Assignment 2

Design of an Experiment

Mestrado em Engenharia Informática
Metodologias Experimentais em Informática

Francisco Gouveia 2023186817, franciscoggouveia@gmail.com
Miguel Pedroso 2019218176, miguelpedroso@student.dei.uc.pt
Rui Santos 2020225542, rpsantos@student.dei.uc.pt

15 de dezembro de 2023

Conteúdo

Resumo

Design e execução de uma experiência sobre a eficácia de *code reviews*. Contextualização teórica sobre os tipos de *code review*, onde decidimos utilizar *over the shoulder reviews* para a execução da mesma. A definição do problema passa pela definição de eficácia aplicado a *code reviews*, citação de estudos que definam as metas para a experiência a executar, o que pode ser definido como qualidade do revisor, e cuidados a tomar com a variabilidade de experiências com humanos. Aborda-se também a definição utilizada para os paradigmas de programação e o que é a dificuldade do código, neste caso o número de linhas e a complexidade ciclomática. Utilizando todo este conhecimento, define-se a experiência, os seus objetivos e as suas metas. O planeamento experimental é abordado de seguida, onde se definiu as variáveis independentes - complexidade do código, a experiência do reviewer e o paradigma da linguagem de programação utilizada. Durante este estudo, temos como objetivo medir o efeito destes fatores na eficácia das *code reviews*, e nos falsos positivos resultantes. Adicionalmente, definiu-se o procedimento utilizado, o protocolo da experiência, e as ferramentas utilizadas para fazer as medições tais como os testes estatísticos. Também explicitamos que fizemos alterações à execução da experiência, após um teste piloto da experiência. No seguinte tópico passamos à análise estatística das amostras produzidas, para cada uma das variáveis independentes escolhidas, tendo antes disto, estudado a normalidade destas através do teste de Kolmogorov-Smirnov. Tendo exposto os nosso resultados, passamos à discussão dos mesmos, nas notas finais, indicando também que estudos pertinentes terceiros poderão efetuar com os dados produzidos da experiência realizada.

1 Introdução

No âmbito da disciplina de Metodologias Experimentais em Informática, foi realizado o segundo *assignment*, relacionado com o design e realização de uma experiência efetuando todos os passos envolventes da mesma. A experiência em si, relaciona-se com a eficácia de *Code Reviews* e principalmente a sua utilidade, como uma ferramenta ativa na deteção de *bugs* em códigos de produção num ambiente empresarial.

2 Contextualização

Uma *code review* pode ser definida como uma atividade de equipa, normalmente efetuada num ambiente empresarial, que ajuda a manter ou aumentar a qualidade de um código produzido, antes de ser introduzido num ramo mais importante do programa. Normalmente, ao introduzir-se este passo na produção de código, não estamos só a tentar detetar que erros ou *bugs* existem. Também, atualmente é considerado uma atividade de alta eficácia para a transmissão de conhecimento (principalmente entre o *reviewer* e o autor do código); o obedecer a *standards* que uma empresa define em relação a segurança, nomeação de variáveis, metodologias preferidas; melhorar uma solução, como não só corrigi-la, entre outros. Como conseguimos evidenciar, é uma atividade que não só cria maior sentido de responsabilidade entre todos os *developers*, como também dá um maior ênfase à ideia de que devemos desenvolver de forma sustentável, tendo em conta que outros colegas poderão enfrentar problemas derivado a uma possível falta de consideração nossa. Apesar de existirem diferentes tipos de *review*, todas têm o mesmo fundamento e objetivo principal adjacente a elas. Justapondo-as, iremos, no próximo subtópico, evidenciar esta mesma afirmação.

2.1 Tipos de *Code Review*

Existem neste momento (e são postas em prática, aliás) diversas metodologias de *code review*, estas dependendo claro, do tipo de programa ou de infraestrutura a que nos referimos. Existe uma separação clara no que toca ao tipo de *reviews* clássicas, e as modernas. A primeira foca-se numa processo pesado e rígido, com três a seis participantes, numa estrutura de reuniões, analisando o código. Cada pessoa nessa reunião terá de a preparar previamente, e cada um terá funções diferentes dentro da reunião, como por exemplo o moderador, o leitor, etc. Atualmente preferem-se metodologias mais baratas, menos trabalhosas e mais flexíveis do que estas, tal como umas variações de *Modern Code reviews*, que iremos mencionar.

Em primeiro lugar, temos as *Tool-Assisted Reviews*, que consistem em utilizar ferramentas automatizadas para verificar os ficheiros com alterações, utilização de comentários, coletar métricas, etc. facilitando assim a identificação de bugs que potencialmente possam existir e por conseguinte, também facilitar a sua integração no código. O problema da utilização deste tipo, advém da possível dependência de uma ferramenta que tem de ser mantida, muito custosa, e continua a envolver a *review* de um humano, mais como um supervisor do que como um desenvolvedor. Outro tipo de *reviews*, mais popularizada através de metodologias agile de desenvolvimento de software é o *Pair Programming* - juntar dois *developers* a colaborar ao mesmo tempo, sendo que um escreve o código e outro analisa-o. Este, devido à sua natureza flexível e casual, torna-se aliciante até pelo seu potencial eficaz de reposta a problemas de uma forma rápida. Porém ao mesmo tempo, a sua maior qualidade é ao mesmo tempo o seu maior defeito. De ter uma natureza tão flexível, podemos ter por vezes uma falta de rigor no que toca à sua utilização e mais grave ainda - falta de métricas associadas à mesma. Em terceiro, e por último, temos as *over the shoulder reviews*. Nesta temos dois *developers*, um autor e um *reviewer* e remotamente através da partilha do ecrã, o autor explica as mudanças que efetuou ao código, enquanto que o *reviewer* faz sugestões e coloca questões. Este é bastante similar ao *Pair Programming*, sendo que a maior diferença é o facto de que o código não está a ser desenvolvido ao mesmo tempo que o *reviewer* o verifica e adiciona o seu *input*. Neste cenário o código já está completo, e o *reviewer* pode é adicionar o seu conhecimento ao mesmo. A maior vantagem deste é a sua flexibilidade tanto na reunião em si (visto que pode ser feito remotamente) tal como na sua rapidez. As desvantagens encontram-se também relacionadas com esta flexibilidade, que é a causadora de uma falta de métrica na *review*, menor velocidade de integração do código, pelo menos comparado ao *Pair Programming* e por último uma falta de objetividade relacionado ao *review* em si - por exemplo, enquanto que num tipo de *review*

mais aproximado ao clássico, problemas são colocados e consequentemente tenta-se aproximar de uma solução, aqui o *review* é feita sem objetivo claro nenhum, e só com o propósito de se "corrigir o código".

Dito isto, e expondo os diferentes tipos de *review*, este tipo, será o alvo principal desta experiência, principalmente pelo facto de, com um ajuste rápido serem as mais indicadas para realizar com algum tipo de experiência simulada. Estes ajustes que o grupo decidiu efetuar, de forma a tornar a experiência mais regularizada e num ambiente mais controlado, irão ser expostas no tópico seguinte.

3 Definição do Problema

A tarefa proposta pelo Docente refere à utilidade de *code reviews*. Ao falarmos de utilidade, podemos-nos extender no significado etimológico da palavra e concluir que para sabermos a sua utilidade, precisamos de saber a sua eficiência, visto que, no campo da estatística principalmente, "algo perde a sua utilidade quando não é eficaz a realizar o que se pressupõe dessa mesma coisa". Assim afirmamos que o problema proposto pelo Docente, refere-se sim à "eficácia de *code inspections*"ou de *code reviews*. O problema que advém desta definição, é da palavra "eficaz", que à primeira vista, tem um significado muito abrangente. Neste contexto o que é que podemos considerar como algo eficaz? A eficácia de algo é medida através da relação entre o efeito da ação e os objetivos pretendidos. Quando nos referimos ao objetivo pretendido, referimos-nos à deteção de *bugs*. Em relação ao efeito da ação, estamos a falar do *code review*. Adicionalmente, para falarmos da eficácia, temos também de adicionar uma medida de esforço. Esta medida de esforço pode-se relacionar ao tempo demorado a realizar a *review*.

3.1 A eficácia de *Code Reviews*

Assim, o grupo defende que a ideia de uma *code review* eficaz é uma tal que consegue detetar o maior número de *bugs* no menor tempo possível ou com o menor esforço possível. Por esta razão, numa primeira instância devemos verificar quais são os tipos de *reviews* que normalmente aparentam ter o maior número de *bugs* detetadas, num menor tempo de tempo.

Segundo uma análise realizada por Casper Jones (1) concluiu-se que a percentagem de *bugs* encontradas por *review* formal é de 60-65%. Será que esta afirmação comprova que para ter *code reviews* eficazes, necessitamos de voltar à metodologia rígida e formal? Não - isto porque mais tarde foi também comprovado para as *reviews* modernas, que elas também apresentam uma taxa de deteção muito similar enquanto que possuem é processos gerais muito mais rápidos e menos caros(2). Isto faz-nos acreditar que de facto, as *reviews* modernas são mais "eficazes- aplicando claro, o sentido que previamente mencionamos.

Dito isto, questões tão ou mais pertinentes aparecem neste momento: "será que numa *review* as taxas de deteção de diferentes tipos de *bugs* são parecidas, ou existe algum tipo que prevalece sobre a outra?".

3.2 Deteção de *Bugs* em *Code Reviews*

Quando mencionamos o conceito de *bugs* ao longo desta exposição, o leitor pode ficar com a impressão errada. Dito isto é imperativo que haja uma definição concreta sobre o assunto. Podemos afirmar que *bugs* num *software* se relaciona não só com problemas funcionais, mas lógicos, requisitos que faltem implementar, riscos de segurança, necessidade de simplificação, problemas estruturais, má nomeação de variáveis ou comentários insuficientes/pouco explícitos.

Outra conclusão, em relação a este problema, refere-se ao facto de que estatísticas sobre o tipo de *bugs* encontradas, resultantes de estudos empíricos, mostram que grande parte dos *bugs* detetados (sensivelmente 75%), são na realidade mais relacionadas com a manutenção/gerenciabilidade de software do que com uma implementação de funcionalidades propriamente ditas(3). Isto ajuda-nos a perceber, que grande parte do foco das *reviews* está sim numa constante manutenção do código, *standards* de nomeação de variáveis, boas práticas de código, e práticas de desenvolvimento sustentável(4), do que em parte, resolver problemas concretos e diretos, como era o esperado - visto que a taxa de deteção é de 15%. Assim podemos concluir que, maioritariamente um *code review* tem mais utilidade quando aplicado no sentido de manter e garantir a qualidade e gerenciabilidade de um código do que consertar ou remediar problemas já existentes. Em ambientes de longos ciclos de desenvolvimento e de produtos longos, é que se consegue perceber a verdadeira utilidade de um *code review*.

Entrando já neste assunto, referente à utilização de *code reviews* em ambientes industriais mais complexos, outra variável importante que devemos ter em conta, e que *a priori* possivelmente deve ter uma influência considerável, é a qualidade do *reviwer*.

3.3 A qualidade do *Reviewer*

A questão essencial deste sub-tópico é objetivamente simples numa primeira instância, mas rapidamente podemos observar que é uma ciência "muito pouco exata", no que toca à metodologia que podemos utilizar para medir a qualidade de um *reviewer*. Isto deve-se ao facto de a experiência se relacionar com o comportamento, raciocínio lógico, e pensamento de um humano, até no processo de redigir o código para ser revisto. Como estas são tarefas intensivas, no que toca à esfera da ação humana, os resultados serão sempre muito variáveis e por conseguinte, pode-se tornar muito difícil quantificar as variáveis que são ou não relevantes para o estudo. Assim o nosso objetivo torna não só por tornar esta, numa experiência realista e significativa, como tentar ao máximo utilizar indivíduos de diferentes esferas de conhecimento, com diferentes habilitações de forma a poder mitigar ao máximo esta variabilidade que pode ser altamente destrutiva. Dito isto, podemos de facto perceber que a qualidade de um *reviewer* num certo espaço de tempo não deve ser tomada como um valor absoluto e finito.

Outro assunto importante, não passa só pela qualidade inerente do revisor em si, mas também como não desgastar o *reviewer* ao ponto em que a sua "qualidade"se deteriore significativamente. Este caso, se verificável, pode alterar com ponderância, os resultados produzidos durante a execução da experiência.

Devido à natureza da atual experiência iremos, por questões de simplicidade assumir três níveis diferentes de perícia e proficiência para uma linguagem de programação - *Beginner*, ou Iniciante; *Intermediate*, ou Intermédio, e por último *Advanced* ou *Expert*. Para chegarmos a esta avaliação utilizamos as notas das Unidades Curriculares das disciplinas respetivas aos paradigmas de Programação, tal como uma auto-avaliação dos próprios reviewers em relação ao que toca à sua capacidade na respetiva linguagem de programação (C e Java). O grupo tem a noção da limitação imposta por esta abordagem mas considera que devida à natureza da experiência, e de forma a não perder o foco principal da mesma, não aprofundar mais o tema.

No que toca ao segundo tema discutido neste tópico seguimos recomendações de diversos documentos encontrados em *websites* fidedignos que expõem regras, métricas e *standards* importantes para as *reviews*, e como manter a consistência de um *reviewer* tal como é verificável em (6) (7). Segundo estudos realizados pela empresa Smartbear a equipas de desenvolvimento da Cisco Systems, o ponto ótimo de linhas de código que os*developers* devem rever está entre duzentas (200) a quatrocentas (400). Quando esta *range* se utiliza temos níveis de eficiência nos termos que foram abordados anteriormente - sensivelmente 65%. Sendo assim, o grupo decide limitar o número de linhas por *review* a sensivelmente quinhentas (500) linhas. Adicionalmente, uma preocupação expressa também pelos Discentes é o tempo das *reviews*. Decidiu-se limitar a experiência a uma hora, de forma a que não se degradasse os níveis de atenção dos *reviewers* e até dos membros do grupo ao monitorizar a mesma. Mesmo assim, decidiu-se implementar uma pausa de dez minutos com marca na terceira revisão. O tempo por código também foi bloqueado e esse não foi revelado ao revisor, por motivos que iremos discutir mais à frente nos próximos subtemas.

Adicionalmente, um outro detalhe que causou preocupação nos discentes relaciona-se com o *IDE* utilizado. Acreditamos que diferentes ambientes de programação podem causar bastante imprevisibilidade na experiência devido a diversas razões: número de linhas não ser bem contabilizado; o utilizador conseguir correr o programa ou ter acesso a certos *warnings* que lhe podem indicar os erros, etc. Desta forma decidiu-se bloquear esta variável estipulando que seria obrigatório utilizar o *Visual Studio Code*.

Como podemos verificar, diversos fatores podem influenciar a qualidade da *review* e por consequência, a qualidade da *review*. Como mencionado anteriormente, a qualidade do *reviewer* e o ambiente em que se encontra são das variáveis mais difíceis de quantificar. Dito isto, temos de ter em conta outros fatores que pode influenciar esta medida (a eficiência do revisor), como é o caso da linguagem utilizada e da complexidade do programa em si.

3.4 Paradigmas e Dificuldade da Programação

Para a realização da experiência, utilizou-se duas linguagens de programação, com dois paradigmas diferentes - C e Java, Programação Procedimental e Programação Orientada a Objetos, respetivamente. Os discentes consideram que é importante ter em consideração este aspeto para ter resultados mais abrangentes e mais consideratívos do que pode ser a experiência do reviewer.

Para quantificar a dificuldade da linguagem de programação utilizou-se duas medidas objetivas: a complexidade ciclomática e o número de linhas de código. Como o número de linhas de código é relativamente o mesmo para todos os programas produzidos (e não se relaciona de todo com a sua dificuldade), decidimos utilizar complexidade ciclomática para ditar a dificuldade do programa. Adicionalmente criamos uma escala (Fácil, Médio, Difícil) para cada um destes programas. Juntando a este facto decidiu-se informalmente utilizar temas importantes de programação para cada um destes programas, sendo estes: estrutura de dados, alocação de memória, listas ligadas, árvores binárias, e *self-balancing binary search tree*. A seguinte tabela expressa as diferentes medidas retiradas das complexidades ciclomáticas medidas utilizando uma extensão do Visual Studio Code - *Codalyze*.

Programa	**Complexidade ciclomática**	**Número de Linhas**
BuggyBackpack1.java	1.35	110
BuggyTree.java	2.16	183
BuggyPharmacy.java	1.17	76
BuggyPyramid.c	2.5	21
BuggySalaries.c	2.75	53
BuggyTennisTournament.c	3.75	73

Tabela 1: complexidade Ciclomática (*com threshold de 1-5*), e número de linhas

É importante denotar que a tabela 1 interpretada unicamente pelos valores nela presentes pode incorrer numa falsa noção de complexidade. Temos também de relacionar os assuntos que é abordado em cada um destes programas, pois, apesar de terem menos complexidade ciclomática, e até puderem ter um número inferior de linha pode ter mesmo assim, uma complexidade que o grupo considere ser maior, devido à inerente dificuldade associada a cada um destes assuntos. Como tal, o grupo decidiu abordar de uma forma muito generalizada os seguintes assuntos (associando também aos ficheiros criados):

- *Buggy Backpack*: Estrutura de dados, listas ligadas e algoritmos de ordenamento.
- *Buggy Tree*: *Self Balacing Binary Search Tree*
- *Buggy Pharmacy*: Lógica, ciclos e condições
- *Buggy Pyramid*: Lógica, ciclos e condições
- *Buggy Salaries*: Alocação Dinâmica, estruturas.
- *buggy Tenis Tournament*: Árvore Binária, leitura de Ficheiros.

Concluindo assim, os discentes do presente trabalho consideram que os programas são representativos de diferentes dificuldades, desafios e temáticas que um poderia encontrar deparando-se com uma *code review*. De seguida, iremos aplicar todos estes conhecimentos e conceitos ao *design* da experiência.

3.5 A Experiência

Iremos agora aplicar o que foi discutido previamente, ao desenho da nossa experiência em concreto. Queremos provar que as ***code inspections* ou *code reviews* devem pelo menos encontrar 70% dos bugs, até 25% de falsos positivos com 95% de confiança.** Para isto iremos utilizar uma metodologia moderna de *review - over the shoulder*. No total os revisores irão tentar encontrar *bugs* de diferentes tipos (sendo que a pesquisa não se foca no tipo de *bug* mas sim na quantidade em que encontram). Iremos ter seis (6) programas que os *reviewers* analisarão. Cada programa tem um tempo máximo associado ao mesmo, tal como uma dificuldade, sendo que a última, não é relevado ao testando. Adicionalmente, o máximo de linhas revista também será bloqueado, de acordo com

os padrões previamente discutidos. Para impedir ao máximo a utilização de ferramentas de ajuda externas, as *reviews* foram monitorizadas pelos discentes tentando sempre manter a integridade da mesma.

4 Planeamento Experimental

Para o planeamento e design da experiência a realizar, após algum estudo introdutório dos assuntos a ela associados, começou-se por decidir, em grupo, quais possivelmente poderiam ser as variáveis independentes da nossa experiência. Após uma breve reflexão, optámos pelas seguinte:

- Complexidade do código a ser revisto
- Experiência do reviewer
- Linguagem de programação procedimental ou orientada a objetos

Após isto, delineou-se os níveis para cada variável independente. Para a primeira variável independente - "Complexidade de código a ser revisto, foram criados códigos com três graus de complexidade (fácil, intermédio, difícil), dificuldade essa que tem por base a complexidade ciclomática de cada código e o número de linhas do mesmo, como discutido anteriormente na Secção 3.4.

Para medir a experiência de cada *reviewer* foi pedido a cada um deles uma auto avaliação no que toca às suas capacidades nas linguagens de C e Java, juntamente com a avaliação nas Unidades Curriculares das disciplinas respetivas aos paradigmas de Programação. Os participantes auto-avaliaram-se numa escala de um a cinco (1 a 5), sendo que esta foi posteriormente transposta para uma avaliação de 3 níveis - *Beginner*, *Intermediate* e *Expert*.

Por último, mas não menos importante, de modo a analisar a influência do tipo de linguagem (procedimental ou orientada a objetos) na eficácia das *code reviews*, foram criados códigos em duas linguagens representativas de cada um desses tipos: C e Java. No total foram criados 6 códigos, de modo a ter um código de cada grau de complexidade em ambas as linguagens.

Seguindo o plano, começou-se por elaborar um guião, contendo todas as regras e condições que os participantes teriam de respeitar para o bom funcionamento da experiência. Estes aspetos que iremos demonstrar são basais para que consigamos eliminar a imprevisibilidade de experiências com pessoas. Assim, estas permitem uniformizar os testes, com o objetivo de garantir que apenas as variáveis independentes previamente selecionadas exerçam influência nos resultados obtidos de uma forma que seja controlável:

- A experiência tem uma duração de sensivelmente 1H10 minutos
- O *reviewer* deve rever 6 códigos, 3 em C e 3 em Java
- Os 3 códigos de cada linguagem têm 3 graus de dificuldade distintos (fácil, intermédio e difícil), relacionados com a sua complexidade ciclomática e o seu número de linhas
- Cada código está comentado, com informação sobre a sua funcionalidade
- O participante pode colocar questões durante a experiência, desde que estas não sejam sobre o código
- Garantir que o ambiente onde se está a fazer a review é calmo e livre de barulhos e peturbações
- O código a rever estará limitado a 100 linhas
- O código é compilável, mas o participante não o pode correr
- Os bugs presentes no código poderão ser de 4 tipos distintos: *Logical Bug, Performance Bug, Variable Type Bug e Memory Management Bug*
- O número de bugs por código é variável, podendo haver códigos sem qualquer bug
- Todos os reviewers devem usar o mesmo IDE para analisar o código
- Para assinalar um bug, basta fazer um comentário junto ao mesmo, não sendo necessário corrigi-lo

- A utilização de ferramentas auxiliares para ajudar na deteção de bugs resultará na anulação da participação do reviewer na experiência

Note-se que este guião estará disponível integralmente para leitura no *Apêndice 8.2* e *Apêndice 8.3*, juntamente com a Declaração de Consentimento Informado. Este foi exposto a todos os participantes *a priori* da sua participação na experiência.

5 Execução da Experiência

5.1 Ferramentas Utilizadas

De forma a seguir o recomendado pelo Docente, os discentes decidiram utilizar uma ferramenta EDA (*Exploratory Data Analysis*, que ficasse responsável pela análise dos dados obtidos na experiência. Como IDE, foi utilizado o *Visual Studio Code* tanto para a realização dos *source codes* em Java como para C. Por último, mas não menos importante, utilizou-se o *Google Docs* para ajudar na realização e planeamento de todo o material de suporte à experiência, nomeadamente: guião/roteiro da experiência, declaração de consentimento informado, etc. Para a realização das videochamadas com os supostos *reviewers* foi utilizada a aplicação *Discord*, providenciando uma forma simples e rápida de partilha de ecrã, evitando qualquer fraude nos testes efetuados. Após a realização da experiência, utilizou-se o *Google Sheets* para expor os dados e efetuar todos os cálculos necessários para a criação das tabelas amostrais para cada hipótese. Adicionalmente para realizar todos os testes estatísticos, utilizou-se a ferramenta *DataTab*.

5.2 Segunda iteração da experiência

Inicialmente, foi decidido que antes da experiência, os participantes seriam informados de que o tempo que teriam para rever cada código estaria relacionado com a sua complexidade. No entanto, após a realização de um teste piloto, pôde-se observar que os participantes, tendo em posse esta informação, acabavam por estar mais atentos e colocar um maior foco nos códigos cujo limite de tempo era maior, devido à sua complexidade associada. Como tal, de modo a que o nível de atenção fosse consistente em todos códigos, independentemente da sua complexidade, foi alterado o protocolo para que essa informação não fosse dada aos participantes. Esta alteração permitiu refletir melhor as condições reais de uma *code review*, em que os *reviewers*, normalmente, não são informados da complexidade do código. Adicionalmente um outro problema foi detetado. Os códigos na experiência piloto foram entregues ao revisor por ordem de dificuldade, o que resultou em *reviews* mais inconsistentes na parte final da sua *review*. Assim também se passou a fazer uma randomização da ordem dos testes. Desta forma, pudemos eliminar estes efeitos negativos na experiência, nomeadamente o efeito natural do cansaço nos revisores, sempre nos mesmos testes.

6 Discussão de Resultados

Nos seguintes tópicos o grupo irá abordar as diferentes amostrar produzidas através dos resultados produzidos pela experiência. Utiliza-se como guia para a criação das amostras, as variáveis independentes, e tenta-se abordar cada uma delas de forma apropriada, utilizando testes estatísticos diferentes e portanto, análises diferentes - isto graças em parte, à sua natureza diferente. Iremos portanto, abordar os seguintes tópicos: análise da hipótese principal, esta relacionada com a eficácia de *code reviews*; avaliação da influência da linguagem de programação nos *bugs* detetados; e por último, mas não menos importante, análise da influência da dificuldade de cada programa, na deteção de bugs. Juntando a isto, para cada amostra, o grupo decide também fazer um estudo sobre os mesmos temas, mas relacionado com os falso positivos que os revisores encontraram nos códigos com *bugs*. Assim não só estamos a influência das variáveis no processo de encontrar *bugs*, como também no processo de detetar falsos positivos. Porém, antes disto, o grupo decidiu testar a normalidade de cada uma destas amostras, evitando qualquer problema que poderia ocorrer durante o cálculo dos testes estatísticos.

Reviewer	Java	C
1	0	1
2	0	0
3	3	1
4	0	0
5	0	1
6	6	1
7	8	2
8	4	2
9	7	2
10	9	3
11	4	2
12	4	4
13	13	1
14	13	1
15	4	3

Tabela 3: número de *bugs* encontrados por linguagem de programação.

6.1 Tratamento dos Dados

No total, foram realizadas quinze (*15*) experiências utilizando o protocolo revisado, implicando que todos os *reviewers* analisaram tanto os códigos de C, como os de Java. Este número assim se apresenta, graças a uma necessidade essencial de criar representatividade no espetro da qualidade do reviewer. Isto, na prática significa que, para evitar qualquer tipo de enviesamento dos dados (porque possivelmente podíamos ter mais *reviewers* com fraca performance, resultando em dados que não chegariam perto da eficácia possível), tentamos utilizar um número que fosse similar por todos os níveis das diferentes habilidades. O mesmo é expresso na seguinte tabela, notando-se que o mesmo *reviewer* poderá ser *Expert* numa linguagem, e *Beginner* noutra. Sumarizando assim, temos 50% dos testes realizados por *beginners* e 36% por *Intermediate*. O número de *Experts* mais reduzido (14%) deve-se à dificuldade que o grupo enfrentou de tentar encontrar revisores com alguma experiência, nomeadamente profissional.

Nível do *Reviewer*	Testes Efetuados
Beginner	15
Intermediate	11
Expert	4

Tabela 2: número de testes efetuados por nível de reviewer (cada revisor faz 2 *reviews*)

Como conseguimos verificar, asseguramos uma boa amostra para os dois primeiros níveis, sendo que para o nível de *Expert*, a amostra poderá ser insuficiente. Mesmo assim, devido à sua natureza, acreditamos que seja possível conseguir obter resultados significativos com esta amostra.

De seguida o grupo utilizou os dados produzidos na experiência do decorrer da experiência e sumariou-os na Tabela 31. É partindo dessa que tentamos traduzir os dados para um contexto mais apropriado, tendo em conta as diferentes hipóteses que estamos a tentar rejeitar. Assim por exemplo, através da tabela presente no Apêndice 1, que podemos gerar uma segunda tabela, como a seguidamente apresentada:

A partir desta, é calculado a proporção para cada revisor, de forma a que possamos utilizar estes dados para preencher uma terceira tabela. Esta será introduzida no *DataTab* para que possamos realizar os testes que assim entendermos sobre ela. A fórmula utilizada para este exemplo (referindo-nos agora à proporção que um revisor possuí de *bugs* encontradas de Java, no seu teste) é a seguinte:

$$\frac{\text{n bugs encontrados em programas Java}}{\text{n bugs injetados em programas Java}} = \widehat{p} \tag{1}$$

Este processo foi efetuado para cada uma das variáveis independentes e o grupo considerou ser a metodologia mais eficiente tendo em conta o grande volume de dados e variáveis independentes a

testar. Adicionalmente também se realizou o mesmo processo para os falsos positivos, mas com uma pequena alteração à fórmula:

$$\frac{\text{n falsos positivos em programas Java}}{\text{n bugs detetados na totalidade pelo } \textit{reviewer} + \text{falsos positivos detetados}} = \widehat{p} \tag{2}$$

Resumidamente então, a fase de tratamento de dados pode ser delineada pelos seguintes pontos:

1. Tratamento de dados e inserção dos dados na tabela geral
2. Transposição dos dados para uma tabela relativa à amostra, com o cálculo da proporção individual para cada revisor
3. Calculo do teste estatístico
4. Análise dos resultados produzidos pelos testes estatísticos
5. Posterior iteração do teste estatístico

6.2 Testes de Normalidade

Antes de começarmos a efetuar testes estatísticos, iremos testar a normalidade dos dados descritos anteriormente. Para testar a normalidade destas amostras, o grupo decidiu utilizar um teste não paramétrico - Kolmogorov-Smirnov. Assim, o grupo introduziu os dados referentes a cada amostra diferente (as mencionadas anteriormente), na ferramenta *DataTab* e procedeu ao cálculo da normalidade, através do teste Kolmogorov-Smirnov, para cada coluna de dados. Os resultados desses mesmos cálculos podem ser expressos pela tabela seguinte:

Amostra	**Variável Independente**	**Estatística**	***P-Value***
Comparar as duas Linguagens:	Java	0.19	.581
	C	0.24	.316
Comparar as duas Linguagens c/ Falsos Positivos:	Java:	0.22	.394
	C	0.24	.303
Dificuldade das duas linguagens:	Fácil	0.23	.329
	Médio	0.23	.363
	Difícil	0.27	.196
Dificuldade das duas linguagens c/ Falsos Positivos:	Fácil	0.25	.254
	Médio	0.37	.026
	Difícil	0.24	.296
Habilidade dos *Reviewers*:	*Beginner*	0.26	.387
	Intermediate	0.19	.834
	Expert	0.28	.836
Habilidade dos Reviewers c/ Falsos Positivos:	*Beginner*	0.2	.69
	Intermediate	0.2	.821
	Expert	0.28	.821

Tabela 4: normalidade das amostras, segundo Kolmogorov-Smirnov

Na seguinte tabela conseguimos observar dois valores, referentes às colunas de cada uma destas amostras - isto no sentido prático, é colocar os dados referentes à amostra dos *bugs* encontrados na linguagem de Java, por cada teste, e utiliza-los para realizar o teste de Kolmogorov-Smirnov. Assim na coluna conseguimos visualizar na coluna *Estatísticas* o valor referente ao teste estatístico. Ao seu lado, encontra-se o *P-value* desse teste. Temos que, como todos os valores presentes aqui, não são muito superiores a 0.05, o valor da nossa significância, podemos assumir a normalidade dos nossos dados e proceder aos testes estatísticos.

6.3 Análise da experiência

Para provar que as *codes reviews* encontram pelos menos 70% dos *bugs* e sem mais de 25% de falsos positivos, com 95% de confiança, fizemos primeiro uma análise à primeira parte da hipótese, "provar que as *codes reviews* encontram pelos menos 70% dos *bugs*", onde definimos as seguinte hipóteses:

Hipótese Nula (H0):

$$H_0 : \text{A proporção de } bugs \text{ encontrados nas code reviews é inferior a 70\%.}$$

Hipótese Alternativa (H1):

$$H_1 : \text{A proporção de } bugs \text{ encontrados nas code reviews é de pelo menos 70\%.}$$

Ao analisarmos o gráfico abaixo com as percentagens de *bugs* encontrados por cada *reviewer*, podemos constatar que existe uma grande disparidade entre os dados obtidos, o que se pode comprovar pelo valor elevado do desvio padrão (27.93). A média situou-se nos 38.82% de *bugs* encontrados, tendo os dois melhores *reviewers* encontrado 82.35% dos *bugs*, enquanto que o pior *reviewer* não encontrou qualquer bug (0%). Esta disparidade dos resultados poderá estar relacionada com a experiência de cada *reviewer*, uma das variáveis independentes da nossa experiência. Mais abaixo, na secção 6.5, é testada essa hipótese. No geral, a percentagem de *bugs* encontrados situou-se acima dos 30%, com apenas quatro *reviewers* a encontrarem menos que este valor. Desses quatro, três apresentaram uma percentagem inferior a 6%. Atendendo à generalidade dos valores, os resultados desses três *reviewers* podem ser considerados *outliers*. Por outro lado, dos quinze *reviewers*, apenas dois conseguiram encontrar mais do que 75% dos *bugs*, o que poderá levar a crer, ainda antes da realização de qualquer teste estatístico, que a hipótese nula não pode ser rejeitada.

Reviewer	Percentagem de *bugs* encontrados
1	5,88
2	23,53
4	0,00
5	5,88
6	41,18
7	58,82
8	35,29
9	52,94
10	70,59
11	35,29
12	47,06
13	82,35
14	82,35
15	41,18
Média	38,82
Desvio padrão	27,93

Tabela 5: *bugs* encontrados por *reviewer* (*em percentagem)*

Foi escolhido o *t-test* com uma amostra para testar se a hipótese nula pode ser rejeitada ou não, com grau de liberdade igual a 14 (tamanho da amostra menos 1). A partir do valor de t, -4.32, e com recurso à tabela estatística para os *t-test*, foi obtido um *p-value* com valor 1, maior do que o nível de significância 0.05, o que sugere que os resultados observados não são estatisticamente significantes, ou seja, a partir destes não é possível rejeitar a hipótese nula. A diferença média obtida implica que, em média, a percentagem de *bugs* encontrados foi 31.18% menor do que o valor alvo de 70%, ou seja, situou-se nos 38.82%, como já se tinha observado na tabela acima. O intervalo de confiança indica-nos que a verdadeira média de *bugs* encontrados se situa entre os 23.35% e os 54.29%, com um grau de confiança de 95%.

	n	Mean	Std. Deviation	Std. Error Mean
Total	15	38,82	27,93	7,21

Tabela 6: análise descritiva dos *bugs* encontrados por *reviewer*

	t	df	p	Mean Difference	Lower limit	Upper limit
Total	-4,32	14	1	-31,18	-46,65	-15,71

Tabela 7: variância para os *bugs* encontrados por *reviewer*

Perante os valores obtidos no *t-test*, não podemos rejeitar a hipótese nula, o que nos impede de concluir que as *code reviews* encontram em média mais de 70% dos *bugs*. Mesmo a análise inicial da média dos dados obtidos fazia prever esta conclusão, já que 38.82% se encontra muito distante dos 70% pretendidos.

De seguida, foi feita a análise à percentagem de falsos positivos detetados. Para este efeito definiu-se as seguintes hipóteses:

Hipótese Nula (H0):

H_0 : A proporção de falsos positivos nas *code reviews* é superior a 25%.

Hipótese Alternativa (H1):

H_1 : A proporção de falsos positivos nas *code reviews* é inferior ou igual a 25%.

Neste caso, é observável dois grupos distintos: o das *reviews* que obtiveram 80% ou mais de falsos positivos e outro que contém as que tiveram menos de 25% de falsos positivos. Apenas uma das quinze *reviews* não se insere num destes grupos, sendo que o segundo é um pouco mais numeroso que o primeiro. Esta amplitude de resultados leva a um desvio padrão de 34.66. A média de 39.39% leva a crer, numa primeira análise, que não será possível rejeitar a hipótese nula de a percentagem de falsos positivos é superior a 25%. Concluindo assim, não se pode garantir que as *code reviews* tenham uma eficácia de 65%, nem que a taxa de falsos positivos detetados seja abaixo de 25%.

Reviewer	Percentagem de falsos positivos
1	85,71
2	100,00
3	55,56
4	100,00
5	80,00
6	22,22
7	9,09
8	25,00
9	10,00
10	20,00
11	25,00
12	20,00
13	12,50
14	12,50
15	12,50
Média	39,34
Desvio padrão	34,66

Foi novamente escolhido o *t-test* com 1 amostra para testar se a hipótese nula podia ser rejeitada ou não, com grau de liberdade igual a 14. A partir do valor de *t*, 1.6, e com recurso à tabela estatística para os *t-test*, foi obtido um *p-value* com valor 0.934, maior do que o nível de significância 0.05, o que sugere que os resultados observados não são estatisticamente significantes, ou seja, que a partir destes não é possível rejeitar a hipótese nula. A diferença média obtida implica que, em média, a percentagem de falsos positivos encontrados em cada *review* seja 14.34% maior do que o valor alvo de 25%, ou seja,

situou-se nos 39.34%, como já se tinha observado na tabela acima. O intervalo de confiança indica-nos que a verdadeira média de falsos positivos encontrados se situa entre os 34.48% e os 72.87%, com um grau de confiança de 95%.

	n	Mean	Std. Deviation	Std. Error Mean
Total	15	39,34	34,66	8,95

Tabela 9: análise descritiva dos falsos positivos por *reviewer*

	t	df	p	Mean Difference	Lower limit	Upper limit
Total	1,6	14	0,934	14,34	-4,86	33,53

Tabela 10: variância para os falsos positivos por *reviewer*

Perante os valores obtidos nos *t-test*, não podemos rejeitar nenhuma das hipóteses nula, o que nos impede de concluir que as *code reviews* encontram em média mais de 70% dos *bugs* ou que estas têm menos de 25% de falsos positivos. Mesmo a análise inicial da média dos dados obtidos fazia prever esta conclusão, já que 38.82% se encontra muito distante dos 70%, no caso da percentagem de bugs encontrados, enquanto que no casos dos falsos positivos, a média 39.34 se encontra também bastante acima do valor limite de 25%.
Assim, a partir desta análise, poderíamos concluir que, perante os objetivos delineados, as *code reviews* não "valem a pena".

De seguida, procedeu-se à análise do impacto das diferentes variáveis independentes escolhidas para a nossa experiência (Paradigma de Programação, Complexidade do Código e Qualidade do *Reviewer*) na percentagem de bugs detetados e de falsos positivos encontrados nas *code reviews*.

6.4 Resultados por paradigma de programação (procedimental e de objetos)

Como podemos observar na tabela abaixo, a percentagem média de *bugs* encontrados nas *code reviews* e o respetivo desvio padrão é semelhante entre ambas as linguagens de programação. Tanto em Java com em C, existiu uma grande dispersão de resultados, com percentagens de 0 e 100 porcento, o que provavelmente resultou da presença de *reviewers* com diferentes níveis de *skill* na experiência.

Reviewer	Java	C	Total
1	0,00	25,00	5,88
2	0,00	0,00	0,00
3	23,08	25,00	23,53
4	0,00	0,00	0,00
5	0,00	25,00	5,88
6	46,15	25,00	41,18
7	61,54	50,00	58,82
8	30,77	50,00	35,29
9	53,85	50,00	52,94
10	69,23	75,00	70,59
11	30,77	50,00	35,29
12	30,77	100,00	47,06
13	100,00	25,00	82,35
14	100,00	25,00	82,35
15	30,77	75,00	41,18
Média	38,46	40	38,82
Desvio padrão	33,53	28,03	27,93

Tabela 11: *bugs* encontrados por *reviewer* e por tipo de programação (procedimental e objetos), *em percentagem*

De seguida, foi realizado um *t-test* com 2 amostras independentes, uma vez que os programas em Java são independentes dos programas em C. O objetivo foi determinar se há ou não diferença entre o número de *bugs* encontrados por linguagem. Para isso, definimos as nossas hipóteses:

Hipótese Nula (H0):

H_0 : "Não há diferença na percentagem de *bugs* encontrados entre os programas em Java e os programas em C."

Hipótese Alternativa (H1):

H_1 : Há diferença na percentagem de *bugs* encontrados entre os programas em Java e os programas em C."

Após a observação dos resultados deste teste, podemos concluir, com um intervalo de 95% de confiança, que não podemos rejeitar a hipótese nula, uma vez que o valor de p é 0,893. Este valor de p significa que a probabilidade de obter os resultados observados na tabela acima, caso a hipótese nula seja verdadeira, é de 89,3%. O nosso intervalo de confiança indica que a diferença entre a percentagem de *bugs* encontrados anda entre os -24.65 e os 21.57 porcento, para 95% de confiança, ou seja, o facto de o intervalo incluir 0 é outro fator que indica que estes resultados não são estatisticamente significativos para rejeitar a hipótese nula.

Linguagem de programação	n	Mean	Std. Deviation	Std. Error Mean
Java	15	38,46	33,53	8,66
C	15	40	28,03	7,24

Tabela 12: análise descritiva para a influência do tipo de programação (procedimental e objetos)

	t	df	p	Cohen's d
Equal variances	-0,14	28	0,893	0,05
Unequal variances	-0,14	27,15	0,893	0,05

Tabela 13: variância para a influência do paradigma de programação (procedimental e objetos)

Assim, não podemos concluir a partir destes resultados que o tipo de linguagem de programação, no nosso caso orientada a objetivos (Java) ou procedimental (C), tenha influência na quantidade de *bugs* encontrados durante uma *code review*.

Também foi medida a percentagem de falsos positivos para ambas as linguagens de programação. Neste caso, houve uma diferença na média de falsos positivos presentes em cada *review*, com o Java a apresentar cerca de menos 13% do que o C, sendo que este último apresentou um desvio padrão de 38,89, o que revela uma grande disparidade entre os diferentes *reviewers*.

Reviewer	Java	C	Total
1	0,00	85,71	85,71
2	0,00	100,00	100,00
3	40,00	75,00	55,56
4	100,00	100,00	100,00
5	100,00	0,00	80,00
6	0,00	66,67	22,22
7	0,00	33,33	9,09
8	0,00	50,00	25,00
9	0,00	33,33	10,00
10	18,18	25,00	20,00
11	33,33	0,00	25,00
12	33,33	0,00	20,00
13	13,33	0,00	12,50
14	13,33	0,00	12,50
15	20,00	0,00	12,50
Média	24,77	37,94	39,34
Desvio padrão	33,49	38,89	34,66

Tabela 14: percentagem de falsos positivos encontrados por reviewer e por tipo de programação (procedimental e objetos)

Após esta análise mais superficial, procedeu-se à realização de um teste t com 2 amostras independentes, tal como foi feito para o número de *bugs*. O objetivo deste teste foi determinar se há ou não diferença no número de falsos positivos entre as duas linguagens de programação. Neste sentido, definimos as nossas hipóteses:

Hipótese Nula (H0):

H_0 : "Não há diferença na percentagem de falsos positivos encontrados entre os programas em Java e os programas em C."

Hipótese Alternativa (H1):

H_1 : Há diferença na percentagem de falsos positivos encontrados entre os programas em Java e os programas em C."

Ao observar os resultados deste teste, pode-se concluir, com o intervalo de 95% de confiança, que não podemos rejeitar a hipótese nula, uma vez que o valor de p é 0,329. Este valor significa que a probabilidade de obter os resultados observados na tabela acima, caso a hipótese seja verdadeira, é de 32,9%. O intervalo de confiança revela que, com 95% de confiança, a diferença entre a percentagem de falsos positivos encontrados nas duas linguagens está estimada entre -40,31% e 13,97%.

Linguagem de programação	n	Mean	Std. Deviation	Std. Error Mean
Java	15	24,77	33,5	8,65
C	15	37,94	38,89	10,04

Tabela 15: análise Descritiva dos falsos positivos para a influência do tipo de programação (procedimental e objetos)

	t	df	p	Cohen's d
Equal variances	-0,99	28	0,329	0,36
Unequal variances	-0,99	27,4	0,329	0,36

Tabela 16: variância da análise aos falsos positivos para a influência do tipo de programação (procedimental e objetos)

Assim, não podemos concluir a partir destes resultados que o tipo de linguagem de programação tenha influência na quantidade de falsos positivos presentes numa *code review*, apesar de, uma primeira

análise da média dos dados parecer apontar que as *code reviews* de programas em Java são menos propensas a falsos positivos do que as de programas em C.

Concluiu-se então que, como não se conseguiu provar a existência de uma diferença estaticamente significativa entre os resultados de ambas as linguagens de programação, não fazia sentido testar individualmente, para cada uma das linguagens, as duas hipóteses base, ou seja, se as *code reviews* conseguem identificar mais de 70% dos bugs com menos de 25% de falsos positivos. Se o fizéssemos, uma vez que não há diferença estatisticamente relevante entre as linguagens, chegariamos às mesmas conclusões a que chegámos na análise conjunta.

6.5 Resultados por complexidade do código

Nesta secção pretendemos avaliar a influência da complexidade do código no resultado das *code reviews*. Cada reviewer testou dois códigos para cada nível de dificuldade.

Reviewer	**Fácil**	**Médio**	**Difícil**
1	20	0	0
2	0	0	0
3	40	25	0
4	0	0	0
5	20	0	0
6	60	25	50
7	60	50	75
8	80	25	0
9	80	50	25
10	100	50	75
11	20	50	25
12	60	50	25
13	80	75	100
14	60	87.5	75
15	60	50	0

Tabela 17: *bugs* encontrados por reviewer, em testes classificados (*em percentagem*)

Podemos afirmar que existe uma diminuição do número de *bugs* detetados à medida que aumenta a dificuldade, o que é esperado porque à medida que aumenta a complexidade do código os *bugs* são mais difíceis de encontrar e aumenta a margem de erro na deteção de *bugs*, que vai ser discutido mais à frente quando forem analisados os falsos positivos. De forma a estudar o tema desta secção foi realizado um teste ANOVA *one-way*, sem repetição de variáveis, de forma a verificar se os fatores independentes(uns aos outros) em causa influenciam a quantidade de *bugs* encontradas no código. Foram criadas as seguintes hipóteses:

Hipótese Nula (H0):

H_0 : "Não há diferença na percentagem de *bugs* encontrados entre os programas de diferentes dificuldades."

Hipótese Alternativa (H1):

H_1 : "Há diferença na percentagem de *bugs* encontrados entre os programas de diferentes dificuldades."

	n	Mean	Std. Deviation
Fácil	15	49,33	31,05
Médio	15	35,83	27,9
Difícil	15	30	35,61

Tabela 18: estatística descritiva para influência da complexidade de código

	Sum of Squares	df	Mean Squares	F	p
Factor	2950,28	2	1475,14	1,47	,241
Residual	42139,17	42	1003,31		
Total	45089,44	44			

Tabela 19: ANOVA para influência da complexidade de código.

Com recurso ao *DataTab* foi então feito o cálculo do teste ANOVA one-way, sem repetição de variáveis e foi possível observar um p-valor de 0.241, para o intervalo de confiança de 95%, o que é superior ao nível de significância de 0.05 que foi estabelecido. Por isso, o resultado deste teste não é significativo para os dados analisados. Posto isto, não iremos rejeitar a hipótese H0, ou seja, concluímos que "Não há diferença na percentagem de *bugs* encontrados entre os programas de diferentes dificuldades."

Para além desta análise, foi feito o cálculo dos falsos positivos para as diferentes complexidades de código.

Reviewer	Fácil	Médio	Difícil
1	50,00	100,00	100,00
2	100,00	100,00	100,00
3	50,00	0,00	100,00
4	100,00	0,00	100,00
5	50,00	100,00	100,00
6	0,00	33,33	33,33
7	0,00	20,00	0,00
8	0,00	33,33	100,00
9	0,00	20,00	0,00
10	0,00	20,00	40,00
11	50,00	20,00	0,00
12	25,00	20,00	0,00
13	20,00	14,29	0,00
14	25,00	12,50	0,00
15	0,00	0,00	100,00
Média	31,33	32,90	51,56
Desvio padrão	34,77	36,20	48,43

Tabela 20: falsos positivos encontrados por reviewer, em testes classificados (em percentagem)

Pela análise da tabela anterior conseguimos ver uma subida da percentagem de falsos positivos à medida que aumenta a dificuldade do código, o que é de certa forma expectável porque com o aumento da dificuldade a sintaxe torna-se mais complexa o que torna mais fácil a identificação de *bugs* que na realidade não são *bugs*. Foi feito novamente um teste ANOVA one-way, sem repetição de variáveis para verificar se há diferença na percentagem de falsos positivos encontrados entre os programas de várias dificuldades. Foram então criadas as hipóteses:

Hipótese Nula (H0): Hipótese Alternativa (H1):

H_0 : "Não há diferença na percentagem de *bugs* encontrados entre os programas de várias dificuldades."

Hipótese Alternativa (H1):

H_1 : "Há diferença na percentagem de *bugs* encontrados entre os programas de várias dificuldades."

	n	Mean	Std. Deviation
Fácil	15	31,33	34,77
Médio	15	32,9	36,2
Difícil	15	51,56	48,43
Total	45	38,6	40,43

Tabela 21: estatística descrivita para falsos positivos da complexidade do código

	Sum of Squares	df	Mean Squares	F	p
Factor	3797,6	2	1898,8	1,17	,32
Residual	68114,31	42	1621,77		
Total	71911,9	44			

Tabela 22: ANOVA para falsos positivos da complexidade do código

De acordo com o teste ANOVA aqui realizado, podemos concluir, com um intervalo de confiança de 95% que não podemos rejeitar a hipótese nula, dado que o p-valor é de 0,32, que é superior ao nível de significância de 0.05, logo não conseguimos rejeitar a hipótese H0 porque não há diferença significativa entre as dificuldade do código.

6.6 Resultados por Qualidade do *Reviewer*

Nesta secção pretendemos avaliar a influência da qualidade do reviewer no nosso resultado. Para esse mesmo efeito decidiu-se utilizar um teste ANOVA *one-way*, sem repetição de variáveis, conseguindo assim verificar se estes 3 fatores independentes (uns dos outros) têm alguma influência ou não na quantidade de *bugs* encontradas nos códigos. Definiu-se assim em primeiro lugar, *H0* e *H1*:

Hipótese Nula (H0):

H_0 : Não há influência entre as três categorias de qualidade do revisor no número de *bugs* descobertos.

Hipótese Alternativa (H1):

H_1 : Há influência entre as três categorias de qualidade do revisor no número de *bugs* descobertos.

Após o cálculo efetuado através da ferramenta *DataTab*, podemos concluir com um intervalo de 95% de confiança que não podemos rejeitar a hipótese Nula, ou seja, que de facto não podemos garantir que não haja uma influência no número de *bugs* encontrados, tendo em conta a qualidade dos *reviewers*. Isto deve-se ao facto de que o *p-value* obtido foi menor do que *0.001*, logo sendo assim, menor que o valor da significância (*0.05*). Isto, na prática, significa que, tendo em conta os valores obtidos, não temos resultados estatisticamente relevantes ou suficientes de forma a podermos rejeitar a hipótese nula.

Reviewer	***Beginner***	***Intermediate***	***Expert***
1	5.88		
2	0		
3	23.08	25	
4	0		
5	0	25	
6	25	46.15	
7	50	61.54	
8	35.29		
9		41.18	
10		70.59	
11	50	30.77	
12		30.77	100
13	25		100
14	25		92.31
15		30.77	75

Tabela 23: *bugs* encontrados por *reviewer*, classificados por habilidade (*em percentagem*)

	n	Mean	Std. Deviation
Beginner	11	1.55	1.75
Intermediate	9	5.22	3.49
Expert	4	8	5.23

Tabela 24: estatística descritiva para a Influência da qualidade do *reviewer*

	Sum of Squares	df	Mean Squares	F	p
Factor	143.72	2	71.86	7.18	.004
Residual	210.28	21	10.01	01.07	
Total	354	23			

Tabela 25: resultados ANOVA para influência da qualidade do *reviewer*

De acordo com o teste ANOVA realizado para esta amostra de dados, o grupo conclui que, com um intervalo de confiança de 95%, podemos rejeitar a hipótese nula, dado que o valor obtido do *p-value* é de 0.04. Este é inferior à significância de 0.05, sendo que por estas razão, podemos rejeitar o facto de que a habilidade nos 3 diferentes grupos de reviewer não é um fator relevante para taxa de deteção. Este resultado também é facilmente evidente por uma análise da tabela 25. Como conseguimos evidenciar, até mais especificamente no terceiro grupo, mesmo tendo em conta que o número de amostra é pequeno, conseguimos ver uma diferença muito grande entre os resultados.

Desta forma, passamos para a análise das taxas de falsos positivos em amostras de grupos de habilidade diferentes.

	n	Mean	Std. Deviation
Beginner	11	50,06	41,9
Médio	9	21,57	24,02
Expert	4	13,15	10,24
Total	24	33,23	35,16

Tabela 26: estatística descritiva para falsos positivos da influência da qualidade do *reviewer*

	Sum of Squares	df	Mean Squares	F	p
Factor	5952,37	2	2976,19	2,78	0,085
Residual	22485,19	21	1070,72		
Total	28437,57	23			

Tabela 27: ANOVA para falsos positivos da influência da qualidade do *reviewer*

De acordo com o teste ANOVA realizado para esta amostra de dados, o grupo conclui que, com um intervalo de confiança de 95%, podemos rejeitar a hipótese nula, dado que o valor obtido do *p-value* é de 0.085. Este é superior à significância de 0.05, sendo que por estas razão, podemos rejeitar o facto de que a habilidade nos três diferentes grupos de reviewer não é um fator relevante para a taxa de deteção de falsos positivos. Este resultado também é tão esperável como o último, graças à mesma análise da tabela, porém, nesta instância, referimos-nos à taxa de falsos positivos.

Como conseguimos evidenciar, pelo que foi anteriormente mencionado, os resultados expressam que poderá eventualmente existir grupos de *reviewers* que consigam atingir a meta de 70% de eficácia nas *code review*, com uma taxa de deteção inferior a 25%. Dito isto e substanciado pelos resultados dos testes ANOVA, o grupo decide que é da maior pertinência, fazer uma maior separação dos dados, estudando assim individualmente cada grupo de revisores. Na seguinte secção iremos abordar essa mesma questão.

6.6.1 Resultados por Qualidade do *Reviewer* Individuais

De seguida, decidimos testar a nossa hipótese apenas contabilizando os resultados dos *reviewers* que considerámos como *experts*, uma vez que foram os que mostraram os resultados que mais se aproximavam do valor mínimo de 70% pretendido para a percentagem de *bugs* encontrados.
Na tabela abaixo, podemos constatar que a percentagem de *bugs* encontrados pelos *reviewers expert* se situou acima dos 70% para todos os casos.

Reviewer	Percentagem de *bugs* encontrados
12	100
13	100
14	92,30769231
15	75

Tabela 28: percentagem de *bugs* encontrados pelos reviewers *expert*

	t	df	p	Mean Difference	Lower limit	Upper limit
Expert	3,7	3	,017	21,83	3,07	40,58

Tabela 29: *t-test* para o grupo dos *expert* para a taxa de deteção de *bugs*

Como podemos verificar, foi obtido um valor para t de 3.7, que, através das tabelas estatísticas para os testes do tipo t, nos permite chegar a um valor para p de 0,017. Este valor de p é inferior a 0.05, o que nos permite rejeitar a hipótese nula de que a percentagem de *bugs* encontrados nas *reviews* era inferior a 70%. O valor verdadeiro situa-se nos entre os 73.07% e os 100%, com 95% de confiança.

	t	df	p	Mean Difference	Lower limit	Upper limit
Expert	-4,77	3	,009	-15,6	-25,99	-5,2

Tabela 30: *t-test* para cada um dos grupos, taxa de deteção de *bugs*

Como conseguimos evidenciar pela tabela acima, obteve-se um *p-value* de 0.009, que está abaixo ao valor de significância estabelecido de 0.05. Assim temos que, através do *t-test* efetuado, somos

obrigado a rejeitar a hipótese nula. Adicionalmente podemos assumir que a *sample* utilizada para este teste, não é assumida de ser parte da população em que a taxa de deteção de falsos positivos é maior ou igual a 25%.

Estes resultados permitem-nos concluir que, para os *reviewers* do tipo *expert*, as *code reviews* "valem a pena", no sentido de que são capazes, com um grau de 95% de certeza, de apanhar pelo menos 70% dos *bugs* com uma percentagem de falsos positivos inferior a 25%.

7 Notas Finais

Na presente secção o grupo irá abordar todas as hipóteses formuladas ao longo do estudo, e adicionar também alguns comentários, tanto a este como a estudos possíveis através do *set* de amostras produzidos. Após efetuada a análise à hipótese base e a todas as variáveis independentes que considerámos, podemos tirar várias conclusões.

7.1 Eficácia de *Code Reviews* e Falsos Positivos

Em relação à hipótese base, não conseguimos rejeitar a hipótese nula, tanto para o teste dos 70% de *bugs* encontrados como no teste dos 25% de falsos positivos, o que leva o grupo a concluir, tendo por base os resultados da nossa experiência que as *code reviews* não detetam, em média, mais de 70% dos bugs com menos de 25% de falsos positivos.

7.2 O efeito do Paradigma de Linguagem de Programação nas *Code Reviews* e Falsos Positivos

Da análise aos resultados para as duas linguagens de programação presentes na nossa experiência, uma orientada a objetos e a outro procedimental, não conseguimos encontrar diferenças estatisticamente relevantes entre ambas, o que nos leva a concluir que esta variável não tem o efeito esperado na quantidade de *bugs* detetados e falsos positivos encontrados. O grupo suspeita que este resultado também esteja relacionado com a elevada diferença de *bugs* entre os programas de C e de Java.

7.3 O efeito da complexidade dos programas nas *Code Reviews* e Falsos Positivos

Dos testes realizados aos resultados obtidos das *samples* da complexidade dos programas (no nosso caso, complexidade ciclomática e número de linhas de código), também não foi possível provar que existisse uma diferença estatisticamente relevante entre os níveis da complexidade dos programas. Este resultado surpreende o grupo, visto que, supostamente o resultado deveria ser o contrário, tendo um maior número de *bugs* detetados nos programas mais simples.

7.4 O efeito da qualidade dos Revisores nas *Code Reviews* e Falsos Positivos

Ao contrário das variáveis independentes anteriores, foi possível, através do teste ANOVA, encontrar diferenças estatisticamente relevantes nas proporções produzidas. De facto, é possível observar que a percentagem de bugs encontrados pelos *reviewers* considerados *expert* foi consideravelmente superior àquele dos *reviewers* do tipo médio e *beginner*. Também a percentagem de falsos positivos foi inferior. De seguida, foram testadas as duas hipóteses base apenas os resultados dos *reviewers expert*, sendo que foi possível rejeitar ambas as hipóteses nulas e concluir que, para estes *reviewers*, as *code reviews* encontram mais de 70% de bugs com menos de 25% de falsos positivos. Podemos assim afirmar que um treino correto, tanto nos revisores como nos desenvolvedores podem resultar em resultados exponencialmente melhores.

7.5 Conclusão

De facto que é possível obter percentagens de bugs detetados acima dos 70% com menos de 25% de falsos positivos, independentemente do tipo da linguagem (orientada a objetos ou procedimental) e da

sua complexidade. No entanto, para o conseguir, é necessário que os *reviewers* tenham um elevado conhecimento e familiaridade com a linguagem de programação do código a rever. Assim, se uma empresa pretender fazer uso de *code reviews* para diminuir a quantidade de bugs no seu *software*, deve garantir que os seus *reviewers* têm a formação e os conhecimentos adequados para o fazerem, de modo a garantirem a sua eficácia. Caso contrário, torna-se muito difícil atingir as percentagens de bugs encontrados e de falsos positivos pretendidas.

Adicionalmente, podemos afirmar que um estudo relevante, mas não realizado, seria a percentagem de *bugs* a detetar nos programa. Apesar de, e agora citando o protocolo da experiência a que os testados foram submetidos, o tipo de *bugs* possíveis de encontrar no programa ser mostrado aos revisores, um poderia classificar o tipo, e estudar a sua proporção avaliando quais os mais facilmente detetados, quais os que não teriam sido tão facilmente detetados, e como é que as variáveis independentes poderiam, ou não, influenciar essa proporção.

8 Apêndice

8.1 Apêndice 1 - Tabela geral dos resultados da experiência

Teste	Programa	Positivos	Falsos Positivos	Qualidade do *Reviewer*
1	pyramid	1	1	*Beginner*
	employee	0	2	
	b_tree	0	3	
	pharmacy	0	0	*Beginner*
	backpack	0	0	
	tree	0	0	
2	pyramid	0	1	*Beginner*
	employee	0	3	
	b_tree	0	2	
	pharmacy	0	0	*Beginner*
	backpack	0	0	
	tree	0	0	
3	pyramid	0	2	*Intermediate*
	employee	1	0	
	b_tree	0	1	
	pharmacy	2	0	*Beginner*
	backpack	1	1	
	tree	0	1	
4	pyramid	0	1	*Beginner*
	employee	0	0	
	b_tree	0	1	
	pharmacy	0	1	*Beginner*
	backpack	0	0	
	tree	0	0	
5	pyramid	1	0	*Intermediate*
	employee	0	0	
	b_tree	0	0	
	pharmacy	0	1	*Beginner*
	backpack	0	1	
	tree	0	2	
6	pyramid	1	0	*Beginner*
	employee	0	1	
	b_tree	0	1	
	pharmacy	2	0	*Intermediate*
	backpack	2	0	
	tree	2	0	
7	pyramid	2	0	*Beginner*
	employee	0	1	
	b_tree	0	0	
	pharmacy	1	0	*Intermediate*
	backpack	4	0	
	tree	3	0	
8	pyramid	2	0	*Beginner*
	employee	0	1	
	b_tree	0	1	
	pharmacy	2	0	*Beginner*
	backpack	2	0	
	tree	0	0	
9	pyramid	1	0	*Intermediate*
	employee	1	1	
	b_tree	0	0	

	pharmacy	3	0	*Intermediate*
	backpack	3	0	
	tree	1	0	
10	pyramid	2	0	*Intermediate*
	employee	1	1	
	b_tree	0	0	
	pharmacy	3	0	*Intermediate*
	backpack	3	0	
	tree	3	2	
11	pyramid	0	0	*Beginner*
	employee	1	0	
	b_tree	1	0	
	pharmacy	1	1	*Intermediate*
	backpack	3	1	
	tree	0	0	
12	pyramid	2	0	*Expert*
	employee	2	0	
	b_tree	0	0	
	pharmacy	1	1	*Intermediate*
	backpack	2	1	
	tree	1	0	
13	pyramid	1	0	*Beginner*
	employee	0	0	
	b_tree	0	0	
	pharmacy	3	1	*Expert*
	backpack	6	1	
	tree	4	0	
14	pyramid	0	0	*Beginner*
	employee	1	0	
	b_tree	0	0	
	pharmacy	3	1	*Expert*
	backpack	6	1	
	tree	3	0	
15	pyramid	2	0	*Expert*
	employee	1	0	
	b_tree	0	0	
	pharmacy	1	0	*Intermediate*
	backpack	3	0	
	tree	0	1	

Tabela 31: dados gerais recolhidos durante a experiência (por programa, por dificuldade, e por habilidade)

8.2 Apêndice 2 - Declaração de Consentimento Informado

8.2.1 Purpose of this study

The purpose of this study is to understand if code inspections are worthwhile. Your participation in this study will help our team to answer that question by reviewing some pieces of code.

- The researcher has explained the purpose of the research to me.
- I have had an opportunity to ask questions about the study.

8.2.2 Freedom to Widthdraw

Your participation in this study is voluntary.

- You can refuse to take part at any time.
- You can take a break at any time.
- You can ask questions at any time.

8.2.3 Information we will collect

We will provide you with six pieces of code (3 in java 3 in C) with different difficulties. You will review them and try to identify *bugs* in the code. The code may have lots of *bugs* or none.

- I understand what I am expected to do.

8.2.4 Privacy and Confidentiality

Our team will be present during your coding review. We will not record those sessions. We may publish research reports that include your comments. The data used in these reports will be anonymous. This means you will not be identifiable and your comments will be confidential.

- I understand that people on the team will be present during the review session.
- I understand that my comments are confidential.

8.2.5 Your Agreement

To take part in the research, please sign this form showing that you consent to us collecting these data.
Your Name:
Signature:
Date:

8.3 Apêndice 3 - Guião da Experiência

8.3.1 Propósito da Experiência

A experiência é realizada no âmbito da disciplina de Metodologias Experimentais em Informática, onde nos foi requerido para desenharmos e realizarmos a experiência. Esta relaciona-se com a eficácia de code reviews. De forma mais concreta, pretendemos responder à questão: Será que as *code reviews* valem a pena de serem realizadas". O que nós iremos fazer durante a experiência é uma *code review*, do tipo *"over the shoulder"*, onde o testando irá tentar encontrar todos os *bugs* num código fornecido por nós, dentro de um espaço de tempo limitado.

8.3.2 Regras da Experiência

- A experiência tem uma duração de sensivelmente 1H10 minutos
- O *reviewer* deve rever 6 códigos, 3 em C e 3 em Java
- Os 3 códigos de cada linguagem têm 3 graus de dificuldade distintos (fácil, intermédio e difícil), relacionados com a sua complexidade ciclomática e o seu número de linhas
- Cada código está comentado, com informação sobre a sua funcionalidade
- O participante pode colocar questões durante a experiência, desde que estas não sejam sobre o código
- O código a rever estará limitado a 100 linhas
- O código é compilável, mas o participante não o pode correr
- Os *bugs* presentes no código poderão ser de 4 tipos distintos: *Logical Bug, Performance Bug, Variable Type Bug e Memory Management Bug*
- O número de *bugs* por código é variável, podendo haver códigos sem qualquer bug

- Todos os reviewers devem usar o mesmo IDE para analisar o código
- Para assinalar um bug, basta fazer um comentário junto ao mesmo, não sendo necessário corrigi-lo
- A utilização de ferramentas auxiliares para ajudar na deteção de *bugs* resultará na anulação da participação do reviewer na experiência

8.3.3 Entrega de Tarefas

Antes da *code review* em si, terás de assinar um documento onde consentes a participar na nossa experiência. Todas as tarefas serão te entregues pela plataforma utilizada pela comunicação, tal como a tua realização.

8.4 Apêndice 4a - *BuggyBackpack.java*

```java
import java.util.*;
import java.util.stream.Collectors;

// This code creates represents a backpack, which can be packed with items using different strategies
// Each item has a type, name and weight in Kg and is unique.
// The backpack class stores the items.
// The backpack can be packed with random items, lightest items first,
// heaviest items first, or with a special rule that packs food items separately.
// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug

// The review should take no longer than 10 minutes.
//Review starts on line 61 and ends on line 151.
public class BuggyBackpackReview {

  // type of items
  public enum ItemType {
    FOOD, CLOTHING, BOOK, TOY
  }

  // Create item class
  public static class Item {
    private ItemType type;
    private String name;
    private double weight;

    public Item(ItemType type, String name, double weight) {
      this.type = type;
      this.name = name;
      this.weight = weight;
    }

    public ItemType getType() {
      return type;
    }

    public String getName() {
      return name;
    }
```

```
  public double getWeight() {
    return weight;
  }

  @Override
  public String toString() {
    return name + " (" + type + ", " + weight + " kg)";
  }
}

// Packing strategy
public enum Strategy {
  RANDOM, // Pack random items
  HEAVY_FIRST, // Pack the heaviest items first
  LIGHT_FIRST // Pack the lightest items first
}

// START THE REVIEW HERE
// Create backpack class
public static class Backpack {
  private List<Item> items;

  public Backpack() {
    this.items = new ArrayList<>();
  }

  public void addItem(Item item) {
    items.add(item);
  }

  // Sorts items by weight
  public void sortItemsByWeight() {
    items.sort(Comparator.comparingDouble(Item::getWeight));
  }

  // Displays backpack contents
  public void displayContents() {
    System.out.println("Backpack Contents:");
    for (Item item : items) {
      System.out.println("- " + item);
    }
    System.out.println(this);
  }

  // Packs food items separately from non-food items
  public void packItemsWithSpecialRules(List<Item> availableItems, int totalItemsToPack, int foodIt
    // Simulate a scenario where FOOD items need special handling
    List<Item> foodItems = availableItems.stream()
        .filter(item -> item.getType() == ItemType.FOOD)
        .collect(Collectors.toList());

    List<Item> nonFoodItems = availableItems.stream()
        .filter(item -> item.getType() != ItemType.FOOD)
        .collect(Collectors.toList());

    // Ensure that FOOD items are packed separately to maintain freshness
```

```
    int numFoodItemsToPack = foodItemsToPack;
    int numNonFoodItemsToPack = totalItemsToPack - numFoodItemsToPack;

    packRandomItems(nonFoodItems, numNonFoodItemsToPack);
    packRandomItems(foodItems, numFoodItemsToPack);

    // Display a message indicating the special packing scenario
    System.out.println("Packing items with special rules:");
    System.out.println("Non-FOOD items packed: " + numNonFoodItemsToPack);
    System.out.println("FOOD items packed: " + numFoodItemsToPack);
  }

  public void packItems(List<Item> availableItems, Strategy strategy, int totalItemsToPack) {
    switch (strategy) {
      case RANDOM:
        packRandomItems(availableItems, totalItemsToPack);
        break;
      case HEAVY_FIRST:
        packHeaviestItemsFirst(availableItems, totalItemsToPack);
        break;
      case LIGHT_FIRST:
        packLightestItemsFirst(availableItems, totalItemsToPack);
        break;
    }
  }

  private void packRandomItems(List<Item> availableItems, int totalItemsToPack) {
    Random random = new Random();
    for (int i = 0; i < totalItemsToPack; i++) {
      Item randomItem = availableItems.get(random.nextInt(availableItems.size()));
      addItem(randomItem);
    }
  }

  private void packHeaviestItemsFirst(List<Item> availableItems, int totalItemsToPack) {
    availableItems.sort(Comparator.comparingDouble(Item::getWeight));
    addItemRange(availableItems, totalItemsToPack);
  }

  private void packLightestItemsFirst(List<Item> availableItems, int totalItemsToPack) {
    availableItems.sort(
        Comparator.comparingDouble(Item::getWeight).reversed());
    addItemRange(availableItems, totalItemsToPack);
  }

  private void addItemRange(List<Item> itemsToAdd, int totalItemsToPack) {
    for (int i = 0; i < itemsToAdd.size(); i++) {
      addItem(itemsToAdd.get(i));
    }
  }
}
// REVIEW ENDS HERE

public static void main(String[] args) {
  Backpack backpack1 = new Backpack();
  Backpack backpack2 = new Backpack();
```

```
    Backpack backpack3 = new Backpack();
    Backpack backpack4 = new Backpack();
    List<Item> availableItems = generateAvailableItems();

    // Pack items using different strategies
    backpack1.packItems(availableItems, Strategy.RANDOM, 3);
    backpack2.packItems(availableItems, Strategy.HEAVY_FIRST, 4);
    backpack3.packItems(availableItems, Strategy.LIGHT_FIRST, 4);
    backpack4.packItemsWithSpecialRules(availableItems, 5, 2);

    // Display the contents of the backpack
    backpack1.displayContents();
    backpack2.displayContents();
    backpack3.displayContents();
    backpack4.displayContents();
  }

  private static List<Item> generateAvailableItems() {
    return Arrays.asList(new Item(ItemType.FOOD, "Apple", 0.2),
        new Item(ItemType.FOOD, "Sandwich", 0.5),
        new Item(ItemType.CLOTHING, "T-shirt", 0.3),
        new Item(ItemType.CLOTHING, "Jeans", 0.7),
        new Item(ItemType.BOOK, "Novel", 0.8),
        new Item(ItemType.BOOK, "Notebook", 0.4),
        new Item(ItemType.TOY, "Teddy Bear", 0.6),
        new Item(ItemType.TOY, "Toy Car", 0.3));
  }
}
```

8.5 Apêndice 4b - *BuggyTree.java*

```
import java.util.Scanner;

// This code creates an AVL Tree and performs operations like insertion,
// deletion, searching, etc. The AVL tree is a self-balancing binary search
// tree. In an AVL tree, the heights of the two child subtrees of any node
// differ by at most one.
// If the tree only has one node, the height of the tree is 0.
// Lack of handling of user input errors doesnt count as a bug.

// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug

// The review should take no longer than 15 minutes.
// Please write down the bugs you find.
// The review starts on line 66 and ends on line 167.

//  create Node class to design the structure of the AVL Tree Node
class Node {
    int element;
    int h; // for height
    Node leftChild;
    Node rightChild;
```

```
    // default constructor to create null node
    public Node() {
        leftChild = null;
        rightChild = null;
        element = 0;
        h = 0;
    }

    // parameterized constructor
    public Node(int element) {
        leftChild = null;
        rightChild = null;
        this.element = element;
        h = 0;
    }
}

// create class ConstructAVLTree for constructing AVL Tree
class ConstructAVLTree {
    private Node rootNode;

    // Constructor to set null value to the rootNode
    public ConstructAVLTree() {
        rootNode = null;
    }

    // create removeAll() method to make AVL Tree empty
    public void removeAll() {
        rootNode = null;
    }

    // create checkEmpty() method to check whether the AVL Tree is empty or not
    public boolean checkEmpty() {
        if (rootNode == null)
            return true;
        else
            return false;
    }

    // START THE REVIEW HERE
    // create insertElement() to insert element to to the AVL Tree
    public void insertElement(int element) {
        rootNode = insertElement(element, rootNode);
    }

    // create getHeight() method to get the height of the AVL Tree
    private int getHeight(Node node) {
        return node == null ? 0 : node.h;
    }

    // create maxNode() method to get the maximum height from left and right node
    private int getMaxHeight(int leftNodeHeight, int rightNodeHeight) {
        return leftNodeHeight >= rightNodeHeight ? leftNodeHeight : rightNodeHeight;
    }
```

```
// create insertElement() method to insert data in the AVL Tree recursively
private Node insertElement(int element, Node node) {
    if (node == null)
        node = new Node(element);
    else if (element < node.element) {
        node.leftChild = insertElement(element, node.leftChild);
        if (getHeight(node.leftChild) - getHeight(node.rightChild) == 2)
            if (element < node.leftChild.element)
                node = rotateWithLeftChild(node);
            else
                node = doubleWithLeftChild(node);
    } else if (element > node.element) {
        node.rightChild = insertElement(element, node.rightChild);
        if (getHeight(node.rightChild) - getHeight(node.leftChild) == 2)
            if (element > node.rightChild.element)
                node = rotateWithRightChild(node);
            else
                node = doubleWithRightChild(node);
    } else
        ; // if the element is already present in the tree, we will do nothing
    node.h = getMaxHeight(getHeight(node.leftChild), getHeight(node.rightChild)) + 1;

    return node;
}

// method to perform rotation of binary tree node with left child
private Node rotateWithLeftChild(Node node2) {
    Node node1 = node2.leftChild;
    node2.leftChild = node1.rightChild;
    node1.rightChild = node2;
    node2.h = getMaxHeight(getHeight(node2.leftChild), getHeight(node2.rightChild)) + 1;
    node1.h = getMaxHeight(getHeight(node1.leftChild), node2.h) + 1;
    return node1;
}

// method to perform rotation of binary tree node with right child
private Node rotateWithRightChild(Node node1) {
    Node node2 = node1.rightChild;
    node1.rightChild = node2.leftChild;
    node2.leftChild = node1;
    node1.h = getMaxHeight(getHeight(node1.leftChild), getHeight(node1.rightChild)) + 1;
    node2.h = getMaxHeight(getHeight(node2.rightChild), node1.h) + 1;
    return node2;
}

// method to get total number of nodes in the AVL Tree
public int getTotalNumberOfNodes() {
    return getTotalNumberOfNodes(rootNode);
}

private int getTotalNumberOfNodes(Node head) {
    if (head == null)
        return 0;
    else {
        int length = 1;
        length = length + getTotalNumberOfNodes(head.leftChild);
```

```
            length = length + getTotalNumberOfNodes(head.rightChild);
            int leftHeight = getHeight(head.leftChild);
            int rightHeight = getHeight(head.rightChild);
            getMaxHeight(leftHeight, rightHeight);
            return length;
        }
    }

    // create searchElement() method to find an element in the AVL Tree
    public boolean searchElement(int element) {
        return searchElement(rootNode, element);
    }

    private boolean searchElement(Node head, int element) {
        boolean check = false;
        while ((head != null) && !check) {
            int headElement = head.element;
            if (element < headElement)
                head = head.leftChild;
            else if (element > headElement)
                head = head.rightChild;
            else {
                check = true;
                break;
            }
            check = searchElement(head, element);
        }
        return check;
    }
    // END THE REVIEW HERE

    // create doubleWithLeftChild() method to perform double rotation of binary
    // tree node. This method first rotate the left child with its right child,
    // and after that, node3 with the new left child
    private Node doubleWithLeftChild(Node node3) {
        node3.leftChild = rotateWithRightChild(node3.leftChild);
        return rotateWithLeftChild(node3);
    }

    // create doubleWithRightChild() method to perform double rotation of binary
    // tree node. This method first rotate the right child with its left child and
    // after that node1 with the new right child
    private Node doubleWithRightChild(Node node1) {
        node1.rightChild = rotateWithLeftChild(node1.rightChild);
        return rotateWithRightChild(node1);
    }

    // create inorderTraversal() method for traversing AVL Tree in in-order form
    public void inorderTraversal() {
        inorderTraversal(rootNode);
    }

    private void inorderTraversal(Node head) {
        if (head != null) {
            inorderTraversal(head.leftChild);
            System.out.print(head.element + " ");
```

```
            inorderTraversal(head.rightChild);
        }
    }

    // create preorderTraversal() method for traversing AVL Tree in pre-order form
    public void preorderTraversal() {
        preorderTraversal(rootNode);
    }

    private void preorderTraversal(Node head) {
        if (head != null) {
            System.out.print(head.element + " ");
            preorderTraversal(head.leftChild);
            preorderTraversal(head.rightChild);
        }
    }

    // create postorderTraversal() method for traversing AVL Tree in post-order
    // form
    public void postorderTraversal() {
        postorderTraversal(rootNode);
    }

    private void postorderTraversal(Node head) {
        if (head != null) {
            postorderTraversal(head.leftChild);
            postorderTraversal(head.rightChild);
            System.out.print(head.element + " ");
        }
    }
}

// create AVLTree class to construct AVL Tree
public class BuggyTreeReview {
    // main() method starts
    public static void main(String[] args) {
        // creating Scanner class object to get input from user
        Scanner sc = new Scanner(System.in);

        // create object of ConstructAVLTree class object for costructing AVL Tree
        ConstructAVLTree obj = new ConstructAVLTree();

        char choice; // initialize a character type variable to choice

        // perform operation of AVL Tree using switch
        do {
            System.out.println("\nSelect an operation:\n");
            System.out.println("1. Insert a node");
            System.out.println("2. Search a node");
            System.out.println("3. Get total number of nodes in AVL Tree");
            System.out.println("4. Is AVL Tree empty?");
            System.out.println("5. Remove all nodes from AVL Tree");
            System.out.println("6. Display AVL Tree in Post order");
            System.out.println("7. Display AVL Tree in Pre order");
            System.out.println("8. Display AVL Tree in In order");
```

```
            // get choice from user
            int ch = sc.nextInt();
            switch (ch) {
                case 1:
                    System.out.println("Please enter an element to insert in AVL Tree");
                    obj.insertElement(sc.nextInt());
                    break;
                case 2:
                    System.out.println("Enter integer element to search");
                    System.out.println(obj.searchElement(sc.nextInt()));
                    break;
                case 3:
                    System.out.println(obj.getTotalNumberOfNodes());
                    break;
                case 4:
                    System.out.println(obj.checkEmpty());
                    break;
                case 5:
                    obj.removeAll();
                    System.out.println("\nTree Cleared successfully");
                    break;
                case 6:
                    System.out.println("\nDisplay AVL Tree in Post order");
                    obj.postorderTraversal();
                    break;
                case 7:
                    System.out.println("\nDisplay AVL Tree in Pre order");
                    obj.preorderTraversal();
                    break;
                case 8:
                    System.out.println("\nDisplay AVL Tree in In order");
                    obj.inorderTraversal();
                    break;
                default:
                    System.out.println("\n ");
                    break;
            }
            System.out.println("\nPress 'y' or 'Y' to continue \n");
            choice = sc.next().charAt(0);
        } while (choice == 'Y' || choice == 'y');

        sc.close();
    }
}
```

8.6 Apêndice 4c - *BuggyPharmacy.java*

```
import java.util.*;

// This code creates represents a pharmacy, which has a stock of products that can be purchased by us
// The stock class stores the products and their quantities.
// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
```

```java
// The review should take no longer than 5 minutes.
// Please write down the bugs you find and the type of bug.
// The review starts on line 47.

public class BuggyPharmacyReview {

    // create product class
    public static class Product {
        private String name;
        private double price;
        private String barcode;

        public Product(String name, double price, String barcode) {
            this.name = name;
            this.price = price;
            this.barcode = barcode;
        }

        public String getName() {
            return name;
        }

        public double getPrice() {
            return price;
        }

        public String getBarcode() {
            return barcode;
        }

        @Override
        public String toString() {
            return name + " (Barcode: " + barcode + ", Price: $" + price + ")";
        }
    }

    // START THE REVIEW HERE

    // create stock class, which has a map of products and their quantities
    public static class Stock {
        private Map<Product, Integer> products;

        public Stock() {
            this.products = new HashMap<>();
        }

        // add a certain quantity of product to stock
        public void addProduct(Product product, int quantity) {
            products.put(product, quantity);
        }

        public void removeProduct(Product product) {
            products.remove(product);
        }
```

```
    // updates the product quantity on stock
    public void updateQuantity(Product product, int quantity) {
        products.put(product, quantity + 1);
    }

    // purchase products from stock
    public void purchase(User user, Map<Product, Integer> productsToPurchase) {
        System.out.println(user.getName() +
                " is purchasing the following products:");
        for (Map.Entry<Product, Integer> productEntry : productsToPurchase.entrySet()) {
            int availableQuantity = products.get(productEntry.getKey());
            if (availableQuantity > 0) {
                System.out.println("- " + productEntry.getKey() +
                        " (Quantity: " + productEntry.getValue() + ")");
                updateQuantity(productEntry.getKey(), availableQuantity - 1);
            } else {
                System.out.println("- " + productEntry.getKey() + " (Out of stock)");
            }
        }
    }

    @Override
    public String toString() {
        return "Stock: " + products;
    }
}

// create user class
public static class User {
    private String name;
    private String phone;
    private float fiscalNumber;

    public User(String name, String phone, float fiscalNumber) {
        this.name = name;
        this.phone = phone;
        this.fiscalNumber = fiscalNumber;
    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    public float getFiscalNumber() {
        return fiscalNumber;
    }

    @Override
    public String toString() {
        return "User: " + name + " (Phone: " + phone +
                ", Fiscal Number: " + fiscalNumber + ")";
    }
```

```
    }

    public static void main(String[] args) {
        // Sample products
        Product paracetamol = new Product("Paracetamol", 5.99, "123456");
        Product aspirin = new Product("Aspirin", 3.49, "789012");

        // Sample Stock
        Stock Stock = new Stock();
        Stock.addProduct(paracetamol, 50);
        Stock.addProduct(aspirin, 30);

        // Sample user
        User customer = new User("John Doe", "+123456789", 39288123);

        Map<Product, Integer> productsToPurchase = new HashMap<BuggyPharmacyReview.Product, Integer>(
        productsToPurchase.put(aspirin, 2);
        productsToPurchase.put(paracetamol, 1);
        Stock.purchase(customer, productsToPurchase);
    }
}
```

8.7 Apêndice 4d - *BuggyPyramid.java*

```
/*This program uses a function to print a pyramid 'like' structure, increment the number (+1) in ever
        1
       2 3
      4 5 6
     6 7 8 9
*/

// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug

// The review should take no longer than 5 minutes.
// Review starts on Line 23

#include <stdio.h>

// Functions
void printPyramid(int rows);

// REVIEW STARTS HERE

int main() {
   printPyramid(5);

   return 0;
}

/*****************************************************************
Function: print pyramid 'like' pattern, increment numbers throughtout
```

```
the positions of the pyramid
Parameters: rows - number of rows.
Comments:
Out: void
*********************************************************************/
void printPyramid(int rows){
   int i, j = 1; // counters
   int spaces = 1; // spaces and rows for user input
   int k, numb = 1;  // k and t for spaces

   spaces = rows + 4 - 1;  // initial number of spaces

   for (i = 1; i < rows; i++) {
      for (k = spaces; k >= 1; k--) {  // print spaces before numbers
         printf(" ");
      }

      for (j = 1; j <= i; j++) {  // loop to print numbers based on the current row.
         printf("%d ", numb++);
         numb++;
      }

      printf("\n");  // move to the next line for the next row.
      spaces--;  // decrement the number of spaces for the next row.
   }
}
```

8.8 Apêndice 4e - *BuggySalaries.java*

```
/*
This C program is designed to manage employee data. It dynamically allocates memory for an array of e
 */
// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug

// The review should take no longer than 10 minutes.
// Review starts on Line 24 and ends in line 116

#include <stdio.h>
#include <stdlib.h>

// define Struct
typedef struct {
    int id;
    char name[50];
    float salary;
} Employee;

// REVIEW STARTS HERE

int main() {
    int employeeCount;
    printf("Enter the number of employees: ");
```

```
    scanf("%d", &employeeCount);

    if (employeeCount <= 0) {
        printf("Please enter a positive number of employees.\n");
        return 1;
    }

    Employee *employees = (Employee *)malloc(sizeof(Employee) * employeeCount);

    if (!employees) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    for (int i = 0; i < employeeCount; i++) {
        printf("Enter details for employee %d:\n", i + 1);
        employees[i].id = i + 1;

        printf("Name: ");
        scanf("%s", employees[i].name);

        printf("Salary: $");
        scanf("%f", &employees[i].salary);

        printf("\n");
    }

    // print employees details
    printf("Employee details before salary update:\n");
    for (int i = 0; i < employeeCount; i++) {
        printEmployeeDetails(&employees[i]);
    }

    // Update salaries
    for (int i = 0; i < employeeCount; i++) {
        updateSalary(&employees[i], 10.0);
    }

    // Print employees details
    printf("\nEmployee details after salary update:\n");
    for (int i = 0; i < employeeCount; i++) {
        printEmployeeDetails(&employees[i]);
    }

    // calculate the average salary for all employees
    float averageSalary = calculateAverageSalary(employees, employeeCount);
    printf("\nAverage Salary: $%.2f\n", averageSalary);

    free(employees);

    return 0;
}
/************************************************************
* Function: Update salary
*
* Parameters: employe structure pointer and percent float
```

```c
* Comments:  increase 10% in main
* Out: void
*****************************************************************/
void updateSalary(Employee *employee, float percent) {
    employee->salary *= (1 + percent);
}
/*****************************************************************
* Function: print employee details
*
* Parameters: employee structure pointer
* Out: void
*****************************************************************/
void printEmployeeDetails(const Employee *employee) {
    printf("ID: %d\n", employee->id);
    printf("Name: %s\n", employee->name);
    printf("Salary: $%.2f\n", employee->salary);
}
/*****************************************************************
* Function: calculate an average of the salary of all employees
*
* Parameters: pointer to the struct of employees; count variable
* Out: average salary
*****************************************************************/
float calculateAverageSalary(const Employee *employees, int count) {
    float totalSalary = 0;

    for (int i = 0; i < count; i++) {
        totalSalary += employees[i].salary;
    }

    return totalSalary / count;
}
// REVIEW ENDS HERE
```

8.9 Apêndice 4f - *BuggyTennisTournament.java*

```c
/*This program uses a binary tree structure for displaying
        a tournament of tennis. The main function shows information
        about the winner, the structure of the tournament, the
        matches played, etc. It reads this information from a textfile
        with this format: [player][num_of_sets]

    Example: "Tournament.txt"
    Jose;0
    Daniel;3
    Peter;1
    Diogo;2
*/

// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug
```

```c
// The review should take no longer than 15 minutes.
// Review starts on Line 81 and ends in line 150

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_NAME 20
#define STAGES 15
#define EMPTY NULL
#define NO_LINK NULL

// Define Structs
typedef struct _PLAYER{
    char name[MAX_NAME];
    int sets;
}PLAYER;

typedef struct _BTREE_NODE{
    void *data;
    struct _BTREE_NODE* left;
    struct _BTREE_NODE* right;
}BTREE_NODE;

// Define Macros
typedef BTREE_NODE* BTREE;
typedef enum _BTREE_LOCATION{BTREE_LEFT, BTREE_RIGHT} BTREE_LOCATION;
typedef enum _BOOLEAN {FALSE = 0, TRUE = 1} BOOLEAN;
typedef enum _STATUS {ERROR = 0, OK = 1} STATUS;

// REVIEW STARTS HERE

int main(){
        BTREE Btree;
        void* players [STAGES];
        char file_name [MAX_NAME];

        printf("Nome do ficheiro: ");
        scanf("%s", file_name);

        if (readPlayersFromFile(players, file_name))
        {
        Btree = createBtree(players, 0, STAGES);

                printf("\nGames played by the Tournament Winner: %d\n");
                printWinnerGames(Btree);

                printf("\nSets won by the Tournament Winner: %d\n", countWinnerSets(Btree));
        }
        else
                printf("Error Reading File\n");
    return 0;
}
/*************************************************************
* Function: Read players from file
*
```

```
* Parameters: double pointer to player type structure; pointer to filename
* Comments:  scans from a file with the structure mentioned above. Allocates the player, and fills th
* Out: ERROR if Problem ocorred in function. File pointer if OK
*****************************************************************/

// REVIEW STARTS HERE

STATUS readPlayersFromFile(void** players, char* file_name)
{
        FILE* fp;
        int j, i = 0;
        void* ptr_data;
        if ((fp = fopen(file_name, "r")) != NULL)
        {
                while (!feof(fp))
                {
                        if ((ptr_data = malloc(sizeof(PLAYER))) != NULL)
                        {
                                fscanf(fp, "%d[^;];", ((PLAYER*)ptr_data)->name);
                                fscanf(fp, "%d\n", &(((PLAYER*)ptr_data)->sets));
                                players[i] = ptr_data;
                                i++;
                        }
                        else
                        {
                                for (j = i; j >= 0; j--)
                                        free(players[j]);
                                return(ERROR);
                        }
                }
                fclose(fp);
                return(OK);
        }
        else
                return(ERROR);
}
/*****************************************************************
* Function: Count winner sets
* Comments: Uses a function "btreeLeaf" to know if a node is a Leaf (this function returns TRUE if no
* Parameters: Binary tree
* Out: the number of winner sets
*****************************************************************/
int countWinnerSets(BTREE btree)
{
    int count = 0;
    BTREE BT=btree;
    count += ((PLAYER*)BT->data)->sets;
    if (btree != NULL && !bTreeLeaf(btree)) {
        if (!strcmp(((PLAYER*)btree->left->data)->name, ((PLAYER*)btree->data)->name)){
            count +=  countWinnerSets(BT->left);
        }
        else
        {
            count += countWinnerSets(BT->right);
        }
    }
```

```
    return (count);
}

/*************************************************************
* Function: Print winner games
* Parameters: A binary Tree
* Comments: prints the game. To continue this logic, compares which player name is the same (one tier
Uses a function "btreeLeaf" to know if a node is a Leaf (this function returns TRUE if node is a leaf
* Out: void
*************************************************************/
void printWinnerGames(BTREE btree)
{
    if (btree != NULL && !bTreeLeaf(btree)) {
        printf("%s sets -> %d : %s sets -> % d => Winner : %s \n", ((PLAYER*)btree->left->data)->name

        if (!strcmp(((PLAYER*)btree->left->data)->name, ((PLAYER*)btree->data)->name))
            printWinnerGames(btree->left);
        else
            printWinnerGames(btree->right);
    }
        return;
}

// REVIEW ENDS HERE
```

Referências

[1] Jones, Capers (June 2008). "Measuring Defect Potentials and Defect Removal Efficiency"(PDF). Crosstalk, The Journal of Defense Software Engineering. Archived from the original (PDF) on 2012-08-06. Retrieved 2010-10-05.

[2] Jason Cohen (2006). Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.). Smart Bear Inc. ISBN 978-1-59916-067-2.

[3] Czerwonka, Jacek; Greiler, Michaela; Tilford, Jack (2015). "Code Reviews do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down". 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (PDF). Vol. 2. pp. 27–28. doi:10.1109/ICSE.2015.131. ISBN 978-1-4799-1934-5. S2CID 29074469. Retrieved 2020-11-28.

[4] Pei Breivold, Hongyu. (2023). Using Software Evolvability Model for Evolvability Analysis.

[5] Best Practices for Code Review — SmartBear. (n.d.). Retrieved December 5, 2023, from https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/

[6] How to Make Good Code Reviews Better - Stack Overflow. (n.d.). Retrieved December 5, 2023, from https://stackoverflow.blog/2019/09/30/how-to-make-good-code-reviews-better/

[7] How to do a code review — eng-practices. (n.d.). Retrieved December 5, 2023, from https://google.github.io/eng-practices/review/reviewer/