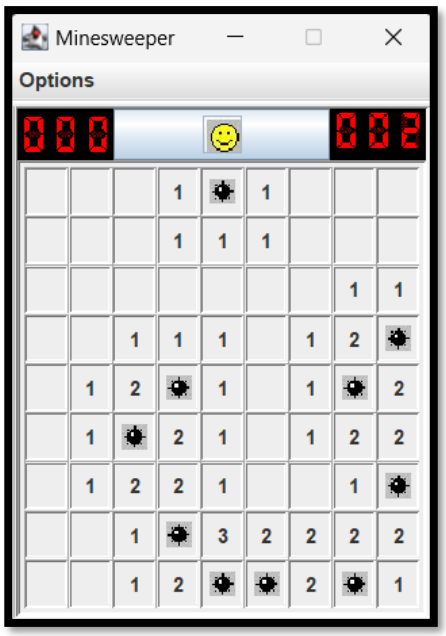
ISEL
Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

Programação III
Licenciatura em Engenharia Informática, Redes e Telecomunicações

Minesweeper

3º Trabalho Prático



Trabalho realizado por:

João Evaristo, n.º 51730

Jordão Bruno, n.º 51593

Docente: Manuela Sousa

1º Semestre letivo 2024/2025
4 de janeiro de 2025

RESUMO

O relatório descreve o desenvolvimento de uma aplicação gráfica do jogo ***Minesweeper***, realizada no âmbito da unidade curricular de Programação III. O objetivo principal foi consolidar os conhecimentos adquiridos durante o semestre, aplicando conceitos de programação orientada a objetos, como **herança**, **polimorfismo** e **programação orientada por eventos**, bem como a utilização de **estruturas de dados** e ***Streams*** da ***API Java***.

O jogo ***Minesweeper*** consiste num tabuleiro de **NxM** quadrículas, onde o jogador deve identificar a localização de minas escondidas, evitando destapá-las. A aplicação foi desenvolvida com base numa hierarquia de classes fornecida, que inclui as classes principais ***GameMinesweeper***, responsável pela lógica do jogo; ***MinesweeperFrame***, que implementa a interface gráfica e as interações com o utilizador e subclasses de ***Cell***, que representam as células individuais do tabuleiro.

A lógica do jogo é gerida pela classe ***GameMinesweeper***, que controla o estado do tabuleiro, verificando as condições de vitória ou derrota e processando as ações do jogador, como destapar células ou colocar bandeiras. As células do tabuleiro são armazenadas numa matriz bidimensional e cada uma delas contém informações sobre o estado, o conteúdo e a posição no tabuleiro. Adicionalmente, foi criada uma variável booleana ***isGameOver***, essencial para evitar interações após o fim do jogo.

A interface gráfica é implementada na classe ***MinesweeperFrame***, que apresenta o tabuleiro como uma grelha de elementos interativos. O utilizador pode clicar nas células para as descobrir ou marcar com bandeiras. A interface também inclui funcionalidades como um cronómetro, mensagens de vitória ou derrota e a possibilidade de iniciar novos jogos. Para capturar as ações do utilizador, foram utilizados ***MouseListener*** para cliques do rato e ***GameListener*** para notificar a interface sobre alterações no estado do jogo.

Outro ponto destacado no relatório é a implementação da gestão de recordes dos jogadores, realizada pela classe ***LeaderBoard***. Esta classe utiliza uma estrutura de dados baseada em um ***HashMap*** para associar níveis de dificuldade a listas de recordes. Os dados são persistidos num ficheiro de texto, permitindo que os melhores tempos sejam armazenados entre sessões de jogo. A classe ***PlayerRecord*** foi criada para encapsular os dados de cada jogador, como o nome e o tempo, e para permitir a ordenação eficiente dos recordes.

O relatório também reflete sobre os conceitos aplicados durante o desenvolvimento. A herança e o polimorfismo foram fundamentais para criar uma estrutura modular e extensível, como evidenciado pela especialização da classe ***Square*** para representar células gráficas. A utilização de estruturas de dados e ***Streams*** simplificou operações como ordenação e manipulação de registos de tempos. Já a programação orientada por eventos garantiu uma interação fluida entre o utilizador e o jogo, permitindo que as ações do jogador fossem processadas de forma imediata e refletidas na interface gráfica.

Em conclusão, o relatório avalia que o trabalho alcançou os objetivos propostos, consolidando os conhecimentos de programação orientada a objetos e destacando a importância da modularidade e da organização do código no desenvolvimento de software interativo. Além disso, a inclusão de funcionalidades adicionais, como a gestão de recordes, contribuiu para melhorar a experiência do utilizador e reforçou a aprendizagem prática dos conceitos estudados ao longo do semestre.

ÍNDICE

ÍNDICE DE FIGURAS

1. INTRODUÇÃO

1.1. CONTEXTUALIZAÇÃO

Este documento apresenta o relatório referente ao desenvolvimento de uma aplicação gráfica do jogo ***Minesweeper***, desenvolvido no âmbito da unidade curricular de Programação III do curso de Engenharia Informática, Redes e Telecomunicações. O trabalho consiste na implementação de um jogo interativo que incorpora elementos de herança, polimorfismo, estruturas de dados e interfaces funcionais, com o objetivo de consolidar os conhecimentos adquiridos ao longo do semestre.

O jogo ***Minesweeper*** é jogado num tabuleiro de **NxM** quadrículas, sendo **N e M** dependentes do nível de dificuldade escolhido. O objetivo principal é identificar corretamente a localização das minas escondidas, evitando que o jogador as destape inadvertidamente. A aplicação foi desenvolvida a partir de uma hierarquia de classes base fornecida, que inclui as classes principais ***GameMinesweeper***, ***MinesweeperFrame*** e as subclasses de ***Cell***.

1.2.OBJETIVO

Os objetivos deste trabalho são apresentados em duas categorias: os gerais, que abrangem o propósito amplo do estudo, e os específicos, que detalham as metas práticas a serem alcançadas.

1.2.1. OBJETIVO GERAL

O objetivo geral deste trabalho é implementar uma aplicação gráfica interativa, respeitando os princípios de programação orientada a objetos e utilizando as funcionalidades providenciadas pela hierarquia de classes fornecida.

1.2.2. OBJETIVOS ESPECÍFICOS

- Desenvolver os componentes principais do jogo, garantindo uma interação eficiente entre eles;
- Implementar e gerir os ***listeners*** necessários para a interatividade do jogo, assegurando que respondem adequadamente aos eventos;
- Criar estruturas de dados para armazenar e gerir os tempos dos jogadores;
- Implementar funcionalidades adicionais como o controlo do estado do jogo e a exibição das minas após o fim da partida;
- Explorar os conceitos de herança, polimorfismo, interfaces funcionais, ***Streams*** e programação orientada por eventos na implementação do jogo;
- Documentar detalhadamente todas as soluções encontradas, explicando o raciocínio por trás de cada decisão.

1.3. Organização do Documento

Este documento está organizado da seguinte forma:

Capítulo 2: Desenvolvimento e Análise do Jogo: Este capítulo explora detalhadamente como os principais componentes do jogo ***Minesweeper*** interagem para criar uma experiência coesa e funcional. Primeiramente, aborda-se a estrutura e a função de cada componente, nomeadamente ***GameMinesweeper***, ***MinesweeperFrame*** e as subclasses de ***Cell***. Em seguida, são analisados os métodos mais relevantes que regem o funcionamento desses componentes, desde a lógica de inicialização e gestão do estado do jogo até à exibição gráfica e resposta às ações do utilizador. Finalmente, é discutida a integração entre esses elementos e como contribuem para a mecânica geral do jogo. Este capítulo engloba as secções:

- 2.1 Interação entre os Componentes Principais do Jogo;
- 2.2 Funcionamento dos ***Listeners*** do Jogo;
- 2.3 Estruturas de Dados e Métodos Implementados.

Capítulo 3: Escolhas e Justificativas Técnicas: Neste capítulo, justificam-se as escolhas realizadas para a organização dos elementos na hierarquia de classes fornecida. Também são destacados os métodos mais importantes que suportam essas decisões, explicando o impacto das mesmas na estrutura do jogo. Este capítulo engloba as secções:

- 3.1 Justificação das Escolhas na Hierarquia de Classes;
- 3.2 Métodos e Decisões Relevantes.

Capítulo 4: Reflexão sobre os Temas Estudados: Este capítulo apresenta uma reflexão sobre os conceitos estudados ao longo do semestre que foram aplicados no desenvolvimento do jogo, como herança, polimorfismo, utilização de estruturas de dados, ***Streams*** e programação orientada por eventos. Destacam-se as vantagens resultantes dessas aplicações. Este capítulo engloba as secções:

- 4.1 Aplicação de Herança e Polimorfismo;
- 4.2 Utilização de Estruturas de Dados e ***Streams***;
- 4.3 Benefícios da Programação Orientada por Eventos.

Capítulo 5: Considerações Finais: No último capítulo, avalia-se o trabalho realizado, incluindo uma discussão sobre os objetivos alcançados e as dificuldades encontradas.

2. Desenvolvimento e Análise do Jogo

Este capítulo explora detalhadamente como os principais componentes do jogo ***Minesweeper*** interagem. Primeiramente, aborda-se a estrutura e a função de cada componente, nomeadamente ***GameMinesweeper***, ***MinesweeperFrame*** e as subclasses de ***Cell***. Em seguida, são analisados os métodos mais relevantes que regem o funcionamento desses componentes, desde a lógica de inicialização e gestão do estado do jogo até à exibição gráfica e resposta às ações do utilizador. Finalmente, é discutida a integração entre esses elementos e como contribuem para a mecânica geral do jogo, garantindo uma interação eficiente.

2.1. Interação entre os Componentes Principais do Jogo

O jogo ***Minesweeper*** é composto por três componentes principais que interagem entre si para garantir tanto o funcionamento interno da lógica do jogo como a sua representação gráfica. Estes componentes são: ***GameMinesweeper***, ***MinesweeperFrame*** e ***Cell***.

GameMinesweeper desempenha o papel central na lógica do jogo. Este componente é responsável por gerir o estado do tabuleiro, que é representado como uma matriz de objetos do tipo ***Cell***. Para cada jogo, define o número de linhas, colunas e minas, controla a execução das ações do jogador, como descobrir células ou colocar/remover bandeiras. Além disso, este componente é responsável por verificar condições de vitória ou derrota e notificar por exemplo o ***MinesweeperFrame***, sempre que ocorre uma alteração relevante no estado do jogo.

MinesweeperFrame é a interface gráfica do jogo. Este componente apresenta o tabuleiro do jogo como uma grelha de elementos gráficos (instâncias da classe ***Square***), que correspondem às células geridas por ***GameMinesweeper***. É através do ***MinesweeperFrame*** que os jogadores interagem com o jogo, clicando nas células para as descobrir ou colocando/removendo bandeiras com o rato. O ***MinesweeperFrame*** também é responsável por atualizar a interface sempre que o estado do jogo muda, como no caso de uma célula ser descoberta ou de uma bandeira ser colocada. Adicionalmente, gere funcionalidades como o cronómetro, o botão de reiniciar e a exibição de mensagens de vitória ou derrota.

A classe ***Cell*** representa uma única célula do tabuleiro e armazena informações importantes como o seu estado (coberta ou descoberta), se contém uma mina ou bandeira, e o número de minas adjacentes. Este componente é manipulado pelo ***GameMinesweeper*** durante o progresso do jogo e fornece os dados necessários para que o ***MinesweeperFrame*** atualize a representação gráfica da respetiva célula.

A interação entre estes componentes segue uma arquitetura que separa claramente a lógica do jogo (modelo), a interface gráfica (visualização) e os eventos de interação do utilizador (controlador). Esta separação facilita a manutenção do código e permite a expansão do jogo de forma modular. Quando o jogador interage com o tabuleiro através do ***MinesweeperFrame***, os eventos são encaminhados para ***GameMinesweeper***, que processa a ação e atualiza o estado das ***Cell*** envolvidas. Posteriormente, o ***GameMinesweeper*** notifica o ***MinesweeperFrame*** sobre as alterações, garantindo que a interface seja atualizada de forma consistente com o estado atual do jogo.

2.2. FUNCIONAMENTO DOS *LISTENERS* DO JOGO

O funcionamento dos ***listeners*** no jogo ***Minesweeper*** é essencial para a comunicação entre a lógica do jogo e a interface gráfica. Os dois principais tipos de ***listeners*** utilizados são o ***GameListener*** e o ***MouseListener***.
GameListener é uma interface implementada por componentes que desejam ser notificados sobre alterações no estado do jogo. O ***MinesweeperFrame***, por exemplo, implementa esta interface para atualizar a interface gráfica sempre que o estado do jogo muda. O ***GameMinesweeper*** gere a lista dos ouvintes registados através do método ***addGameListener(GameListener listener)***, que é chamado durante a inicialização do jogo. Sempre que ocorre uma alteração relevante, como a descoberta de uma célula ou a colocação de uma bandeira, o ***GameMinesweeper*** invoca os métodos apropriados do ***GameListener***, como ***CellUncover*** ou ***bannerPlaced***. Desta forma, garante-se que a interface gráfica é sincronizada com o estado do jogo.
MouseListener é utilizado para capturar eventos de interação do utilizador, como cliques do rato nas células do tabuleiro. Cada instância da classe ***Square***, que representa graficamente uma célula do tabuleiro, possui um ***MouseListener*** associado. Este listener é registado no momento em que o tabuleiro gráfico é criado, através do método ***makeSquare***(int l, int c) no ***MinesweeperFrame***. Quando o utilizador clica numa célula, o ***MouseListener*** chama os métodos correspondentes no ***GameMinesweeper***, como ***uncover(int l, int c)*** para descobrir uma célula, quando clicamos com o botão esquerdo, ou ***turnBanner(int l, int c)***, quando clicamos no botão direito para colocar ou remover uma bandeira. Estes métodos processam a ação e atualizam o estado do jogo, desencadeando notificações para os ***GameListeners***.
A interação entre os ***listeners*** garante que o jogo responda de forma fluida às ações do utilizador e que a interface gráfica reflete com precisão o estado atual do jogo.

2.3. ESTRUTURAS DE DADOS E MÉTODOS IMPLEMENTADOS

- **Estruturas de Dados**

O estado do tabuleiro no jogo ***Minesweeper*** é armazenado numa matriz bidimensional de objetos da classe ***Cell***, onde cada célula contém três componentes principais: o estado, que indica se a célula está coberta, descoberta ou marcada com uma bandeira; o conteúdo, que define se a célula contém uma mina ou o número de minas adjacentes; e a posição, representada pelos índices de linha e coluna na matriz. A manipulação desta matriz é realizada diretamente pela classe ***GameMinesweeper***, que gere as ações de revelar células ou colocar bandeiras, garantindo que o estado do jogo seja atualizado de forma consistente e que as condições de vitória ou derrota sejam verificadas.
Adicionalmente, a classe ***GameMinesweeper*** utiliza a variável ***isGameOver*** que foi uma variável que criamos, pois, consideramos crucial na lógica do jogo. Esta variável é do tipo ***boolean*** e é inicializada como ***false*** no início de cada partida e alterada para ***true*** quando o jogo termina, seja por vitória ou derrota. A sua principal função é impedir qualquer interação adicional com o tabuleiro após o fim do jogo. Métodos ***como revealCell(int row, int col)*** e ***placeFlag(int row, int col)*** verificam o estado de ***isGameOver*** antes de executar qualquer operação.
Relativamente ao registo dos tempos dos jogadores, a classe ***LeaderBoard*** utiliza um ***HashMap*** onde as chaves correspondem aos níveis de dificuldade do jogo ("***Beginner***", "***Intermediate***", "***Advanced***") e os valores são

listas de objetos ***PlayerRecord***. Cada ***PlayerRecord*** guarda o nome do jogador e o tempo total gasto para concluir o jogo, expresso em segundos. Esta estrutura permite um acesso rápido aos registos por nível, enquanto as listas associadas facilitam a ordenação e manipulação dos tempos. Os registos são armazenados num ficheiro de texto, garantindo a persistência dos dados entre diferentes sessões de jogo, com cada linha do ficheiro a seguir um formato contendo o nome do jogador, o nível de dificuldade e o tempo utilizado para completar o jogo.

Uma adição relevante à lógica do jogo é o método criado ***determineLevel()***, que avalia as dimensões do tabuleiro (***rows*** e ***cols***) e o número de minas configuradas para determinar o nível correspondente. Se as dimensões forem 9x9 com 10 minas, o nível é classificado como "***Beginner***"; para 16x16 com 40 minas, como "***Intermediate***"; e para 16x30 com 99 minas, como "***Advanced***". Qualquer configuração fora destes padrões é classificada como "***CustomMode***". Esta função é utilizada principalmente nos métodos ***playerWin()*** e ***playerLose()*** para determinar o nível de jogo atual, permitindo atualizar corretamente os registos na ***LeaderBoard***.

Durante a vitória, o método ***playerWin()*** chama ***determineLevel()*** para identificar o nível, atualiza os registos de acordo com o tempo gasto pelo jogador e exibe uma mensagem com o seu tempo, o melhor tempo registado e os melhores tempos globais. De forma semelhante, no caso de derrota, o método ***playerLose()*** utiliza a mesma função para obter o nível e apresentar ao jogador os melhores tempos. Essa funcionalidade melhora a organização dos dados, assegurando que cada tempo é registado no nível correto e que os jogadores podem visualizar os resultados relevantes de forma clara.

- **Armazenamento dos dados**

Os registos de tempos dos jogadores são armazenados num ficheiro de texto, permitindo a manutenção dos dados entre as execuções do jogo. O formato do ficheiro é padronizado, com cada linha representando o nome do jogador, o nível de dificuldade e o tempo gasto para completar o jogo (em minutos:segundos).

- **Métodos Implementados**

A classe ***GameMinesweeper*** implementa vários métodos para gerir o estado do jogo. O método ***revealCell(int row, int col)*** é responsável por revelar o conteúdo de uma célula. Se a célula for vazia, o método revela recursivamente as células adjacentes. O método ***placeFlag(int row, int col)*** alterna o estado da bandeira de uma célula, permitindo ao jogador marcar ou desmarcar a célula. O método ***checkWinCondition()*** verifica se todas as células não-minadas foram descobertas, determinando assim se o jogador venceu.

Em relação à atualização e classificação de registos, o método ***updateRecord(String level, String playerName, int time)*** permite adicionar ou atualizar o tempo de um jogador num determinado nível de dificuldade. O método ***sortAndTrimRecords()*** ordena as listas de tempos de cada nível em ordem crescente e mantém apenas os 10 melhores.

A leitura e escrita de registos é gerida pelos métodos ***loadRecords()*** e ***saveRecords()***. O primeiro lê os registos do ficheiro no início do jogo, ignorando entradas mal formatadas para garantir a integridade dos dados. O segundo grava os registos em memória de volta no ficheiro (***LeaderBoard.txt***), garantindo que os dados sejam

mantidos.

Por fim, a consulta dos registos é feita através dos métodos ***getTopRecords(String level)*** e ***getPlayerRecord(String level, String playerName)***. O primeiro retorna os 10 melhores tempos para um nível de dificuldade, formatados para exibição, enquanto o segundo recupera o melhor tempo de um jogador específico para um nível. Na classe ***MinesweeperFrame*** criamos uma função denominada ***startTimer***.

A função ***startTimer*** foi criada com o objetivo de gerir o cronómetro do jogo ***Minesweeper***, garantindo que o tempo de jogo é iniciado, atualizado e exibido corretamente na interface gráfica.

Quando chamada a função, verifica inicialmente se um cronómetro (*timer*) já está em execução. Caso exista, o cronómetro atual é parado utilizando o método ***stop()***, garantindo que não há interferência entre diferentes instâncias do cronómetro. De seguida, a variável ***initialTime*** é atualizada com o momento atual em milissegundos, utilizando ***System.currentTimeMillis()***. Este valor marca o início da contagem de tempo.

A função então cria uma nova instância da classe ***Timer***, configurada para executar uma ação a cada segundo (1000 milissegundos). A ação associada ao cronómetro calcula o tempo decorrido desde o início da partida subtraindo ***initialTime*** ao tempo atual, convertendo o resultado para segundos. O valor obtido é, por fim, exibido no componente gráfico ***timerDisplay*** através do método ***setValue***. Após a configuração, o cronómetro é iniciado com o método ***start()***.

Esta função é utilizada em vários momentos importantes do jogo para garantir que o cronómetro funcione de forma coerente com o estado do jogo:

1. **Início de um Novo Jogo**: Sempre que o jogador inicia uma nova partida, seja através do botão de reiniciar ou selecionando um nível no menu, a função ***startTimer*** é chamada. Isso assegura que o tempo começa a ser contado desde o início da partida, refletindo corretamente o tempo que o jogador levou para completar o jogo.
2. **Mudança de Nível**: Quando o jogador escolhe alterar o nível de dificuldade, a função é novamente chamada para iniciar o cronómetro, já que um novo tabuleiro é gerado.
3. **Respostas a Eventos do Jogo**: A função é integrada ao fluxo do jogo através do método ***gameStart()*** da classe ***MinesweeperFrame***, que é chamado sempre que um novo jogo é iniciado. Isso garante que o cronómetro está sincronizado com o início efetivo da partida.

```
private void startTimer() {  5 usages
    if (timer != null) {
        timer.stop();
    }
    initialTime = System.currentTimeMillis();
    timer = new Timer( delay: 1000, e -> {long elapsedSeconds = (System.currentTimeMillis() - initialTime) / 1000;
    timerDisplay.setValue((int) elapsedSeconds);});
    timer.start();
}
```

Figura 1- Função ***startTimer***()

➢ Classes criadas:

○ *PlayerRecord*

A classe ***PlayerRecord*** foi criada para organizar e gerir de forma eficiente os dados relacionados aos recordes dos jogadores no ***Minesweeper***. Esta classe é essencial porque encapsula informações fundamentais como o nome do jogador e o tempo que este demorou a completar o jogo, expresso em segundos.

O funcionamento da ***PlayerRecord*** baseia-se numa estrutura simples onde o campo name armazena o nome do jogador, enquanto o campo time regista o tempo, em segundos, que este levou para terminar o jogo. Para garantir uma ordenação eficiente, a classe implementa a interface ***Comparable<PlayerRecord>***, o que permite que os recordes sejam organizados por tempo, destacando automaticamente os melhores desempenhos. Além disso, o método ***toString*** facilita a apresentação dos dados ao formatar as informações como "nome - tempo (minutos:segundos". Por sua vez, o método ***formatTime*** converte o tempo de segundos para o formato "minutos:segundos".

A classe ***PlayerRecord*** é utilizada em várias outras classes do ***Minesweeper***. Na classe ***LeaderBoard***, por exemplo, é utilizada para armazenar e gerir os recordes dos jogadores por nível de dificuldade. Os objetos ***PlayerRecord*** são adicionados, atualizados e ordenados de forma a manter apenas os 10 melhores tempos em cada nível. Métodos como ***updateRecord***, que verifica se o jogador conseguiu melhorar o seu tempo, ou ***getTopRecords***, que apresenta os melhores desempenhos, dependem diretamente desta classe para funcionar corretamente.

Na interface gráfica, implementada na classe ***MinesweeperFrame***, os objetos ***PlayerRecord*** são usados para mostrar os resultados aos jogadores. Quando um jogador termina o jogo, o tempo atual é comparado com os recordes existentes através da ***PlayerRecord***. Caso seja um novo recorde, a informação é atualizada e apresentada. Em caso de vitória ou derrota, as mensagens exibidas incluem o tempo do jogador, o seu melhor desempenho e os melhores tempos globais, tudo formatado e extraído da estrutura de dados ***PlayerRecord***.

```java
package trabs.trab3.anexos.minesweeper;

public class PlayerRecord implements Comparable<PlayerRecord> {  16 usages
    public String name; // Nome do jogador
    public int time;    // Tempo em segundos

    // Construtor para inicializar os valores
    public PlayerRecord(String name, int time) {  2 usages
        this.name = name;
        this.time = time;
    }

    // Compara os tempos para ordenar os recordes
    @Override
    public int compareTo(PlayerRecord other) {
        return Integer.compare(this.time, other.time);
    }

    // Formata a saída do recorde como "nome - tempo (minutos:segundos)"
    @Override
    public String toString() {
        return name + " - " + formatTime(time);
    }

    // Converte tempo de segundos para o formato "minutos:segundos"
    public static String formatTime(int timeInSeconds) {  2 usages
        int minutes = timeInSeconds / 60;
        int seconds = timeInSeconds % 60;
        return minutes + ":" + String.format("%02d", seconds);
    }
}
```

Figura 2- Classe ***PlayerRecord***

- ***LeaderBoard***

A classe ***LeaderBoard*** foi desenvolvida para gerir os recordes dos jogadores no jogo ***Minesweeper***, oferecendo funcionalidades como guardar, carregar, atualizar e exibir os melhores tempos.

A estrutura da classe começa com a definição de constantes e estruturas fundamentais para a sua funcionalidade. A constante ***FILE_NAME*** especifica o caminho do ficheiro onde os recordes são armazenados, garantindo que o ficheiro utilizado seja sempre o mesmo. A constante ***MAX_RECORDS*** estabelece o limite máximo de 10 registos por nível de dificuldade. A estrutura ***records*** utiliza um mapa para

associar cada nível de dificuldade a uma lista de recordes de jogadores.

Um dos aspetos mais importantes da classe é o seu bloco estático. Este bloco é executado automaticamente quando a classe é carregada pela primeira vez e chama a função ***loadRecords()***.

```
// Carrega os recordes assim que a classe é usada
static {
    loadRecords();
}
```

Figura 3- Bloco estático que chama loadRecords()

Esta função verifica se o ficheiro ***LeaderBoard.txt*** existe e, caso positivo, carrega os dados para a estrutura ***records***. Cada linha do ficheiro é processada para extrair o nome do jogador, o nível de dificuldade e o tempo. Caso a linha esteja mal formatada, é ignorada, evitando assim erros. O bloco estático garante que os dados previamente guardados estão sempre disponíveis quando a classe é usada, eliminando a necessidade de inicialização manual.

A função ***saveRecords()*** salva todos os recordes no ficheiro ***LeaderBoard.txt***, organizados por nível de dificuldade. Esta função escreve os recordes num formato consistente e é chamada sempre que há uma atualização, garantindo que os dados permanecem sincronizados com o ficheiro.

```
// Salva os recordes no ficheiro
private static void saveRecords() {  1 usage
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
        for (Map.Entry<String, List<PlayerRecord>> entry : records.entrySet()) {
            String level = entry.getKey();
            for (PlayerRecord record : entry.getValue()) {
                // Escreve os recordes no formato "nome - nível - tempo"
                writer.write( str: record.name + " - " + level + " - " + PlayerRecord.formatTime(record.time) + "\n");
            }
        }
    } catch (IOException e) {
        // Trata erros de gravação no ficheiro
    }
}
```

Figura 4- Função saveRecords()

A função ***updateRecord(String level, String playerName, int time)*** é usada para atualizar ou adicionar o recorde de um jogador num nível específico. Verifica se o jogador já possui um registo para o nível em questão. Caso já exista, o tempo é atualizado apenas se for menor que o atual; caso contrário, um novo registo é adicionado. Após esta operação, os recordes são ordenados e limitados a um máximo de 10 entradas através da função ***sortAndTrimRecords()***, que é chamada internamente. Esta função é invocada na classe ***MinesweeperFrame*** pelo método ***playerWin()***, sempre que o jogador vence o jogo.

A função ***getTopRecords(String level)*** devolve uma lista formatada com os 10 melhores tempos para o nível especificado, apresentando os tempos em minutos e segundos. Este método é utilizado nos métodos ***playerWin()*** e ***playerLose()*** da classe ***MinesweeperFrame***, permitindo que o jogador veja os melhores tempos após o fim de cada partida.

Por fim, a função ***getPlayerRecord(String level, String playerName)*** retorna o melhor tempo de um jogador específico num determinado nível. Caso o jogador não tenha um registo, é exibida uma mensagem apropriada.

A classe ***LeaderBoard*** interage diretamente com a classe ***MinesweeperFrame***, principalmente nos métodos ***playerWin()*** e ***playerLose()***. No método ***playerWin()***, a função ***updateRecord()*** é usada para guardar o tempo do jogador, enquanto a função ***getTopRecords()*** exibe os 10 melhores tempos. Já no método ***playerLose()***, apenas a função ***getTopRecords()*** é utilizada para mostrar os melhores tempos. Estas interações garantem uma

experiência de jogo envolvente, permitindo que o jogador compare o seu desempenho com o de outros jogadores. No caso das figuras seguintes só são mostrados dois jogadores pois são os únicos com recordes nesse nível.

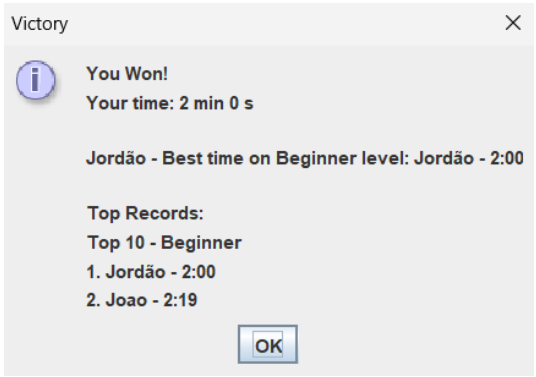


Figura 5- Mensagem de vitória

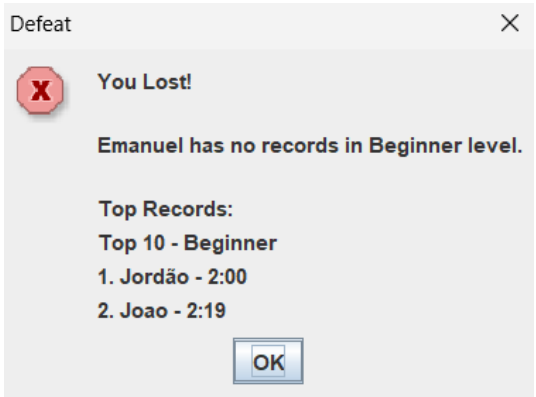


Figura 6- Mensagem de derrota

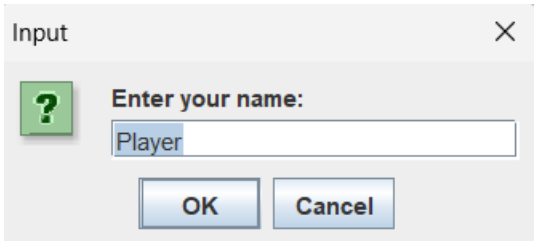


Figura 7- Mensagem para inserir o nome do jogador

3. Reflexão sobre os Temas Estudados

Este capítulo apresenta uma reflexão sobre os conceitos estudados ao longo do semestre que foram aplicados no desenvolvimento do jogo, como herança, polimorfismo, utilização de estruturas de dados, ***Streams*** e programação orientada por eventos. Destacam-se as vantagens resultantes dessas aplicações.

3.1. Aplicação de Herança e Polimorfismo

A classe ***Square*** é uma subclasse de ***JComponent*** e herda as suas funcionalidades básicas de manipulação gráfica. A herança permitiu que a classe fosse especializada para atender às necessidades específicas do jogo, como a exibição de células e a integração com eventos do rato.

Adicionalmente, o polimorfismo é evidente na forma como os diferentes eventos do jogo são tratados. A interface ***GameListener*** define métodos abstratos, como ***playerWin()*** e ***playerLose()***, que são implementados de forma específica na classe ***MinesweeperFrame***.

3.2. Utilização de Estruturas de Dados e *Streams*

A implementação do ***Minesweeper*** faz uso de estruturas de dados para armazenar e processar informações do jogo. A matriz bidimensional de objetos ***Cell*** é um exemplo claro. Ela permite representar o estado do tabuleiro de forma clara e organizada, onde cada célula contém informações sobre o estado, conteúdo e posição.
Adicionalmente, a classe ***LeaderBoard*** utiliza um ***HashMap*** para organizar os tempos dos jogadores, associando níveis de dificuldade (chaves) a listas ordenadas de registos de jogadores (valores). Essa estrutura de dados oferece um acesso rápido e eficiente aos registos, enquanto as listas associadas permitem a ordenação e manipulação dos tempos para exibir os dez melhores resultados.
Os ***Streams*** da ***API Java*** foram utilizados para simplificar operações sobre as listas de registos. Métodos como ***getTopRecords()*** utilizam ***Streams*** para filtrar, ordenar e formatar os dados.

3.3. Benefícios da Programação Orientada por Eventos

A programação orientada por eventos (***event-driven***) é a base da interação entre o jogador e o jogo ***Minesweeper***. Cada ação do jogador, como clicar numa célula ou iniciar um novo jogo, é capturada por eventos que utilizam métodos específicos. Isto é realizado através do ***MouseAdapter*** e ações associadas aos itens de menu.
O modelo de eventos é complementado pela interface ***GameListener***, que comunica mudanças no estado do jogo para a interface gráfica. Por exemplo, quando o jogador descobre uma célula, o jogo notifica a interface gráfica, que atualiza o estado visual do tabuleiro em tempo real. O mesmo ocorre no final do jogo, quando métodos como ***playerWin()*** ou ***playerLose()*** são chamados para exibir mensagens ao jogador e bloquear novas interações.
Esta abordagem traz diversos benefícios, incluindo:

- **Modularidade**: A separação clara entre a lógica do jogo e a interface gráfica facilita a manutenção e evolução do sistema.
- **Responsividade**: As ações do jogador são processadas e refletidas instantaneamente, proporcionando uma experiência de jogo fluida.
- **Extensibilidade**: Novos eventos ou ações podem ser adicionados sem modificar significativamente o código existente.

4. CONCLUSÃO

A realização deste trabalho permitiu consolidar de forma prática os conhecimentos adquiridos ao longo do semestre na disciplina de Programação III, através do desenvolvimento de uma aplicação gráfica interativa do jogo ***Minesweeper***. Durante o trabalho, foi possível explorarmos e aplicar os conceitos fundamentais de programação orientada a objetos, como herança, polimorfismo e programação orientada por eventos, bem como técnicas avançadas relacionadas com a manipulação de estruturas de dados e o uso de ***Streams*** da ***API Java***.

A separação entre lógica, interface gráfica e eventos do utilizador demonstrou ser uma abordagem eficaz para garantir modularidade e facilitar a manutenção do sistema. A implementação de funcionalidades adicionais, como a gestão e persistência dos tempos dos jogadores, reforçou a importância de organizar os dados de forma eficiente.

Em suma, este trabalho não só alcançou os objetivos propostos como também proporcionou uma experiência enriquecedora na implementação de aplicações interativas, consolidando os fundamentos da programação orientada a objetos e destacando a importância de conceitos como modularidade, responsividade e extensibilidade no desenvolvimento de *software*.

REFERÊNCIAS

[1] ORACLE. Java Platform, Standard Edition 8 - API Specification: Class Files. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html. Acedido em dezembro de 2024.

[2] ORACLE. Java Platform, Standard Edition 8 - API Specification: JFrame. Disponível em: https://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html. Acedido em dezembro de 2024.

***PowerPoints* da Docente:**

- chap_09.pdf
- chap_12.pdf
- chap_14.pdf