INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Reparação automática de programas: Exploração do potencial em contexto industrial

ANDRÉ EMANUEL GOMES QUEIRÓS
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Reparação automática de programas

Exploração do potencial em contexto industrial

André Emanuel Gomes Queirós

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Alberto Sampaio

Co-orientador: Isabel Sampaio

Porto, Outubro 2021

Abstract

Today the automatic repair of programs is an issue increasingly discussed. The discussion of it comes from the evolution and advancement of technology, namely the environments of continuous integration and continuous delivery (CI/CD). These environments imply that the delivery of functional software is increasingly accelerated, and sometimes defects delay this process, implying the traditional debugging by a programmer, with the aim of detecting the defect and repairing it with quality.

Automatic programming has been showing an increasing evolution, namely through techniques of automatic repair of APR programs. These techniques focus on the identification, localization, generation of correction and verification of the repaired version of the defective program, avoiding the programmer's effort in the identification and debugging process.

This work aims at experimenting with two automatic repair tools, Arja and jGenProg, implanted in a pipeline, in an industrial context. The tools evaluation was done through the process of comparing them between themselves and with manual repairing process, and with the application of two questionnaires.

In conclusion, both tools were able to be implanted in a pipeline, despite the necessary time required. The jGenProg tool revels a better performance, because it could detect a suspicious case and presented an even more completed report. However, the inquired developers choose the manual repairing process, against any of the evaluated tools.

Keywords: Automatic Program Repair (APR), Continuous Integration (CI), Continuous Delivery (CD), APR tools, Real world

Resumo

A reparação automática de software ou programas é uma temática que, hoje em dia, é cada vez mais discutida. A discussão da mesma advém da evolução e do avanço da tecnologia, nomeadamente os ambientes de integração e entrega contínuos. Estes ambientes requerem que a entrega de software funcional seja cada vez mais acelerada, sendo que por vezes os defeitos atrasam este processo, implicando a tradicional depuração por parte de um programador, com o objetivo de detetar o defeito e o reparar com qualidade.

A programação automática tem vindo a demonstrar uma crescente evolução, nomeadamente através de técnicas de reparação automática de programas. Estas técnicas concentram-se na identificação, localização, geração de correção e verificação da versão reparada do programa com defeito, evitando o esforço do programador no processo de identificação e depuração.

Este trabalho visa a experimentação de duas ferramentas de reparação automática, a Arja e a jGenProg, implantadas numa *pipeline*, em contexto industrial. A avaliação das ferramentas foi efetuada através da comparação entre elas e com a reparação manual e recorrendo a dois questionários.

Conclui-se que ambas as ferramentas são passíveis de implantar em *pipeline*, apesar do tempo necessário poder ser superior ao desejado. A ferramenta jGenProg apresentou um melhor desempenho, pois conseguiu detetar um caso suspeito e apresenta um relatório mais completo. Todavia, os programadores inquiridos preferem a reparação manual a quaisquer uma das duas ferramentas avaliadas.

Agradecimentos

Reconhecer e agradecer a todos os que apoiaram o desenvolvimento deste trabalho é essencial.

Agradeço aos meus colegas de trabalho, e a todos os integrantes da empresa Medsky, pela possibilidade da realização deste trabalho e disponibilização de recursos para a sua experimentação e concretização. Um agradecimento especial aos quatro colegas da área da programação pela disponibilidade em participar nos questionários realizados e pela partilha de experiências e conhecimento.

Gostaria também de agradecer ao professor Alberto Sampaio pela sua disponibilidade e apoio ao longo deste trabalho. A sua orientação, assim como a da professora Isabel Sampaio, foram essenciais para a realização, revisão e correção do trabalho.

Por último, agradecer à minha família, em especial à minha esposa, por todo o apoio e força dada ao longo do trabalho e nunca me deixarem desistir.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Equações

Lista de Códigos

Acrónimos e Símbolos

Lista de Acrónimos

AST	*Abstract Syntax Tree*
APR	Reparação Automática de Programas (*Automatic Program Repair*)
CI	Integração Contínua (*Continuous Integration*)
CD	Entrega Contínua (*Continuous Delivery*)
FAST	*Function Analysis and System Technique*
NCD	*New Concept Development*
NPD	*New Product Development*
SBFL	*Spectrum-Based Fault Localization*
SPA	*Single Page Application*
TOPSIS	*Technique for Order of Preference by Similarity to Ideal Solution*
TSG	*Technology Stage Gate*

1 Introdução

Este capítulo pretende clarificar a temática e organização do presente trabalho, cuja principal motivação baseia-se na pesquisa e experimentação de ferramentas de Reparação Automática de Programas (APR).

1.1 Contexto

No decorrer do desenvolvimento, o programador depara-se frequentemente com defeitos que custam tempo (Tao et al., 2014). e, consequentemente, dinheiro. De forma a otimizar a correção dos defeitos, surgiram ferramentas que visam reparar automaticamente o *software*. Apesar da tendência de utilização crescente, torna-se necessário reforçar o estudo da sua utilização em contexto industrial. Através das novas formas de desenvolvimento e entrega contínuas de código, associadas às crescentes metodologias ágeis implementadas na indústria, considera-se importante analisar ferramentas que permitam reparar automaticamente o código, sempre que seja detetado algum defeito.

O que motivou a realização deste trabalho foi a vontade em pesquisar e experimentar ferramentas de APR. Para tal, selecionou-se a empresa onde o investigador desenvolve a sua atividade laboral como o ambiente de experimentação. A empresa desenvolve a sua atividade na área da saúde e recorre a *pipelines* para o desenvolvimento e entrega contínuas de *software* com o *runtime NodeJs*. Dada a incapacidade em encontrar ferramentas adaptadas a este *runtime*, com a autorização da empresa, migrou-se parte do projeto para a linguagem Java, pelo que se opta pela experimentação de ferramentas adaptadas à linguagem em questão, como se irá abordar.

1.2 Problema

A depuração de erros no *software* continua a ser um processo moroso, difícil e desgastante. Além disso, a intervenção humana no processo de depuração e localização do defeito consome demasiado tempo, implicando um aumento exponencial dos custos do *software* (Phung & Lee, 2021). Na indústria, estes custos consideram-se substanciais pois envolvem os programadores em processos de depuração que se desdobram, não apenas na localização e correção de um defeito, mas também na verificação da reparação. A automatização deste processo procura facilitá-lo, tornando-se necessário verificar a sua aplicabilidade na indústria.

1.3 Objetivos

Considerando o problema exposto, o propósito deste estudo é analisar e experimentar ferramentas de APR em contexto profissional. Para tal, pretende-se:

- Contribuir para o aumento do conhecimento relativo à utilização de ferramentas de reparação automática em contexto industrial;
- Compreender o impacto que a sua utilização possa trazer nos processos de entrega e integração contínuas;
- Implantar duas ferramentas de APR em *pipeline*, num intervalo de tempo adequado;
- Comparar os resultados obtidos através da implantação das ferramentas com os resultados obtidos sem recurso à automatização;
- Avaliar a satisfação dos programadores com as ferramentas selecionadas;
- Estudar o impacto, a nível de recursos, custos e tempo, dos achados da investigação na utilização da APR em contexto industrial.

1.4 Abordagem

De forma a responder aos objetivos do estudo, delinearam-se as seguintes etapas para o desenvolvimento do trabalho:

1. Estudo do estado da arte – Rever a literatura e analisar técnicas e ferramentas existentes que permitam a APR;

2. Analisar e comparar, empiricamente, ferramentas – Classificar as ferramentas que trazem um maior valor à APR no contexto do ambiente de experimentação;

3. Selecionar e implantar a ferramenta – Escolher as ferramentas mais adequadas e proceder à sua implantação prática;

4. Avaliar o processo de experimentação e a satisfação dos programadores – Extrair os resultados das implantações desenvolvidas, e compará-los com a correção manual de defeitos.

1.5 Estrutura do documento

O documento encontra-se estruturado nos seguintes capítulos:

- Estado da arte: introduz o que existe acerca da informação, explicando os conceitos necessários à compreensão da dissertação;
- Análise de valor: propõe de que forma este trabalho vai criar valor para a investigação;
- Análise e conceção da solução: introduz o planeamento da implantação e avaliação, bem como as metodologias, a definição das grandezas e as hipóteses em teste;
- Implantação: explica o processo de implantação das ferramentas em um *pipeline* de reparação;
- Discussão dos resultados: apresenta os resultados e discute-os com base na mais recente evidência científica encontrada;

2 Estado da Arte

O presente capítulo apresenta técnicas e ferramentas utilizadas na APR. Através do estudo do estado da arte torna-se possível compreender a história e os estudos elaborados nesta área, nomeadamente as técnicas existentes de localização, verificação e correção que conduzem a uma versão do programa reparado.

2.1 Visão geral

A deteção de defeitos é um passo importante na procura pela melhoria do *software.* Essa mesma deteção conduz à localização, correção e verificação do defeito num programa. De facto, a APR surge com o propósito de sugerir correções para determinados defeitos de um programa (le Goues et al., 2019).

A construção de um *software* surge a partir de determinadas especificações. Por vezes, uma dessas especificações pode falhar, dando início ao processo de reparação. Esse processo é iniciado, baseando-se em testes de *software*, de modo a gerar uma versão corrigida que seja capaz de responder à especificação com defeito (Tao et al., 2014).

Recentemente considerou-se a investigação da APR em contexto real, através das técnicas de reparação de programas já estudadas (Aleti & Martinez, 2020). De forma geral, a ideia consiste na tentativa de reparar automaticamente sistemas de *software*, capacitando uma correção gerada que seja validada pelo programador antes de ser, por último, aceite, ou, até mesmo, adaptada de forma a ser incorporada no sistema. Os benefícios da utilização das técnicas de reparação automática baseiam-se na capacidade de fornecer uma solução possível,

evitando o esforço de identificação e correção de erros (Gazzola et al., 2019; Mousavi et al., 2020).

As primeiras abordagens na APR consistiram em definir modos de falta, possibilitando a deteção de defeitos mais frequentes em programas para o desenho de circuitos digitais que permitiam testar os circuitos antes de os mesmos serem produzidos. Estes modos de falta verificam automaticamente o tipo de falta, e quando a reconhecem, adotam a estratégia mais correta (Mousavi et al., 2020).

Para se garantir a linha temporal, numa primeira instância procede-se à substituição de componentes de um sistema, cujo objetivo é a reparação, tendo em conta o ambiente onde esse mesmo sistema opera e que provoca a falha da reparação (Nguyen et al., 2013).

Segundo Gazzola et al, (2019), uma outra abordagem consiste em encontrar o menor conjunto de alterações a realizar nas chamadas aos métodos de um programa, para satisfazer as especificações. Esta abordagem foi estudada a partir da reparação de um programa que viola uma especificação de política de segurança que especifica os caminhos permitidos no gráfico de fluxo de controlo do mesmo.

2.1.1 Software Repair e Software Healing

No tratamento automático de defeitos, consideram-se duas abordagens: *software repair*, a reparação automática de programas propriamente dita; e *software healing*. Embora ambas tenham conceitos semelhantes, também apresentam diferentes propósitos e diferentes soluções relativamente aos defeitos e, por vezes, um funcionamento diferente no momento em que a APR é despoletada (Carbin Michael and Misailovic, 2011; Mousavi et al., 2020).

Na abordagem de *software repair* existe o processo de detetar o defeito, mas também de localizar onde este consiga ser corrigido, realizando os ajustes necessários e prevenindo defeitos semelhantes. Esses ajustes podem ser gerados numa fase de testes e de desenho, refletindo-se no código fonte (Psaier & Dustdar, 2011).

O conceito de *software repair* tem como finalidade a correção do defeito, e não a sua prevenção. No decorrer deste processo, as falhas são toleradas para que a informação relevante possa ser extraída e explorada pelas soluções de reparação de *software* com o intuito de atingir a identificação e correção das mesmas (Gazzola et al., 2019).

No processo de *repair* a falha é detetada a partir de um conjunto de observações acerca do defeito e de execuções sem a sua inclusão. Existe, então, um processamento de forma a identificar e corrigir a origem dos defeitos. Este processo não previne as falhas, por outro lado, explora a informação armazenada durante o processo de identificação das mesmas (Hua et al., 2018).

Segundo Martinez & Monperrus. (2019a), a forma como a deteção de falhas serve os processos *healing* e *repair* também influencia a implementação do componente de deteção de falha. Quando a deteção do defeito é utilizada como parte do processo de reparação, existe uma implementação para verificar o estado do programa. Para este caso, podem utilizar-se as *assertions*, incluídas nos casos de teste. Neste ponto, existe a necessidade de distinguir o comportamento correto desejado, do comportamento incorreto. Quando estes testes são inexistentes, torna-se necessário a intervenção humana para que se observe que o comportamento seja o mais adequado (Gazzola et al., 2019).

O processo de reparação integra três etapas, como apresentado na Figura 1. A primeira etapa passa pela localização espacial onde a correção pode ser aplicada, mesmo podendo existir outros locais onde as correções podem ser aplicadas. A segunda etapa é a correção propriamente dita ou *patch*, que se manifestam em correções que alteram o *software* nos locais referenciados pelo passo anterior. A última é a verificação, onde existe uma confirmação face ao defeito que identifica o funcionamento da correção no *software*. De facto, o processo pode ser efetuado múltiplas vezes, tanto nesta etapa como na da correção, até que a falha se encontre corrigida e que nenhuma outra correção consiga ser gerada, ou até que o tempo alocado ao processo de reparação atinja o seu fim (le Goues et al., 2019).

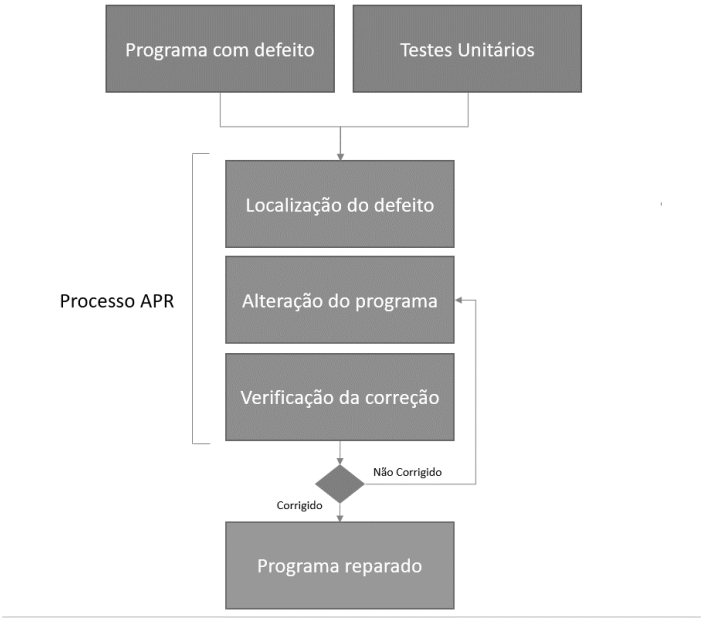

Figura 1 – Processo de reparação (Adaptado de: Assiri et al., (2017))

Os testes e a sua cobertura são importantes para as técnicas de reparação. A partir deles, é possível gerar várias correções baseadas no seu estado, consoante a sua aprovação ou reprovação. Além disso, as técnicas de reparação de *software* exploram blocos de múltiplas execuções e, de modo a gerar correções, incluí nos seus conjuntos blocos com falhas e sem falhas (Zuddas et al., 2014).

O processo de reparação é despoletado a partir da existência de uma execução falhada do programa com defeito, sendo que no final do processo são esperados dois resultados distintos. Um dos resultados é a correção do programa, o que significa que as execuções falharam, todavia, o programa é corrigido. A outra alternativa de resultado é o programa continuar em falha após as execuções falharem e o processo ter sido executado (DeMarco et al., 2014; Gazzola et al., 2019; le Goues et al., 2019).

De forma a melhorar compreender a APR, nos subcapítulos 2.2 e 2.3 serão exploradas as diferentes etapas desta abordagem. Posteriormente, no subcapítulo 2.4, serão analisadas e comparadas as ferramentas e técnicas mais úteis na realização deste estudo.

Relativamente ao conceito de *software healing*, o princípio baseia-se na deteção do defeito do *software* em ambiente de produção, apresentando, para o efeito, todos os ajustes necessários para o sistema retomar o seu fluxo normal. Esses ajustes não se refletem no código fonte do programa, mas sim, ao nível do *runtime* do sistema onde o programa está a ser executado.

Todo este processo pode ser repetido várias vezes, nos momentos em que um erro do mesmo tipo é encontrado (Y. Liu et al., 2018).

Este conceito baseia-se na deteção de falhas e atua numa fase embrionária detetando os sintomas dos defeitos e prevenindo problemas graves que possam levar à perda de dados através de falhas do sistema (Agarwal & Agrawal, 2014).

O processo de *healing* consiste na previsão de uma operação que corrige e previne ou então, mitiga a falha que foi detetada. A verificação tenta perceber de que forma a aplicação está a ser executada imediatamente após o passo de *healing* ser realizado. Não obstante, algumas operações de *healing* são aplicadas apenas quando se torna possível garantir que o sistema é afetado de forma neutra ou positiva (Y. Liu et al., 2018; Mousavi et al., 2020).

De facto, quando se comparam os processos de reparação e de *healing* torna-se possível identificar de forma imediata, que o processo de reparação contém mais um passo. Contrariamente ao processo de *healing* que reage diretamente às falhas, o processo de *repair* primeiro localiza as falhas para que, em seguida, as consiga corrigir (Aleti & Martinez, 2020; Y. Liu et al., 2018).

Sempre que seja detetada uma falha que dá início a uma operação de *healing,* existe uma consequência negativa sobre o sistema que a opera. Nesta situação, e conforme se pode observar na Figura 2, o processo de *healing* volta a ser repetido e verificado até que a falha seja corrigida ou até que mais nenhuma ação possa ser aplicada (Mousavi et al., 2020). Naturalmente, este processo exige uma maior rapidez, uma vez que o mesmo é executado enquanto a aplicação está a correr.

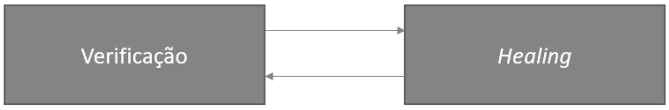

Figura 2 – Processo de healing (Adaptado de: Gazzola et al., (2019))

Considerando que este processo é orientado para a execução de *software*, mais concretamente para a transformação de uma falha que origina o problema, duas situações podem emergir a partir dele:

- Execução bem-sucedida ou programa defeituoso: uma execução é corrigida mas o programa continua com defeito;
- Execução falhada ou programa defeituoso: as execuções não são corrigidas mas o programa continua com defeito.

As técnicas de *repair* e *healing* reagem a falhas através da execução de algumas operações. Elas distinguem-se em dois tipos de operações. Operações de *healing*, executadas num contexto de *runtime,* transformando a falha numa execução bem-sucedida e operações de reparação, executadas no código fonte do programa com o objetivo de remover a falha (Assiri & Bieman, 2017).

As técnicas podem originar duas respostas para o problema. Uma delas é a forma de o contornar, a outra resposta é através das correções. A primeira consiste em uma solução temporária e não otimizada para resolver rapidamente um defeito de *software* e manter o programa ativo, enquanto os programadores trabalham numa forma mais definitiva para resolver o problema. Já o inverso ocorre no caso da correção*,* em que a solução é permanente para um defeito de *software* e cuja qualidade detém o mesmo nível que uma correção desenvolvida por um programador (Gazzola et al., 2019; Mousavi et al., 2020).

Os processos de *repair* ou *healing* permitem a correção dos defeitos de *software* e têm por objetivo garantir a disponibilidade do *software*, independentemente dos defeitos que possam ocorrer (Psaier & Dustdar, 2011). A identificação e remoção permanente do fator que causa a falha não é o objetivo principal destas técnicas, pelo que normalmente são utilizadas operações que mascaram as falhas, resultando na minimização do impacto que as mesmas possam trazer (Assiri & Bieman, 2017).

Apesar de ambos os conceitos serem importantes, no âmbito deste trabalho considera-se a abordagem de *software repair* como central. Desta forma, explicado o conceito de *software healing*, o restante documento remete-se somente a *software repair*, explorando-o empiricamente.

2.1.2 Orquestração de Pipelines

A orquestração de *pipelines* é um conceito bastante interessante utilizado em contexto industrial. Esta permite que após uma alteração realizada ao código fonte, esse mesmo seja

disponibilizado e refletido rapidamente no seu ambiente de produção. Naturalmente, este processo de automatização carece de validação relativamente ao seu código base (Gaikwad & Shah, 2015).

A entrega contínua requer, frequentemente, testes de todos os tipos. De facto, o seu objetivo passa pela redução dos esforços manuais para testar, bem como da necessidade de programadores responsáveis por se focarem em determinadas áreas de testes, de modo a detetarem possíveis problemas quanto antes. Por isto, torna-se necessário acelerar os testes (Gazzola et al., 2019).

Numa *pipeline* para além da solução ser validada a partir dos seus testes unitários, também se torna possível integrar, numa fase final do trabalho, o processo de reparação automática, de forma a corrigir o programa e notificar o programador acerca do erro e respetiva sugestão de reparação.

2.2 Software repair: Localização de Defeitos

A primeira etapa da APR consiste na localização do defeito. Esta fase revela-se essencial uma vez que o processo de reparação apenas pode ser executado após a localização de um defeito. Assim, uma reparação bem-sucedida não depende apenas da correção, mas também da localização do defeito a reparar (Wei et al., 2010).

Relativamente à localização, podem distinguir-se dois métodos pelo local a ser corrigido, o defeito ou a correção em si. No primeiro, o objetivo é localizar o defeito e espera-se que as declarações em falta identificadas se transformem em correções (Mousavi et al., 2020). Por outro lado, a localização da correção pretende detetar declarações onde o efeito da falta possa ser observado e eliminado. Não obstante a diferença entre estes métodos, um pode complementar o outro. A técnica de localização de falhas pode também identificar a correção e, da mesma forma, o inverso também pode acontecer, sendo o defeito identificado através da técnica de localização da correção (Goues et al., 2013).

Relativamente às técnicas de localização do defeito, pode afirmar-se que estas se apoiam num mecanismo geral e extensivo denominado *Spectrum-Based Fault Localization* (SBFL). De forma geral, o SBFL baseia-se em dados recolhidos no decorrer da execução dos testes, analisando informações acerca do comportamento do *software* para identificar elementos do programa

que se encontram com defeito. Este mecanismo gera uma lista de elementos, classificados com uma probabilidade de terem algum defeito, baseando-se nos testes e nos seus resultados (Zuddas et al., 2014).

De uma forma geral, quando a maioria dos testes falha e poucos são aprovados em determinados elementos do programa, revela-se intuitivo que o programa é, provavelmente, defeituoso. Contrariamente, o inverso também acontece e quando a maioria dos testes são aprovados e existem poucos em falha, a probabilidade de o programa estar correto aumenta (Wei et al., 2010).

Sempre que a técnica de reparação funcionar ao nível das declarações do programa, o algoritmo SBFL utilizado para detetar a localização do defeito também funciona ao nível da declaração, ou seja, a localização de falhas e a reparação do programa funcionam no mesmo nível de granularidade. Contudo, as entidades do programa que tenham diferentes tipos de granularidade podem perfeitamente ser considerados para a localização, como é o caso das declarações, ramos e caminhos (Mousavi et al., 2020).

De forma a implementar o SBFL, revela-se necessário recorrer a um conjunto de testes com aprovação e reprovação. Quando ocorre a reprovação do teste, importa identificar o defeito e, posteriormente, repará-lo. Neste âmbito, foi possível compreender que têm sido utilizadas três técnicas de localização de falhas com maior frequência: Tarantula, Ochiai e Jaccard (Gazzola et al., 2019; Mousavi et al., 2020).

Analisando a técnica *Tarantula*, verifica-se que se procede ao cálculo de uma declaração suspeita como a proporção entre a taxa de casos com falha que executam essa mesma declaração e, a soma de testes com aprovação e os testes de reprovação. Assim, o cálculo de casos suspeitos é dado pela seguinte fórmula (Mousavi et al., 2020; Wong et al., 2016):

$$\text{Casos suspeitos(S)} = \frac{\frac{\text{Testes Reprovados(S)}}{\text{Testes totais falhados}}}{\frac{\text{Testes Aprovados(S)}}{\text{Testes totais aprovados}} + \frac{\text{Testes Reprovados(S)}}{\text{Testes totais falhados}}} \quad (1)$$

Por outro lado, *Ochiai* procede ao cálculo de declarações suspeitas como sendo o rácio entre o número de testes reprovados e, a raiz quadrada do produto entre os testes totais reprovados e a soma entre os testes reprovados e aprovados (Wong et al., 2016).

$$\text{Casos suspeitos(S)} = \frac{\text{Testes Reprovados(S)}}{\sqrt{\text{Total Reprovados(S)} \times (\text{Testes Aprovados(S)} + \text{Testes Reprovados(S))}}} \quad (2)$$

Por último, *Jaccard* calcula o caso suspeito em relação a uma declaração suspeita como uma proporção entre o número de testes com falha que a executa e a soma entre o número total de testes executados e os testes em falha que não a executam (Y. Liu et al., 2018; Wong et al., 2016).

$$\text{Casos suspeitos (S)} = \frac{\text{Testes Reprovados(S)}}{\text{Testes executados(S)} \times (\text{Total Testes reprovados} + \text{Testes Reprovados(S))}} \quad (3)$$

Para um maior número de testes com falha a executar uma instrução correspondem pontuações mais altas para as técnicas *Tarantula*, *Ochiai* e *Jaccard*. O inverso também acontece, verificando-se que para um maior número de testes aprovados a executar uma instrução existe uma correspondência de uma diminuição das pontuações para as mesmas 3 técnicas (Mechtaev et al., 2016; Wong et al., 2016).

Relativamente à localização das correções, estas pretendem identificar locais onde o programa pode não conter faltas, mas em que é potencialmente modificável para permitir que o mesmo funcione corretamente. Um comportamento com defeito pode, grande parte das vezes, não remover a falha, mas ser corrigido de modo a compensar o seu efeito (Z. Qi et al., 2015; Wong et al., 2016).

Considerando a localização da correção, tome-se como exemplo o seguinte algoritmo apresentado no estudo de Gazzola:

```
if( x == 0 && y == 0) {
    printf("%d"", y);
    exit(0);}
while(y != 0){
    if(x > y)
        x = x - y;
    else
```

```
        y = y - x;}
printf("%d", x);
exit(0);
```

Código 1 – Ciclo infinito sempre que x = 0 e y > 0 (adaptado de: Gazzola et al. (2019))

Neste exemplo, a metodologia da localização do defeito iria corrigir a primeira condição deste algoritmo através da modificação da condição, adicionando uma condição quando *x = 0.* Contudo, recorrendo à localização da correção, torna-se preferível proceder à correção da condição no interior do mesmo bloco. Desta forma, adiciona-se outra condição que imprima *y*, sempre que *x = 0* (Gazzola et al., 2019).

Como referido, estas técnicas ajudam a identificar as localizações elegíveis para sintetizar a correção, independentemente da localização do defeito. A ideia principal consiste na exploração do defeito a partir de um local já reconhecido, de modo a compensar o seu efeito. Existem duas técnicas principais que derivam desta: *Model-Based Fix Locus Localization* e *Angelic Fix Localization*. A primeira procura identificar o uso inapropriado de objetos recorrendo a modelos minerados pré-definidos. Por sua vez, *Angelic Fix Localization* pretende localizar as declarações relevantes em relações a condições *if* que se encontrem em erro (Agarwal & Agrawal, 2014; Mechtaev et al., 2016).

O objetivo da técnica *Model-Based Fix Locus Localization* passa pela recolha de dados do ambiente a partir da execução de cenários de teste. Partindo da correta execução do programa, definem-se modelos minerados. Sempre que uma execução com falha viole um modelo minerado indica que o programa usa incorretamente objetos. A mineração dos modelos é concretizada através de máquinas de estados finitos que representam o ciclo de vida do objeto, ou seja, através da forma como eles são criados a partir da invocação dos seus métodos, e também, quando os seus campos podem ser lidos e escritos. Isto permite que eles possam ser verificados dinamicamente enquanto os testes em falha são executados. A ideia da mineração de modelos parte das execuções corretas até às execuções falhadas, identificando os elementos do código responsáveis pela falha (Agarwal & Agrawal, 2014; Wong et al., 2016).

Por sua vez, a técnica *Angelic Fix Localization*, objetiva a localização de falhas através da modificação de condições *if* num programa (Mechtaev et al., 2016). Assim, torna-se importante proceder à localização das condições suspeitas de modo a identificar o erro e

forçar sistematicamente a falha através dos casos de teste, de forma a gerar várias alternativas de decisão, sem considerar o resultado da condição avaliada (Wen et al., 2018).

Tomando como exemplo o excerto de Código 1, anteriormente mencionado, torna-se possível entender que o algoritmo falha sempre que $x = 0$ e $y > 0$, fazendo com que a execução não entre no bloco if e provoque a falha. Esta técnica permite verificar o que aconteceria no momento que esse mesmo bloco é executado, mesmo tendo a condição um resultado avaliado em falso. Para a execução passar, é necessário que a mesma entre no bloco *if*, de modo a terminar após imprimir o resultado de *y* (Gazzola et al., 2019).

Todos os pontos de decisão que podem ser utilizados na transformação de casos de teste com falha em casos de teste com sucesso, em conformidade com esta estratégia, são selecionados e devolvidos como potenciais alvos passíveis de uma solução de reparação automática, orientada às condições presentes no programa (Koyuncu et al., 2020; Wong et al., 2016).

Todavia esta abordagem mostra um comportamento ligeiramente diferente na forma em como se tenta detetar condições *if* em falta. Esta técnica permite verificar casos de teste em falha que possam ser transformados em testes aprovados, avançando a execução de uma condição simples ou composta. Neste caso, uma correção consiste em adicionar novas condições *if*, que consigam controlar a execução das declarações que devem ser ignoradas. A inclusão de condições *if* apropriadas fazem parte do processo de reparação (Abdessalem et al., 2020).

Segundo Agarwal & Agrawal, Wong et al, (2014; 2016), existe uma versão melhorada desta técnica, na medida em que se incrementou a capacidade de considerar qualquer expressão de um programa, estendendo a técnica para além da aplicação a expressões condicionais. O procedimento é efetuado a partir da identificação das localizações suspeitas no programa, e expressões SBFL são correspondidas a essas localizações para que uma execução específica possa ser executada de forma a substituir as expressões suspeitas por símbolos.

2.3 Software repair: Formulação da correção

A etapa seguinte foca-se na reparação propriamente dita do *software*, que consiste na criação da correção para um programa. A formulação da correção parte da existência de algoritmos que aproximam o problema do programa a reparar com a solução, neste caso, através da

geração de um conjunto de soluções candidatas. Todavia, classifica-se uma solução gerada como uma solução plausível, uma vez que, em termos práticos não é garantida a sua solução. Dependendo do modo como o problema é definido e endereçado, existem duas abordagens distintas, que seguidamente são abordadas: *Generate-and-validate* e *Semantics-driven*.

Contudo é importante referir que independentemente das abordagens utilizadas que geram uma solução, os programadores são notificados para que possam verificar a sua qualidade e eficiência de modo a, sempre que necessário, efetuem uma reparação mais correta e satisfatória.

2.3.1 Generate and Validate

Relativamente ao processo *Generate-and-Validate*, pode afirmar-se que este consiste em duas atividades principais, a geração (*generate*), que produz as soluções candidatas para o problema e, a validação (*validate*), cuja responsabilidade passa pela verificação eficiente das soluções que são geradas (Koyuncu et al., 2020).

A atividade de geração utiliza um conjunto de operadores de mudança, produzindo novos programas através da modificação, e que são adicionados às soluções candidatas. Essas modificações são implementadas nos locais suspeitos, tendo por base a identificação, através de técnicas de localização, e o nível de suspeita. As técnicas de reparação de programa utilizam 4 tipos de operadores de mudança: substituição e negação de condição, chamadas a métodos de inserir ou eliminar, eliminar a funcionalidade e, por último as manipulações AST. O conjunto de operadores de mudança subdividem-se em 3 tipos (Y. Qi et al., 2013):

- *Atomic change*: modificam o programa original num único ponto;
- *Pre-defined template*: alteram o programa tomando em consideração a potencialidade dos padrões complexos pré definidos;
- *Example-based template*: atuam da mesma forma do ponto anterior, no entanto as templates são extraídas de um histórico, tal como num sistema de versionamento.

Algumas técnicas também aplicam operadores de mudança às soluções candidatas (Wilkerson & Tauritz, 2011). Desta forma, os programas modificados podem incluir alterações produzidas por múltiplos operadores de mudança. Esse conjunto de múltiplos operadores de mudança resulta também em um conjunto de soluções candidatas, sendo que nenhuma solução é

produzida para o problema sempre que o tempo definido para o processo de reparação expira ou quando cada elemento do conjunto de soluções candidatas é considerado (Hua et al., 2018).

A validação procura estabelecer a veracidade das soluções candidatas através da execução de casos de teste que estejam disponíveis. Neste processo, sempre que um programa reparado passe nos testes disponíveis, o mesmo é devolvido ao programador com uma possível correção para o problema. O resultado esperado deste processo prevê que todos ou apenas alguns elementos, provenientes do conjunto de possíveis soluções, sejam descartados, como é o caso das soluções candidatas cujos testes, na sua maioria, falham, revelando-se insatisfatórios (Alarcon et al., 2020; Hua et al., 2018).

No âmbito da geração e validação, existem duas principais estratégias que podem ser aplicadas a cada um dos três conjuntos de operadores de mudança descritos e diferem apenas na forma como são manipuladas (Z. Qi et al., 2015):

- *Search-based*: aplica os operadores de mudança aleatoriamente ou guiados através de um algoritmo de heurística ou meta-heurística;
- *Brute-force*: produzem todas as mudanças que podem ser obtidas, ainda que dentro de alguns limites, tendo em consideração o algoritmo de localização e os operadores de mudança.

Face ao exposto, considera-se pertinente conduzir uma análise mais aprofundada aos três conjuntos de operadores de mudança, tendo por base cada uma das estratégias definidas.

Um operador de mudança atómico ou a*tomic change operator* altera um programa num único local da sua *Abstract Syntax Tree* (AST), como é o caso de um operador único de uma expressão ou até mesmo a inserção, modificação ou remoção de uma declaração (Gazzola et al., 2019).

A análise da literatura encontrada permite compreender a existência de diferentes técnicas associadas aos operadores de mudança atómica, que se podem dividir pelas estratégias abordadas. Assim, apresenta-se na Tabela 1 um resumo das técnicas encontradas, tendo por base a estratégia e o modelo de mudança utilizados (Gazzola et al., 2019; Mousavi et al., 2020).

Tabela 1 – Generate and validate: Atomic change operators (Gazzola et al., 2019)

Estratégia	**Fault Model**	**Change model**	**Técnicas**
Search-based	Geral	AST inserir, apagar, reutilizar	GenProg, Marriagent, SCRepair, RSRrepair
	Geral	AST modificações	JAFF
	Geral	Substituição de operadores e/ou nomes de variáveis	pyEDB, MUT-APR, CASC
Brute-force	Geral	Substituição de operador, negação de condição	Debroy and Wong
	Geral	Inserir, apagar chamadas a métodos	PACHIKA
	Geral	Apagar funcionalidade	KALI
	Geral	AST inserir, apagar, reutilizar	AE

As técnicas que se baseiam na estratégia *search based* utilizam algoritmos e heurísticas ligeiramente diferentes de forma a maximizar a eficácia de um conjunto pertencente a este operador de mudança, definidos ao nível AST do programa (Z. Qi et al., 2015).

Relativamente às técnicas GenProg, Marriagent, RSRepair e SCRepair utilizam três operadores atómicos para inserção, modificação e exclusão de declaração no AST. Este conjunto de operadores permite a modificação em livre-arbítrio do programa, contando que o mesmo inclua uma variedade suficiente de declarações passíveis de serem copiadas. Por sua vez, a técnica JAFF permite a aplicação destes mesmos operadores, permitindo a manipulação contínua através de ramos filho (Kou et al., 2016; Mousavi et al., 2020).

Por outro lado, pyEDB, MUT-APR e CASC são técnicas que recorrem a uma abordagem diferente para adotar uma diferente alteração no modelo. Elas utilizam um pequeno conjunto de operadores de mudança focados, ou seja, conseguindo modificar operadores e variáveis utilizadas no programa (Gazzola et al., 2019; Mousavi et al., 2020).

Ainda relativamente ao operador de mudança atómica, surgem as técnicas associadas à estratégia de reparação *brute-force*, que procuram uma correção para um programa em reparação através da exploração sistemática do espaço. Podem ser utilizadas diferentes técnicas, que usando diferentes conjuntos de operadores de mudança atómicos, endereçam diferentes tipos de falhas (X.-B. D. Le, 2016).

Considerado um outro conjunto de operadores de mudança, as técnicas de reparação baseadas em *pre-defined templates* procuram modificar programas, em conformidade com um conjunto de operadores de mudança que podem afetar uma ou mais declarações do programa. O uso das mesmas, permite que os programadores possam definir padrões complexos que, de forma coerente, afetem em múltiplas localizações o programa sobre reparação. De certa forma, isto seria difícil de obter utilizando combinações de mudança atómicas aleatórias. Alguns exemplos da sua utilização passam por expandir a sincronização entre blocos, de modo que as condições do programa sejam manipuladas, e também, adicionadas implementações de pré-condições de controlo de acesso ao código (X. B. D. Le et al., 2018).

O uso da estratégia *brute-force* revela-se presente na maioria deste tipo de técnicas. Contudo, a aplicação de uma *template*, apesar de útil, pode nem sempre ser suficiente para corrigir o problema, o que torna as estratégias *search-based* viáveis. Mais concretamente, a técnica de reparação APR, que observa *concurrency faults,* faz uso de *templates* manuais no contexto do algoritmo *search-based*, de modo que os problemas não-triviais de uma classe possam ser corrigidos (Kelk et al., 2013).

Por outro lado, as técnicas de *brute-force* definem *templates* que podem corrigir falhas de um programa quando aplicadas somente uma vez (Hua et al., 2018). Este processo decorre através da aplicação de cada *template* em cada local que seja possível, executando o processo até que o tempo seja excedido ou consiga corrigir o programa. Dentro destas técnicas gerais, destacam-se: AutoFix-E, SPR e Prophet, explorando as *pre-defined templates* que não se encontrem relacionadas com modelos de falha específicos (Gazzola et al., 2019).

As técnicas de reparação *example-based*, de acordo com um conjunto de operadores de mudança, procuram extrair um conjunto de amostras de correções anteriormente utilizadas para corrigir programas. Para tal, modificam os programas através do uso de técnicas *search-based*, atingindo a evolução, ou através de mudanças sistemáticas como é o caso da *brute-force* (Gazzola et al., 2019; le Goues et al., 2019; Monperrus, 2018).

Estas técnicas podem ser extraídas quer manualmente, quer automaticamente, embora nos casos de extração manual, as *templates* são definidas uma única vez para tudo, criando um histórico e, depois utilizadas para reparar os programas. Nestas condições, são utilizados

conjuntos pré-definidos de operadores e *example-based templates*, atuando como operadores de mudança (Gazzola et al., 2019; Mousavi et al., 2020).

Em contrapartida, também podem ser utilizadas técnicas de mineração ou outros algoritmos para extrair automaticamente as *templates*, fazendo com que o processo de extração possa ser repetido ao longo do tempo, para diferentes programas, apurando, desta forma a técnica em geral (Gazzola et al., 2019; Mousavi et al., 2020).

As técnicas *search-based* gerais usam uma mudança de modelo específica derivada da análise com base em muitas correções do mundo real, levando a que algoritmos baseados em pesquisa possam ser recombinados para maximizar a sua eficácia na reparação de falhas (Gazzola et al., 2019; Monperrus, 2018).

Por outro lado, a reparação de programas explora *history-driven*. Esta abordagem apresenta-se semelhante a GenPog, embora utilize 12 operadores de mutação derivados. No decorrer da extração de informação dos projetos são sintetizados padrões de correção de *bugs* que representam conjuntos AST já utilizados para corrigir os mesmos, orientando, através desse padrão, o processo de seleção (Goues et al., 2013). Somente as soluções candidatas que registam evolução em cada iteração são capazes de incorporar mudanças similares àqueles representados nos padrões de correção de *bugs* (Gazzola et al., 2019; Mousavi et al., 2020).

Técnicas gerais de *brute-force* são projetadas para reparar quaisquer falhas de classe, partindo-se do princípio que já exista um conjunto de correções do mesmo tipo no mundo real (Gazzola et al., 2019).

A Tabela 2 pretende resumir as técnicas utilizadas nos conjuntos de operadores de mudança *Pre-defined template* e *Example based template*.

Tabela 2 – Generate and validate: Template-based change operators (Mousavi et al., 2020)

Template-type	**Estratégia**	**Fault Model**	**Change Model**	**Técnicas**
Pre-defined	Search-based	Concurrency-faults	Manipulação sincronizada da região	ARC
	Brute-force	Geral	Tranformação de templates de Código	AutoFix-E, AutoFix-E2
			Alteração da condição e valor da variável	SPR, Prophet
		Buffer-overflow	Substituição de funções, manipulação	PASAN, AutoPag

			de buffer	
Example-based	Search-based	Geral	Transformação de templates de Código, reutilização de declarações na aplicação	History-driven repair
			Transformação de templates de Código	PAR, Relifix
	Brute-force	Geral	Transformação de templates de Código	R2Fix
		Buffer-overflow	Inserção de fragmentos de Código no programa sob reparação	CodePhage

2.3.2 Semantics-Driven Repair

Explicitamente, os tipos de técnicas de reparação semântica codificam formalmente o problema do programa a reparar. O problema pode ser explicito, como é o caso das fórmulas que permitem corresponder possíveis correções do programa em reparação, ou implícito, transformado numa correção, através de um procedimento analítico. Desta forma, é possível garantir uma solução reparada, caso esta seja encontrada, para o problema, não necessitando de ser validada contra o programa em reparação (Asad et al., 2019; X. B. D. Le et al., 2018).

Na realidade, o programa em reparação pode ser visto como uma representação aproximada e real do problema a ser solucionado, embora existindo a possibilidade de a solução ser problemática e resolvida, podem surgir outros problemas como são exemplo o desempenho e problemas funcionais. Por consequência, as soluções geradas automaticamente carecem de validação manual ou automática para que seja possível serem aceites (Mousavi et al., 2020).

O processo geral da técnica de reparação semântica subdivide-se em 3 principais atividades (Aleti & Martinez, 2020; X. B. D. Le et al., 2018; X.-B. D. Le et al., 2016b):

- Análise comportamental: através da extração de informação semântica a partir do programa em reparação, para que seja possível proceder à extração de comportamentos corretos e incorretos do programa;
- Geração do problema: explorando a informação recolhida pela atividade de análise comportamental de forma que seja gerada uma representação formal da reparação

do problema, onde as soluções representam correções atuais através de alterações do código;

- Geração da correção: procura identificar a alteração ao código capaz de corrigir o programa ou, no caso de não haver solução ou a mesma não poder ser encontrada num tempo útil.

Ainda dentro do campo da reparação semântica, uma das principais chaves desta técnica é a síntese do programa, que tem por objetivo a construção de uma correção para o programa em reparação e que inclui as técnicas gerais e de condições erradas ou pré-condições em falta (Mousavi et al., 2020). Nas técnicas gerais destacam-se:

- SemFix, procura substituir a expressão no programa em reparação, por outra simbólica capaz de representar uma condição genérica ou até mesmo uma substituição de um valor por uma variável;
- DirectFix, revela-se muito semelhante à técnica anterior, embora seja mais orientada a programas com uma abordagem monolítica;
- Angelix, é uma técnica que contrasta com a anterior, destacando-se por preservar a escalabilidade através da sintetização de correções em várias linhas;
- SearchRepair, utiliza uma base de dados escrita por pessoas, de *patches* codificados.

Relativamente às condições erradas e pré-condições em falta, é de salientar a identificação das condições mais suspeitas através de algoritmos de localização, identificando as variáveis incluídas nelas, sintetizando o problema de modo a gerar uma condição reparada. Nopol é uma das técnicas principais que procura sintetizar correções partindo de uma condição ou de um ciclo. Além disso, identifica as declarações suspeitas através que incluem condições e, *angelic fix* de modo a confirmar a potencial existência de uma correção (Kou et al., 2016).

As falhas relacionadas com a concorrência, como é o caso de violação de atomicidade, *deadlock*, *livelock* e violação de políticas de segurança podem ser detetadas através da análise de regiões do código menos consistentes e, normalmente responsáveis pelas falhas de concorrência, podendo ser reforçadas através de condições apropriadas e sincronização entre mecanismos que podem prevenir problemas de concorrência (Lin & Kulkarni, 2014; Mousavi et al., 2020). São exemplos AFix, CFix, OFix, HFix, Axis, Grail e DFixer.

Também em contexto de falhas de geração HTML pode ser destacada a técnica PHPQuickFix que permite detetar e reparar declarações com HTML incorreto. Outra técnica é o PHPRepair

que consegue endereçar as falhas, como é o caso do fecho de nós HTML em falta (Samirni et al., 2012).

Além destas técnicas, também existem os recomendadores de correção que podem sugerir alterações ao programador de modo a obter a correção final. No entanto, este tipo de técnicas não pressupõem a geração de uma nova versão do *software* correta, mas sim a apresentação de um resultado que rapidamente possa ser transformado numa correção.

A seguinte tabela agrupa as técnicas de reparação semântica (Mousavi et al., 2020):

Tabela 3 – Técnicas *semantic-driven* (Gazzola et al., 2019)

Fault-Model	**Change-Model**	**Técnicas**
Geral	Sintetizar novas expressões	SemFix, DirectFix, Angelix, SearchRepair
Condições erradas e pré-condições em falta	Alteração de condição e inserção de confição *if*	Nopol, Infinitel, DynaMoth
Concurrency-faults	Manipulação de região crítica, paralelização para mover palavra-chave	AFix, CFix, HFix, Axis, Grail, DFixer
Geração de falhas HTML	Modificação de *strings*	PHPQuickFix, PHPRepair
String sanitization	Inserir verificações, modificar *strings*	SemRep
Access Control Violations	Inserção de verificações de funções	FixMeUp
Memory Leaks	Inserção de declarações *free()*	LeakFix

Também os recomendadores de correção se revelam úteis como ferramentas de apoio à produção de *software.* Desta forma, a seguinte tabela representa algumas técnicas utilizadas dependendo do modelo de falha ocorrido (Chang et al., 2013):

Tabela 4 – Técnicas de recomendação de correções (Gazzola et al., 2019)

Fault-Model	**Técnicas**
Geral	BugFix, MintHint, QACrashFix
Falhas de segurança	BovInspector, CDRep
Concurrency-faults	ConcBugAsssit
Falhas de desempenho	CARAMEL

Em suma, um dos maiores desafios da reparação automática passa pela identificação de situações, tentando responder à questão: Onde e de que forma a reparação automática pode ser aplicada? Conforme o contexto e o objetivo da reparação, o programador tem ao seu dispor várias ferramentas e técnicas que conduzem à otimização do processo de ARP, conforme exposto neste capítulo.

2.4 Abordagens existentes e avaliação

Após compreender as diferentes etapas da APR, importa analisar e comparar empiricamente as técnicas de formulação da correção que apresentam maior potencial de utilização no contexto deste trabalho. Dada a natureza automatizada das *pipelines*, onde o trabalho se realiza, considera-se o processo *Generate and validate* o mais adequado à reparação automática. Neste sentido, apresenta-se de seguida uma descrição mais detalhada das diferentes técnicas deste processo, e respetiva comparação empírica.

Tal como anteriormente referido, existem diferentes processos que abrangem diferentes técnicas capazes de detetar os erros (Koyuncu et al., 2020; X.-B. D. Le et al., 2018). No que respeita ao processo *Generate and validate*, o qual consiste em duas principais atividades, produzindo em primeiro lugar a solução candidata para o problema, e, por último, validar que a solução é eficiente (Gazzola et al., 2019).

No que concerne aos operadores de mudança atómicos, através da análise da localização do defeito é possível implementar o ou os operadores de mudança adequados (Wong et al., 2016). Efetivamente, estes operadores podem ser utilizados por várias técnicas de modo a combiná-los eficientemente para obtenção de múltiplas variantes do mesmo programa a ser reparado e, por consequência, encontrar a melhor correção possível (Mehne et al., 2018). No entanto, estes operadores requerem uma análise mais dispendiosa relativamente ao programa em reparação. Na verdade, em algumas situações podem até mesmo provocar um abrandamento no processo de procura, ainda que, em alguns casos, acabem por se revelar mais eficazes (Z. Qi et al., 2015). Em contrapartida, as técnicas baseadas no *Pre-Defined Templates* podem ser menos custosas de aplicar em relação às mudanças atómicas, o que pode reduzir a velocidade da evolução de algoritmos *search-based* (Mehne et al., 2018).

De seguida, apresenta-se a análise e comparação de diferentes técnicas baseadas na estratégia *search-based* e, posteriormente, na estratégia *brute-force*.

2.4.1 Técnicas baseadas em Search-based

Partindo para a análise das técnicas baseadas no *search-based* dos operadores de mudança atómicos, relativamente à técnica GenProg, pode afirmar-se que utiliza *genetic programming* para orientar e validar as atividades (Kou et al., 2016; Tan & Roychoudhury, 2015). Em cada iteração em que esta técnica é utilizada, a localização onde a mudança atómica é aplicada é aleatória, de acordo com uma probabilidade de distribuição que corresponda às declarações suspeitas calculadas através de algoritmos de localização, levando a que exista uma maior probabilidade de afetar o defeito quando estas declarações se relacionam com o mesmo (le Goues et al., 2019).

Num nível prático, existe uma função cuja responsabilidade é medir o programa reparado através do resultado de um conjunto de testes (Y. Liu et al., 2018). Através desse resultado torna-se possível atribuir um peso, que pode ser positivo quando os testes são aprovados, ou negativo quando o inverso acontece. Quanto mais elevado for o grau de aptidão, calculado a partir dos pesos mencionados, maior é a probabilidade de a solução candidata ser preservada enquanto as outras são descartadas. GenProg também foi estudada face a otimizações, como é o caso da exploração das técnicas de recompilação e reinstalação de modo que soluções candidatas possam ser geradas de forma incremental, evitando a recompilação completa do programa (Gazzola et al., 2019).

Um exemplo das otimizações realizadas à técnica anterior é a Marriagent. Esta técnica é semelhante a GenProg, no entanto, a diferença passa pela seleção com base na medição da diversidade entre um par de programas, tendo em conta o conjunto total de alterações aplicadas a cada programa relativamente ao original. A diversidade dos programas do par aumenta a probabilidade da seleção de um par (Mehne et al., 2018; Mousavi et al., 2020).

Por outro lado, RSRepair procura determinar o operador de mudança que deve ser aplicado, de forma aleatória, ao programa a ser reparado enquanto a localização onde a mudança vai ser aplicada é determinada pela probabilidade que depende das declarações suspeitas. Esta técnica valida e descarta, caso a solução candidata não passe todos os casos de teste, ou seja, cada solução candidata inclui apenas uma mudança em relação ao programa. Desta forma, atribui-se uma estratégia de prioridade com base nos casos de teste definidos e executados anteriormente, depreendendo-se que os testes eficazes a revelar as alterações que não corrigem o programa deveriam ser executados mais cedo (X.-B. D. Le et al., 2016a). SCRepair é

uma extensão de RSRepair capaz de que introduz uma métrica que calcula a semelhança entre dois fragmentos de código do seu AST.

Relativamente à técnica JAFF, a sua grande vantagem é a capacidade na geração de declarações aleatórias, o que a torna bastante útil para um grande número de falhas, apesar de conter uma probabilidade pequena para produzir as novas declarações aleatórias necessárias (Gazzola et al., 2019).

Outra técnica que recorre ao *genetic programming* é a pyEDB. Contudo, invés de funcionar tal como GenProg, esta técnica representa uma evolução de soluções candidatas, identificada através das mutações aplicadas às soluções candidatas do programa em reparação, revelando-se bastante compacta e eficiente (Gazzola et al., 2019). A sua base consiste na geração de duas tabelas de modificação, uma tabela incluí todas as mudanças que podem ser efetuadas aos operadores relacionais do programa, e a segunda, todas as mudanças possíveis aos nomes das variáveis do programa. A ideia do foco no delta foi explorada de modo a reduzir o tempo e o custo de compilação.

Uma outra técnica reconhecida neste âmbito é a MUT-APR. Esta técnica de reparação utiliza SBFL para apoiar a localização e identificar declarações com defeito, e utiliza *genetic programming*, sendo, portanto, muito semelhante a GenProg. Através delas, procura-se explorar no espaço de pesquisa as possíveis variantes do programa. As falhas são reparadas através do seu endereçamento, podendo ser reparadas através da modificação de operadores de mudança que são limitados à substituição relacional, aritmética, deslocamento ou por bit (Zhang et al., 2017).

Por último, a técnica CASC usa, também, a programação genética e operadores de mudança atómica, tendo implicação direta relativamente à programação de nomes e variáveis. Além disso, elabora o conceito de coevolução, isto é, envolvendo também os testes e não apenas as soluções candidatas. Desta forma, quanto maior for o número de casos de teste, maior também é o número de soluções candidatas descartadas. O desafio desta técnica passa pela evolução do conjunto de testes sem gerar casos incorretos que possam orientar o programa para a geração de derivações de programas com defeito (X.-B. D. Le et al., 2016a).

Entre vários artigos analisados é de destacar que a maioria dos autores recorrem à ferramenta GenProg, que por sua vez utiliza *genetic programming* como sua base. Esta última revela-se

uma técnica de procura com base em heurística que visa escolher a solução ótima entre um conjunto de soluções candidatas de um programa (Martinez & Monperrus, 2019a).

Uma alternativa apresentada por Tan & Roychoudhury (2015), baseada no tipo de *template Example-based*, é Relifix. Esta técnica tem a finalidade de reparar *software* utilizando uma abordagem orientada à reparação de regressões. Uma regressão surge à medida que o *software* evolui, levando a falhas de testes que anteriormente passavam.

Moussavi et al (2020) no âmbito da sua revisão da literatura procura colmatar o que falta para que seja atingida a automatização completa no processo de reparação automática. É realizada uma comparação entre GenProg e Relifix, orientada à reparação automática de erros de regressão. Este estudo conclui que Relifix consegue corrigir erros de regressão sem introduzir novos com maior eficácia em relação a GenProg. Ainda segundo Moussavi et al (2020) outro desafio para a técnica GenProg é a escalabilidade, relacionando o tempo e a quantidade de código a ser manuseada.

No mesmo sentido, em um estudo comparativo de Tan & Roychoudhury (2015) é sugerida uma abordagem de auto-reparação, podendo ser uma possível solução para sistemas auto-adaptativos através de regressão na reparação. Adicionalmente, foram comparadas as técnicas Relifix e GenProg, utilizando um conjunto conceptual de operadores, incluindo 73 regressões de *software* no mundo real, revelando que a evidência do estudo indica Relifix como sendo uma técnica mais forte neste âmbito, comparativamente a GenProg (Tan & Roychoudhury, 2015).

Desta forma, torna-se evidente que quanto maior se torna o programa, mais difícil é o processo de *debugging* para depuração de erros. Adicionalmente, Kou et al.(2016) indica que o mesmo sucede com as ferramentas de reparação automática, tornando o processo mais demorado e com maior probabilidade de exceder o tempo limite. Porém, existem outras ferramentas baseando-se na procura aleatória através de padrões ou análise semântica.

Por forma a dar resposta a esta lacuna, surgiram estudos de APR que se focam no *crossover* de técnicas existentes conduzidas por algoritmos aleatórios (Kou et al., 2016). Um desses estudos foi também desenvolvido por Kou et al. (2016) onde apresenta uma proposta de técnica, passando pela seleção e mutação de testes com base em propriedades, de forma a obedecer a três passos que consistem em (Kou et al., 2016):

- Comparar programas modificados: comparar programas provenientes de soluções possíveis;
- Selecionar pares de programas modificados: selecionar pares de possíveis soluções por ordem de valor mais alto;
- *Crossover*: Gerar novos programas modificados pelos pares selecionados.

2.4.2 Técnicas baseadas em Brute-force

Noutra perspetiva, surgem as técnicas associadas à estratégia de *brute force* dos operadores de mudança atómicos. Debroy and Wong é uma técnica que investiga a aplicação de operadores utilizados em testes de mutação para a reparação automática de um programa. A técnica de reparação atua através da alteração sistemática das declarações no programa em reparação, até que a lista de suspeitas seja percorrida ou o tempo se esgote. Incluindo apenas os programas que podem ser reparados no espaço de procura, através da uma alteração única (Gazzola et al., 2019).

Outra técnica de reparação é a PACHIKA que procura explorar técnicas de especificação de mineração, tendo em vista a reparação de programas de *software* orientados a objetos. Através da observação e comparação de casos de teste para inferir pré-condições de forma a atingir uma execução bem-sucedida e as mesmas serem cumpridas. Alguns casos de teste com falha devem ser definidos para que existam violações de pré-condições e se despolete a reparação através desta técnica (le Goues et al., 2019).

Ainda neste âmbito, Kali é uma técnica de reparação que se baseia na localização das 500 declarações mais suspeitas de um programa e, sistematicamente as altera utilizando um conjunto de operadores de mudança atómicos desenhados para remover funcionalidades. De facto, muitos programas reparados que passam nos casos de teste, podem ser obtidos através da remoção de funcionalidades. Todavia esta solução não seria considerada por um programador que procura proceder à reparação sem perder funcionalidades (Martinez & Monperrus, 2019a).

Finalmente, a técnica de AE revela-se semelhante à de RSRepair, no entanto, o grande desafio passa pela redução do tamanho de pesquisa através da verificação de equivalência semântica. Esta última, consiste na capacidade de deteção de conjuntos de soluções candidatas semanticamente equivalentes e sintaticamente diferentes, permitido descartar várias

soluções candidatas sem recorrer à execução dos casos de teste. Esta técnica permite a deteção de duas soluções semanticamente equivalentes caso elas difiram para qualquer um dos seguintes elementos (X.-B. D. Le et al., 2016b):

- Igualdade sintática: diferindo, por exemplo, na presença de nomes de afirmações ou variáveis duplicadas;
- Código não utilizado: diferindo na presença de código não usado;
- Agendamento de instruções: reordenação sem afetar a semântica do programa.

Em jeito de conclusão, compreende-se que a constante evolução das técnicas de APR se traduz em melhores resultados na sua implementação. Não obstante, a escolha da melhor técnica irá sempre depender do programa onde a implementação será realizada.

2.5 Conclusões

A APR continua a ser uma possibilidade que pode melhorar a qualidade do programa e também facilitar a experiência de desenvolvimento dos programadores. No âmbito técnico, apresenta desafios na definição, priorização e orientação no espaço, dos *patches*, evidenciando os benefícios da inclusão dos testes de *software* e da qualidade do produto final, na qual a reparação tenta basear-se.

A pesquisa na área procura colmatar a lacuna entre o esforço manual gasto na escrita de programas corretos e a APR que segue abordagens de aprendizagem. De facto, dado os constantes desafios na geração de correções múltiplas, pode perceber-se a dificuldade em gerar programas corretos sem perder funcionalidades.

Ao longo do tempo, vários estudos indicam métricas relativamente ao número de *patches,* bem como o custo e tempo necessários para avaliar o desempenho na reparação de sistemas através da APR. No entanto, as decisões de design e configuração raramente são divulgadas na sua totalidade, não incluindo o impacto das mesmas no desempenho, podendo levar a resultados errados.

Também se torna possível entender a necessidade de gerir mudanças em projetos de *software*, na medida em que se torna necessário automatizar a reparação, uma vez que entre depurar e fazer a manutenção de um projeto são necessários bastantes recursos e, por vezes, os projetos necessitam de ser reconstruídos.

Em suma, as ferramentas de reparação podem ser incluídas nos projetos de *software* de forma a facilitar a resolução do dilema dos programadores, sempre que é necessária alguma alteração. Assim, os estudos na área têm vindo a trazer evoluções no que respeita à qualidade e ao desempenho na reparação de programas.

3 Análise de Valor

Este capítulo descreve a análise de valor relacionada com o contexto deste trabalho. A análise de valor é um processo sistemático, formal e organizado de análise e avaliação, cujo propósito passa por melhorar o valor de um produto, passando pela análise de possíveis soluções para um problema específico.

3.1 Visão geral

A criação de valor atende a dois fatores importantes que são a capacidade e o desempenho, sem descurar o custo. Este último não influencia o valor diminuindo-o, porém, o objetivo passa por aumentar o custo para que o desempenho e capacidade sejam sempre mensuravelmente maiores.

O processo da análise de valor inicia-se a partir da orientação que visa perceber-se o uso e valor do produto para o destinatário. Este processo desdobra-se na análise e identificação funcionais, tornando possível verificar os benefícios e sacrifícios necessários fazer de modo que sejam consolidadas alternativas para análise e avaliação.

Neste contexto existem métodos, técnicas e ferramentas capazes de auxiliar na orientação da construção da análise de valor. Assim é relevante utilizar o *New Concept Development Model* (NCD *model*) de Koen (2004) uma vez que o mesmo fornece orientação e uma definição e linguagem comum acerca dos diferentes passos do processo de inovação. Este modelo desdobra-se em três partes fundamentais (Koen, 2004):

1. O motor, que representa a organização através da liderança, cultura e estratégia de negócio, originando cinco elementos que possíveis de controlar;
2. Os cinco elementos: identificação da oportunidade, análise da oportunidade, criação e enriquecimento da ideia, seleção da ideia e definição de conceito;
3. Fatores externos que influenciam a organização, como são o caso dos canais de distribuição, leis, políticas governamentais, clientes e concorrentes, bem como os fatores ambientais.

O modelo NCD apresenta-se sob formato circular, sugerindo que as ideias possam fluir, circular e iterar entre os elementos. Assim, as ideias não precisam de seguir nenhuma ordenação ou combinação, podendo, várias vezes, serem utilizados um ou mais elementos, constituindo uma relação, e não um processo linear. A Figura 3 representa o modelo NCD:

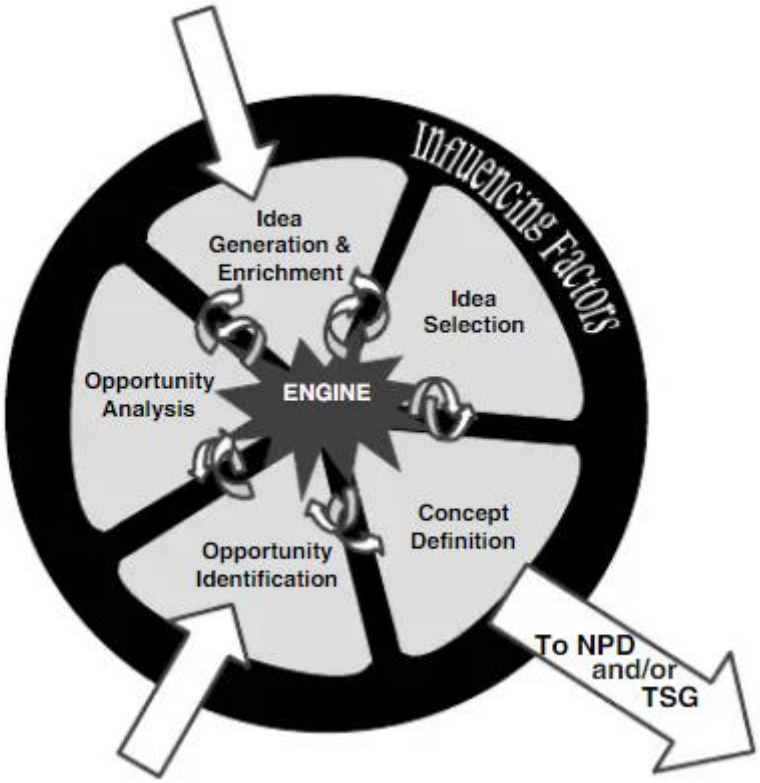

Figura 3 – New Concept Development Model (Koen, 2004)

Finalmente, o modelo sugere como início a identificação de oportunidade e a criação e enriquecimento da ideia, terminando após a definição do conceito dando origem a um ou ambos os processos: *new product development* (NPD) ou a *technology stage gate* (TSG) (Koen, 2004).

3.2 Identificação da oportunidade

Neste subcapítulo procura-se identificar oportunidades que possam ser abordadas. Este elemento, normalmente é conduzido por objetivos do negócio, passando esses objetivos pela identificação de uma vantagem competitiva ou até mesmo na simplificação de operações. Estes últimos, levando à redução de custos e à sua aceleração (Koen, 2004).

No contexto deste trabalho é possível identificar a oportunidade através da necessidade de reforço da evidência científica relativamente à APR e consequente experimentação.

Através da identificação de processos, técnicas e ferramentas relacionados com a reparação automática, bem como a partir da comparação de resultados entre as técnicas e ferramentas existentes descritas, torna-se possível a identificação da oportunidade na experimentação e comparação de técnicas em contexto empresarial, para que seja possível reduzir o custo na depuração e reparação de programas, e ao mesmo tempo, objetivar a intervenção humana na correção.

3.3 Análise da oportunidade

Após a identificação da oportunidade torna-se oportuno complementar esta informação através da especificação de negócio e das oportunidades tecnológicas. A análise da oportunidade é um processo formal e pode ocorrer de forma iterativa. O esforço despendido na análise da oportunidade depende do valor das informações que permitem reduzir a incerteza acerca da oportunidade, tendo em vista o esforço de desenvolvimento, enquadrado com a estratégia e cultura, bem como a tolerância ao risco (Koen, 2004).

A APR é uma área em constante expansão. Existe, cada vez mais uma tendência na comparação de processos e técnicas de reparação automática, contudo, ainda são escassos os estudos destas técnicas em ambientes empresariais. Os estudos que comparam ferramentas de reparação de programas têm vindo a trazer melhores resultados, tornando-se importante reforçar e dar continuidade à evidência científica.

3.4 Criação e enriquecimento da ideia

A criação da ideia diz respeito ao esboço inicial da mesma, contudo, é importante salientar que o processo de construção de uma ideia é evolutivo, resultando no seu desenvolvimento e na maturação. Este processo pode implicar que as ideias sejam construídas, eliminadas ou modificadas, podendo existir algumas iterações e mudanças à medida que sejam estudadas, examinadas, discutidas e desenvolvidas em conjunto com outros elementos do modelo NCD (Borza, 2011).

No contexto deste trabalho é utilizada a técnica *Function Analysis and System Technique* (FAST) que procura responder ao tipo de perguntas do "Como" e "Porquê", sendo representada sob o formato de gráfico. Esta técnica é capaz de fornecer uma representação gráfica e uma estrutura lógica para análise dos passos da metodologia de valor (Borza, 2011).

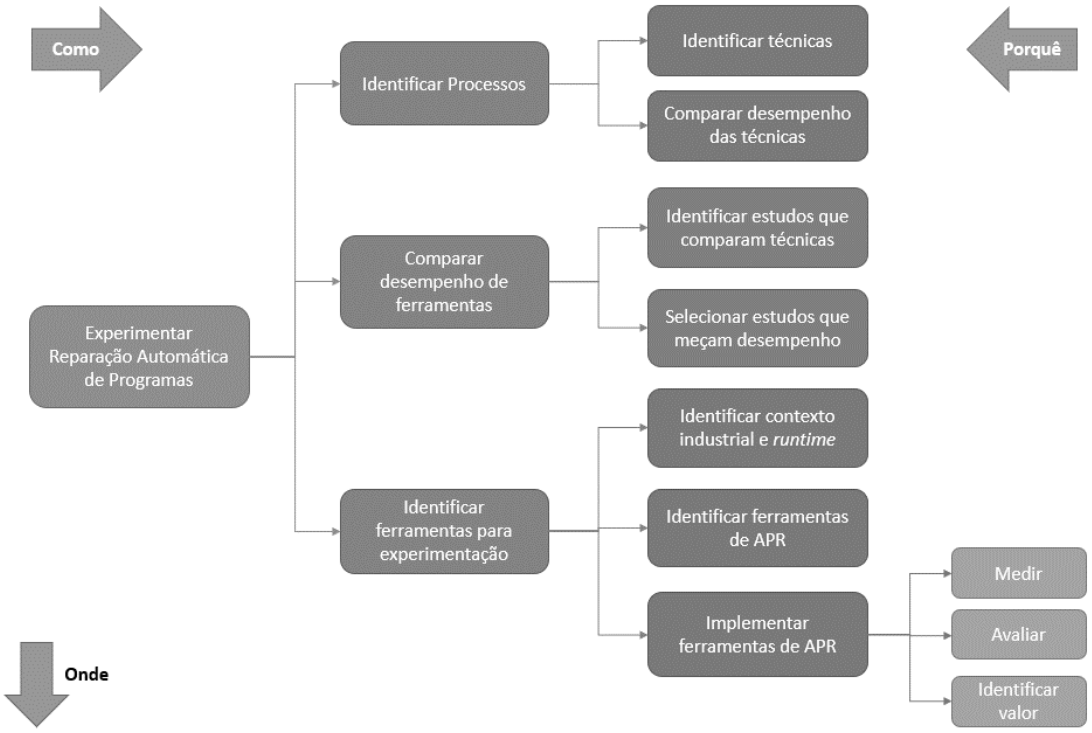

Figura 4 – Diagrama FAST aplicado ao contexto deste trabalho

A Figura 4 representa um diagrama que traduz o conjunto de ideias que visam objetivar o contexto deste trabalho. A partir da ideia inicial, neste caso a experimentação da APR em contexto industrial, torna-se possível clarificar outras ideias. Essas ideias expressam-se através das perguntas: "Como" interpretadas da esquerda para a direita, "Porquê" interpretadas no sentido oposto e "Onde" interpretadas na vertical. A primeira questão a levantar é a forma de

como fazer essa experimentação? Assim, subdivide-se a ideia principal em outras três ideias (Koen, 2004):

1. Identificar processos para a APR, nomeadamente a identificação de técnicas de reparação automática e a comparação funcional a nível de desempenho;
2. Comparar o desempenho de ferramentas através da identificação e análise de estudos que comparam técnicas e ferramentas, bem como através de estudos que meçam o desempenho das mesmas de forma comparativa;
3. Identificar ferramentas para a experimentação, identificando o contexto industrial de experimentação e respetivo *runtime*, identificar ferramentas úteis de reparação automática relacionadas com o contexto do projeto e implementar as ferramentas de reparação, por forma a medir, avaliar e identificar resultados.

3.5 Seleção da ideia

A seleção da ideia advém do processo de criação e identificação de ideias, contudo o problema inerente à seleção passa pela classificação de cada uma delas consoante o seu valor comercial. De facto, não existe um processo para garantir uma boa seleção de ideias, uma vez que o mesmo envolve uma série de iterações tendo em conta o meio-envolvente. A identificação e análise de oportunidades, bem como a criação e enriquecimento das ideias, contribuem para uma nova visão acerca dos fatores influenciadores (Aguarón et al., 2021).

Para apoio a esta secção é utilizada a técnica *Technique for Order Preference by Similiarity to Ideal Solution* (TOPSIS). Esta técnica permite avaliar o desempenho de alternativas através da similaridade da mesma, resultando numa solução ideal. Assim, a melhor alternativa é a que se encontra mais próxima da solução ideal e, do mesmo modo, mais distante da solução não-ideal (Fei et al., 2016). Seguidamente, na Tabela 5, são apresentados os critérios e definidos os respetivos pesos, face a cada uma das ideias.

Tabela 5 – Matriz de decisão com atribuição de pesos, adaptado de: (Fei et al., 2016)

Ideias \ Critérios	**Duração**	**Implementação técnica**	**Análise resultados**	**Infraestrutura**
Pesos	**0,18**	**0,37**	**0,16**	**0,29**
Seleção estudos	3	5	1	1
Seleção	3	5	1	1

técnicas				
Implementação de técnicas	2	6	1	1
Comparação de resultados	2	1	6	1

O primeiro passo no processo TOPSIS é a obtenção da matriz normalizada pesada, que é apresentada de seguida na Tabela 6, levando à obtenção das soluções ideais positiva e negativa:

Tabela 6 – Matriz normalizada pesada, adaptado de: (Fei et al., 2016)

Ideias \ Critérios	**Duração**	**Implementação técnica**	**Análise resultados**	**Infraestrutura**
Seleção estudos	0,58	0,53	0,16	0,5
Seleção técnicas	0,58	0,53	0,16	0,5
Implementação de técnicas	0,39	0,64	0,16	0,5
Comparação de resultados	0,39	0,10	0,96	0,5

O segundo passo passa é a multiplicação de cada elemento da matriz pelo peso do critério da respetiva coluna, conforme apresentado abaixo na Tabela 7:

Tabela 7 – Multiplicação de cada elemento da matriz pelo peso do critério, adaptado de: (Fei et al., 2016)

Ideias \ Critérios	**Duração**	**Implementação técnica**	**Análise resultados**	**Infraestrutura**
Seleção estudos	0,10	0,19	0,02	0,14
Seleção técnicas	0,10	0,19	0,02	0,14
Implementação de técnicas	0,07	0,23	0,02	0,14
Comparação de resultados	0,07	0,03	0,15	0,14

Após este passo, torna-se necessário proceder à separação da solução ideal positiva e negativa, calculando o quadrado de cada elemento subtraído pelo seu máximo e,

respetivamente o mínimo. Após obterem-se as tabelas finais resultantes, é necessário proceder ao cálculo da raiz quadrada da soma de cada linha das novas tabelas.

Tabela 8 – Cálculo da solução ideal positiva e negativa, adaptado de: (Fei et al., 2016)

Ideias	**Solução ideal positiva**	**Solução ideal negativa**
Seleção estudos	0,134104	0,162552
Seleção técnicas	0,134104	0,162552
Implementação de técnicas	0,132877	0,198341
Comparação de resultados	0,201458	0,128103

Estes valores auxiliam ao cálculo da aproximação da solução ideal, apresentada na Tabela 9:

Tabela 9 – Aproximação relativa da solução ideal, adaptado de: (Fei et al., 2016)

Ideias	**Aproximação da solução ideal relativa**
Seleção estudos	0,547948
Seleção técnicas	0,547948
Implementação de técnicas	0,598822
Comparação de resultados	0,388707

Através dos resultados obtidos a partir desta tabela é possível perceber-se que a ideia da implementação de técnicas tem um peso maior, seguido pela seleção de estudos e técnicas e, por fim, a comparação de resultados. A ideia implementação de técnicas é muito importante para este trabalho porque dela depende a comparação de resultados. A seleção de estudos e técnicas é importante para entender a cobertura que já existe relativamente ao tema.

3.6 Definição de conceito

A definição de conceito é o elemento final do modelo NCD. Este elemento providencia a saída para NPD ou TSG. Normalmente é necessário que exista uma compilação do cenário para a

proposta tecnológica ou de negócio. O caso de investimento consiste em informação tanto qualitativa quanto quantitativa, de forma a ser determinada e objetivada (Daou et al., 2020).

Tal como mencionado no capítulo anterior, este trabalho é direcionado para a experimentação de ferramentas de APR, com o objetivo de reforçar a evidência científica através da experimentação, mensuração e comparação de diferentes ferramentas em contexto empresarial real.

Assim, a implantação de ferramentas é a ideia mais importante, uma vez que dela depende a experimentação e respetiva obtenção de resultados. De modo a refinar a proposta de valor, seguidamente é apresentado o *business model canvas*, expondo as principais ideias, custos e benefícios, tal como apresentado na Tabela 10 (Gaikwad & Shah, 2015):

Tabela 10 – Business Model Canvas, adaptado de: (Meertens et al., 2012)

<table>
<tr>
<td rowspan="2">Parceiros Chave

-Bibliotecas digitais: IEEE, ACM, Google Scholar e outras

-A empresa Medsky, detentora do projeto onde a reparação automática vai ser experimentada.</td>
<td>Atividades Chave

-Identificação dos principais problemas e soluções através da revisão sistemática da literatura.

-Adaptar e implantar as ferramentas de reparação automática para que seja possível medir a sua qualidade.</td>
<td rowspan="2">Proposta de valor

-Identificar as ferramentas de reparação automática com mais utilização e melhores resultados de desempenho

-Experimentar as ferramentas de APR, medindo o desempenho na correção de defeitos, de forma a contribuir para a evidencia científica.</td>
<td>Relacionamento com clientes

-Implantar e experimentar a solução de reparação automática baseada nas ferramentas disponíveis para o contexto.</td>
<td rowspan="2">Segmento de clientes

-Empresas interessadas em prosseguir o estudo de ferramentas de reparação automática, e potenciar a sua utilização</td>
</tr>
<tr>
<td>Recursos Chave

- Documentação disponível acerca da reparação automática.

- APR em contexto industrial.</td>
<td>Canais

- Bibliotecas digitais, blogues, conferências.</td>
</tr>
</table>

Estrutura de custos	**Fluxo de receita**
- O conhecimento decorrente do trabalho não tem custo.	- Reduzir o tempo de *debbuging*. - Potenciar a resolução de defeitos. - Correção rápida de programa em produção. - Integração em *pipeline* de produção.

4 Metodologia de Avaliação das Ferramentas

Este capítulo descreve a metodologia seguida para o teste das ferramentas de reparação automática. A avaliação das ferramentas visa perceber a dificuldade da sua implantação em *pipeline*, bem como, a eficácia, a eficiência e a perceção dos profissionais quanto à utilização das mesmas.

4.1 Visão geral

O objetivo principal do trabalho consiste em testar ferramentas e técnicas relacionadas com a APR em contexto industrial moderno. Neste âmbito, torna-se necessário testar diferentes técnicas e ferramentas de reparação de *software*, de modo a compreender a potencialidade das mesmas, em particular nos casos em que existe a CI e a CD através das *pipelines*.

O propósito da CI é assegurar que o *software* se encontra num estado funcional durante todo o tempo (Mysari & Bejgam, 2020). A orquestração ajuda na medida de tornar o processo mais rápido e eficiente.

Este trabalho é realizado em contexto industrial, centrando a sua aplicabilidade num projeto de uma empresa. O projeto consiste em uma solução *web*, representada através de um *Single Page Application* (SPA), contruído em ReactJS para suportar a experiência de utilização em navegador de internet. Este SPA, por sua vez, utiliza uma interface contruída em *javascript*, utilizando o *runtime* NodeJS, cuja responsabilidade é processar toda a lógica de negócio. A

Figura 5 apesenta uma representação da arquitetura SPA, adaptada de Shahzad (2017) à realidade da empresa em questão.

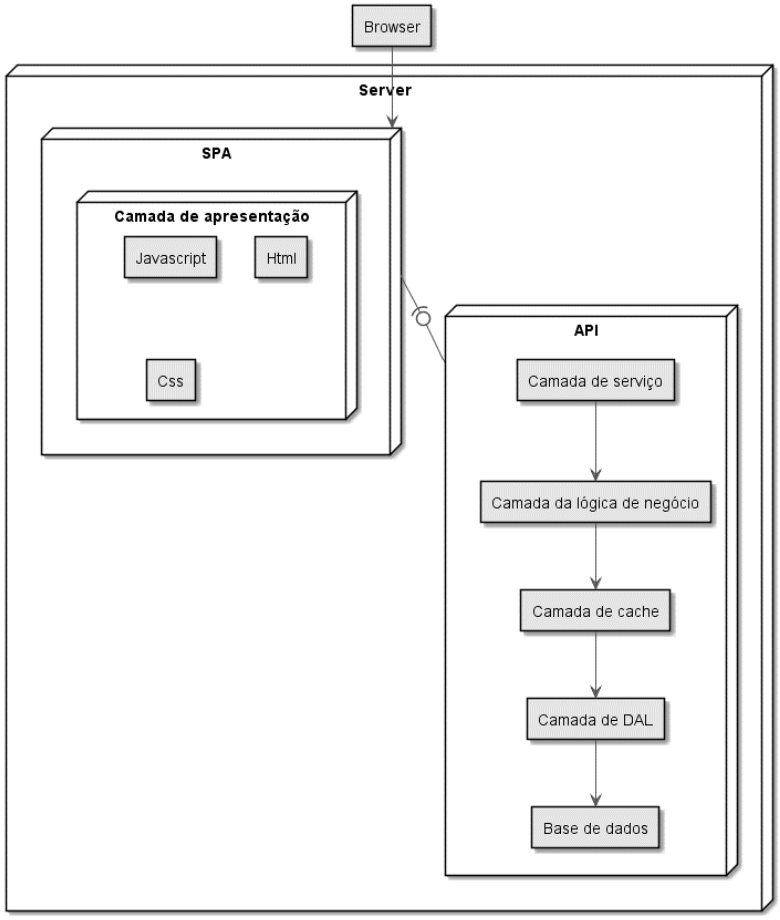

Figura 5 – Arquitetura SPA, adaptado de Shahzad (2017)

Ao analisar as diferentes ferramentas de APR disponíveis e acessíveis não se conseguiu encontrar nenhuma ferramenta adaptada à linguagem *Javascript*, com *runtime* NodeJS utilizada pela empresa. Dado que com este trabalho se pretende analisar uma ferramenta já existente, e não adaptar uma ferramenta a outra linguagem, solicitou-se autorização à empresa e migraram-se dois módulos do código para a linguagem *Java*. A escolha desta linguagem de programação relaciona-se com o facto da maioria das ferramentas que foi possível analisar, recorrerem às linguagens de programação *C* e *Java*. Desta forma, apesar de só o investigador usar a linguagem *Java*, assume-se a mesma como a utilizada pela empresa, por forma a dar continuidade a este trabalho.

4.2 Processo de avaliação

Neste contexto, o processo de avaliação da reparação automática passa por algumas fases distintas:

1. Inserir testes para que seja possível avaliá-los e, desta forma, potenciar a reparação automática;
2. Adaptar e experimentar ferramentas de reparação automática, tendo em conta a linguagem de programação e o *runtime* da solução da empresa, o que pode envolver o ajuste das técnicas de localização e reparação utilizadas pelas ferramentas;
3. Incluir a ferramenta na *pipeline*, operacionalizando o processo de reparação automática no contexto de CI. Desta forma, pretende-se notificar um programador acerca da sugestão de reparação, decorrente de um defeito.
4. Obter informação acerca do desempenho e qualidade das ferramentas para discussão de resultados e avaliação deste estudo;
5. Conhecer a perceção dos programadores da empresa quanto ao valor da utilização das ferramentas;

4.3 Escolha das ferramentas

Tendo em consideração a segunda fase do processo, relativa à adaptação e experimentação de ferramentas automáticas no contexto da empresa onde o estudo se vai desenvolver, importa selecionar uma ferramenta desenvolvida e/ou adaptada à linguagem *Java*.

Através da investigação foi possível proceder a um levantamento das ferramentas de APR mais utilizadas, conforme apresentado na tabela 11:

Tabela 11 – Ferramentas de APR

Ferramenta	**Benchmark**	**Bugs**	**Fixed (Patched)**	**Observações**	**Estudos**
Generate and validate					
Arja	Defects4J	228	18 (59)	Combina 3 abordagens diferentes	(Yuan & Banzhaf, 2018)
	QuixBugs	40	2(4)		(Ye et al., 2019)
Cardumen	Defects4J	356	0(77)	Recorre a templates pré-definidos	(Martinez & Monperrus, 2018)

Elixir	Defects4J	82	26(41)	Programa orientado ao objeto	(Saha et al., 2017)
	Bugs.jar	127	22(39)		
GenPat	Defects4J, Opensource Java projects	113	0 (19)	Recorre a um algoritmo de inferência	(Jiang et al., 2019)
JGenProg	Defects4J	224	5(27)	Recorre a hipóteses redundantes	(Martinez et al., 2017)
Jkali	Defects4J	224	1(22)	Ideal para deteter fraquezas nos testes	(Martinez et al., 2017)
JMutRepair	Defects4J	224	0(17)	Reparações baseadas em mutações	(Martinez & Monperrus, 2016)
LS Repair	Defects4J	395	19(38)	Explora repositórios de código	(K. Liu et al., 2018)
Repairnator	Defects4J	102	5(12)	Adequada a CI e CD	(Monperrus et al., 2019)
RS Repair -A	Defects4J	224	0(44)	Reparação baseada em testes	(Yuan & Banzhaf, 2018)
	QuixBugs	40	2(4)		(Ye et al., 2019)
SimFix	Defects4J	357	34(56)	Recorre a histórico de correções e código idêntico	(Jiang et al., 2018)
ssFix	Defects4J	357	20(60)	Recorre a uma base de dados de código	(Xin & Reiss, 2017)
Semantic-Driven					
Dynamoth	Defects4J	224	0(27)	Integrada na Nopol, recorre a Java Debug Interface	(Durieux & Monperrus, 2016)
Nopol	Defects4J	224	5(35)	Recorre à Satisfiability Modulo Theory e a valores angelicais	(Martinez et al., 2017)
	Real-world bugs dataset	22	13(17)		(Xuan et al., 2017)
	QuixBugs	80	1(4)		(Ye et al., 2019)
Metaprogramming-based					
NPEFix	NPE Dataset	16	0(14)	Reparação de null pointer exceptions	(Durieux et al., 2017)
	QuixBugs	40	1(2)		(Ye et al., 2019)
Framework					

Astor	Defects4J	224	0(33)	Contém 5 ferramentas de APR	(Martinez & Monperrus, 2016)
		357	0(98)		(Martinez & Monperrus, 2019a)
	QuixBugs	200	3(11)		(Ye et al., 2019)
RepairThemAll	Defects4J	395	0(187)	Contém 11 ferramentas de APR	(Durieux et al., 2019)
	Bugs.jar	1158	0(173)		
	Bears	251	0(25)		
	IntroClassJava	297	0(62)		
	QuixBugs	40	0(12)		

A Tabela 11 encontra-se dividida em quatro tipos de ferramentas que se distinguem na formulação da correção. A maioria das ferramentas analisadas (12) propõe correções com base na abordagem *generate and validate*. Em contrapartida, duas ferramentas seguem a abordagem *Semantic-Driven*, uma baseia-se em meta programação e duas constituem *frameworks*. A *benchmark* escolhida para testar cada ferramenta encontra-se na segunda coluna e pela sua análise compreende-se que a Defects4J é a mais comummente utilizada. Nas duas colunas seguintes apresentam-se o número de defeitos (*bugs*) identificados e que se tentou reparar; o número de correções confirmadas e validadas como corretas (*fixed*), tendo-se assumido zero quando este número estava omisso no estudo; e o número de correções propostas pela ferramenta, não validadas ou confirmadas (*patched*). As últimas colunas apresentam observações sobre a ferramenta e os autores responsáveis pelo seu desenvolvimento ou adaptação.

Partindo da análise das ferramentas de APR presentes na tabela, foi possível compreender e comparar diferentes ferramentas desenvolvidas ou adaptadas para a linguagem Java. No contexto deste trabalho, importa selecionar uma ou mais ferramentas a testar e comparar.

Considerando a análise da literatura apresentada no capítulo 2, excluíram-se as ferramentas baseadas em *semantic driven* e em *metaprogramming* por se considerar que não respondem aos objetivos propostos. Desta forma, optou-se por iniciar a análise pelas ferramentas mais completas, as *framework*, e pela ferramenta direcionada às *pipelines*, a Repairnator.

A Repairnator é uma ferramenta que se encontra orientada para CI e CD. Através da monitorização contínua de defeitos em CI, esta ferramenta propõe automaticamente correções para os defeitos identificados. Essas propostas são enviadas para o programador,

responsável por as aceitar ou declinar. A experiência realizada por Monperrus et al. (2019) conseguiu produzir cinco correções aceites pelos desenvolvedores. Considera-se que a principal vantagem desta ferramenta é adaptar-se a CI. Contudo, ao tentar implantar esta ferramenta, verificou-se que ela foi desenhada para Travis e Github e envia um *pull request* diretamente para o servidor. No entanto, a empresa recorre ao Jenkins e Bitbucket para correr as *pipelines*. Paralelamente, um dos objetivos do trabalho é que a ferramenta envie uma mensagem ao programador com a localização do erro e a ou as propostas de correção, pelo que o envio direto de um *pull request* não respeita os objetivos. Por estes motivos, excluiu-se a ferramenta Repairnator.

Por sua vez, a *framework* RepairThemAll foi desenvolvida através de um estudo de largo espectro, que compreendeu 11 ferramentas de APR (jGenProg, GenProg-A, jKali, Kali-A, jMutRepair, Nopol, Dynamoth, NPEFix, Arja, Cardumen e RSRepair-A) e cinco *benchmarks* (Bears, Bugs.jar, Defects4J, IntroClassJava e Quixbugs). Foram analisados 2141 defeitos nas 11 ferramentas tendo gerado entre 15 e 213 propostas de correção, existindo sobreposição de defeitos corrigidos. Concluíram também que todas as ferramentas apresentam maior afinidade com a *benchmark* Defects4J, conseguindo reparar mais defeitos do que nas restantes. (Durieux et al., 2019). No entanto, ao analisar com mais pormenor o estudo, compreende-se que este se encontra projetado para a localização e correção de projetos incluídos nas *benchmarks* designadas, não sendo possível utilizá-la no código desenvolvido na empresa.

Por outro lado, a *framework* ASTOR inclui cinco ferramentas jGenProg, jKali, jMutRepair, DeepRepair e Cardumen e duas *benchmarks* Defects4J e Quixbugs. Uma das vantagens desta *framework* é estar disponível ao público e recetiva a melhorias contínuas, através de *pull requests*. Através desta *framework*, a localização de defeitos varia consoante a ferramenta selecionada, não obstante, por predefinição, providencia a fórmula *Ochiai* para usufruto dos programadores. Esta *framework* permite a utilização das ferramentas de forma individual. Desta forma, após análise dos resultados obtidos pelas diferentes ferramentas, concluiu-se que a jGenProg apresenta o maior número de *patched bugs* no *benchmark* Defects4J. Esta ferramenta tem três operadores de reparação, inserir, substituir e remover declarações (Martinez & Monperrus, 2019b).

Analisando outras ferramentas, de forma individual, a Arja apresentou-se como uma ferramenta acessível publicamente e com boa usabilidade e bons resultados. No estudo

desenvolvido por Durieux et al. (2019) esta ferramenta apresenta o maior número de *patched bugs* na *benchmark defects4J* e no total das cinco *benchmarks*, se excluirmos as ferramentas de *semantic-driven*. A ferramenta Arja foi desenvolvida por Yuan & Banzhaf (2018) que se inspiraram em *genetic programming* para a desenvolver. Como método de localização de defeitos usa a fórmula *Ochiai* e combina três métodos distintos para propor correções de defeitos, seja num *benchmark* conhecida seja defeitos reais (*real world defects*) (Yuan & Banzhaf, 2018).

4.4 Grandezas

A partir destes pontos, torna-se importante identificar as principais grandezas que vão servir para avaliar este trabalho. Consideram-se as seguintes:

- Desempenho: consiste na capacidade de a ferramenta gerar uma versão reparada, que seja eficaz e eficiente
- Qualidade da correção: consiste na pesquisa de versões funcionais do programa, sem perda de funcionalidades;
- Satisfação do programador: consiste em compreender a perceção dos programadores acerca da usabilidade, eficácia e eficiência das ferramentas;
- Tempo: consiste na duração do processo de reparação.

4.5 Hipóteses de investigação

As hipóteses de investigação foram definidas com base na informação incluída acerca das grandezas, conforme se segue:

Hipótese A

H1: Os profissionais consideram que é possível implantar as ferramentas em *pipeline*, fazendo-o em um período de tempo adequado

H0: Os profissionais consideram que não é possível implantar as ferramentas em *pipeline*, ou fazê-lo em um período de tempo adequado

Hipótese B

H1: Os profissionais consideram que o recurso a ferramentas de APR corrige com mais eficácia os defeitos do programa

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera a eficácia da correção dos defeitos do programa

Hipótese C

H1: Os profissionais consideram que o recurso a ferramentas de APR corrige com mais eficiência os defeitos do programa

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera a eficiência da correção dos defeitos do programa

Hipótese D

H1: Os profissionais consideram que o recurso a ferramentas de APR diminui os recursos necessários ao processo de reparação

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera os recursos necessários ao processo de reparação

Hipótese E

H1: Os profissionais consideram que o recurso a ferramentas de APR diminui os custos necessários ao processo de reparação

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera os custos necessários à ao processo de reparação

Hipótese F

H1: Os profissionais consideram que o recurso a ferramentas de APR diminui o tempo necessário ao processo de reparação

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera o tempo necessário ao processo de reparação

Hipótese G

H1: Os profissionais consideram que o recurso a ferramentas de APR aumenta a sua satisfação

H0: Os profissionais consideram que o recurso a ferramentas de APR não altera a sua satisfação

4.6 Instrumentos/Métodos de Avaliação

Para avaliar as hipóteses de investigação definidas foram seguidas duas estratégias.

A primeira estratégia inclui a entrega de dois questionários aos programadores. O primeiro (Anexo 1), permite conhecer as suas expectativas face às ferramentas APR, antes do contacto com as mesmas. O segundo (Anexo 2) pretende avaliar a satisfação dos programadores. Ambos os questionários utilizam uma escala de *Likert*. Este tipo de escala permite dimensionar as respostas por parte dos utilizadores. Por sua vez, as dimensões traduzem-se em escalas que podem variar entre valores numéricos, positivos e negativos, ou outros. Assim, torna-se possível transformar as respostas em escalas de intervalo de forma a obter dados estatísticos (Trojan & Sipraki, 2015).

Paralelamente, a segunda estratégia consiste em avaliar a execução de cada uma das ferramentas individualmente e comparar esses resultados com a reparação manual. Adicionalmente, compara-se as ferramentas selecionadas entre si. Os parâmetros de avaliação são:

- a eficácia, isto é, a capacidade de encontrar uma versão reparada;
- a eficiência, que se traduz na capacidade da versão reparada selecionada ser a melhor entre as soluções candidatas;
- os recursos e o tempo necessários ao processo de reparação.

4.7 Participantes e recursos

Conforme referido, este trabalho desenvolve-se no contexto da empresa Medsky. Para a sua execução, são necessários recursos humanos e materiais/tecnológicos.

Relativamente aos recursos humanos ou participantes na investigação, importa distinguir as duas estratégias de avaliação. Assim, a implantação e avaliação das ferramentas realiza-se pelo investigador principal do estudo. Paralelamente, os restantes programadores da empresa são convidados a participar através da resposta a questionários e experimentação da *pipeline* de reparação desenvolvida.

Quanto aos recursos materiais e tecnológicos, salienta-se as ferramentas escolhidas, Arja e jGenProg; o *software* necessário para a implantação das ferramentas, tais como o Jenkins, BitBucket, Docker, entre outros; e o hardware necessário, disponibilizado pela empresa.

5. Implantação das Ferramentas

Este capítulo aborda os aspetos técnicos para a implantação de ferramentas de reparação automática em *pipeline*. Tal como já referido anteriormente, os ambientes industriais modernos recorrem à automatização para CI e CD, pelo que são apresentados os principais requisitos.

5.1 Introdução

A solução proposta para implantação é composta por duas estruturas principais distintas. Uma delas trata-se de uma imagem Docker que contém as ferramentas Arja e jGenProg preparadas para que sejam utilizadas em qualquer código fonte desenvolvido em Java. A outra, trata-se de uma *pipeline* capaz de fazer *download* do código fonte do repositório, e depois, através do Docker, executar o *container* para proceder à APR.

Desta forma, de seguida abordam-se os requisitos funcionais e não funcionais, e descrevem-se as etapas da implantação.

5.2 Requisitos

Compreender os requisitos funcionais e não-funcionais do trabalho é importante para, através deles, delinear os casos de uso necessários à implantação da solução.

Desta forma, foi possível delinear alguns casos de uso (UC) necessários à implantação de ferramentas APR em *pipeline* Jenkins:

- Iniciar a *pipeline* de APR;

- Executar *container* com as ferramentas APR;
- Enviar email com o relatório da ferramenta e/ou resultados;

5.2.1 UC1: Iniciar a pipeline de APR

Tabela 12 – UC1: Iniciar a *pipeline* APR

Objetivo:	Despoletar a *pipeline* de APR sempre que é encontrado um erro na execução dos testes
Precondições:	Falha encontrada na execução dos testes
Condição de sucesso:	A *pipeline* de APR é iniciada
Condição de Falha:	A *pipeline* de APR não é iniciada
Atores:	Programador, Jenkins
Despoletado:	*Pull request* feito pelo programador

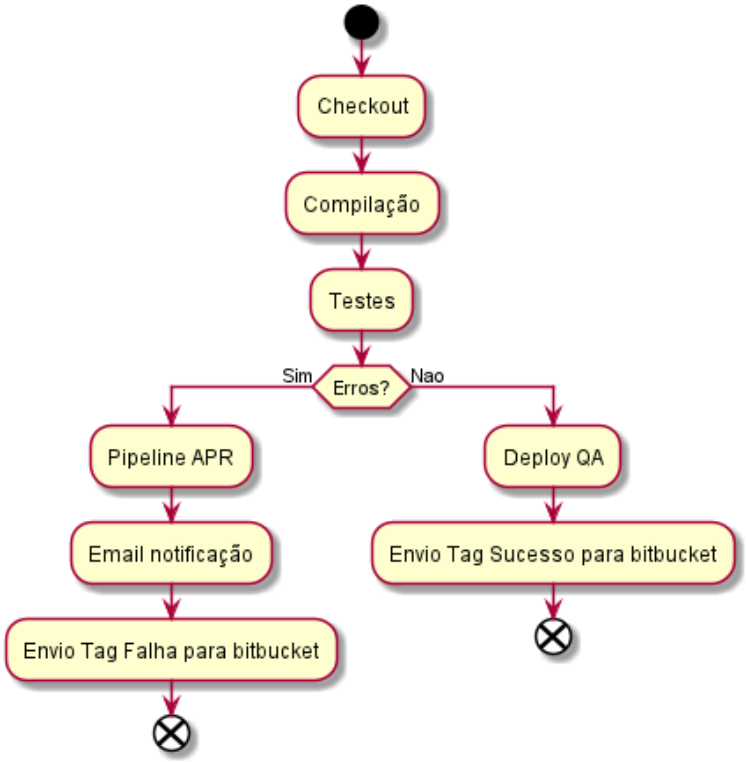

Figura 6 – Diagrama de Atividade UC1

5.2.2 UC2: Executar container com as ferramentas APR

Tabela 13 – Executar *container* com as ferramentas APR

Objetivo:	Executar a *pipeline* de APR para tentativa de reparação automática em ambiente isolado

Precondições:	A *pipeline* de entrega e integração continua encontra uma falha através da execução dos testes
Condição de sucesso:	A execução de um *container* é executada através da análise do código-fonte e execução de ferramenta de APR, gerando um relatório e/ou resultados
Condição de Falha:	A *pipeline* de APR falha
Atores:	Jenkins
Despoletado:	*Pipeline* de entrega e integração continua

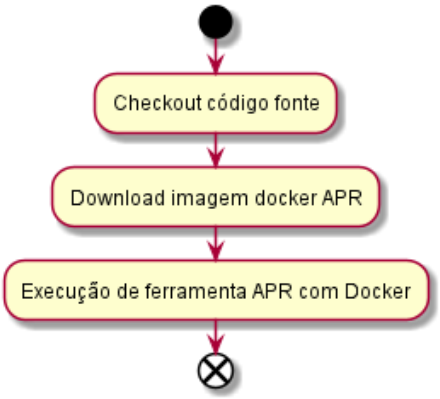

Figura 7 – Diagrama de atividade de UC2

5.2.3 UC3: Enviar de email com relatório e/ou resultados da ferramenta

Tabela 14 – Enviar email com relatório e/ou resultados da ferramenta

Objetivo:	Envio de email com o relatório da execução da ferramenta APR e, caso aplicável, o resultado com a sugestão de correção de falhas
Precondições:	Execução de *container* com as ferramentas de APR
Condição de sucesso:	O email é enviado ao programador
Condição de Falha:	A *pipeline* de APR falha
Atores:	Jenkins, Programador
Despoletado:	*Pipeline* de APR

Figura 8 – Diagrama de atividade de UC3

5.2.4 Requisitos não funcionais

Quanto aos requisitos não funcionais, estes foram derivados a partir das necessidades do estudo de avaliação e serão avaliados pelo estudo.

Tabela 15 – Requisitos Não-Funcionais

Requisitos:	**Descrição:**
Desempenho	Gerar versão reparada de forma eficaz e eficiente
Tempo	Duração do processo de reparação, medindo a sua eficiência
Qualidade da correção	Obter versões funcionais do programa, sem perda de funcionalidades, medindo a eficácia
Satisfação / Usabilidade	Perceção dos programadores acerca das ferramentas

5.3 Descrição

O contexto deste trabalho envolve a integração de duas ferramentas numa *pipeline* de CI e CD, em contexto empresarial. A empresa Medsky utiliza a plataforma Jenkins para automatizar esse processo, sendo esta uma plataforma de código aberto e providenciando centenas de plugins para que seja possível suportar todos os processos inerentes à entrega e integração de qualquer projeto.

Uma vez que a empresa também integra Docker, optou-se pela construção de uma imagem Docker, que inclui várias ferramentas e onde se torna possível executar isoladamente um *container* cuja responsabilidade é dar início ao processo de reparação automática.

5.3.1 Docker

Docker é uma plataforma de código aberto capaz de iniciar aplicações, tornando mais fácil o processo de distribuição e desenvolvimento. Através desta ferramenta torna-se possível automatizar e virtualizar as aplicações em c*ontainers*. *Containers* são ambientes isolados e leves, onde todo o processo de teste de uma aplicação pode ser simulado, mesmo antes dela ser enviada para o ambiente de produção (Bashari Rad et al., 2017).

A plataforma Docker inclui alguns componentes internos, sendo que, no contexto deste trabalho, importa abordar: imagens Docker, servidor Docker, cliente Docker e Docker *containers*. Desta forma, uma imagem Docker é composta por uma *template*, normalmente trata-se de uma versão base de um sistema operativo, onde se torna possível construir uma nova imagem Docker através da compilação de algumas instruções, através de um ficheiro *dockerfile*, dando origem a uma nova imagem Docker (Bashari Rad et al., 2017).

O servidor e cliente Docker podem encontrar-se na mesma máquina, tal como acontece no contexto deste trabalho. Através do cliente Docker é possível executar instruções para que o servidor tenha a capacidade de compilar imagens e instanciar *containers*.

As vantagens da utilização desta plataforma são: velocidade, portabilidade, escalabilidade, entrega rápida e densidade. O tempo de instância de um *container* é muito rápido, pelo mesmo ser muito pequeno, sendo que as aplicações podem ser movidas mantendo o mesmo desempenho e, em qualquer tipo de servidor. Além disso, é capaz de utilizar os recursos de forma mais eficiente, tendo a capacidade de executar múltiplos *containers* através do *hypervisor*. O *hypervisor* trata-se de uma plataforma virtual capaz de manusear múltiplos sistemas operativos, operando entre o sistema operativo e o processador (Bashari Rad et al., 2017).

Optou-se pela integração das ferramentas de APR: Arja e jGenProg numa imagem Docker customizada, que posteriormente foi enviada para um repositório do Docker Hub. O Docker Hub armazena imagens, tornando possível a sua utilização em qualquer máquina que disponha de uma instalação Docker. Para essa imagem foram preparados dois *scripts* distintos

que permitem iniciar a reparação automática, sendo o relatório armazenado na raiz do código-fonte da aplicação analisada, conforme se observa no Código 2. Além disso, sempre que existam resultados, os mesmos são compilados para esse ficheiro, tendo como finalidade o envio de uma notificação com o relatório da ferramenta e os resultados obtidos.

```
FROM maven:3.6.3-jdk-8
(…)
RUN echo "#!/bin/sh" >> /arja \
    && echo "" >> /arja \
    && echo "cd /source-code" >> /arja \
    && echo "mvn clean install -DskipTests" >> /arja \
    && echo "cd /apr/arja" >> /arja \
    && echo "rm -f /source-code/arja_report.txt || true" >> /arja \
    && echo "java -cp lib/*:bin us.msu.cse.repair.Main Arja -DsrcJavaDir
/source-code/src/ -DbinJavaDir /source-code/target/classes/ -DbinTestDir
/source-code/target/test-classes/ -Ddependences /source-code/target/medcd-0.0.1-
SNAPSHOT.jar > /source-code/arja_report.txt" >> /arja \
    && echo "cat | jq '.patches' >> /source-code/arja_report.txt" >> /arja

RUN echo "#!/bin/sh" >> /jgenprog \
    && echo "" >> /jgenprog \
    && echo "cd /source-code" >> /jgenprog \
    && echo "mvn clean install -DskipTests" >> /jgenprog \
    && echo "cd /apr/astor/examples/math_70" >> /jgenprog \
    && echo "mvn clean compile test || true" >> /jgenprog \
    && echo "cd /apr/astor" >> / jgenprog \
    && echo "rm -f /source-code/jgenprog_report.txt || true" >> /jgenprog \
    && echo "rm -f /source-code/jgenprog _results.json || true" >> /jgenprog \
    && echo "java -cp target/astor-1.1.0-jar-with-dependencies.jar
fr.inria.main.evolution.AstorMain -mode jGenProg -srcjavafolder /source-
code/src/main/java/ -srctestfolder /source-code/src/test/java/ -binjavafolder
/target/classes/ -bintestfolder /target/test-classes/ -location
/apr/astor/examples/math_70 -dependencies /apr/astor/examples/libs/junit-4.4.jar
-parameters jsonoutputname:/source-code/jgenprog_results.json > /source-
code/jGenProg_report.txt" >> /jgenprog \
    && echo "cat /source-code/jgenprog_results.json >> /source-
code/jGenProg_report.txt" >> /jgenprog \

RUN chmod +x /arja
RUN chmod +x /jgenprog
ENTRYPOINT ["sh"]
```

Código 2 – Dockerfile para imagem Docker dos dois scripts de entrada das ferramentas

Aqui definiram-se dois scripts cuja responsabilidade é o início de cada uma das ferramentas de reparação automática, no momento que o *container* Docker é instanciado. Cada um deles contém instruções para iniciar as ferramentas de APR: Arja e jGenProg. O Código 2 apresenta

entre as linhas três e 13 o script de entrada da ferramenta Arja e entre as linhas 15 e 32 o script de entrada da ferramenta jGenProg.

Além disso torna-se necessário utilizar uma imagem base que inclua o projeto Maven. Este projeto é fundamental para a compilação de código em Java. Todavia, as versões mais recentes negam o acesso a bibliotecas externas, assim optou-se pela versão 3.6.3 e o *runtime* jdk8, que permite a preparação deste ambiente.

Paralelamente, adicionaram-se algumas bibliotecas essenciais à compilação de código das ferramentas APR. Existiu também a necessidade de proceder à criação de uma pasta onde o código a analisar se encontra. Essa pasta encontra-se vazia inicialmente, todavia, no momento em que o *container* é iniciado, é fornecida uma instrução que permite mapear o conteúdo de uma pasta externa ao Docker. O caminho da pasta fornecida é o caminho da raiz do código-fonte da aplicação a ser analisada pelo conjunto de ferramentas de APR.

De seguida, encontram-se a extração das ferramentas APR da plataforma GitHub e as instruções relacionadas com a preparação das ferramentas de APR: Arja e Astor. Essas instruções estão definidas de acordo com a informação disponibilizada nos respetivos repositórios. Ainda foi adicionado o *dataset defects4j*, que se trata do banco de dados mais completo de defeitos conhecidos na linguagem java (Martinez et al., 2017; Martinez & Monperrus, 2019b, 2016; Yuan & Banzhaf, 2018).

5.3.2 Pipeline Jenkins

Previamente à realização deste projeto, a empresa já contava com um *pipeline* de integração do projeto desenvolvido em ambiente de QA e, posteriormente, de produção. Existem dois tipos de *pipeline*: a declarativa onde não existe liberdade de injeção de código, e a escrita em que já é possível injetar código para determinadas ocorrências (Denis Richtárik, 2018). A tipologia da *pipeline* utilizada pela empresa é declarativa, envolvendo uma série de passos essenciais, que fazem parte da integração, iniciando com o checkout do código-fonte do projeto e finalizando na entrega. Sempre que exista uma falha relacionada com os testes no processo de integração, a ferramenta de APR atua de forma automática e/ou manual.

A Figura 9 representa os passos da *pipeline* supramencionada:

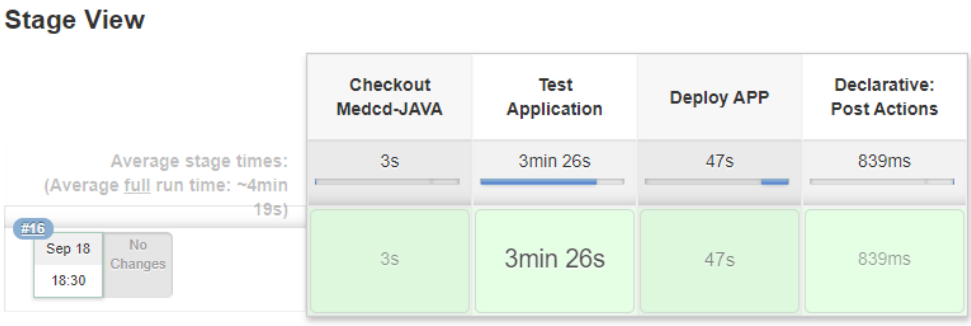

Figura 9 – *Pipeline* CI/CD do projeto da empresa

Nesta *pipeline* de integração, sempre que exista uma etapa em erro são despoletadas as ações pós-declarativas. Nesta etapa extra, sempre que a integração termine, é enviada uma nova *tag* para o BitBucket indicando o estado atual da *pipeline* de integração. É nesta etapa que se torna importante, sempre que exista uma falha, despoletar a *pipeline* de APR criada para o efeito.

A Figura 10 representa a *pipeline* de APR criada para o efeito:

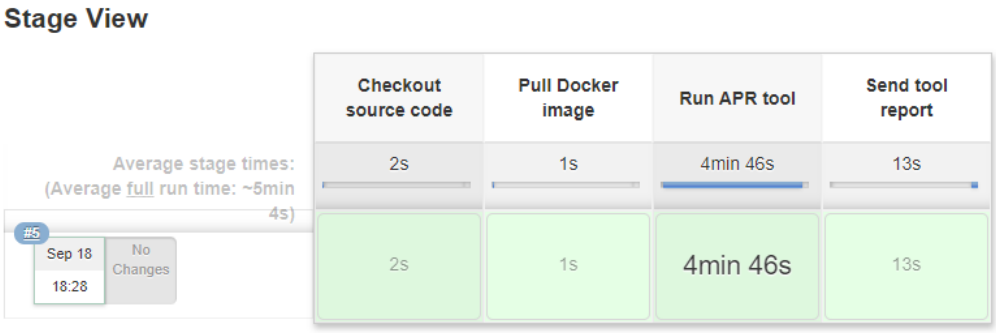

Figura 10 – *Pipeline* de APR

A *pipeline* de APR inicia-se da mesma forma da anterior, obtendo o código-fonte do ramo a ser analisado. No entanto, nesta *pipeline* é possível receber um parâmetro indicando o nome da ferramenta de APR a ser executada: *Arja* ou *jGenProg*. Seguidamente, é feito o *download* da imagem Docker anteriormente mencionada para execução das ferramentas de APR e, após este passo, conforme o valor presente no parâmetro *tool* providenciado à *pipeline*, essa ferramenta é executada, gerando um relatório e, quando aplicável, também anexando os resultados a esse ficheiro de texto.

No Docker existem diferentes alternativas para gerir os volumes criados. Esses volumes existem para que a informação seja persistida e utilizada, mesmo após o término da execução de um *container* (Xu et al., 2017). No contexto deste trabalho torna-se importante que seja da

responsabilidade da *pipeline* o envio de um email com a informação relativa à reparação automática da ferramenta executada no *container*. Assim, o Docker disponibiliza três alternativas: através do sistema de ficheiros, através do mapeamento ou através do volume de dados (Xu et al., 2017)

No caso deste projeto, no momento em que o *container* de Docker é instanciado, também, é utilizado um parâmetro -v que permite proceder ao mapeamento entre o caminho do projeto no servidor Jenkins e o *container* de Docker instanciado. Este procedimento permite ao *container* Docker escrever num ficheiro de texto dados relativos ao relatório da ferramenta e aos resultados obtidos.

Após o término da execução do *container* é possível verificar a existência do ficheiro de texto, contendo o relatório na raiz do projeto, no servidor Jenkins. Através desse ficheiro é possível extrair a informação e enviar o seu conteúdo para um email, tratando-se da última etapa da *pipeline* de APR. As Figuras 11 e 12 representam o exemplo de relatórios enviados para cada uma das ferramentas APR:

```
Fault localization starts...
Number of positive tests: 0
Number of negative tests: 0
Fault localization is finished!
AST parsing starts...
AST parsing is finished!
Detection of local variables starts...
Detection of local variables is finished!
Detection of fields starts...
Detection of fields is finished!
Detection of methods starts...
Detection of methods is finished!
Ingredient screener starts...
Ingredient screener is finished!
Initialization of manipulations starts...
Initialization of manipulations is finished!
Filtering of the tests starts...
Number of positive tests considered: 0
Filtering of the tests is finished!
```

Figura 11 – Relatório da ferramenta Arja

```
[343138] WARN AstorCoreEngine - ----------Running generation: 63, population size: 1
[343139] WARN AstorCoreEngine - ----------Running generation: 64, population size: 1
[343139] WARN AstorCoreEngine - ----------Running generation: 65, population size: 1
[343139] WARN AstorCoreEngine - ----------Running generation: 66, population size: 1
[343140] WARN AstorCoreEngine - ----------Running generation: 67, population size: 1
[343140] WARN AstorCoreEngine - ----------Running generation: 68, population size: 1
[343140] WARN AstorCoreEngine - ----------Running generation: 69, population size: 1
[343141] WARN AstorCoreEngine - ----------Running generation: 70, population size: 1
[343141] WARN AstorCoreEngine - ----------Running generation: 71, population size: 1
[343141] WARN AstorCoreEngine - ----------Running generation: 72, population size: 1
[343142] WARN AstorCoreEngine - ----------Running generation: 73, population size: 1
[343142] WARN AstorCoreEngine - ----------Running generation: 74, population size: 1
[343142] WARN AstorCoreEngine - ----------Running generation: 75, population size: 1
[343143] WARN AstorCoreEngine - ----------Running generation: 76, population size: 1
[343143] WARN AstorCoreEngine - ----------Running generation: 77, population size: 1
[343143] WARN AstorCoreEngine - ----------Running generation: 78, population size: 1
[343144] WARN AstorCoreEngine - ----------Running generation: 79, population size: 1
[343144] WARN AstorCoreEngine - ----------Running generation: 80, population size: 1
[343144] WARN AstorCoreEngine - ----------Running generation: 81, population size: 1
[343145] WARN AstorCoreEngine - ----------Running generation: 82, population size: 1
[343145] WARN AstorCoreEngine - ----------Running generation: 83, population size: 1
[343145] WARN AstorCoreEngine - ----------Running generation: 84, population size: 1
[343146] WARN AstorCoreEngine - ----------Running generation: 85, population size: 1
[343146] WARN AstorCoreEngine - ----------Running generation: 86, population size: 1
[343146] WARN AstorCoreEngine - ----------Running generation: 87, population size: 1
[343146] WARN AstorCoreEngine - ----------Running generation: 88, population size: 1
[343147] WARN AstorCoreEngine - ----------Running generation: 89, population size: 1
[343147] WARN AstorCoreEngine - ----------Running generation: 90, population size: 1
[343147] WARN AstorCoreEngine - ----------Running generation: 91, population size: 1
[343148] WARN AstorCoreEngine - ----------Running generation: 92, population size: 1
[343148] WARN AstorCoreEngine - ----------Running generation: 93, population size: 1
[343148] WARN AstorCoreEngine - ----------Running generation: 94, population size: 1
[343148] WARN AstorCoreEngine - ----------Running generation: 95, population size: 1
[343149] WARN AstorCoreEngine - ----------Running generation: 96, population size: 1
[343149] WARN AstorCoreEngine - ----------Running generation: 97, population size: 1
[343150] WARN AstorCoreEngine - ----------Running generation: 98, population size: 1
[343150] WARN AstorCoreEngine - ----------Running generation: 99, population size: 1
[343150] WARN AstorCoreEngine - ----------Running generation: 100, population size: 1
[343151] ERROR AstorCoreEngine - Stopping main loop at 100 generation
[343151] INFO AstorCoreEngine - Time Repair Loop (s): 0.038
[343152] INFO AstorCoreEngine - generationsexecuted: 100
[343152] WARN AstorCoreEngine - ----SUMMARY_EXECUTION---
[343152] WARN AstorCoreEngine - End Repair Search: NOT Found solution
[343152] WARN AstorCoreEngine - Number suspicious:1
[343154] INFO AstorMain - Time Total(s): 343.146
```

Figura 12 – Relatório da ferramenta jGenProg

5.4 Conclusão

A implantação das ferramentas *Arja* e *jGenProg* em *pipeline* industrial revelou dificuldade na sua integração, nomeadamente a nível de bibliotecas e dependências necessárias para o seu bom funcionamento. Deste modo, optou-se pela escolha de uma imagem Docker para que a execução de ambas ferramentas ocorra de forma isolada.

Atualmente, as características industriais e aceleradas do desenvolvimento de *software* requerem falhas rápidas para correções rápidas e, nem sempre estas ferramentas conseguem detetar e providenciar uma solução para os defeitos, o que decorre muitas vezes derivada da falta de testes nas aplicações, mas também pelo facto destas ferramentas requererem algum amadurecimento.

6. Avaliação e Discussão dos Resultados

Este capítulo apresenta os resultados do estudo e a sua discussão. Estes envolvem, as expectativas dos profissionais, a implantação das ferramentas e a sua posterior utilização.

6.1 Introdução

No seguimento do capítulo anterior, considera-se importante clarificar que antes de ser possível iniciar a *pipeline* de reparação, que contém as ferramentas APR, é necessário proceder à compilação do programa em análise. O processo de compilação é interrompido quando o programa contém erros conhecidos e previstos pela linguagem Java, muitas vezes envolvendo falhas relacionadas com as dependências a nível da obtenção de repositórios ou até mesmo da incompatibilidade após atualização de bibliotecas relacionadas com as mesmas (Raemaekers et al., 2014). Desta forma, alguns erros têm de ser corrigidos de forma manual para se conseguir iniciar a *pipeline* de reparação. Por conseguinte, o código do programa em análise pelas ferramentas APR encontra-se desenvolvido da melhor forma possível, sem a presença de erros conhecidos pelo desenvolvedor, tal como sucede em contexto industrial. Durante este capítulo apresenta-se a análise dos questionários aplicados a uma equipa de programadores e os resultados por hipótese de investigação. Adicionalmente, reflete-se sobre as implicações do exposto no sucesso da implantação.

6.2 Análise dos questionários de expectativas e satisfação

Uma das estratégias de avaliação deste trabalho inclui a aplicação de dois questionários, como referido no subcapítulo 4.7. O primeiro questionário denomina-se "Expectativas dos programadores sobre APR em contexto industrial". Posteriormente, solicitou-se o preenchimento do segundo questionário, intitulado "Satisfação dos programadores relativa a APR em contexto industrial". Neste questionário, os participantes são desafiados a experimentar a *pipeline* de reparação desenvolvida e responder em conformidade com a sua experiência. Neste questionário, além das questões fechadas, inclui-se uma pergunta aberta, por permitir uma abordagem livre do assunto, obtendo, naturalmente mais dados (Trojan & Sipraki, 2015).

A amostra de participantes inclui os quatro programadores, que participaram neste estudo, da empresa Medsky, onde este projeto se desenvolve. Esses programadores apresentam habilitações literárias distintas, sendo que um (25%) conclui o nível de mestrado, um (25%) concluiu o nível de licenciatura e dois (50%) realizaram pós-graduação. Relativamente à faixa etária, 50% (2) da amostra tem entre 20 e 25 anos, e 50% (2) apresenta uma faixa etária entre 26 e 30 anos. Quanto à experiência profissional na área de informática, 50% (2) apresenta uma experiência profissional superior a quatro anos, e os restantes 50% (2) abaixo.

Conforme referido, o primeiro questionário foi preenchido antes da experimentação das ferramentas APR. Assim, pretendeu-se compreender o conhecimento e contacto prévio dos participantes com as tecnologias de reparação automática e as suas expectativas quanto à possibilidade de implantação em *pipeline* e de utilização no contexto da empresa. Os resultados obtidos encontram-se expressos na Tabela 16.

Tabela 16 – Análise da questão 1 do questionário 1

	DC		D		N		C		CC	
	N	%	N	%	N	%	N	%	N	%
Estou familiarizado com a noção de APR	2	50	2	50						
Tenho experiência com APR	4	100								
Considero a APR o futuro da programação industrial			1	25	3	75				
Gostaria de fazer formação em APR					2	50	2	50		
Gostaria de aplicar APR no contexto da Medsky					1	25	3	75		
Considero que a APR ainda não se encontra suficientemente desenvolvida					3	75	1	25		
Considero o método de pesquisa manual de defeitos mais vantajoso							2	50	2	50

Legenda: DC- Discordo Completamente; D- Discordo; N- Neutro; C- Concordo; CC- Concordo Completamente; N – amostra; %- percentagem

Através dos dados da tabela anterior, pode-se perceber que a maioria dos programadores não se encontra familiarizada, nem tem qualquer experiência com APR. Observa-se também que 75% dos programadores não sabem se a APR pode ser o futuro da programação industrial, enquanto 25% discorda, o que possivelmente se deve ao desconhecimento do termo. Também se verifica que existe uma motivação na aplicabilidade de APR em contexto empresarial de mais de metade da amostra, e de formação sobre a temática em metade dos participantes. Estes resultados podem ser explicados pela vontade de aprender e pelo intervalo de idades jovem que a equipa apresenta. Todavia a amostra considera a depuração manual de defeitos mais vantajosa do que as ferramentas APR, o que pode ser justificado, não só pela falta de experimentação destas ferramentas, como também pela relutância na substituição humana pelo apuramento automático de defeitos que existe na comunidade de programadores, como defendido por Monperrus et al. (2019).

Uma outra questão presente neste questionário relaciona-se com a opinião dos programadores sobre as características da ferramenta de APR ideal e qual das características referidas como adequadas consideram mais importantes, ordenando-as.

Tabela 17 - Análise da questão 2 do questionário 1

A ferramenta de APR ideal …	DC		D		N		C		CC	
	N	%	N	%	N	%	N	%	N	%
É fácil de instalar					1	25	3	75		
Encontra-se adaptada a diferentes linguagens de programação			1	25			2	50	1	25
Consegue localizar os defeitos							1	25	3	75
Consegue propor as correções adequadas para os defeitos localizados							1	25	3	75

Legenda: DC- Discordo Completamente; D- Discordo; N- Neutro; C- Concordo; CC- Concordo Completamente; N – amostra; %- percentagem

Os resultados presentes na Tabela 17 demonstram que todas as características apresentadas são consideradas importantes, sendo a facilidade de instalação e a adaptação a diferentes línguas menos valorizadas, com 25% dos programadores a consideram neutro ou discordar da afirmação, respetivamente. Quanto à ordenação das características, a capacidade para localizar os defeitos e a capacidade de propor correções adequadas são as mais valorizadas,

seguindo-se a facilidade de instalação e, por último, a adaptabilidade a diferentes *runtimes*, o que é confirmado pelos resultados presentes na Tabela 17.

A temática da reparação automática apresenta um interesse crescente na comunidade científica (Gazzola et al., 2019; K. Liu et al., 2021; Monperrus et al., 2019). Os dados obtidos através da aplicação deste primeiro questionário permitem corroborar essa afirmação, na medida em que apresentam interesse em estudar mais a temática e até gostariam de a aplicar no contexto da empresa, apesar de considerarem a reparação manual mais vantajosa neste momento.

Relativamente às características das ferramentas de APR, os resultados obtidos através do questionário, que salientam a importância da capacidade de localizar e de propor correções adequadas, são corroborados pela literatura encontrada (Koyuncu, 2020; Monperrus et al., 2019).

Após a implementação do primeiro questionário, procedeu-se à distribuição do segundo, relativo à satisfação dos programadores. Antes de preencher o questionário, os participantes foram desafiados a explorar a *pipeline* de reparação e a testá-la, seguindo algumas indicações expressas na fase inicial do mesmo. Adicionalmente, procedeu-se novamente à obtenção do consentimento informado para participar no estudo e explicou-se o objetivo do mesmo e que seria realizado através da experimentação de duas ferramentas distintas: *Arja* e *jGenProg*.

Desta forma, o segundo questionário pode ser dividido em duas fases. A primeira fase consiste na experimentação da *pipeline* de reparação que contém ambas as ferramentas. A segunda fase relaciona-se com o preenchimento do questionário em si. Nesta fase, apresenta-se dois grupos de questões com cinco opções de resposta tipo Likert, variando entre discordo completamente e concordo completamente. Cada grupo de questões é direcionado a uma ferramenta, sendo o conteúdo das questões idêntico. Pretende-se compreender a facilidade de acesso à *pipeline* de APR, a capacidade de a ferramenta gerar documento de texto e relatório, detetar defeitos, gerar resultados e enviar o relatório por email. Adicionalmente, aborda-se a opinião dos participantes sobre a eficácia, eficiência, os recursos, os custos e o tempo das ferramentas de APR. Por último, procura-se compreender se os programadores escolheriam uma ferramenta em detrimento de outra e se escolheriam alguma delas, em detrimento da reparação manual.

A apresentação e análise dos resultados obtidos no segundo questionário encontra-se nos subcapítulos que se seguem.

6.3 Os profissionais consideram que é possível implantar as ferramentas em pipeline, fazendo-o em um período de tempo adequado

A primeira hipótese (Hipótese A) requer a análise da integração de ferramentas em *pipeline* num contexto industrial. A primeira etapa do trabalho abrange a integração de duas ferramentas de APR em *pipeline*. Desta forma, no decorrer do projeto, desde cedo existiram alguns entraves colocados pelas versões Java e Maven utilizadas entre ferramentas, uma vez que todas elas necessitam de ser compiladas antes de utilizadas, conforme explicado.

Por sua vez, o requisito de manter a *pipeline* de integração do programa em análise a funcionar da mesma forma, levou à criação de uma nova *pipeline* orientada para a APR, onde existe a possibilidade de se utilizar uma de duas ferramentas: *Arja* ou *jGenProg*.

Para que não existam conflitos entre as dependências e as versões das bibliotecas utilizadas pelas ferramentas de APR, utilizou-se o Docker com o objetivo de isolar o ambiente que executa as ferramentas de APR. Assim, as ferramentas não dependem das versões utilizadas predefinidas no servidor Jenkins.

Através da utilização de Docker, considera-se exequível a implantação das ferramentas de APR em *pipeline*, no entanto, é de notar que são necessárias alterações à imagem Docker, uma vez que nem todas as ferramentas recebem os mesmos *inputs* para a execução, e mesmo os *outputs* gerados acabam por ser diferenciados. Por este motivo, a implantação de ferramentas em *pipeline* pode consumir um período de tempo mais longo que, advém da necessidade da compatibilização de versões utilizadas para a compilação dessas ferramentas.

De forma a validar a opção H1, da hipótese A, dois fatores devem ser assegurados. A exequibilidade de implantação da ferramenta e a adequação do tempo necessário para o fazer. Pelos resultados já apresentados, compreende-se que é possível implantar as ferramentas de APR em *pipeline*, através de imagem Docker. Contudo, se o tempo necessário para tal for demasiado longo, a sua aceitação pela indústria pode ser colocada em causa.

Na realidade, em contexto industrial, o tempo é precioso. Assim, o tempo necessário para a implantação das ferramentas APR em *pipelines* e se considerar a ação exequível, deve ser o menor possível. No entanto, a resolução de conflitos entre versões, relacionadas com as dependências das ferramentas APR, implica um estudo que pode aumentar o tempo de implantação. Paralelamente, algumas características das ferramentas APR dificultam esta implantação. Na sua tese de doutoramento sobre a adequação de APR às necessidades dos programadores e da indútria, Koyuncu (2020) refere a ferramentas de APR, quando introduzidas em *pipeline*, normalmente falham pela sua dependência e obrigatoriedade de testes, que nem sempre estão previstos nos projetos das empresas, por esconderem o processo que conduz à proposta de correções e por não permitirem alterações no código e adaptações da ferramenta ao projeto da empresa. De forma a colmatar estas falhas, desenvolveram uma *framework* adaptada à linguagem C, que por esse motivo não integrou este trabalho.

As falhas elencadas por Koyuncu (2020) justificam algumas das dificuldades sentidas ao implantar as ferramentas, tendo como exemplo, a realização de testes de verificação que não é prática na empresa, mas é requisito das ferramentas, pelo que foram desenvolvidos especificamente neste contexto, o que aumenta o tempo de implantação. Em estimativa, foram necessárias cerca de 100 horas para proceder à implantação das ferramentas, partindo do ponto em que o programa está desenvolvido e incluído em uma *pipeline* de CI e CD da empresa. Os testes de verificação, a decisão pela versão mais correta e a melhor estratégia de implantação inclui-se no tempo estimado por se tratar do cenário mais similar à realidade da empresa.

Assim, apesar de ser exequível a integração das ferramentas APR em *pipeline*, este trabalho revela que fazê-lo num espaço de tempo curto, compatível com o contexto industrial, constitui um desafio.

6.4 Os profissionais consideram que o recurso a ferramentas de APR corrige com mais eficácia/eficiência os defeitos do programa

A segunda e terceira hipóteses (Hipótese B e C) relacionam-se com o estudo da eficácia e da eficiência na utilização de ferramentas APR, respetivamente. Por um lado, a eficácia consiste

em encontrar, pelo menos, uma possível solução gerada pelas ferramentas APR. Por outro lado, a eficiência mede-se através da qualidade da correção gerada por cada uma das ferramentas de APR.

A evolução das ferramentas APR tem sido no sentido de melhorar a localização de defeitos e respetiva proposta de correção (Durieux et al., 2017; K. Liu et al., 2018; Martinez & Monperrus, 2019b; Yuan & Banzhaf, 2018). No entanto, este trabalho relaciona-se com a integração das mesmas em *pipeline*, o que acarreta desafios extra no que se refere à eficácia e eficiência.

A capacidade das ferramentas APR para localizar e propor correção de defeitos das ferramentas APR depende da presença de testes que os detetem. Para tal, o recurso ao *benchmark* defects4J pretende apoiar este processo por se tratar do mais completo e ser capaz de listar os erros comumente encontrados em aplicações java. Contudo, as ferramentas APR só conseguem encontrar parte desses erros, mas não todos eles (Durieux et al., 2019; Koyuncu, 2020). A investigação desenvolvida por Koyuncu (2020) vem confirmar esta limitação, afirmando que aceder a todos os defeitos presentes neste *benchmark* nem sempre é fácil, pelo que só usaram parte deles no seu estudo. Esta dificuldade demonstra que os obstáculos para implementar uma ferramenta APR em pipeline começam pela base, o acesso, identificação e localização dos defeitos.

A ausência de resultados satisfatórios na localização de defeitos observada neste trabalho vem corroborar as dificuldades e limitações apresentadas por outros autores (Durieux et al., 2019; Koyuncu, 2020). De facto, foi possível verificar que apenas a ferramenta jGenProg detetou um caso suspeito, não o considerando um defeito e não propondo uma ou mais correções para o mesmo. Desta forma, pode afirmar-se que a ferramenta jGenProg apresentou alguma eficácia, pois detetou um caso suspeito, mas não foi eficiente porque não identificou claramente um defeito real e uma possível correção válida para o mesmo. Relativamente à ferramenta Arja, considera-se que não foi eficaz nem eficiente pela inexistência de erros detetados.

Por forma a obter mais opiniões sobre a eficácia e a eficiência de ambas as ferramentas, introduziu-se duas questões sobre a temática no questionário “Satisfação dos programadores relativa a APR em contexto industrial”. Segundo a maioria dos participantes (75%), o desempenho das ferramentas foi idêntico em relação a ambas as questões em análise, sendo

que discordam que as ferramentas sejam eficazes ou eficientes. As opiniões diferem apenas em um participante (25%) que discorda completamente que a ferramenta Arja seja eficiente ou eficaz e, relativamente à ferramenta jGenProg, um participante (25%) é neutro em relação a esta questão. Compreende-se que a análise dos resultados do questionário corrobora os resultados obtidos, pois os participantes, de uma forma geral, não consideram as ferramentas eficazes ou eficientes.

Não obstante, importa relembrar que o código foi desenvolvido sem erros conhecidos pelo desenvolvedor e que se trata de um código estático, isto é, apesar de estar incluído numa *pipeline*, não está submetido a inserções contínuas e novas versões que podem introduzir novos erros. Desta forma, assume-se a possibilidade de as ferramentas não detetarem erros pela inexistência efetiva de erros. Considerando este cenário, a avaliação da eficácia e da eficiência não é passível de ser tida em consideração.

Deste modo, considera-se que as ferramentas de APR testadas não se revelam capazes de corrigir com eficácia ou eficiência os defeitos deste programa, pelo que, ambas as hipóteses H0, da hipótese B "Os profissionais consideram que o recurso a ferramentas de APR não altera a eficácia da correção dos defeitos do programa" e C "Os profissionais consideram que o recurso a ferramentas de APR não altera a eficiência da correção dos defeitos do programa" foram confirmadas. No entanto, estes resultados podem estar influenciados pelas limitações do estudo apresentadas.

6.5 Os profissionais consideram que o recurso a ferramentas de APR diminui os recursos/custos/tempo necessários ao processo de reparação

As hipóteses D, E e F relacionam-se com a relação de recursos, custos e tempos necessários ao processo de reparação. Os recursos podem ser humanos ou físicos/materiais, por outro lado, os custos são financeiros e incluem a contratação de programadores e de ferramentas ou programas, por último, o tempo inclui o intervalo entre a localização do defeito e a correção efetiva do mesmo.

Nesta perspetiva, compreende-se que um dos objetivos e das vantagens da utilização de ferramentas APR será, na teoria, a diminuição dos recursos, humanos e materiais, dos custos e do tempo (Durieux et al., 2019; Durieux & Monperrus, 2016; Martinez & Monperrus, 2019a).

Os recursos humanos são fundamentais para o desenvolvimento de programas. Contudo, em contexto industrial, realça-se a sua importância no processo de reparação. Este processo engloba a depuração do código em falta e a respetiva correção e pode ser feita de forma manual, automática. Quando o processo é manual, o programador tem de localizar e corrigir o defeito. Se recorrer ao processo automático, como realizado neste trabalho, o desenvolvedor continua a ser necessário para analisar as sugestões de correção e validar uma delas ou ignorar e apresentar uma correção distinta. Como tal, mesmo quando o processo é automático, a presença do programador é essencial.

Contrariamente aos recursos humanos, cuja necessidade se mantém idêntica, os recursos físicos aumentam quando se recorre a ferramentas APR. Este facto deve-se à necessidade de executar a *pipeline* de APR, que contém a ferramenta, através da virtualização de um *container* Docker e recorrendo ao servidor da empresa, aumentando o seu processamento e ocupando a sua memória.

Relativamente aos custos, estes relacionam-se com os recursos, na medida em que se pode pensar nos gastos com os recursos humanos e com os recursos materiais. Assumindo a premissa defendida de que a presença do programador continua a ser necessária, compreende-se que os custos com esta fração continuam a existir e só poderão diminuir se o tempo de reparação for inferior e for possível diminuir o número de desenvolvedores contratados para este efeito.

Não obstante, este trabalho incorpora as ferramentas APR em *pipeline*, notificando os programadores com a sugestão de reparação automática. Nesta perspetiva, sempre que essa notificação existe, pode afirmar-se que os custos diminuem, uma vez que o programador tem conhecimento relativamente à localização do código que se encontra com defeito, podendo retomar a atividade anterior ao procedimento da correção mais rapidamente. No entanto, tal como já referido anteriormente, são escassas as vezes que a *pipeline* de APR é iniciada, e mesmo quando executada, nem sempre é garantida a localização de defeitos.

Quanto aos custos com os recursos financeiros, se a empresa pagar pelo processamento e memória do servidor, dado que a utilização de ferramentas APR aumenta a sua necessidade,

os seus custos também irão aumentar. Por outro lado, para realizar este trabalho selecionaram-se ferramentas de APR de acesso livre e gratuito, no entanto, algumas ferramentas são pagas, o que também aumenta os custos.

Apesar do referido, considera-se que o real impacto nas empresas se relaciona com o tempo necessário. A definição do tempo vai sempre depender daquilo que se considera. Assim sendo, o tempo necessário para implantar a ferramenta em uma *pipeline* foi discutido no sub-capitulo 6.1. e permitiu compreender que fazê-lo, num período de tempo adequado, pode ser um desafio. Discute-se agora o tempo necessário para a ferramenta APR localizar o erro e propor correções para o mesmo.

O trabalho apresentado foi desenvolvido especificamente para este contexto, pelo que é relativamente pequeno e simples. Desta forma, ambas as ferramentas demoraram um tempo aproximado de cinco minutos a gerar o relatório. Considerando a rapidez na execução, poderia afirmar-se que o tempo necessário diminui. No entanto, sendo um código pequeno, também o tempo necessário para o programador localizar e reparar o defeito de forma manual é curto. Na realidade, este achado é validado por Mousavi et al. (2020), que defende que quanto mais complexo e extenso for o código, mais tempo é necessário para a sua reparação, seja manual ou automática. Outro fator que influencia o tempo de execução é a complexidade dos defeitos, como referido por Aleti & Martinez (2020), pois quanto mais complexo o defeito, maior o tempo necessário para o localizar, sendo que por vezes as ferramentas são mesmo incapazes de localizar este tipo de erros. De forma a limitar o tempo de execução, alguns autores optam por estipular um tempo limite para a execução da ferramenta APR (Martinez & Monperrus, 2018; Saha et al., 2017).

Conforme referido, se o tempo necessário para a ferramenta localizar e propor correções for menor que o tempo que o programador necessita, a utilização de recursos e os custos necessários podem diminuir, e tornar a ferramenta competitiva, como defendido por Monperrus et al. (2019). Contudo, este autor refere também que se a correção proposta não for adequada e validada pelo programador torna-se insignificante o facto de a execução ter sido rápida.

Uma solução proposta por Monperrus et al. (2019) relaciona-se com a simbiose entre o programador e a ferramenta. Desta forma, a ferramenta de APR pode detetar erros simples, enquanto o programador se dedica aos erros mais complexos ou que a ferramenta não

consegue identificar. Não obstante, de forma a tornar as ferramentas de APR competitivas e a diminuir os recursos, os custos e o tempo, é necessário torná-las mais eficazes e eficientes no seu processo (Koyuncu, 2020; Monperrus et al., 2019).

Relativamente aos dados obtidos com o preenchimento do questionário que avalia a satisfação dos programadores, conclui-se que nenhum participante concorda ou concorda plenamente que as ferramentas diminuem os recursos, os custos ou o tempo. Analisando a Tabela 18, compreende-se ainda que a maioria dos participantes discorda das afirmações, sendo a ferramenta Arja que obtém piores resultados, com 50% e 25% dos inquiridos a discordar completamente das afirmações. Os achados da análise das respostas ao questionário vêm corroborar os resultados apresentados e a literatura encontrada sobre a temática.

Tabela 18 - Análise das questões 1 e 2 (recursos, custos e tempo) do questionário 2

		DC		D		N		C		CC	
		N	%	N	%	N	%	N	%	N	%
A ferramenta diminui os recursos necessários se comparado com a reparação manual de defeitos	Arja	2	50	1	25	1	25				
	JGen Prog	1	25	2	50	1	25				
A ferramenta diminui os custos necessários se comparado com a reparação manual de defeitos	Arja	1	25	3	75						
	JGen Prog			3	75	1	25				
A ferramenta diminui o tempo necessário se comparado com a reparação manual de defeitos	Arja	2	50	2	50						
	JGen Prog	1	25	3	75						

Legenda: DC- Discordo Completamente; D- Discordo; N- Neutro; C- Concordo; CC- Concordo Completamente; N – amostra; %- percentagem

Em suma, as ferramentas estudadas no contexto deste trabalho, não contribuem para a diminuição dos recursos e dos custos humanos, nem do tempo necessário para a reparação. Paralelamente, pode considerar-se que aumentam os recursos e os custos materiais necessários. Assim, confirma-se as opções H0 das hipóteses D “Os profissionais consideram que o recurso a ferramentas de APR não altera os recursos necessários ao processo de reparação”, da hipótese E “Os profissionais consideram que o recurso a ferramentas de APR não altera os custos necessários à ao processo de reparação” e da hipótese F “Os profissionais consideram que o recurso a ferramentas de APR não altera o tempo necessário ao processo de reparação”.

6.6 Os profissionais consideram que o recurso a ferramentas de APR aumenta a sua satisfação

A última hipótese (Hipótese G) de investigação proposta relaciona-se com a satisfação dos programadores relativamente ao uso de ferramentas de APR, no contexto deste trabalho. A satisfação é um conceito abstrato e complexo de avaliar, pelo que se optou pelo questionário para o fazer, como explicado no subcapítulo 6.1.

O referido questionário contempla questões relacionadas com a eficácia e eficiência e recursos, custos e tempo que foram abordadas nos subcapítulos 6.3 e 6.4, respetivamente. Desta forma, neste subcapítulo pretende-se analisar as primeiras sete questões, sobre o funcionamento das ferramentas, e as últimas duas questões, sobre as escolhas dos programadores. Na Tabela 19 apresentam-se os resultados obtidos no primeiro grupo de questões em análise.

Tabela 19 - Análise das questões 1 e 2 (funcionamento da *pipeline* e ferramentas) do questionário 2

		DC		D		N		C		CC	
		N	%	N	%	N	%	N	%	N	%
Considero fácil aceder à *pipeline* APR e fazer build da ferramenta	Arja							2	50	2	50
	JGen Prog							2	50	2	50
A ferramenta consegue gerar o documento de texto e o relatório	Arja							3	75	1	25
	JGen Prog							3	75	1	25
A ferramenta consegue detetar defeitos	Arja			2	50	2	50				
	JGen Prog			1	25	2	50	1	25		
A ferramenta consegue gerar resultados, isto é, propor correções para os defeitos encontrados	Arja	2	50	2	50						
	JGen Prog			3	75	1	25				
A ferramenta enviou o email com o relatório e os resultados	Arja							2	50	2	50
	JGen Prog							2	50	2	50
O relatório gerado é completo e de boa qualidade	Arja					2	50	2	50		
	JGen Prog					2	50	1	25	1	25
O tempo despendido desde o início do build até receber o email parece adequado	Arja			2	50	1	25	1	25		
	JGen Prog			1	25	1	25	2	50		

Legenda: DC- Discordo Completamente; D- Discordo; N- Neutro; C- Concordo; CC- Concordo Completamente; N – amostra; %- percentagem

A observação da Tabela 19 permite detetar as semelhanças entre as duas ferramentas. Ambas apresentam facilidade no acesso à *pipeline* de reparação e à execução da ferramenta na mesma, conseguindo gerar o documento de texto e o relatório de modo a enviar o email

esperado. No entanto, diferem nos restantes parâmetros. Relativamente à deteção de defeitos, a ferramenta jGenProg apresenta um resultado ligeiramente mais positivo, na medida em que um (25%) participante concorda que consegue detetar erros. Também relativamente aos resultados apresentados pela ferramenta, na jGenProg um (25%) programador é neutro em relação à questão e nenhum discorda completamente, o que constitui um resultado melhor que a ferramenta Arja, onde metade (50%) discorda completamente e metade (50%) discorda da afirmação. O relatório gerado apresenta uma qualidade satisfatória, sendo que, uma vez mais, a ferramenta jGenProg supera a Arja, em que nenhum participante concorda completamente com a afirmação. Relativamente à última afirmação, relacionada com o intervalo de tempo entre o *build* da ferramenta e a receção do email, as opiniões variam entre o discordo e o concordo, sendo que a ferramenta Arja apresenta dois (50%) discordo e um (25%) concordo e um (25%) neutro, e a jGenProg apresenta um (25%) discordo, dois (50%) concordo e um (25%) neutro.

Relativamente às escolhas dos programadores, foram realizadas duas questões relativas a cada ferramenta, que se apresentam na Tabela 20.

Tabela 20 - Análise das questões 1 e 2 (escolhas dos programadores) do questionário 2

Arja	DC		D		N		C		CC	
	N	%	N	%	N	%	N	%	N	%
Escolhia a ferramenta Arja para a reparação de defeitos, em detrimento da ferramenta JGenProg	1	25	2	50	1	25				
Escolhia a ferramenta Arja para a reparação de defeitos, em detrimento da reparação manual	4	100								
JGenProg	DC		D		N		C		CC	
	N	%	N	%	N	%	N	%	N	%
Escolhia a ferramenta JGenProg para a reparação de defeitos, em detrimento da ferramenta Arja					2	50	1	25	1	25
Escolhia a ferramenta JGenProg para a reparação de defeitos, em detrimento da reparação manual	4	100								

Legenda: DC- Discordo Completamente; D- Discordo; N- Neutro; C- Concordo; CC- Concordo Completamente; N – amostra; %- percentagem

A análise da Tabela 20 permite compreender que os programadores preferem a ferramenta jGenProg à ferramenta Arja. Não obstante, todos concordam que nenhuma delas supera a reparação manual de defeitos.

Estes dados vêm corroborar os resultados obtidos neste estudo. A ferramenta jGenProg apresentou um relatório mais completo e conseguiu detetar um caso suspeito, pelo que o seu desempenho foi superior à ferramenta Arja. Contudo, nenhuma foi capaz de detetar defeitos concretos e propor correções, pelo que, no geral, as opiniões sobre esses parâmetros são menos satisfatórias. Estes fatores explicam o motivo pelo qual os participantes preferem o processo manual de reparação de defeitos, pela incapacidade das ferramentas em serem competitivas com o humano (Koyuncu, 2020; Monperrus et al., 2019).

A última questão do questionário é uma pergunta de resposta aberta, onde se desafiou os participantes a referirem aspetos não contemplados que pudessem ter surgido durante a experimentação, bem como realizarem sugestões de melhoria. Analisando as respostas, apenas um desafio é referido por metade dos participantes, sendo que os restantes não apresentaram desafios. Nenhum programador apresentou sugestões de melhoria.

Desta forma, dois programadores referem que o código-fonte do programa a analisar requer uma pré-compilação e que essa compilação falhou por existirem erros mais gerais, que foi necessário corrigirem antes de proceder com a experiência. Este desafio foi também sentido pelo investigador no decorrer deste trabalho e pode estar relacionado com a ausência de erros detetados, como discutido anteriormente.

Em suma, considera-se que os programadores não ficaram satisfeitos com as ferramentas pois preferem a reparação manual. Assim, a opção H0 da Hipótese G “Os profissionais consideram que o recurso a ferramentas de APR não altera a sua satisfação” foi comprovada, sendo até possível inferir que a satisfação diminuiu.

6.7 Limitações do estudo

O presente estudo apresenta algumas limitações, nomeadamente:

- O código do programa em análise é reduzido e pouco complexo, não apresentado erros conhecidos. Consequentemente poderá não haver erros para detetar por parte das ferramentas de APR.
- Apesar do código estar inserido em um *pipeline*, que pressupõe CI, trata-se de um código estático, na medida em que não se efetuaram alterações ou adições ao código.

Desta forma, não é possível compreender a real eficácia e eficiência das ferramentas APR, em um ambiente de CI.

- O número reduzido de inquiridos torna os dados obtidos pouco significativos em termos estatísticos. Contudo, analisados no contexto da empresa Medsky e deste trabalho, fornecem informações relevantes. Todavia, a extrapolação dos dados obtidos a outros ambientes não é recomendada.

Em suma, considera-se que apesar das limitações identificadas impedirem a generalização dos resultados a outros contextos, os mesmos são válidos no presente contexto e justificam uma análise cuidada e estudos futuros sobre a temática.

6.8 Conclusão

Os resultados apresentados ao longo deste capítulo são, de uma forma geral, corroborados pela evidência científica encontrada.

A integração das ferramentas em uma *pipeline* de reparação é possível, contudo executá-la num período de tempo adequado é um desafio, pelas dificuldades encontradas durante o processo. Por outro lado, as ferramentas revelaram pouca ou nenhuma eficácia e nenhuma eficiência. Contudo, tal pode dever-se às características próprias do programa em análise que poderá não conter erros. Relativamente aos recursos materiais e aos custos associados é esperado que estes aumentem pela maior necessidade de memória e *software*.

Quanto aos recursos humanos e custos associados estes relacionam-se intrinsecamente com o tempo necessário, pois quanto maior o tempo, maiores os recursos e os custos. Contudo, vários fatores afetam o tempo requerido para a reparação automática, tais como a complexidade e a quantidade do código em análise.

Por último, a análise dos questionários permitiu compreender que a capacidade da ferramenta em localizar e sugerir correções para os defeitos são as características mais importantes identificadas pelos programadores. Adicionalmente, conclui-se que a ferramenta jGenProg apresentou um desempenho ligeiramente melhor, mas não suficientemente bom, pelo que todos os inquiridos escolhem a reparação manual de programas.

7. Conclusão

Este capítulo apresenta as principais conclusões relativas a este trabalho, nomeadamente resultados, dificuldades, limitações e propostas para trabalho futuro.

7.1 Dificuldades e resultados

Este projeto consistiu na implantação de ferramentas APR em *pipeline*, de modo a se perceber a viabilidade da sua utilização em contexto industrial. As ferramentas de APR selecionadas para o trabalho foram Arja e jGenProg. A segunda ferramenta integra-se em uma biblioteca denominada Astor.

No decorrer do trabalho existiram alguns obstáculos que dificultaram a implantação e utilização das ferramentas de APR. O obstáculo inicial teve como motivo a utilização, por parte da empresa onde este trabalho se desenvolve, de uma linguagem de programação diferente das linguagens utilizadas para o desenvolvimento das ferramentas de APR. Perante este cenário foi realizada uma migração parcial do projeto da empresa, de Javascript para a linguagem Java e assumiu-se a linguagem Java como a utilizada na empresa.

Outros problemas encontrados relacionaram-se com as atualizações de bibliotecas e a falta de manutenção das ferramentas de APR. Para tal foi encontrada uma solução que se ajusta aos requisitos de ambas as ferramentas, através do Docker, utilizando uma imagem base mais antiga, que permitisse fazer a instalação de ambas as ferramentas de APR. Através desta solução, foi possível facilitar a preparação das ferramentas, uma vez que a sua integração direta na *pipeline* revelaria muitos problemas.

Estas ferramentas dependem da existência de testes para a deteção de defeitos, o que, em contexto industrial, condiciona algumas restrições à sua utilização, uma vez que nem sempre existem testes suficientes que suportem esse processo. Além disso, apesar das ferramentas utilizarem o *benchmark* defects4j e o mesmo se revelar bastante completo, nem todos os defeitos se revelam passíveis de serem encontrados, o que revela uma dificuldade acrescida em encontrar uma solução para o problema, e em tempo útil.

Em suma, o presente trabalho objetivou a criação de uma *pipeline* de APR, na qual se torna possível indicar o repositório com o código-fonte do projeto a ser analisado por uma das ferramentas APR, produzir o relatório e o resultado, quando aplicável, e por último proceder ao envio do mesmo através de email. Considera-se que estas tarefas foram executadas com sucesso, apesar da incapacidade das ferramentas em detetar defeitos.

Relativamente às hipóteses de investigação, em todas se comprovou a opção H0 e negou a opção H1, com exceção da hipótese A, onde parte da hipótese é confirmada, pois é possível integrar as ferramentas em *pipeline*, mas a outra parte não é confirmada, mas também não pode ser refutada, pois o tempo necessário para o fazer é variável e a sua adequação ou não depende dos critérios da empresa. Apesar de se ter comprovado a hipótese que se tentava refutar, considera-se que os objetivos propostos no início do trabalho foram cumpridos. Paralelamente, conclui-se que a grandeza qualidade da correção não foi possível avaliar pela inexistência de defeitos localizados e consequentemente de propostas de correção. As grandezas desempenho, satisfação e tempo, tal como revelado pela análise das hipóteses, apresentaram uma avaliação negativa, com necessidade de melhoria das ferramentas.

7.2 Limitações

Uma das principais limitações deste trabalho refere-se com o código desenvolvido e a ser testado pela ferramenta ser reduzido e pouco complexo. Como referido, foi necessário proceder à migração de parte do código para a linguagem Java, pelo que foram selecionados apenas dois módulos e realizados testes de integração. Adicionalmente, após o desenvolvimento do código, não se procedeu a alterações e novos *commits* no BitBucket, pelo que não se conseguiu reproduzir o ambiente de integração continua esperados para uma *pipeline*.

7.3 Trabalho futuro

No futuro considera-se importante dar continuidade ao amadurecimento das ferramentas de APR em Java, promovendo a usabilidade e a facilidade de integração em novos projetos, aumentando a sua eficácia, eficiência e reduzindo o tempo de localização. Paralelamente, considera-se importante a investigação e o desenvolvimento ou adaptação de ferramentas APR a outras linguagens e plataformas, como é o caso de NodeJS.

Referências

Abdessalem, R. ben, Panichella, A., Nejati, S., Briand, L. C., & Stifter, T. (2020). Automated repair of feature interaction failures in automated driving systems. In S. Khurshid & C. S. Pasareanu (Eds.), *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020* (pp. 88–100). ACM. https://doi.org/10.1145/3395363.3397386

Agarwal, P., & Agrawal, A. P. (2014). Fault-Localization Techniques for Software Systems: A Literature Review. *SIGSOFT Softw. Eng. Notes*, *39*(5), 1–8. https://doi.org/10.1145/2659118.2659125

Aguarón, J., Escobar, M. T., & Moreno-Jiménez, J. M. (2021). Reducing inconsistency measured by the geometric consistency index in the analytic hierarchy process. *European Journal of Operational Research*, *288*(2), 576–583. https://doi.org/https://doi.org/10.1016/j.ejor.2020.06.014

Alarcon, G. M., Walter, C., Gibson, A. M., Gamble, R. F., Capiola, A., Jessup, S. A., & Ryan, T. J. (2020). Would You Fix This Code for Me? Effects of Repair Source and Commenting on Trust in Code Repair. *Systems*, *8*(1). https://doi.org/10.3390/systems8010008

Aleti, A., & Martinez, M. (2020). *E-APR: Mapping the Effectiveness of Automated Program Repair*.

Asad, M., Ganguly, K., & Sakib, K. (2019). *Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair*. 328–332. https://doi.org/10.1109/ICSME.2019.00050

Assiri, F., & Bieman, J. (2017). Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal*, *25*. https://doi.org/10.1007/s11219-016-9312-z

Bashari Rad, B., Bhatti, H., & Ahmadi, M. (2017). An Introduction to Docker and Analysis of its Performance. *IJCSNS International Journal of Computer Science and Network Security*, *173*, 8.

Borza, J. (2011). FAST diagrams: The foundation for creating effective function models. *General Dynamics Land Systems*.

Chang, H., Mariani, L., & Pezzè, M. (2013). Exception Handlers for Healing Component-Based Systems. *ACM Trans. Softw. Eng. Methodol.*, *22*(4). https://doi.org/10.1145/2522920.2522923

Daou, A., Mallat, C., Chammas, G., Cerantola, N., Kayed, S., & Saliba, N. A. (2020). The Ecocanvas as a business model canvas for a circular economy. *Journal of Cleaner Production*, *258*, 120938. https://doi.org/https://doi.org/10.1016/j.jclepro.2020.120938

DeMarco, F., Xuan, J., le Berre, D., & Monperrus, M. (2014). Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, 30–39. https://doi.org/10.1145/2593735.2593740

Denis Richtárik. (2018). *Cloud Integration of Continuous Integration Declarative Pipelines*.

Durieux, T., Cornu, B., Seinturier, L., & Monperrus, M. (2017). *Dynamic Patch Gener-ation for Null Pointer Exceptions Using Metaprogramming*. 349–358. https://doi.org/10.1109/SANER.2017.7884635ï

Durieux, T., Madeiral, F., Martinez, M., & Abreu, R. (2019). Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 302–313. https://doi.org/10.1145/3338906.3338911

Durieux, T., & Monperrus, M. (2016). DynaMoth: Dynamic code synthesis for automatic program repair. *Proceedings - 11th International Workshop on Automation of Software Test, AST 2016*, 85–91. https://doi.org/10.1145/2896921.2896931

Fei, L., Hu, Y., Xiao, F., Chen, L., & Deng, Y. (2016). A Modified TOPSIS Method Based on D Numbers and Its Applications in Human Resources Selection. *Mathematical Problems in Engineering*, *2016*, 1–14. https://doi.org/10.1155/2016/6145196

Gaikwad, S. G., & Shah, M. A. (2015). Pipeline Orchestration for Test Automation using Extended Buildbot Architecture. In *International Journal of Computer Applications*.

Gazzola, L., Micucci, D., & Mariani, L. (2019). Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.*, *45*(1), 34–67. https://doi.org/10.1109/TSE.2017.2755013

Goues, C., Forrest, S., & Weimer, W. (2013). Current Challenges in Automatic Software Repair. *Software Quality Journal*, *21*(3), 421–443. https://doi.org/10.1007/s11219-013-9208-0

Hua, J., Zhang, M., Wang, K., & Khurshid, S. (2018). Towards Practical Program Repair with On-Demand Candidate Generation. *Proceedings of the 40th International Conference on Software Engineering*, 12–23. https://doi.org/10.1145/3180155.3180245

Jiang, J., Ren, L., Xiong, Y., & Zhang, L. (2019). Inferring program transformations from singular examples via big code. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 255–266. https://doi.org/10.1109/ASE.2019.00033

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., & Chen, X. (2018). Shaping program repair space with existing patches and similar code. *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 298–309. https://doi.org/10.1145/3213846.3213871

Kelk, D., Jalbert, K., & Bradbury, J. S. (2013). Automatically Repairing Concurrency Bugs with ARC. *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools - Volume 8063*, 73–84. https://doi.org/10.1007/978-3-642-39955-8_7

Koen, P. A. (2004). The Fuzzy Front End for Incremental, Platform, and Breakthrough Products. In *The PDMA Handbook of New Product Development* (pp. 81–91). John Wiley & Sons, Ltd. https://doi.org/https://doi.org/10.1002/9780470172483.ch6

Kou, R., Higo, Y., & Kusumoto, S. (2016). A Capable Crossover Technique on Automatic Program Repair. *7th International Workshop on Empirical Software Engineering in Practice, IWESEP@SANER 2016, Osaka, Japan, March 13, 2016*, 45–50. https://doi.org/10.1109/IWESEP.2016.15

Koyuncu, A. (2020). *Boosting Automated Program Repair for adoption by practitioners*. University of Luxembourg- The Faculty of Sciences, Technology and Medicine.

Koyuncu, A., Bissyandé, T. F., Klein, J., & Traon, Y. le. (2020). *FlexiRepair: Transparent Program Repair with Generic Patches*. http://arxiv.org/abs/2011.13280

le Goues, C., Pradel, M., & Roychoudhury, A. (2019). Automated program repair. In *Communications of the ACM* (Vol. 62, Issue 12, pp. 56–65). Association for Computing Machinery. https://doi.org/10.1145/3318162

Le, X. B. D., Thung, F., Lo, D., & Goues, C. le. (2018). Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, *23*(5), 3007–3033. https://doi.org/10.1007/s10664-017-9577-2

Le, X.-B. D. (2016). Towards Efficient and Effective Automatic Program Repair. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 876–879. https://doi.org/10.1145/2970276.2975934

Le, X.-B. D., Lo, D., & Goues, C. le. (2016a). History Driven Program Repair. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, *1*, 213–224.

Le, X.-B. D., Lo, D., & Goues, C. le. (2016b). Empirical Study on Synthesis Engines for Semantics-Based Program Repair. *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 423–427. https://doi.org/10.1109/ICSME.2016.68

Le, X.-B. D., Thung, F., Lo, D., & Goues, C. le. (2018). Overfitting in Semantics-Based Automated Program Repair. *Proceedings of the 40th International Conference on Software Engineering*, 163. https://doi.org/10.1145/3180155.3182536

Lin, Y., & Kulkarni, S. S. (2014). Automatic Repair for Multi-Threaded Programs with Deadlock/Livelock Using Maximum Satisfiability. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 237–247. https://doi.org/10.1145/2610384.2610398

Liu, K., Koyuncu, A., Kim, K., Kim, D., & Bissyande, T. F. (2018). LSRepair: Live Search of Fix Ingredients for Automated Program Repair. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, *2018-Decem*, 658–662. https://doi.org/10.1109/APSEC.2018.00085

Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J., & Bissyandé, T. F. (2021). A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, *171*. https://doi.org/10.1016/j.jss.2020.110817

Liu, Y., Zhang, L., & Zhang, Z. (2018). A Survey of Test Based Automatic Program Repair. *Journal of Software*, *13*, 437–452. https://doi.org/10.17706/jsw.13.8.437-452

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., & Monperrus, M. (2017). Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, *22*(4), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

Martinez, M., & Monperrus, M. (2019a). Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, *151*, 65–80. https://doi.org/10.1016/j.jss.2019.01.069

Martinez, M., & Monperrus, M. (2019b). Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, *151*, 65–80. https://doi.org/10.1016/j.jss.2019.01.069

Martinez, M., & Monperrus, M. (2016). ASTOR: A program repair library for Java. *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, 441–444. https://doi.org/10.1145/2931037.2948705

Martinez, M., & Monperrus, M. (2018). Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *11036 LNCS*, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

Mechtaev, S., Yi, J., & Roychoudhury, A. (2016). Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. *Proceedings of the 38th International Conference on Software Engineering*, 691–701. https://doi.org/10.1145/2884781.2884807

Meertens, L. O., Iacob, M. E., Nieuwenhuis, L. J. M., van Sinderen, M. J., Jonkers, H., & Quartel, D. (2012). Mapping the Business Model Canvas to ArchiMate. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 1694–1701. https://doi.org/10.1145/2245276.2232049

Mehne, B., Yoshida, H., Prasad, M., Sen, K., Gopinath, D., & Khurshid, S. (2018). Accelerating Search-Based Program Repair. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 227–238.

Monperrus, M. (2018). Automatic Software Repair: a bibliography. *ACM Computing Surveys*, *51*(1), 1–24. https://doi.org/10.1145/3105906

Monperrus, M., Urli, S., Durieux, T., Martinez, M., Baudry, B., & Seinturier, L. (2019). *Repairnator Patches Programs Automatically*.

Mousavi, S. A., Babani, D. A., & Flammini, F. (2020). *Obstacles in Fully Automatic Program Repair: A survey*.

Mysari, S., & Bejgam, V. (2020). Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible. *2020 International Conference on Emerging Trends in Information Technology and Engineering (Ic-ETITE)*, 1–4. https://doi.org/10.1109/ic-ETITE47903.2020.239

Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013). SemFix: Program Repair via Semantic Analysis. *Proceedings of the 2013 International Conference on Software Engineering*, 772–781.

Phung, Q.-N., & Lee, E. (2021). Incremental Formula-Based Fix Localization. *Applied Sciences*, *11*(1). https://doi.org/10.3390/app11010303

Psaier, H., & Dustdar, S. (2011). A survey on self-healing systems: Approaches and systems. *Computing*, *91*, 43–73. https://doi.org/10.1007/s00607-010-0107-y

Qi, Y., Mao, X., & Lei, Y. (2013). Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. *2013 IEEE International Conference on Software Maintenance*, 180–189.

Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 24–36. https://doi.org/10.1145/2771783.2771791

Raemaekers, S., Van Deursen, A., & Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*, 215–224. https://doi.org/10.1109/SCAM.2014.30

Saha, R. K., Lyu, Y., Yoshida, H., & Prasad, M. R. (2017). ELIXIR: Effective Object Oriented Program Repair. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 648–659.

Samirni, H., Schafer, M., Artzi, S., Millstein, T., Tip, F., & Hendren, L. (2012). Automated repair of HTML generation errors in PHP applications using string constraint solving. *Proceedings - International Conference on Software Engineering*, 277–287. https://doi.org/10.1109/ICSE.2012.6227186

Shahzad, F. (2017). Modern and Responsive Mobile-enabled Web Applications. *Procedia Computer Science*, *110*, 410–415. https://doi.org/https://doi.org/10.1016/j.procs.2017.06.105

Tan, S. H., & Roychoudhury, A. (2015). Relifix: Automated Repair of Software Regressions. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 471–482.

Tao, Y., Kim, J., Kim, S., & Xu, C. (2014). Automatically Generated Patches as Debugging Aids: A Human Study. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 64–74. https://doi.org/10.1145/2635868.2635873

Trojan, R. M., & Sipraki, R. (2015). Perspectivas de estudos comparados a partir da aplicação da escala Likert de 4 pontos: um estudo metodológico da pesquisa TALIS. *Revista Ibero-Americana de Estudos Em Educação*, *10*(2), 275–300. https://doi.org/10.21723/riaee.v10i2.7761

Wei, yi, Pei, Y., Furia, C., Silva, L., Buchholz, S., Meyer, B., & Zeller, A. (2010). Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering*, *40*, 61–72. https://doi.org/10.1145/1831708.1831716

Wen, M., Chen, J., Wu, R., Hao, D., & Cheung, S.-C. (2018). Context-Aware Patch Generation for Better Automated Program Repair. *Proceedings of the 40th International Conference on Software Engineering*, 1–11. https://doi.org/10.1145/3180155.3180233

Wilkerson, J. L., & Tauritz, D. R. (2011). Scalability of the Coevolutionary Automated Software Correction System. *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, 243–244. https://doi.org/10.1145/2001858.2001995

Wong, W., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, *42*, 1. https://doi.org/10.1109/TSE.2016.2521368

Xin, Q., & Reiss, S. P. (2017). Leveraging Syntax-Related Code for Automated Program Repair. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 660–670.

Xu, Q., Awasthi, M., Malladi, K. T., Bhimani, J., Yang, J., & Annavaram, M. (2017). Docker characterization on high performance SSDs. *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 133–134. https://doi.org/10.1109/ISPASS.2017.7975282

Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., & Monperrus, M. (2017). Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, *43*(1), 34–55. https://doi.org/10.1109/TSE.2016.2560811

Ye, H., Martinez, M., Durieux, T., & Monperrus, M. (2019). *A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark*. https://doi.org/10.1016/j.jss.2020.110825

Yuan, Y., & Banzhaf, W. (2018). ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2018.2874648

Zhang, M., Li, X., Zhang, L., & Khurshid, S. (2017). Boosting Spectrum-Based Fault Localization Using PageRank. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 261–272. https://doi.org/10.1145/3092703.3092731

Zuddas, D., Jin, W., Pastore, F., Mariani, L., & Orso, A. (2014). MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 815–826. https://doi.org/10.1145/2642937.2643014

Anexos

Anexo 1- Questionário 1 “Expectativas dos programadores sobre APR em contexto industrial”

Expectativas dos programadores sobre APR em contexto industrial

O presente questionário pretende compreender as expectativas dos programadores da empresa Medsky em relação às ferramentas de Reparação Automática de Programas (APR) em contexto de integração contínua (CI) e entrega contínua (CD).

Desenvolvido no âmbito da dissertação de mestrado em Engenharia Informática, na área de especialização de Engenharia de Software, do ISEP, entitulada "Reparação automática de programas: Exploração do potencial em contexto industrial".

A participação no estudo é voluntária e a qualquer momento pode desistir. Todas as respostas são anónimas e serão usadas exclusivamente no contexto académico exposto. O preenchimento e entrega do questionário pressupõe o consentimento para participação no estudo.

(Tempo estimando de resposta: 5mts)

Obrigado pela vossa colaboração,

André Queirós.

Declaração de consentimento

Declaro que a minha participação neste estudo é voluntária, que compreendi o objetivo do estudo e fornecerei respostas honestas e ponderadas.

☐ Sim, dou o meu consentimento para participar no estudo.

PERFIL DO PARTICIPANTE

Esta secção pretende compreender melhor o perfil de distribuição dos participantes. Se não desejar fornecer essas informações, poderá deixar os campos em branco.

Educação

Selecione o maior nível de educação que completou até ao momento.

☐ Secundário

☐ Licenciatura

☐ Mestrado

☐ Doutoramento

☐ Outro (especifique) _____

Anos de experiência profissional na área de programação informática _____

CONHECIMENTO E EXPECTATIVAS SOBRE FERRAMENTAS APR

Nesta secção pretende-se compreender o conhecimento e as expectativas dos participantes sobre as ferramentas APR. Selecione a opção que melhor se adequa.

	Discordo completamente	Discordo	Neutro	Concordo	Concordo completamente
Estou familiarizado com a noção de APR.					
Tenho experiência com APR					
Considero a APR o futuro da programação industrial					

Gostaria de fazer formação em APR					
Gostaria de aplicar APR no contexto da Medsky					
Considero que a APR ainda não se encontra suficientemente desenvolvida					
Considero o método de pesquisa manual de defeitos mais vantajoso					

CARACTERÍSTICAS DA APR IDEAL

Nesta secção pretende-se compreender as características que os participantes consideram essenciais numa ferramenta APR.

Selecione a opção que melhor se adequa.

A ferramenta de APR ideal …		Discordo completamente	Discordo	Neutro	Concordo	Concordo completamente
1	… É fácil de instalar					
2	… Encontra-se adaptada a diferentes linguagens de programação					
3	… Consegue localizar os defeitos					
4	… Consegue propor as correções adequadas para os defeitos localizados					

Considerando as afirmações anteriores que concordou ou concordou totalmente. Ordene-as por ordem de importância, da mais importante para a menos importante.

_____

Obrigado pela vossa colaboração.

Anexo 2 - Questionário 2 “Satisfação dos programadores relativa a APR em contexto industrial”

Satisfação dos programadores relativa a APR em contexto industrial

O presente questionário pretende conhecer a satisfação dos programadores da empresa Medsky em relação às ferramentas de Reparação Automática de Programas (APR) em contexto de integração contínua (CI) e entrega contínua (CD), especificamente as ferramentas Arja e JGenProg.

Desenvolvido no âmbito da dissertação de mestrado em Engenharia Informática, na área de especialização de Engenharia de Software, do ISEP, entitulada "Reparação automática de programas: Exploração do potencial em contexto industrial".

A participação no estudo é voluntária e a qualquer momento pode desistir. Todas as respostas são anónimas e serão usadas exclusivamente no contexto académico exposto. O preenchimento e entrega do questionário pressupõe o consentimento para participação no estudo.

(Tempo estimando de resposta: 20mts,10mts para a experimentação,10mts para resposta)

Obrigado pela vossa colaboração,

André Queirós.

Declaração de consentimento

Declaro que a minha participação neste estudo é voluntária, que compreendi o objetivo do estudo e fornecerei respostas honestas e ponderadas.

☐ Sim, dou o meu consimento para participar no estudo.

PERFIL DO PARTICIPANTE

Esta secção pretende compreender melhor o perfil de distribuição dos participantes. Se não desejar fornecer essas informações, poderá deixar os campos em branco.

Educação

Selecione o maior nível de educação que completou até ao momento.

- ☐ Secundário
- ☐ Licenciatura
- ☐ Mestrado
- ☐ Doutoramento
- ☐ Outro (especifique) _____

Anos de experiência profissional na área de programação informática _____

EXPERIMENTAÇÃO DAS FERRAMENTAS APR: ARJA E JGENPROG

Antes de proceder ao preenchimento do questionário, solicita-se que experimente a *pipeline* no servidor jenkins, desenvolvida no contexto deste trabalho e que inclui uma imagem docker, permitindo iniciar uma das duas ferramentas disponíveis. É possível selecionar a ferramenta através de um parametro fornecido à *pipeline*, no momento em a mesma é iniciada.

Por um período de 10 minutos, considerando 5 minutos para cada ferramenta, solicita-se que experimente a *pipeline*, fazendo correr as ferramentas à procura de erros.

Posteriormente, preencha o questionário com as respostas que considera mais adequadas.

REFLEXÃO SOBRE A FERRAMENTA ARJA

Nesta secção pretende-se compreender a experiência do participante em relação à ferramenta Arja. Selecione a opção que melhor se adequa.

	Discordo completamente	Discordo	Neutro	Concordo	Concordo completamente
Considero fácil aceder à *pipeline* APR e fazer build da ferramenta Arja					
A ferramenta consegue gerar o documento de texto e o relatório					
A ferramenta consegue detetar defeitos					
A ferramenta consegue gerar resultados, isto é, propôr correções para os defeitos encontrados					
A ferramenta enviou o email com o relatório e os resultados					
O relatório gerado é completo e de boa qualidade					
O tempo dispendido desde o início do build até receber o email parece adequado					
A ferramenta é eficaz, isto é, consegue propôr soluções					
A ferramenta é eficiente, isto é, consegue propôr soluções adequadas					
A ferramenta diminui os recursos necessários se comparado com a reparação manual de defeitos					
A ferramenta diminui os custos necessários se comparado com a reparação manual de defeitos					
A ferramenta diminui o tempo necessário se comparado com a reparação manual de defeitos					
Escolhia a ferramenta Arja para a reparação					

de defeitos, em detrimento da ferramenta JGenProg					
Escolhia a ferramenta Arja para a reparação de defeitos, em detrimento da reparação manual					

REFLEXÃO SOBRE A FERRAMENTA JGENPROG

Nesta secção pretende-se compreender a experiência do participante em relação à ferramenta JGenProg. Selecione a opção que melhor se adequa.

	Discordo completamente	Discordo	Neutro	Concordo	Concordo completamente
Considero fácil aceder à *pipeline* APR e fazer build da ferramenta JGenProg					
A ferramenta consegue gerar o documento de texto e o relatório					
A ferramenta consegue detetar defeitos					
A ferramenta consegue gerar resultados, isto é, propôr correções para os defeitos encontrados					
A ferramenta enviou o email com o relatório e os resultados					
O relatório gerado é completo e de boa qualidade					
O tempo dispendido desde o início do build até receber o email parece adequado					
A ferramenta é eficaz, isto é, consegue propôr soluções					
A ferramenta é eficiente, isto é, consegue propôr soluções adequadas					
A ferramenta diminui os recursos necessários se comparado com a reparação					

manual de defeitos					
A ferramenta diminui os custos necessários se comparado com a reparação manual de defeitos					
A ferramenta diminui o tempo necessário se comparado com a reparação manual de defeitos					
Escolhia a ferramenta JGenProg para a reparação de defeitos, em detrimento da ferramenta Arja					
Escolhia a ferramenta JGenProg para a reparação de defeitos, em detrimento da reparação manual					

REFLEXÃO SOBRE A EXPERIÊNCIA

Nesta secção solicita-se que escreva sobre as dificuldades sentidas que não foram abordadas nas seções anteriores. Poderá também realizar sugestões de melhoria.

Obrigado pela vossa colaboração.