Este trabalho foi realizado no âmbito do projeto PTDC/CCI-INF/31581/2017 - POCI-01-0145-FEDER-031581 - Engenharia de Software Aumentada de Biofeedback (BASE), apoiado pelo Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra - DEI-FCTUC.

Agradecimentos

Em primeiro lugar, os meus agradecimentos é dirigido ao Todo-Poderoso Soberano Senhor *Jeová* criador de todas as coisas que me tem apoiado incondicionalmente apesar das minhas imperfeições, dando-me orientações e consolo sempre. (Sal. 83:18 e Fil. 4:6,7 - *Bíblia Tradução do Novo Mundo*)

Agradeço e dedico esta tese a minha querida esposa, Marquinha Domingos Cahisso pelo apoio dado e o sacrifício de ficar com os nossos filhos, deixando-me mais a vontade nesta estrada de aprendizagem que foi o mestrado. Os meus agradecimentos vão igualmente aos meus queridos filhos, Jacinta Pedro Cahisso (Jacira), Sílvio da Silva Cahisso, Luzia Pedro Cahisso (Luzineide) e Francisco Bongo Pedro Cahisso (Cahifran) pela abnegação dos seus direitos terem um pai ao lado.

Agradeço ainda aos meus amigos pelo apoio prestado, apesar da distância ou de nos conhecermos a pouco tempo: Alfredo Manuel José, Francisco Capita, Kissema Eduardo Rafael, Jomar Domingos, Pedro Kissumba Phd, Herinque Madeira Phd e todo aquele que de uma ou de outra forma contribui para a realização desta formação.

Por último mas não menos importante, agradeço a minha querida mãe Jacinta Bongo Caquele e os meus irmãos de sangue por estarem sempre presente na minha vida.

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

IDE com Destaque de Biofeedback Inteligente

Eclipse UC

Bongo Francisco Cahisso

Dissertação no âmbito do Mestrado em Engenharia Informática, Especialização em Sistemas Inteligentes Orientado pelos Professores Doutor Henriques Madeira e Raul Barbosa e Apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

Janeiro de 2020

UNIVERSIDADE D
COIMBRA

2

Abstract

Software faults (i.e., bugs) may have disastrous consequences depending on the area of use of the software. The mitigation of programmers' errors in software development is carried out by software quality control, namely by testing and software inspection, but the applied processes, in addition to being laborious (due to the drastic increase in the number of lines of code) also entail very high costs. The integrated development environments (IDE) that are used in software development only have mechanisms for identifying compilation and execution errors. That is, IDEs do not identify the true software faults, which result from human errors as consequence of high cognitive load, mental fatigue, stress and other emotional states in which the programmer constantly live. In this master thesis we present the first prototype of a radical approach that allows associating the programmer's cognitive load to actual lines of code or lexical *tokens*, through *biofeedback* annotations. The prototype developed consists of an expanded version of the Eclipse IDE, which makes annotations and markup of source code using *biofeedback* data from programmers, including data from operations performed on source code and metrics of code complexity. With this prototype, and with the use of *Machine Learning* on the annotation data, it opens the way to the development of models for estimating error density and software risk analysis, through the use of information on the emotional and cognitive states of the programmer, in conjunction with code complexity metrics.

Keywords

Software failures, Software Engineering, Programmer biofeedback, cognitive overload, plugins, data synchronization, source code complexity, source code enhancement, source code annotation, Eclipse IDE, Machine Learning.

Resumo

As falhas de softwares (i.e., bugs) podem ter consequências desastrosas dependendo da área de utilização do software. O controlo dos erros dos programadores no desenvolvimento de software é realizado pelas equipas de testes, controlo de qualidade e de inspecção de software, mas os processos aplicados, para além de serem trabalhosos (devido do aumento drástico do número de linhas de código) também acarretam custos elevados. Os ambientes de desenvolvimento integrado (IDE) que se usam no desenvolvimento de software possuem apenas mecanismo de identificação de erros de compilação e de execução, ou seja, não identificam as verdadeiras falhas de software, que resultam de erros humanos em consequência de elevada carga cognitiva, cansaço mental, stress e outros estados emocionais em que o programador vive constantemente. Nesta tese de mestrado apresentamos o primeiro protótipo de uma abordagem radical que permite associar a carga cognitiva do programador a linhas de código e a *tokens* lexicais, através de anotações de *biofeedback*. O protótipo desenvolvido consiste numa versão expandida do IDE Eclipse, que realiza anotações e marcações de código-fonte usando dados de *biofeedback* dos programadores, dados das operações realizadas no código-fonte e métricas de complexidade do código. Com este protótipo e com o uso de *Machine Learning* sobre os dados de anotação, abre-se caminho ao desenvolvimento de modelos de estimativa da densidade de erros e de análise de risco de software, através do uso da informação sobre os estados emocionais e cognitivos do programador, em conjunto com métricas de complexidade de código.

Palavras-Chave

Falhas de Software, Engenharia de Software, Biofeedback de programadores, carga cognitiva, plugins, sincronização de dados, complexidade do código-fonte, realce do código-fonte, anotação do código-fonte, IDE Eclipse, Machine Learning.

Conteúdo

Acrônimo

API Interface de Programação de Aplicações. 6, 8, 14, 29, 31, 39, 40, 52

AWT Abstract Window Toolkit. 31

BASE Engenharia de Software Aumentada de Biofeedback. i, 2, 8, 9, 16, 17, 23, 29, 38, 39, 41, 52, 56, 63, 64

CS Confiabilidade de Software. 1

CVS Sistema de Versões Concorrentes. 30, 31

DEI Departamento de Engenharia Informática. 22

EDA Actividade Electrodérmica. 41

EI Engenharia Informática. 3

EPL Eclipse Public Licence. 27, 29, 35, 45

ER Engenharia de Requisitos. 3, 42, 43, 46

ES Engenharia de Software. 1–3

GP Gestão de Projecto. 3, 22

GUI Graphical User Interface. 31

IA Inteligência Artificial. 3, 13, 64

IBM International Business Machines Corporation. 1, 29

IC Investigação Científica. 9

IDE Ambiente de Desenvolvimento Integrado. 3, 6, 8, 11, 12, 16, 19, 21, 22, 29, 31, 32, 38–40, 44–46, 49, 51, 57, 59, 60, 63

JDT Kit de Ferramentas de Desenvolvimento em Java. 29

KLoC 1.000 linhas de códigos. 1

ML Aprendizagem Máquina (Machine Learning). 14

NTP protocolo de tempo de rede do Inglês *Network Time Protocol*. 27

ODC Classificação de Defeitos Ortogonais. 1

OS Sistema Operativo. 11, 31

PDE Ambiente de Desenvolvimento de Plug-ins. 29–32

POO Programação Orientada a Objectos. 11

PRT Plataforma de Execução. 30

RCP Rich Client Platform. 31

SDK Software Development Kits. 11

SW Software. 7, 8, 16, 21, 31, 40, 41, 44

SWT Standard Widget Toolkit. 31

UC Universidade de Coimbra. 22

VFC Variabilidade da Frequência Cardíaca. 8, 17

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Falhas de software (ou seja, bugs) permanecem como um dos problemas mais persistentes da qualidade do software. Após décadas de intensa pesquisa em engenharia de software e confiabilidade de software, a "média da indústria é de 15 a 50 erros por 1.000 linhas de códigos (KLoC) entregue" [22]. Mesmo quando o software é desenvolvido usando processos altamente maduros, o código implantado ainda possui alta densidade de erros residuais, de 2 a 5 erros por KLoC [23, 29]. Nos últimos anos, esse problema tem piorado, não apenas devido à pressão constante para reduzir o tempo de colocação no mercado e o custo do software, mas também porque o tamanho do código aumentou drasticamente. Quanto mais o código cresce evidentemente, maior é a probabilidade dos erros também crescerem. A realidade é que não se consegue saber exactamente a localização dos erros no código-fonte, quando eles se revelarão e, acima de tudo, as consequências da sua activação, senão pela aplicação de processos de verificação e validação de software com custos altos e bastante falíveis. É visível o aumento da utilização do software em quase todos os sectores, logo a má qualidade do código devido a erros residuais representa um dos desafios técnicos mais duradouros e difíceis nas áreas de Engenharia de Software (ES) e de Confiabilidade de Software (CS).

Embora os estudos de campo que analisam a natureza dos bugs reais não sejam abundantes, os relatórios existentes [5, 9] sobre bugs encontrados no software implantado em projetos de código aberto [9] e em grandes sistemas da International Business Machines Corporation (IBM) [5] mostram que, apesar de estarmos perante metodologias de desenvolvimento radicalmente diferentes e de cultura técnicas bem diferentes, os tipos de defeitos de software classificados usando a Classificação de Defeitos Ortogonais (ODC) nesses dois estudos são semelhantes [9]. O elemento comum é, obviamente, o factor humano, sugerindo que os humanos tendem a errar de maneiras semelhantes e a originar um conjunto limitado de tipos de falhas de software. De facto, os estudos de campo [1, 9] sugerem uma "normalização"dos tipos de erros devido ao elemento humano e identificam N tipos de bugs mais comuns, mostrando que apenas 18 tipos de erros (ou seja, tipos de construção de código) são responsável por 67,6% de todos os bugs encontrados em produtos reais (observe que "mesmo tipo de bug" não significa "mesmo bug").

Essa evidência de que os programadores tendem a falhar de maneira semelhante, de acordo com um pequeno número de tipos de erros, sugere a possibilidade de monitorizar os programadores para identificar as condições cognitivas dos programadores que podem precipitar erros de codificação ou erros que escapam à atenção humana[26].

O projecto propõe a construção de um protótipo que realiza anotação do código-fonte com base nos dados de biofeedback dos programadores, a fim de marcar (*highlighting*) linhas de

código ou tokens com informações relacionadas com a carga cognitiva dos programadores, incluindo estados como esforço mental, nível de stress, nível de atenção ou fadiga mental. O pressuposto é que essas anotações de código com dados sobre a carga cognitiva dos programadores podem ser usadas para identificar condições potenciais que podem levar os programadores a cometer erros ou fazer com que os erros escapem da atenção humana. Chamamos essa abordagem de metodologia da Engenharia de Software Aumentada de Biofeedback (BASE).

1.1 Enquadramento

Códigos mal concebidos exigem um esforço maior durante os testes e validações. A implantação de um sistema mal produzido resulta em desastres cuja dimensão depende da área aplicada.

As falhas nos sistemas informáticos normalmente são involuntárias que passam despercebidas de toda equipa de desenvolvimento e encontram-se directamente ligados com o estado cognitivo e emocional do programador. Estudos na área da Psicologia mostram que o homem é um ser instável a nível das emoções, pensamentos e atitude. Esta instabilidade afecta quase sempre a decisão, especificamente na sua produção laboral. Por esta razão, a qualidade do produto[1] de um programador (i.e., o código) varia, dependendo do seu estado cognitivo e emocional[2]. O programador com o nível anormal de stress (elevado ou baixo), tem a tendência de se irritar ou adormecer com facilidade, aumentando a probabilidade de produzir códigos com defeitos[4].

A engenharia de software possui várias técnicas de identificação de erros. Tais técnicas (conhecidas como técnicas de testes e validações), normalmente são aplicadas de forma estática[3] (*off-line*) por equipas dedicadas e envolvem custos[4] elevados. Encontrar soluções que minimizem os erros cometidos pelos programadores e "testers" ou inspectores de código, tem sido um grande desafio para a ES, uma vez que os erros cometidos ocorrem por diferentes factores e, na sua maioria, estão sempre relacionados com a intervenção humana[5].

1.2 Principais problemas no desenvolvimento de software

O sucesso no desenvolvimento de um software depende igualmente do conhecimento que se tem sobre a natureza do problema, da boa comunicação com as partes interessadas (também conhecidos como *stakeholder*), do orçamento[6] disponível, de pessoal qualificado, de um ambiente saudável com os colegas de equipa. Na ausência ou na carência de um destes factores, a probabilidade do produto sair com defeito é maior. Por exemplo, quando o orçamento é curto e a complexidade do projecto é alta a equipa de desenvolvimento, trabalha sob pressão com a intenção de cumprir com o cronograma. Esta pressão, assim como os outros factores que surgem com a ausência ou carência de uma das condições mencionadas acima, podem causar os seguintes problemas [14]:

[1] Entende-se aqui como produto ou artefacto o código resultante de um esforço humano, aplicado a programação com o computador que seja capaz de solucionar um determinado problema. Por exemplo Um software ou um algoritmo.

[2] Existem outros factores que causam a baixa qualidade do produto, como por exemplo a experiência do programador, a complexidade do problema e outros.

[3] No sentido de ser um código ou programa finalizado.

[4] Os custos podem ser: Tempo, valores financeiros ou esforços.

[5] Não foi levado em conta as metas heurísticas capazes de desenvolverem softwares.

[6] Orçamento no ponto de vista do tempo estimado e acordado para a desenvolvimento do produto.

- Insatisfação dos clientes e utilizadores;
- Má reputação;
- Cronogramas comprometidos;
- Custos elevados na manutenção;
- Incumprimento dos requisitos funcionais;
- Conflitos e Ambiguidade entre os requisitos;
- E altos custos financeiros com as equipas montadas para localizar as falhas.

Outros problemas que não foram mencionados acima podem ser controlados por meio da combinação de técnicas das ciências do ramo da Engenharia Informática (EI) como as disciplinas de ES, Gestão de Projecto (GP), Engenharia de Requisitos (ER). Embora sejam controlados ou até mesmo resolvidos, o problema da baixa produtividade, o custo financeiro pago pelo tempo longo que as equipas de inspecção e de teste consomem para localizar os defeitos e a possibilidade de adição de mais defeitos ou indicação de defeitos falsamente (os falsos positivos encontrados) na fase da inspecção de código, tende a persistir.

Por outro lado, o processo de desenvolvimento de um software envolve várias fases (como concepção, análise e desenho, implementação, testes e implantação) feitas por humanos, o que significa que podem conter defeitos pelas razões anteriormente mencionadas. Nesta tese, nos focamos apenas nas fases do desenho (codificação), teste e inspecção, visto que podem ser realizadas num Ambiente de Desenvolvimento Integrado (IDE).

1.3 Motivação

Melhorar a qualidade do software sem a necessidade do aumento dos custos e esforços, representa a nossa principal motivação. Para isso sugerimos a introdução de informação sobre a carga cognitiva do programador, que vai ser associada ao código sob a forma de anotações nas IDE e a aplicação das técnicas de Inteligência Artificial (IA) para processar a informação das anotações e outros aspectos como a complexidade do código-fonte, para estimar risco de conter bugs num determinado segmento do software.

Os riscos e as anotações podem ser usadas para apoiar o programador não só durante a codificação mas também para a optimização os testes. Resumidamente a motivação foca-se em:

- Reduzir os custos na produção do "software";
- Aumentar a produtividade;
- Diminuir o esforço e o valor financeiro aplicado na correcção de erros;
- Controlar a tensão ou "stress" dos programadores durante o desenvolvimento de software.

Por outro lado, os *Stakeholders* na maior das vezes necessitam de respostas rápidas e eficientes; uma combinação muito difícil de garantir, uma vez que se encontram em sentidos opostos no que se refere à qualidade do código.

1.3.1 Custo no desenvolvimento do software

A boa gestão de projetos recomenda fortemente a identificação de riscos no desenvolvimento de software, acompanhada com o plano de mitigação de acordo com a sua gravidade. É verdade que com a introdução das metodologias ágeis os projectos tornaram-se mais dinâmicos[7], o que altera o plano e o custo de desenvolvimento.

Os custos no desenvolvimento de software, muitas vezes não estão relacionados apenas com o tempo, valores financeiros pagos e o esforço. Muito além disto, envolve a reputação, a perda de emprego do gestor do mais alto escalão[8], fechos de instituições, perigos para as cidades e nações [3].

A título de exemplo, o estudo feito pela revista da *Harvard Business Review* revelou problemas nos sistemas de informação do aeroporto de *Hong Kong*, tais como as falhas no sistema de informações dos voos e nos bancos de dados para o acompanhamento das cargas, acarretando custos à economia local num total de (USD) 600$ milhões desde 1998 a 1999[3].

[7]No sentido que os requisitos podem mudar ou surgir novos e os testes são realizados mais cedo, enquanto ainda outras partes do projecto está em desenvolvimento

[8]Um exemplo da perda de emprego, aconteceu com o CEO da EADS, Noel Forgeard

1.3.2 Perda da produtividade

Quando conseguimos produzir menos do que podemos e se este processo é constante, então estamos diante de uma perda ou baixa de produtividade. A produtividade não é um elemento singular, envolve vários factores. Certamente que um dos factores directo é a ferramenta de produção, que quando se encontra com debilidades, a produção baixa e o produto pode sair sem qualidade ou com defeitos, o que nos remete para os problemas apresentados na secção anterior 1.3.1.

O contrário também chega a ser verdade. Se conseguirmos produzir mais do que anterior ou com maior qualidade ou ainda com menos esforço, e se o processo for constante estamos diante do aumento da produtividade. Pressupõe-se que diante de uma alta carga cognitiva, stress ou cansaço mental, o software produzido levará mais tempo ou possuirá mais defeito. O que resulta em entregas atrasadas, frutos do maior tempo consumido nas fases de teste e da recodificação.

1.3.3 O Stress e suas consequências

O homem enfrenta vários problemas, quer seja pelos vários papéis que representa na sociedade como pelas as influências e as pressões que recebe dela. Uma possível causa tem sido o stress que actualmente quase toda humanidade sente ou de forma directa ou indirecta. O stress não é um tema novo, foi no início do século XX que estudiosos das ciências biológicas e sociais iniciaram a investigação dos seus efeitos na saúde física e mental das pessoas [13].

Figura 1.1: Reacções do stress

A figura 1.1 ilustra as reacções (mentais, físicas e emocionais) que o stress pode proporcionar no individuo, sendo que as reações se estão interligadas e umas são causas de outras. Estas reacções são accionadas muitas vezes (no ponto de vista de um programador) pelas disputas ou concorrências, prazos curtos, trabalho longo e pelas relações mal geridas dentro das equipes.

Pretende-se com este tópico mostrar que as reacções do stress influenciam na qualidade do trabalho do programador, abrindo assim portas para injecção de defeitos no código.

Uma ferramenta que ajudaria muito o programador durante a actividade de programação ou o líder de desenvolvimento na gestão dos seus recursos (isto é, o recurso humano, a equipa de desenvolvimento), talvez seja um IDE que notifique, aconselha e recomenda repouso tão logo depois de identificar factores de stress ou alta carga cognitiva.

1.4 Objectivos

Nesta secção é apresentado o objectivo geral do trabalho de mestrado e as tarefas definidas para a persecução desse objectivo.

1.4.1 Objectivo Geral

O objectivo geral do presente trabalho é desenvolver a primeira versão do protótipo do IDE Eclipse, para integrar os dados de biofeedback dos programadores e das operações sobre o código em tempo real, com a finalidade de fazer anotação no código-fonte que representem o estado cognitivo dos programadores enquanto manipularem o código.

1.4.2 Objectivos Específicos

A seguir é apresentada a lista dos objectivos específicos:

1. Analisar a literatura existente relacionada ao problema;
2. Entender a arquitectura da IDE Eclipse;
3. Adaptar o plugin *Fluorite* para a capturar os logs das operações feitas pelo programador periodicamente;
4. Desenvolver as Interface de Programação de Aplicações (API) para a importação dos dados enviados pelos módulos de recolha e extracção dos biofeedbacks;
5. Desenvolver o plugin Eclipse *Biofeedback-UC* para a anotação de código;
6. Modelar os classificadores/regras para determinar a probabilidade (que também chamamos de "risco") de haver defeitos num token com base nos dados das operações, complexidade do código e da carga cognitiva;
7. Incorporar os modelos/regras no plugin Eclipse *Biofeedback-UC*;
8. Adaptar o plugin Eclipse (*Biofeedback-UC*) para realizar "realce de biofeedback" do código por meio dos riscos associados aos tokens;
9. Realizar testes do sistema em condições realistas;
10. Avaliar a IDE aprimorada com a utilização do plugin *Biofeedback-UC*;
11. Escrever o relatório final da tese de mestrado.

1.5 Contribuições

A principal contribuição desta tese de mestrado reside na construção do primeiro protótipo de um IDE com anotações de biofeedback integradas, demonstrando a viabilidade da utilização da técnica em ambientes reais de desenvolvimento de software. Este protótipo é um elemento fundamental para a investigação desenvolvida e a desenvolver no projecto BASE, permitindo posteriormente avaliar as diversas possibilidades de utilização das anotações para apoio aos programadores, optimização de testes e de inspecções de código, entre outras possibilidades que estão a ser investigadas no projecto.

No âmbito do projecto base o protótipo desenvolvido vai ser usado para desenvolver e avaliar uma longa lista de funcionalidades previstas no projecto, com vista a melhorar a qualidade do código produzido e a produtividade dos programadores, que podemos resumir nos seguintes pontos:

- Anotação do código escrito que pode ser usado para orientação dos programadores durante a escrita ou inspecção de código;
- A identificação das condições[9] que podem fazer com que os programadores criem defeitos durante o desenvolvimento do código;
- Aconselhamento em tempo real (*online*) para programadores e testadores sobre o risco de ocorrência de bugs, desde o simples aviso de áreas de código do Software (SW) que podem precisar de uma segunda análise (para remover possíveis bugs) até aos cenários de suporte de programadores mais sofisticados, que consideram as anotações de biofeedback em conjunto com a complexidade do código manipulado pelo programador e seu histórico de bugs anteriores (chamamos essa programação de pares alternativos, como uma analogia à abordagem de programação de pares nos processos de desenvolvimento ágeis, mas sem a necessidade do segundo programador do par);
- Teste de software orientados por biofeedback para optimizar o esforço de teste, levando em consideração as informações individuais colectadas de cada programador que participou do desenvolvimento do código;
- Modelos aprimorados de estimativa da densidade de erros e análise de risco de SW, através do uso de informações adicionais sobre os estados emocionais e cognitivos do programador, em conjunto com métricas de complexidade de código e cobertura de teste;
- Ambientes de desenvolvimento integrados amigáveis aos programadores, com aviso ou aplicação automática dos momentos de descanso dos programadores, quando sinais acumulados de fadiga e tensão mental mostram que não apenas a qualidade do código é duvidosa, mas, sobretudo, o bem-estar mental dos programadores deve ser protegido;
- Necessidades de treinamento optimizadas para biofeedback através da criação de perfis de programadores individuais para ajudar a definir planos de treinamento com base nos meta-dados do biofeedback;
- Disponibilizar dados de anotados de código-fonte, contendo informações sobre as cargas cognitivas, a complexidade do código e o risco calculado de conter defeito no token, num formato particularmente fácil de entender e ser manipulado;

[9] As condições que podem causar defeitos são: as distracções, complexidade do problema, elevada concentração e o stress elevado.

- Criar condições para que outros plugins e API, utilizarem as anotações de código com o objectivo de estenderem as funcionalidades dos IDE ou outros fins;
- Facilitar a criação perfis de programadores que ajudarão a definir as necessidades de treino mais adequadas.

1.6 Projecto BASE

A presente dissertação está directamente vinculada ao projecto BASE (*Biofeedback Augmented Software Engeneering*) do Centro de Informática e Sistemas da Universidadede Coimbra (*CISUC*), representando uma parte importante dos resultados objectivos do projecto.

O projecto BASE tem como objectivo principal pesquisar falhas de SW na perspectiva da neuro-ciência, para encontrar os mecanismos cerebrais envolvidos na criação de erros de SW e nas manifestações psico-fisiológicas envolvidas que podem ser capturadas por dispositivos portáteis compatíveis com ambientes típicos de desenvolvimento de SW, no intuito de aprimorar os paradigmas de desenvolvimento através da introdução de um novo elemento denominado *"biofeedback dos programadores"*.

O BASE permitirá recursos radicalmente novos, como os que se encontram apresentados na lista dos benefícios da secção 1.5.

1.7 Enquadramento do estágio ao projecto BASE

Estudos muito recentes, publicados em 2019 [25, 26, 27], realizados no âmbito do projecto BASE comprovaram que embora diferentes programadores sigam estratégias específicas de leitura e compreensão de código, a conjugação da Variabilidade da Frequência Cardíaca (VFC), pupilografia e rastreamento ocular permite a identificação de linhas de código específicas que correspondem aos picos de carga cognitiva dos programadores individuais, sugerindo assim a anotação de código no nível da linha e do token como sendo viável. Uma vez que a VFC [26] e pupilográfia [26] podem ser usadas para avaliar os estados cognitivos dos programadores (e.g., esforço mental, distração, etc.), perspetiva-se que as anotações de biofeedback possam ser usadas para identificar automaticamente zonas do código consideradas pelos desenvolvedores como mais difíceis e potencialmente mais susceptíveis de conterem bugs. A associação desses métodos com o rastreamento ocular [27] permite anotações precisas do código em desenvolvimento, no espaço[10] e no tempo[11].

Os resultados da investigação desenvolvida até ao momento no projeto BASE indicam que é possível associar informações de carga cognitiva dos programadores a linhas de código específicas, manter essas informações actualizadas através de plugins num IDE e usar essas anotações para melhorar a qualidade do código e reduzir os custos na produção de software.

A presente dissertação uma parte central da solução do projecto BASE que materializa os resultados científicos já obtidos na construção de um protótipo de um IDE com capacidade de efectuar anotação e marcação de linhas de código com base nos dados de biofeedback e fornecer suporte a programadores, testadores e inspectores de software. O protótipo desenvolvido nesta tese de mestrado não apenas demostra a viabilidade de aplicar os resultados já obtidos no BASE de forma concreta no IDE Eclipse das anotações de biofeedback

[10] Na linha do código.
[11] A tempo de fornecer avisos *online* ou ajudar o programador em tempo corrente.

mas também dota o projeto de uma ferramenta essencial para explorar as anotações e desenvolver a linha de funcionalidades descritas na secção 1.5.

1.8 Estrutura do Relatório

O relatório encontra-se estruturado por capítulos, secções e subsecções.

- O primeiro capítulo, apresenta a introdução do trabalho, a sua justificação e as motivações que nos levam a resolver este problema;
- O segundo capítulo, apresenta um enquadramento a nível dos conceitos necessários para a compreensão do tema, as contribuições feitas até ao momento, um enquadramento da tese no projecto BASE e por fim o estado da arte;
- O terceiro capítulo, apresenta as metodologias de Investigação Científica (IC) escolhidas para a realização do trabalho, a formulação dos objectivos, estratégia montada para o controlo e execução das diversas fases do projecto e termina com a apresentação dos riscos e os seus planos de mitigação;
- O quarto capítulo, prepara o leitor para o entendimento das tecnologias aplicadas, ou seja, combinadas para dar uma solução evolucionária. Neste capítulo é definida as opções tecnológicas e as razões da sua escolha.
- Quinto capítulo, apresenta a engenharia de software aplicada para a expansão do IDE Eclipse usando os *"Biofeedback"*.
- Sexto capítulo, dedicado para apresentação da implementação da solução proposta no IDE Eclipse, desde os protótipos de alta fidelidade, os modelos de anotação, a recolha de dados e a construção dos modelos de classificação.
- O capítulo sete, apresenta a conclusão e os trabalhos futuro;
- No final do documento encontram-se as referências bibliográficas apoiadas para a realização do trabalho e o apêndice.

Capítulo 2

Conceitos e Estado de Arte

O presente capítulo encontra-se dividido em duas (2) partes. A primeira apresenta alguns conceitos importantes que foram levados em conta para a realização do presente trabalho e igualmente útil para a sua compreensão. A segunda parte dedica-se a apresentação dos trabalhos relacionados no âmbito da identificação de bugs por meio de sinais vitais e o calculo do riscos de um trecho do código-fonte conter bugs pela utilização modelos inteligentes numa IDE.

2.1 Conceitos fundamentais

Nesta secção é apresentado e definido alguns termos (essencial para a compreensão do presente documento), técnicas e ferramentas utilizadas para a concretização deste projecto.

2.1.1 *Software Development Kits* (SDK)

É uma estrutura que garante as condições necessárias para os programadores criarem os softwares, utilizando uma plataforma ou um Sistema Operativo (OS). Ela inclui as ferramentas e documentação necessária para os programadores trabalharem [11].

O Ambiente de Desenvolvimento Integrado (IDE) Eclipse, não é instalável por meio de um código binário como outras IDEs, pelo contrário é disponibilizado um Software Development Kits (SDK) Eclipse que pode ser configurado de acordo com a sua necessidade.

2.1.2 *Application Programming Interface* (API)

Representa o conjunto de protocolos estabelecidos por uma aplicação ou software para permitir a interacção com outros sistemas externos sem a necessidade de conhecer as técnicas ou requisitos aplicados no desenvolvimento e implantação do software. Funciona como os objectos que se podem instanciar nas linguagem de programação que usam o paradigma orientado a objectos[1].

[1]O paradigma Orientado a Objecto usa as linguagens de Programação Orientada a Objectos (POO). O plugin desenvolvido neste projecto usou a linguagem Java que é orientada a Objectos.

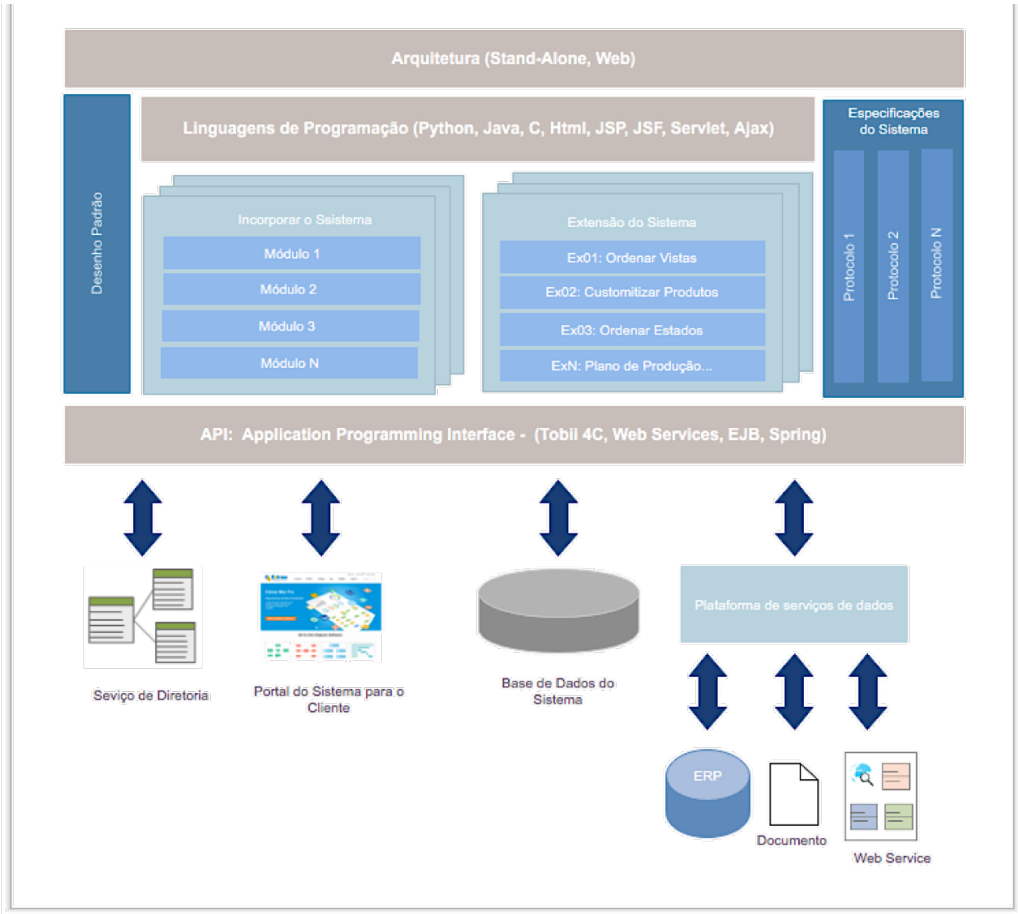


Figura 2.1: Exemplo da arquitectura de um Software e a sua API

2.1.3 *Integrated Development Environment* (IDE)

De todas as ferramentas utilizadas nas construções de *Softwares*, as IDE são as mais utilizadas. O seu surgimento acelerou em muito a produção das soluções informáticas, quer seja pela facilidade que é proporcionada ao programador na criação de códigos por meio duma Linguagem de Alto Nível[2], como pela rápida tradução de código escrito em alto nível para a versão do código da Linguagem Máquina[3], por possuírem internamente os compiladores e interpretadores. Exemplos de algumas IDE populares são: *Eclipse*, *NetBeans*, *IntelliJ IDEA*, *Blue J*, *JCreator*.

O `IDE Eclipse` foi o escolhido nesta tese de mestrado para ser expandida as suas funcionalidades. As razões que definem a sua escolha, as suas funcionalidades, bem com arquitectura é apresentada nas secções 2.1.4 e 4.1.

[2]Linguagem de Alto Nível é uma linguagem de programação mais próxima da linguagem dos humanos.
[3]Linguagem Maquina: Também conhecida como linguagem de Baixo Nível, é aquela que as máquinas, no caso o computador conhece.

2.1.4 *Plugin*

`Plugin` é um subprograma que permite a interacção de um programa com outros programas externos, estender as suas funcionalidades, assim como adicionar novas funcionalidades.

As principais vantagens do IDE Eclipse são: A facilidade de utilização, o acesso grátis, possui uma grande comunidade e é composto por plugins. A propriedade de ser constituído por plugins permite que as suas funcionalidade sejam estendidas e consequentemente o seu crescimento aumenta progressivamente. Por outro lado, embora o seu crescimento é alto, a sua utilização pode ser feito em partes pequenas.

2.1.5 Sincronização

O dicionário de Cambrige define sincronização com sendo *"o ato de garantir que relógios apresentam exactamente o mesmo tempo, ou o ato de fazer as coisas acontecerem ao mesmo tempo"* [4].

No campo das ciências da computação o processo de sincronização é amplo e pode ser aplicado ou entendido de diversas maneiras. Por exemplo, a sincronização pode ser aplicada sobre processos, tarefas, dados e ficheiros. Pode ser feita por meio de algoritmos próprios como, o algoritmos de Sincronização Mutuamente Exclusiva, Concorrente, Paralelo, Semáforo, SpinLock e por Barreiras.

Neste trabalho nos focamos apenas na sincronização de dados que e definido como sendo "a técnica ou processo comummente utilizado para estabelecer a consistência dos dados provenientes de uma ou várias fontes de dados diferentes".

O diagrama 6.1, ilustra as três fontes de dados que utilizamos, cujos os dados necessitavam de ser sincronizados. O método de sincronização escolhido foi o da *União*, com objectivo de interpolar linearmente os dados (a secção 6.2 explica como é feito o processo de sincronização).

2.1.6 Aprendizagem Máquina (Machine Learning)

Esta secção é dedicada a introdução de alguns conceitos básicos de aprendizagem máquina considerados para a realização do presente trabalho.

A aprendizagem máquina é uma área da Inteligência Artificial (IA) com o foco em construções de modelos inteligentes capazes de dotar uma máquina com a capacidade de aprender por conta própria (isto é, sem as instruções convencionais que indicam como fazer) a partir de um conjunto de dados (esse conjunto de dados é conhecido como Dados de *Dados de Treino*).

A definição clássica e mais utilizada sobre aprendizagem máquina foi dada por *Tom Mitchell* que diz: *"Diz-se que um programa de computador aprende com a experiência E com relação a alguma classe de tarefas T e com a medida de desempenho P, se seu desempenho nas tarefas em T, medido por P, se o sistema melhorar de forma confiável com a experiência E"* [8].

As duas formas fundamentais de construção dos modelos inteligentes são:

[4] `https://dictionary.cambridge.org/dictionary/english/synchronization` - Acessado 05-02-2020.

1. Criar os algoritmos com fórmulas matemáticas e/ou estatística por meio de programação;
2. Usar algoritmos prontos por meio de ferramentas e Interface de Programação de Aplicações (API).

A primeira forma exige um tempo e esforço maior que a segunda, mas possui a vantagem do desenvolvedor conhecer a fundo o modelo (como este chega a certas conclusões). Na segunda, obviamente, é o contrário e conta com a vantagem de o desenvolvedor participar nas comunidades onde pode recolher várias opiniões, sem esquecer a aceleração no processo de construção do modelo garantida pelo uso das ferramentas previamente prontas.

Dados em Aprendizagem Máquina

Os dados são muito importantes em aprendizagem máquina. É a partir dos dados que se realiza a descoberta do conhecimento (quando aplicados as técnicas apropriadas como, por exemplo, as técnicas de mineração de dados), onde o modelo depois de várias iterações (também conhecido como épocas) ajusta-se ou aproxima-se do conhecimento da classe verdadeira. Quando assim acontece dizemos que o modelo aprendeu e é inteligente.

Os algoritmos de Aprendizagem Máquina (Machine Learning) (ML) executam uma grande quantidade de dados (também conhecidos como Big Data) durante a fase de treino e teste, isto é, antes mesmo de serem aplicados a novos dados no domínio do problema (previsão, classificação, decisão). A quantidade e a qualidade dos dados brutos, ou seja, a boa cobertura do domínio do problema, determina a qualidade final do sistema de ML [8].

É muito comum dividir-se os dados brutos em duas ou três partes. A divisão por duas partes recomenda-se usar 70% para o treino do modelo e 30% para teste e validação. Quando dividida em três, o critério custa ser 50% para o treino, 30% para testes e 20% para validação. Este último critério é recomendado apenas para os casos em que os dados são suficientemente grandes.

Como vimos, os dados têm grande influência na qualidade do modelo. Isso quer dizer que devemos ter alguns cuidados antes de os usar. Exemplos de cuidados a ter com os dados, dependendo dos tipos de Algoritmos de aprendizagem, são:

- Balanceamento: quando os dados estiverem enviesados o modelo pode ser bom na classificação de um terminada classe e péssimo nas outras. Pode ainda criar do sobre ajuste (*Overfitting*);
- Tamanho: poucos dados significa treino insuficiente, o que causa conclusões pouco credíveis. Dependendo do problema e do tipo de algoritmo recomenda-se uma amostra acima de 500 registos diferentes, quanto maior melhor;
- Valores: devem estar em conformidades quanto ao tipo para cada classe de todos os registos. O processos aplicados são a normalização, eliminação ou substituição dos valores;
- Limpeza: valores com significados não calculáveis deve ser removidos ou ajustados. E têm de ser estar limpos (i.e., sem ruído, valores espúrios, etc.);

O *Pré-processamento dos Dados* é uma fase importante na construção de modelos inteligentes e tem o objectivo garantir a qualidade ou tratar os dados. Os quatros passos essenciais na fase do pré-processamento dos dados são [17]:

1. Limpeza dos dados;
2. Junções ou sincronização dos dados;
3. Transformação dos dados;
4. E a Redução ou filtragem dos dados;

Estes passos e algumas técnicas encontram-se igualmente ilustradas na figura 2.2.

Figura 2.2: Técnicas de tratamento dos dados.

Tipos de Aprendizagem Máquina

A aprendizagem máquina agrupam-se em três tipos:

- **Aprendizagem Supervisionada**: Neste tipo de aprendizagem, nos dados do treino é acompanhado com a saída desejada, ou seja, dado um conjunto de valores das variáveis independentes conhece-se (normalmente feito com a intervenção humana) o valor da variável dependente (alvo). Algumas técnicas mais conhecidas neste tipo de aprendizagem são: *Decision Trees*, *Naive Bayes*, *Linear Regression*, *Logistic Regression*, *Nearest k-neighbors*, *Artificial Neural Networks* e *Support Vector Machine*.
- **Aprendizagem não Supervisionada**: É não supervisionada pelo facto do valor da saída desejada ser desconhecida. O modelo aprende somente com os dados. Pode fazer isso através da criação de valores semelhantes o que permitirá realizar as classificações futuras. Algumas técnicas que usam este tipo de aprendizagem são: *Supported Vector Machine*, *Artificial Neural Networks*, *Restricted Boltzmann Machines*, *Expectation- Maximization*, *K-medium Clustering*, *Hierarchical Clustering*, *Principal Component Analysis*, *Isolating Forests* e *Self-Organizing Maps*.

- **Aprendizagem por Reforço**: Não existe uma técnica especifica para este tipo de aprendizagem. A ideia central gira em torno de uma "recompensa" quando a máquina escolhe um valor no caminho da solução ou uma "punição" caso contrário. O estágio mais elevado deste tipo de aprendizagem é quando a máquina tem de decidir escolher a melhor "recompensa" entre duas ou várias que possuem valores diferentes.

2.2 Estado da Arte: Erros dos programadores e estados cognitivos

Actualmente nos IDE's encontram-se incorporados mecanismos de identificação de erros, mas apenas para os erros de *"Sintaxe"* e os erros de *"Compilação"*. Apesar disto, e de muitos anos de experiência acumulada em programação de software, do grande conhecimento sobre metodologias de treino e desenvolvimento por parte dos programadores, as falhas de software continuam aumentando, especialmente porque o software cresceu em complexidade e tamanho[5]. Estas falhas são muito preocupantes, uma vez que, o programador e as equipas de testes e de inspecção encontram muitas dificuldades na sua identificação, o que muitas vezes leva a adicionar outros bugs na tentativa de encontrar os bugs existentes no código.

Estudos de campo sobre bugs encontrados nos softwares implantados, cobrindo diferentes projetos, tipos de sistemas e construídos de acordo com diferentes metodologias, mostraram que os tipos de bugs encontrados são semelhantes [24, 28, 32] e que a maioria dos bugs seguem um número relativamente pequeno de tipos de falhas de software. Essas descobertas mostram que os desenvolvedores de software (especialmente os programadores) tendem a errar de maneira semelhante, independentemente da metodologia de software, linguagem de programação e ambiente de desenvolvimento (IDE), sugerindo que um número finito (e não muito grande) de mecanismos de erro humano está na origem da maioria dos bugs de software.

Com esta ideia, a área de pesquisa de defesa de falhas de software, baseada em modelos e mecanismos cognitivos de erros humanos, emergiu e ganhou terreno nos últimos anos. Esta está enraizada nas teorias e modelos de erro humano [28] e geralmente adapta modelos e taxonomias de erro humano desenvolvidos na psicologia cognitiva ao processo de desenvolvimento de software [1, 16], com o objectivo de definir estratégias defensivas ou de melhorar etapas específicas, como o levantamento de requisitos [32].

Um caminho de pesquisa diferente que também surgiu nos últimos anos é o estudo dos mecanismos cerebrais humanos por trás da compreensão do código de software usando abordagens neuro-científicas, particularmente usando equipamentos pesados como ressonância magnética funcional (fMRI), espectroscopia de infravermelho próximo ao campo (fNIRS) e eletroencefalografia (EEG). Segundo *J. Siegmund N. Peite* [24], identifica a memória de trabalho, a atenção e processamento de linguagem como sendo as regiões do cérebro envolvidas na compreensão de código e na identificação de erros de sintaxe. Outro estudo feito por *T. Santander B. Floyd* e *W. Weimer* [2] comparou o código e a compreensão do texto em linguagem natural para identificar os mecanismos cerebrais envolvidos em cada actividade, já *T. Nakagawa,* e *Y. Kamei.* [31] estudaram a execução mental do código-fonte por programadores.

Outros três estudos muito mais recentes (2019) realizados no âmbito do projecto Engenha-

[5]Por exemplo o Software (SW) de um carro moderno e sofisticado tem mais de 100 milhões de linhas de código

ria de Software Aumentada de Biofeedback (BASE) trouxeram as seguintes contribuições:

O *primeiro estudo* [26] comprovou que o esforço mental dos programadores em tarefas de compreensão de código pode ser verificado através da Variabilidade da Frequência Cardíaca (VFC) usando dispositivos portáteis não intrusivos. Os resultados encontrados mostraram que é possível medir o esforço mental dos programadores com alta confiança, enquanto os programadores lêem e compreendem diferentes unidades de código, por meio do monitoramento em tempo real dos sinais biológicos, como VFC. O estudo mostrou também a relação entre o esforço mental medido usando VFC e as métricas de complexidade de software dos diferentes trechos de código e o esforço mental subjectivo percebido pelos programadores usando o NASA-TLX[6]. Apenas com estes resultados o BASE já propunha uma abordagem radical de neurociência que era a introdução do *biofeedback* no desenvolvimento de software.

O *Segundo estudo* [25], esteve focado na avaliação da possibilidade de usar a pupilografia[7] (ou seja, as rápidas mudanças no tamanho da pupila) como um indicador do esforço mental dos programadores e da sobrecarga cognitiva, que está relacionada a alguns modos predominantes de erro cognitivo humano. Os resultados experimentais mostraram um mapeamento claro entre o esforço mental medido usando pupilografia e as métricas de complexidade de software dos diferentes trechos de código e o esforço mental subjectivo percebido pelos programadores usando o *NASA-TLX*. Assim, com este estudo, ficou provado que a pupilografia é uma abordagem eficaz para medir o esforço mental e a carga cognitiva dos programadores em tarefas de compreensão de código, como em cenários de inspecção de código, bem como durante actividades gerais de desenvolvimento de código. Por esta abordagem não ser intrusiva do ponto de vista físico e poder ser facilmente adaptado aos ambientes actuais de desenvolvimento de software o estudo á sugere como um mecanismo de *biofeedback* para um futuro próximo.

O *Terceiro estudo* [27], mostrou a viabilidade do uso do rastreamento ocular, juntamente com a VFC e pupilografia, como o conjunto básico para a recolha de dados de forma não invasivas de biofeedback dos programadores para permitiram a identificação precisa das linhas de código (e até tokens lexicais dentro de linhas de código) que correspondem a picos da carga cognitiva dos programadores (e possivelmente outros estados emocionais, como esforço mental, nível de stress, nível de atenção, fadiga mental). A ideia central é que essas anotações de código com dados sobre a carga cognitiva dos programadores podem ser usadas para identificar condições potenciais que podem levar os programadores a cometer erros ou fazer com que os erros escapem da atenção humana.

A presente tese de mestrado desenvolve um protótipo de um IDE Eclipse modificado para poder anotar código com informação sobre o estado cognitivo dos programadores usando precisamente os resultados de investigação destes três artigos [25, 26, 27]. O sistema desenvolvido definiu a arquitetura de plugins, APIs e de processos para inserir anotações de biofeedback no código e as manter atualizadas à medida que o programador executa as operações normais de edição (inserir linhas de código, corrigir, apagar), bem como as estruturas de dados para receber as anotações. A definição da estrutura de dados para as anotações permite também compatibilizar o sistema com a possibilidade de utilização de diferentes sensores (para HRV, pupilografia e rastreamento ocular), pois a informação sobre biofeedback será sempre registada no mesmo formato, qualquer que seja o equipamento sensorial utilizado.

[6] *NASA-TLX*: É uma ferramenta administrado pela *NASA* utilizada para classificar a carga de trabalho e avaliar a eficácia de uma tarefa, sistema, equipe ou outros aspectos do desempenho.

[7] A grande vantagem do uso da pupilografia é por ser um método não intrusivo totalmente compatível com os ambientes tradicionais de desenvolvimento de software.

Capítulo 3

Metodologia e Gestão do Projecto

No Capítulo 1 foi apresenta o objectivo, o problema e as suas implicações mas de uma forma generalizada. Neste Capítulo, o problema é delimitado e detalhado por fases. Ainda, é igualmente apresentado a formulação dos objectivos específicos, as hipóteses, o desenho metodológico e um cronograma com as etapas necessárias para o alcance do objectivo.

3.1 Desenho Metodológico

O trabalho abrange várias áreas do saber, por este facto foi utilizado vários métodos adaptados de acordo as suas necessidade. Os critérios levado em consideração para a escolha dos métodos foram: A natureza das variáveis; os objectivos e o grau do problema; a amplitude e profundidade e finalmente o controlo [7].

Os métodos que se destacam são o da Pesquisa documental[1] e o da Pesquisa experimental; sabendo que na segunda parte é realizado uma experiência sobre um ambientes o mais controlados possível.

Quanto as técnicas, uma vez que a primeira parte esteve centrada não apenas no estudo da viabilidade como também a analise documental; fez se o uso do método da Pesquisa Bibliográfica o que permitiu a construção de ideias concretas[2], já na segunda parte, propõem-se o uso da pesquisa de campo e experimental.

[1] Pelo facto desta tese, se focar em grande parte na investigação cientifica e a necessidade de necessário conhecer as contribuições anteriores.

[2] É importante realçar que não existe um Ambiente de Desenvolvimento Integrado (IDE) com suporte a tecnologia Biofeedback; Existe sim, é algumas técnicas que quando combinadas ajudam a construir ideias concretas.

3.2 Cronograma detalhado

Nesta secção assim como o título diz, é apresentado um cronograma detalhando, das tarefa e datas previstas para o início e o término da dissertação. Este tem como objectivo, dividir o projecto em pequenas fases de prazos aceitáveis com a finalidade de guiar o trabalho dentro do recurso tempo disponível. As actividades estimadas (firmando-se na experiência pessoal e no esforço que se espera aplicar com os recursos disponíveis) com menos tempo de duração, estiveram entre 7 a 9 dias úteis, por este facto os intervalos dos dias foram estimados para 15 dias. As actividades acima destes devem ocupar dois ou mais períodos para a sua realização.

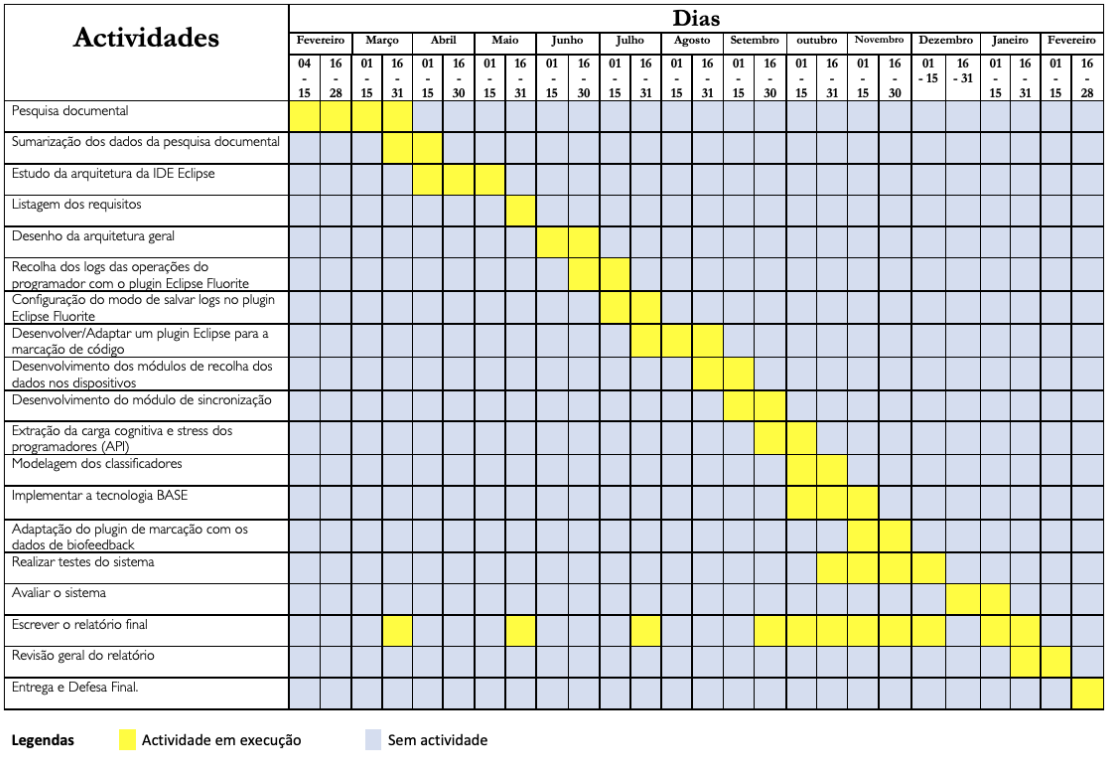


Figura 3.1: Cronograma detalhado

O cronograma segue o formato habitual de um diagrama de Gantt, devendo ser lido da seguinte forma:

- A primeira coluna da tabela representa as tarefas ou actividade que serão levadas a cabo para a conclusão do projecto;
- A Segunda linha indica o tempo em meses que uma dada actividade pode levar;
- A terceira linha indica o tempo em dias com um intervalo de 15 dias úteis, com excepção do mês de Fevereiro por ser curto;
- Os "quadrados" com a cor amarela representam a actividade que corresponde à linha, que será realizada na data correspondente à coluna;

3.2.1 Primeiro semestre

Os primeiros semestre serviu principalmente para pesquisa de trabalhos relacionados e da arquitetura e funcionamento do IDE Eclipse. Serviu ainda para o estudo do funcionamento do *plugin Fluorite* e o desenvolvimento de um *plugin* modelo capaz de realçar código. Embora tivesse sido realizadas estas tarefas, os resultados eram apenas superficiais.

Os requisitos, a arquitetura preliminar do sistema e a estratégia de identificação dos *bugs* (apresentado na figura 5.1) começaram a ficar claros apenas perto do fim do semestre.

3.2.2 Segundo semestre

No primeiro semestre, identificamos os requisitos iniciais do sistema, as estratégias de identificação dos *bugs*, a arquitetura preliminar do Software (SW). Cultivamos ainda, á rotina de reunir as terças-feiras durante 30 á 45 minutos para discutirmos o avanço do trabalho e definirmos novas tarefas. No segundo semestre, decidimos usar uma metodologia de desenvolvimento de SW que se aproxima do processo que utilizamos (i.e. reunirmos periodicamente, selecionarmos algumas funcionalidades da lista das funcionalidade inicialmente identificadas, revisarmos e testar a parte feita e integrar nas restantes partes concluídas). *Scrum* foi a metodologia escolhida, embora no que toca ao tamanho da equipa não se recomendo para o nosso caso.

Uma vez conhecida a metodologia, definiu-se o *product backlog* (apresentado na tabela 5.1) que guiou o processo de construção do protótipo (ressalvamos que *product backlog* apresentado na tabela 5.1 não possui a lista todas funcionalidades). O *product backlog* foi devidido em *sprints* com duração de 3 á 4 semanas. O *sprint* no estado pronto (ou acabado de ser desenvolvido) passava para o teste independente e colectivo para avaliar o funcionamento de protótipo e o seu impacto na performance do IDE.

3.3 Ciclo de vida do projeto

O ciclo de vida real do projeto foi a iterativo incremental com o uso da metodologia agíl *Scrum* (ilustrado na figura 3.2). A escolha desta metodologia deve-se ao facto de a equipa ser relativamente pequena (apenas um estudante), de os requisitos serem mutáveis (sofriam muitas mudanças), de o período de tempo para produzir o protótipo ser curto e de exigir uma carga relativamente leve na documentação.

A aplicação desta metodologia permitiu-nos trabalhar por módulos que correspondem a *Sprints* (que é uma série de funcionalidades a serem implementadas e testadas), dividindo facilmente o trabalho em pequenos períodos de 2 a 3 semanas. Ou seja, a cada 2 semanas produzia-se um incremento de funcionalidades para o protótipo.

Quanto às funções, o estudante teve de ocupar os papéis de *Desenvolvedor*, *Analista de Requisitos* e de partilhar os papeis de *Scrum Master* e *Tester* com os Phd. Henrique Madeira e o Phd. Raul Barbosa, que também desempenharam o papel de *Project Owner*.

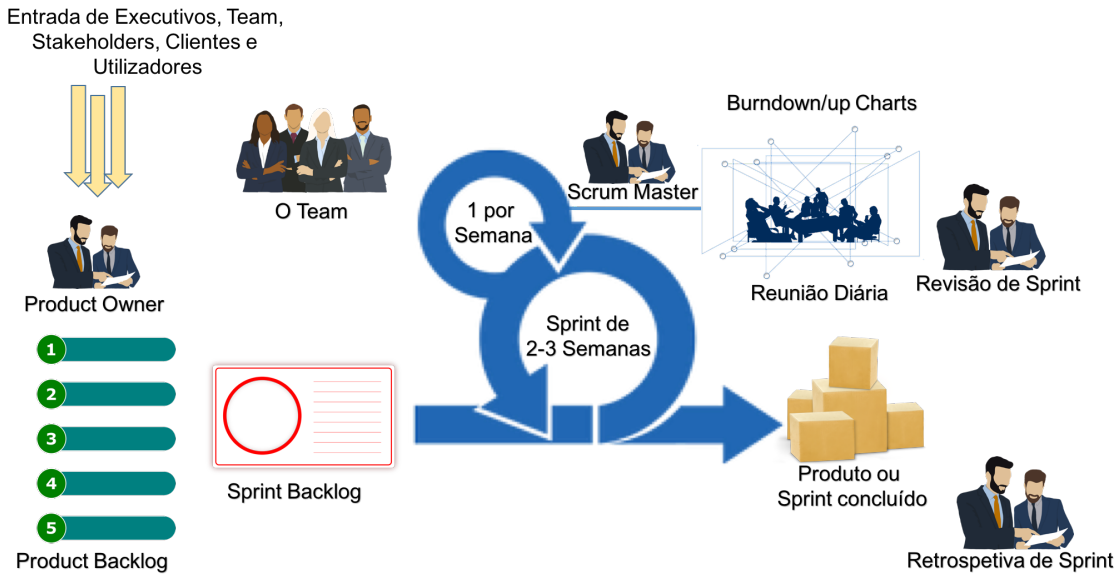


Figura 3.2: Metodologia Scrum

O cronograma acima (figura 3.1) não corresponde com o plano original seguido durante a tese. Algumas actividades duraram mais do que o tempo programado. Isso deveu-se ao facto de alguns requisitos sofreram alterações, da estimativa de tempo basear-se apenas nas experiências pessoais e do surgimento de desafios desconhecidos na fase da calendarização.

3.4 Metas de sucesso

- Implementamos todos os requisitos "obrigatórios" antes do dia 1 de Dezembro de 2019;
- O projecto é entregue em Janeiro de 2020 junto do relatório final da tese, conforme definido pelo Departamento de Engenharia Informática (DEI) da Universidade de Coimbra (UC);
- O protótipo atende os atributos de qualidade mais importantes (conforme definido na Especificação de Requisitos de Software na secção 5.3.2);
- O *IDE Eclipse* realiza a anotação e a marcação das linhas e tokens levando em conta os sinais de *biofeedback*.

3.5 Gestão dos Riscos

O controlo e a gestão dos riscos representa uma actividade de elevada importância no processo da Gestão de Projecto (GP). É nesta actividade que se identificam as ameaças ou os riscos, fruto das dependências e relações existente dentro de um projecto ou com origem externa. Os riscos quando não levados em consideração podem resultar na inviabilidade do projecto ou no incumprimento da meta estabelecida pela má qualidade que o produto final apresentará. Nesta secção, começamos por fazer a identificação dos riscos, apresentamos de seguida alguns conceitos básicos e relevantes que levamos em conta na classificação dos riscos do projecto, e terminamos com o plano de mitigação.

3.5.1 Identificação de Riscos

Após análise, foram identificados os seguintes riscos principais:

1. A pouca informação que temos sobre a arquitectura da IDE Eclipse, e as implicações na construção e integração de um plugin nela, pode levar a que o desenvolvimento de todo o protótipo demore mais do que o tempo inicialmente estimado;
2. O protótipo é construído ao mesmo tempo que a equipa Engenharia de Software Aumentada de Biofeedback (BASE) aperfeiçoar algumas metodologias. Isso pode causar problemas caso haja falhas de comunicação entre o aluno de mestrado e o resto da equipa;
3. O trabalho é feito por apenas um estudante, está implícito que executará várias funções ao mesmo tempo e também terá uma carga de trabalho escolar noutras disciplinas. Isso pode causar desgaste e, por sua vez, afectar o desenvolvimento do projecto;
4. O protótipo dependerá dos dados de biofeedback recolhidos por dispositivos e sistemas externos, isso pode afectar o desenvolvimento do projecto, causando atraso pela morosidade da sua disponibilização;
5. Os dados de biofeedback são recolhidos e disponibilizados por sistemas externos, isso pode causar dificuldades técnicas na sincronização (*timestamp* [3] comum para dados provenientes de subsistemas diferentes) difíceis de superar;
6. O protótipo usará dois plugins open source publicamente disponíveis (o *Fluorite* e o *HighlightOnSelection*), pelo que há o risco potencial de falhas no funcionamento caso no de esses plugins serem actualizados ou descontinuados no futuro.

3.5.2 Atributos dos Riscos

Níveis dos Atributos

A seguir, são apresentados os níveis dos atributos da declaração dos riscos. Neste projecto, consideraremos apenas 3 atributos:

- **Impacto:** (alto, médio, baixo);
- **Probabilidade:** (alta, média, baixa);
- **Tempo:** (longo, médio, curto).

Definição do atributo

- Impacto: efeito negativo na meta de sucesso do projecto, se o risco ocorrer;
- probabilidade de o risco ocorrer com o impacto esperado;
- Tempo: tempo de reação, desde a identificação até quando é necessário lidar com esse risco.

[3] Visto que o tempo representa o elemento principal da sincronização dos dados.

	Impacto			**Probabilidade**			**Tempo**		
Nº Risco	Baixo	Médio	Alto	Baixo	Médio	Alto	Curto	Médio	Longo
1			**X**		**X**				**X**
2		**X**		**X**					**X**
3			**X**	**X**					**X**
4			**X**			**X**		**X**	
5			**X**	**X**			**X**		
6			**X**			**X**		**X**	

Tabela 3.1: Tabela das avaliações dos riscos

Definição do nível

- Impacto
  - Alto: significa que não é possível atingir a meta de Sucesso;
  - Médio: significa que é possível alcançar a meta de sucesso, mas apenas com um grande esforço e/ou custo;
  - Baixo: significa que é possível alcançar a meta de sucesso, com uma pequena quantidade de esforço e/ou custo.
- Probabilidade
  - Alto ($>$ 70%);
  - Médio (entre 30% e 70%)
  - Baixo ($<$30%).
- Tempo
  - Longo ($>$ 3 semanas);
  - Médio (entre 1 e 3 semanas);
  - Curto ($<$1 semana).

3.5.3 Avaliação dos Riscos

Nesta secção, definimos os níveis dos atributos para cada declaração de risco. A avaliação dos riscos foi desenvolvida levando em conta as experiências pessoais e o esforço que será aplicado a cada fase do projecto, com base nos dados das pesquisas feita na fase da análise documental. A tabela 3.1 apresenta avaliação de riscos.

3.5.4 Matriz de Exposição do Risco

Nesta secção apresentamos a priorização dos riscos feita para determinar as suas prioridades. Os riscos foram ordenados de acordo com a gravidade, o que significa que os riscos com maior impacto e probabilidade são os mais críticos. Depois, determinou-se quais os riscos dignos de se fazer um plano de mitigação. Isso foi conseguido analisando a tabela anterior.

A partir da Tabela de Avaliação de Risco, foi criada uma matriz de exposição, na qual todas as declarações de risco foram distribuídas de acordo com seu impacto e probabilidade. A

partir desse gráfico, decidiu-se os riscos que seriam considerados mais críticos. Estivemos concentrados apenas nesses riscos e elaboramos um plano de mitigação.

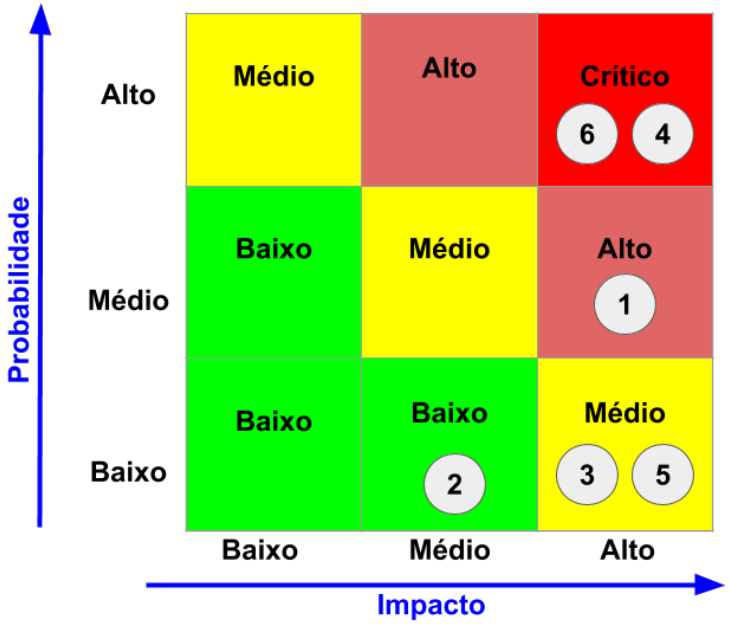


Figura 3.3: Matriz de classificação dos riscos.

3.5.5 Plano de Mitigação

Nesta secção, descrevemos as medidas e os planos elaborados para lidarmos com os riscos que foram escolhidos para serem mitigados na secção anterior. Para mitigar um risco, podemos fazer alterações nas condições dos riscos, nos resultados (impacto) do risco.

Risco 1

A pouca informação que temos sobre a arquitectura da IDE Eclipse, e as implicações na construção e integração de um plugin nela, pode levar a que o desenvolvimento de todo o protótipo demore mais do que o tempo inicialmente estimado.

- Impacto: Alto
- Probabilidade: Média
- Tempo: Longo

Para mitigar este risco, foi necessário reservar um período para a investigação e o estudo profundo sobre o IDE Eclipse.

Risco 3

O trabalho é feito por apenas um estudante, está implícito que executará várias funções ao mesmo tempo e também terá uma carga de trabalho escolar noutras disciplinas. Isso pode causar desgaste e, por sua vez, afectar o desenvolvimento do projecto.

- Impacto: Alto
- Probabilidade: Baixa
- Tempo: Longo

Para mitigar este risco, decidiu-se definir com mais precisão o conjunto mínimo de funcionalidades base do protótipo, que constituem a primeira versão do protótipo. O avanço para versões seguintes estaria condicionado ao estudante ter condições de disponibilidade para concluir novas funcionalidades entretanto identificadas (resultantes da investigação em curso). Por exemplo, a inclusão de informação sobre a complexidade do código nas anotações, e a manutenção dessa informação atualizada à medida que o programador vai editando o código (o que altera a complexidade), não fazia parte do conjunto mínimo de funcionalidades definidas inicialmente, mas foi posteriormente acrescentado, depois de as anotações de biofeedback já estarem implementadas e testadas.

É importante notar que o protótipo a construir resulta do trabalho de investigação do projeto BASE, que decorria (e ainda decorre) em simultâneo com o trabalho de mestrado. Por isso, o conjunto de funcionalidades a incluir no protótipo, para além das funcionalidades base relativas à anotação automática do código e às manutenção da informação dessas anotações atualizadas, ainda está em aberto e espera-se que novas funcionalidades venham a ser definidas. Estas novas funcionalidades serão desenvolvidas e disponibilizadas nas versões sucessivas do protótipo.

Risco 4

O protótipo dependerá dos dados de *biofeedback* recolhidos por dispositivos e sistemas externos, isso pode afectar o desenvolvimento do projecto, causando atraso pela morosidade da sua disponibilização.

- Impacto: Alto
- Probabilidade: Alta
- Tempo: Médio

Esse risco foi mitigado pela definição da estrutura dos dados e a técnica da troca dos dados, numa reunião com a equipa do BASE. Com essas informações a construção do protótipo passou a ser independente do sistema externo (apenas para a fase do criação e teste preliminares).

Risco 5

Os dados de *biofeedback* são recolhidos e disponibilizados por um sistemas externos, isso pode afectar na veracidade da informação que o protótipo dará porque o tempo (*timestamp*) podem ser diferentes.

- Impacto: Alto
- Probabilidade: Baixa
- Tempo: Curto

Este risco foi mitigado com a decisão técnica dos dispositivos usarem o protocolo de tempo de rede do Inglês *Network Time Protocol* (NTP) [4] para o ajuste e sincronização do tempo.

Risco 6

O protótipo usará dois plugins open source publicamente disponíveis (o *Fluorite* e o *HighlightOnSelection*), pelo que há o risco potencial de falhas no funcionamento caso no de esses *plugins* serem actualizados ou descontinuados no futuro.

- Impacto: Alto
- Probabilidade: Alta
- Tempo: Médio

Para mitigar este risco, recorremos ao benefício da licença *Eclipse Public Licence (EPL)* que permite a extensão dos *plugins*. O que se fez basicamente foi estudar estes *plugins* e reaproveitar as partes de interesse para construirmos um novo *plugin* totalmente independente, que passará a estar disponível com o protótipo.

[4] NTP é protocolo de sincronização dos relógios dos dispositivos. Para tal, basta apenas definir no servidor onde os dispositivos encontram-se conectados.

Capítulo 4

Opções Tecnológicas

Este capítulo apresenta as informações básicas relacionadas com a tecnologia necessária para a realização deste trabalho. Começa por apresentar o Ambiente de Desenvolvimento Integrado (IDE) seleccionado (*Eclipse*) para comportar as metodologias desenvolvidas pelo projecto *Engenharia de Software Aumentada de Biofeedback (BASE)* (ver a secção 1.6). A seguir apresenta a linguagem de programação utilizada para a construção do protótipo, segue com a apresentação dos plugins adaptados para corresponder aos objectivos do protótipo que se pretende construir e termina com a apresentação da Interface de Programação de Aplicações (API) (*WEKA*) e os conceitos de aprendizagem máquina que foram considerados neste trabalho.

4.1 *IDE Eclipse*

O *Eclipse* é uma plataforma de desenvolvimento de software produzido com a linguagem de programação *Java*. O primeiro lançamento do projecto *Eclipse* foi realizado em 7 de Novembro de 2001, pela *International Business Machines Corporation (IBM)*. Posteriormente a IBM disponibilizou pela comunidade dando assim a liberdade de utilização e extensão mas sempre dentro da licença *Eclipse Public Licence (EPL)*. Por natureza, o *Eclipse* foi desenvolvido por módulos, que na verdade representam plugins dedicados a uma tarefa específica. Os módulos que constituem a IDE são: O Kit de Ferramentas de Desenvolvimento em Java (JDT), Ambiente de Desenvolvimento de Plug-ins (PDE)[1], *Workbench*, *WorkSpace*, *Team* e o *Help* (como apresentados na figura 4.1), que serão descritos com mais pormenores nas secções seguintes.

4.1.1 A arquitectura do Eclipse

Os utilizadores dos IDE certamente estão familiarizado com os termos, executar, *"debugar"*, *"depurar"* ou *"inspecionar"*. Estes termos representam as operações mais comuns que os programadores utilizam quando pretendem realizar os primeiros teste do seu código fonte. No IDE Eclipse, estas operações são apenas uma parte do muito que esta poderosa ferramenta pode oferecer, embora as funcionalidades mais utilizadas pelos programadores são as do módulo JDT do Inglês *Java Development Toolkit* ou Kit de Ferramentas de Desenvolvimento em C (CDT) do Inglês *C Development Toolkit*, que varia de acordo a linguagem de programação utilizada, o IDE Eclipse realiza muito mais do que testes e criação de um

[1]PDE do inglês *Plug-in Development Environment*

novo software no *JDT*; nele é possível ainda efectuar a criação de *plugins* e até mesmo a construção de novos IDE's. Esta possibilidade é garantida pelo módulo denominado PDE, que também é o responsável pelo desenvolvimento dos plugins. Resumidamente, o IDE Eclipse é formada por módulo de plugins, para além do seu *Kernel (Plataforma de Execução (PRT))* do Inglês (*Platform RunTime*), que é uma pequena plataforma para gerir as execuções dos software dentro do Eclipse.

A figura 4.1 apresenta os componentes principais da plataforma Eclipse, que passamos a descrever:

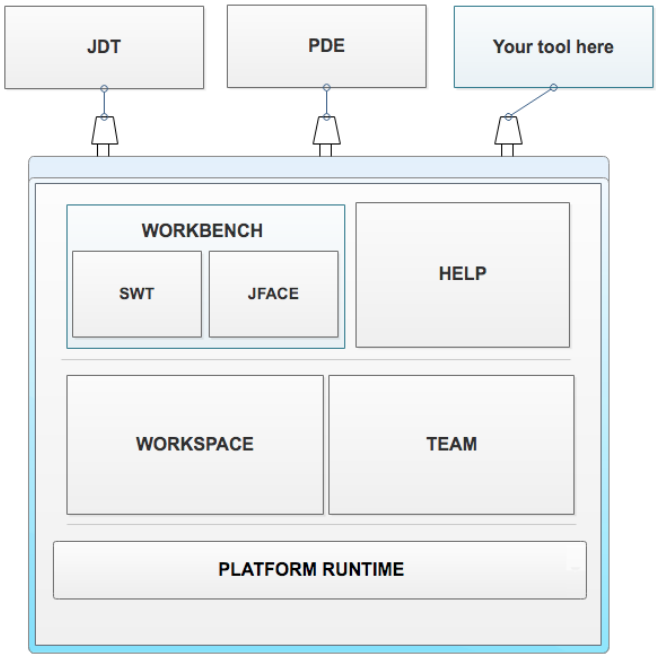


Figura 4.1: Arquitectura do IDE Eclipse [6]

Workspace

O componente *espaço de trabalho* também conhecido como "*workspace*" é responsável pela gestão dos recursos do utilizador. Os recursos são organizados em um ou mais projectos do nível superior ou na raiz. Cada projecto corresponde a um subdiretório do directório no espaço de trabalho do Eclipse. Um projecto pode conter ficheiros e pastas. Sempre que se faça uma execução de um projecto o Eclipse procura os ficheiros relacionados (ou linkados) com este projeto a partir do *workspace*, explorando assim a árvore de directórios em função do caminho indicado na importação.

O *workspace* também mantém um histórico das alterações em cada projecto, o que permite efectuar a recuperação dos estados anteriores e minimizar o risco da perda de dados. Por esta razão o *workspace* controla os estados dos ficheiros (salvo ou não salvo).

Outra responsabilidade importante deste componente é o controlo das alterações externas[2] dos projectos, o que lhe permite notificar os utilizador sobre as mudanças ocorridas [6].

[2]Alteração externa pode ser mudança do nome do diretório, subdiretório ou ficheiros. É importante não confundir as alterações externas com as internas que tem haver com a alteração do conteúdo do ficheiro. O controlo das alterações externas está sobre a responsabilidade do componente *Team* por meio dos provedores como o Sistema de Versões Concorrentes (CVS) do inglês *Concurrent Versions System*; muito útil quando se trabalho com projetos partilhado e projetos propenso a constantes alterações.

Workbench

É o componente que permite o utilizador interagir de forma gráfica com o IDE Eclipse. Disponibiliza ao utilizador menus e ferramentas e outras interfaces. Encontra-se constituído por Standard Widget Toolkit (SWT) e *JFACE*[3] que lhe atribui uma característica notável no que toca a emulação de interfaces semelhante a do Sistema Operativo (OS) nativo.

As aplicações desenvolvidas com o SWT e *JFACE* também ganham uma aparência semelhante ao do OS, mas não é obrigatório nem a única opção utilizada para a criação das interfaces gráficas dos Software (SW), podendo ainda recorrer ao uso de outras bibliotecas ou a uma API, como o caso do Abstract Window Toolkit (AWT) ou o *Swing*[4].

Team

O controlo das alterações internas dos projectos, garantido pelo IDE Eclipse quando um ou mais programadores alteram o mesmo recurso, só é possível graças ao componente *Team*. Este componente permite que vários utilizadores se conectem nos seus próprios sistemas de controle de versão usando um determinado provedor, como por exemplo o plugin CVS [10].

Help

É o componente responsável pela gestão de toda a documentação dos plugins. É uma boa prática, se não mesmo obrigatório os desenvolvedores com o desejo de estender as funcionalidades do IDE Eclipse, criarem e adicionem uma documentação que explica o funcionamento do plugin e a relação ou dependências com outros plugins (caso exista). O formato da documentação é *HTML*, que deve estar acompanhado com um ficheiro *XML* que define a estrutura de navegação do plugin.

4.1.2 Construção de Plugins - (Módulo PDE)

Um *plugin* ou módulo de extensão é um programa de computador usado para adicionar funções a outros programas maiores, provendo alguma funcionalidade especial ou muito específica. Geralmente é pequeno e leve.

Recentemente, a partir da versão 2.1 do Eclipse, foi criado um módulo que facilita a criação, testes e depuração de *plugins*. Este kit de ferramentas denominado como PDE, permite ainda realizar a instalação e actualização dos *plugins* existentes no repositório do Eclipse, assim como os produtos do Rich Client Platform (RCP)[5].

A construção de um *plugin* pode tornar-se uma tarefa muito complicada dependendo da complexidade do módulo que se pretende construir. Um *plugin* deve ser construído levando em conta as suas dependência e os relacionamentos com outros plugins existentes, sob risco de bloquear a interface do utilizador ou originar problemas de desempenho [15] no IDE. Outros aspectos a ter em conta são as aberturas dos processos, os recurso que se pretende

[3]JFACE: pode ser visto como um kit de ferramentas de interface do utilizador ou um framework para a criação de interfaces gráficas que interagir com o SWT. Possui componentes de imagens, textos, vistas, acções, perspectivas e muito mais.

[4]Swing é uma API criada especificamente para a linguagem Java, com o objectivo de permitir uma rápida criação das Graphical User Interface (GUI).

[5]RPC: Plataforma que possibilidade a criação de aplicações *stand-alone* ricas em recursos.

ter acesso e a gestão da memória necessária. De forma resumida, pelo menos cinco passos são necessários para a construção de um *plugin* no Eclipse [18]:

1. Configuração do PDE
2. Criação de um projecto do tipo *plugin*;
3. Configuração do arquivo de *Manifest.xml*;
4. Testar o *plugin*
5. Importar o *plugin* no formato *.zip* ou *.jar*.

Como se pode observar na figura 4.1, o Eclipse é extensível (*Your tool here*). Esta sua propriedade garante a possibilidade de que esta plataforma esteja aberta para novas implementações que estendem as suas funcionalidade, o que é precisamente o objetivo do presente trabalho de mestrado que visa dotar o Eclipse de capacidade para gerar e manter anotações de biofeedback.

4.2 *JAVA*

Java é uma linguagem de programação Orientada a Objectos que surgiu de um projecto de pesquisa da empresa *Sun Mycrosystems* nos anos 90. Em 1991 foi lançada oficialmente como linguagem de programação pelo canadense *James Gosling*. É actualmente uma da linguagem de programação com estabilidade e voltada para a construção de aplicações *Stand-alone*.

A razão da escolha desta tecnologia para o presente trabalho está intimamente relacionada com as qualidades da linguagem, que são:

1. Orientação a objectos: esta característica atribui ao projecto um alto nível de abstracção e a garantia da reutilização de código, o que torna também o projecto mais pequeno e mais fácil de compreender;
2. Robustez: permite produzir códigos resistentes aos erros e excepções tratados;
3. Portabilidade: o projecto torna-se multi-plataforma e mais seguro.

Um razão adicional que esteve envolvida na escolha desta linguagem foi o facto do IDE Eclipse ser construída em linguagem *Java*.

4.3 Plugin Fluorite

O *Plugin Fluorite* tem como principal objectivo registar os eventos que acorrem na realização de um projecto usando o IDE Eclipse. Estes eventos são as operações realizadas pelo programador sobre o código, como a inserção de nova linha, a selecção do código, o movimento do rato (*mouse*) e do cursor, as entradas do teclado. Este *plugin* regista até mesmo os caracteres ao nível de uma palavra (dependendo apenas do tempo configurado para o registo), o que torna o plugin muito bom a nível de granularidade dos *tokens*. As

operações capturadas, referentes a todos os eventos que ocorrem no editor de código, são guardadas pelo *Fluorite* num ficheiro de registos (ou *Logs*) com o formato *XML*.

```
<Events startTimestamp="1574731473899" logVersion="0.11.0.qualifier" osName="Mac OS X" osVersion="10.15.1" lineSeparator="\n"
numMonitors="1" monitorBounds="[0, 0, 1280, 800]">
  <Command __id="0" _type="ShellBoundsCommand" bounds="[0, 23, 1280, 735]" timestamp="17" />
  <Command __id="1" _type="FileOpenCommand" docASTNodeCount="83" docActiveCodeLength="372" docExpressionCount="44" docLength="372"
  projectName="CAlculadora" timestamp="113">
    <filePath><![CDATA[/Users/bongocahisso/Documents/eclipse-ide/2018/runtime-EclipseApplication/CAlculadora/src/Operacao.java]]></
    filePath>
    <snapshot><![CDATA[
public class Operacao {

  public static String mensagem() {
    return "Testando o log com o fluorite";
  }

  public static float soma(float x, float y) {
    return x+y;
  }

  public static void main(String[] args) {
    System.out.println(mensagem());
    int x = 3;
    int y = 4 - x;

    String xx = "Banana";
    System.out.println(xx);
    System.out.println(soma(x,y));
  }

}
]]></snapshot>
  </Command>
  <Command __id="2" _type="MoveCaretCommand" caretOffset="108" docOffset="108" timestamp="606" />
  <DocumentChange __id="3" _type="Insert" docASTNodeCount="83" docActiveCodeLength="374" docExpressionCount="44" docLength="374" length
  ="2" offset="108" timestamp="4907">
    <text><![CDATA[
  ]]></text>
  </DocumentChange>
  <Command __id="4" _type="InsertStringCommand" timestamp="4926">
    <data><![CDATA[
]]></data>
  </Command>
  <Command __id="5" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.LINE_UP" timestamp="6052" />
  <DocumentChange __id="6" _type="Insert" docASTNodeCount="83" docActiveCodeLength="376" docExpressionCount="44" docLength="376" length
  ="2" offset="108" timestamp="6941">
    <text><![CDATA[
  ]]></text>
  </DocumentChange>
  <Command __id="7" _type="InsertStringCommand" timestamp="6953">
    <data><![CDATA[
]]></data>
  </Command>
  <DocumentChange __id="8" _type="Insert" docASTNodeCount="83" docActiveCodeLength="383" docExpressionCount="44" docLength="383" length
  ="7" offset="110" repeat="7" timestamp="8639" timestamp2="10581">
    <text><![CDATA[Public ]]></text>
  </DocumentChange>
  <Command __id="9" _type="InsertStringCommand" repeat="7" timestamp="8643" timestamp2="10585">
    <data><![CDATA[Public ]]></data>
  </Command>
  <Command __id="22" _type="MoveCaretCommand" caretOffset="111" docOffset="111" timestamp="14106" />
  <DocumentChange __id="23" _type="Delete" docASTNodeCount="83" docActiveCodeLength="382" docExpressionCount="44" docLength="382"
  endLine="7" length="1" offset="110" startLine="7" timestamp="15245">
    <text><![CDATA[P]]></text>
  </DocumentChange>
  <Command __id="24" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.DELETE_PREVIOUS" timestamp="15249" />
  <DocumentChange __id="25" _type="Insert" docASTNodeCount="84" docActiveCodeLength="383" docExpressionCount="44" docLength="383"
  length="1" offset="110" timestamp="17132">
    <text><![CDATA[p]]></text>
  </DocumentChange>
  <Command __id="26" _type="InsertStringCommand" timestamp="17138">
    <data><![CDATA[p]]></data>
  </Command>
  <Command __id="27" _type="MoveCaretCommand" caretOffset="117" docOffset="117" timestamp="18027" />
  <DocumentChange __id="28" _type="Insert" docASTNodeCount="84" docActiveCodeLength="386" docExpressionCount="44" docLength="386"
  length="3" offset="117" repeat="3" timestamp="19214" timestamp2="19675">
    <text><![CDATA[sta]]></text>
  </DocumentChange>
  <Command __id="29" _type="InsertStringCommand" repeat="3" timestamp="19219" timestamp2="19679">
    <data><![CDATA[sta]]></data>
  </Command>
  <DocumentChange __id="34" _type="Insert" docASTNodeCount="85" docActiveCodeLength="390" docExpressionCount="44" docLength="390"
  length="4" offset="120" repeat="4" timestamp="22044" timestamp2="23025">
    <text><![CDATA[tic ]]></text>
  </DocumentChange>
  <Command __id="35" _type="InsertStringCommand" repeat="4" timestamp="22049" timestamp2="23030">
    <data><![CDATA[tic ]]></data>
  </Command>
```

Figura 4.2: Logs com o Plugin Fluorite (cabeçalho e o *"snapshot"*).

```
</Command>
<DocumentChange __id="42" _type="Insert" docASTNodeCount="85" docActiveCodeLength="396" docExpressionCount="44" docLength="396" length="6" offset="124" repeat="6" timestamp="25851" timestamp2="27963">
  <text><![CDATA[float ]]></text>
</DocumentChange>
<Command __id="43" _type="InsertStringCommand" repeat="6" timestamp="25857" timestamp2="27969">
  <data><![CDATA[float ]]></data>
</Command>
<DocumentChange __id="54" _type="Insert" docASTNodeCount="89" docActiveCodeLength="404" docExpressionCount="45" docLength="404" length="8" offset="130" repeat="8" timestamp="31844" timestamp2="33901">
  <text><![CDATA[subtrair]]></text>
</DocumentChange>
<Command __id="55" _type="InsertStringCommand" repeat="8" timestamp="31849" timestamp2="33907">
  <data><![CDATA[subtrair]]></data>
</Command>
<DocumentChange __id="70" _type="Insert" docASTNodeCount="89" docActiveCodeLength="406" docExpressionCount="45" docLength="406" length="2" offset="138" timestamp="36554">
  <text><![CDATA[()]]></text>
</DocumentChange>
<Command __id="71" _type="InsertStringCommand" timestamp="36597">
  <data><![CDATA[()]]></data>
</Command>
<DocumentChange __id="72" _type="Insert" docASTNodeCount="89" docActiveCodeLength="410" docExpressionCount="45" docLength="410" length="4" offset="139" repeat="4" timestamp="38645" timestamp2="39576">
  <text><![CDATA[foat]]></text>
</DocumentChange>
<Command __id="73" _type="InsertStringCommand" repeat="4" timestamp="38650" timestamp2="39578">
  <data><![CDATA[foat]]></data>
</Command>
<Command __id="80" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.COLUMN_PREVIOUS" repeat="3" timestamp="41030" timestamp2="41371" />
<DocumentChange __id="83" _type="Insert" docASTNodeCount="89" docActiveCodeLength="411" docExpressionCount="45" docLength="411" length="1" offset="140" timestamp="42201">
  <text><![CDATA[l]]></text>
</DocumentChange>
<Command __id="84" _type="InsertStringCommand" timestamp="42204">
  <data><![CDATA[l]]></data>
</Command>
<Command __id="85" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.COLUMN_NEXT" repeat="3" timestamp="42963" timestamp2="43325" />
<DocumentChange __id="88" _type="Insert" docASTNodeCount="95" docActiveCodeLength="422" docExpressionCount="47" docLength="422" length="11" offset="144" repeat="11" timestamp="44335" timestamp2="48351">
  <text><![CDATA[ x, float y]]></text>
</DocumentChange>
<DocumentChange __id="189" _type="Delete" docASTNodeCount="101" docActiveCodeLength="462" docExpressionCount="50" docLength="462" endLine="11" length="3" offset="196" repeat="3" startLine="11" timestamp="210095" timestamp2="210446">
  <text><![CDATA[oal]]></text>
</DocumentChange>
<Command __id="190" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.DELETE_PREVIOUS" repeat="3" timestamp="210098" timestamp2="210449" />
<DocumentChange __id="195" _type="Insert" docASTNodeCount="101" docActiveCodeLength="466" docExpressionCount="50" docLength="466" length="4" offset="196" repeat="4" timestamp="211175" timestamp2="212927">
  <text><![CDATA[load]]></text>
</DocumentChange>
<Command __id="196" _type="InsertStringCommand" repeat="4" timestamp="211180" timestamp2="212932">
  <data><![CDATA[load]]></data>
</Command>
<DocumentChange __id="203" _type="Delete" docASTNodeCount="101" docActiveCodeLength="465" docExpressionCount="50" docLength="465" endLine="11" length="1" offset="199" startLine="11" timestamp="215359">
  <text><![CDATA[d]]></text>
</DocumentChange>
<Command __id="204" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.DELETE_PREVIOUS" timestamp="215363" />
<DocumentChange __id="205" _type="Insert" docASTNodeCount="105" docActiveCodeLength="484" docExpressionCount="51" docLength="484" length="19" offset="199" repeat="18" timestamp="215598" timestamp2="225802">
  <text><![CDATA[t multiplicar(flao)]]></text>
</DocumentChange>
<Command __id="206" _type="InsertStringCommand" repeat="18" timestamp="215603" timestamp2="225803">
  <data><![CDATA[t multiplicar(flao]]></data>
</Command>
<DocumentChange __id="241" _type="Delete" docASTNodeCount="105" docActiveCodeLength="482" docExpressionCount="51" docLength="482" endLine="11" length="2" offset="215" repeat="2" startLine="11" timestamp="226622" timestamp2="226834">
  <text><![CDATA[ao]]></text>
</DocumentChange>
<Command __id="242" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.DELETE_PREVIOUS" repeat="2" timestamp="226626" timestamp2="226837" />
<DocumentChange __id="245" _type="Insert" docASTNodeCount="108" docActiveCodeLength="494" docExpressionCount="52" docLength="494" length="12" offset="215" repeat="12" timestamp="227458" timestamp2="232473">
  <text><![CDATA[oat x, floar]]></text>
</DocumentChange>
<Command __id="246" _type="InsertStringCommand" repeat="12" timestamp="227460" timestamp2="232475">
  <data><![CDATA[oat x, floar]]></data>
</Command>
<DocumentChange __id="269" _type="Delete" docASTNodeCount="108" docActiveCodeLength="493" docExpressionCount="52" docLength="493" endLine="11" length="1" offset="226" startLine="11" timestamp="233846">
  <text><![CDATA[r]]></text>
</DocumentChange>
```

Figura 4.3: Logs com o Plugin Fluorite (com mais atributos e operações).

Neste trabalho utilizou-se o plugin *Fluorite* para capturar as operações e os instantes de tempo (timestamp) das suas execuções. As figuras 4.2 e 4.3 mostram um exemplo da estrutura do ficheiro XML em que o *Fluorite* regista os eventos. Primeiro é criado o cabeçalho com a data e hora da do início em o projecto é carregado. A seguir o *Fluorite* cria um *"snapshot"*[6] do código encontrado ou seja, que será alterado. Finalmente, passa

[6]O snapshot é uma cópia do código-fonte no estado inicial da recolha dos logs. Esta fase, permite

para os registos de todas as operações com os seus instantes de tempo.

Ainda nas figuras (4.2 e 4.3) observa-se dois elementos do tipo *XML* que são armazenados pelo *plugin Fluorite*, o *Command* e *CommandChange*, ambos são resultados de uma operação realizada pelo programador no editor com a diferença de que o segundo ocorre somente quando a operação é feita sob uma linha de código não vazia, e possuem os seguintes atributos:

- *_type*: Atributo que define o tipo de operação efectuada pelo programador (inserção, atualização, eliminação, movimentação e outras);
- *timestamp*: Atributo que apresenta o instante que iniciou a operação;
- *timestamp2*: Atributo que define o instante que terminou a operação;
- *offset*: Atributo que define a posição do token no espaço do editor;
- *repeater*: Atributo que indica o número de visitas feitas na linha ou no *token*;
- *startline*: Atributo que indica a linha inicial no código fonte onde ocorreu a operação;
- *endline*: Atributo que indica a linha final no código fonte onde ocorreu a operação;
- E o *tempo de duração da operação*: É um atributo calculável que resulta da subtracção entre os atributos *timestamp2* e o *timestamp*.

Os atributos acima foram combinados com os dados de biofeedback e da complexidade do código com objectivo de realizar-se as anotações e as marcações do código-fonte. A secção 6 aborda com maior detalhes o uso dos dados das operações realizadas pelo programador no processo de construção do protótipo.

4.4 *Plugin Eclipse HighlightOnSelection*

A IDE Eclipse possui um repositório com vários plugins prontos a usar. Naturalmente, cada plugin é concebido para um determinado propósito. O *plugin HighlightOnSelection*, quando instalado na IDE Eclipse, realiza o realce de todos os *tokens* idênticos a um *token* chave seleccionado, ou seja, quando clicado duas vezes sobre um *tokens* é accionado um mecanismo de busca dos restantes *tokens* iguais e este são marcados com uma cor[7] definida pelo programador nas preferências da IDE.

O código fonte deste plugin pode ser encontrado no repositório repositório github. É de acesso livre, desenvolvido com a linguagem de programação *JAVA* o lhe permite que suporte as plataformas *Windows*, *Mac OS* e *Linux*. O seu código pode ser adaptado ou melhorado de acordo ao abrigo da licença *EPL*. Foi criado em 26-03-2018 e actualizado pela última vez em 22-10-2019, por ***Ben Jiang***[8].

No presente trabalho, o *plugin HighlightOnSelection* foi adaptado de modo a utilizar quatro (4) cores de uma única vez, onde cada cor representa um cenário (para mais detalhes sobre os cenários das cores consulte a sub-secção 6.4.2). Foi ainda, modificado o tipo do evento

restaurar código fonte desde o estado que se encontra para o estado inicial de toda actividade, caso seja necessário.

[7] A cor padrão é a verde.

[8] Não foram encontradas outras informações referente ao desenvolvedor do *HighlightOnSelection*, *Ben Jiang*.

(ou elemento accionador) para a selecção ou marcação (*highlight*). A marcação passou a ser feita de forma periódica[9] ou por meio da chamada de uma acção pelo programador usando uma interface com botões (para saber mais veja a figura 5.4 e a sub-secção 5.3.1).

Quanto à busca dos *tokens* a serem marcados, esta não foi considerada na adaptação do *plugin*, pois passou-se a enviar as regiões e tamanhos precisos pelo algoritmo de sincronização depois de pré-processado os dados do *plugin Fluorite*.

4.5 *WEKA*

WEKA é uma espécie de ave terrestre originária da Nova Zelândia, que é conhecida pela sua curiosidade extrema, ou natureza investigadora. No mundo tecnológico, o nome *WEKA* significa *Waikato Environment for Knowledge Analysis*, sendo um ambiente/ferramenta de análise desenvolvido na *Universidade de Waikato*, da *Nova Zelândia*[12].

O *WEKA* pode ser visto de três formas diferentes. A primeira, como um software de aprendizagem máquina acompanhado de uma interface gráfica, e que pode ser utilizadas sem a necessidade de conhecimento de programação. A segunda forma, como apenas uma ferramenta de mineração de dados quando o objectivo é somente descobrir conhecimentos num conjunto de dados, novamente sem que seja necessário ter conhecimentos de programação. A terceira forma (de ver o *WEKA*) é encará-lo como uma biblioteca de algoritmos de aprendizagem e ferramentas de pré-processamento de dados, o que inclui a preparação dos dados de entrada, a avaliação estatística dos esquemas de aprendizado e a visualização dos dados de entrada e do resultado da aprendizagem [12].

Figura 4.4: Interface gráfica da ferramenta WEKA

Neste trabalho usamos o *WEKA* como uma biblioteca de algoritmos de aprendizagem máquina. Foi escolhido porque o seu código é aberto e estável, multi-plataforma e foi desenvolvido com a linguagem de programação *JAVA*.

[9]Marcação Periódico: Entende-se aqui como um processo de marcação dos *tokens* de modo automático, realizado num intervalo de tempo pré-estabelecido. O tempo por padrão é meio minuto e pode ser ajustado para um outro valor.

Capítulo 5

Expansão da IDE Eclipse com Biofeedback e Highlighting

Este capítulo apresenta as metodologias aplicadas para a materialização da solução no intuito de construir um protótipo de uma IDE com capacidade de gerar e manter as atualizações de biofeedback e produzir indicações no programador através de *highlighting* das zonas de código específicas. Começamos com a análise da metodologia, no intuito de apresentar as ideias gerais de como pretendemos alcançar o objectivo. Em seguida, apresentamos uma análise das características (com a listagens dos requisitos funcionais, restrições e os impactos), segue-se com uma análise da arquitectura do protótipo e, por fim, apresentamos a aplicação, em termos gerais das metodologias da Engenharia de Software Aumentada de Biofeedback (BASE) no protótipo.

5.1 Abordagens metodológicas

O que pretendemos obter com este trabalho é a expansão do Ambiente de Desenvolvimento Integrado (IDE) Eclipse para anotar e realçar linhas de códigos ou *tokens* em função do estado cognitiva do programador enquanto manipula essas linhas de código/*tokens*. Para tal, três estratégias foram adoptadas como ilustrado na figura 5.1:

Figura 5.1: Estratégia para anotação de código.

5.1.1 Metodologia de *Biofeedback*

As metodologias de *Biofeedback* foram propostas recente no âmbito do projecto BASE (como descrito na secção 2.2) e visam monitorar o estado cognitivo (particularmente esforço mental) dos programadores usando dispositivos não intrusivos (e.g., *smart watches* e *eye trackers*), com vista a identificar de forma automática excertos de código com maior potencial para conter bugs (e.g., porque são exigem maior esforço dos programadores ou porque os programadores estavam pouco focados durante o seu desenvolvimento e/ou inspecção.

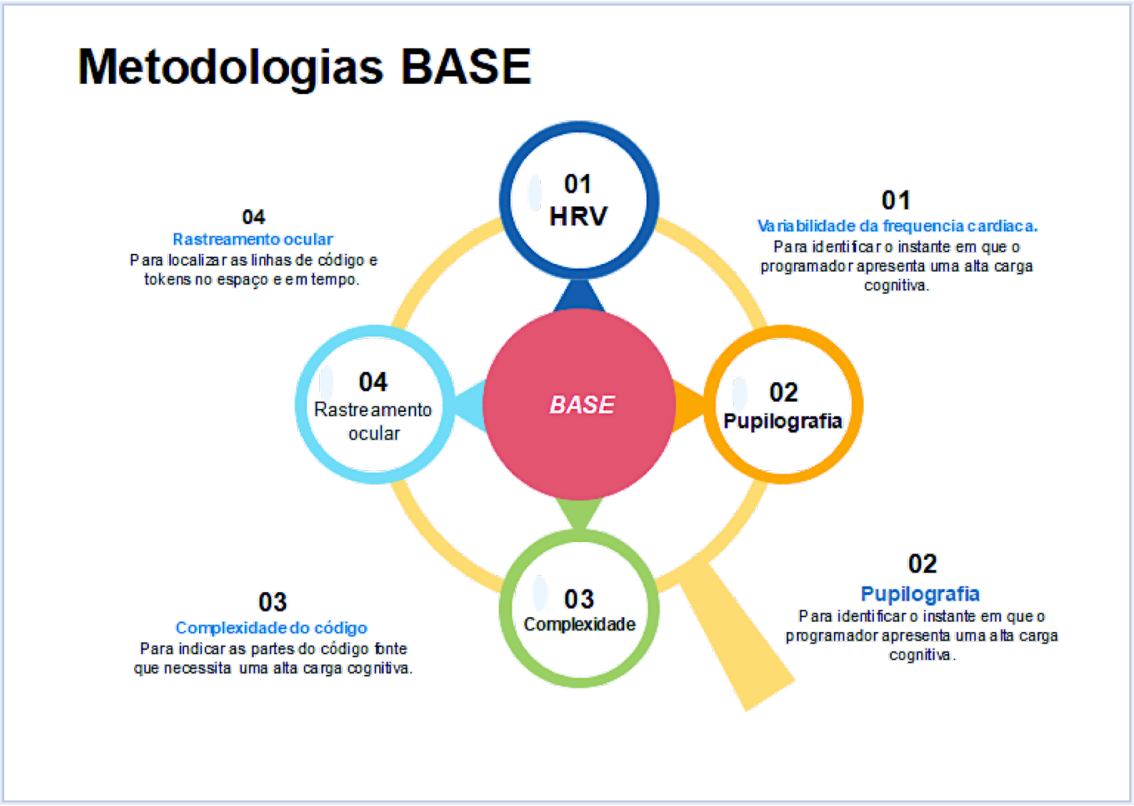


Figura 5.2: Resumo das metodologias BASE.

A figura 5.2 ilustra as quatros características (ou *feature* de alta dimensionalidade) que o projecto BASE usa na elaboração das metodologias, mas apenas as característica 1,2 e 4 fazem parte da metodologia biofeedback que praticamente é definida pela combinação das variáveis de rastreamento ocular, variabilidade da frequência cardíaca (VFC ou HRV em inglês) e da pupilografia e avaliado por um classificador. Este classificador tem vindo a ser desenvolvido e sucessivamente melhorado no projeto BASE, usando *Matlab*, tendo como resultado final uma versão *standalone*, uma vez que o *Matlab* permite a criação de versões em linguagem C dos modelos e classificadores desenvolvidos.

O resultado do classificador é denominado *"Carga Cognitiva do Programador"*, e é armazenado num ficheiro de extensão *.csv*, sendo disponibilizado via Interface de Programação de Aplicações (API) para o consumo no IDE Eclipse (como ilustrado no diagrama de contexto da secção 5.2). Quanto aos dados disponibilizado, a sua descrição e estrutura é relativamente simples e compreensível como se pode observar na figura 6.2.

5.1.2 Classificação das operações

Durante o desenvolvimento de Software (SW) usando o IDE Eclipse, o programador realiza um conjunto de operações, seja pelo teclado manual ou digital, ou pelo *mouse*. A informação sobre estas operações é armazenado pelo *plugin* de recolha de evento (*Flourite*) e permite determinar a atividade do programador sobre o código aberto no editor do IDE. A figura 5.1 apresenta, as operações que foram levadas em consideração neste trabalho. Elas são: inserção (*INSERT*), selecção (*ONLY SELECTION*), actualização (*UPDATE*), selecção e actualização (*SELECTION AND UPDATE*), deslocamento ou movimentação (*MOVE*), colagem (*PAST*), substituição (*REPLACE*) e eliminação (*DELETE*) de *tokens*. A figura ilustra ainda as classificações que atribuímos às operações dada as suas sensibilidades face a probabilidade da injecção de *bugs*, e elas são:

- *Normal*: representam as operações em que o programador realiza com uma medida de carga cognitiva baixa;
- *Esforço*: operações idênticas a *Normal* mas em que o programador evidenciou maior carga cognitiva;
- *Crítica*: operações em que o programador apresenta carga cognitiva alta. Nestas operações o programador pode enganar-se com maior facilidade;
- *Muito Crítica*: operações realizadas em condições semelhantes às do tipo *Crítica* mas com carga cognitiva do programador ainda mais alta. Nestes casos, é fortemente recomendado a verificação do código por parte do programador, sendo usado o *highlighting* das linhas de código para indicar essa necessidade.

5.1.3 Significado das Anotações e Complexidade do Código

As anotações de linhas de código e de *tokens* com a classificação de quatro níveis apresentada na secção anterior (*Normal, Esforço, Critica, Muito Crítica*) representa a forma mais simples para concretizar as anotações de *biofeedback*. A decisão de concretizar o protótipo usando esta abordagem tem as seguintes vantagens:

- Permite desenvolver o protótipo da IDE com anotações de *biofeedback* numa fase inicial do projecto, enquanto a investigação sobre as melhores técnicas para a extração de *features* relacionadas com estados cognitivos do programador está a decorrer;
- Separa conceptualmente (e na prática) as tarefas de criar e manter as anotações (feitas pelo protótipo desenvolvido) da extração de *features* a partir dos sensores usados (de momento, apenas HRV, pupilometria e rastreamento ocular), cujos resultados são, como já foi apresentado, disponibilizados através de uma API. Isto permite manter em aberto a possibilidade de usar mais sensores e melhores classificadores, que permitam, por exemplo, diferenciar diferentes estados cognitivos, como esforço mental, atenção, stress ou fadiga;
- Separa a criação e manutenção das anotações da sua utilização para dar indicações ao programador (através de *code highlighting*) ou para indicar prioridades nos testes dos diferentes módulos em desenvolvimento. Ou seja, permite que posteriormente se use diferentes algoritmos e técnicas de aprendizagem máquina para analisar as anotações, integrando essa informação com a complexidade do código e com outras informações como, por exemplo, o histórico de tipos de *bugs* de um programador, para prever quais as zonas do código com maior probabilidade de ter *bugs*.

Os sensores utilizados no BASE (HRV, pupilometria e rastreamento ocular) permitem medir diretamente carga cognitiva e inferir, de forma indireta, outros estados cognitivos, como atenção e fadiga, especialmente se a informação de HRV e pupilometria (que indicam primariamente esforço mental) for conjugada com a informação fornecida pelo *eye tracker* sobre a atividade do programador.

Há outros sensores que também são compatíveis como a atividade de desenvolvimento de código como é o caso de Actividade Electrodérmica (EDA), também conhecido por resposta galvânica da pele. Existem sensores comerciais sob a forma de anéis ou de pulseiras, que permitem medir a resposta galvânica da pele de forma não intrusiva. O projeto BASE está agora a avaliar a utilização desses sensores, pois é bem conhecido da literatura que EDA permite detectar com segurança situações de stress cognitivo (as pessoas suam em situações de stress)[19, 30].

A abordagem usada na construção do protótipo nesta tese de mestrado permite facilmente acrescentar novos sensores e novos classificadores para gerarem a informação a inserir nas anotações de código e, ao mesmo tempo, abre a possibilidade de utilizar as anotações de múltiplas formas usado técnicas de aprendizagem de máquina. Desta forma, o protótipo desenvolvido é um elemento chave para o trabalho futuro no projeto BASE.

A complexidade do código (indicada pelas conhecidas métricas de complexidade de código tais como as métricas de *McCabe* e *Halstead*) é, naturalmente, um elemento muito importante na análise do significado da informação relativa à carga cognitiva. Por exemplo, é normal (e esperado) que a carga cognitiva aumente em zonas de código mais complexo. Por isso é muito fácil e intuitivo formular algumas regras para inferir o risco de haver *bugs* em determinadas zonas do código analisando em simultâneo a carga cognitiva e a complexidade do código. A título de exemplo, indica-se algumas destas regras:

- Carga cognitiva *Muito Crítica* em zonas de código de baixa complexidade: sugere que o programador está distraído ou em stress devido a causas externas ao código (que é simples), pelo que a probabilidade de se enganar aumenta;

- Carga cognitiva *Normal* em zonas de código de muito alta complexidade: sugere que o programador não está concentrado na análise do código (por exemplo, numa inspecção de código).

Estes exemplo simples e intuitivos mostram a necessidade de as anotações de código também incorporarem métricas de complexidade.

5.2 Diagrama de Contexto

O diagrama de contexto apresenta a interacção do SW que se pretende desenvolver com os elementos (que podem ser sistemas, API, pessoas ou máquinas) externos. Este diagrama possui o nível de abstracção zero o que significa que os detalhes sobre o funcionamento do SW é limitado apenas aos fluxos de entradas e saídas, o que facilita a descoberta dos atores externos (ou seja, os elementos que iram interagir com o sistema).

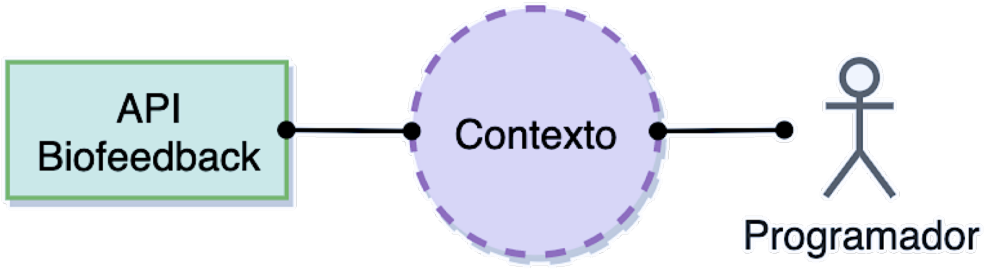


Figura 5.3: Diagrama de contexto.

A figura 5.3 ilustra o diagrama de contexto do protótipo que desenvolvido. Este possui duas interacções externas, uma com a *API Biofeedback* que é um serviço fora do que se vai desenvolver mas que o consumimos, e a segunda é o *Programador*, representa o actor que vai interagir com o sistema.

5.3 Requisitos do Sistema

Esta secção apresenta as técnicas da Engenharia de Requisitos (ER) utilizadas na identificação dos requisitos do sistema e, principalmente, na resolução dos conflitos entre os requisitos.

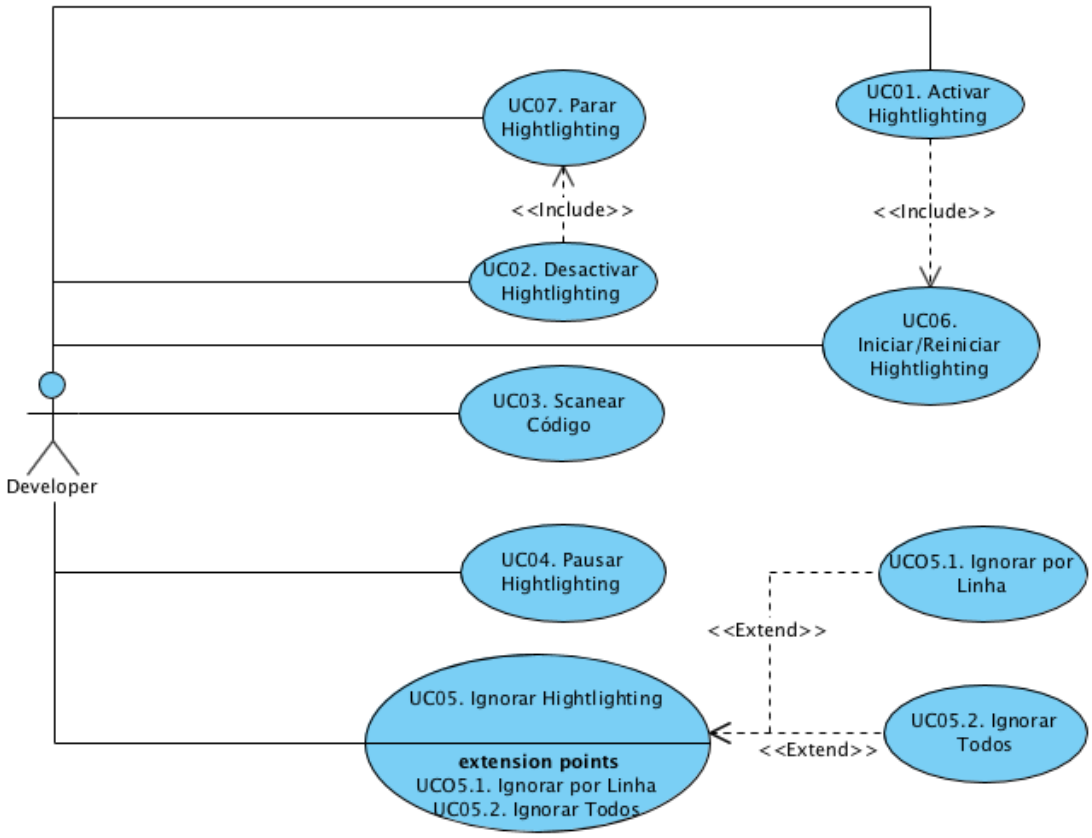


Figura 5.4: Diagrama de caso de uso do sistema

Das técnicas da ER a que mais se destacam na recolha dos requisitos funcionais são os Modelos de Casos de Usos. Estes modelos são apreciados, não só pela facilidade no entendimento como no modo em que as funcionalidades ou casos de usos são descritos.

O modelo de caso de uso (*Use Case*) descreve o comportamento que o sistema deverá apresentar na interacção com os seus intervenientes (actores e entidades externas). A figura 5.4 ilustra o diagrama de caso de uso do sistema como apenas um actor (o *Developer* ou programador) e nove casos de usos ou nove funcionalidades. Na figura é possível identificar as funcionalidades que o actor Developer tem acesso directo e indirecto, e ainda como as funcionalidade interagem entre si. Para a construção destes diagrama foi utilizado a ferramenta Visual Paradigm[1].

5.3.1 Requisitos Funcionais

Em seguida apresentamos a lista das funcionalidades do sistema, incluindo aquelas que o programador não interage com elas directamente:

Nº.	**Requisitos**	**Prioridade**
1	UC01. Activar Hightlighting	**1**
2	UC02. Desactivar Hightlighting	**2**
3	UC03. Scanear Código	**2**
4	UC04. Pausar Hightlighting	**3**
5	UC05. Ignorar Hightlighting	**5**
6	UC05.1. Ignorar por Linha	**5**
7	UC05.2. Ignorar Todos	**5**
8	UC06. Reiniciar Hightlighting	**4**
9	UC07. Parar Hightlighting	**3**
10	UC08. Importar dados	**1**
11	UC08.1. Importar dados Logs	**1**
12	UC08.2. Importar dados Complexidade	**3**
13	UC08.3 Listar ficheiros de anotações	**3**
14	UC08.4 Importar dados Biofeedback	**3**
15	UC09 Sincronizar dados	**2**
16	UC10 Classificar Operação	**3**
17	UC11 Marcar tokens	**1**
18	UC12 Pesquisar linha	**2**
19	UC13 Pesquisar tokens	**2**
20	UC14 Exportar ficheiro de anotação	**5**

Tabela 5.1: Lista dos requisitos funcionais

[1]Visual paradigm. [Online]. Available: https://www.visual-paradigm.com/

Os requisitos funcionais representam as propriedades ou operações que os sistemas realizam. A tabela 5.1 apresenta a lista dos requisitos funcionais do sistema a ser modelado. As funcionalidade de 1 a 9 têm como início do fluxo o desenvolvedor (ver o diagrama 5.4) e as restante, o sistema. É ainda, importante realçar as prioridades que vão de 1 a 5 onde quanto menor for, maior é a prioridade. Estas prioridades foram atribuídas de acordo a importância e as dependências. Assim, quanto mais dependente é um requisito menor é a sua prioridade (i.e., é mais prioritário) e tem um valor maior. Está informação é muito útil para guiar o desenvolvimento e os testes do SW.

5.3.2 Atributos de Qualidade

Os atributos de qualidade são propriedades que representam outros tipos de requisitos do sistema ligados aos requisitos não funcionais. Essas propriedades são aplicadas nas fases do funcionamento e desenvolvimento do sistema. Para a fase do desenvolvimento do SW. Para a fase do desenvolvimento do sistema, os atributos de qualidade que nos pareceram mais importante são:

- Para o utilizador:

    1. AQ01 - O Sistema deve manter o desempenho do IDE, sem o tornar inaceitavelmente lento;
    2. AQ02 - O Sistema deve ser capaz de manter `integro` o código produzido pelo programador;
    3. AQ03 - O Sistema deve possuir uma interface `compreensível` e fácil de usar, idêntica à da IDE.

- Para o programador:

    1. AQ04 - O Sistema deve funcionar em qualquer plataforma, desde que use o IDE Eclipse;
    2. AQ05 - O Sistema deve ser desenvolvido de modo a facilitar a sua manutenção e extensão;
    3. AQ06 - O Sistema deve ser desenvolvido com os padrões de projectos que permitem a reutilização de códigos.

Impacto dos atributos de qualidade por área técnica

Pretende-se com isto, apresentar o impacto de cada atributo de qualidade nas diferentes implementações das áreas técnicas do projecto. Essas áreas são: arquitectura do sistema; requisitos funcionais; restrições de design; directivas de design e restrições de implementação, como apresentado na Tabela 5.2.

Áreas Técnicas	**Atributos de Qualidade**
Arquitectura do Sistema	AQ01 - Desempenho
Requisitos Funcionais	AQ02 - Integridade
	AQ03 - Usabilidade
Restrições de Design	AQ03 - Usabilidade
Directivas de Design	AQ03 - Usabilidade
	AQ05 - Facilidade de manutenção
	AQ06 - Reutilização
Restrições de Implementação	AQ04 - Portabilidade

Tabela 5.2: Impacto dos atributos de qualidade por áreas técnicas

5.3.3 Restrições Comerciais e Técnicas

Nesta secção as restrições técnicas correspondem a uma dependência de produto, software, algoritmos, ou Sistema Operativo (OS) que condicione e possa alterar o modelo arquitectónico que se pretende desenhar. Estas restrições tem um grande impacto não só na arquitectura como nos atributos de qualidade do sistema em análise. Uma outra razão para nos preocuparmos com estas restrições é que estas são mutáveis, e quando ocorre uma mudança no negócio ou na representação de um dos produtos, esta afecta necessariamente no software construído. As restrições também têm uma grande influência na gestão do projecto durante a fase de desenvolvimento. Por estas razões, as restrições técnicas não devem ser menosprezadas mas antes analisadas em detalhe e controladas. Deste modo pode-se minimizar a complexidade do desenho e evitar a realização de tarefas espúrias, que podem resultar em grande prejuízo no futuro [20].

O IDE Eclipse é uma plataforma (*"Open Source"*) aberta a extensões e novas implementações, mas isso não quer dizer que a extensão deva ser feita usando de qualquer linguagem. Pelo contrário, permite a inclusão de novos blocos de código ou *plugins* com autonomia e capacidade de realizar uma determinada tarefa na sua própria linguagem (ver a secção 4.1.2). Está é uma das restrições técnicas. Outra restrição técnica tem haver com a obrigação de criar e configurar um ficheiro com o nome *plugin.xml*.

Face ao âmbito do protótipo, durante sua construção, apenas uma restrição comercial foi levada em consideração. Está restrição esteve relacionada com a licença Eclipse Public Licence (EPL) que exige que as soluções criadas não devem ser comercializadas.

5.4 Prototipagem

A prototipagem é uma técnica da ER utilizada para a recolha e validação dos requisitos funcionais e não funcionais e na realização dos testes de usabilidade entre os requisitos para resolver conflitos entre os requisitos. Nesta secção apresentamos apenas os protótipos finais das duas fases desta técnica que são a prototipagem baixa e a alta.

5.4.1 Protótipo de baixa fidelidade

Os protótipos de baixa fidelidade ilustram o sistema no seu estágio inicial. Normalmente é feito em papel para facilitar as mudanças e estudo de vários cenários. Com este tipo de protótipos conseguimos identificar e organizar as funcionalidades futuras.

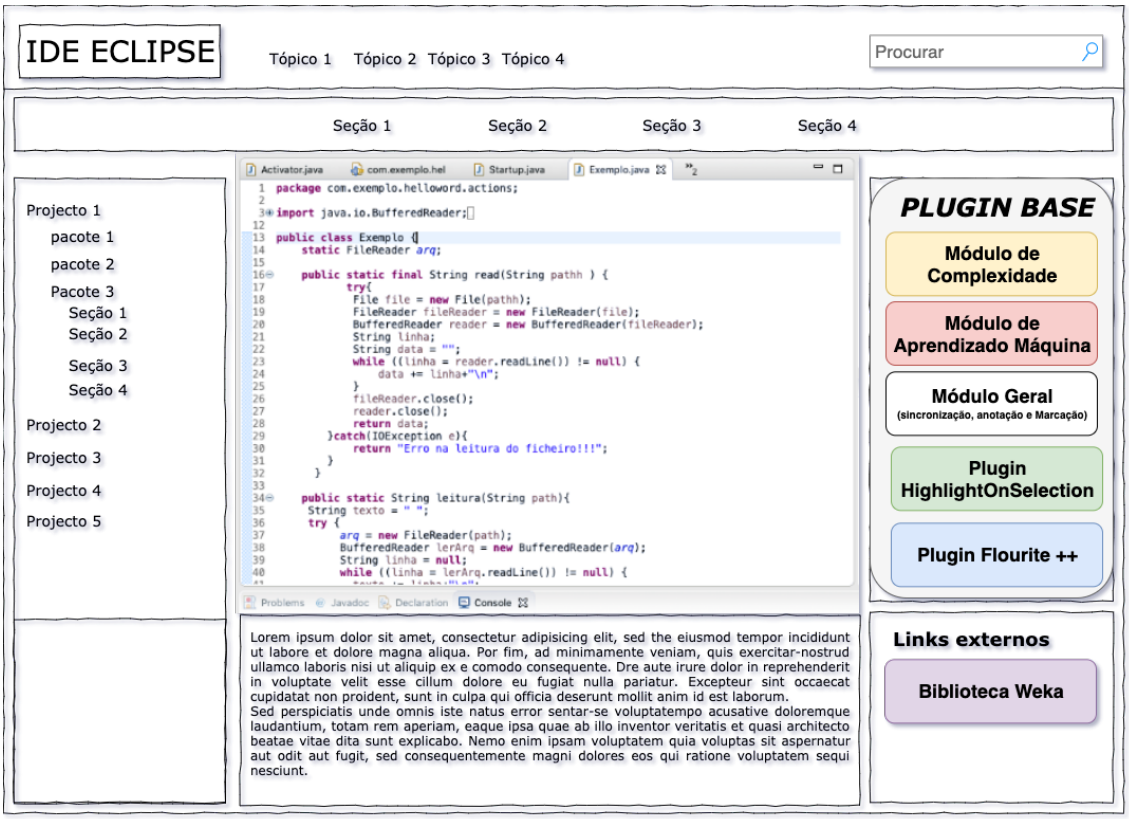


Figura 5.5: Protótipo de baixa fidelidade - visto com os componentes

A figura 5.5 apresenta um destes protótipos de baixa fidelidade que nos permitiu, em iterações sucessivas, identificar os requisitos e agrupa-los por módulos. Os módulos se encontram incorporados no *plugin BASE* que funcionará internamente no IDE Eclipse, e visto na perspectiva de agrupamento de funcionalidade (*package*) são:

- *Módulo de complexidade*: Módulo com todas as funcionalidades necessárias para o recolha, calculo e manipulação dos dados de complexidade;
- *Módulo de Aprendizagem Máquina*: Possui funcionalidade de aprendizagem máquina (treino, processamento dos dados e classificação);
- *Módulo de Geral*: Módulo com as funcionalidades gerais, é o núcleo do *plugin BASE*, permite a interligação e o funcionamento dos restantes módulos;

- *Módulo de Marcação (plugin HighlightOnSelection)*: É uma combinação das funcionalidade estendidas do *plugin HighlightOnSelection* com outras necessárias para a marcação do código-fonte;

- *Módulo das Operações (plugin Fluorite)*: É uma combinação das funcionalidade estendidas do *plugin Fluorite* com outras necessárias para a recolha dos dados das operações realizadas pelo programador.

5.4.2 Protótipo de alta fidelidade

A prototipagem de alta fidelidade ilustrar os modelos (desenhos) do sistema, o mais próximo do produto final. O seu principal objectivo é testar a usabilidade a nível do designer com os utilizadores finais. As figuras 5.7 e 5.6 ilustram o comportamento do IDE Eclipse após a integração do protótipo. A primeira, mostra os botões na barra de ferramentas e a segunda ilustra o menu com as suas respectivas operações.

A avaliação crítica que se pode fazer sobre a interface do sistema é sem dúvida Normal e Natural.

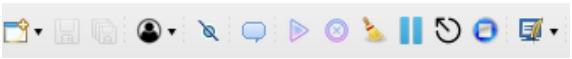


Figura 5.6: Protótipo de alta fidelidade das operações na barra de ferramentas

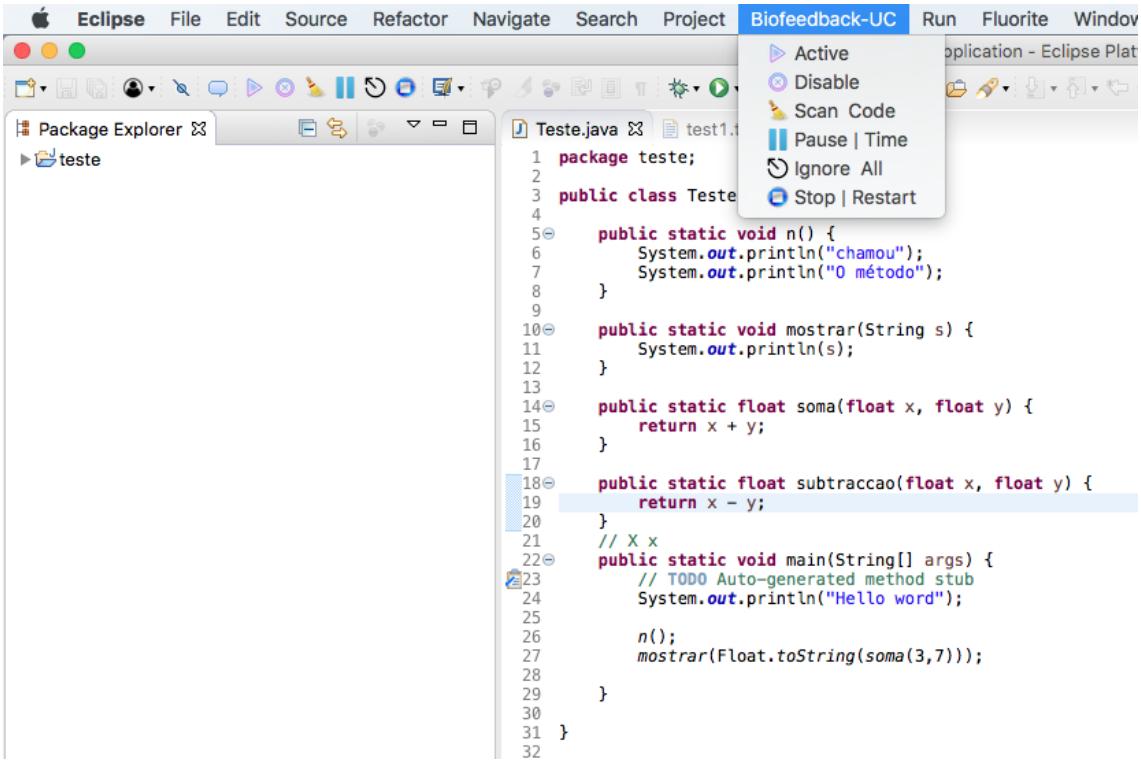


Figura 5.7: Protótipo de alta fidelidade com as operações na barra de menu

5.5 Arquitectura do Software

A figura 5.8 apresenta o modelo da arquitectura do projecto. A arquitectura foi modelada levando em conta as limitações técnicas, a garantia do melhoramento futuro, os algoritmos de sincronização, algoritmos inteligentes e todas as tecnologias escolhidas para o desenho desta solução.

Com este diagramas pretendemos não só apresentar os componentes, as relações com os actores e sistemas externos, mas também o funcionamento interno do sistema. No Diagrama de arquitectura ilustrado na figura 5.8, os componentes do software encontram-se identificados por rectângulos (ressalvamos que o sistema funcionará na IDE Eclipse e está apresentada pela área denominada por *Plugin BASE*) com cores diferentes. Os cinco componentes do sistema na perspetiva de modelos arquitetónicos são:

1. **Módulo Geral**: É o componente principal, o núcleo, onde todas as operações se originam. É responsável pelo processamento dos dados vindo de fontes diferentes, a sincronização destes dados, anotações de códigos, activação dos restantes componentes e outras funcionalidades auxiliares como o armazenamento dos históricos das anotações;

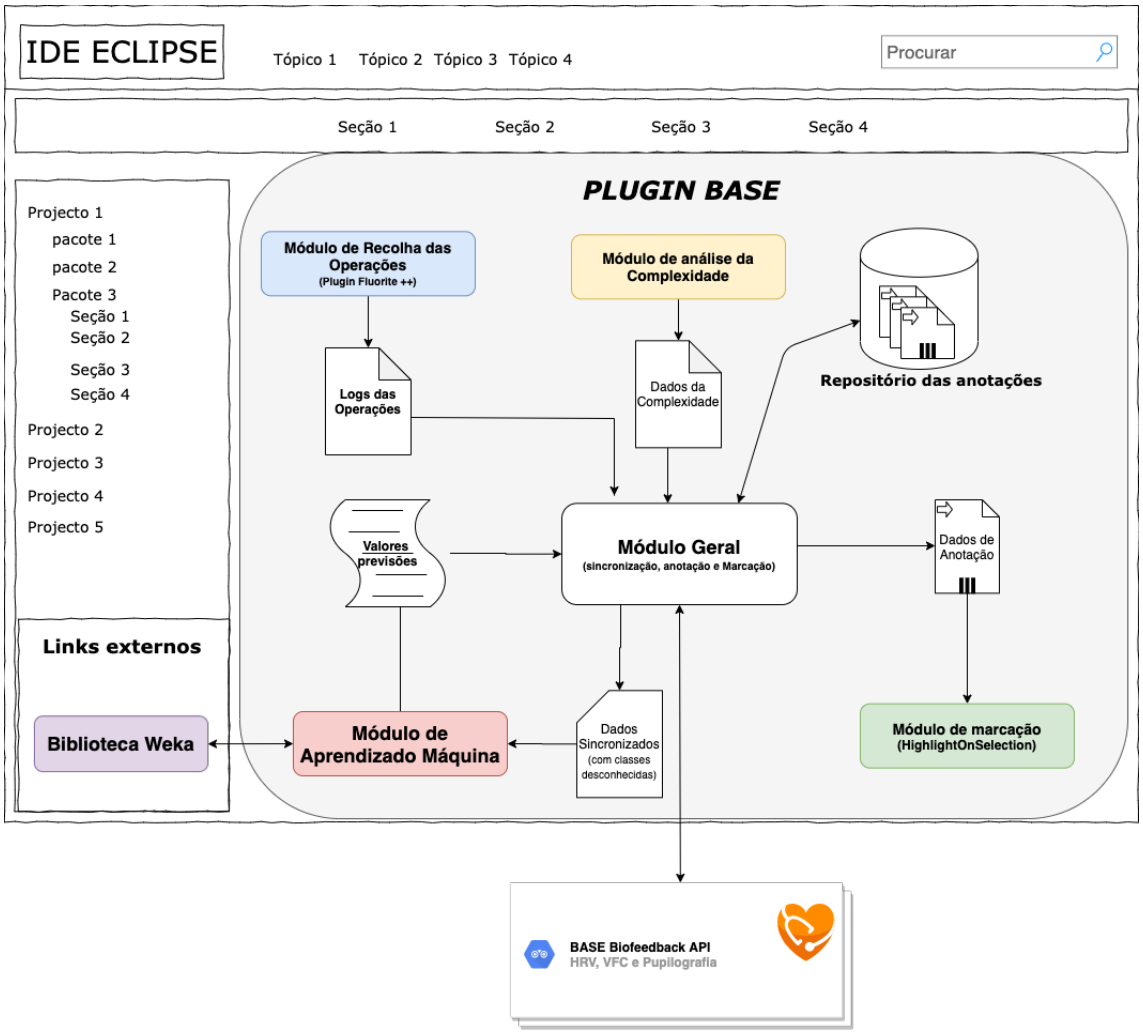


Figura 5.8: Diagrama da arquitectura do projecto

2. **Módulo de Marcação**: Este módulo é responsável pelas marcações e alertas no editor da IDE com base nos dados de anotação recebidos pelo módulo geral. Com a sua activação o programador tem a oportunidade de usar a abordagem programação por par (isto é, o sistema irá notificação da ocorrência de um possível bug em tempo real sem a necessidade de esperar até fase dos testes). Este módulo usa algumas funcionalidades estendidas do *Plugin HighlightOnSection* (descrito na secção 4.4);

3. **Módulo de Aprendizagem Máquina**: Representa um **Wrapper**, ou seja, é neste módulo onde se encontra os algoritmos de aprendizagem máquina utilizados para prever os riscos de **bugs** associados a zonas específicas de código, com bases nos dados das anotações. O resultado é colocado no ficheiro denominado Valores previsões;

4. **Módulo de Registo das Operações**: Este módulo resulta da extensão do *Plugin Fluorite* (descrito na secção 6.4.1), basicamente continua a fazer o mesmo que o *Plugin Fluorite* realiza com uma pequena diferença no tempo de armazenamento. É responsável em coletar e registar os eventos (*Logs*) das operações realizadas no código aberto no editor do IDE. O resultado é armazenado num ficheiro *XML* que chamamos de *Logs da Operações* que mais tarde é usado no processo de sincronização para a classificação dos defeitos e criação das anotações do código;

5. **Módulo das Analises de Complexidade**: Este componente tem por função o estudo da complexidade em pequenos blocos de códigos. Devolve um ficheiro com os dados sobre as métricas de complexidades associados as pequenas secções de códigos ou linhas.

Convém ainda realçar que o módulo de aprendizagem máquina possui uma dependência da biblioteca *WEKA* (descrito na secção 4.5).

Capítulo 6

Implementação do Protótipo

Neste capítulo apresentamos a implementação do protótipo usando os resultados das análises e planos apresentados nas secções anteriores. A implementação foi levada a cabo em quatro passos (fases) fundamentais:

1. Desenvolvimento dos mecanismos de recolha dos dados;

2. Processamento e transformação dos dados;

3. Aplicação das técnicas de aprendizagem máquina (*Machine Learning*) para a construção das anotações e marcação de código.

4. Avaliação do protótipo no Ambiente de Desenvolvimento Integrado (IDE) Eclipse.

6.1 Mecanismos de recolha dos dados

O mecanismos de recolha dos dados envolveu as técnicas de recolha e o pré-processamento dos dados (como ilustrado na figura 2.2). Este processo foi dividido em três (3) fases fundamentais:

1. Carregamento dos dados (sobre a complexidade do código, *biofeedback* dos programadores e os logs das operações feita no código-fonte pelo programador);

2. Pré-processamento dos dados (com destaque para a sincronização);

3. Construção das anotações de códigos.

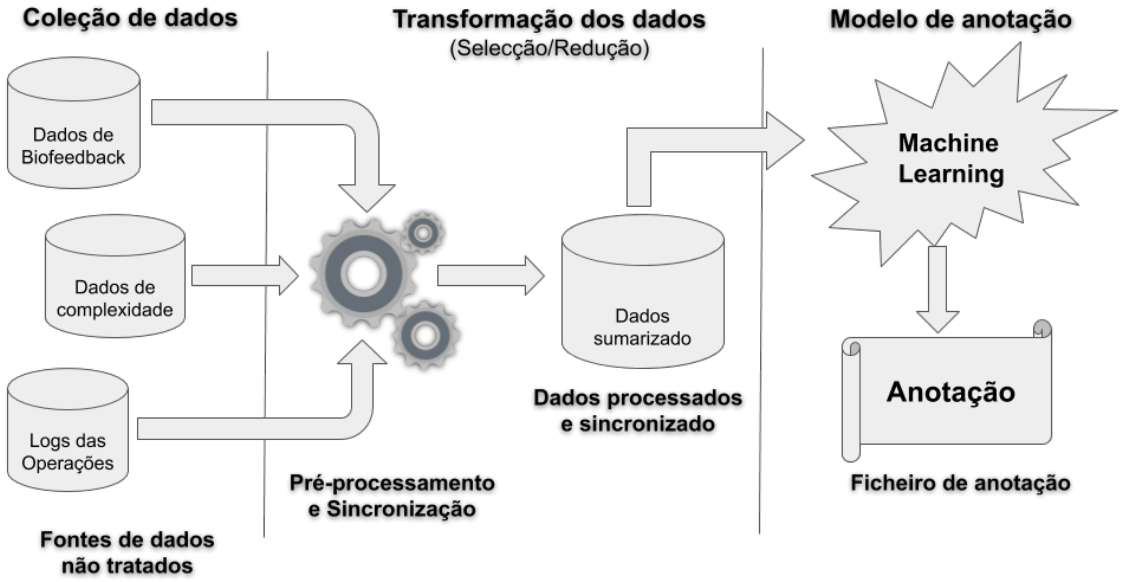


Figura 6.1: Fontes de dados

A figura 6.1 ilustra o esquema de alto-nível do processo completo que aplicamos aos dados de uma forma generalizada, cuja a sua descrição apresentamos nas sub-secções seguintes:

6.1.1 Dados de Biofeedback

Na *API BASE Biofeedback*, a colecta dos sinais de biofeedback é feita durante as tarefas de programação e fornece um classificador da carga cognitiva experimentada pelos programadores. É com base nesse classificador que o projecto Engenharia de Software Aumentada de Biofeedback (BASE) apresenta os conceitos básicos do realce de *biofeedback*, de testes guiados por *biofeedback* e de desenvolvimento de software com suporte de *biofeedback* [25].

Os dados recolhidos e classificados (dos sensores de HRV, pupilometria e eye tracking) são processados pelo classificador e o resultado (carga cognitiva) é disponibilizado via Interface de Programação de Aplicações (API). A sua estrutura é relativamente simples, como se pode observar na figura 6.2, contendo apenas a carga cognitiva, cujos valores variam de 0 a 3 (sendo 0 carga normal e 3 carga cognitiva muito elevada) e o tempo do registo do esforço mental, dado no formato *timestamp* para possibilitar a sincronização com outros dados (os outros dados e o processo de sincronização encontram-se descrito na secção 6.1.1). Os dados de *biofeedback* são usamos para a criação das anotações de códigos (o processo da criação das anotações é apresentado na secção 6.4.2), na construção das regras de associação e do modelo inteligente (apresentado na secção 6.3), ambos com objectivo de prever os riscos de conter *bugs* num *token*. O modelo inteligente, inicialmente é orientado pela carga cognitiva (*target*), por iterações a iterações aprende prever o risco usando os dados das operações do programador e da complexidade do código (descritos nas secções 6.1.3 e 6.1.2). As regras de associação são aplicadas somente no estagio inicial quando os dados forem insuficiente (abaixo de 500 casos) para a construção do modelo inteligente que resulta no risco calculado pela carga cognitiva associada a complexidade do código (apresentado na secção 6.3).

	A	B	C	D
1	timestamp	coord_x	coord_y	cognitive_burden
2	1574871714000	-	-	0
3	1574871715000	-	-	0
4	1574871716000	-	-	0
5	1574871717000	-	-	3
6	1574871718000	-	-	0
7	1574871719000	-	-	0
8	1574871720000	-	-	0
9	1574871721000	-	-	0
10	1574871722000	-	-	0
11	1574871723000	-	-	0
12	1574871724000	-	-	2
13	1574871725000	-	-	0
14	1574871726000	-	-	0
15	1574871727000	-	-	0
16	1574871728000	-	-	0
17	1574871729000	-	-	0
18	1574871730000	-	-	1
19	1574871731000	-	-	0
20	1574871732000	-	-	1
21	1574871733000	-	-	0
22	1574871734000	-	-	0
23	1574871735000	-	-	0
24	1574871736000	-	-	1
25	1574871737000	-	-	1
26	1574871738000	-	-	1
27	1574871739000	-	-	1
28	1574871740000	-	-	1
29	1574871741000	-	-	1

base_risk_output(Recuperado Aut

Figura 6.2: Ficheiro dos Dados de Biofeedback.

Os dados disponibilizados na *API BASE Biofeedback* incluem também informação proveniente do *eye tracker* que indica as coordenadas do ponto no ecrã para onde o programador estava a olhar quando foi registada a carga cognitiva. Isto permite que seja possível associar a carga cognitiva a *tokens*/linhas de código, mesmo que o programador não altere nada nesses tokens. Como a informação de coordenadas fornecidas pelo *eye tracker* é possível associar carga cognitiva a acções de leitura de código, como por exemplo acontece em cenários de inspeções de código. É importante notar que o plugin *Flourite* só captura acções que levem à alteração do código fonte que está a ser desenvolvido ou inspecionado.

6.1.2 Dados de Complexidade

O módulo de complexidade, ilustrado no diagrama da arquitectura do sistema 5.8, é responsável pela recolha dos dados da complexidade do código-fonte durante a programação. A recolha é feita sempre que o programador executa o código-fonte, a operação de salvar ou pela sincronização de um relógio interno mas apenas das estruturas prontas[1]. Quando lidas, as estruturas podem ser repartidas em pequenas sub-estruturas para o ajuste da precisão da avaliação de complexidade do token (ressalvamos que a métrica de complexidade utilizada é a de *MacCabe* que quantifica os caminhos possíveis da execução de um trecho de código; essa característica exigiu os trechos fossem os próprios métodos).

Depois de calculadas as métricas de complexidade é gerado um ficheiro cujos dados representam uma combinação entre o espaço, que corresponde o intervalo de linhas (ou a linha) em que se encontram os tokens, e a métrica de complexidade escolhida. A figura 6.3 ilustra um modelo do ficheiro de dados gerado mostrando as métricas da complexidade do código.

```
<complexity datestamp="03-02-2020 02:08:19" Description="Data complexity of Base
Biofeedback" lineSeparator="\n">
    <class name="Tempo">
        <filePath>!CDATA[/Users/bongocahisso/Eclipse/workspaces/laboratorio/
        Tempo.java]</filePath>
        <method name="converterTime" lineStart="20" lineEnd="44" mccabe="4" />
        <method name="main" lineStart="77" lineEnd="86" mccabe="2" />
        <method name="converterTime1" lineStart="45" lineEnd="60" mccabe="1" />
        <method name="converterTime2" lineStart="61" lineEnd="75" mccabe="1" />
        <method name="a" lineStart="92" lineEnd="95" mccabe="1" />
    </class>
</complexity>
```

Figura 6.3: Ficheiro XML dos Dados de Complexidade de uma classe.

A figura 6.4 mostra que os níveis de complexidade variam de 0 a 4 (de zero a quatro). A métrica usada para o cálculo é a de *Thomas J. McCabe*, conhecida com *Complexidade Ciclomática*[21]. Quando aplicada a métrica *McCabe* os valores da complexidade corresponde um número inteiro positivo entre 1 a $+\infty$ (um a mais infinito). Para o propósito do que deste trabalho esses valores são "*discretizados*" na escala apresentada na figura acima com o seguinte significado:

Como se pode observar nas figuras 6.3 e 6.4 a métrica usada para o cálculo é a de *Thomas J. McCabe*, também conhecida como *Complexidade Ciclomática*[21], quando aplicada os valores da complexidade corresponde um número inteiro positivo entre 1 a $+\infty$ (um a mais infinito). Para o propósito deste trabalho esses valores são *"discretizados* na escala de 1 a 6 (de um a seis) como apresentado na figura 6.4.

[1]definimos uma estrutura pronta como sendo a estrutura que tem início e fim, para linguagem de programação Java as aberturas e fechamentos das chavetas

	A	B	C
1	start_line	end_line	complexity
2	0	5	1
3	6	12	2
4	13	16	1
5	17	30	3
6	31	40	5
7	41	48	6
8	49	60	6
9	61	77	4
10	78	100	3
11	101	101	1
12	103	120	3
13	121	144	3
14	150	155	2
15	156	170	6
16	171	200	4

Figura 6.4: Ficheiro em CSV dos Dados de Complexidade do código-fonte.

Para que o protótipo seja capaz de indicar a diferença de complexidade entre dois blocos de códigos diferente os valores da *"discretização"* foram distribuídos em quatro categorias a citar:

1. A Complexidade é `Muito pouco complexo` para os valores entre 1 e 2
2. A Complexidade é `Pouco complexo` para os valores entre 3 e 4
3. A Complexidade é `Medianamente complexo` para os valores entre 5 e 6
4. A Complexidade é `Complexo` para os valores entre acima de 6

Naturalmente, a alternativa de se usar a métrica diretamente (em vez da *"discretização"*) é também uma opção viável. A avaliação futura das anotações de biofeedback com métricas complexidade determinará se é mais vantajoso usar a *"discretização"* ou o valor da *complexidade ciclomática* diretamente.

6.1.3 Dados das Operações do programador

Estes dados constituem todas as operações que o sistema de registo de eventos captura (no caso, feito pelo *plugin Fluorite*).

Como nem todos os registos dos *logs* se correlacionam com a avaliação dos riscos, no algoritmo de sincronização (presente no *Módulo Geral*) é aplicado um pré-processamento destes dados que envolve a limpeza, transformação e a redução dos dados (como ilustrado na figura 2.2). Após o pré-processamento, os dados das operações que definimos como sendo relevantes para o propósito do protótipo são:

- O tipo da operação (ilustrado na figura 5.1 ao lado direito). O tipos das operações encontram no atributo (*_type*) dos elementos *XML*;
- O número de vezes que o *token* é visitado[2] (*repeat*);
- A localização do *token* no espaço;
- E a sua granularidade (*lenght*).

6.2 Sincronização dos dados

O *"algoritmo de sincronização*[3] *"* dos dados foi desenvolvido com o objectivo de estabelecer as consistências dos dados provenientes das três fontes diferentes (*biofeedback*, complexidade e dos logs da operações). Para além da sincronização dos dados, este algoritmo é também responsável pelo pré-processamento dos dados.

A metodologia adoptada para a sincronização consistiu em `unir` os conjuntos de dados, `interpolando` registos por registos (ou seja, linha a linha) a partir de um factor de ligação (o `tempo` para o caso dos dados da complexidade com os logs das operações e os `números das linhas` para o caso dos logs das operações com os dados da complexidade).

As junções dos dados, normalmente são efectuadas por meio de um elemento comum entre os dados[4]. As três fontes de dados apresentados na secções 6.1.1, 6.1.3 e 6.1.2 possuem atributos (isto é, nome das colunas) diferentes com a excepção do atributo tempo (*timestamp*) nos dados de biofeedback e nos dados das operações do programador e as linhas nos dados das operações do programador e nos dados da complexidade do código. Isso permitiu usar como "mediador" (para a junção) os *Dados das Operações* por conter ambos os factores ou atributos:

- **Factor tempo**: Uma vez que o tempo não representa um valor preciso no que se refere a sua captura em máquinas, sistemas ou processos diferentes, o BASE definiu uma janela de tempo de ±2,5 segundos como a margem de enquadramento dos valores. E é com esse avanço ou atraso de tempo que os dados de *biofeedback* são enviados. Por esta razão, o algoritmo de sincronização compara os tempos dos *Logs das Operações* com os *Dados de Biofeedback* levando em conta esta margem.
- **Factor linha**: Representa o elemento de ligação entre os dados dos *Logs das Operações* e os *Dados da Complexidade*. Visto que as linhas é fornecida por intervalo

[2]Corresponde as alterações feitas no próprio token e nos tokens anteriores que alteram a sua posição ou significado no código.

[3]Ou Junção.

[4]Os elementos ou factores podem ser também chamadas de chaves primárias ou estrangeiras.

(como ilustrado na figura 6.4), para cada registo verificou o valor da linha e o seu encaixe no intervalo dos dados de complexidade. Significa que a complexidade de um *token* corresponde ao valor complexidade do trecho de código em que o *token* se encontra.

Resumindo: Sabendo que o *plugin Fluorite* armazena num ficheiro *XML* os dados sobre os eventos que ocorrem no IDE durante a escrita do código-fonte (como ilustrado na figura 4.1), e que a estratégia adoptada para as marcação de código define as operações de interesse e os seus níveis de classificação (ilustrado na figura 5.1), o processo de recolha dos dados resume-se em percorrer o ficheiro *XML* dos *Logs* das Operações com o objectivo de localizar os atributos correspondentes as operações definidas. Este processo é realizado pelos *Módulos de Recolha das Operações* (usando as funções de extensão do *Fluorite*) e o *Módulo Geral* (usando o algoritmo de sincronização). Os dados processados são armazenados no ficheiro de `dados sincronizados`. Ao mesmo tempo são verificadas as linhas dos `dados das Operações` com os intervalos das linhas dos `dados de complexidade`, e quando encontrado adiciona-se o valor da complexidade no `ficheiro das anotações`.

6.3 Construção dos modelos inteligentes

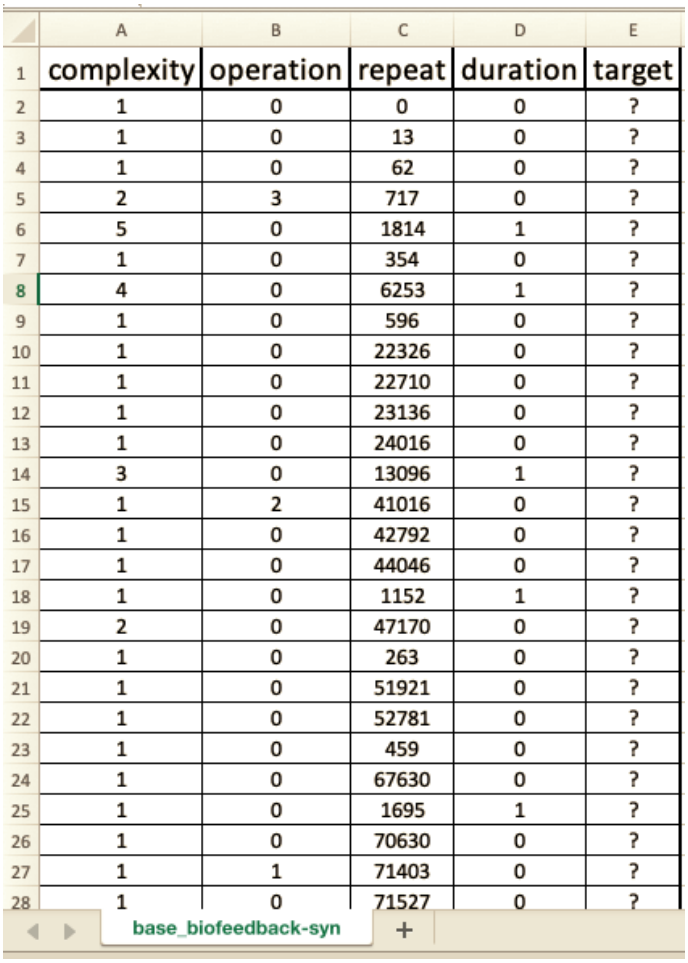
Das várias abordagens possíveis (como regras de associação, avaliação direita, classificação por grupo) para o cálculo do risco de um *token* estar sujeito a bug, aplicáveis na arquitetura ilustrada na 4.1, foi escolhida a que chamamos de *Abordagem de Construção de Perfil do Programador*. Nesta abordagem queremos que o protótipo seja capaz de continuar realizar as anotações e marcações do código-fonte mesmo para o caso que a *API Biofeedback UC* esteja indisponível, o que naturalmente significa não haver dados sobre a carga cognitiva do programador. Está é uma das motivações para a aplicação dos modelos inteligentes nesta tese, o que em verdade pode ser estendidos para outras aplicações como mencionado na secção das contribuições 1.5.

A *abordagem de construção de perfil por programador*, consiste em recolher primeiramente os **n** primeiros dados da *API Biofeedback UC*[5] como um "professor" (conhecido também como a classe principal ou *target* em *machine learning*) que "ensina" o modelo a calcular o risco do *token* usando os dados das operações do programador (o tipo de operação, duração e o número de vezes que visitou o *token*) e da complexidade do *token*.

A anotação é independente do modelo inteligente mas a marcação de código (*highlighting*) é dependente do modelo ou do conjunto de regras. Para o caso em que o modelo ainda não existir o risco é dado por um conjunto de regras que relaciona a carga cognitiva, o tipo de operação e a complexidade do código. Outros modelos poderão (e vão ser) desenvolvidos no futuro de forma a optimizar a avaliação do risco de existência de bugs em *tokens*, linhas ou excertos de código.

Os dados de entrada para o *Módulo de Aprendizagem Máquina* como ilustrado no diagrama de arquitectura (figura 4.1), origina-se do *Módulo Geral* e é denominado de *Dados de Sincronização* (ilustrado na figura 6.5) e a saída é denominado por Dados de preditos que serve para realizar o realce do código.

[5]Dados suficientemente grande Ex. $\mathbf{n} \geq c * 500$. Com $\mathbf{c} = 4$ que representa o número de classes preditoras.



	A	B	C	D	E
1	complexity	operation	repeat	duration	target
2	1	0	0	0	?
3	1	0	13	0	?
4	1	0	62	0	?
5	2	3	717	0	?
6	5	0	1814	1	?
7	1	0	354	0	?
8	4	0	6253	1	?
9	1	0	596	0	?
10	1	0	22326	0	?
11	1	0	22710	0	?
12	1	0	23136	0	?
13	1	0	24016	0	?
14	3	0	13096	1	?
15	1	2	41016	0	?
16	1	0	42792	0	?
17	1	0	44046	0	?
18	1	0	1152	1	?
19	2	0	47170	0	?
20	1	0	263	0	?
21	1	0	51921	0	?
22	1	0	52781	0	?
23	1	0	459	0	?
24	1	0	67630	0	?
25	1	0	1695	1	?
26	1	0	70630	0	?
27	1	1	71403	0	?
28	1	0	71527	0	?

Figura 6.5: Exemplo modelo dos Dados de Sincronização.

A figura 6.5 ilustra o resultado da sincronização dos dados das três fontes diferentes, denominado como *dados sincronizados*.

6.4 Modelo de Anotação e Highlighting

Nesta secção é apresentada as técnicas aplicadas para a anotação e a marcação de código.

6.4.1 Extensão do Plugin Flourite

A descrição do seu funcionamento é apresentando na secção 4.3. De acordo com o que pretendemos fazer, este *plugin* apresenta um comportamento que acaba sendo um problema. O ficheiro dos *logs* ou eventos não apresenta uma sintaxe válida para leitura e manipulação durante o seu funcionamento, mas sim depois do encerramento do IDE. Neste caso, como o protótipo em desenvolvimento possui uma dependência forte dos *logs* registados pelo *plugin Fluorite*, o cenário ou ciclo de vida seria: `Ligar e desligar o IDE`, o que causaria mais problemas do que a solução por assim dizermos.

Para solucionarmos este problema, o *plugin* teve de ser estendido. A única diferença com o modelo original, é o facto de não ser mais necessário `encerrar o IDE` para que tenhamos acesso ao ficheiro dos *logs*. A solução foi adaptar o *plugin Fluorite* para armazenar os *logs* em dois ficheiros ao mesmo tempo. Um para o *Fluorite* funcionar no seu modo normal e outro com estrutura própria e manipulável para o protótipo utilizar.

Esta solução foi encontrada depois de exploramos várias ideias e foi levado em consideração os problemas da geração de vários ficheiros e o seu tamanho. A justificação foi não alterar o conteúdo em memória do *plugin Fluorite* mas sim capturá-lo e salvá-lo num único ficheiro temporário (também conhecido como ficheiro *"Sombra"* ou *"gémeo"*). Deste modo, no próximo período da leitura (dos dados em memória do *plugin Fluorite*) encontraremos a informação completa (inclusive a lida anteriormente), o que descartou a necessidade da criação de vários ficheiros. A estrutura e o formato do ficheiro é idêntica do (formato) apresentado na figura 4.2. A secção 6.2 explica como é manipulado e utilizado os dados do ficheiro *"Sombra"*.

6.4.2 Anotação do Código

O registo de uma anotação corresponde a combinação entre a carga cognitiva do programador, o risco do *token* conter defeito, a complexidade do código, a localização do *token* e a sua granularidade (como apresentado na expressão 6.1), feita a nível do *token*.

$$Tk(n) < offset, lenght, complexity, cognitiveLoad, risk > \tag{6.1}$$

O *token* pode ser um carácter (por exemplo, o sinal "*" que representa a multiplicação na linguagem *JAVA*) ou maior ainda contendo mais do que um carácter (por exemplo, o nome de uma variável ou método). Neste protótipo fomos ainda mais além no toca a representação do *token*. Escolhemos utilizar *tokens lexical*, que são símbolos reconhecidos pelos compiladores durante a análise lexical e representam as menores unidades de código que podem ser adicionadas, modificadas ou excluídas pelos programadores[25].

A definição do uso do *tokens lexical* como unidade de anotação fornece-nos a granularidade adequada porque a mesma anotação pode ser usada para rotular qualquer parte do código de uma única linha (todos os *tokens* têm o mesmo valor de risco) ou uma região quadrada envolvendo várias linhas. Além disso, como um programador escreve código, a sequência de modificações pode envolver *tokens* únicos em diferentes linhas de código[25].

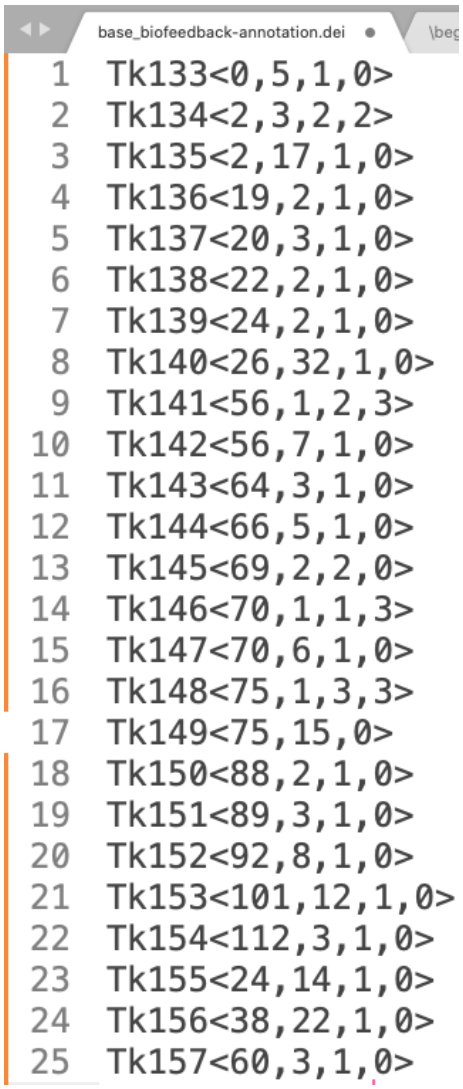


Figura 6.6: Ficheiro de anotação final

O *Módulo Geral* é responsável pela geração do ficheiro de anotação e essas anotações são armazenadas num ficheiro, não só para o uso futuro (por exemplo, servir de guia na fase dos testes do código para a fácil identificação das falhas produzidas no desenvolvimento) mas também facilitar nas mudanças do projecto de directório para directório. A sua estrutura e sintaxe é muito simples e fácil de manipular como ilustrado na figura 6.7.

6.4.3 Realce de biofeedback (Highlighting)

O conceito de destaque de *biofeedback* ou *Highlighting* consiste em colorir os *tokens* apresentados por um IDE de acordo com a anotação do código de *biofeedback* (apresentado na figura 6.6). Isso complementa a sintaxe clássica destacando e fornecendo pistas visuais para quem lê o código com o objectivo de fazê-lo conhecer as partes do código com maior risco de conter defeitos [25].

O *Módulo de Marcação* tem a função de realçar o código-fonte com base na avaliação de risco. Os riscos foram categorizados em quatro níveis de acordo a sua gravidade, onde cada [nível] equivale a um valor inteiro que corresponde a uma cor no IDE como apresentado abaixo:

- 0 - **Sem Risco** (Sem cor)
- 1 - **Risco Baixo**
- 2 - **Risco Médio**
- 3 - **Risco Elevado**

A acção realizada na presença dum determinado risco, depende do utilizador final, sendo no entanto recomendado que a avaliação seja feita na ordem descendente, isto é, do *Risco Elevado* ao caso de *Sem Risco*.

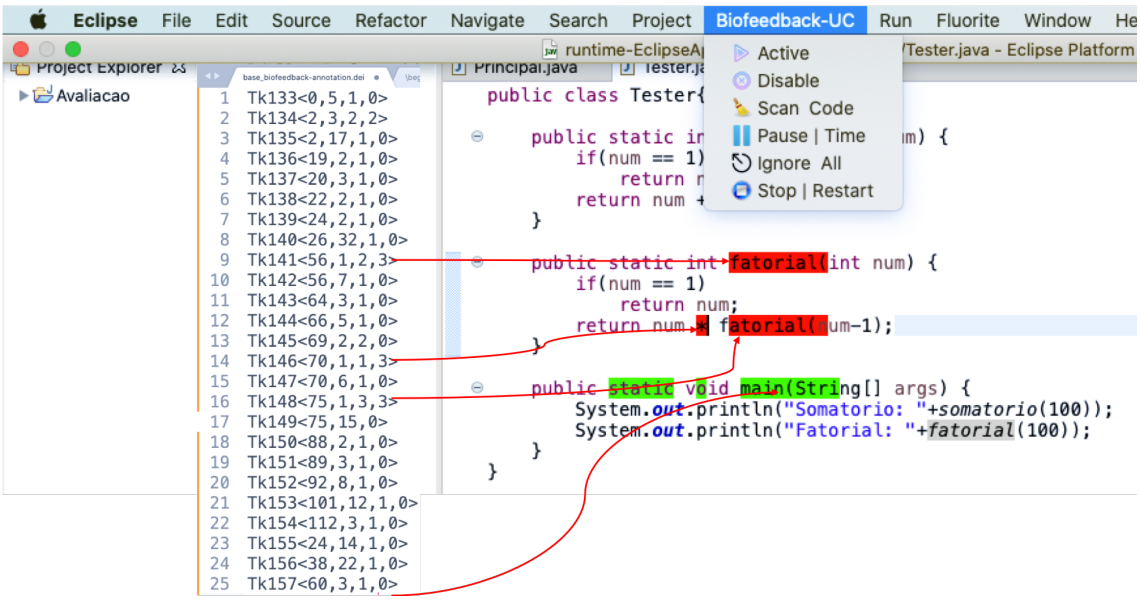


Figura 6.7: Protótipo do IDE Eclipse com a marcação e anotação do código-fonte

A figura 6.7 ilustra o resultado final depois de aplicado as abordagens apresentada ao longo do trabalho.

Capítulo 7

Conclusão e trabalho futuro

O trabalho da tese concentrou-se em grande parte na investigação das alternativas para a construção do protótipo e na definição das funcionalidades a implementar, antes de passar à construção do protótipo propriamente dito. Isso deveu-se pela natureza do problema que é totalmente nova e radical. Ainda assim foi possível cumprir as metas inicialmente traçadas e desenvolver a primeira versão do protótipo que é, por concepção, aberta e com possibilidade de evoluir. Por um lado, o trabalho realizado envolvia várias variáveis que se traduziam em incertezas na sua concretização, por outro as abordagens de que fortemente dependia o trabalho encontravam-se ainda em avaliação o que criou algumas convulsões a nível do que se devia ser feito.

O objectivo principal desta tese de mestrado foi desenvolver uma versão do IDE Eclipse para integrar os dados de biofeedback dos programadores com a finalidade de fazer anotações do código-fonte. Para além da anotação de código com os dados de *Biofeedback* dos programadores, as anotações incluem ainda a complexidade do código em que cada *token* se insere, permitindo desenvolver estratégias para indicação da possível existência de defeitos num determinado *token*. Esta funcionalidade constitui um elemento importante para o trabalho de investigação a desenvolver no projeto Engenharia de Software Aumentada de Biofeedback (BASE) (o projeto ainda nem está a meio da sua duração de 3 anos).

A Ambiente de Desenvolvimento Integrado (IDE) Eclipse estendida e a realizar anotações e marcações de código fonte foi validada com base nos dados de *biofeedback* dos programadores previamente recolhidas no projeto BASE, incluindo também as operações sobre o código e a sua complexidade em tempo real. Em contrapartida, os dados aplicados para treinar os modelos foram improvisados só para os teste e o desenvolvimento do protótipo.

Espera-se com esta solução minimizar os erros dos programadores, aumentar a produtividade na produção de software, facilitar a localização dos erros, apoia empresas produtoras de software a identificarem as principais condições que possibilitam a inserção dos erros, monitorizar as reacções fisiológicas causadas pelo estados de stress ou carga cognitiva do programador e emitir alertas em tempo real, guiar as estratégia de teste, recomendar descanso ao programador quando estiver esgotado e muitas outras vantagens que podem surgir com a analise das anotações. O protótipo construído lança as bases para o desenvolvimento, no projeto BASE, destas funcionalidades.

Como trabalho futuro perspetivamos a integração da API Biofeedback UC no próprio IDE com o objectivo de eliminar o possível problema da falha na sincronização que pode ocorrer por vários motivos, incluindo desajustes do tempo, problemas nos drivers de comunicação, ficheiro corrompido ou ligação interrompida pelo sistema de segurança do computação em

que se encontra o IDE.

Naturalmente, a qualidade das anotações e a sua capacidade de traduzir de forma fiel o estado cognitivo do programador em cada momento e de o associar a *tokens* e linhas de código depende fortemente dos sensores utilizados e das técnicas de análise de dados e de extração de *features* (essencialmente técnicas aprendizagem máquina) como as usadas nos artigo recentes do projeto BASE [25, 26, 27]. Esta parte do trabalho tem sido tratada de forma externa à construção do protótipo, mas a perspetiva de integrar a *API Biofeedback UC* no próprio IDE levará a que a geração da informação sobre carga cognitiva e a sua associação a *tokens* seja unificada, integrando a informação do *eye tracker* (que indica onde o programador está a olhar, mesmo que não esteja a editar nada no ficheiro) e do *plugin Flourite*, que regista as alterações feitas pelo programador. A utilização de outras métricas de cálculo da complexidade do código será também explorada com vista a explorar métricas compostas incluindo a *complexidade ciclomática* de *McCabe* e métricas *Halstead*, que dão uma melhor perspetiva da complexidade na perspetiva das estruturas de dados do programa em desenvolvimento.

O principal foco do trabalho futuro estará na exploração dos dados das anotações usando as várias abordagens da Inteligência Artificial (IA) para desenvolver a totalidade das funcionalidades previstas e que foram apresentadas na secção 1.5 desta tese.

Referências

[1] B. Bin Liu A. Fuqun Huang and C. Bing Huang. A taxonomy system to identify human error causes for software defects. In *18th ISSAT International Conference on Reliability and Quality in Design*, 2012.

[2] T. Santander B. Floyd and W. Weimer. Decoding the representation of code in the brain. In *An fMRI Study of Code Review and Expertise, ICSE 2017, pp 175–186*, Piscataway, NJ, USA, 2017.

[3] Alexander Budzier Bent Flyvbjerg. Why your it project may be riskier than you think. In *Harvard Business Review*, Harvard, 2011.

[4] Peter J. Burke. Identity processes and social stress. In *Revista Sociológica Americana, American Sociological Association p 836-849*, Washington State University, 1991.

[5] J. Christmansson and R. Chillarege. Generation of an error set that emulates sw faults. In *Proc. of the 26th International Fault Tolerant Computing Symposium*, FTCS-26, Sendai, Japan, 1996.

[6] Ed Burnette David Gallardo and Robert McGovern. Eclipse in action. In *A Guide for Web Developers*, Book, 2003.

[7] Estelbina Miranda de Alvarenga. Metodologia da investigação quantitativa e qualitativa. In *Normas técnicas de apresentação de trabalhos científicos - 2^a ed*, Assunção, Paraguai, 2012.

[8] Professor António Dourado. Disciplina de aprendizagem computacional. In *@ADC/20018-2019/FCTUC/AC/MEI/CNSD/MIEB/Chapt1IML*, Faculdade de Ciências da Universidade de Coimbra, Ano lectivo 2018/2019.

[9] J. Durães and H. Madeira. Emulation of sw faults. In *A Field Data Study and a Practical Approach, IEEE Transactions on SW Engineering, vol. 32, no. 11, pp. 849-867*, November 2006.

[10] Chris Aniszczyk e David Gallardo. Introdução à plataforma eclipse. In *Use os plug-ins para editar, compilar, depurar e atuar como uma base para seus aplicativos*, Article published by IBM, 2012.

[11] Paul Deitel e Harvey Deitel. Java. In *Como programar - 10^a ed*, Câmara Brasileira do Livro, SP, Brasil, 2017.

[12] Lan H. Witten Christopher J. Pal Eibe Frank, Mark A. Hall. Data mining. In *Practical Machine Learning Tools and Techniques - 4rd ed.*, 2016.

[13] Bianchi ERF. Conceito de stress. In *evolução histórica*, Nursing (São Paulo), 2001.

[14] Michael Fagan. Reviews and inspections. In *Conference 2001, Software Pioneers 215 Eds.: M. Broy, E. Denert, Springer 2002*, Palo Alto, California - USA, 2000.

[15] David Gallardo. Developing eclipse plug-ins. In *How to create, debug, and install your plug-in*, Article published by IBM, 2002.

[16] Fuqun Huang. Human error analysis in software engineering. In *chapter of the book "Theory and Application on Cognitive Factors and Risk Management*, F. Felice and A. Petrillo Editors, IntechOpen, 2017.

[17] Jian Pei Jiawei Han, Micheline Kamber. Data mining. In *Concepts and Techniques - 3rd ed.*, 2011.

[18] OTI Jim Amsden, OTI updated by Andrew Irvine. Your first plug-in. In *Developing the Eclipse "Hello World" plug-in*, Article published by IBM, 2003.

[19] Günther Sagl Andreas Petutschnig Christian Werner David Niederseer Michael Liedlgruber Frank Wilhelm Tess Osborne Kalliopi Kyriakou, Bernd Resch and Jessica Pykett. Detecting moments of stress from measurements of wearable physiological sensors. In *Sensors, 19(17), 3805*, https://doi.org/10.3390/s19173805, 2019.

[20] Anthony J. Lattanze. Architecting software intensive systems. In *A Practitioner's Guide*, Boca Raton London New York, 2009.

[21] Thomas J. McCabe. A complexity measure. In *IEEE Transactions on Software Engineering Vol. 2*, Institute for Advanced Technology, the University of California, December 1976.

[22] Steve McConnell. In *A Practical Handbook of Software Construction*, Microsoft Press, 2004.

[23] S. Yamada N. Honda. Empirical analysis for high quality sw development. In *Ameri. Jour. Op. Research*, 2012.

[24] J. Siegmund N. Peitek. A look into programmers' heads. In *IEEE Transactions on Software Engineering*, August, 2018.

[25] J. Durães J. Castelhano C. Duarte C. Teixeira M. Castelo Branco P. Carvalho H. Madeira R. Couceiro, G. Duarte. Pupillography as predictor of programmers' mental effort and cognitive overload. In *The 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019*, International Conference on Software Engineering, New Ideas and Emerging Results, ICSE, June 2019.

[26] J. Durães J. Castelhano C. Duarte C. Teixeira M. Castelo Branco P. Carvalho H. Madeira R. Couceiro, G. Duarte. Biofeedback augmented software engineering. In *monitoring of programmers' mental effort*, International Conference on Software Engineering, New Ideas and Emerging Results, ICSE, May 2019.

[27] J. Durães J. Castelhano C. Duarte C. Teixeira M. Castelo Branco P. Carvalho H. Madeira R. Couceiro, G. Duarte. Spotting problematic code lines using nonintrusive programmers' biofeedback. In *International Symposium on Software Reliability Engineering (ISSRE 2019)*, Berlin, October 2019.

[28] James Reason. Human error. Cambridge University Press, 1990.

[29] M. Torchiano S. Shah, M. Morisio. The impact of process maturity on defect density. In *ACM-IEEE International Symposium on on Empirical Software Engineering and Measurement*, 2012.

[30] Bert Schumm Johannes Marca Roberto Tröster Gerhard Ehlert Ulrike Setz, Cornelia Arnrich. Discriminating stress from cognitive load using a wearable eda device. In *IEEE Transactions on Information Technology in Biomedicine*, 2010.

[31] et al T. Nakagawa, Y. Kamei. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement. In *A Controlled Experiment", Proc. of ICSE*, 2014.

[32] et. Al V. Anu. Using a cognitive psychology perspective on errors to improve requirements quality. In *An Empirical Investigation*, IEEE 27th International Symposium on Software Reliability Engineering, 2016.

Appendices

Apêndice A

Módulo Geral

```
public static String getDuration(String timestamp1, String timestamp2) {
    if (timestamp2.isEmpty()) {
        return timestamp1.length() == 0 ? "0" : timestamp1;
    }
    long time1 = Integer.parseInt(timestamp1);
    long time2 = Integer.parseInt(timestamp2);
    long dur = (time2 - time1);
    String s = String.valueOf(dur);
    return (s.length() == 0) ? "0" : s;
}

/**
 * Verifica a gravidade com base na duração da operação
 *
 * @param data
 * @param timestamp
 * @param tG
 * @return String - a diferença de tempo
 */
public static String getRisk(List<String[]> data, String timestamp, long tG) {
    int index = -1;
    if (timestamp.isEmpty()) {
        return "0";
    }
    Long timeHRV;
    long menorTime = 12500; // JANELA
    long dista;

    String input;
    long time = Integer.parseInt(timestamp) + tG;
    for (int i = 1; i < data.size() - 1; i++) {
        input = data.get(i)[0];
        if (!"#VALUE".equalsIgnoreCase(input)) {
            timeHRV = new Long(input);
            if ((time - PropertyFilesBiofeedback.TIME_WINDOWS) <=
                    timeHRV && (time + PropertyFilesBiofeedback.TIME_WINDOWS) >= timeHRV) { // Encontrar o tempo
                dista = distance(time, timeHRV);
                if (menorTime > dista) {
                    menorTime = dista;
                    index = i;
                }
            }
        }
    }
    return -1 == index ? "0" : data.get(index)[1];
}

public static long distance(long x, long y) {
    return Math.abs(x - y);
}
```

Figura 1: Algoritmo de Sincronização Usando a janela de tempo de 2.5 segundos

```
public static synchronized void initAll() {
    try {
        String filename = getLastFile(PropertyFilesBiofeedback.DIR_FILE_FLOURITE); // Pega o último ficheiro
        if (null != filename) {
            filename = PropertyFilesBiofeedback.DIR_FILE_FLOURITE + "/" + filename;
            String[] tags = {"Events", "Command", "DocumentChange"}; // Configura os atributos de interesse
            writerLogsXml(readerLogsXml(filename), PropertyFilesBiofeedback.FILE_FLOURITE_ACTIVE);        // Prepara o fic

            List<String[]> data;
            data = synchronizerFiles(PropertyFilesBiofeedback.FILE_FLOURITE_ACTIVE, tags);               // sincroniza os
            if (!DATA_FOR_CLASSIFY.isEmpty()) {
                if (writeDataAtOnce(data)) {
                    System.err.println(Arrays.toString(data.get(0))+" :DIR_FILE_OUT_SYN: "+Arrays.toString(data.get(1)));
                    APIWeka.converterCSVtoARFF(PropertyFilesBiofeedback.DIR_FILE_OUT_SYN);
                } else {
                    System.err.println("Erro ao escrever o ficheiro .csv");
                }
            }
        }

        String[] targets = APIWeka.runBiofeedbackIA();
        if (null != DATA_FOR_ANNOTATION && null != targets) {
            String tokens = "";
            for (int t = 1; t < DATA_FOR_ANNOTATION.size(); t++) {
                String offset = DATA_FOR_ANNOTATION.get(t)[1];
                String size = DATA_FOR_ANNOTATION.get(t)[0];
                if (!offset.isEmpty() && !size.isEmpty()) {
                    tokens += "Tk" + t + "<" + offset + "," + size + "," + targets[t - 1] + ">\n";
                }
            }
            Exemplo.escritaArquiv(tokens, PropertyFilesBiofeedback.PATH_FILE_ANNOTATION);
            Thread.sleep(100);
            new Exemplo().ActiveHighlighting();
        }
    } catch (Exception e) {
    }

}
```

Figura 2: Algoritmo de Sincronização - Usando a API Biofeedback UC

Algoritmo de Sincronização

```
private static List<String[]> synchronizerFiles(String filename, String[] tags) throws Exception {
    File fXmlFile = new File(filename);
    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    Document doc = dBuilder.parse(fXmlFile);
    DATA_FOR_CLASSIFY.clear();
    DATA_FOR_ANNOTATION.clear();
            /**
             * Passos: 1. Ler o ficheiro de riscos .csv 2. Ordenar em função
             * do tempo 3. algorimo de busca do risco correspondente tendo
             * em conta a margem de 12.5s
             */
    List<String[]> dataHrv = readerDataCSV(PropertyFilesBiofeedback.DIR_FILE_HRV);
    long timeG = 0;
    DATA_FOR_ANNOTATION.add(0, new String[]{"length", "offset"});
    DATA_FOR_CLASSIFY.add(0, new String[]{"hrv_risck", "operation", "repeat", "duration", "target"});
    for (String tag : tags) {
        NodeList nList = doc.getElementsByTagName(tag);

        for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                Element eElement = (Element) nNode;
                String offset;

                if (tag.equalsIgnoreCase("Events")) {
                    timeG = new Long(eElement.getAttribute("startTimestamp"));
                }

                if (tag.equalsIgnoreCase("Command")) {
                    offset = eElement.getAttribute("docOffset");
                } else {
                    offset = eElement.getAttribute("offset");
                }
                DATA_FOR_ANNOTATION.add(new String[]{
                    eElement.getAttribute("length"),
                    offset,});

                DATA_FOR_CLASSIFY.add(new String[]{
                    getRisk(dataHrv, eElement.getAttribute("timestamp"), timeG), // Adicionar o tempo de inicio
                    getOperation(eElement.getAttribute("_type")),
                    getDuration(eElement.getAttribute("timestamp"), eElement.getAttribute("timestamp2")),
                    getRepeat(eElement.getAttribute("repeat")),
                    "?"
                });
            }
        }
    }
    return DATA_FOR_CLASSIFY;
}
```

Figura 3: Algoritmo de Sincronização - Pré-processamento dos dados

```
private void highlight(IEditorPart editor, ITextViewer viewer) {
    IPreferenceStore store = Activator.getDefault().getPreferenceStore();
    if (!store.getBoolean(PreferenceConstants.P_ENABLE)) {
        log("Highlight is not enabled");
        return;
    }
    IFileEditorInput ife = (IFileEditorInput) editor.getEditorInput().getAdapter(IFileEditorInput.class)
    String ext = ife.getFile().getFileExtension();

    String config = PreferenceConstants.getConfigForExt(ext);
    String val = store.getString(config);
    if (val == null || val.length() == 0) {
        config = PreferenceConstants.getConfigForExt("*");
    }
    if (!store.getBoolean(config)) {
        log("Highlight is not enabled for ext " + config);
        return;
    }

    StyledText text = viewer.getTextWidget();
    if ((text == null) || text.isDisposed()) {
        return;
    }

    clear(viewer);

    String stext = text.getText();
    int maxFileSize = store.getInt(PreferenceConstants.P_MAX_SIZE_IN_MB) * 1024 * 1024;
    if (maxFileSize != 0 && stext.length() > maxFileSize) {
        log("Document size " + stext.length() + " bytes is greater than allowed max size "
                + store.getInt(PreferenceConstants.P_MAX_SIZE_IN_MB) + "mb");
        return;
    }

    log("Highlighting doing");

    ArrayList<Annotation> ofs = Exemplo.readerAnnotationFile();

    for(int ii = 0; ii < ofs.size(); ii++) {
        IRegion region = new Region(ofs.get(ii).getOffset(), ofs.get(ii).getLenght());
        int offset = region.getOffset();
        int length = region.getLength();

        TreeSet<StyleRange> newStyleRanges = new TreeSet<StyleRange>(new Comparator<StyleRange>() {

            @Override
            public int compare(StyleRange o1, StyleRange o2) {
                if (o1.start < o2.start)
                    return -1;
                return 1;
```

Figura 4: Algoritmo de Marcação de Código 1 de 3

```
                return 1;
            }
        });

        int cc = ofs.get(ii).getCor();
        bColor = null;
        StyleRange[] oldStyleRanges = text.getStyleRanges(offset, length);

        for (int i = 0; i < oldStyleRanges.length; i++) {
            StyleRange oldStyleRange = oldStyleRanges[i];
            Color foregroundColor = getForegroundColor(text, oldStyleRange);
            Color backgroundColor = getBackgroundColor(text, oldStyleRange,cc);

            newStyleRanges.add(
                    new StyleRange(oldStyleRange.start, oldStyleRange.length, foregroundColor, backgroundColor));
        }

        int end = offset + length;
        NEXT_CHAR: for (int i = offset; i < end; i++) {
            for (StyleRange sr : newStyleRanges) {
                if (i < sr.start) {
                    newStyleRanges.add(new StyleRange(i, sr.start, getForegroundColor(text, null),
                            getBackgroundColor(text, null,cc)));

                    i = sr.start + sr.length - 1;
                    continue NEXT_CHAR;
                }
                if (sr.start <= i && i < sr.start + sr.length) {
                    // in range:
                    i = sr.start + sr.length - 1;
                    continue NEXT_CHAR;
                }
            }


            newStyleRanges.add(
                    new StyleRange(i, end - i, getForegroundColor(text, null), getBackgroundColor(text, null, cc)));
            break;
        }

        text.replaceStyleRanges(offset, length, newStyleRanges.toArray(new StyleRange[newStyleRanges.size()]));

        for (StyleRange sr : newStyleRanges)
            text.redrawRange(sr.start, sr.length, true);

        viewersWithStyleChange.add(viewer);
    }
}
```

Figura 5: Algoritmo de Marcação de Código 2 de 3

```
private Color getBackgroundColor(StyledText text, StyleRange oldStyleRange, int c) {
    if (bColor == null) {
        IPreferenceStore store = Activator.getDefault().getPreferenceStore();
    switch(c) {
        case 3: // Vermelho
        bColor = new Color(Display.getCurrent(),255,0,0); break;
        case 2: // Verde
            bColor = new Color(Display.getCurrent(),0,255,0); break;
        case 1: // Amarela ou laranja
            bColor = new Color(Display.getCurrent(),255,255,0); break;
        case 0: // Sem cor
            bColor = new Color(Display.getCurrent(),255,255,255); break;
        default:
            bColor = new Color(Display.getCurrent(),
                    PreferenceConverter.getColor(store, PreferenceConstants.P_BACKGROUND_COLOR)); break;
        }
    }
    return bColor;
}
```

Figura 6: Algoritmo de Marcação de Código 3 de 3

Módulo de Recolha das Operações

```
public static String getLastFile(String path) {
    FileFilter filter = (File file) -> file.getName().endsWith(".xml");
    ArrayList<String> fich = new ArrayList<>();
    File dir = new File(path);
    try {
        File[] files = dir.listFiles(filter);
        for (File file : files) {
            fich.add(file.getName());
        }
        Collections.sort(fich);
        return (fich.get(fich.size() - 1));
    } catch (Exception e) {

    }
    return null;
}
```

Figura 7: Extensão do Plugin Fluorite

Módulo de Machine Learning

```
1  package pt.uc.fct.dei.biofeedback;
2
3  import java.io.BufferedReader;
38
39 /**
40  *
41  * @author bongocahisso
42  */
43 public class APIWeka {
44
45     private static Instances ins = null;
46     private static double prob[] = null;
47     private static final int CLASSE[] = {0, 1, 2, 3};
48
49     public static Instances loadDataset() {
50         if (null == ins && Exemplo.fileExists(PropertyFilesBiofeedback.PATH_FILE_TRAIN)) {
51             try {
52                 DataSource ds = new DataSource(PropertyFilesBiofeedback.PATH_FILE_TRAIN);
53                 ins = ds.getDataSet();
54                 ins.setClassIndex(ins.numAttributes() - 1);
55             } catch (Exception ex) {
56                 Logger.getLogger(APIWeka.class.getName()).log(Level.SEVERE, null, ex);
57             }
58         }
59         return ins;
60     }
61
62     public static void evaluteModel(Vote model) {
63         if (null != model && null != ins) {
64             try {
65                 Evaluation evol = new Evaluation(ins);
66                 evol.evaluateModel(model, ins);
67                 System.out.println(evol.toSummaryString());
68             } catch (Exception e) {
69                 Logger.getLogger(APIWeka.class.getName()).log(Level.SEVERE, null, e);
70             }
71         } else {
72             System.err.println("Impossivel avaliar porque o modelo ou a instância dos dados
73         }
74
75     }
76
77     public static int predite() {
78         int classe = 0;
79         int index = 0:
```

Figura 8: Carregamento dos dados e Avaliação Modelo

```
public static Vote agragateClassify(Instance novo) {
    Vote vote = null;
    try {
        /**
         * AdaBoost
         */
        AdaBoostM1 m1 = new AdaBoostM1();
        m1.setClassifier(new DecisionStump());
        m1.setNumIterations(100);
        m1.buildClassifier(ins);

        /**
         * bagging
         */
        Bagging bagging = new Bagging();
        bagging.setClassifier(new RandomTree());
        bagging.setNumIterations(25);
        bagging.buildClassifier(ins);

        /**
         * stacking
         */
        Stacking stack = new Stacking();
        stack.setMetaClassifier(new Logistic());
        weka.classifiers.Classifier[] classifiers = {new J48(), new NaiveBayes(),
            new RandomForest(), new SMO(), new MultilayerPerceptron()
        };
        stack.setClassifiers(classifiers);
        stack.buildClassifier(ins);

        /**
         * voting
         */
        vote = new Vote();
        vote.setClassifiers(classifiers);
        vote.buildClassifier(ins);

        vote.classifyInstance(novo);
        prob = vote.distributionForInstance(novo);
        predite();
    } catch (Exception e) {
    }
    return vote;
}
```

Figura 9: Construção dos Modelo Inteligentes

```java
/**
 * Classifica os metadados recolhido e enviado via file
 * (biofeedback-for-classify), usando o modelo de votação
 *
 * @param vote modelo combinado de vários classificadores
 * @return boolean true - caso classifique e gera uma saída do ficheiro
 * biofeedback-annotation. false caso contrário
 *
 */
public static boolean classifyFromFile(Vote vote) {

    try {
        if (Exemplo.fileExists(PropertyFilesBiofeedback.PATH_FILE_FOR_CLASSIFY)) {
            Instances newtestdata = null;
            /**
             * load training and testing data
             */
            DataSource source2 = new DataSource(PropertyFilesBiofeedback.PATH_FILE_FOR_CLASSIFY);
            Instances testdata = source2.getDataSet();
            testdata.setClassIndex(testdata.numAttributes() - 1);

            /**
             * filter the data and output the prediction to test file
             */
            AddClassification addClass = new AddClassification();
            addClass.setClassifier(vote);
            addClass.setRemoveOldClass(true);
            addClass.setOutputClassification(true);
            addClass.setInputFormat(ins);
            Filter.useFilter(ins, addClass);
            newtestdata = Filter.useFilter(testdata, addClass); // Encontro
            return saveFileClassify(newtestdata);
        } else {
            System.out.println("Dados para marcação inexistente");
        }
    } catch (Exception ex) {
        Logger.getLogger(APIWeka.class.getName()).log(Level.SEVERE, null, ex);
    }
    return false;
}
```

Figura 10: Cálculo do risco reutilizado o modelo inteligente

```
public static String[] getTarget() {
    try {
        DataSource source = new DataSource(PropertyFilesBiofeedback.PATH_FILE_OUT);
        Instances data = source.getDataSet();
        int numInst = data.numInstances();
        String[] tarNames = new String[numInst];
        if (data.classIndex() == -1) {
            data.setClassIndex(data.numAttributes() - 1);
        }
        for (int j = 0; j < numInst; j++) {
            Instance instance = data.instance(j);
            double cv = instance.classValue();
            tarNames[j] = instance.classAttribute().value((int) cv);
              System.out.println(instance.classAttribute().value((int) cv));
        }
        return tarNames;
    } catch (Exception ex) {
        Logger.getLogger(APIWeka.class.getName()).log(Level.SEVERE, null, ex);
    }
    return null;
}

public static void converterCSVtoARFF(String pathFileCSV) {
    try {
        CSVLoader loader = new CSVLoader();
        loader.setSource(new File(pathFileCSV));
        Instances data = loader.getDataSet();
        ArffSaver saver = new ArffSaver();
        saver.setInstances(data);
        saver.setFile(new File(PropertyFilesBiofeedback.PATH_FILE_FOR_CLASSIFY));
        saver.writeBatch();
        ajustFileArff();
    } catch (IOException ex) {
        Logger.getLogger(APIWeka.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Figura 11: Busca dos riscos e Criação ficheiro Dados de Sincronização

```
public static Vote agragateClassify() {
    Vote vote = null;
    try {
        /**
         * AdaBoost
         */
        AdaBoostM1 m1 = new AdaBoostM1();
        m1.setClassifier(new DecisionStump());
        m1.setNumIterations(100);
        m1.buildClassifier(ins);

        /**
         * bagging
         */
        Bagging bagging = new Bagging();
        bagging.setClassifier(new RandomTree());
        bagging.setNumIterations(25);
        bagging.buildClassifier(ins);

        /**
         * stacking
         */
        Stacking stack = new Stacking();
        stack.setMetaClassifier(new Logistic());
        weka.classifiers.Classifier[] classifiers = {new J48(), new NaiveBayes(),
            new RandomForest(), new SMO(), new MultilayerPerceptron()
        };
        stack.setClassifiers(classifiers);
        stack.buildClassifier(ins);

        /**
         * voting
         */
        vote = new Vote();
        vote.setClassifiers(classifiers);
        vote.buildClassifier(ins);
    } catch (Exception e) {
    }
    return vote;
}
```

Figura 12: Construção de um modelo conjunto

Formato do ficheiro disponibilizado pela API Biofeedback UC

```
<?xml version="1.0" encoding="UTF-8"?>
    <biofedback version="1.0"? osName="Mac OS X" date="">
        <Event id="" timestamp="">
            <pup coodX="" coodY="" size="" risco=>
            <hrv picoR1="" picoR1="" dist="">
            <log line="" offset="" size="">
        </Event>
        .
        .
        .
    </biofedback>
</xml>
```

Figura 13: Estrutura do ficheiro disponibilizado pela API Biofeedback UC