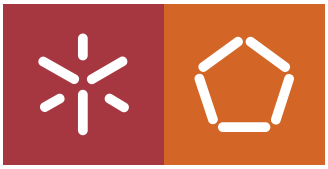


Universidade do Minho
Escola de Engenharia

Análise e Testes de Software

Trabalho Prático

4 de junho 2025

Humberto Gomes	José Lopes	Mariana Cristino
A104348	A104541	A90817

Resumo

Neste projeto de Análise e Testes de *Software*, foram utilizadas diversas técnicas para encontrar falhas numa aplicação Java. Em primeiro lugar, utilizando *white-box testing*, foram escritos testes unitários para os diversos subsistemas da aplicação, cuja cobertura (*branch* e *mutant coverage*) foi analisada. Também foram utilizadas diversas metodologias para geração automática de casos de teste, como o EvoSuite, um sistema construído à mão com recurso ao QuickCheck, e LLMs, cuja cobertura dos resultados também foi analisada. Em relação aos resultados obtidos, concluiu-se que os testes unitários, apesar de serem os que mais exigem recursos humanos, foram os mais capazes de encontrar falhas no código.

1 Gradle

Para automatizar a compilação e testagem deste projeto, foi utilizado o Gradle. No Gradle, a definição de um projeto é feito de forma imperativa, enquanto que no Maven, a ferramenta utilizada nas aulas práticas, é feita de forma declarativa. Deste modo, foi possível implementar diversas funcionalidades no sistema de compilação que teriam sido consideravelmente mais difíceis (ou até impossíveis) de implementar em Maven.

Em primeiro lugar, o projeto suporta todas as tarefas comuns de um projeto Java, como `run`, `build`, `test`, `jar`, *etc.*. No entanto, estas tarefas foram estendidas para suportarem mais funcionalidades, sendo as maiores mudanças as da tarefa `test`. Uma destas mudanças foi a medição automática de cobertura de código, feita com o JaCoCo. Ademais, foi adicionado suporte para mais do que uma *suite* de testes, para ser possível fazer comparações entre a qualidade dos diversos tipos de teste utilizados (testes unitários ou gerados automaticamente). Para trocar de *suite* de testes, a propriedade `testDir` deve ser utilizada, como mostra o exemplo abaixo:

```
gradle test -PtestDir=oldunittests
```

Outra funcionalidade implementada foi a asserção da versão utilizada do Java conforme a tarefa a ser executada. Por exemplo, na geração e execução de testes EvoSuite, o Gradle foi programado para apenas aceitar a versão 8 do JDK, enquanto que a versão 21 deve ser utilizada para as restantes tarefas. Também durante a execução de testes EvoSuite, o Gradle foi programado para automaticamente utilizar o JUnit 4, em vez do JUnit 5.

Muitas tarefas implementadas no Gradle servem para interagir com o projeto Cabal utilizado para geração de testes com o QuickCheck: o Gradle faz *pass-through* de alguns comandos,

garantindo antes que as dependências de cada tarefa foram executadas. Por exemplo, para gerar os teste unitários, é necessário que o projeto Java seja compilado em primeiro lugar, e o Gradle é responsável por garantir que isso acontece.

Por último, também foram implementas outras tarefas, para uso do PIT e para formatação automática de código (feita com recurso ao `clang-format`). Segue-se uma lista das tarefas mais importantes:

- `gradle build` – Compilar os projetos Java e Haskell
- `gradle run` – Executar a aplicação Java (`MakeItFit`)
- `gradle test -PtestDir=...` – Executar uma *suite* de testes JUnit
- `gradle pit` – Executar o PIT para *mutation testing*
- `gradle format` – Formatar o código Java e Haskell

- `gradle generateQuickCheckTests` – Gerar testes com o QuickCheck
- `gradle generateEvoSuiteCheckTests` – Gerar testes com o EvoSuite

- `gradle repl` – Execução do GHCi no projeto Haskell
- `gradle haddock` – Geração de documentação para o projeto Haskell

O Gradle, apesar da sua curva de aprendizagem acentuada, provou-se extremamente capaz de lidar com um projeto complexo, que envolvia o uso de diversas ferramentas, várias versões de Java, e duas linguagens de programação.

2 Testes Unitários

Para o desenvolvimento de testes unitários, utilizou-se *white-box* testing, ou seja, o código fonte a ser testado foi utilizado na criação dos testes unitários. Estes testes utilizam o *runtime* do JUnit 5 para execução automática e suporte para asserções.

2.1 Utilizadores

No subsistema dos utilizadores, os testes unitários revelaram várias falhas no código. Uma das mais relevantes é a incorreção na implementação dos métodos `equals`:

```
if (o == this)
    return true;
if (this instanceof User)
    return false;

User u = (User) o;
return ...
```

Como se pode calcular, quando se comparam dois utilizadores iguais em conteúdo mas de tipos diferentes (ex: um utilizador amador e um profissional), o resultado obtido é o `true`. Para corrigir este erro, o segundo *if-statement* foi substituído pelos seguintes:

```
if (o == null)
    return false;
if (this.getClass() != o.getClass())
    return false;
```

Também foi encontrado um erro em que o método `addActivities` da classe `User` violava o encapsulamento. No entanto, a maioria dos erros encontrados tratavam-se de violações do princípio SLAP (*Single Layer of Abstraction Principle*). Por exemplo, a class `User` não validava os valores passados nos construtores e nos *setters*, sendo esta validação feita pelo `UserManager`, uma *facade* do subsistema. Deste modo, era possível obter um utilizador, executar um *setter* com um valor inválido, e voltar a armazenar o utilizador, colocando o sistema num estado inválido. Neste caso, foi possível melhorar a API sem quebrar o resto do programa, mas a violação deste princípio trouxe outros problemas maiores, como a comparação incorreta de endereços eletrónicos no `UserManager`. Os *emails* devem ser comparados de forma *case-insensitive*. No entanto, devido a violações do princípio SLAP (não havia um tipo `Email` com um método `equals` adequado), esta comparação por vezes era feita de forma *case-sensitive*, um outro erro que também foi detetado por testes e corrigido.

2.2 `Activities`

Durante o processo de desenvolvimento e execução de testes unitários para os componentes relacionados às atividades físicas, foram identificadas diversas inconsistências e oportunidades de melhoria tanto na implementação quanto na cobertura dos testes. Abaixo detalham-se os

principais pontos identificados, as correções aplicadas e a garantia de cobertura total através dos frameworks *JaCoCo* e *PIT Mutation Testing*.

Na classe `Activity.java`, verificou-se a ausência do método `getSpecialization()`, necessário para garantir o contrato polimórfico com as subclasses.

Na classe `Trail.java`:

- O método `setTrailType(int trailType)` não limitava corretamente os valores do tipo de trilho, o que foi identificado através de um *mutant* que sobreviveu no PIT. Logo, foi realizada uma substituição da lógica condicional por:

  ```
  this.trailType = Math.max(TRAIL_TYPE_EASY, Math.min(TRAIL_TYPE_HARD, trailType));
  ```

  Esta abordagem assegura que `trailType` seja sempre mantido entre os limites definidos (`TRAIL_TYPE_EASY` e `TRAIL_TYPE_HARD`).

- O método `caloriesWaste()` permitia valores negativos devido a um erro aritmético, pois utilizava `-` onde deveria ser `+` na fórmula do cálculo de calorias. A fórmula foi atualizada para:

  ```
  return (int) (
    (getDistance() * 0.5 + getElevationGain() * 0.1 + getElevationLoss() * 0.1)
    * index * 0.01);
  ```

  Esta alteração assegura que o cálculo de calorias gastas reflita corretamente o esforço físico baseado nos parâmetros da atividade.

2.3 Planos de Treino

Na classe `TrainingPlan`, a implementação de testes unitários revelou quebras de encapsulamento (método `getActivities`) e tipos de exceção errados. Também se descobriram problemas que, devido a falta de documentação, não se sabe se constituem erros ou a intenção do programador: em várias funções, é feita validação de argumentos (ex.: rejeição de valores nulos), enquanto que noutras não é, não se sabendo se nestes últimos casos o programador se esqueceu de uma exceção, ou se é suposto que, em caso de erro, o código falhe com uma exceção *built-in* do Java (ex.: `NullPointerException`).

Não foi possível testar a função `constructTrainingPlanByObjectives` da classe `TrainingPlanManager` devido à sua aleatoriedade intrínseca. No entanto, esta função seria ideal para testes baseados

em propriedades, visto que o seu propósito é a geração de planos de treino que cumprem um conjunto de condições, que dariam origem a propriedades nos testes.

2.4 *Queries*

Durante os testes unitários realizados ao package `Queries`, foram identificadas falhas lógicas e omissões no código original. Estas falhas foram corrigidas, garantindo a exatidão dos resultados produzidos pelas *queries* e a cobertura total dos testes através dos *frameworks JaCoCo* e *PIT Mutation Testing*.

Na classe `HowManyAltimetryDone`, detetaram-se dois problemas principais:

- No método `executeQuery` com intervalo de datas, não era verificado se a data da atividade se encontrava dentro do intervalo fornecido. Esta lógica foi acrescentada de forma a evitar contagens erradas de altimetria.
- No segundo método `executeQuery`, o cálculo da altimetria estava incorreto, subtraindo a perda de elevação em vez de a somar. A expressão foi corrigida para somar corretamente o ganho e a perda de elevação.

Na classe `MostDoneActivity`, foram feitas as seguintes correções:

- Um dos ramos correspondente à atividade `Repetitions` nunca era executado, pois apresentava uma condição que nunca se iria verificar: a de existir uma atividade de tipo diferente das apresentadas. Este ramo foi, por isso, removido, tendo ficado apenas o `else`.
- O bloco `default` do `switch` representava código morto, pois todas as possibilidades válidas já estavam cobertas. Foi simplificado para refletir apenas casos efetivos e garantir a cobertura total dos testes.

Na classe `QueriesManager.java`, o construtor inicial recebia parâmetros que não eram usados. Estes foram removidos para refletir corretamente a funcionalidade da classe e evitar confusão na sua utilização.

2.5 Utilitários

Durante a execução dos testes unitários ao *package* `Utils`, foi identificado um problema relevante na classe `EmailValidator.java`, responsável por validar endereços de correio eletrónico.

O padrão original utilizado para a validação de emails era demasiado restritivo e falhava em reconhecer múltiplos endereços válidos de acordo com a especificação RFC. Esta limitação resultava em várias falhas nos testes, nomeadamente com emails com carateres especiais ou domínios válidos menos comuns.

Para resolver esta situação, o padrão original:

```
"^[a-zA-Z0-9.+_-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"
```

foi substituído por uma expressão regular mais abrangente que, ainda que não atenda a todos os padrões RFC, é mais competente que a anterior, e permite uma maior correção na hora de avaliar endereços eletrónicos.

2.6 `MakeItFit`

Diversos métodos de atualização de atributos do utilizador estavam incorretamente a chamar `this.userManager.updateUser(user)`, mesmo não sendo necessária tal invocação devido ao padrão de agregação usado (e não composição). Foram aplicadas as seguintes correções:

Métodos corrigidos:

- `updateUserName`
- `updateUserAge`
- `updateUserWeight`
- `updateUserHeight`
- `updateUserBpm`
- `updateUserLevel`
- `updateUserPhone`

Foi removida a chamada de `updateUser(user)`, pois as alterações ao objeto `User` são refletidas automaticamente devido à agregação.

No método `updateUserEmail`, o caso é diferente, pois altera a chave identificadora do utilizador (o email). De modo que, a chamada de `updateUser(user)` foi substituída por:

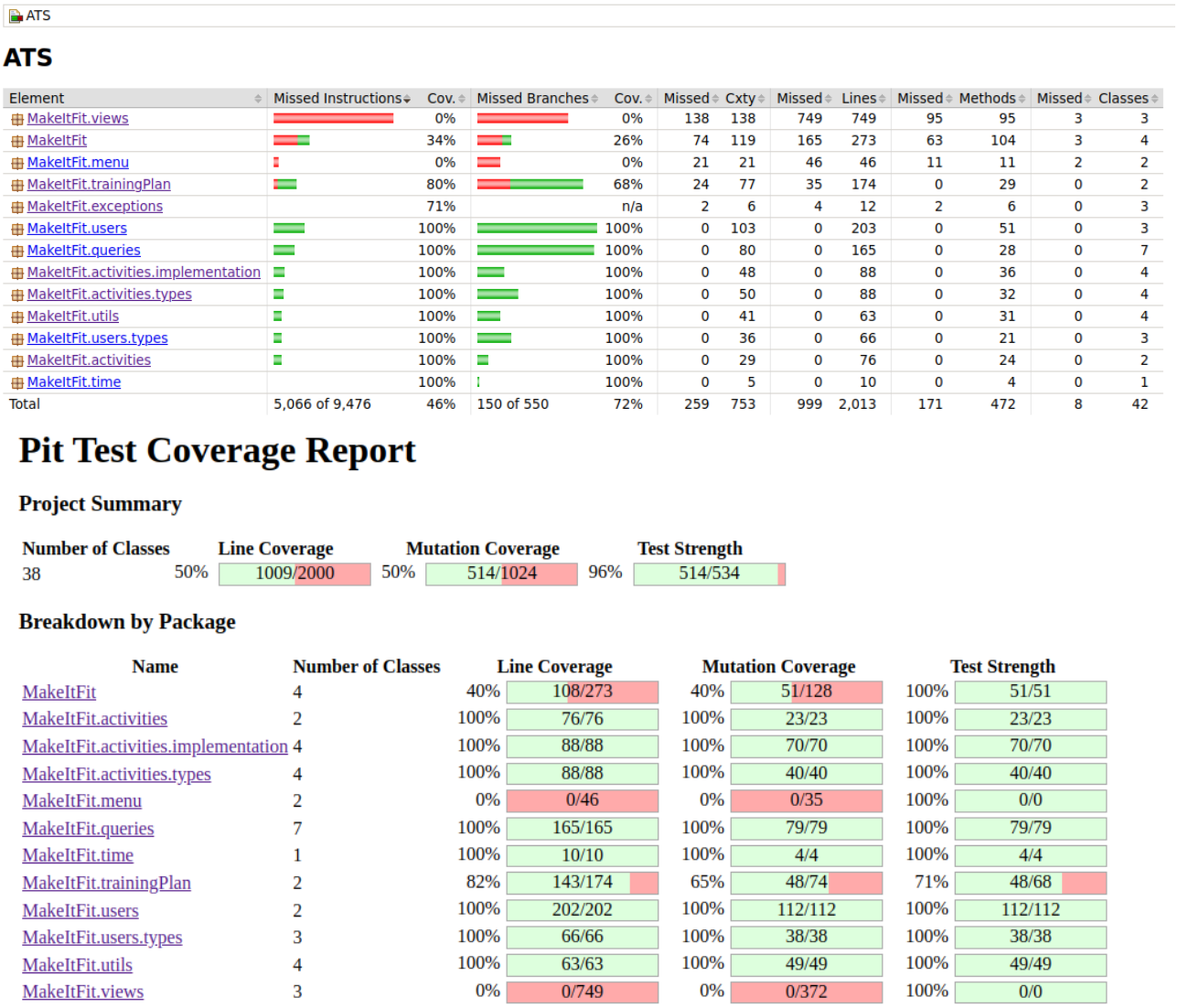
```
this.userManager.removeUserByEmail(email);
this.userManager.insertUser(user);
```

Este erros foram detetados, devido ao desenvolvimento de testes unitários robustos que asseguram a cobertura total do código.

Os testes cobrem condições normais, limites, entradas inválidas e casos extremos, garantindo assim confiabilidade e robustez no comportamento das funcionalidades relacionadas às atividades e à gestão de utilizadores.

2.7 Resultados Obtidos

Para os testes unitários desenvolvidos, apresentam-se as coberturas medidas abaixo:



ATS

ATS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MakeItFit.views		0%		0%	138	138	749	749	95	95	3	3
MakeItFit		34%		26%	74	119	165	273	63	104	3	4
MakeItFit.menu		0%		0%	21	21	46	46	11	11	2	2
MakeItFit.trainingPlan		80%		68%	24	77	35	174	0	29	0	2
MakeItFit.exceptions		71%		n/a	2	6	4	12	2	6	0	3
MakeItFit.users		100%		100%	0	103	0	203	0	51	0	3
MakeItFit.queries		100%		100%	0	80	0	165	0	28	0	7
MakeItFit.activities.implementation		100%		100%	0	48	0	88	0	36	0	4
MakeItFit.activities.types		100%		100%	0	50	0	88	0	32	0	4
MakeItFit.utils		100%		100%	0	41	0	63	0	31	0	4
MakeItFit.users.types		100%		100%	0	36	0	66	0	21	0	3
MakeItFit.activities		100%		100%	0	29	0	76	0	24	0	2
MakeItFit.time		100%		100%	0	5	0	10	0	4	0	1
Total	5,066 of 9,476	46%	150 of 550	72%	259	753	999	2,013	171	472	8	42

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
38	50%	1009/2000	50%	514/1024	96%	514/534

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
MakeItFit	4	40%	108/273	40%	51/128	100%	51/51
MakeItFit.activities	2	100%	76/76	100%	23/23	100%	23/23
MakeItFit.activities.implementation	4	100%	88/88	100%	70/70	100%	70/70
MakeItFit.activities.types	4	100%	88/88	100%	40/40	100%	40/40
MakeItFit.menu	2	0%	0/46	0%	0/35	100%	0/0
MakeItFit.queries	7	100%	165/165	100%	79/79	100%	79/79
MakeItFit.time	1	100%	10/10	100%	4/4	100%	4/4
MakeItFit.trainingPlan	2	82%	143/174	65%	48/74	71%	48/68
MakeItFit.users	2	100%	202/202	100%	112/112	100%	112/112
MakeItFit.users.types	3	100%	66/66	100%	38/38	100%	38/38
MakeItFit.utils	4	100%	63/63	100%	49/49	100%	49/49
MakeItFit.views	3	0%	0/749	0%	0/372	100%	0/0

Figura 1: Coberturas medidas com o Jacoco e com o PIT, respetivamente, nos novos testes unitários.

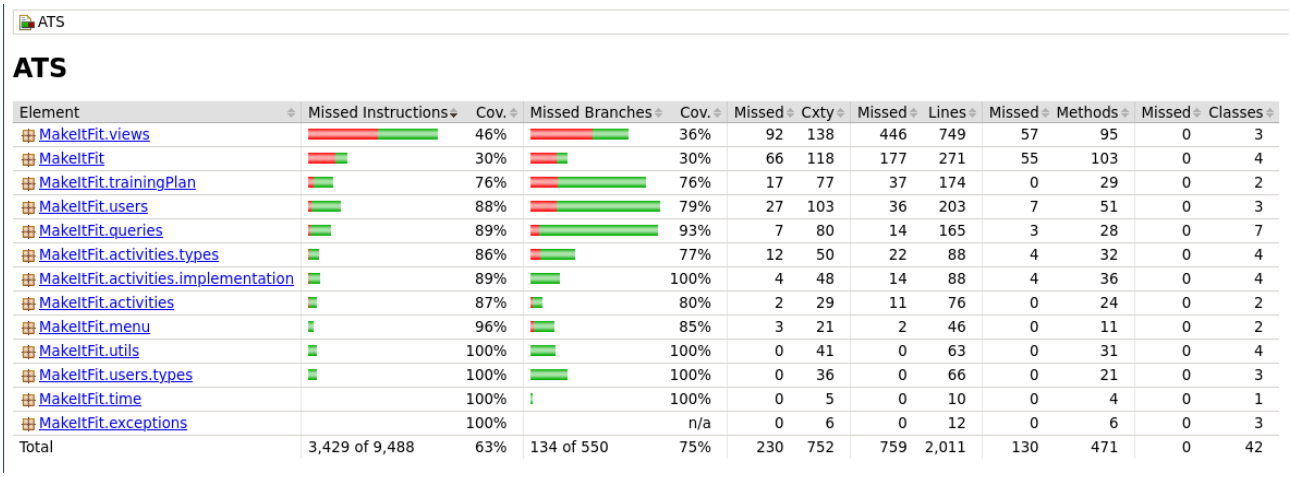
3 Geração de Testes com o EvoSuite

Para a geração de testes com o EvoSuite, foi necessário, em primeiro lugar, converter o projeto para Java 8, visto que versões mais recentes do Java não são suportadas pelo EvoSuite. Também é necessário que o *runtime* destes testes seja executado em Java 8, pelo que não foi possível analisar a *mutation coverage* destes testes, visto que o PIT não funciona nesta versão de Java.

Os testes do EvoSuite apresentam uma boa cobertura do código. No entanto, fazendo uma análise manual dos mesmos, conclui-se que não são bons testes unitários em si, apesar da sua leitura poder ajudar o desenvolvedor a encontrar erros. A título de exemplo, nos subsistemas de utilizadores e atividades, vários testes são feitos com valores inválidos (ex.: altura negativa), o que mostra que a validação destes valores não está a ser feita. Ademais, o EvoSuite deteta

exceções que não estão explícitas no código (ex.: `NullPointerException`), o que pode informar o desenvolvedor das verificações que deve implementar.

Segue-se a cobertura dos testes gerados por EvoSuite:



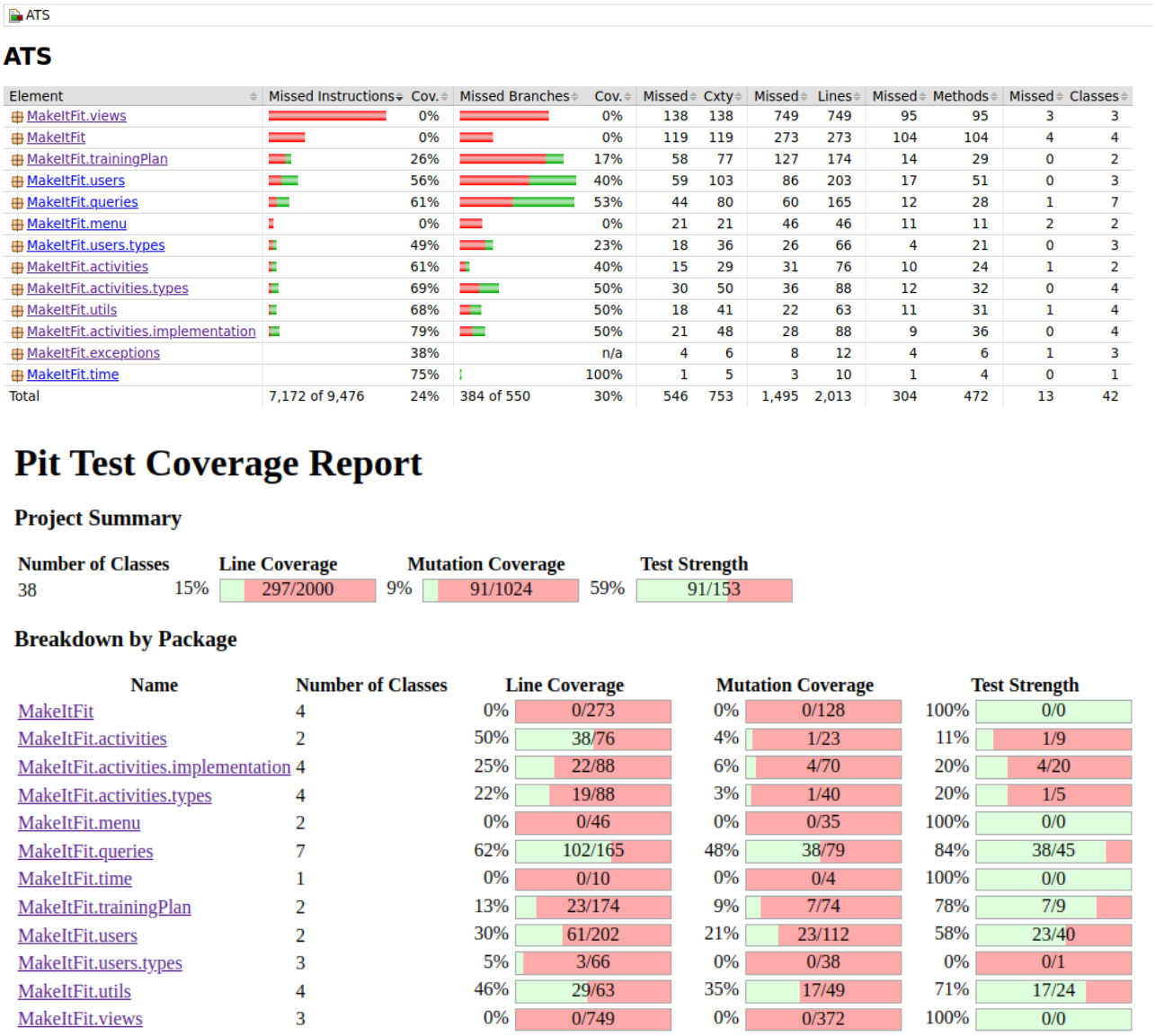
ATS

ATS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MakeItFit.views		46%		36%	92	138	446	749	57	95	0	3
MakeItFit		30%		30%	66	118	177	271	55	103	0	4
MakeItFit.trainingPlan		76%		76%	17	77	37	174	0	29	0	2
MakeItFit.users		88%		79%	27	103	36	203	7	51	0	3
MakeItFit.queries		89%		93%	7	80	14	165	3	28	0	7
MakeItFit.activities.types		86%		77%	12	50	22	88	4	32	0	4
MakeItFit.activities.implementation		89%		100%	4	48	14	88	4	36	0	4
MakeItFit.activities		87%		80%	2	29	11	76	0	24	0	2
MakeItFit.menu		96%		85%	3	21	2	46	0	11	0	2
MakeItFit.utils		100%		100%	0	41	0	63	0	31	0	4
MakeItFit.users.types		100%		100%	0	36	0	66	0	21	0	3
MakeItFit.time		100%		100%	0	5	0	10	0	4	0	1
MakeItFit.exceptions		100%		n/a	0	6	0	12	0	6	0	3
Total	3,429 of 9,488	63%	134 of 550	75%	230	752	759	2,011	130	471	0	42

Figura 2: Cobertura medidas com o Jacoco dos unitários gerados pelo EvoSuite.

Observa-se que estas são consideravelmente superiores às da antiga *suite* de testes unitários:



ATS

ATS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MakeItFit.views		0%		0%	138	138	749	749	95	95	3	3
MakeItFit		0%		0%	119	119	273	273	104	104	4	4
MakeItFit.trainingPlan		26%		17%	58	77	127	174	14	29	0	2
MakeItFit.users		56%		40%	59	103	86	203	17	51	0	3
MakeItFit.queries		61%		53%	44	80	60	165	12	28	1	7
MakeItFit.menu		0%		0%	21	21	46	46	11	11	2	2
MakeItFit.users.types		49%		23%	18	36	26	66	4	21	0	3
MakeItFit.activities		61%		40%	15	29	31	76	10	24	1	2
MakeItFit.activities.types		69%		50%	30	50	36	88	12	32	0	4
MakeItFit.utils		68%		50%	18	41	22	63	11	31	1	4
MakeItFit.activities.implementation		79%		50%	21	48	28	88	9	36	0	4
MakeItFit.exceptions		38%		n/a	4	6	8	12	4	6	1	3
MakeItFit.time		75%		100%	1	5	3	10	1	4	0	1
Total	7,172 of 9,476	24%	384 of 550	30%	546	753	1,495	2,013	304	472	13	42

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
38	15%	297/2000	9%	91/1024	59%	91/153

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
MakeItFit	4	0%	0/273	0%	0/128	100%	0/0
MakeItFit.activities	2	50%	38/76	4%	1/23	11%	1/9
MakeItFit.activities.implementation	4	25%	22/88	6%	4/70	20%	4/20
MakeItFit.activities.types	4	22%	19/88	3%	1/40	20%	1/5
MakeItFit.menu	2	0%	0/46	0%	0/35	100%	0/0
MakeItFit.queries	7	62%	102/165	48%	38/79	84%	38/45
MakeItFit.time	1	0%	0/10	0%	0/4	100%	0/0
MakeItFit.trainingPlan	2	13%	23/174	9%	7/74	78%	7/9
MakeItFit.users	2	30%	61/202	21%	23/112	58%	23/40
MakeItFit.users.types	3	5%	3/66	0%	0/38	0%	0/1
MakeItFit.utils	4	46%	29/63	35%	17/49	71%	17/24
MakeItFit.views	3	0%	0/749	0%	0/372	100%	0/0

Figura 3: Coberturas medidas com o Jacoco e com o PIT, respetivamente, nos antigos testes unitários.

4 Geração de Testes com o QuickCheck

4.1 Geradores

Para ser possível gerar casos de teste aleatórios, é necessário gerar os seus *inputs*. Estes são, frequentemente, instâncias de classes da aplicação a ser testada, pelo que foi necessário implementar geradores para as mesmas. Foram implementados diversos geradores, com o objetivo de criar utilizadores, atividades e planos de treino realistas. A implementação teve atenção a vários detalhes, nomeadamente:

- Geração realista de nomes: Através de uma combinação de nomes próprios e apelidos comuns em Portugal, foram gerados nomes plausíveis como "José Lopes" ou "Humberto Silva".
- Endereços completos e variados: Foram geradas moradas completas, incluindo ruas, números de porta, andares e cidades como "Lisboa", "Porto" e "Braga".
- Telefones válidos: Os números de telemóvel e de telefone fixo são criados com base nos prefixos corretos e comprimentos típicos.
- Emails com domínios reais: Domínios populares como `gmail.com`, `hotmail.com` e `yahoo.com` são utilizados, promovendo realismo, além disso, a frequência de geração de emails é ajustada para refletir a prevalência de cada domínio, sendo `gmail.com` o mais comum.
- Datas válidas e bem distribuídas: Foi utilizado um gerador específico para datas válidas, respeitando os dias de cada mês e anos entre 2001 e 2025.
- Valores fisiológicos credíveis: Os valores gerados para peso, altura, BPM e nível de treino situam-se dentro de intervalos realistas e coerentes.
- Frequência de treino: Utilizadores do tipo `Occasional` e `Professional` incluem um campo de frequência semanal, gerado com base em valores plausíveis.
- Atividades específicas e detalhadas: Cada tipo de atividade possui os seus próprios campos (ex: número de repetições e séries para `PushUp`, distância e velocidade para `Running`, tipo de trilho para `Trail`, entre outros).
- Planos de treino compostos: Foram definidos *training plans* que incluem listas de atividades com número de repetições associado, simulando planos personalizados.

4.2 Geração de Código

Para simplificar a geração de testes, foi criada uma *framework* simples para geração de código Java, contendo diversas funções auxiliares para se gerar código com facilidade. Abaixo, apresenta-se a classe `JavaData`, que pode ser utilizada para transformar um tipo de dados que a implemente numa expressão ou variável Java:

```
class JavaData a where
  javaTypeName :: a -> String
  toJavaExpression :: a -> String

  toJavaVariable :: String -> a -> [String]
  toJavaVariable name obj =
    [javaTypeName obj ++ " " ++ name ++ " = " ++ toJavaExpression obj ++ ";"]
```

Esta classe encontra-se implementada para os vários tipos nativos de Haskell, bem como os tipos para os quais foram escritos geradores. Ademais, também foram implementadas outras funções utilitárias, como `decorateTest`, que encapsula o corpo de um teste num método Java `@Test`, e várias funções para asserções, que geram o código Java para diversas asserções JUnit 5.

4.3 Interoperabilidade com a JVM

Para a geração de certos testes, utilizou-se a mesma metodologia que o EvoSuite: gerar *inputs*, avaliá-los de acordo com o código Java, e finalmente colocar o resultado obtido como valor esperado do teste. Deste modo, os testes deste tipo gerados apenas são viáveis para teste de regressões. No entanto, são necessários para testar as partes da aplicação que não são facilmente modeláveis em Haskell, como as diversas *queries*. Afinal, não é desejado que se reimplemente o programa dado noutra linguagem, para depois comparar as duas implementações.

Para executar código Java, procurou-se inicialmente utilizar a biblioteca JVM, que permite que código Haskell execute *statements* Java. No entanto, não se teve sucesso na sua compilação, pelo que se adotou um método de interoperabilidade com a JVM pouco ortodoxo, porém funcional: sempre que é necessário executar código Java, a `jshell` (um REPL para Java) é executada, os *statements* são enviados para o `stdin` do processo criado, e o `stdout` é analisado para extrair os resultados obtidos. Este método, apesar de pouco eficiente, é possível em qualquer distribuição Java recente, sem ser necessário o *linking* com bibliotecas internas da JVM, como seria caso as bibliotecas JVM e inline-java fossem utilizadas.

4.4 Geração Automática de Testes

O principal objetivo desta abordagem é automatizar a geração de código de teste em Java, assegurando cobertura funcional das principais operações do sistema, tais como:

- Obtenção de atividades associadas a um utilizador (`getActivitiesFromUser`);
- Adição e remoção de atividades (`addActivityToUser`, `removeActivityFromUser`);
- Criação, obtenção, atualização e remoção de planos de treino (`createTrainingPlan`, `getTrainingPlan`, `updateTrainingPlan`, `removeTrainingPlan`);
- Construção de planos de treino com base em objetivos (`constructTrainingPlanByObjectives`);
- Adição de atividades a planos de treino (`addActivityToTrainingPlan`).

A geração de testes é realizada por funções em Haskell que produzem listas de instruções em Java, representando os testes a serem executados. Cada teste é encapsulado num `TestTemplate`, que inclui:

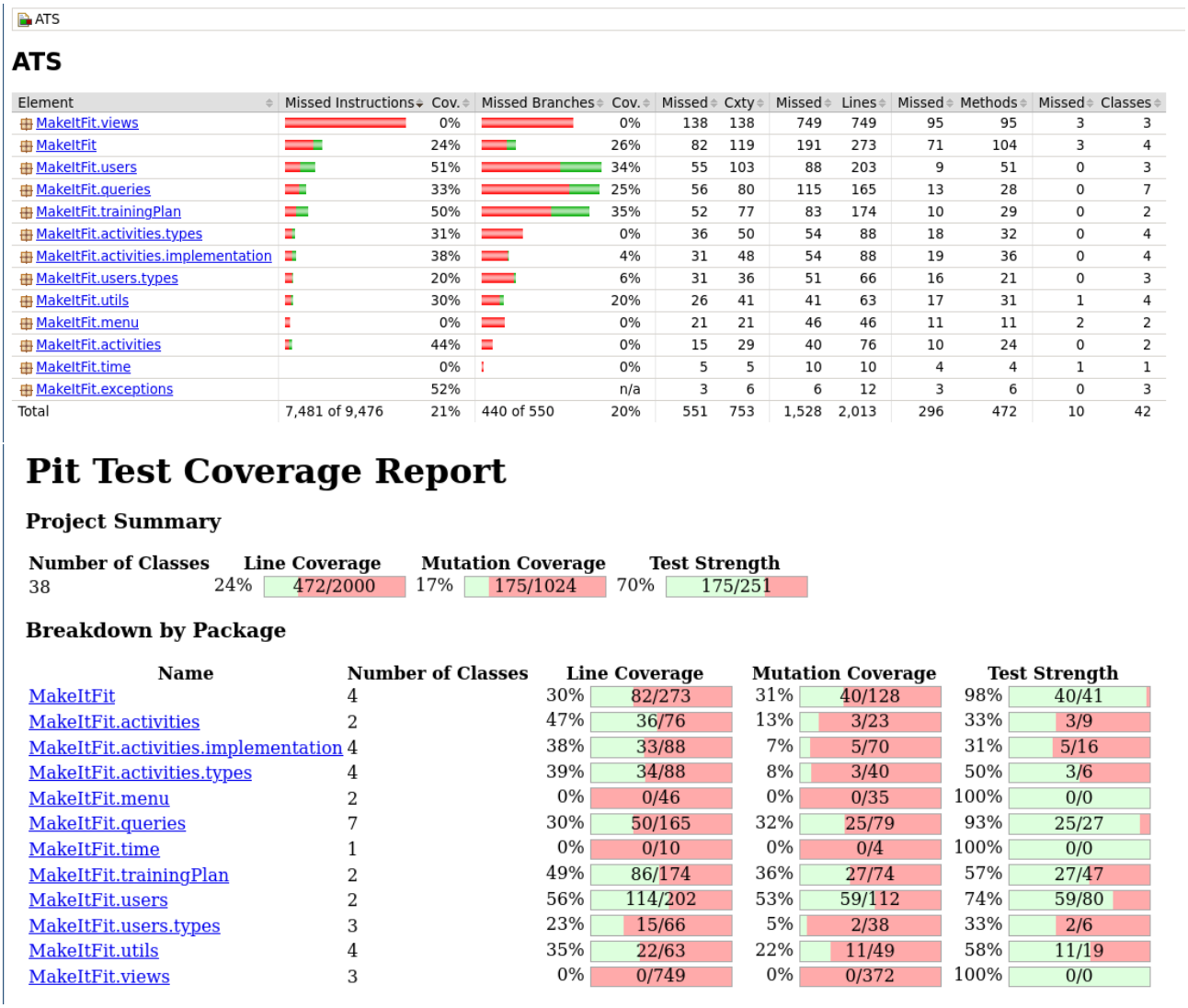
- A configuração inicial do modelo (`MakeItFit`);
- A criação do utilizador com os seus respetivos dados e atividades;
- As chamadas aos métodos da API Java a serem testados;
- Instruções de verificação, como `assertEquals`, `assertTrue` e `assertThrows`.

A estrutura dos testes é parametrizada por geradores, como `Gen User` e `Gen MakeItFitDate`, e utiliza funções auxiliares como `toJavaCreateUserArgs`, `toJavaExpression` e `assertThrows` para gerar código sintaticamente válido em Java.

Além dos testes de fluxo normal, foram gerados testes para cenários de exceção, onde se verifica se os métodos lançam exceções apropriadas como `IllegalArgumentException` ou `EntityDoesNotExistException`. Isto contribui para uma verificação mais rigorosa do comportamento esperado do sistema em casos inválidos.

4.5 Resultados Obtidos

Seguem-se os resultados da cobertura dos testes gerados por QuickCheck:



ATS

ATS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MakeItFit.views		0%		0%	138	138	749	749	95	95	3	3
MakeItFit		24%		26%	82	119	191	273	71	104	3	4
MakeItFit.users		51%		34%	55	103	88	203	9	51	0	3
MakeItFit.queries		33%		25%	56	80	115	165	13	28	0	7
MakeItFit.trainingPlan		50%		35%	52	77	83	174	10	29	0	2
MakeItFit.activities.types		31%		0%	36	50	54	88	18	32	0	4
MakeItFit.activities.implementation		38%		4%	31	48	54	88	19	36	0	4
MakeItFit.users.types		20%		6%	31	36	51	66	16	21	0	3
MakeItFit.utils		30%		20%	26	41	41	63	17	31	1	4
MakeItFit.menu		0%		0%	21	21	46	46	11	11	2	2
MakeItFit.activities		44%		0%	15	29	40	76	10	24	0	2
MakeItFit.time		0%		0%	5	5	10	10	4	4	1	1
MakeItFit.exceptions		52%		n/a	3	6	6	12	3	6	0	3
Total	7,481 of 9,476	21%	440 of 550	20%	551	753	1,528	2,013	296	472	10	42

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
38	24%	472/2000	17%	175/1024	70%	175/251

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
MakeItFit	4	30%	82/273	31%	40/128	98%	40/41
MakeItFit.activities	2	47%	36/76	13%	3/23	33%	3/9
MakeItFit.activities.implementation	4	38%	33/88	7%	5/70	31%	5/16
MakeItFit.activities.types	4	39%	34/88	8%	3/40	50%	3/6
MakeItFit.menu	2	0%	0/46	0%	0/35	100%	0/0
MakeItFit.queries	7	30%	50/165	32%	25/79	93%	25/27
MakeItFit.time	1	0%	0/10	0%	0/4	100%	0/0
MakeItFit.trainingPlan	2	49%	86/174	36%	27/74	57%	27/47
MakeItFit.users	2	56%	114/202	53%	59/112	74%	59/80
MakeItFit.users.types	3	23%	15/66	5%	2/38	33%	2/6
MakeItFit.utils	4	35%	22/63	22%	11/49	58%	11/19
MakeItFit.views	3	0%	0/749	0%	0/372	100%	0/0

Figura 4: Coberturas medidas com o Jacoco e com o PIT, respetivamente, nos testes gerados por QuickCheck.

5 Testes gerados por AI

Durante a fase de testes da aplicação *MakeItFit*, foi também explorada a utilização de inteligência artificial generativa como ferramenta de apoio à criação de testes unitários. Para este fim, recorreu-se à IA *Deepseek* treinado em grandes volumes de código-fonte aberto.

A abordagem consistiu em fornecer à IA os métodos de cada classe a serem testados. Com base nesta informação, a IA gerou *suites* de testes em Java, seguindo a API do `JUnit`. A IA procurou inferir os casos típicos de uso, incluindo:

- Criação e configuração de instâncias da classe `MakeItFit`;

- Invocação de métodos públicos com argumentos válidos e inválidos;
- Utilização de instruções de verificação, como `assertEquals` e `assertThrows`, para validar os comportamentos esperados.

A IA demonstrou competência na estruturação básica dos testes e na utilização adequada da API da aplicação. No entanto, a sua geração baseou-se maioritariamente em inferência sem contexto profundo do domínio da aplicação, o que resultou em testes frequentemente superficiais ou redundantes.

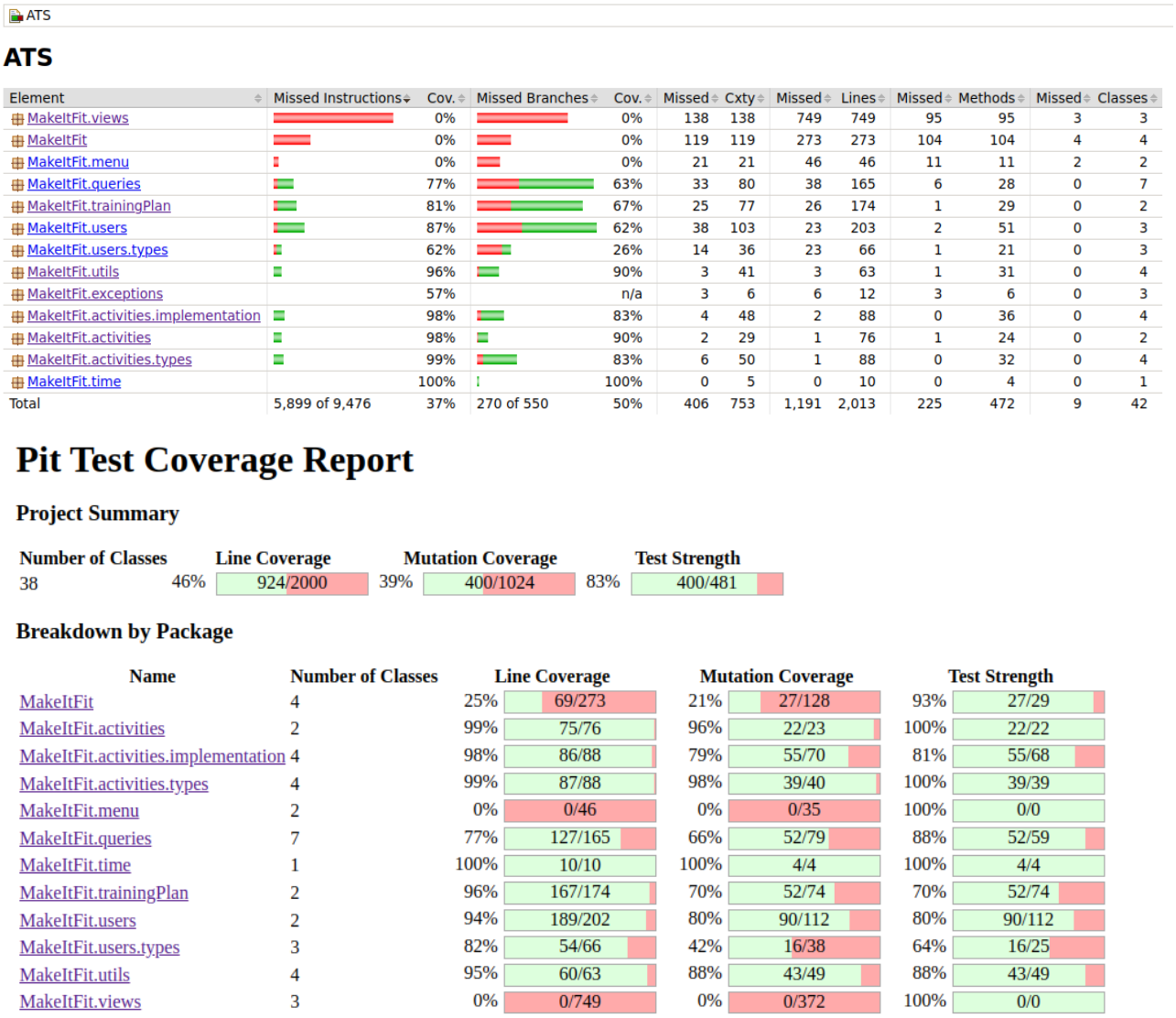
Apesar de gerar código funcional, a cobertura de testes obtida com a abordagem baseada em IA foi limitada. Os testes criados cobriam sobretudo os caminhos felizes (*happy paths*) e não exploravam cenários de exceção ou combinações mais complexas de estado interno da aplicação.

Com o auxílio de ferramentas de análise de cobertura, verificou-se que os testes gerados pela *Deepseek* obtiveram uma cobertura inferior em comparação com os testes escritos manualmente. As principais limitações observadas foram:

- Ausência de testes negativos ou de fronteira;
- Cobertura parcial de ramos condicionais e blocos de exceção;
- Falta de conhecimento contextual sobre as regras de negócio da aplicação.

A utilização de IA como *Deepseek-Coder* mostra-se útil para gerar testes iniciais ou esboços de casos simples, servindo como base para desenvolvimento manual posterior. No entanto, para atingir uma cobertura robusta e garantir a validação de cenários críticos, foi necessário complementar esta abordagem com testes gerados por métodos mais estruturados, como o EvoSuite.

Segue-se a cobertura dos testes unitários gerados por AI:



ATS

ATS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MakeItFit.views		0%		0%	138	138	749	749	95	95	3	3
MakeItFit		0%		0%	119	119	273	273	104	104	4	4
MakeItFit.menu		0%		0%	21	21	46	46	11	11	2	2
MakeItFit.queries		77%		63%	33	80	38	165	6	28	0	7
MakeItFit.trainingPlan		81%		67%	25	77	26	174	1	29	0	2
MakeItFit.users		87%		62%	38	103	23	203	2	51	0	3
MakeItFit.users.types		62%		26%	14	36	23	66	1	21	0	3
MakeItFit.utils		96%		90%	3	41	3	63	1	31	0	4
MakeItFit.exceptions		57%		n/a	3	6	6	12	3	6	0	3
MakeItFit.activities.implementation		98%		83%	4	48	2	88	0	36	0	4
MakeItFit.activities		98%		90%	2	29	1	76	1	24	0	2
MakeItFit.activities.types		99%		83%	6	50	1	88	0	32	0	4
MakeItFit.time		100%		100%	0	5	0	10	0	4	0	1
Total	5,899 of 9,476	37%	270 of 550	50%	406	753	1,191	2,013	225	472	9	42

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
38	46%	924/2000	39%	400/1024	83%	400/481

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
MakeItFit	4	25%	69/273	21%	27/128	93%	27/29
MakeItFit.activities	2	99%	75/76	96%	22/23	100%	22/22
MakeItFit.activities.implementation	4	98%	86/88	79%	55/70	81%	55/68
MakeItFit.activities.types	4	99%	87/88	98%	39/40	100%	39/39
MakeItFit.menu	2	0%	0/46	0%	0/35	100%	0/0
MakeItFit.queries	7	77%	127/165	66%	52/79	88%	52/59
MakeItFit.time	1	100%	10/10	100%	4/4	100%	4/4
MakeItFit.trainingPlan	2	96%	167/174	70%	52/74	70%	52/74
MakeItFit.users	2	94%	189/202	80%	90/112	80%	90/112
MakeItFit.users.types	3	82%	54/66	42%	16/38	64%	16/25
MakeItFit.utils	4	95%	60/63	88%	43/49	88%	43/49
MakeItFit.views	3	0%	0/749	0%	0/372	100%	0/0

Figura 5: Coberturas medidas com o Jacoco e com o PIT, respetivamente, nos unitários gerados por AI.

6 Teste da Camada de Apresentação

Para testar a camada de apresentação, procurou-se convertê-la para uma aplicação Web e testá-la com Selenium. No entanto, devido a falta de tempo, apenas foi possível investigar as melhores ferramentas possíveis para esta tarefa. Procuraram-se ferramentas que permitiriam executar código Java num navegador Web, e encontrou-se o CheerpJ e a TeaVM. O CheerpJ, apesar da sua premissa de executar aplicações sem qualquer alteração, não o é capaz de fazer para aplicações na linha de comandos. Já o TeaVM exigiria ligeiras adaptações da camada de apresentação para o I/O ser feito para uma página Web, ou seja, a implementação de um *terminal emulator*. No entanto, seja qual fosse a ferramenta utilizada, não seria possível medir a cobertura dos testes por falta de ferramentas adequadas, pelo que também se optou por não

proceder com esta solução. Uma forma mais simples de testar a camada de apresentação da aplicação, apesar de não utilizar as ferramentas usadas nas aulas práticas, seria a construção de *strings* de entrada para a aplicação, passadas via `stdin`, e a asserção sobre o conteúdo lido do `stdout`.

7 Conclusão

Neste projeto, foram aplicadas diversas metodologias de testes de *software*, para procurar detetar falhas num programa Java. Este projeto permitiu ao grupo de trabalho aprender sobre diversas ferramentas e, de um modo mais geral, como as integrar corretamente. Em relação aos resultados obtidos, verifica-se que, apesar de haver várias formas de gerar casos de teste de forma automática, a qualidade destes ainda não se aproxima da de testes unitários escritos à mão, pelo que, apesar desta metodologia ser a que mais requer recursos humanos, é que consideramos mais adequada para a deteção de falhas num programa.