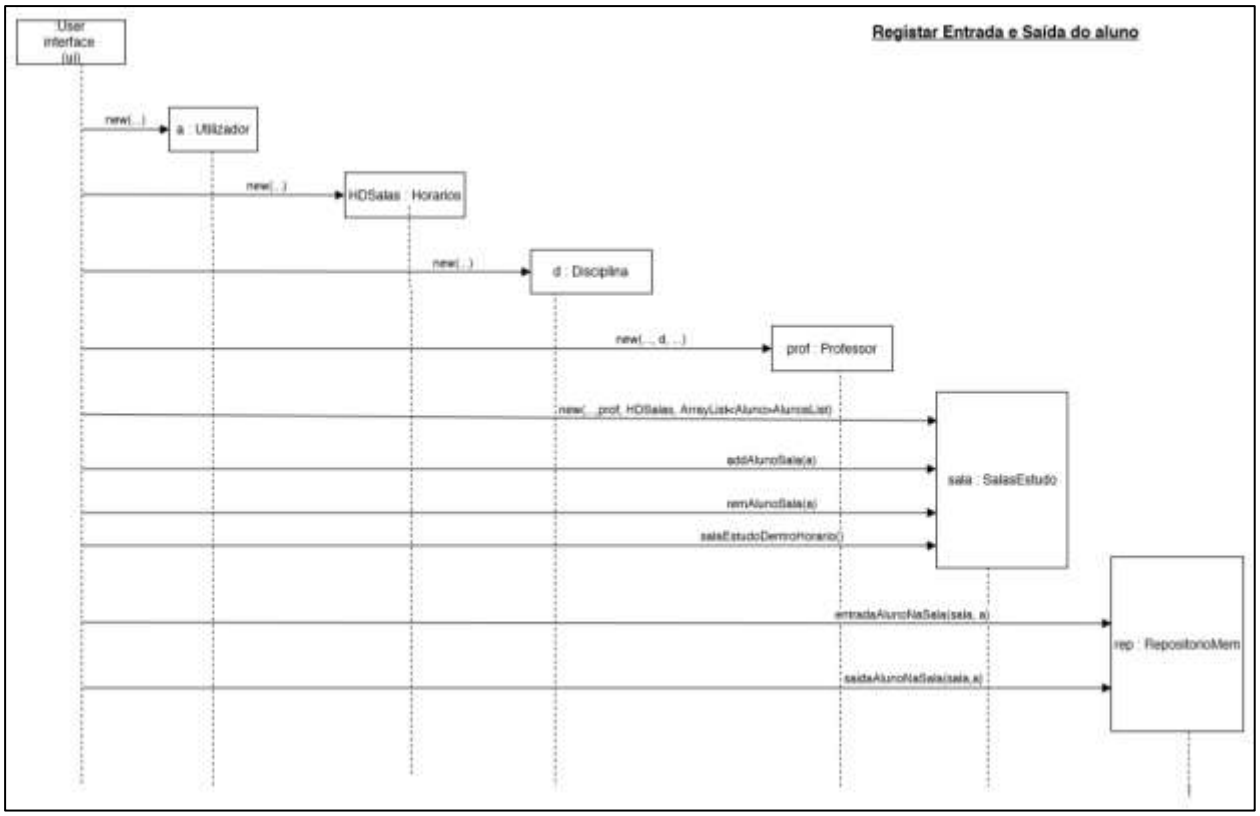


Figura 12: Diagrama de sequência do registo de entrada e saída de um utilizador.

A partir do código presente na Figura 11, fez-se o diagrama de sequencia da Figura 12, no qual para o registo de entrada e saída de um utilizador é preciso a classe Utilizador, Horario para verificar se esta ou não dentro do horário da sala de estudo, Disciplina, Professor, SalasEstudo e RepositorioMem para proceder ao registo.

```
Funcionario funcionario = new Funcionario("Luis",LocalTime.of(8,00,00),LocalTime.of(17,00,00),LocalTime.of(13,00,00),LocalTime.of(1

rep.CriarHorarioProfessor(funcionario,prof,LocalTime.of(8,00,00),LocalTime.of(15,00,00),LocalTime.of(13,00,00),LocalTime.of(14,00,0

System.out.println("Horario do funcionario");
System.out.println("Inicio: " + funcionario.getHorario().getHora_Inicio());
System.out.println("Fim: " + funcionario.getHorario().getHora_Fim());

System.out.println("_______");

System.out.println("Horario do Prof");
System.out.println("Inicio: "+ prof.getHorario().getHora_Inicio());
System.out.println("Fim: " + prof.getHorario().getHora_Fim());
```

Figura 13: Código para a funcionalidade da definição de horários.

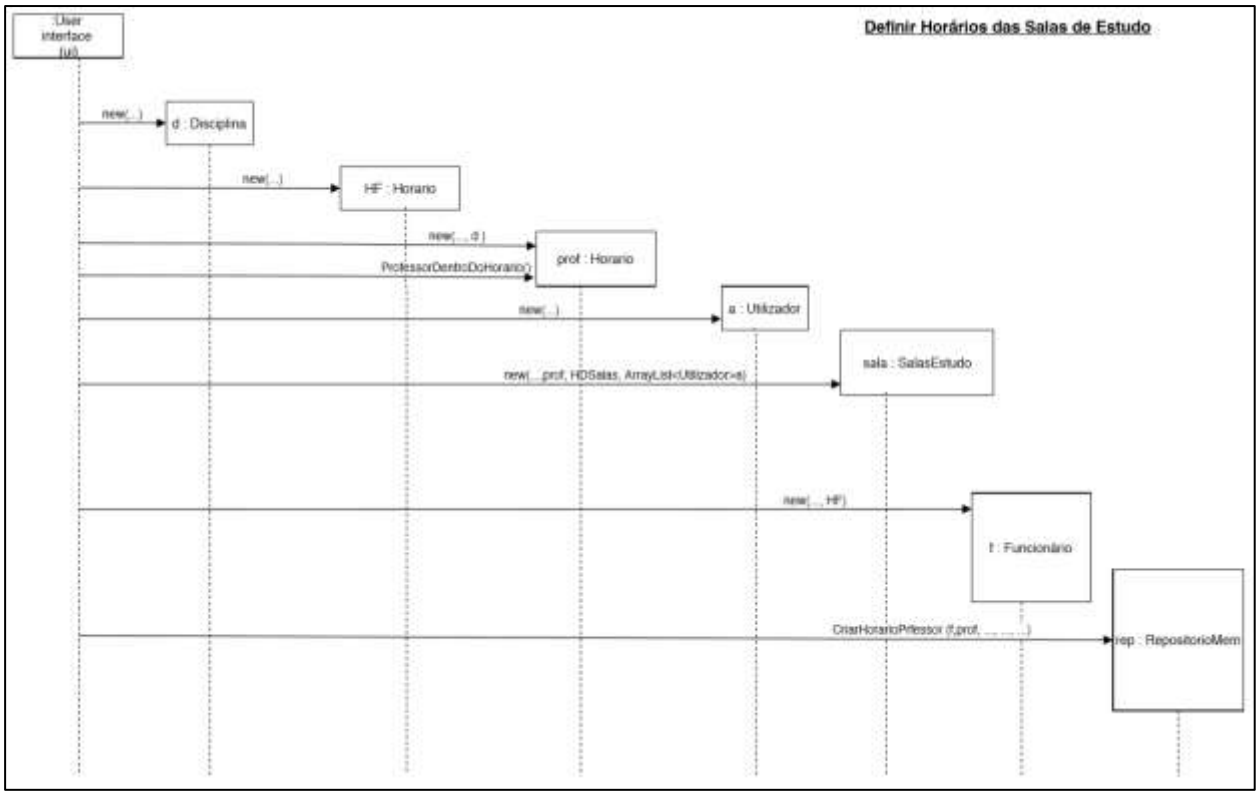


Figura 14: Diagrama de sequência da definição de horários.

Através do código da Figura 13, procedeu-se à formação do diagrama de sequência da Figura 14. Assim, para o funcionário criar o horário das salas de estudo, utilizava-se as classes Disciplina, Horario, Professor, Utilizador, SalasEstudo, Funcionario, RepositorioMem.

```
Fichas ficha = new Fichas("Geometria",disciplina3,12);

System.out.println("Dificuldade da ficha Inicio: " + ficha.getDificuldade());

rep.EntregarFichaAAluno(ficha,aluno);
rep.EntregarFichaAAluno(ficha,aluno2);
rep.EntregarFichaAAluno(ficha,aluno3);

rep.AlunoAvaliaFicha(aluno,ficha,14);
rep.AlunoAvaliaFicha(aluno2,ficha,16);
rep.AlunoAvaliaFicha(aluno3,ficha,16);

System.out.println("Dificuldade da ficha depois de avaliada pelos alunos: " + ficha.getDificuldade());

System.out.println(salaEstudo.getAlunosList().size());
```

Figura 15: Código da funcionalidade da atribuição de avaliação de fichas.

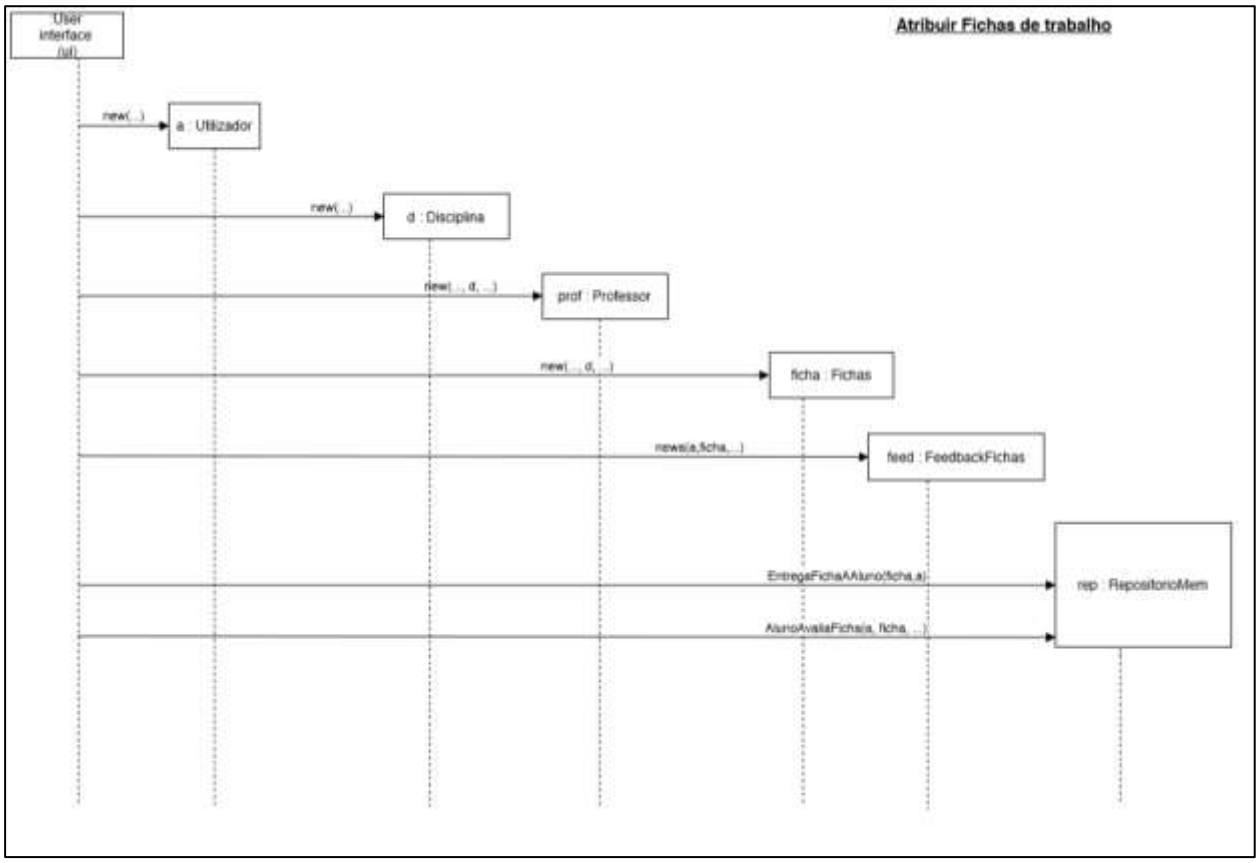


Figura 16: Diagrama de sequência da atribuição e avaliação de fichas.

Com o código da Figura *15*, fez-se o diagrama de sequência presente na Figura 16 para a funcionalidade de o professor atribuir a cada aluno uma ficha de trabalho e este avalia-la depois de a fazer. Dito isto, recorreu-se às classes Utilizador, Disciplina, Professor, fichas, FeedbackFichas e RepositorioMem.

8.Diagramas de Empacotamento

O Diagrama de empacotamento, em UML, é um mecanismo de agrupamento genérico, ajudando a descrever a arquitetura lógica de um sistema.

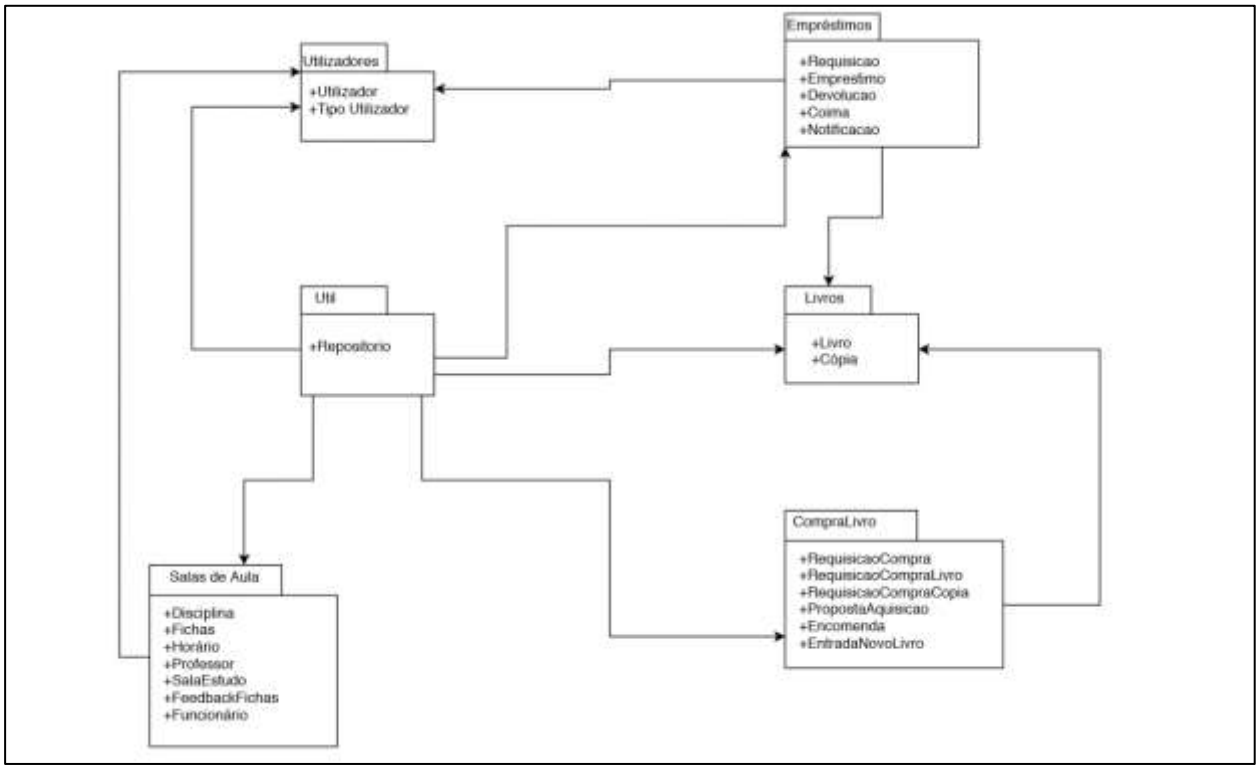


Figura 17: Diagrama de empacotamento.

No projeto base, o diagrama de empacotamento continha cinco pastas: (1) Utilizadores, que possuía dados do TipoUtilizador e do Utilizador; (2) Emprétimos, que tinha dados da Requisição, do Emprestimo, da Devolução, do Utilizador, da Coima e da Notificação; (3) Livros, que tinha dados da Cópia e do livro; (4) CompraLivro, que era constituída por dados vindos do livro, da RequisicaoCompra, da RequisicaoCompraCopia, da RequisicaoCompraLivro, da PropostaAquisicao, da Encomenda e da EntradaNovoLivro; (5) por fim, Rpositorio que engloba todas as outras pastas.

Analisando-se a Figura 7, foi adicionado uma nova pasta derivado às novas funcionalidades. Esta pasta denominou-se por Salas de Aula que contém dados sobre a disciplina, Utilizador, Horario, Professor, SalaEstudo, FeedbackFichas e Funcionário.

9.Explicação do código

Neste capítulo, irá abordar-se sobre o código e o raciocínio que se teve para a realização das novas funcionalidades.

De modo a implementar o sistema pretendido adicionou-se ao projeto as seguintes classes:

(1) Sala de Estudo:

```
public class SalaEstudo {
    private int nSala;
    private int nLugares;
    private int nLugaresDisponiveis;
    private Professor prof;
    private ArrayList<Utilizador> AlunosList;
    private Horario HorarioSalaEstudo;

    public SalaEstudo(int nSala, int nLugares) {...}

    public int getnSala() { return nSala; }

    public void setnSala(int nSala) { this.nSala = nSala; }

    public int getnLugares() { return nLugares; }

    public void setnLugares(int nLugares) { this.nLugares = nLugares; }

    public Professor getProf() { return prof; }

    public void setProf(Professor prof) {...}

    public void removeProf() {this.prof = null;}

    public ArrayList<Utilizador> getAlunosList() { return AlunosList; }

    public void setAlunosList(ArrayList<Utilizador> alunosList) { AlunosList = alunosList; }

    public int getnLugaresDisponiveis() { return nLugaresDisponiveis; }

    public void setnLugaresDisponiveis(int nLugaresDisponiveis) { this.nLugaresDisponiveis = nLugaresDisponiveis; }

    public Horario getHorarioSalaEstudo() { return HorarioSalaEstudo; }

    public void setHorarioSalaEstudo(Horario horarioSalaEstudo) { HorarioSalaEstudo = horarioSalaEstudo; }

    public boolean addAlunoSala(Utilizador aluno){...}

    public boolean remAlunoSala(Utilizador aluno) {...}

    public boolean FecharSala(SalaEstudo salaEstudo) {...}

    public boolean SalaEstudoDentroDoHorario() {...}
```

Figura 18: Classe sala de estudos.

```
public boolean SalaEstudoDentroDoHorario() { //Verifica se a sala de estudo está aberta
    if(LocalTime.now().compareTo(this.HorarioSalaEstudo.getHora_Inicio()) >=0 &&
            LocalTime.now().compareTo(this.HorarioSalaEstudo.getHora_Fim()) <= 0 ) {
        return true;
    }
    else
        return false;
}
```

Figura 19: Função da classe Sala de Estudo: SalaEstudoDentroDoHorario().

```java
public boolean FecharSala(SalaEstudo salaEstudo) {

    if (salaEstudo == null) return false;

    for (Utilizador u : AlunosList) {
        u.setDentroSaladeEstudo(0);
    }
    AlunosList.removeAll(AlunosList); //Todos os alunos saiem da sala
    prof = null; //o professor sai da sala
    nLugaresDisponiveis=this.nLugares; //Visto que todos os alunos saiem da sala
    return true;
}
```

Figura 20: Função da classe Sala de Estudo: FecharSala().

Na classe presente na Figura 18, destacou-se as funções de verificação do estado da sala - Figura 19 - (dentro do horário, fora do horário), bem como a função *FecharSala()* - Figura 20- que quando é chamada remove todos os alunos e professor da sala.

(2) Professor:

```java
public class Professor  {
    private static final AtomicInteger count = new AtomicInteger( initialValue: 0);
    private Disciplina disciplina;
    private String nome;
    private int nProfessor;
    private Horario horario;


    public Professor(Disciplina disciplina , String nome) {...}

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }

    public int getnProfessor() { return nProfessor; }

    public Disciplina getDisciplina() { return disciplina; }

    public void setDisciplina(Disciplina disciplina) { this.disciplina = disciplina; }

    public Horario getHorario() {...}

    public void setHorario(Horario horario) {...}

    public boolean ProfessorDentroDoHorario() {...}
}
```

Figura 21: Classe Professor.

Mais uma vez, existiu a funcionalidade de verificar se o Professor se encontra dentro ou fora do seu Horário de trabalho, como se observou na Figura 21.

(3) Disciplina:

```java
public class Disciplina {
    private static final AtomicInteger count = new AtomicInteger( initialValue: 0);
    private int nDisciplina;
    private String nome;

    public Disciplina( String nome) {
        this.nDisciplina = count.incrementAndGet();
        this.nome = nome;
    }

    public int getnDisciplina() { return nDisciplina; }

    public void setnDisciplina(int nDisciplina) { this.nDisciplina = nDisciplina; }

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }
}
```

Figura 22: Classe Disciplina.

Na Figura 22, criou-se a classe disciplina para dar auxílio à identificação dos professores.

(4) Funcionário:

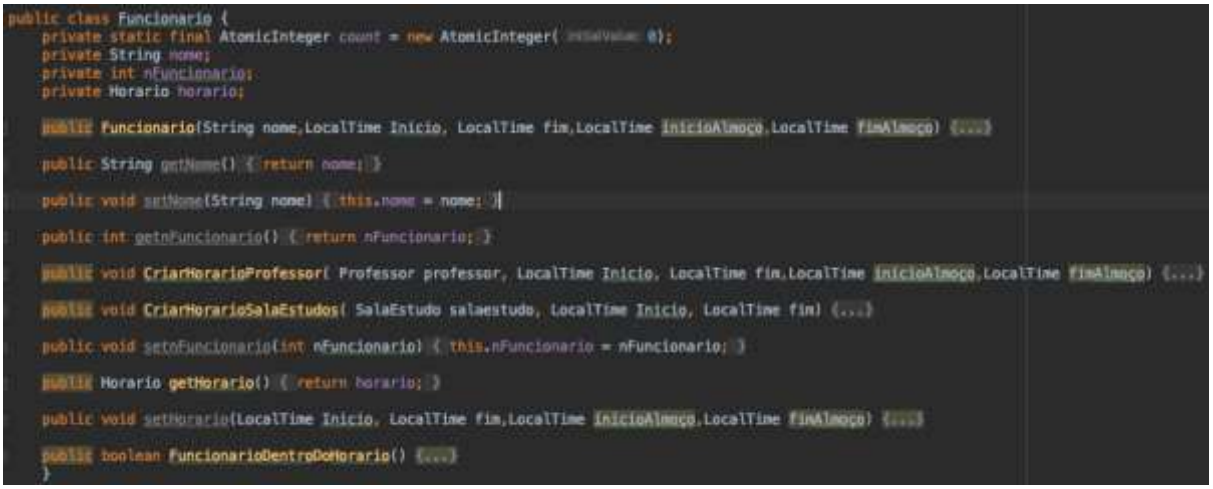
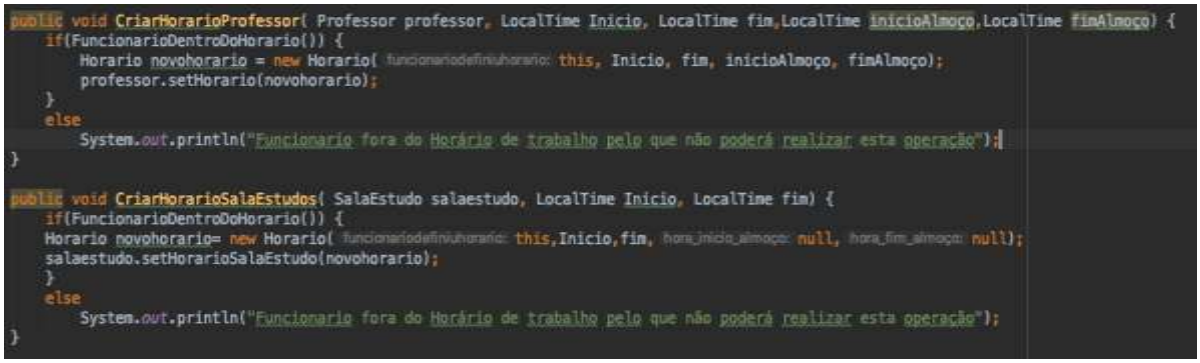


Figura 23: Classe Funcionário.



```
public void CriarHorarioProfessor( Professor professor, LocalTime Inicio, LocalTime fim,LocalTime inicioAlmoço,LocalTime fimAlmoço) {
    if(FuncionarioDentroDoHorario()) {
        Horario novohorario = new Horario( funcionariodefiniuhorario: this, Inicio, fim, inicioAlmoço, fimAlmoço);
        professor.setHorario(novohorario);
    }
    else
        System.out.println("Funcionario fora do Horário de trabalho pelo que não poderá realizar esta operação");
}

public void CriarHorarioSalaEstudos( SalaEstudo salaestudo, LocalTime Inicio, LocalTime fim) {
    if(FuncionarioDentroDoHorario()) {
    Horario novohorario= new Horario( funcionariodefiniuhorario: this,Inicio,fim, hora_inicio_almoço: null, hora_fim_almoço: null);
    salaestudo.setHorarioSalaEstudo(novohorario);
    }
    else
        System.out.println("Funcionario fora do Horário de trabalho pelo que não poderá realizar esta operação");
}
```

Figura 24: Função da classe Funcionário: CriarHorarioSalaEstudo () e CriarHorarioProfessor ().

De novo, apresentou-se a funcionalidade de verificar se o Funcionário se encontra dentro ou fora do seu Horário- Figura 23. Contudo, nesta classe procedemos à criação de funções que têm como objetivo criar horários quer dos professores quer das salas de estudo, uma vez que esta é uma tarefa do Funcionário, por sua vez o funcionário apenas consegue realizar estas tarefas durante o seu horário de trabalho- Figura 24.

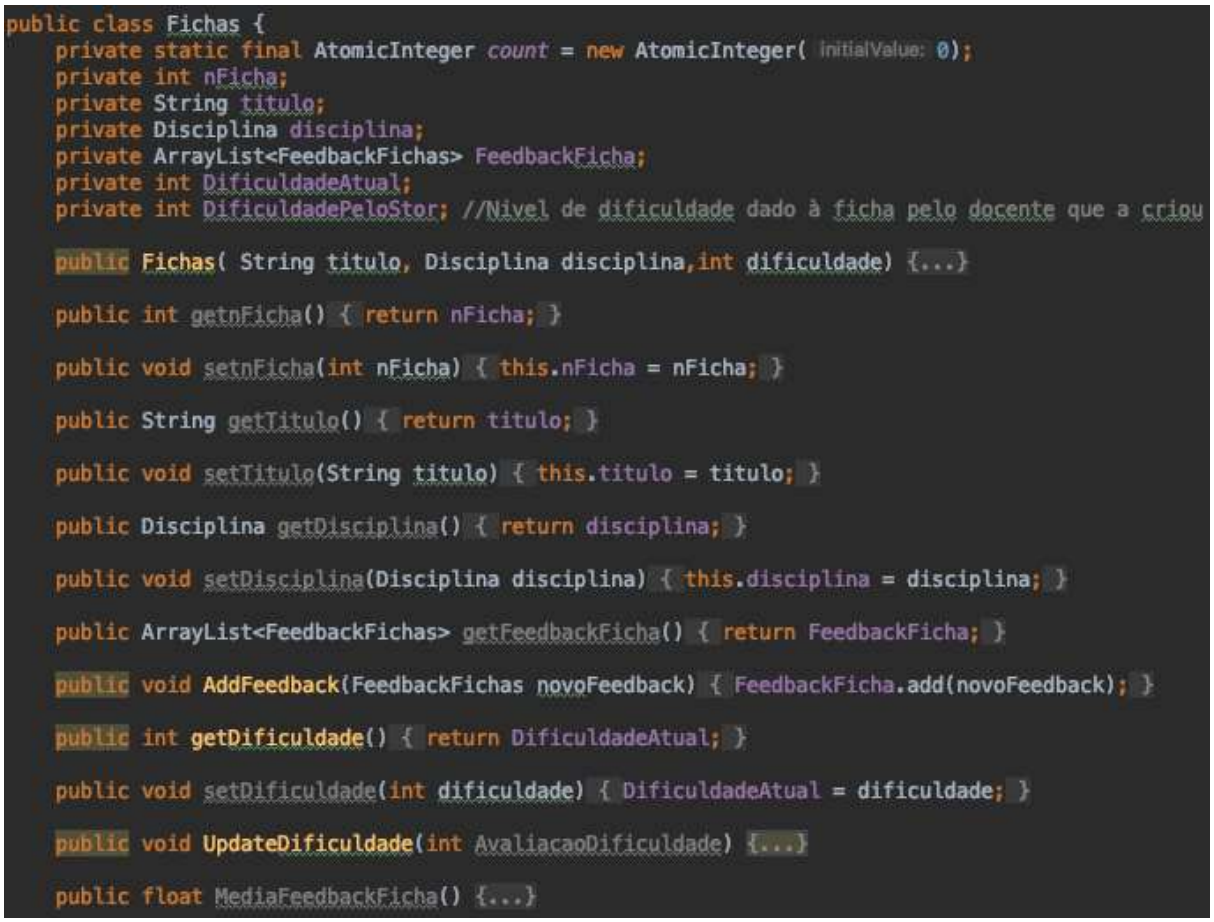
(5) Horário:

```
public class Horario {
    private LocalTime hora_Inicio;
    private LocalTime hora_Fim;
    private LocalTime hora_inicio_almoço;
    private LocalTime hora_fim_almoço;
    private Funcionario FuncionarioDefiniuHorario;

    public Horario(Funcionario funcionariodefiniuhorario,LocalTime hora_Inicio, LocalTime hora_Fim,LocalTime hora_inicio_almoço,LocalTime hora_fim_almoço ) {...}

    public LocalTime getHora_Inicio() {
        return hora_Inicio;
    }

    public void setHora_Inicio(LocalTime hora_Inicio) { this.hora_Inicio = hora_Inicio; }

    public LocalTime getHora_Fim() { return hora_Fim; }

    public void setHora_Fim(LocalTime hora_Fim) { this.hora_Fim = hora_Fim; }

    public Funcionario getFuncionarioDefiniuHorario() { return FuncionarioDefiniuHorario; }

    public void setFuncionarioDefiniuHorario(Funcionario funcionarioDefiniuHorario) {...}

    public LocalTime getHora_inicio_almoço() {
        return hora_inicio_almoço;
    }

    public void setHora_inicio_almoço(LocalTime hora_inicio_almoço) { this.hora_inicio_almoço = hora_inicio_almoço; }

    public LocalTime getHora_fim_almoço() { return hora_fim_almoço; }

    public void setHora_fim_almoço(LocalTime hora_fim_almoço) { this.hora_fim_almoço = hora_fim_almoço; }
}
```

Figura 25: Classe Horario.

Na *Figura 25* foi criada a classe Horario com o intuito do funcionário fazer os horários das salas de estudo, durante os dias da semana, definindo os professores e disciplinas em cada horário.

(6) Fichas:



```
public class Fichas {
    private static final AtomicInteger count = new AtomicInteger( initialValue: 0);
    private int nFicha;
    private String titulo;
    private Disciplina disciplina;
    private ArrayList<FeedbackFichas> FeedbackFicha;
    private int DificuldadeAtual;
    private int DificuldadePeloStor; //Nivel de dificuldade dado à ficha pelo docente que a criou

    public Fichas( String titulo, Disciplina disciplina,int dificuldade) {...}

    public int getnFicha() { return nFicha; }

    public void setnFicha(int nFicha) { this.nFicha = nFicha; }

    public String getTitulo() { return titulo; }

    public void setTitulo(String titulo) { this.titulo = titulo; }

    public Disciplina getDisciplina() { return disciplina; }

    public void setDisciplina(Disciplina disciplina) { this.disciplina = disciplina; }

    public ArrayList<FeedbackFichas> getFeedbackFicha() { return FeedbackFicha; }

    public void AddFeedback(FeedbackFichas novoFeedback) { FeedbackFicha.add(novoFeedback); }

    public int getDificuldade() { return DificuldadeAtual; }

    public void setDificuldade(int dificuldade) { DificuldadeAtual = dificuldade; }

    public void UpdateDificuldade(int AvaliacaoDificuldade) {...}

    public float MediaFeedbackFicha() {...}
```

Figura 26: Classe Fichas.

```
public void UpdateDificuldade(int AvaliacaoDificuldade) {
    int i;
    int SomaFeedbackAlunos = 0;
    float FeedbackAlunos;
    float Total;
    for (i=0;i<FeedbackFicha.size();i++) {
        SomaFeedbackAlunos +=FeedbackFicha.get(i).getAvaliacao();
    }
    FeedbackAlunos= SomaFeedbackAlunos /FeedbackFicha.size(); //Média da soma das avaliações dos alunos

    Total= (FeedbackAlunos / 3) + (2 * DificuldadePeloStor / 3); //Definimos que a avaliação da dificuldade do stor tem um peso de 2/3
                                                                 //enquanto que a dos alunos tem 1/3
    this.DificuldadeAtual=Math.round(Total);
}
```

Figura 27: Função da classe fichas: UpdateDificuldade().

Uma das funções existentes na classe Fichas - Figura 26 - designou-se *UpdateDificuldade ()* - Figura 27 -, esta será introduzida pelo professor aquando da criação da Ficha, contudo irá sofrer alterações à medida que é avaliada pelos alunos (FeedbackFichas).

(7) FeedbackFichas:

```
public class FeedbackFichas {
    private Utilizador aluno;
    private int Avaliacao;
    private Fichas ficha;

    public FeedbackFichas(Utilizador aluno, Fichas ficha, int avaliacao) {
        this.aluno = aluno;
        this.ficha = ficha;
        Avaliacao = avaliacao;
    }

    public Utilizador getAluno() { return aluno; }

    public void setAluno(Utilizador aluno) { this.aluno = aluno; }

    public Fichas getFicha() { return ficha; }

    public void setFicha(Fichas ficha) { this.ficha = ficha; }

    public int getAvaliacao() {
        return Avaliacao;
    }

    public void setAvaliacao(int avaliacao) { Avaliacao = avaliacao; }
}
```

Figura 28: Classe FeedbackFichas.

A classe presente na Figura 28 foi utilizada para o aluno, depois de fazer a ficha de trabalho (atribuída pelo professor), deixar o seu comentário à cerca dela.

Depois das classes serem criadas, procedeu-se ao desenvolvimento das funcionalidades adicionais propostas pelo grupo.

```
private ArrayList<SalaEstudo> salaEstudoL = new ArrayList<>();
private ArrayList<Professor> professorL = new ArrayList<>();
private ArrayList<Disciplina> disciplinaL = new ArrayList<>();
private ArrayList<Fichas> FichasL = new ArrayList<>();
```

Figura 29: *Array´s* adicionados à classe RepositorioMem.

Com o intuito de guardar a informação de cada um dos elementos, procedeu-se à criação de ArrayLists na classe RepositorioMem - Figura 29 -, é de mencionar que não se criou um ArrayLists para os horários e FeedbackFichas visto que os mesmos estão ligados aos seus respetivos elementos e considerou-se que não faria sentido guardá-los de outra forma.

```
private static final AtomicInteger count = new AtomicInteger( initialValue: 0);
private String nome;
private int nFuncionario;
private Horario horario;

public Funcionario(String nome,LocalTime Inicio, LocalTime fim,LocalTime inicioAlmoço,LocalTime fimAlmoço) {
    this.nome = nome;
    this.nFuncionario = count.incrementAndGet();
    Horario horario= new Horario( funcionariodefiniuhorario: null,Inicio,fim,inicioAlmoço,fimAlmoço);
    this.horario=horario;
}
```

Figura 30: Auto- incrementação nas classes.

De modo a criar de uma forma mais simples os Utilizadores, Funcionários, Professores, Disciplinas e Salas de Estudo recorreu-se à utilização de uma funcionalidade que permite realizar *auto increment* -Figura 30 - do ID (número) do elemento criado, ao invés de termos de especificar qual o ID (número) na criação de cada elemento, esta funcionalidade é demonstrada na figura abaixo.

Com as classes demonstradas procedeu-se à criação de Horários para as Salas de Estudo e Professores, como já demonstrou em cima. Decidiu-se, também, estabelecer horários para os Funcionários, contudo como não seria o funcionário que estabeleceria o seu próprio horário designou-se o mesmo aquando da criação do funcionário.

```
public void entradaAlunoNaSala(SalaEstudo salaEstudo, Utilizador aluno){

    if(salaEstudo == null || aluno == null) return;

    if(salaEstudo.SalaEstudoDentroDoHorario()) {

        if (salaEstudo.addAlunoSala(aluno) == true)
            System.out.println("Foi efetuada uma nova entrada de aluno na sala!\n");
        else
            System.out.println("Nao foi efetuada a entrada do aluno na sala\n");
    }else{
        System.out.println("Sala de estudo encontra-se fechada");
    }

}
```

Figura 31: Função para a entrada do aluno na sala de estudo.

```
public boolean addAlunoSala(Utilizador aluno){

    if(aluno.getDentroSaladeEstudo() == 1) {
        return false; //aluno já se encontra dentro de uma sala de estudo
    }

    if(this.nLugares >=1) {
        assert this.AlunosList != null;
        this.AlunosList.add(aluno);
        this.nLugares--;
        return true;
    }else
        return false;
}
```

Figura 32: Função para registar o aluno na sala de estudo.

De modo a registar a entrada de um aluno numa sala de Estudo, como se observa na Figura 31 e Figura 32, irá ser necessário verificar se a mesma existe, bem como se o aluno existe caso seja afirmativo verifica-se se a sala de estudo se encontra aberta, para tal procedeu-se à função já mencionada *SalaEstudoDentroDoHorario ()*, caso esta função retorne positivo utiliza-se outra função existente na classe *SalaEstudo, addAlunoSala ()*, a qual tem como objetivo registar a entrada do aluno na sala, garantindo que existe lugar para o mesmo, bem como, que o aluno não está a frequentar outra sala de estudo.

Por outro lado, para registar a saída do aluno na sala procedeu-se à utilização da função *saidaAlunoNaSala ()*. Esta função verifica se existe sala de estudo bem como o aluno, caso o mesmo se verifique executa a função *remAlunoSala ()*, que irá registar a saída do mesmo da sala de estudo, aumentando assim o número de lugares livres.

10.Conclusão

O trabalho proposto pela unidade curricular de Engenharia de Software I foi realizado com sucesso, uma vez que, todos os objetivos inicialmente definidos foram cumpridos, apesar de haver inicialmente dificuldades na implementação do código nas novas funcionalidades.

Com a realização deste projeto, compreendeu-se melhor que com a criação de User Story's, a entrega de um sprint ou do próprio é realizada no prazo ou até antes do prazo, para além de facilitar a execução do projeto.

A criação dos diferentes diagramas facilitou a realização do presente projeto, uma vez que definiu desde início a implementação das diferentes classes e dos respetivos métodos e atributos, bem como, os elos de ligação com as outras classes.

Por fim, o trabalho superou os objetivos inicialmente pretendidos. Para este bom sucesso do projeto, contribuiu a boa dinâmica de grupo, pois trabalharam todos os elementos de igual forma e com o mesmo objetivo.