

UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Luiz Laerte Nunes da Silva Junior
Thiago Nazareth de Oliveira

Vertical Code Completion

Niterói
2010

Luiz Laerte Nunes da Silva Junior

Thiago Nazareth de Oliveira

Vertical Code Completion

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Orientadores: Leonardo Gresta Paulino Murta

Co-orientador: Alexandre Plastino de Carvalho

Niterói

2010

Luiz Laerte Nunes da Silva Junior

Thiago Nazareth de Oliveira

Vertical Code Completion

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovado em Julho de 2010

BANCA EXAMINADORA

Prof. Leonardo Gresta Paulino Murta, D.Sc.
Orientador
UFF

Prof. Alexandre Plastino de Carvalho, D.Sc.
Co-Orientador
UFF

Prof. Teresa Cristina de Aguiar, D.Sc.
UFF

Prof. Bianca Zadrozny, D.Sc.
UFF

Niterói
2010

RESUMO

No domínio do desenvolvimento de sistemas, a quantidade de dados envolvida, relativa a documentações e ao próprio código fonte, é muito grande. Muito conhecimento importante pode ser extraído deste montante de dados, desde que as ferramentas adequadas sejam utilizadas. Neste contexto, a mineração de dados se apresenta como uma das principais ferramentas disponíveis.

Este trabalho visa minerar padrões sequenciais frequentes, presentes nos repositórios de código fonte, e em seguida sugerir esses padrões aos programadores, de acordo com o que está sendo codificado no momento. Como resultado, um plugin para a IDE Eclipse, chamado Vertical Code Completion, foi desenvolvido e aplicado no repositório de código fonte do sistema IdUFF. Os resultados foram analisados por desenvolvedores desse sistema com diferentes níveis de experiência, e suas percepções são apresentadas ao final desta monografia.

Palavras Chave:

Engenharia de Software, Code Completion, Mineração de Dados, Mineração de Padrões Sequenciais.

ABSTRACT

In system development area, the amount of data involved on documentations and source code is very large. Very important knowledge can be extracted from this amount of data, provided that adequate tools are used. In this context, the data mining is presented as one of the main tools available.

This paper aims to mine frequent sequential patterns present in source code repositories, and then suggest these patterns to developers, according to what is being encoded at this time. As a result, a plugin for the Eclipse IDE, called Vertical Code Completion, was developed and applied in the source code repository system IdUFF. The results were analyzed by developers of this system with different levels of experience, and their perceptions are included at the end of this monograph.

Keywords:

Software Engineering, Code Completion, Data Mining, Sequence Mining.

LISTA DE ACRÔNIMOS

VCC:	<i>Vertical Code Completion</i>
IDE:	<i>Integrated Development Environment</i>

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO	6
CAPÍTULO 2 - REVISÃO DA LITERATURA	8
2.1 Mineração de Dados	8
2.1.1 Classificação	9
2.1.2 Regras de Associação	9
2.1.3 Clusterização	9
2.1.4 Padrões em Séries Temporais	10
2.1.5 Padrões Sequenciais	10
2.2 Mineração de Dados na Engenharia de Software	11
CAPÍTULO 3 - VERTICAL CODE COMPLETION	13
3.1 Obtenção de Padrões Frequentes de Codificação de Software.	13
3.1.1 Análise do Código Fonte	13
3.1.2 Mineração do Código Fonte	14
3.1.3 Geração de Árvore de Chamadas	16
3.2 Sugestão de Padrões Frequentes de Código Fonte	18
CAPÍTULO 4 - O <i>PLUGIN</i> VCC	21
4.1 A Análise do Código Fonte Através da ASTParser	22
4.2 Mineração do Código Fonte	23
4.3 Árvore de Padrões Frequentes	24
4.4 Geração e Poda das Combinações de Chamadas de Métodos	25

CAPÍTULO 5 - RESULTADOS EXPERIMENTAIS	27
5.1 O Sistema Objeto de Estudo - IdUFF	27
5.2 Planejamento do Experimento	28
5.3 Aplicação do Experimento	28
5.4 Resultados Obtidos	30
5.4.1 Análise Quantitativa	30
5.4.2 Análise Qualitativa	34
5.5 Ameaças à Validade	35
CAPÍTULO 6 - CONCLUSÕES	37
REFERÊNCIAS BIBLIOGRÁFICAS	38
APÊNDICE I	40
APÊNDICE II	42
APÊNDICE III	44

LISTA DE FIGURAS

FIGURA 1: FLUXO DE USO DO VCC.	14
FIGURA 2: EXEMPLO DE CODIFICAÇÃO DE MÉTODO E A SEQUÊNCIA GERADA.	15
FIGURA 3: EXEMPLO DE ÁRVORE DE PADRÕES FREQUENTES.	16
FIGURA 4: EXEMPLO DE ÁRVORE DE PADRÕES FREQUENTES COM SUPORTES E CONFIANÇAS.	18
FIGURA 5: MENU DO PLUGIN VCC	22
FIGURA 6: ESTRUTURA DE UM WORKSPACE JAVA.	23
FIGURA 7: CHAMADA DO USUÁRIO COM O INTUITO DE OBTER AS SUGESTÕES DE CÓDIGO FONTE.	25
FIGURA 8: SUGESTÕES BASEADAS NOS PADRÕES SEQUENCIAIS FREQUENTES.	26
FIGURA 9: TOTAL DE RESPOSTAS DOS PARTICIPANTES PARA CADA ITEM.	31
FIGURA 10: TOTAL DE RESPOSTAS, POR ITEM, DOS PARTICIPANTES EXPERIENTES NAS QUESTÕES COM PADRÕES DE CONFIANÇA ALTA.	32
FIGURA 11: TOTAL DE RESPOSTAS, POR ITEM, DOS PARTICIPANTES NÃO EXPERIENTES NAS QUESTÕES COM PADRÕES DE CONFIANÇA ALTA.	32
FIGURA 12: TOTAL DE RESPOSTAS, POR ITEM, DOS PARTICIPANTES EXPERIENTES NAS QUESTÕES COM PADRÕES DE CONFIANÇA BAIXA.	33
FIGURA 13: TOTAL DE RESPOSTAS, POR ITEM, DOS PARTICIPANTES NÃO EXPERIENTES NAS QUESTÕES COM PADRÕES DE CONFIANÇA ALTA.	33

LISTA DE TABELAS

TABELA 1: RESUMO DO QUESTIONÁRIO DE CARACTERIZAÇÃO DOS PARTICIPANTES.	29
TABELA 2: QUESTÕES COM AS SUGESTÕES DO QUESTIONÁRIO DE AVALIAÇÃO	30
TABELA 3: RESUMO DAS RESPOSTAS DE CADA QUESTÃO, EM PORCENTAGEM.	34

CAPÍTULO 1 - INTRODUÇÃO

Uma das preocupações da Engenharia de Software consiste na busca por melhores resultados em termos de produtividade e qualidade durante o desenvolvimento de software [10]. Diversas técnicas e ferramentas foram criadas com esse propósito, e muitas delas utilizam o próprio conhecimento produzido durante o desenvolvimento para apoiar os desenvolvedores nas suas tarefas [8].

Dentre essas ferramentas, as de *code completion*, que são adotadas em praticamente todas as IDEs (*Integrated Development Environment*) utilizadas atualmente [16], analisam a estrutura sintática do software e sugerem o preenchimento automático de código durante a atividade de programação. Por exemplo, ao iniciar a codificação em Java da chamada de um método com “System.id”, a IDE automaticamente termina a codificação, ficando “System.identityHashCode”, tendo em vista que esse é o único método da classe System iniciado por “id”. Este apoio gera efeitos positivos na produtividade dos programadores e os encoraja, por exemplo, a usar nomes de variáveis mais descritivos, melhorando a qualidade do código produzido.

Todavia, as ferramentas convencionais de *code completion* analisam somente a estrutura sintática do software, sugerindo o término da codificação de um elemento (i.e., classe, método, atributo, etc.) em função do casamento perfeito entre o que já foi digitado e o início de algum nome de elemento disponível na IDE. Além disso, essa sugestão se restringe ao elemento em questão, exigindo que o programador inicie a codificação da linha seguinte para que novas sugestões sejam fornecidas. Essas limitações motivam a elaboração de abordagens que saiam da análise puramente sintática e forneçam sugestões mais completas e elaboradas em relação ao código em desenvolvimento.

Desta forma, o objetivo desse trabalho é propor uma nova abordagem para *code completion*, complementar à existente, mas que atue de forma mais abrangente, sugerindo sequências de código frequentes. Para isso, são utilizados algoritmos de mineração de dados [7] que analisam o software como um todo e identificam padrões sequenciais recorrentes. Durante a atividade de codificação, as linhas já codificadas são confrontadas com os padrões sequenciais previamente identificados e caso haja similaridade, o restante do padrão sequencial é automaticamente codificado. Por exemplo, ao iniciar a codificação com “BD.openConnection()”,

poderia ser sugerida a sequência de código contendo “BD.beginTransaction()”, “BD.commit()” e “BD.closeConnection”, desde que essa sequência ocorra com frequência em outras partes do software.

Essas sugestões podem aumentar a produtividade do desenvolvedor assim como evitar o surgimento de erros, devido ao questionamento natural que o desenvolvedor irá fazer sempre que uma informação pertinente, que não se relaciona com o que estava para ser desenvolvido, for sugerida no decorrer da atividade. Contudo, esses benefícios esperados são fortemente dependentes da utilidade das sugestões fornecidas. Desta forma, um protótipo utilizando a IDE Eclipse foi implementado e a utilidade das sugestões foi avaliada em um experimento junto à equipe de desenvolvimento do sistema IdUFF. O experimento mostrou resultados positivos, indicando que 71,6% das sugestões foram pertinentes.

O restante deste trabalho está organizado da seguinte forma.

O Capítulo 2 apresenta uma introdução à área de mineração de dados, descrevendo e exemplificando as suas tarefas: extração de regras de associação, classificação, clusterização e extração de padrões sequenciais, sendo esta última a técnica usada neste trabalho para extrair sugestões de código. Além disso, são apresentados alguns trabalhos que aplicam mineração de dados na área de Engenharia de Software.

O Capítulo 3 apresenta uma visão geral da abordagem proposta neste trabalho, descrevendo em alto nível os passos tomados para se chegar à solução proposta.

O Capítulo 4 discute os detalhes de implementação da abordagem proposta, descrevendo técnicas, ferramentas e tecnologias utilizadas.

O Capítulo 5 apresenta os resultados experimentais obtidos, descrevendo o planejamento do experimento bem como a avaliação e discussão dos resultados.

Finalmente, o Capítulo 6 apresenta a conclusão deste trabalho, relatando as suas contribuições, limitações e possíveis trabalhos futuros.

CAPÍTULO 2 - REVISÃO DA LITERATURA

Neste capítulo, são abordados alguns conceitos e técnicas de Mineração de Dados, além da aplicação dessas técnicas em problemas de Engenharia de Software, citando trabalhos desenvolvidos nessa área.

2.1 MINERAÇÃO DE DADOS

A evolução computacional das últimas décadas, ocasionada pela evolução tecnológica, permitiu um grande aumento no poder de processamento e na capacidade de armazenamento de dados a baixo custo, inserindo no mercado novas tecnologias de transmissão e disponibilização de dados [4]. Isso permitiu que empresas e centros de pesquisa acumulassem grandes quantidades de dados históricos a partir dos anos 70 e 80 [7].

Porém, essa enorme quantidade de dados não refletia uma grande riqueza em conhecimento, pois não era analisada com ferramentas adequadas, já que um volume tão extenso ultrapassa a habilidade humana de compreensão. Consequentemente, importantes decisões eram frequentemente tomadas somente por intuição, simplesmente pela falta dessas ferramentas que poderiam extrair conhecimentos valiosos dos repositórios de dados [7].

Isso motivou diversos estudos a partir do início dos anos 90, que resultaram no surgimento do campo de pesquisa de Mineração de Dados, área que se refere ao processo de descoberta de novas informações e conhecimento, no formato de regras e padrões, a partir de grandes bases de dados [18]. A partir daí, foram desenvolvidas ferramentas para analisar bases de dados e descobrir padrões, contribuindo para diversas áreas, tais como [7]: pesquisas médicas, negócios estratégicos, biologia molecular, entre outras.

Dentre as principais tarefas em Mineração de Dados, destacam-se [6]: classificação, extração de regras de associação, clusterização, extração de padrões em séries temporais e extração de padrões sequencias. Em geral, essas tarefas podem ser classificadas em duas categorias: mineração preditiva e mineração descritiva [7].

Na mineração preditiva, deseja-se prever o valor desconhecido de um determinado atributo, a partir da análise histórica dos dados armazenados na base. Nessa categoria se enquadram as

tarefas de classificação e extração de padrões em séries temporais. Na mineração descritiva, padrões e regras descrevem características importantes dos dados com os quais se está trabalhando. Mineração de regras de associação, clusterização e mineração de padrões sequenciais fazem parte dessa categoria. A seguir serão descritas as principais tarefas de mineração de dados.

2.1.1 CLASSIFICAÇÃO

A tarefa de classificação tem por objetivo identificar, entre um conjunto pré-definido de classes, aquela à qual pertence um elemento a partir de seus atributos. Para inferir a qual classe esse elemento pertence, é necessária uma base de treinamento.

Um sistema de um banco que tem por objetivo inferir a classe à qual o cliente pertence, indicando se o mesmo será ou não um bom pagador, com base nos dados de clientes antigos e nas características do cliente que está sendo classificado, é um exemplo de utilização da tarefa de classificação.

2.1.2 REGRAS DE ASSOCIAÇÃO

Uma regra de associação representa um padrão de relacionamento entre itens de dados de um domínio de aplicação, que ocorre com uma determinada frequência. Essas regras são extraídas a partir de uma base de dados organizada em transações, que são formadas por um conjunto de itens desse domínio. Um exemplo genérico de regra de associação que poderia ser extraída de uma base de dados de qualquer vendas é: “clientes que compram o produto A geralmente compram o produto B”.

2.1.3 CLUSTERIZAÇÃO

A tarefa de clusterização é usada para agrupar elementos de uma base de dados através de seus atributos ou características, fazendo com que elementos similares fiquem no mesmo cluster e elementos não similares entre si fiquem em clusters distintos.

Essa técnica é muito utilizada em sistemas de grandes operadoras de cartão de crédito, separando os clientes em grupos de forma que aqueles que apresentam o mesmo comportamento de consumo fiquem no mesmo grupo. A separação desses clientes por grupo pode ser usada para fazer algum tipo de marketing apropriado ao grupo ou na detecção de fraudes, no caso de um cliente apresentar um comportamento diferente do esperado para o seu perfil.

2.1.4 PADRÕES EM SÉRIES TEMPORAIS

Uma série temporal é uma sequência de valores mensurados em intervalos iguais de tempo [7]. O principal objetivo da análise de padrões em séries temporais é realizar previsões futuras baseando-se no histórico dos dados.

Cotação diária do dólar, faturamento anual de uma empresa e evolução do índice da bolsa de valores são exemplos de séries temporais.

2.1.5 PADRÕES SEQUENCIAIS

Nesta seção, é detalhada com maior profundidade a extração de padrões sequenciais, visto que essa tarefa é de especial interesse para o testante do trabalho.

Existem muitas aplicações envolvendo dados sequenciais, e a ordem com que esses dados aparecem é muito importante para análise e entendimento de alguns padrões. Exemplos típicos incluem sequências de compras de um cliente, sequências biológicas e sequências de eventos na ciência e na engenharia. Padrões sequenciais representam sequências de eventos ordenados, que aparecem com significativa frequência em uma base de dados. Um exemplo de padrão sequencial é: “clientes que compram uma câmera digital Canon comumente compram uma impressora HP colorida dentro de um mês”.

Sequências são listas ordenadas de eventos. Uma sequência s é representada por $\langle e_1 e_2 e_3 \dots e_n \rangle$, onde e_j , $1 \leq j \leq n$, é dito um evento ou elemento da sequência s e e_1 ocorre antes de e_2 , que ocorre antes de e_3 e assim sucessivamente. Por sua vez, um evento ou elemento da sequência é representado por $\mathbf{e} = (i_1 i_2 i_3 \dots i_m)$, onde i_k , $1 \leq k \leq m$, é um item do domínio da aplicação. O tamanho da sequência é determinado pelo seu número de itens.

Podemos dizer que um evento em uma base de dados de uma loja de vendas é uma compra feita por um cliente, e os itens do domínio da aplicação são os produtos que pertencem à essa compra.

Além disso, outras definições são importantes. Uma sequência pode ser parte de outra sequência maior. Nesse caso, a sequência $\alpha = \langle a_1 a_2 \dots a_n \rangle$ é chamada de subsequência de outra sequência $\beta = \langle b_1 b_2 \dots b_m \rangle$, e β é uma supersequência de α , denotado como $\alpha \subseteq \beta$, se existirem inteiros $1 \leq j_1 < j_2 < \dots < j_n \leq m$ tais que $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, ..., $a_n \subseteq b_{j_n}$. Por exemplo, se $\alpha = \langle (ab), (d) \rangle$ e $\beta = \langle (abc), (de) \rangle$, onde a , b , c , d e e são itens, então α é uma subsequência de β e β é uma supersequência de α .

A métrica de avaliação dos padrões sequenciais é o suporte. Um banco de dados de

sequências, S , é composto de um conjunto de tuplas, $\langle SID, s \rangle$, onde SID é o identificador da sequência e s é a sequência. O suporte de uma sequência α num banco de dados de sequências S é o número de ocorrências dessa sequência no banco de dados. Esse suporte pode ser absoluto, representado apenas por um número inteiro, ou relativo, informando o percentual de vezes que a sequência aparece. Dessa forma, uma sequência de eventos é dita frequente se a quantidade de vezes que essa sequência aparecer for superior ao suporte mínimo informado pelo usuário, formando assim um padrão sequencial.

2.2 MINERAÇÃO DE DADOS NA ENGENHARIA DE SOFTWARE

Tarefas de mineração de dados têm sido muito utilizadas em engenharia de software, principalmente quando aplicadas em repositórios de dados para se obter informações sobre a evolução de software ao longo do tempo, com o objetivo de aumentar a qualidade do processo de desenvolvimento de software [1, 2, 17, 19, 21].

Em [17], utilizou-se um algoritmo classificador, a partir do aprendizado em uma base de treinamento, para extrair relações que indicam que arquivos de código fonte, em um sistema legado, são relevantes uns aos outros no contexto da manutenção de software. Tais relações puderam revelar interconexões complexas entre os arquivos do código fonte do sistema, sendo úteis na compreensão deles. Assim, o algoritmo classificador, após receber dois arquivos de código fonte, retorna verdadeiro ou falso, indicando se são relevantes entre si.

Em [19], aplica-se mineração de dados para se encontrar relações de dependências entre arquivos do código fonte, até mesmo dependências difíceis de se determinar, tais como aquelas existentes entre códigos fonte de linguagens diferentes. Em especial, é utilizada a técnica de extração de regras de associação para determinar padrões de mudanças entre arquivos do código fonte para ajudar os desenvolvedores em tarefas de modificação.

Em [2], utilizam-se técnicas de mineração de dados em um repositório UML versionado para se extrair regras de associação que possam identificar elementos do modelo UML que foram modificados juntos no passado e que provavelmente precisarão ser modificados juntos no futuro.

Em [21], aplicam-se técnicas de mineração de dados em repositórios versionados de código fonte para se extrair regras de associação do tipo: “programadores que alteraram a função A também alteraram as funções B e C”. Tais regras são extraídas com o objetivo de guiar desenvolvedores de software na alteração do código fonte.

Em [1], um algoritmo de clusterização foi utilizado num sistema de controle de versões para identificar classes semanticamente relacionadas. A partir do gráfico gerado pelo algoritmo, pôde-se analisar que mudanças em classes de certo cluster eram frequentemente envolvidas com mudanças em classes de outro cluster.

Apesar de existirem diversas abordagens que aplicam mineração de dados em problemas de engenharia de software, a partir da análise do histórico de publicações do *workshop Mining Software Repositories* e de edições especiais das revistas *IEEE Transactions on Software Engineering* e *IEEE Software*, não foi possível identificar outros trabalhos que atuavam no problema de *code completion*, o que motivou a elaboração deste trabalho.

CAPÍTULO 3 - VERTICAL CODE COMPLETION

Neste capítulo, é detalhada a abordagem proposta neste trabalho, intitulada Vertical Code Completion (VCC). O mesmo foi dividido em duas seções, que representam duas fases distintas no processo de aplicação do VCC.

A primeira fase é a de preparação e mineração dos dados, detalhada na seção 3.1, onde são extraídos todos os padrões que serão sugeridos ao programador. Nesta fase, o código fonte é analisado e organizado, para permitir que a mineração de dados desse código seja efetuada, e em seguida seus resultados são armazenados em uma estrutura adequada.

A segunda fase do processo de uso do VCC está detalhada na seção 3.2. Nessa etapa, o código que está sendo produzido em tempo real pelo desenvolvedor será analisado com o intuito de encontrar trechos correspondentes a padrões frequentes, que foram obtidos na primeira fase. Em seguida, esses padrões são classificados através de métricas e sugeridos para o usuário.

A Figura 1 ilustra todo o processo proposto pela abordagem VCC.

3.1 OBTENÇÃO DE PADRÕES FREQUENTES DE CODIFICAÇÃO DE SOFTWARE.

Esta seção está dividida em outras três subseções. O processo de análise de código fonte é apresentado na Subseção 3.1.1. Em seguida a estratégia de mineração de dados é descrita na Subseção 3.1.2. Por último, a estrutura de armazenamento dos padrões obtidos é detalhada na Subseção 3.1.3.

3.1.1 ANÁLISE DO CÓDIGO FONTE

Para que seja possível, através de uma fonte de dados pré-existente, sugerir padrões frequentes de código fonte, é necessário que os dados estejam coesos e estruturados para execução de consultas. Contudo, isso não é uma realidade inicial para o contexto desse trabalho, visto que o código de uma aplicação é armazenado em formato texto, sem obedecer a padrões rígidos de estruturação. Felizmente, cada linguagem de programação obedece a um conjunto de regras

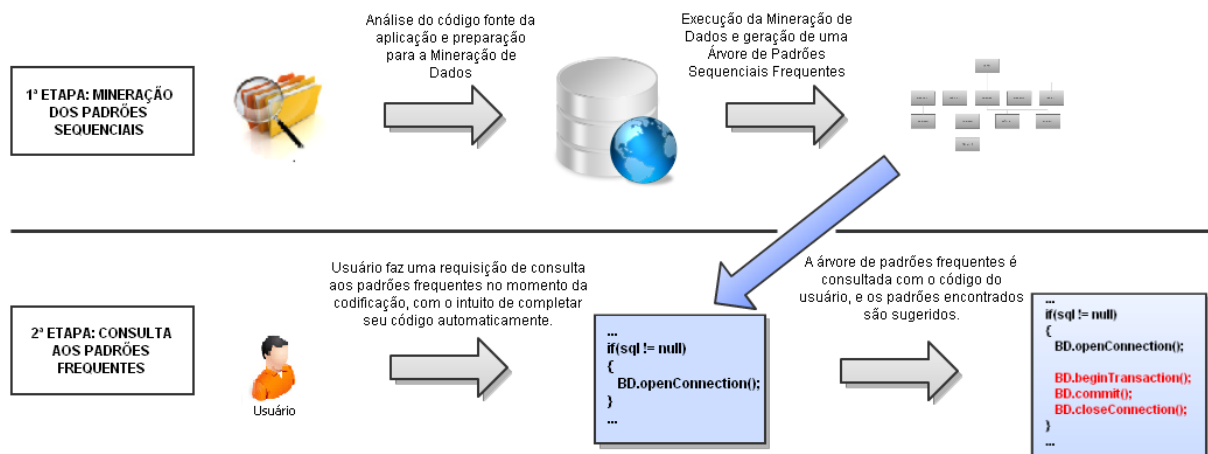


Figura 1: Fluxo de uso do VCC.

de formatação, que são necessárias para a compilação adequada do código fonte em linguagem de máquina.

Dessa forma, embora não seja possível fornecer diretamente arquivos de código como entrada para mineração sequencial de dados, os padrões da linguagem de programação podem ser utilizados para se extrair as informações pertinentes do código fonte e organizá-las no formato de sequências de eventos. Conforme visto no Capítulo 2, através da análise de eventos que ocorrem em sequência, é possível detectar padrões frequentes que obedecem a uma determinada sequência. Todavia, em cada domínio de aplicação, sequências, eventos e os itens que compõem cada evento possuem significados distintos [7].

Neste trabalho, os eventos correlacionados estão todos em um mesmo corpo de método, e cada evento é uma chamada de método. Com isso, não é possível dividir o evento em diferentes itens, sendo então cada evento atômico. Dessa forma, os padrões sequenciais minerados são listas de chamadas de método, que obedecem a uma determinada sequência e se repetem frequentemente em diferentes corpos de métodos.

Na Figura 2, é possível visualizar a relação entre a codificação de um método e sua respectiva sequência de eventos.

3.1.2 MINERAÇÃO DO CÓDIGO FONTE

Existem diversos algoritmos para realizar mineração de padrões sequenciais, cada um com suas particularidades de entrada e saída. Organizando os dados de uma maneira que atenda a essas particularidades, a mineração de padrões sequenciais pode então ser realizada, mas para

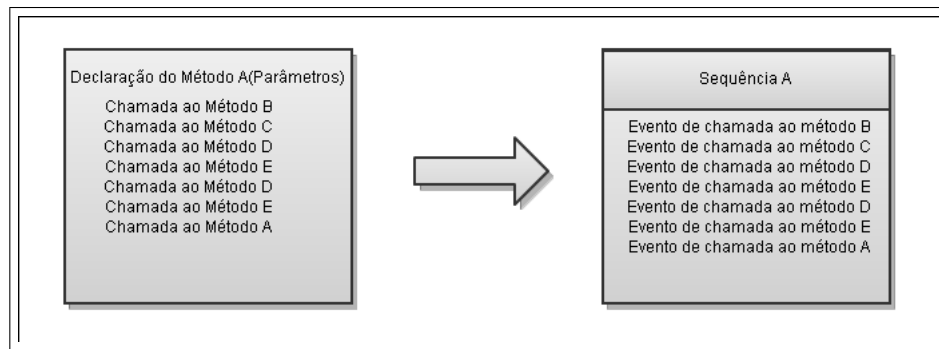


Figura 2: Exemplo de codificação de método e a sequência gerada.

que os resultados desse processo sejam proveitosos, dois conceitos são essenciais: suporte e confiança.

O suporte já foi definido anteriormente, no Capítulo 2, e sabe-se que é a quantidade de vezes que um determinado padrão se repete na base de dados. Portanto, é possível definir um suporte mínimo para a obtenção desses padrões, filtrando os padrões mais frequentes.

A confiança, por outro lado, é muito importante nas regras de associação e representa uma métrica de avaliação que traz uma maior riqueza para a apresentação de regras mineradas. Entretanto, apesar de tradicionalmente não ser um conceito utilizado na mineração de padrões sequenciais, é definido e utilizado neste trabalho.

Para definir a confiança em padrões sequenciais, primeiramente é apresentada a definição de confiança em regras de associação. Consequentemente, é necessário que também seja apresentada a definição do suporte de uma regra de associação.

O suporte de um conjunto de itens A, consiste na porcentagem de transações da base de dados que contêm esse item, e pode ser representado por $\text{Sup}(A)$. Analogamente, um outro conjunto de itens B, que pode conter o conjunto de itens A, terá suporte representado por $\text{Sup}(B)$ [6].

A confiança pode ser definida em termos do suporte. Ou seja, dado uma regra de associação $A \rightarrow B$, a confiança dessa regra representa a porcentagens de transações que contêm B, dentre todas as transações que contêm A, ou seja, $\text{Conf}(A \rightarrow B) = \text{Sup}(B) / \text{Sup}(A)$ [6]. Desta forma, uma regra $A \rightarrow B$ tem suporte equivalente à probabilidade conjunta de A e B, $P(A \cap B)$, e confiança equivalente à probabilidade condicional de B dado que A ocorre, $P(B|A)$.

É possível então definir a confiança em mineração de padrões sequenciais, enxergando-a como uma derivação da existente em regras de associação. Dada uma sequência s , e outra sequência S , que é supersequência de s , a confiança de S em relação a s , será a porcentagem de sequências que contêm S dentre todas as que contêm s . Chamando então s de **SubSeq** e S de

SuperSeq, a confiança de **SuperSeq** em relação à **SubSeq** é calculada como:

$$\text{Confiança}_{\text{SuperSeq/SubSeq}} = \text{Suporte}_{\text{SuperSeq}} / \text{Suporte}_{\text{SubSeq}}$$

Esse conceito pode ser exemplificado da seguinte maneira:

Dado que um padrão sequencial X, formado por $\{A, B, C, D\}$, possui suporte de 28% e que o padrão sequencial Y, formado por $\{A, B\}$, possui suporte de 35%. A confiança de X em função de Y é de 80%.

Com isso, a seguinte afirmação pode ser empregada pelo VCC:

Usuários que chamam os métodos A e B em sequência, também chamam, com 80% de confiança, os métodos C e D.

3.1.3 GERAÇÃO DE ÁRVORE DE CHAMADAS

Para que os padrões obtidos na fase de mineração sequencial possam ser utilizados na sugestão de código fonte, uma estrutura adequada deve ser empregada para o armazenamento e consulta dos mesmos.

Neste trabalho, é utilizada uma árvore de profundidade e largura variável para armazenar os padrões frequentes obtidos. A Figura 3 ilustra a estrutura de uma árvore de padrões frequentes.

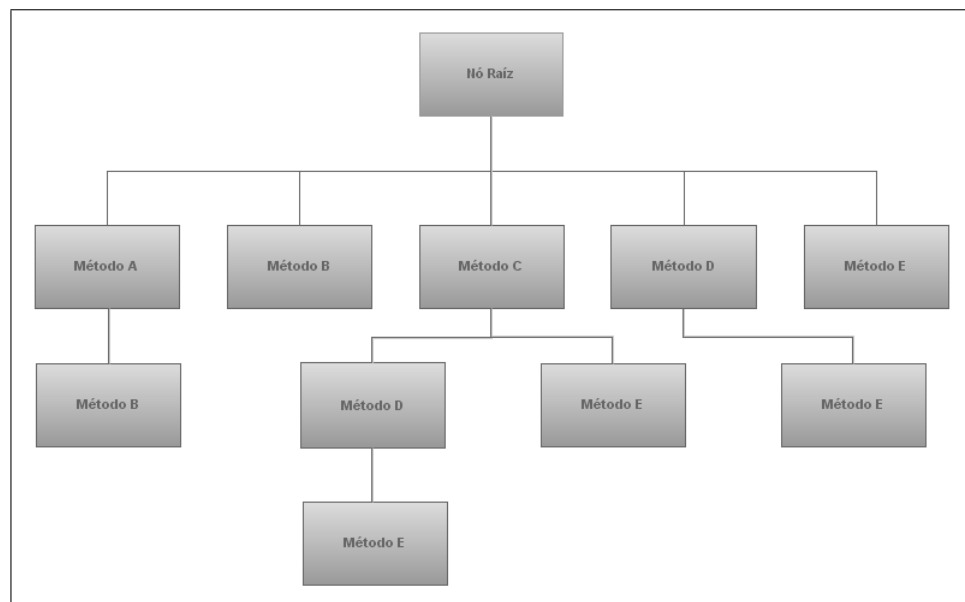


Figura 3: Exemplo de árvore de padrões frequentes.

É importante ressaltar que a estrutura utilizada na criação da árvore permite que a busca por uma sequência de código tenha complexidade assintótica [15] $O(n)$, sendo n o tamanho da sequência pesquisada. Isso acontece porque todos os elementos presentes na árvore em níveis

mais profundos, também estão representados no segundo nível, conforme pode ser visto na Figura 3 nos Métodos B, D e E.

Apesar de parecer um desperdício proposital de espaço de armazenamento para obter um melhor desempenho na busca por elementos da árvore, a presença de todos os elementos frequentes no segundo nível da árvore, se deve a um comportamento inerente à mineração de padrões sequenciais, que diz que se uma sequência é frequente, ou seja, possui suporte superior ao suporte mínimo, todas as suas subsequências também serão frequentes. Isso pode ser observado na Figura 3 nas ocorrências do nó que representa o Método B, por exemplo. Embora esse método já esteja presente no terceiro nível da árvore, representando a sequência $\langle A, B \rangle$, é necessário que o mesmo também esteja presente no segundo nível representando padrões sequenciais que possuam como primeiro evento frequente, justamente o Método B.

Após definir a estrutura de armazenamento como uma árvore, é importante que seja decidido o que cada nó irá armazenar. Considerando cada nó como o fim de um padrão sequencial frequente, nos mesmos será armazenado o suporte desse padrão.

Entretanto, a confiança de um padrão não pode ser vista como um único valor. A confiança de um padrão sequencial depende da subsequência que está sendo consultada. Desta forma, o tamanho do padrão sequencial minerado determinará quantos valores de confiança o mesmo terá. Dada uma sequência de tamanho igual a três, $X = \langle C, D, E \rangle$, as seguintes confianças são definidas:

- Confiança de X em função de uma sequência vazia. O valor dessa confiança é o mesmo do suporte do padrão sequencial;
- Confiança de X em função da sequência $\langle C \rangle$. O valor dessa confiança será o suporte de X dividido pelo suporte de $\langle C \rangle$;
- Confiança de X em função da sequência $\langle C, D \rangle$. O valor dessa confiança será o suporte de X dividido pelo suporte de $\langle C, D \rangle$;

Dessa forma, uma gama de sugestões pode ser fornecida ao usuário utilizador do VCC. Dado que uma chamada ao método C, que possui suporte s , foi codificada, e que a sequência $\langle C, D \rangle$, que está presente na árvore de padrões sequenciais frequentes, possui suporte S_1 , pode-se sugerir a chamada ao método D, com confiança C_1 , dado que $C_1 = S_1 / s$. Além disso, dado que a sequência $\langle C, D, E \rangle$ também possui suporte S_2 , superior ao suporte mínimo, pode-se sugerir a sequência de chamadas $\langle D, E \rangle$ com confiança C_2 , dado que $C_2 = S_2 / s$.

A Figura 4 apresenta a árvore de padrões sequenciais frequentes e a forma com que os suportes e confianças são armazenados.

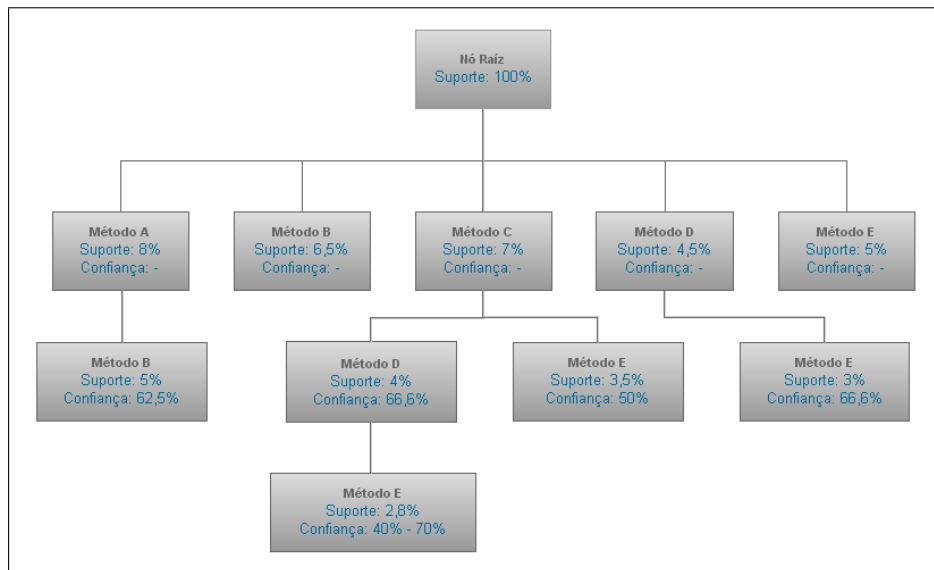


Figura 4: Exemplo de árvore de padrões frequentes com suportes e confianças.

Observando a sequência frequente $\langle C, D, E \rangle$ por exemplo, as confianças armazenadas no nó que representa o método E, 40% e 70%, são respectivamente:

- A confiança do padrão sequencial $\langle C, D, E \rangle$ com relação à sequência $\langle C \rangle$; e
- A confiança do padrão sequencial $\langle C, D, E \rangle$ com relação à sequência $\langle C, D \rangle$.

3.2 SUGESTÃO DE PADRÕES FREQUENTES DE CÓDIGO FONTE

Para que um padrão sequencial possa ser sugerido, é necessário que uma entrada seja disponibilizada pelo usuário. Nesse momento, diversas estratégias podem ser tomadas para decidir como será feita a pesquisa do que está sendo programado.

Enquanto o projeto VCC estava sendo desenvolvido, algumas dessas estratégias foram testadas com o intuito de realizar poucas consultas, otimizando o tempo de resposta do programa. Utilizar apenas as últimas linhas que foram programadas, ou apenas combinações de linhas contíguas, mostrou-se infrutífero, pois muitos padrões interessantes passaram despercebidos por estarem dispostos de diversas maneiras no corpo do método.

Em um método com dez linhas, por exemplo, um padrão sequencial frequente pode ser detectado a partir de chamadas que estão localizadas imediatamente uma após a outra, como em chamadas que se encontram uma no início e outra no fim do que já foi programado. Um exemplo

interessante de padrões sequenciais que não se localizam contiguamente, são as aberturas e fechamentos de conexões com bancos de dados. Ao abrir uma conexão, espera-se que algum procedimento seja realizado na base de dados, antes que a mesma seja fechada.

Com isso, fica explícita a necessidade de se intercalar de diferentes maneiras as chamadas de método disponíveis. Isso é feito através da combinação das chamadas de métodos, de todas as maneiras possíveis. Todavia, muitas combinações diferentes ainda podem ser geradas, fazendo com que o tempo de resposta para a consulta na árvore de padrões frequentes seja longo. Isso porque mesmo sabendo que as boas práticas de programação recomendam a filosofia de dividir para conquistar [3], métodos em um projeto de software podem se tornar grandes demais.

Com isso, para que a consulta a todas as combinações de chamadas de métodos seja realizada em tempo hábil, é necessário que o tamanho máximo dessas combinações seja limitado. No VCC, o tamanho máximo das combinações é configurável, permitindo que um valor que atenda às características do projeto em questão seja alcançado. Entretanto, esse valor pode ser alto, gerando uma enorme quantidade de combinações, e fazendo com que o tempo de resposta das consultas ainda não seja satisfatório.

Com o intuito de minimizar esse problema, a partir da análise das combinações geradas, uma estratégia de poda foi desenvolvida, evitando que todas essas combinações sejam consultadas. Essa estratégia parte do princípio de mineração de padrões sequenciais que garante que se uma sequência não é frequente, então todas as supersequências dessa sequência também não são frequentes. No projeto VCC, a poda das sequências a serem consultadas na árvore de padrões acontece após a consulta de uma sequência que não é frequente. Todas as outras sequências de método que são supersequências desta são então descartadas.

Finalmente, após todas as combinações de chamadas de métodos terem sido consultadas, os padrões sequenciais obtidos são classificados de acordo com seus valores de suporte e confiança e então sugeridos para o usuário do VCC. Este usuário pode então analisar as sugestões e escolher a que se adéqua melhor ao que está sendo desenvolvido.

Utilizando como exemplo a árvore da Figura 1, pode-se supor que o usuário do VCC codifique chamadas aos métodos A, D e F, e em seguida efetue uma requisição ao VCC, fazendo com que as seguintes combinações de chamadas sejam geradas:

- A;
- D;
- F;

- $\langle A, D \rangle$;
- $\langle A, F \rangle$;
- $\langle D, F \rangle$.

Essas combinações são então consultadas na árvore de padrões frequentes, de acordo com o tamanho de cada combinação. Ao consultar a chamada ao método A, a chamada ao método B seria obtida com suporte de 5% e confiança de 62,5%. Já consultando a chamada ao método D, a chamada ao método E seria obtida com suporte de 3% e confiança de 66,6%. Em seguida, ao consultar a chamada ao método F nenhum padrão seria encontrado, fazendo com que a poda de combinações seja realizada. As combinações $\langle A, F \rangle$ e $\langle D, F \rangle$ são então 'podadas', evitando que sejam consultadas desnecessariamente. Finalmente, a combinação $\langle A, D \rangle$ é consultada na árvore de padrões frequentes e novamente nenhum padrão é encontrado, a poda é chamada, mas como não existem outras combinações a serem consultadas, nenhuma ação é realizada. É importante ressaltar que a combinação $\langle A, D, F \rangle$ não é nem mesmo gerada para ser consultada, visto que na árvore não existe nenhum padrão sequencial que possua mais de três chamadas de método.

CAPÍTULO 4 - O *PLUGIN* VCC

Com o intuito de automatizar a abordagem definida no capítulo 3, foi implementado um protótipo do VCC, denominado *plugin* VCC. Neste capítulo, os detalhes das tecnologias utilizadas na implementação do *plugin* VCC são abordados, com foco nos pontos detalhados no capítulo 3.

Para desenvolver uma ferramenta que auxilie o desenvolvedor de *software* no momento da codificação, primeiramente é necessário decidir em que ambiente a mesma será utilizada. Atualmente é difícil se pensar em desenvolver um sistema sem o auxílio de uma IDE (*Integrated Development Environment*), que além de ajudar os programadores, em geral é um ambiente ideal para se acoplar uma nova ferramenta. Por esse motivo, e ainda visando alcançar o maior número de usuários, decidiu-se utilizar o VCC acoplado em uma IDE.

Entretanto, para decidir em que IDE o VCC seria acoplado, foi preciso definir com qual linguagem de programação o mesmo seria utilizado. Devido ao nosso conhecimento prévio e a abrangência na comunidade de desenvolvimento, Java foi escolhida como a linguagem da implementação desse trabalho. Por esses mesmos motivos e também pela facilidade de acoplar novas ferramentas no formato de *plugins*, a IDE Eclipse [11] foi escolhida para receber o *plugin* VCC.

Uma das grandes vantagens de implementar este trabalho no formato de um *plugin* para o Eclipse é que a própria IDE já fornece uma interface de desenvolvimento de *plugins*. É possível então, utilizar diversas funcionalidades que a ferramenta disponibiliza, como por exemplo o ASTParser [9], que transforma código Java para um formato de árvore.

O ASTParser constrói uma *Abstract Syntax Tree* (AST) do código fonte em que o *plugin* está sendo executado. Embora a AST seja uma representação estritamente sintática da estrutura do código fonte de uma aplicação, essa representação funciona exatamente como o volume de dados necessário para efetuar a mineração de padrões sequenciais frequentes. Na construção do VCC, o trabalho de análise do código fonte, que foi destacado no capítulo 3, se reduziu a acessar essa AST, sem que tenha sido preciso construir um *parser* textual para a linguagem Java.

4.1 A ANÁLISE DO CÓDIGO FONTE ATRAVÉS DA ASTPARSER

Para extrair os padrões sequenciais frequentes da aplicação em que o VCC está sendo utilizado, é preciso obter todas as chamadas de métodos utilizadas nos corpos dos métodos deste projeto. Conforme mostra a Figura 5, o usuário deve clicar no item do menu *Generate Tree* para que a árvore de padrões frequentes possa ser gerada.

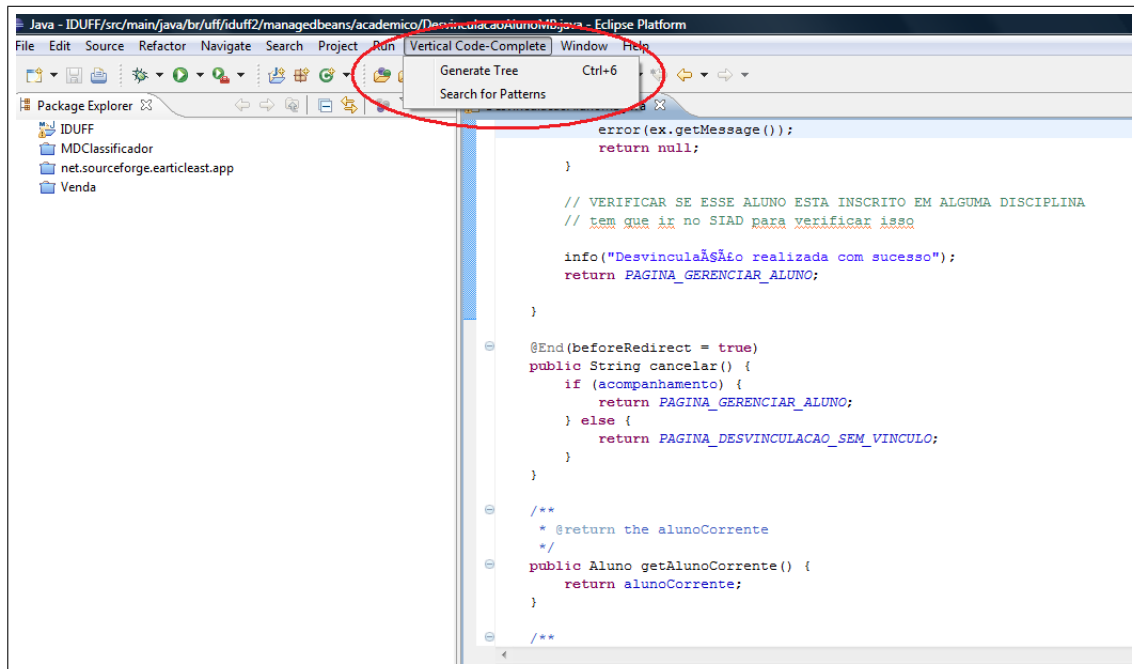


Figura 5: Menu do Plugin VCC

Para construir essa árvore, foram seguidas algumas convenções de acesso à AST. Primeiramente, é necessário definir que no topo da hierarquia da AST está a classe *ASTNode*. As construções Java são representadas por esta classe. É importante notar que através do mecanismo de herança, essas entidades Java são especializadas, permitindo que cada uma possua características próprias. A classe que representa as **Classes** Java possui a lista dos seus métodos e atributos, por exemplo. Já a classe que representa os **Métodos**, possui informações relativas aos parâmetros do método e o código fonte do próprio método.

Sendo assim, para analisar o código fonte, os *ASTNodes* devem ser acessados, extraindo as chamadas de métodos e salvando-as para serem mineradas pelo algoritmo de mineração de padrões sequenciais. Para obter essas chamadas, uma hierarquia deve ser seguida para acessar todas as classes do projeto em questão. Essa hierarquia é intuitiva, pois obedece a estrutura adotada pela linguagem Java e pela IDE Eclipse.

A Figura 6 exibe a estrutura de um *Workspace* Java. Um *Workspace* é um espaço de trabalho em que os projetos do usuário ficam armazenados. Por sua vez, um Projeto é formado por

Pacotes. Apesar de *Workspace* e *Projeto* não serem *ASTNodes*, servem como porta de entrada para acessar as demais entidades Java que são *ASTNodes*.

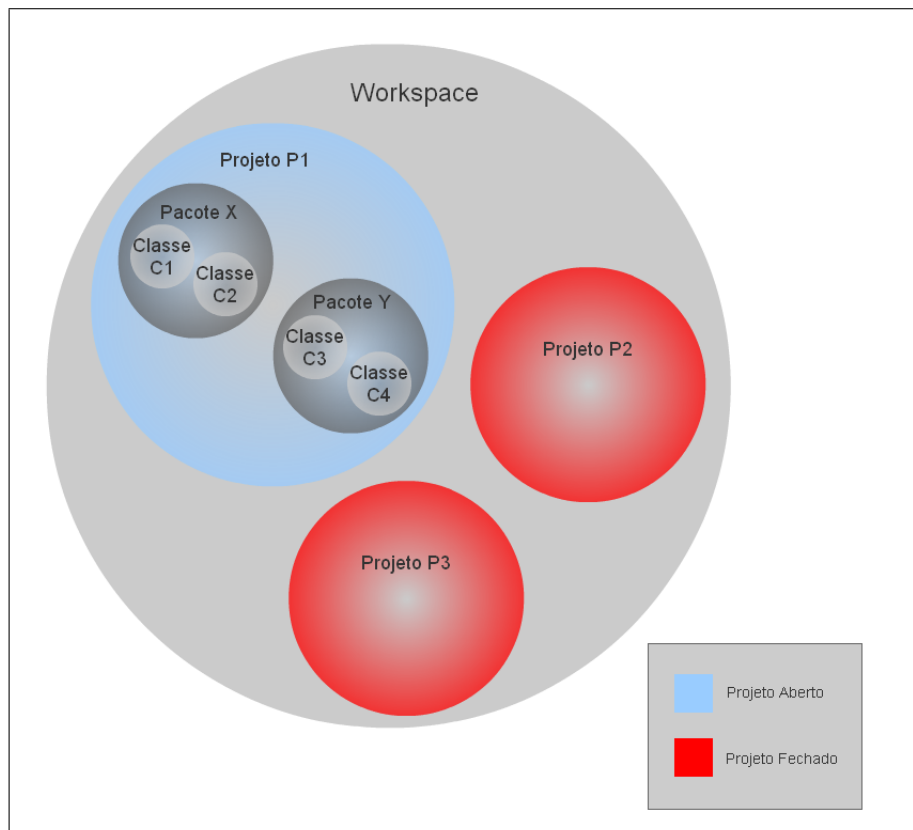


Figura 6: Estrutura de um *Workspace* Java.

O acesso ao *Projeto* começa então pelo *Workspace* em que o mesmo está hospedado. O código do *plugin* VCC acessa todos os Pacotes da aplicação. O Pacote é uma forma de organizar as Classes Java do software que está sendo construído. Com isso, o próximo passo é acessar todas as Classes que estão em cada Pacote e só então todos os Métodos que estão dentro das Classes.

Com isso, é possível obter todas as chamadas de Métodos que se encontram dentro do corpo de um Método, e armazená-las em formato de eventos para a execução da Mineração de Padrões Sequenciais. Por último, é importante citar que todos os acessos implementados pela *ASTParser* são realizados através do *Design Pattern Visitor*[14].

4.2 MINERAÇÃO DO CÓDIGO FONTE

Existem diversas implementações de código aberto de algoritmos de mineração de padrões sequenciais, disponíveis na literatura [5, 7, 18, 20]. Por esse motivo, foi decidido que não seria necessário o desenvolvimento de um programa que realize a mineração desses padrões. Após a

análise dos algoritmos GSP, PLWAP e WapTree, o PLWAP foi escolhido como o algoritmo que seria utilizado para a extração dos padrões por apresentar uma melhor performance com relação aos demais.

Entretanto, embora o desempenho do PLWAP fosse satisfatório, o mesmo apenas informava quais padrões sequenciais eram frequentes, sem disponibilizar o suporte de cada padrão. Foi então necessário efetuar uma modificação no algoritmo para registrar esse valor de suporte. Em seguida, após obter todos os valores de suporte, foi possível calcular as confianças de cada padrão sequencial frequente, conforme citado na seção 3.1.2.

4.3 ÁRVORE DE PADRÕES FREQUENTES

Conforme visto no capítulo 3, todos os padrões sequenciais são armazenados em uma árvore, que é consultada quando um padrão sequencial está sendo buscado. Na implementação do VCC, esta árvore é representada por um conjunto de nós, que são objetos da classe **Method-CallNode**. Essa classe possui além das confianças do padrão e da assinatura completa do método, uma referência para o nó pai e um **HashMap**, que contém todos os filhos desse nó. Um **HashMap** é uma classe Java, que através de uma tabela *hash*, organiza um mapa de dados [12].

Para exemplificar essa estrutura, supondo um padrão sequencial $X = \langle A, B, C \rangle$, e outro padrão $Y = \langle A, B, D \rangle$, o nó **B** será representado na árvore como um objeto do tipo **Method-CallNode**, que contém um **HashMap** com as referências para os nós **C** e **D**, além de uma referência para o nó **A**. Além disso, B também poderá aparecer como filho de outros elemento na árvore, representando outros padrões sequenciais. Também é importante ressaltar que B, obrigatoriamente, aparecerá no segundo nível da árvore, conforme citado na seção 3.1.3.

Por fim, como a fase de construção da árvore de padrões sequenciais é independente da fase de consulta a esses padrões, é necessário persistir a árvore em memória secundária. Para isso, todos os nós da árvore implementam a interface *Serializable* [13]. O objeto que representa o nó raiz dessa árvore e por recursão todos os demais elementos da árvore são armazenados na memória secundária do sistema em que o plugin está sendo executado, para serem acessados posteriormente na fase de consultas.

4.4 GERAÇÃO E PODA DAS COMBINAÇÕES DE CHAMADAS DE MÉTODOS

Como foi citado no Capítulo 3, a geração de todas as combinações de chamadas de método, com o intuito de consultar a árvore de padrões frequentes, foi considerada uma maneira apropriada para garantir que padrões não estão deixando de ser sugeridos ao usuário.

Entretanto ainda é necessário decidir quais chamadas de método, dentre as disponíveis no corpo de cada método, serão utilizadas na consulta à árvore. Isso porque existem duas opções de escolha.

A primeira dessas opções é analisar todas as chamadas de métodos que estão disponíveis no corpo do método que está sendo codificado. A seleção deste método poderia ser feita através da digitação do nome do pacote, da classe e do próprio método. Embora essa opção represente uma facilidade na implementação, prejudica em muito a usabilidade da ferramenta, além de não permitir uma melhor delimitação da área de pesquisa. Um usuário poderia estar realizando uma alteração na metade do corpo do método, por exemplo, e independentemente disso, todas as chamadas, do início ao fim do mesmo seriam utilizadas na pesquisa por padrões de código fonte.

A segunda opção seria utilizar a posição do cursor do mouse do usuário do VCC, assim como funciona no *code completion* tradicional. Quando o usuário posiciona o mouse e faz uma chamada ao VCC, todas as chamadas de método que se localizam acima do cursor, até o início do corpo do método, são combinadas e utilizadas para a consulta na árvore de padrões frequentes. Pode ser visto, então, que essa alternativa, além de delimitar melhor a área de consulta, também incrementa a usabilidade do *plugin*, fazendo com que tenha sido a escolhida para ser implementada.

A Figura 7 mostra a posição do cursor no corpo do método do usuário, no momento da requisição de obtenção das sugestões de código fonte.

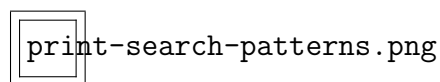


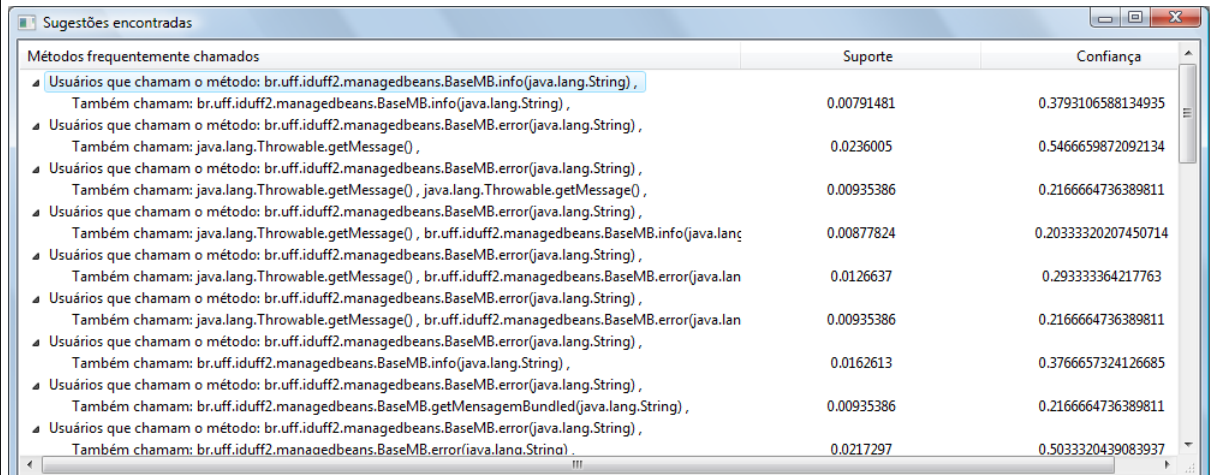
Figura 7: Chamada do usuário com o intuito de obter as sugestões de código fonte.

Em seguida, para que essas combinações possam ser geradas, primeiramente é preciso ler todas as chamadas de métodos que se encontram antes do cursor. Para facilitar essa tarefa, mais uma vez a **ASTParser** foi utilizada, entretanto, não foi preciso iterar em todos os pacotes, classes e métodos, pois a posição do cursor já informa em qual método devem ser lidas todas as chamadas. Após a leitura das chamadas de métodos, cada combinação é armazenada em um

objeto da classe **ComparableList**. Essa classe pertence ao projeto VCC, é uma subclasse de **ArrayList** e implementa a interface **Comparable**. Esta interface exige que um método **compareTo** seja criado, para que dois objetos possam ser comparados. O critério de comparação utilizado para essa classe foi o tamanho da lista. Com isso, é possível armazenar todas as combinações em outro **ArrayList** e ordená-lo pelo tamanho das combinações.

Essa ordenação é muito importante, pois conforme já foi citado na seção 3.2, gerar todas as combinações de chamadas de métodos pode produzir uma enorme quantidade de dados para serem consultados, fazendo com que seja necessária a criação de uma estratégia de poda das combinações. Essa estratégia depende da ordenação dos dados, pois permite que, após consultar uma determinada combinação de chamadas de métodos que não é frequente, o **ArrayList** que contém todas as outras combinações geradas seja percorrido, apenas a partir do índice dessa combinação, ou seja, sem tentar podar combinações que já foram consultadas. Todas as combinações que são supersequências da sequência que não é frequente são então removidas, evitando que sejam feitas consultas inúteis.

Após todas as consultas terem sido feitas, os padrões sequenciais obtidos são então sugeridos ao usuário da aplicação em uma janela, conforme mostrado na Figura 8, permitindo que o mesmo avalie de acordo com o suporte e a confiança da regra, qual melhor se aplica na sua codificação.



Métodos frequentemente chamados	Suporte	Confiança
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code> , Também chamam: <code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code> , 	0.00791481	0.3793106588134935
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>java.lang.Throwable.getMessage()</code> , 	0.0236005	0.5466659872092134
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>java.lang.Throwable.getMessage()</code> , <code>java.lang.Throwable.getMessage()</code> , 	0.00935386	0.2166664736389811
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>java.lang.Throwable.getMessage()</code> , <code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code> , 	0.00877824	0.20333320207450714
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>java.lang.Throwable.getMessage()</code> , <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , 	0.0126637	0.293333364217763
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>java.lang.Throwable.getMessage()</code> , <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , 	0.00935386	0.2166664736389811
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code> , 	0.0162613	0.3766657324126685
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , 	0.00935386	0.2166664736389811
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>br.uff.iduff2.managedbeans.BaseMB.getMensagemBundled(java.lang.String)</code> , 	0.00935386	0.2166664736389811
<ul style="list-style-type: none"> Usuários que chamam o método: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , Também chamam: <code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code> , 	0.0217297	0.5033320439083937

Figura 8: Sugestões baseadas nos padrões sequenciais frequentes.

CAPÍTULO 5 - RESULTADOS EXPERIMENTAIS

Nesse capítulo são discutidos os resultados experimentais obtidos a partir da utilização da abordagem proposta sobre o sistema objeto de estudo, o IdUFF - Sistema de Gestão Acadêmica da Universidade Federal Fluminense. A seção 5.1 apresenta o sistema utilizado no experimento. A seção 5.2 apresenta o planejamento do experimento, seguido pela seção 5.3, que aborda a aplicação do experimento. Finalmente, as seções 5.4 e 5.5 apresentam, respectivamente, os resultados obtidos e as ameaças à validade desse experimento.

5.1 O SISTEMA OBJETO DE ESTUDO - IDUFF

O sistema IdUFF é o sistema de gestão acadêmica utilizado na Universidade Federal Fluminense. Possui como principal funcionalidade a inscrição on-line de alunos em disciplinas, permitindo que aproximadamente 30 mil alunos escolham as disciplinas que vão cursar no período sem precisar comparecer a coordenação do curso, como era feito antes do sistema existir. Outras funcionalidades do IdUFF incluem a geração de históricos e comprovantes de regularidade de matrícula on-line, o módulo de lançamento de notas pelos professores, entre outras.

O IdUFF foi desenvolvido sobre a plataforma Java, a mesma linguagem que o *plugin* VCC foi desenvolvido para ser utilizado. Sua arquitetura utiliza os frameworks Hibernate, que é responsável pelo mapeamento objeto/relacional da aplicação Java, Spring que é responsável pela inversão de controle e injeção de dependências no sistema e Mockito, responsável por simular o comportamento de certos objetos para realizar testes. Do ponto de vista de utilização, possui aproximadamente 40 mil usuários, entre alunos, professores e funcionários da Universidade. O repositório de código fonte do IdUFF possui 850 commits. O sistema possui 779 classes Java, desconsiderando classes de teste, arquivos de interface HTML e arquivos de configuração.

O IdUFF foi escolhido para ser utilizado no experimento pela facilidade de contato com os desenvolvedores, que são ou foram alunos de graduação e pós-graduação da UFF, e por ser um sistema de médio porte, que vem sendo desenvolvido há 4 anos e apresenta uma grande quantidade de código fonte a ser minerada.

5.2 PLANEJAMENTO DO EXPERIMENTO

Com o objetivo de avaliar o quanto as sugestões de código do *plugin* VCC seriam úteis para um desenvolvedor que o estivesse utilizando, duas abordagens de experimento foram discutidas, ambas utilizando como base a experiência de desenvolvimento do programador no sistema IdUFF.

Na primeira abordagem, o próprio desenvolvedor utilizaria o *plugin* VCC, instalado na sua IDE, ou seja, no seu ambiente de desenvolvimento do dia-a-dia de trabalho e, a partir disso, poderia avaliar se as sugestões do *plugin* seriam úteis. Porém, esta abordagem não se mostrou interessante para ser aplicada nesse momento, pois além de demandar muito tempo, causaria muitos transtornos, pois o ambiente de desenvolvimento do sistema IdUFF utiliza uma IDE diferente da qual o *plugin* VCC foi acoplado. Outro ponto negativo dessa abordagem é que a usabilidade do *plugin* não é o foco desse experimento, mas sim os resultados apresentados pelo mesmo.

Na segunda abordagem, uma análise em ambiente controlado poderia ser aplicada. Desta forma, inicialmente os padrões de código seriam obtidos a partir da utilização do *plugin* VCC sobre o repositório de código fonte do IdUFF e, a partir daí, um questionário de avaliação contendo alguns desses padrões de código pré-selecionados seria apresentado aos desenvolvedores voluntários do IdUFF. Assim, cada voluntário poderia classificar, a partir de sua experiência no desenvolvimento do sistema, se as sugestões apresentadas fazem sentido ou não.

A segunda abordagem foi escolhida, justamente por permitir um maior controle sobre a avaliação feita pelos desenvolvedores voluntários, garantindo que os mesmos padrões fossem avaliados por todos. Assim, poderíamos avaliar se os resultados eram interessantes para os diversos perfis de desenvolvedores.

Para participar do experimento, os voluntários deveriam pertencer obrigatoriamente à equipe de desenvolvimento do sistema IdUFF. Seis desenvolvedores foram voluntários na participação desse experimento. Não houve nenhum tipo de compensação para os participantes.

5.3 APLICAÇÃO DO EXPERIMENTO

Inicialmente, cada voluntário foi informado do estudo através do Formulário de Consentimento (Apêndice I). Caso concordasse em participar, o Questionário de Caracterização (Apêndice II) era preenchido, avaliando o nível de conhecimento e experiência do voluntário em diferentes temas relacionados ao estudo. Essas informações foram de grande utilidade, ajudando na

Critério			P1	P2	P3	P4	P5	P6
1		Formação acadêmica	Mestrando	Graduado	Graduando	Graduado	Mestrando	Graduado
2	2.1	Experiência em desenvolvimento	Indústria	Indústria	Indústria	Indústria	Indústria	Indústria
	2.2	Tempo de experiência (anos)	4	4	1	8	3	2
	2.3	Tamanho máximo da equipe (pessoas)	11	11	10	35	30	20
	2.4	2.4.1 Experiência em Java (0-4)	4	4	4	4	4	4
		2.4.2 Experiência em Hibernate (0-4)	4	4	4	4	4	4
		2.4.3 Experiência em Spring (0-4)	4	4	4	4	4	4
		2.4.4 Experiência em Mockito (0-4)	4	0	3	4	0	1
3	3.1	Tempo de experiência no sistema objeto de estudo - IdUFF	3	2	1	4	3	2
	3.2	Cargo na equipe (idUFF)	Gerente	Gerente	Desenvolvedor	Diretor	Gerente	DBA

Tabela 1: Resumo do Questionário de Caracterização dos participantes.

interpretação dos resultados obtidos por cada um dos participantes.

A Tabela 1 apresenta um resumo da caracterização desses participantes. Os níveis de experiência variam de 0 a 4, onde esses números representam:

- 0 - nenhuma experiência;
- 1 - estudou em aula ou livro;
- 2 - praticou em projetos em sala de aula;
- 3 - usou em projetos pessoais;
- 4 - usou em projetos na indústria;

Na situação proposta aos participantes, um cenário fictício foi sugerido, onde os mesmos estariam programando uma linha de código e o *plugin* VCC iria sugerir algum código complementar. Por exemplo, se o usuário estivesse chamando o método *br.uff.iduff2.modelo.academico.Turma.getDisciplina()*, o *plugin* VCC iria exibir um padrão, sugerindo que desenvolvedores que chamam esse método também chamam o método *br.uff.iduff2.modelo.academico.Disciplina.getCargaHorariaTeorica()*.

Assim, 10 padrões de código foram escolhidos, do total de 424 padrões minerados através da utilização do *plugin* VCC, para serem analisados pelos desenvolvedores. Esses padrões foram obtidos utilizando-se um suporte mínimo de 0,3%, após testes com diversos valores, por apresentar um número razoável de sequências mineradas e ter um tempo de processamento relativamente rápido, de aproximadamente 15 segundos. O critério de escolha dos 10 padrões ado-

Questões	Chamada do método	Sugestão
1)	<code>org.mockito.internal.progress.NewOngoingStubbing<Integer>.thenReturn(java.lang.Integer)</code>	<code>org.mockito.Mockito.when(java.lang.Integer)</code>
2)	<code>br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)</code>	<code>java.lang.Throwable.getMessage()</code>
3)	<code>br.uff.commons.utils.oracle.ConexaoOracle.getInstance()</code>	<code>java.sql.Connection.prepareStatement(java.lang.String)</code>
4)	<code>org.hibernate.Query.setParameter(l, java.lang.Object)</code>	<code>org.hibernate.Query.list()</code>
5)	<code>br.uff.iduff2.relatorio.RelatorioFactory.getRelatorio(l)</code>	<code>br.uff.iduff2.relatorio.Relatorio.gerarRelatorio(java.util.List, java.util.Map)</code>
6)	<code>br.uff.iduff2.modelo.academico.Turma.getDisciplina()</code>	<code>br.uff.iduff2.modelo.academico.Disciplina.getCargaHorariaTeorica()</code>
7)	<code>br.uff.publico.core.model.Identificacao.getNome()</code>	<code>br.uff.publico.core.model.Identificador.getIdentificacao()</code>
8)	<code>java.lang.String.isEmpty()</code>	<code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code>
9)	<code>java.util.logging.Logger.log(java.util.logging.Level, java.lang.String, java.lang.Throwable)</code>	<code>java.lang.Throwable.getMessage()</code>
10)	<code>br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)</code>	<code>java.lang.Throwable.getMessage()</code>

Tabela 2: Questões com as sugestões do Questionário de Avaliação

tados foi buscar por padrões de confiança alta e confiança baixa, o que permitiu uma avaliação a partir dessas informações. Cada um desses padrões é exibido na tabela 2.

Os voluntários então preencheram o Questionário de Avaliação (Apêndice III), por meio do qual expressavam sua opinião com relação a cada sugestão, podendo optar pelas seguintes respostas: não sei, discordo totalmente, discordo parcialmente, concordo parcialmente e concordo plenamente. As respostas foram dispostas exatamente nessa ordem de aceitação da sugestão.

O tempo esperado para o preenchimento de todos os documentos necessários para a participação no experimento era de 15 a 20 minutos.

5.4 RESULTADOS OBTIDOS

Como visto na seção 5.3, 6 voluntários participaram do experimento, todos eles com experiência em desenvolvimento de software. Os resultados obtidos do questionário de avaliação foram, em geral, positivos com relação à utilidade das sugestões de código informadas. Grande parte dos voluntários se mostrou interessada no *plugin*, e gostaria de vê-lo sendo utilizado em seu ambiente de trabalho, se disponibilizando para ajudar a implementar as alterações necessárias para que o mesmo seja utilizado na IDE *NetBeans*.

5.4.1 ANÁLISE QUANTITATIVA

Das 60 respostas obtidas a partir do preenchimento do questionário de avaliação pelos voluntários, 43 foram positivas, sendo 11 marcadas como *concordo parcialmente* e 32 marcadas como *concordo totalmente*, indicando assim um índice de 71,66% de aceitação das sugestões de código fonte. Foram 12 respostas negativas, sendo 6 marcadas como *discordo totalmente* e 6 marcadas como *discordo parcialmente*, indicando um índice de 20% de reprovação das su-

gestões apresentadas. Do total das 60 respostas, 5 respostas foram marcadas como *não sei*, sendo 8,33% desse total. A figura 9 apresenta o total de respostas para cada item que poderia ser escolhido. A tabela 3 apresenta um resumo detalhado das respostas, em porcentagem, de cada questão.

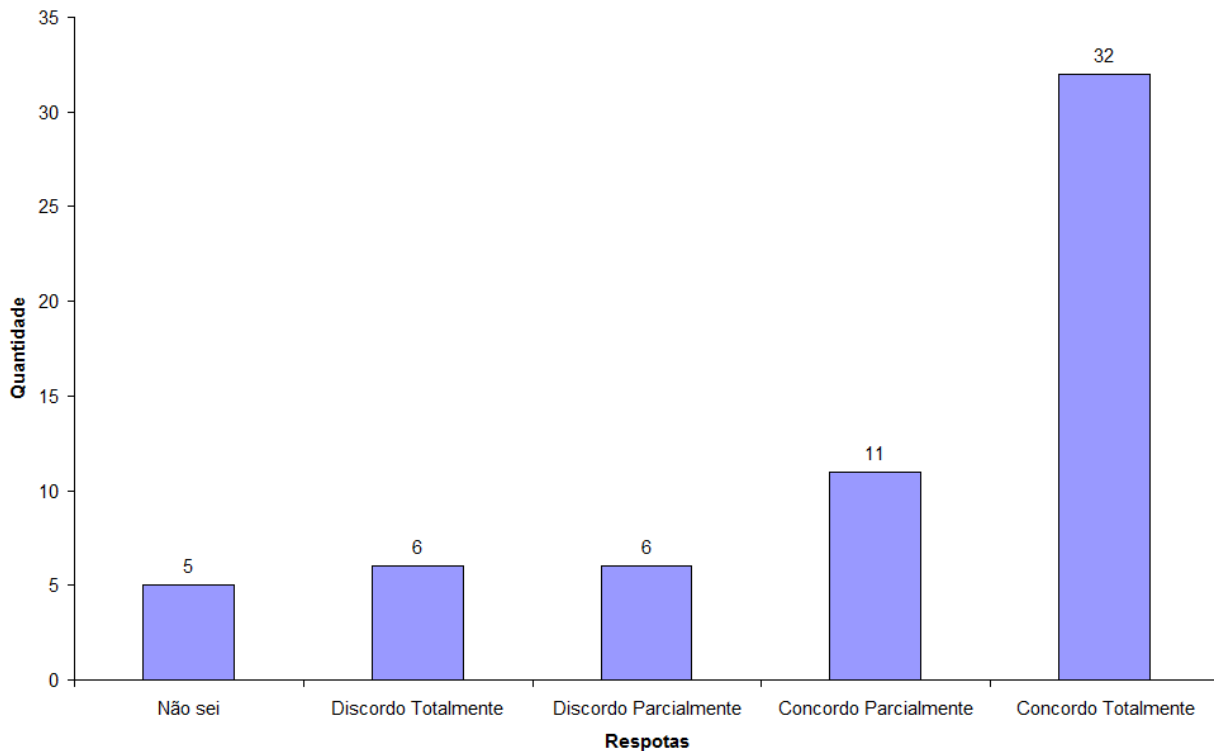


Figura 9: Total de respostas dos participantes para cada item.

Foi feita também uma análise, relacionando a experiência dos desenvolvedores voluntários e o valor de confiança dos padrões encontrados. Para essa análise, foram considerados 2 grupos de desenvolvedores: experientes, com 3 ou mais anos de experiência no sistema IdUFF, e não experientes, com 2 anos ou menos de experiência. Cada um dos grupos ficou com 3 desenvolvedores dentro desse perfil de experiência. As sugestões também foram separadas em 2 grupos: sugestões com confiança alta, com valores iguais ou superiores à 60%, e sugestões com confiança baixa, com valores abaixo de 60%.

Os gráficos das figuras 10, 11, 12 e 13 relacionam a experiência dos desenvolvedores com a confiança dos padrões e permitem a visualização do total de respostas dos voluntários para cada item.

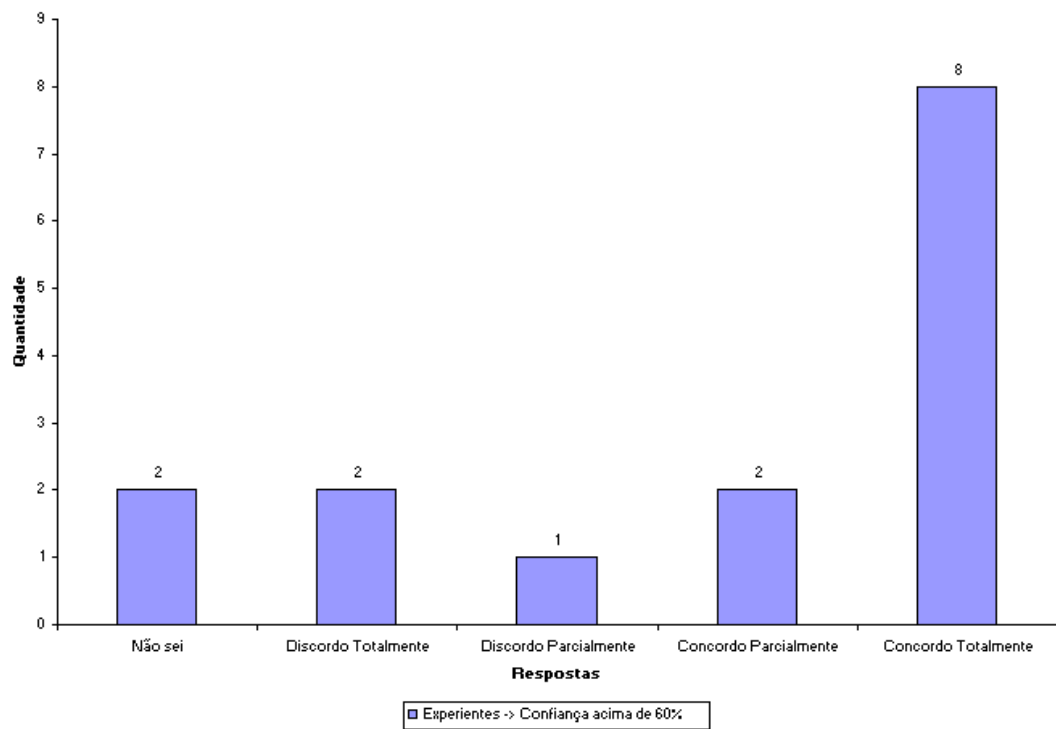


Figura 10: Total de respostas, por item, dos participantes experientes nas questões com padrões de confiança alta.

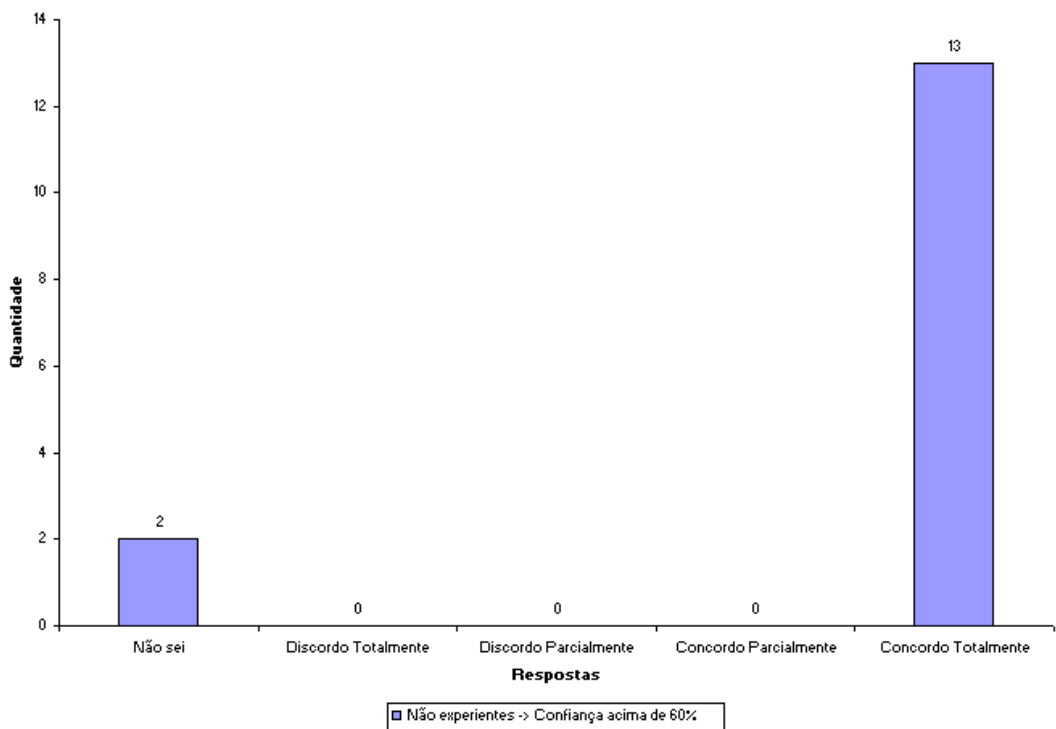


Figura 11: Total de respostas, por item, dos participantes não experientes nas questões com padrões de confiança alta.

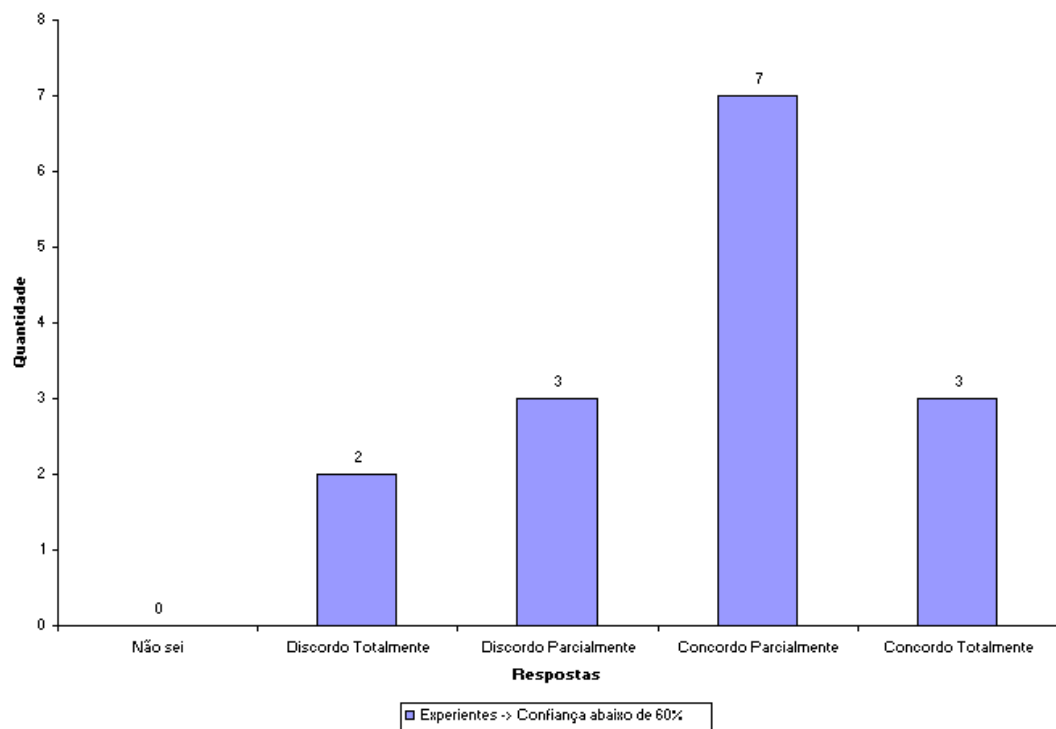


Figura 12: Total de respostas, por item, dos participantes experientes nas questões com padrões de confiança baixa.

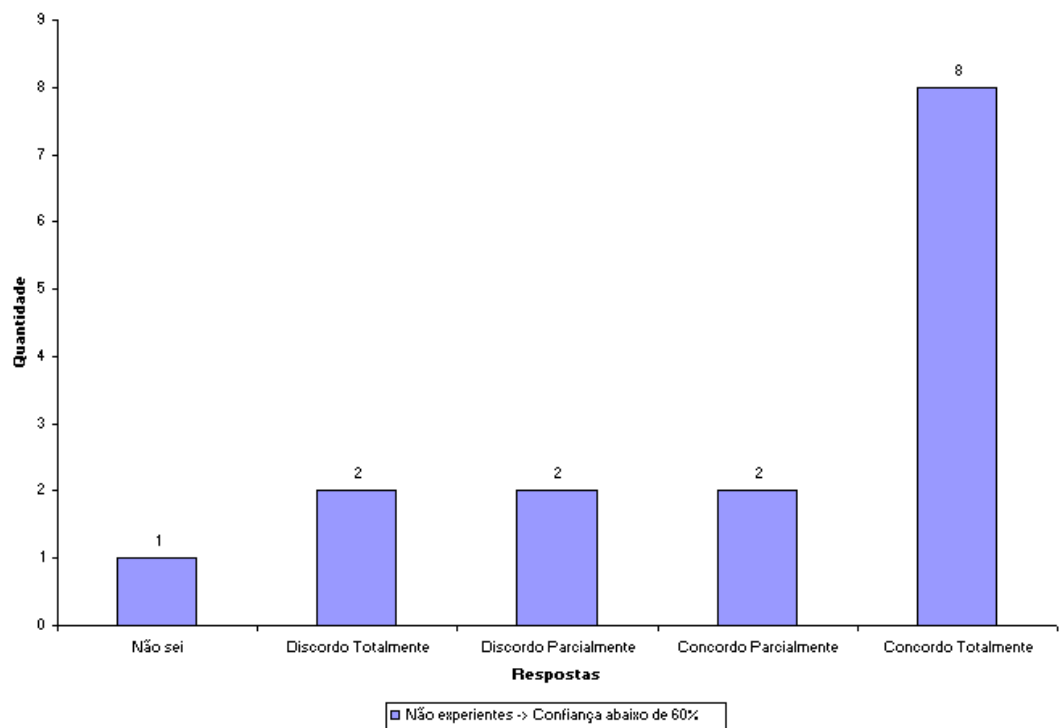


Figura 13: Total de respostas, por item, dos participantes não experientes nas questões com padrões de confiança alta.

Padrão	Suporte	Confiança	Respostas				
			Não sei	Discordo Totalmente	Discordo Parcialmente	Concordo Parcialmente	Concordo Totalmente
Questão 1	0.73%	100.00%	50.00%	0	0	0	50.00%
Questão 2	2.36%	54.66%	0	0	0	16.67%	83.33%
Questão 3	0.33%	100.00%	0	0	0	0	100.00%
Questão 4	0.50%	83.33%	0	16.67%	16.67%	16.67%	50.00%
Questão 5	0.35%	96%	0	0	0	0	100.00%
Questão 6	0.35%	32.43%	0	33.33%	50.00%	16.67%	0
Questão 7	0.32%	62.86%	16.67%	16.67%	0	16.67%	50.00%
Questão 8	0.37%	44.07%	16.67%	16.67%	33.33%	16.67%	16.67%
Questão 9	0.29%	48.78%	0	0	0	33.33%	66.67%
Questão 10	0.32%	15.17%	0	16.67%	0	66.67%	16.67%

Tabela 3: Resumo das respostas de cada questão, em porcentagem.

5.4.2 ANÁLISE QUALITATIVA

Com relação as questões com sugestões de confiança alta, os voluntários não experientes se mostraram muito satisfeitos, respondendo *concordo plenamente* em 13 das 15 respostas que foram apresentadas para esse grupo de desenvolvedores. As outras 2 respostas foram marcadas como *não sei*. Já os desenvolvedores experientes, responderam de forma positiva em 10 das 15 respostas. Esses números servem como indício que as sugestões com confiança alta tiveram um índice de aceitação bom, tanto para os desenvolvedores experientes quanto para os não experientes.

As questões que apresentaram sugestões de confiança baixa, como esperado, foram as que mais receberam respostas negativas. Foram 5 respostas negativas dos desenvolvedores experientes e 4 dos não experientes, totalizando 9 dessas respostas. Com relação às respostas positivas, os desenvolvedores experientes foram os que mais fizeram ressalvas, marcando 7 questões como *concordo parcialmente* e 3 como *concordo plenamente*. Já os desenvolvedores não experientes marcaram 2 questões como *concordo parcialmente* e 8 como *concordo plenamente*. Esses números mostram que, para os desenvolvedores experientes, as sugestões com confiança baixa se mostraram, em geral, úteis, porém com a utilização com restrições.

As questões 6 e 8 foram as que mais receberam respostas negativas, contabilizando 8 do total de 12 respostas nesse contexto. Dessas 8 respostas, 5 foram marcadas por desenvolvedores experientes. Esse resultado já era esperado já que essas questões foram escolhidas por apresentarem sugestões com confiança baixa. Os voluntários que marcaram a questão 6 negativamente, comentaram que a sugestão apresentada não fazia sentido por não ser muito usada, e sugeriram

outros padrões de chamada de método.

A questão 10 recebeu a resposta *discordo totalmente* por um dos voluntários e o comentário do mesmo foi pertinente à sugestão apresentada: “Se usa o método *BaseMB.info(String)* para mensagens que indiquem sucesso, não faz sentido exibir uma mensagem proveniente de uma situação de erro. Na minha opinião, essa regra apresentada é aplicável aos métodos *BaseMB.error(String)* e *BaseMB.warn(String)*”. Esse comentário reflete uma das limitações desse projeto: não levar em conta blocos de repetição, condição e tratamento de erros, como no caso da estrutura *try catch*. Geralmente o método *BaseMB.info(String)* é usado dentro de uma estrutura *try*, e é executado se houver sucesso na operação, enquanto o método *java.lang.Throwable.getMessage()* é utilizado dentro de uma estrutura *catch*, indicando um erro sendo lançado na aplicação. Sendo assim, se a estrutura *try catch* fosse levada em consideração ao se buscar os padrões sequências, essa sugestão não seria minerada, já que o método *java.lang.Throwable.getMessage()* não se enquadraria como uma sequência para o método *BaseMB.info(String)*.

A questão 1 foi a que mais recebeu respostas *não sei*, contabilizando 3 de um total de 5. Isso pode ser justificado pelo questionário de caracterização, onde os voluntários que responderam essa questão dessa maneira são os que indicaram ter pouca ou nenhuma experiência com o *framework* de testes *Mockito*. A sugestão apresentada está relacionada a esse *framework*. As outras respostas obtidas para essa questão foram bem satisfatórias: 3 respostas *concordo plenamente*. Essa questão apresentava uma sugestão com confiança alta, com valor de 100%. Os desenvolvedores que comentaram essa questão disseram que a sugestão apresentada se dá por conta da estrutura do *framework Mockito* e seria bastante útil sua utilização.

As questões que foram respondidas como *concordo totalmente* e *concordo parcialmente* em geral não receberam comentários.

5.5 AMEAÇAS À VALIDADE

A seleção dos participantes foi feita por meio da solicitação de voluntários dentro da equipe de desenvolvimento do sistema IdUFF, a qual um dos pesquisadores faz parte como desenvolvedor. Como consequência, os resultados podem ter sido influenciados pela relação de amizade entre os voluntários e o pesquisador. O pequeno número de participantes foi outro fator importante, pois é possível que os resultados sejam influenciados pelo tamanho e pelas características específicas do grupo.

A quantidade de sugestões avaliadas pelos voluntários era pequena e simplificada, exibindo somente uma sugestão onde poderiam ser sugeridas várias outras. Isso foi necessário devido à

limitação de tempo para o preenchimento do questionário por meio dos voluntários e também para análise dos resultados.

CAPÍTULO 6 - CONCLUSÕES

Nesse trabalho foi apresentada uma abordagem complementar a técnica de *code completion*, o Vertical Code Completion que utiliza a Mineração de Dados para extrair padrões sequenciais frequentes e sugerir linhas de código de fonte complementares, de acordo com o que já foi programado. Assim, diferentemente do *code completion*, que fornece ao desenvolvedor apenas sugestões sintáticas de código fonte, o Vertical Code Completion traz sugestões semânticas.

Como contribuição direta desse trabalho, foi desenvolvido um plugin que auxilia na produção de código fonte, trazendo potenciais resultados positivos no desenvolvimento de software, como a exibição de conhecimento implícito ao código. Isso pôde ser visto na utilização do *plugin* em um sistema real, o IdUFF. Os resultados obtidos nesse experimento foram positivos, indicando sucesso na abordagem proposta e na sua implementação, o *plugin* VCC.

O conceito de confiança na mineração de padrões sequenciais frequentes foi outra contribuição apresentada nessa monografia. Esse conceito foi derivado da tarefa de regras de associação e sua definição é análoga à dessa tarefa, fornecendo mais uma métrica para avaliação dos padrões sequenciais obtidos.

Contudo, algumas limitações e trabalhos futuros foram identificados. A falta de um ambiente com um número maior de desenvolvedores disponíveis para responder ao questionário de sugestões da ferramenta, desfavoreceu a confiabilidade do experimento, tornando-se uma limitação nessa monografia. Além disso, um possível trabalho futuro, que pode aumentar a qualidade das sugestões produzidas pelo VCC, é a análise do código fonte levando em consideração as estruturas de repetição e principalmente de condição. Com isso, uma estrutura condicional, que possui mais de uma alternativa, não seria vista como uma única sequência de chamadas, e sim como sequências independentes. Além disso, outra possível melhoria, com o intuito de aumentar a usabilidade da ferramenta, é o preenchimento automático do código fonte no editor da IDE em resposta à uma sugestão aceita, ao invés de exibir as sugestões em uma janela *popup*.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] T. Ball, J. Kim e A.A. Porter. If your version control system could talk. *Workshop on Process Modelling and Empirical Studies of Software Engineering, Boston, MA*, Maio de 1997.
- [2] C. Dantas, L. Murta e C. Werner. Mining change traces from versioned uml repositories. *XXI Simpósio Brasileiro de Engenharia de Software (SBES 2007). João Pessoa, Brasil*, pp. 236 – 252, Outubro de 2007.
- [3] H. M. Deitel. *Java, How To Program*. Prentice Hall, 4 edição, 2002.
- [4] M. A. Domingues. Generalização de regras de associação. Dissertação de Mestrado, USP - São Carlos, Fevereiro de 2004.
- [5] C. I. Ezeife, Y. Lu e Y. Liu. Plwap sequential mining: Open source code. *International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations. Chicago, USA*, pp. 26–35, Agosto de 2005.
- [6] E.C. Gonçalves. Regras de associação e suas medidas de interesse objetivas e subjetivas. *INFOCOMP Journal Computer Science - Volume 4, Issue 1*, pp. 27–36, Março de 2005.
- [7] J. Han e M. Kamber. *Data Mining: Concepts and Techniques (2nd edition)*. Morgan Kaufmann, 2006.
- [8] R. Holmes e G. C. Murphy. Using structural context to recommend source code examples. *27th International Conference on Software Engineering. St. Louis, USA*, pp. 117 – 125, Maio de 2004.
- [9] W. Holz, R. Premraj, T. Zimmermann e A. Zeller. Predicting software metrics at design time. *9th International conference on Product-Focused Software Process Improvement. Rome, Italy*, pp. 34 – 44, Junho de 2008.
- [10] W. Melo, C. B. S. Holanda e C. A. A. de Souza. Prereuso: um repositório de componentes para web dirigido por um processo de reuso. *XV Simpósio Brasileiro de Engenharia de Software. Rio de Janeiro, Brasil*, pp. 208–223, Outubro de 2001.
- [11] G. C. Murphy, M. K. e L. Findlater. How are java software developers using the eclipse ide? *IEEE Software - Volume 23, Issue 4*, pp. 76–83, Julho de 2006.
- [12] A. C. Myers, J. A. Bank e B. Liskov. Parameterized types for java. *24th ACM Symposium on Principles of Programming Languages (POPL). Paris, France*, pp. 132–145, Janeiro de 1997.
- [13] L. Opyrchal e A. Prakash. Efficient object serialization in java. *19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/ Middleware. Austin, USA*, pp. 96–101, Janeiro de 1999.

- [14] J. Palsberg e C. B. Jay. The essence of the visitor pattern. *22nd IEEE. Int. Computer Software and Applications Conf. (COMPSAC). Vienna, Austria*, pp. 9–15, Agosto de 1998.
- [15] I. Parberry e W. Gasarch. *Problems on Algorithms*. Prentice Hall, 2002.
- [16] R. Robbes e M. Lanza. How program history can improve code completion. *23rd IEEE/ACM International Conference on Automated Software Engineering. L'Aquila, Italy*, pp. 317–326, Setembro de 2008.
- [17] J.S. Shirabad, T. Lethbridge e S. Matwin. Supporting software maintenance by mining software update records. *International Conference on Software Maintenance (ICSM). Florence, Italy*, pp. 22–31, Novembro de 2001.
- [18] R. Srikant e R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. *5th International Conference on Extending Database Technology - EDBT. Avignon, France*, pp. 3–17, Março de 1996.
- [19] A.T.T. Ying, G.C. Murphy, R. Ng e M. C. Chu-Carroll. Predicting source code changes by mining change history. *30th IEEE Transactions on Software Engineering (TSE)*, pp. 574–586, Setembro de 2004.
- [20] B.Y. Zhou, S.C. Hui, e A.C.M. Fong. Cs-mine: An efficient wap-tree mining for web access patterns. *6th Asia Pacific Web Conference. Hangzhou, China*, pp. 523–532, Abril de 2004.
- [21] T. Zimmermann, P. Weisgerber, S. Diehl e A. Zeller. Mining version histories to guide software changes. *26th International Conference on Software Engineering (ICSE). Edinburgh, Scotland*, pp. 563–572, Maio de 2004.

APÊNDICE I

Formulário de Consentimento

Estudo

Este estudo visa avaliar o quanto as sugestões de códigos, informadas através do plugin Vertical Code Completion (VCC), são consistentes e fazem sentido no desenvolvimento, tanto para um desenvolvedor experiente quanto para um desenvolvedor novo na equipe.

Idade

Eu declaro ter mais de 18 anos de idade e concordar em participar de um estudo conduzido por Thiago Nazareth de Oliveira e Luiz Laerte Nunes da Silva Junior na Universidade Federal Fluminense.

Procedimento

Este estudo acontecerá em uma única sessão, que incluirá a análise de algumas sugestões de código, retiradas do sistema IdUFF através do plugin VCC. Eu entendo que, uma vez o experimento tenha terminado, os trabalhos que desenvolvi serão estudados visando entender a eficiência dos procedimentos e as técnicas propostas.

Confidencialidade

Toda informação coletada neste estudo é confidencial, e meu nome não será divulgado. Da mesma forma, me comprometo a não comunicar os meus resultados enquanto não terminar o estudo, bem como manter sigilo das técnicas e documentos apresentados e que fazem parte do experimento.

Benefícios e liberdade de desistência

Eu entendo que os benefícios que receberei deste estudo são limitados ao aprendizado do material que é distribuído e apresentado. Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada à minha pessoa não seja incluída no estudo. Eu entendo que participo de livre e espontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas e processos para a Engenharia de Software.

Pesquisadores responsáveis

Thiago Nazareth de Oliveira

Instituto de Computação - Universidade Federal Fluminense (UFF)

Luiz Laerte Nunes da Silva Junior

Instituto de Computação - Universidade Federal Fluminense (UFF)

Professores responsáveis

Prof. Leonardo Paulino Gresta Murta

Instituto de Computação - Universidade Federal Fluminense (UFF)

Prof. Alexandre Plastino

Instituto de Computação - Universidade Federal Fluminense (UFF)

Nome (em letra de forma): _____

Assinatura: _____ **Data:** _____

APÊNDICE II

Questionário de Caracterização

Este formulário contém algumas perguntas sobre sua experiência acadêmica e profissional.

1) Formação Acadêmica

- ☐ Doutorado
- ☐ Doutorando
- ☐ Mestrado
- ☐ Mestrando
- ☐ Graduação
- ☐ Graduando

Ano de ingresso: _____ Ano de conclusão (ou previsão de conclusão): _____

2) Formação Geral

2.1) Qual é sua experiência com desenvolvimento de software? (marque aqueles itens que melhor se aplicam)

- ☐ Nunca desenvolvi software.
- ☐ Já li material sobre desenvolvimento de software.
- ☐ Já participei de um curso sobre desenvolvimento de software.
- ☐ Tenho desenvolvido software para uso próprio.
- ☐ Tenho desenvolvido software como parte de uma equipe, relacionado a um curso.
- ☐ Tenho desenvolvido software como parte de uma equipe, na indústria.

2.2) Por favor, explique sua resposta. Inclua o número de semestres ou número de anos de experiência relevante em desenvolvimento de software. (E.g. “Eu trabalhei por 2 anos como programador de software na indústria”)

2.3) Qual é sua experiência com desenvolvimento de software em equipes? Qual a maior equipe de que você participou?

2.4) Por favor, indique o grau de sua experiência nesta seção seguindo a escala de 5 pontos abaixo:

0 = nenhum

1 = estudei em aula ou em livro

2 = pratiquei em projetos em sala de aula

3 = usei em projetos pessoais

4 = usei em projetos na indústria

2.4.1) Linguagem JAVA

Resposta: _____

2.4.2) Hibernate

Resposta: _____

2.4.3) Spring

Resposta: _____

2.4.4) Mockito

Resposta: _____

3) Experiência no desenvolvimento do sistema objeto de estudo - IdUFF

3.1) Há quanto tempo você está na equipe de desenvolvimento do sistema IDUFF?

Resposta: _____

3.2) Qual o seu cargo na equipe?

Resposta: _____

APÊNDICE III

Questionário de Avaliação

Imagine que você está codificando e que acabou de escrever uma linha de código para abrir uma conexão com o banco de dados e que a IDE que você está utilizando sugira algo do tipo: desenvolvedores que abrem uma conexão com o banco de dados também fecham a conexão com o banco de dados. Este é o objetivo do *plugin Vertical Code Completion*, que estamos desenvolvendo como projeto final: completar o código com sugestões simples (como no exemplo anterior), assim como com sugestões mais complexas, indo além do atual Ctrl-Espaço.

Para avaliar o quanto as sugestões de códigos informadas são consistentes e fazem sentido no desenvolvimento, escolhemos você, desenvolvedor, que faz parte da equipe do IdUFF. Para isso, mineramos o repositório de códigos do sistema IdUFF para encontrar possíveis padrões de desenvolvimento.

Para medir a utilidade das sugestões encontradas, tanto para um desenvolvedor experiente, quanto para um desenvolvedor novo na equipe, as seguintes respostas podem ser dadas: não sei, discordo totalmente, discordo parcialmente, concordo parcialmente, concordo totalmente.

1) Usuários que chamam o método:

org.mockito.internal.progress.NewOngoingStubbing;Integer;.thenReturn(java.lang.Integer)

Também costumam chamar:

org.mockito.Mockito.when(java.lang.Integer)

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

2) Usuários que chamam o método:

br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)

Também costumam chamar:

java.lang.Throwable.getMessage()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

3) Usuários que chamam o método:

br.uff.commons.utils.oracle.ConexaoOracle.getInstance()

Também costumam chamar:

java.sql.Connection.prepareStatement(java.lang.String)

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

4) Usuários que chamam o método:

org.hibernate.Query.setParameter(I, java.lang.Object)

Também costumam chamar:

org.hibernate.Query.list()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

5) Usuários que chamam o método:

br.uff.iduff2.relatorio.RelatorioFactory.getRelatorio(I)

Também costumam chamar:

br.uff.iduff2.relatorio.Relatorio.gerarRelatorio(java.util.List, java.util.Map)

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

6) Usuários que chamam o método:

br.uff.iduff2.modelo.academico.Turma.getDisciplina()

Também costumam chamar:

br.uff.iduff2.modelo.academico.Disciplina.getCargaHorariaTeorica()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

7) Usuários que chamam o método:

br.uff.publico.core.model.Identificacao.getNome()

Também costumam chamar:

br.uff.publico.core.model.Identificador.getIdentificacao()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

8) Usuários que chamam o método:

java.lang.String.isEmpty()

Também costumam chamar:

br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

9) Usuários que chamam o método:

java.util.logging.Logger.log(java.util.logging.Level, java.lang.String, java.lang.Throwable)

Também costumam chamar:

java.lang.Throwable.getMessage()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____

10) Usuários que chamam o método:

br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)

Também costumam chamar:

java.lang.Throwable.getMessage()

- ☐ não sei
- ☐ discordo totalmente
- ☐ discordo parcialmente
- ☐ concordo parcialmete
- ☐ concordo totalmente

Comentário: _____