Rodrigo Vieira de Moraes

IPS: um plug-in para templates de código associativos

Sorocaba, SP

25 de Novembro de 2016

Rodrigo Vieira de Moraes

IPS: um plug-in para templates de código associativos

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Teoria Aplicada à Computação.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Dr. José de Oliveira Guimarães

Sorocaba, SP

25 de Novembro de 2016

MORAES, RODRIGO

IPS: um plug-in para templates de código associativos / RODRIGO MORAES. -- 2016.
144 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: José de Oliveira Guimarães
Banca examinadora: Alan Mitchell Durham,Mateus Conrad Barcellos da Costa,Gustavo Maciel Dias Vieira
Bibliografia

1. Documentação de API. 2. Reutilização de software. 3. Trechos de código. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de dissertação de mestrado do candidato Rodrigo Vieira de Moraes, realizada em 25/11/2016:

José de Oliveira Guimarães

Prof. Dr. José de Oliveira Guimarães
UFSCar

Prof. Dr. Alan Mitchell Durham
USP

Prof. Dr. Mateus Conrad Barcellos da Costa
IFES

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Certifico que a sessão de defesa foi realizada com a participação à distância dos membros Prof. Dr. Alan Mitchell Durham e Prof. Dr. Mateus Conrad Barcellos da Costa e, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Rodrigo Vieira de Moraes.

José de Oliveira Guimarães

Prof. Dr. José de Oliveira Guimarães
Presidente da Comissão Examinadora
UFSCar

Dedico este trabalho ao meu filho Vinícius, meu maior Professor, que a cada dia me ensina a ser criança novamente dando lições de imaginação e criatividade.

Agradecimentos

Agradeço,

À Deus por ter criado este mundo para nos inspirar, para que possamos conhecer uns aos outros e para nossa evolução.

À minha família, que é a minha principal motivação, pelos momentos que passamos juntos e pela paciência nos momentos que estive ausente.

Aos meus pais, que me criaram e formaram parte da minha personalidade, o que possibilitou eu ter a sabedoria e a perseverança necessária para concluir este trabalho.

Ao meu orientador Prof. Dr. José de Oliveira Guimarães pela oportunidade, dedicação, compartilhamento de informações e conselhos necessários para a realização deste trabalho.

Aos professores Dr. Alan Mitchell Durham, Dr. Gustavo Maciel Dias Vieira e Dr. Mateus Conrad Barcellos da Costa por terem aceitado o convite para participar da banca examinadora, assim como por suas críticas, sugestões e comentários.

Aos professores que tive ao longo da vida que me deram exemplos de dedicação.

Aos professores da UFSCar: Prof. Dr. Alexandre Alvaro, Profa. Dra. Cândida Nunes da Silva e Profa. Dra. Sahudy Montenegro González que souberam colocar os desafios necessários nas disciplinas cursadas para que eu pudesse me amadurecer como aluno e como profissional.

"Nada é seu.
É para ser usado, para ser partilhado.
Se você não quiser compartilhar com os outros, não poderá usar."
(Os Despossuídos, Ursula K. Le Guin)

Resumo

As *APIs* têm se tornado cada vez mais populares e elas têm um papel fundamental no desenvolvimento de *software*. No entanto, muitas *APIs* são grandes e complexas o que torna a aprendizagem delas difícil. Entre os recursos disponíveis para se aprender a utilizar uma *API*, há os exemplos de código. Um exemplo de código pode demonstrar a um programador como componentes de uma *API* podem ser utilizados juntos para atingir um determinado objetivo. Porém, exemplos de código são estáticos e não contêm informações de como adaptar o trecho de código apresentado como exemplo para o contexto do programador. Ferramentas têm sido propostas para auxiliar os programadores a encontrarem e adaptarem trechos de códigos que possam ser úteis para uma tarefa de programação. Um tipo específico dessas ferramentas são as ferramentas de *templates* de código. *Templates* de código são trechos de códigos cadastrados manualmente contendo diretivas de integração. As diretivas de integração instruem a ferramenta a como adaptar um trecho de código para o contexto do programador. Neste trabalho, ferramentas de trechos de código são investigadas e é proposta uma ferramenta de *template* de código denominada *IPS*. Ela apresenta novidades em relação às outras ferramentas de *templates* de código. No *IPS*, um *template* de código pode ser associado a uma classe *Java* e o *IPS* possui um recomendador de *templates* de código que é automaticamente atualizado conforme o código fonte é editado pelo programador. Também é proposto os parâmetros do tipo *role*, um novo conceito que estende o conceito dos parâmetros associados a um tipo que está presente nas ferramentas de *templates* de código atuais. Os parâmetros do tipo *role* têm como objetivo aceitar argumentos que seriam a princípio incompatíveis com um determinado tipo, porém que sigam um determinado padrão esperado.

Palavras-chaves: Documentação de API. Exemplos de código. Reutilização de software. Trechos de código. Templates de código.

Abstract

The APIs are becoming increasingly more popular and they play a important role in software development. However, many APIs are large and complex that makes them hard to learn. Among avaliable resources to learn to use an API, there are the code samples. A code sample can be used to demonstrate to a programmer how to use API's components together to attain a specific goal. But, code sample are static and they haven't information about how to adapt the code snippet shown as example to programmer's context. Tools have been proposed to assist the programmers in finding and adapting code snippets that may be useful for a programming task. A specific type of these tools are the code template tools. Code templates are code snippets manually registered containing integration directives. The integration directives instruct the code template tool on how to adapt a code snippet to programmer's context. In this work, code snippet tools are investigated and it is proposed a code template tool called IPS. It presents some novelties in relation to other code template tools. In the IPS, a code template can be associated to a Java class and the IPS has a code template recommender that is automatically updated as the source code is edited by the programmer. Also it is proposed the role parameters, a new concept for the parameters of code templates that extends the concept of the parameters associated to a type which is present in current code template tools. The role parameters has the objective of accepting arguments that would at first be incompatible with a specific type, but that obey a expected pattern.

Key-words: API documentation. Code samples. Software reuse. Code snippets. Code template.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CSACT	*Code Snippets and API Components Tool*
DOM	*Document Object Model*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
IPS	*Intuitive Programming System*
JAR	*Java ARchive*
JAXP	*Java API for XML Processing*
JDT	*Java Development Tools*
PDF	*Portable Document Format*
SWT	*Standard Widget Toolkit*
UML	*Unified Modeling Language*
VSM	*Vector Space Model*
XCSPT	*XML Code SniPpeT*
XML	*eXtensible Markup Language*

Sumário

Introdução

Motivação

Uma *API*, *Application Programming Interface*, é uma interface que desenvolvedores de *software* utilizam para realizar tarefas específicas quando não pretendem envolver-se em detalhes de implementação ou em detalhes de um domínio de problema sobre o qual não dispõem de conhecimento suficiente (ORENSTEIN, 2000).

> Uma *API* suporta reutilização, provê alta abstração que facilita as tarefas de programação e ajuda a padronizar a experiência de programação provendo uma forma única de acessar e interagir com os recursos que a *API* encapsula. Entretanto, as *APIs* têm se tornado cada vez maiores e complexas levantando questões em relação a usabilidade (ROBILLARD, 2009, p. 27).

Robillard (2009, p. 29), ao investigar os obstáculos na aprendizagem da utilização das *APIs*, verificou que o obstáculo crítico é a falta de recursos de aprendizagem ou a disponibilidade de recursos de aprendizagem inadequados, como por exemplo a documentação da *API*. Dentre as recomendações citadas pelo autor, está que a documentação da *API* deverá incluir bons exemplos. A importância dos exemplos como um recurso de aprendizagem de utilização de uma *API* é mencionada também por McLellan et al. (1998), por Shull, Lanubile e Basili (2000) e por Nykaza et al. (2002). McLellan et al. (1998, p. 83-84), ao resumirem o resultado de um de seus experimentos, afirmam que "o código exemplo suportou diferentes atividades de aprendizagem, tal como entender as propostas da biblioteca, seus protocolos de utilização e seus contextos de uso". Não se pode deixar de destacar também a natureza imediatista dos exemplos, programadores muitas vezes utilizam exemplos de código sem mesmo entendê-los e testá-los, é uma maneira rápida de cumprir uma determinada tarefa. Há desenvolvedores que utilizam a *Web* como uma memória externa, ou seja, alguns programadores não memorizam os trechos de código necessários para realizar determinadas tarefas de programação, pois sabem que tais trechos de código sempre estarão disponíveis na *Web*, bastando procurá-los e copiá-los quando necessário (BRANDT et al., 2009a, p. 1593–1594).

No entanto, exemplos como simples trechos de código estáticos possuem suas limitações, pois, sobretudo, apresentam a utilização de determinados objetos de uma *API* para um contexto específico e não fornecem por si só informações adicionais para o usuário da *API* sobre a utilização da *API* no contexto do usuário. Dessa forma, uma das principais dificuldades ao adotar os exemplos, é a dificuldade de adaptar o código existente para necessidades específicas. Robillard (2009, p. 33) cita em seu trabalho que "estudar exemplos

é uma importante estratégia para aprender sobre o projeto (da *API*), entretanto esta abordagem levou a frustrações quando os exemplos não foram bem adaptados para a tarefa". Dessa forma, o desenvolvimento e a evolução de ferramentas que auxiliam a localização e a adaptação de trechos de código são de grande importância para o desenvolvimento de software.

Trabalhos relacionados

Ferramentas de trechos de código, ou exemplos, podem ser divididas em dois grupos principais: um grupo de ferramentas que utilizam técnicas de busca e mineração de dados e outro grupo que utilizam *templates de código* (WIGHTMAN et al., 2012, p. 219). *Templates de código* são trechos de código cadastrados manualmente e a principal diferença é que eles são codificados manualmente por especialistas de uma determinada *API*, ao passo que ferramentas que utilizam técnicas de busca e mineração de dados extraem trechos de código automaticamente de alguma fonte de dados, como de projetos de código aberto disponíveis na *Internet* (BAJRACHARYA et al., 2007), de páginas *web* (HOFFMANN; FOGARTY; WELD, 2007; BRANDT et al., 2010) ou de testes unitários (ZHU et al., 2014). Apesar dos *templates de código* serem codificados manualmente, ferramentas de *templates de código* não possuem os riscos inerentes à extração automática de trechos de código. Em ferramentas que extraem trechos de código automaticamente de uma fonte de dados, o algoritmo de extração baseia-se em heurísticas (WANG et al., 2011, p. 594), não há uma garantia absoluta que um trecho de código prático e útil será extraído. Além disso, *templates de código* são confeccionados exclusivamente com o objetivo de ensinar e facilitar a utilização de uma *API*, logo há uma tendência de possuir uma qualidade melhor e serem mais concisos. A popularidade e a importância das ferramentas de *templates de código* pode ser confirmada pelo fato que atualmente é possível encontrar tal recurso integrado às *IDEs*[1] modernas e populares como o *Visual Studio 2013* (MICROSOFT CORPORATION, 2014), o *Eclipse Kepler* (ECLIPSE FOUNDATION, 2015) e o *Netbeans 8.0* (ORACLE CORPORATION, 2014). No entanto, poucos são os trabalhos acadêmicos que discorreram especificamente sobre ferramentas de *templates de código* e propuseram melhorias para as ferramentas atuais. Na análise bibliográfica realizada neste trabalho, a qual é apresentada no Capítulo 1 desta dissertação, foi possível identificar somente os trabalhos de Wightman et al. (2012) e de Oney e Brandt (2012) como trabalhos que abordaram os *templates de código*.

[1] ***I**ntegrated **D**evelopment **E**nvironment*, um ambiente integrado para desenvolvimento de software.

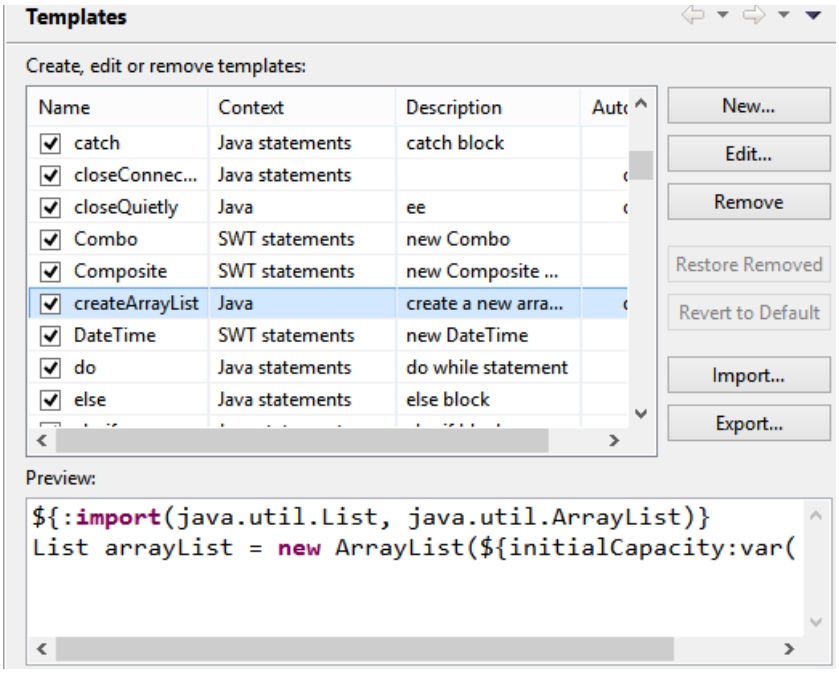
Ferramentas de *templates de código*

Do ponto de vista funcional, ferramentas de *templates de código* podem ser divididas em três aspectos: a criação e manutenção dos *templates*, a busca dos *templates* e a integração dos *templates* no contexto do código fonte em edição. A criação e a manutenção dos *templates* tem como objetivo alimentar e manter um repositório de *templates* que estarão disponíveis para uso na ferramenta. A busca é realizada pelo usuário (um programador utilizando uma *IDE* em um projeto de desenvolvimento de *software*) durante a edição de um código fonte e tem como objetivo encontrar e ativar um *template* desejado. Nesse contexto, o usuário procurará por algum *template* que demonstre como utilizar determinados componentes de uma *API* relevantes para a tarefa que ele esteja trabalhando. Após o *template* desejado ser encontrado e ativado pelo usuário, o trecho de código definido pelo *template* é então adaptado no contexto do código fonte em edição. A fim de exemplificar esses aspectos, nas Figuras 1, 2, 3, 4 e 5 são apresentadas imagens da *IDE Eclipse Kepler*.

No *Eclipse Kepler*, a criação e a manutenção dos *templates de código* são realizadas através de uma interface gráfica que lista o conjunto de *templates* disponíveis para uso na *IDE* (Figura 1). Há a possibilidade de criar um *template* novo, editar um *template* existente ou remover um *template*. Na criação do *template* (item 1, Figura 2), é possível, entre outros atributos, atribuir um nome (atributo *Name*), uma descrição (atributo *Description*) e um trecho de código (atributo *Pattern*). O nome tem a função de identificar o *template*, a descrição resume qual é o objetivo do *template*. O trecho de código do *template* é a combinação entre sentenças *Java* e diretivas de integração. As diretivas de integração são sentenças que informam à ferramenta como o trecho de código deverá ser adaptado no contexto do código fonte em edição ao ativar o *template*. No trecho de código apresentado na Figura 2, há duas diretivas de integração, *import* (item 2 destacado) e *var* (item 3 destacado). A diretiva *import* possui dois argumentos, *java.util.List* e *java.util.ArrayList*, informando que, quando o usuário ativar o *template* para um código fonte *Java* em edição na *IDE*, a ferramenta deverá importar os pacotes *java.util.List* e *java.util.ArrayList*, caso estes já não tenham sido importados no código fonte em edição. A diretiva *var* declara um parâmetro para o *template*. Parâmetros de *templates* são substituídos por seus respectivos argumentos durante a ativação do *template*. No exemplo apresentado (Figura 2), a diretiva *var* (item 3) declara um parâmetro com o nome *initialCapacity* associado ao tipo *int*. A associação de um parâmetro de um *template* a um determinado tipo significa que o parâmetro do *template* aceitará como argumento identificadores de objetos (variáveis locais ou atributos de classe) do tipo associado que estiverem disponíveis no escopo do código fonte em edição durante a ativação do *template*.

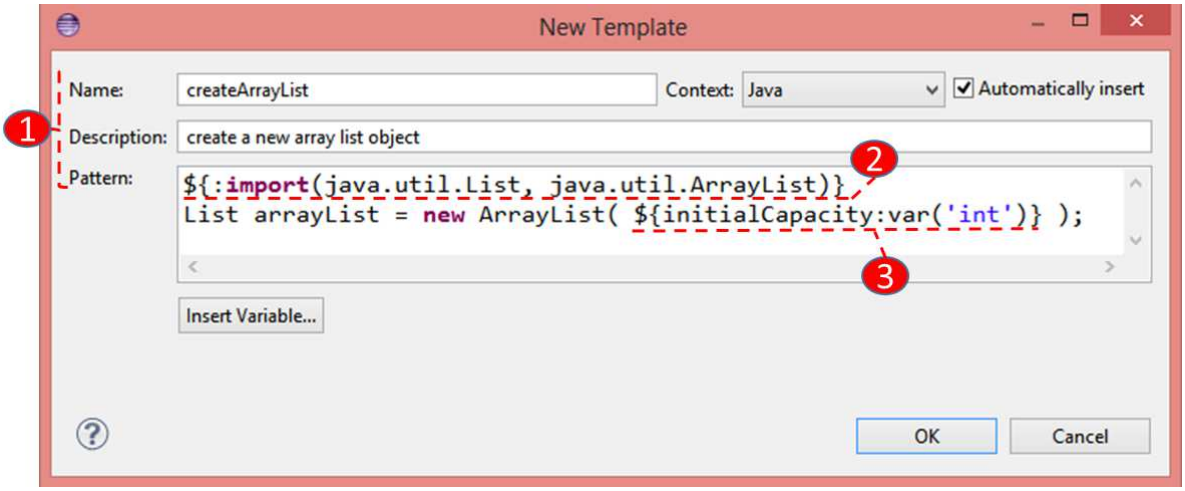
A busca do *template* ocorre quando o usuário está editando um código fonte na *IDE* e aciona o assistente de código. O usuário, então, digita o nome do *template* que deseja utilizar ou uma parte do início do nome. Na Figura 3, é apresentado um cenário de

Figura 1 – Manutenção de *templates* na *IDE Eclipse Kepler*



Fonte: Eclipse Foundation (2015)

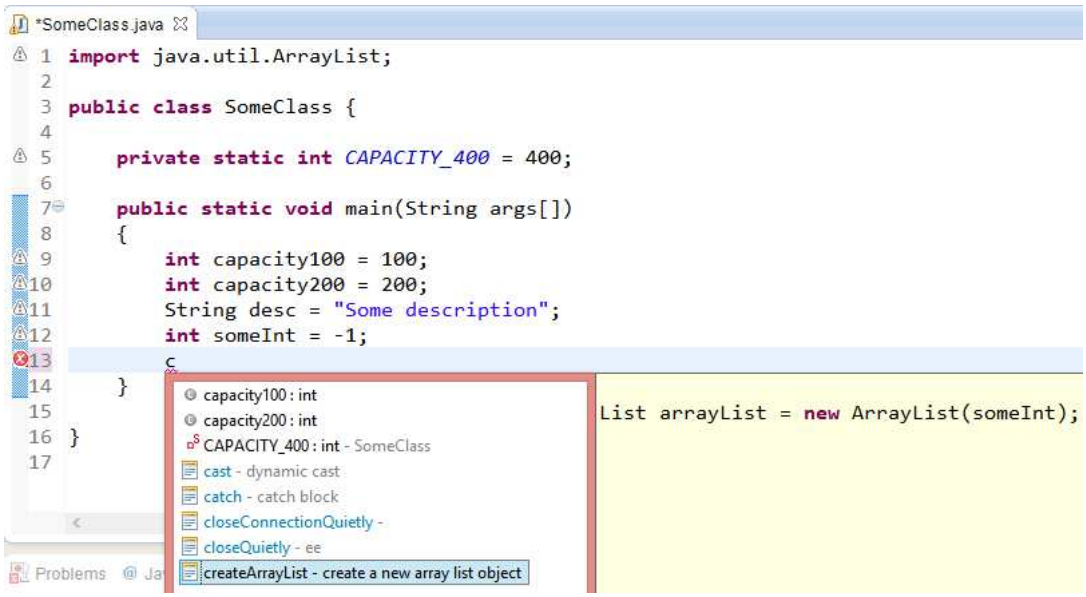
Figura 2 – Criação de *templates* na *IDE Eclipse Kepler*



Fonte: Eclipse Foundation (2015)

exemplo. O usuário está editando o código fonte de um arquivo *Java* (*SomeClass.java*) na *IDE Eclipse Kepler*. Na linha 13, é acionado o assistente de código (através do atalho de teclado *CTRL+ESPAÇO*) e digitado a letra *c*. O assistente de código, então, propõe cinco *templates*, todos eles com o nome iniciando com a letra *c*. Um dos *templates* propostos é o *template* apresentado na Figura 2 de nome *createArrayList*, *template* com a função de criar um novo objeto do tipo *ArrayList* .

Figura 3 – Busca de *templates* na *IDE Eclipse Kepler*



Fonte: Eclipse Foundation (2015)

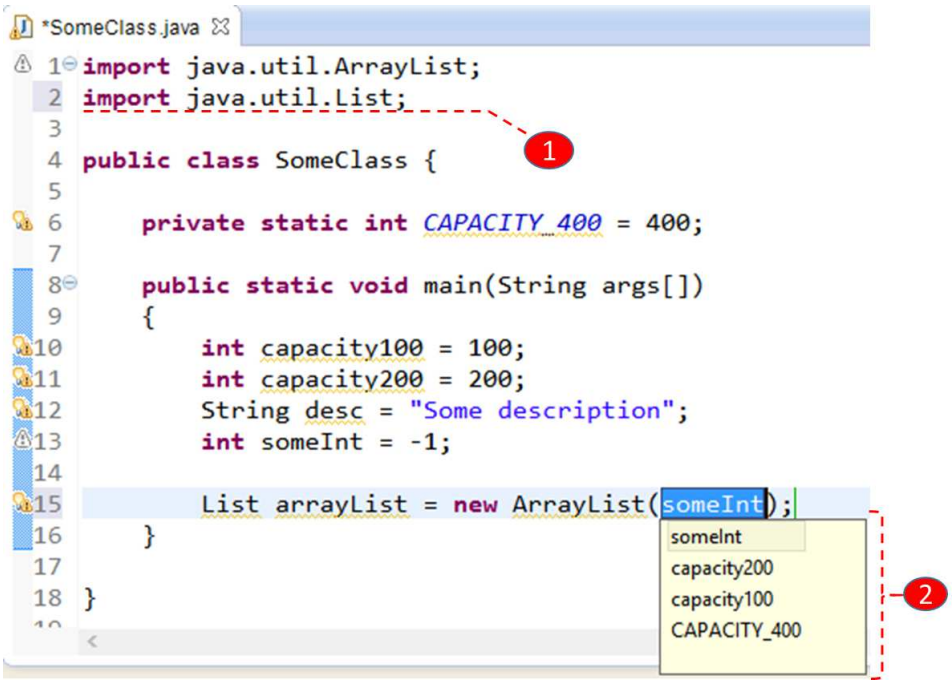
A integração do *template* no contexto do código fonte em edição ocorre quando o usuário seleciona e ativa um *template* desejado. O trecho de código do *template* é, então, copiado e integrado no código fonte em edição de acordo com as diretivas de integração definidas no *template*. Na Figura 4, é apresentado um exemplo que o usuário, após a busca realizada apresentada na Figura 3, seleciona e ativa o *template createArrayList*. Após a ativação do *template*, o trecho de código do *template* é inserido na posição onde o assistente de código foi ativado (linha 15 no código fonte apresentado na Figura 4). Como no *template createArrayList* há uma diretiva *import* (item 2 destacado na Figura 2) com os argumentos *java.util.List* e *java.util.ArrayList*, após a ativação do *template*, o pacote *java.util.List* é importado no código fonte *Java* em edição (item 1 destacado na Figura 4). O pacote *java.util.ArrayList* não é importado, pois já havia sido importado (veja código fonte exibido na Figura 3, antes da ativação do *template de código createArrayList*). Além disso, devido a diretiva *var* (item 3 destacado na Figura 2), a ferramenta sugere identificadores de objetos do tipo *int* disponíveis no escopo do código fonte em edição para serem utilizadas como argumento do parâmetro *initialCapacity* do *template createArrayList* (item 2 da Figura 4). Após o usuário escolher o argumento desejado, a ferramenta substitui o parâmetro pelo respectivo argumento selecionado. Na Figura 5, é apresentado o código fonte após o usuário ter selecionado a variável local *capacity200* como argumento do parâmetro *initialCapacity* do *template createArrayList* (linha 15).

Objetivos e solução proposta

Neste trabalho de mestrado, será apresentado o uso de uma nova abordagem denominada de "parâmetros do tipo *role*", que estende o conceito dos parâmetros associados a um tipo presente nas ferramentas de *templates de código* atuais. Os parâmetros do tipo *role*, os quais serão apresentados de forma mais detalhada na Seção 2.1, tem como objetivo aceitar argumentos que seriam a princípio incompatíveis com uma determinada classe, porém que sigam um determinado padrão esperado. Parâmetros do tipo *role* auxiliam a integração entre objetos de classes pertencentes a uma determinada *API* com objetos de classes pertences ao projeto do usuário da *API*. Um desenvolvedor de uma *API* pode, através do parâmetros do tipo *role*, escrever um *template* com um parâmetro que aceitará como argumento identificadores de objetos (variáveis locais e atributos de classes) cujo o tipo seja uma classe desconhecida pelo autor do *template*. Por exemplo, um especialista de uma *API* para processamento de grafos pode escrever um *template* que implemente um algoritmo de busca em grafos que dependa de uma estrutura de dados do tipo pilha. A classe que implementará a estrutura de dados do tipo pilha não precisa ser conhecida para confeccionar o *template*. Durante a ativação do *template*, as referências no corpo do *template* para a pilha são então adaptadas para uma pilha escolhida pelo usuário do *template*.

A fim de investigar a utilização dos parâmetros do tipo *role*, uma ferramenta, que

Figura 4 – Integração de *template* na *IDE Eclipse Kepler*



Fonte: Eclipse Foundation (2015)

Figura 5 – Código fonte após a seleção do argumento

SomeClass.java

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class SomeClass {
5
6      private static int CAPACITY_400 = 400;
7
8      public static void main(String args[])
9      {
10         int capacity100 = 100;
11         int capacity200 = 200;
12         String desc = "Some description";
13         int someInt = -1;
14
15         List arrayList = new ArrayList(capacity200);
16     }
17 }
```

Fonte: Eclipse Foundation (2015)

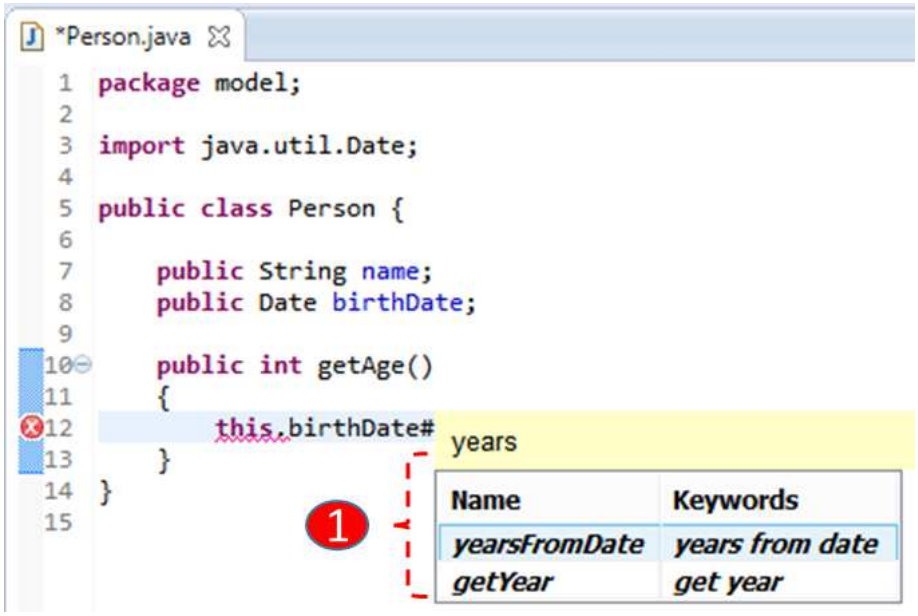
se denomina *IPS* — ***I**ntuitive **P**rogramming **S**ystem* — foi confeccionada. O *IPS* é uma ferramenta integrada ao ambiente de desenvolvimento *Eclipse*[2] e disponibilizada como um *plug-in*. Ela é composta de um editor de *templates de código* e de um assistente de código. Durante a edição de arquivos de código fonte *Java*, no ambiente de programação *Eclipse*, o assistente de código auxilia o desenvolvedor a buscar por um *template* desejado e adaptá-lo no contexto do código fonte em edição. O assistente de código faz uso dos metadados dos *templates de código* a fim de filtrá-los e apresentá-los ao usuário. Além dos parâmetros do tipo *role*, mais duas novidades serão apresentadas neste trabalho. *Templates de código associativos* e um *recomendador de templates de código*.

Diferente de outras ferramentas de *templates de código*, no *IPS* cada *template de código* é associado a uma classe *Java*. Isso permite filtrar os *templates de código* de acordo com os identificadores no escopo do código fonte em edição no ambiente de desenvolvimento *Eclipse*. O usuário pode acionar o *IPS* para um identificador de objeto (variável local ou atributo de classe) de uma determinada classe, informando que ele deseja realizar alguma operação com aquele objeto. Neste caso, a fim de apresentar *templates de código* ao usuário, o *IPS* somente considera os *templates* associados à classe do objeto. Como um exemplo, na Figura 6, é apresentado um cenário de uso que o usuário, durante a edição um código fonte *Java* na *IDE Eclipse*, aciona o assistente de código do *IPS* para o identificador de objeto *birthDate*, atributo da classe *Person* do tipo *java.util.Date*. Para acionar o assistente de

[2] Um ambiente integrado para desenvolvimento de software (ECLIPSE FOUNDATION, 2015).

código para um identificador de objeto, o usuário digita o identificador de objeto seguido do caracter # (linha 12 da Figura 6). O assistente de código do *IPS*, então, sugere dois *templates* (item 1 destacado na Figura 6) associados à classe *java.util.Date*. Em outras ferramentas, essa associação não existe e, portanto, não há a possibilidade da ferramenta de *template* sugerir *templates* para um identificador de objeto. Além disso, no *IPS*, os *templates* associados a uma superclasse também são associados às subclasses.

Figura 6 – Busca de *templates* no *IPS*

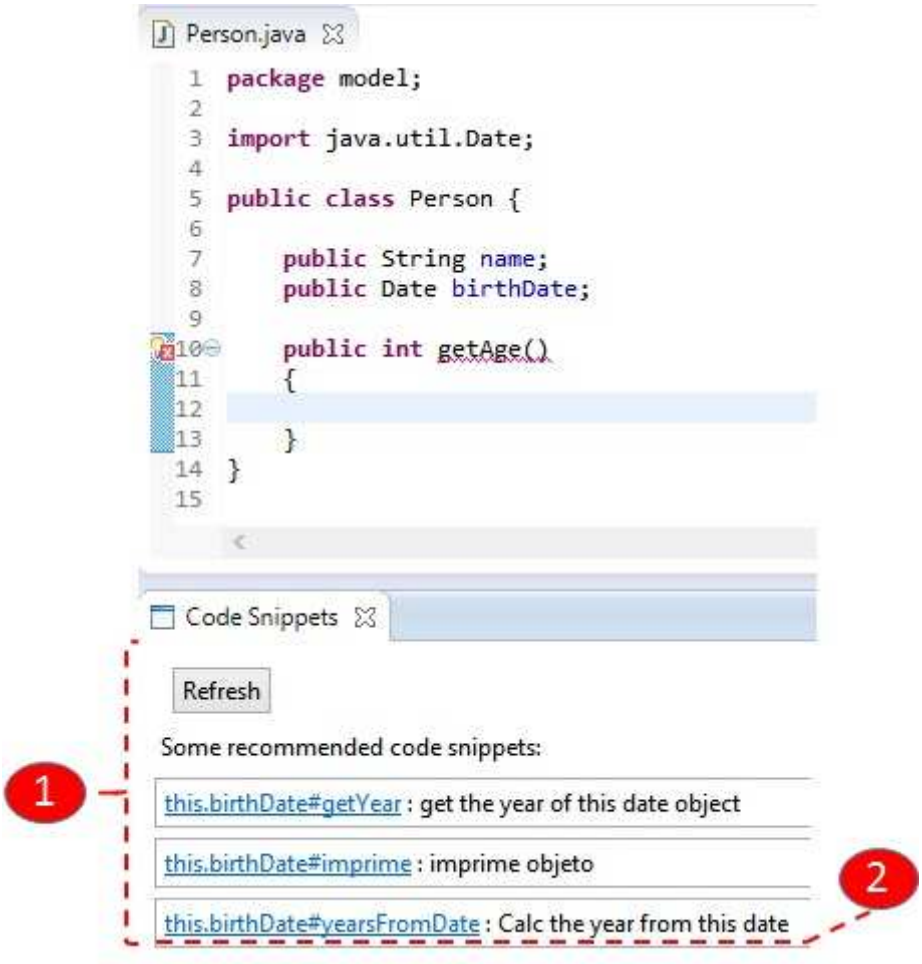


Fonte: Eclipse Foundation (2015)

O *recomendador de templates de código* é uma funcionalidade do *IPS* que recomenda a ativação de *templates de código* ao usuário enquanto este está editando algum código fonte na *IDE*, sem a necessidade de uma ação explícita por parte do usuário. O *recomendador* utiliza informações do contexto do código fonte em edição, como os tipos dos identificadores de objetos disponíveis no escopo, os nomes dos pacotes já importados no código fonte em edição e o nome do método que está sendo codificado, a fim de recomendar uma lista de *templates* possivelmente úteis para o usuário. A lista de *templates* recomendados é constantemente atualizada conforme o conteúdo do código fonte em edição é alterado. Como um exemplo, na Figura 7, é apresentado um cenário que o usuário está editando o conteúdo de um código fonte *Java* (linha 12 do arquivo *Person.java*) e o *recomendador de templates* (item 1 destacado) recomenda a ativação de alguns *templates*. As recomendações são para os identificadores de objetos no escopo do código fonte em edição. Por exemplo, a recomendação destacada como item 2 na Figura 7 (*this.birthDate#yearsFromDate*) está sugerindo a ativação do *template* de nome *yearsFromDate* (*template* associado à classe *java.util.Date*) para o identificador de objeto *this.birthDate*, atributo da classe

Person do tipo *java.util.Date*. Esta recomendação ocorre devido o identificador de objeto *this.birthDate* ser acessível no escopo do código fonte em edição.

Figura 7 – *Recomendador de templates* no *IPS*



Fonte: Eclipse Foundation (2015)

Organização

Esta dissertação está dividida em três capítulos. No Capítulo 1 é apresentada uma análise de trabalhos acadêmicos que foi realizada com o objetivo de compreender o desenvolvimento atual das ferramentas e possibilitar a comparação da ferramenta *IPS*, desenvolvida neste trabalho de mestrado, em relação às outras ferramentas. No Capítulo 2, a ferramenta *IPS* é apresentada através de um cenário de uso hipotético. Neste capítulo, são apresentados também a arquitetura e os detalhes de implementação da ferramenta. No Capítulo 3, é apresentado um teste com programadores profissionais que foi realizado com o objetivo de obter um entendimento melhor em relação à experiência que um programador pode ter com a ferramenta *IPS*.

1 Análise bibliográfica

Uma análise de trabalhos acadêmicos relacionados ao *IPS* foi realizada com os seguintes objetivos: compreender o desenvolvimento atual das ferramentas, posicionar a ferramenta *IPS* em relação às outras ferramentas no que tange as similaridades e os diferenciais funcionais e derivar possíveis trabalhos futuros na área.

O critério utilizado para selecionar as ferramentas para análise foi a similaridade com o objetivo do *IPS* e a estratégia do *IPS* para atingir tal objetivo. O objetivo do *IPS* é auxiliar o programador na utilização de uma *API* e, consequentemente, promover a reutilização de *software*. A estratégia é auxiliar o programador a localizar trechos de código e sugerir trechos de código relevantes para uma determinada tarefa de programação e que apresentem como utilizar a *API*. Aqui localizar e sugerir são diferenciados da seguinte maneira: uma ferramenta localiza um elemento em resposta a uma entrada explícita do usuário, enquanto a ferramenta sugere um elemento baseado em informações no contexto, no caso, no ambiente de programação, sem uma ação explícita do usuário.

Como inicialmente não foram encontradas, nesta análise bibliográfica, ferramentas que sugerissem trechos de código sem a necessidade de uma ação explícita do usuário, esta análise foi expandida para também considerar ferramentas que localizam ou sugerem componentes de uma *API*, como classes e métodos, e não somente ferramentas que localizam trechos de código. Os trabalhos analisados são listados abaixo na Tabela 1, a qual relaciona o nome da ferramenta que o trabalho apresenta.

Neste Capítulo é utilizada a denominação *CSACTs* - ***C****ode* ***S****nippets and* ***A****PI* ***C****omponents* ***T****ools* - para referenciar o conjunto de ferramentas que auxiliam a localização de trechos de código ou de componentes de uma *API*, como classes e métodos, ou ferramentas que sugerem trechos de código ou componentes de uma *API* relevantes para uma determinada tarefa de programação. Na Figura 8 é apresentada a arquitetura das ferramentas *CSACTs*. Essa arquitetura guiará ao longo deste Capítulo a apresentação das características das ferramentas analisadas.

Tabela 1 – Trabalhos analisados e suas respectivas ferramentas

#	Trabalho	Nome da Ferramenta
1	(HENNINGER, 1991)	*CodeFinder*
2	(YE; FISCHER; REEVES, 2000)	*CodeBroker*
3	(MANDELIN et al., 2005)	*Prospector*
4	(MCCAREY; CINNÉIDE; KUSHMERICK, 2005)	*Rascal*
5	(BAJRACHARYA et al., 2006)	*Sourcerer*
6	(BRUCH; SCHÄFER; MEZINI, 2006)	*FrUiT*
7	(HOLMES; WALKER; MURPHY, 2006)	*Strathcona*
8	(SAHAVECHAPHAN; CLAYPOOL, 2006)	*XSnippet*
9	(STYLOS; MYERS et al., 2006)	*Mica*
10	(HOFFMANN; FOGARTY; WELD, 2007)	*Assieme*
11	(LEMOS et al., 2007)	*CodeGenie*
12	(HUMMEL; JANJIC; ATKINSON, 2008)	*Code Conjurer*
13	(BRUCH; MONPERRUS; MEZINI, 2009)	*BMNCCS*
14	(ZHONG et al., 2009)	*MAPO*
15	(BRANDT et al., 2010)	*Blueprint*
16	(KIM et al., 2010)	*eXoaDocs*[1]
17	(MOOTY et al., 2010)	*Calcite*
18	(ROBBES; LANZA, 2010)	*OCompletion*
19	(BHARDWAJ; LUCIANO; KLEMMER, 2011)	*Redprint*
20	(MAR; WU; JIAU, 2011)	*PropER-Doc*
21	(SAWADSKY; MURPHY, 2011)	*Fishtail*
22	(WANG et al., 2011)	*APIExample*
23	(ONEY; BRANDT, 2012)	*Codelets*[2]
24	(WIGHTMAN et al., 2012)	*SnipMatch*
25	(MONTANDON et al., 2013)	*APIMiner*
26	(ZHU et al., 2014)	*UsETeC*
27	(THE SPARS PROJECT & OSAKA UNIVERSITY, 2015)	*SPARS-J*
28	(ARAGON CONSULTING GROUP, 2015)	*Krugle*
29	(BLACK DUCK SOFTWARE, 2015)	*Open Hub Code Search*

Fonte: próprio autor

Os itens destacados na Figura 8 são descritos abaixo:

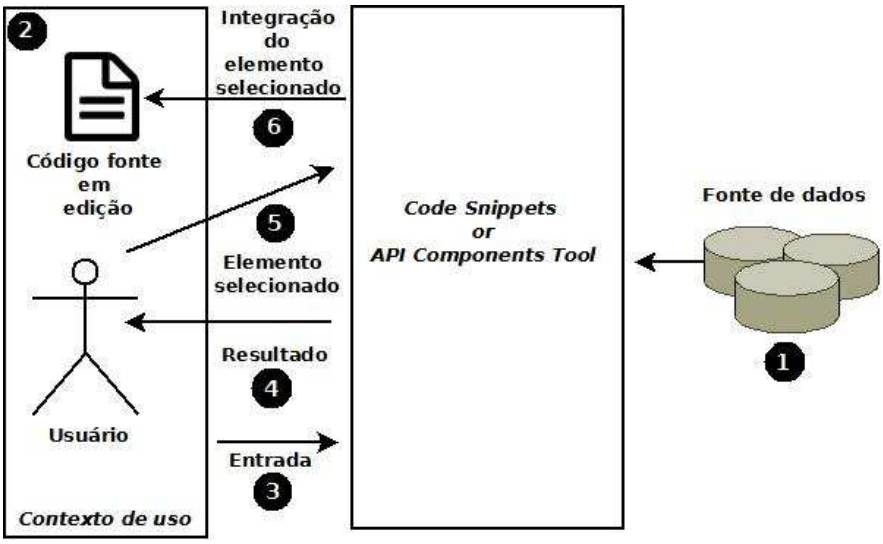
1. As ferramentas *CSACTs* extraem os elementos, trechos de código ou componentes de uma *API*, a serem apresentados ao usuário a partir de uma fonte de dados.

2. O *contexto de uso* é o ambiente no qual a ferramenta está inserida e no qual o usuário interage com ela. Algumas ferramentas *CSACTs* são integradas a uma *IDE*, p. ex. a ferramenta *Blueprint* (BRANDT et al., 2010). Outras, que não são integradas a uma *IDE*, são ferramentas *web* e o usuário necessita utilizar um *browser* a fim de acessá-la, p. ex. a ferramenta *Assieme* (HOFFMANN; FOGARTY; WELD, 2007). O usuário é um programador que está realizando alguma tarefa durante a edição de um código fonte em um projeto de *software* em desenvolvimento e necessita de auxílio a fim de utilizar alguma *API* de interesse.

[1] *eXoaDocs* não é o nome da ferramenta apresentada por Kim et al. (2010), mas a ferramenta apresentada gera páginas *HTML* que são a documentação de uma *API* complementada com trechos de código. Para essa documentação complementada os autores atribuem o nome de *eXoaDocs*. No trabalho a ferramenta apresentada não recebe uma denominação. Aqui a ferramenta é referenciada como *eXoaDocs* por uma questão de simplicidade.

[2] *Codelets* não é o nome da ferramenta apresentada por Oney e Brandt (2012). *Codelets* é utilizado no trabalho para denominar *templates de código* que são associados a um editor gráfico que tem como objetivo auxiliar o usuário a editar os parâmetros do *template*. No trabalho, a ferramenta apresentada não recebe uma denominação. Aqui a ferramenta é referenciada como *Codelets* por uma questão de simplicidade.

Figura 8 – Arquitetura das ferramentas *CSACTs*



Fonte: prório autor

3. O usuário incita a ferramenta através de uma ação explícita que informa a intenção de localizar um trecho de código ou um componente de uma *API* relevante para a tarefa que ele esteja trabalhando. Em algumas ferramentas, não há a necessidade de uma ação explícita por parte do usuário, mas a ferramenta é acionada quando algum evento no *contexto de uso* ocorre, por exemplo, quando o desenvolvedor declara a assinatura de um novo método no código fonte em edição na *IDE* (YE; FISCHER; REEVES, 2000). A entrada pode ser algo digitado pelo usuário, por exemplo, uma *query*[3] contendo palavras chaves (HOFFMANN; FOGARTY; WELD, 2007), ou informações do *contexto de uso*, como informações do código fonte em edição na *IDE* (YE; FISCHER; REEVES, 2000) ou uma combinação dos dois (BRANDT et al., 2010).

4. A ferramenta apresenta o resultado ao usuário. O resultado contém os elementos (trechos de código ou componentes de uma *API*) os quais a ferramenta julga serem relevantes para o usuário baseada nas informações da entrada.

5. O usuário seleciona um elemento entre os elementos apresentados no resultado.

6. O elemento selecionado pelo usuário é então integrado no código fonte em edição. Algumas ferramentas oferecem facilidades para a integração do elemento selecionado, enquanto em outras o usuário necessita copiar o conteúdo do elemento, por exemplo um trecho de código, colá-lo no código fonte em edição e realizar as adaptações necessárias de forma manual.

[3] Termo utilizado na área Recuperação da Informação (RI) para designar a informação produzida pelo usuário como entrada para um sistema de busca (BAEZA-YATES; RIBEIRO-NETO, 2011, p. 26-27).

A arquitetura descrita acima é apenas uma diretriz para apresentar e relacionar as características das ferramentas encontradas nos trabalhos analisados e não representa uma arquitetura exata de todas as ferramentas analisadas. Alguns trabalhos abordam ou implementam apenas um subconjunto dos elementos da arquitetura. Por exemplo, há trabalhos que não apresentam uma solução para a integração do elemento selecionado (HENNINGER, 1991; BAJRACHARYA et al., 2006), outros apresentam ferramentas com uma arquitetura mais peculiar. Por exemplo, a ferramenta apresentada por Kim et al. (2010) aceita como entrada a documentação oficial de uma *API*, no formato *Javadoc*[4,5], e complementa a documentação com trechos de código extraídos a partir de projetos de código aberto. O objetivo da ferramenta é complementar a documentação da *API* com exemplos que facilitem a aprendizagem da utilização da *API*. Apesar de possuir uma arquitetura peculiar (possui como entrada a documentação da *API* e não apenas uma *query* digitada pelo usuário), é possível relacionar os elementos da arquitetura da ferramenta com elementos da arquitetura apresentada acima. A entrada é a documentação oficial da *API* (*Javadoc*), a fonte de dados são os projetos de código aberto, dos quais são extraídos os trechos de código, e o resultado é a documentação da *API* complementada com os trechos de código.

Na Tabela 3 estão sumarizadas as características encontradas nas ferramentas analisadas. Os nomes das colunas e os respectivos valores são acrônimos, os quais são descritos na Tabela 2. Cada coluna da Tabela 3 representa um elemento da arquitetura apresentada na Figura 8, com exceção da coluna AAUT, que representa o aspecto se a ferramenta necessita de uma ação explícita do usuário ou se a ferramenta apresenta elementos automaticamente quando algum evento no *contexto de uso* ocorre (abordado na Subseção 1.3.4). Na Tabela 2, são também relacionadas as Seções deste Capítulo que abordarão cada elemento da arquitetura, assim como as respectivas Subseções. Células não preenchidas na Tabela 3 representam ferramentas cujos os trabalhos não apresentaram informações suficientes para classificar a ferramenta. Os trabalhos analisados que apresentam as respectivas ferramentas são listados na Tabela 1.

A seguir, de acordo com as características encontradas nas ferramentas apresentadas nos trabalhos analisados, os elementos da arquitetura são detalhados através de cinco Seções: *Contexto de uso* (Seção 1.1), Fonte de dados (Seção 1.2), Entrada (Seção 1.3), Resultado (Seção 1.4) e Integração do elemento selecionado (Seção 1.5).

[4] Padrão de documentação de *APIs* utilizado pela plataforma *Java*. Uma ferramenta da plataforma denominada *javadoc* é responsável por compilar os comentários presentes em um conjunto de códigos fontes *Java* de um projeto e produzir um conjunto de páginas *HTML* em um formato padrão que descreve as classes, os métodos, os construtores e os campos de classes.

[5] <http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html>

Tabela 2 – Acrônimos da Tabela 3

Colunas			Valores		
Acrônimo	**Descrição**	**Seção**	**Acrônimo**	**Descrição do valor**	**Subseção**
CONT	**Cont**exto de Uso	1.1	IDE	Integrada à uma ***IDE***	1.1
			WEB	Ferramenta ***web***	1.1
FDADOS	**F**onte de **Dados**	1.2	JDOC	***Javadoc***[6]	—
			PPS	**P**róprio **p**rojeto de *software*	1.2.1
			OUTPS	**Out**ros **p**rojetos de *software*	1.2.2
			PW	**P**áginas ***web***	1.2.3
			TU	**T**este **u**nitário	1.2.4
			TCCM	**T**rechos de **c**ódigo **c**adastrados **m**anualmente	1.2.5
ENT	**Ent**rada	1.3	PC	**P**alavras-**c**haves	1.3.1
			IED	**I**nformações **e**specíficas do **d**omínio	1.3.2
			ICP	**I**nformações no **c**ontexto de **p**rogramação	1.3.3
			JDOC	***Javadoc***	1.3.5
			PC + ICP	Combinação de **p**alavras-**c**haves e **i**nformações no **c**ontexto de **p**rogramação	1.3.1 1.3.3
AAUT	**A**tivação **Aut**omática	1.3.4	S	**S**im, apresenta elementos de forma automática	—
			N	**N**ão, a ferramenta exige uma ação explícita por parte do usuário	—
RES	**Res**ultado	1.4	C	**C**omponentes de uma *API*	1.4.1
			TC	**T**rechos de **c**ódigo	1.4.1
			JDOCTC	***Javadoc*** complementado com **t**rechos de **c**ódigo	1.3.5
INT	**Int**egração do elemento selecionado	1.5	NA	**N**ão **a**presenta funcionalidades para adaptar o elemento apresentado no contexto de programação	—
			A	**A**presenta funcionalidades de adaptação	—

Fonte: próprio autor

1.1 Contexto de uso

O *contexto de uso* é o ambiente no qual a ferramenta está inserida e no qual o usuário, normalmente um programador que está desenvolvendo um software, interage. Entre as ferramentas *CSACTs* apresentadas nos trabalhos analisados, é possível identificar dois grupos principais. Um grupo de ferramentas *web* e outro grupo de ferramentas integradas a uma *IDE*.

[6] A ferramenta *CodeBroker* (YE; FISCHER; REEVES, 2000, p. 64) recomenda ao usuário métodos *Java* os quais são extraídos a partir da própria documentação da *API* no formato *Javadoc*. Como foi a única ferramenta da análise que utilizou esse tipo de fonte de dados, não foi criada uma Seção neste Capítulo para abordar exclusivamente esse tipo de fonte de dados.

Tabela 3 – Ferramentas e suas respectivas características

#	Nome da Ferramenta	CONT	FDADOS	ENT	AAUT	RES	INT
1	*CodeFinder*	—	—	PC	N	C	NA
2	*CodeBroker*	IDE	JDOC[6]	ICP	S	C	NA
3	*Prospector*	IDE	OUTPS	ICP	N	TC	NA
4	*Rascal*	IDE	OUTPS	ICP	S	C	NA
5	*Sourcerer*	WEB	OUTPS	IED	N	TC	NA
6	*FrUiT*	IDE	OUTPS	ICP	N	C	NA
7	*Strathcona*	IDE	OUTPS	ICP	N	TC	NA
8	*XSnippet*	IDE	OUTPS	ICP	N	TC	NA
9	*Mica*	WEB	PW	PC	N	TC	NA
10	*Assieme*	WEB	PW	PC	N	TC	NA
11	*CodeGenie*	IDE	OUTPS	ICP	N	C	A
12	*Code Conjurer*	IDE	OUTPS	ICP	S	C	A
13	*BMNCCS*	IDE	OUTPS	ICP	N	C	NA
14	*MAPO*	IDE	OUTPS	ICP	N	TC	NA
15	*Blueprint*	IDE	PW	PC + ICP	N	TC	NA
16	*eXoaDocs*	WEB	OUTPS	JDOC	N	JDOCTC	NA
17	*Calcite*	IDE	PW	ICP	N	C	A
18	*OCompletion*	IDE	PPS	ICP	S	C	NA
19	*Redprint*	IDE	PW	ICP	S	TC	NA
20	*PropER-Doc*	—	OUTPS	IED	N	TC	NA
21	*Fishtail*	IDE	PW	ICP	S	TC	NA
22	*APIExample*	WEB	PW	IED	N	TC	NA
23	*Codelets*	IDE	TCCM	PC	N	TC	A
24	*SnipMatch*	IDE	TCCM	PC + ICP	N	TC	A
25	*APIMiner*	WEB	OUTPS	JDOC	N	JDOCTC	NA
26	*UsETeC*	WEB	TU	JDOC	N	JDOCTC	NA
27	*SPARS-J*	WEB	OUTPS	IED	N	TC	NA
28	*Krugle*	WEB	OUTPS	IED	N	TC	NA
29	*Open Hub Code Search*	WEB	OUTPS	IED	N	TC	NA
30	***IPS*** [7]	**IDE**	**TCCM**	**PC + ICP**	**S**	**TC**	**A**

Fonte: próprio autor

1.1.1 Ferramentas *web* vs. ferramentas integradas a uma *IDE*

Ferramentas *web* possuem a vantagem de serem independentes de uma *IDE* (BRANDT et al., 2010, p. 513). Por exemplo, um programador que esteja desenvolvendo um *software* na linguagem *Java* através da *IDE Eclipse* (ECLIPSE FOUNDATION, 2015) e outro programador utilizando a *IDE Netbeans* (ORACLE CORPORATION, 2014) ambos podem utilizar a mesma ferramenta *CSACT* a fim de localizar trechos de código ou componentes de uma *API*. Exemplos de ferramentas *web* são: *APIExample* (WANG et al., 2011), *Assieme* (HOFFMANN; FOGARTY; WELD, 2007), *Krugle* (ARAGON CONSULTING GROUP, 2015) e *Open Hub Code Search* (BLACK DUCK SOFTWARE,

[7] Ferramenta proposta neste trabalho de mestrado

2015). Já as ferramentas integradas a uma *IDE* podem explorar o contexto de programação da *IDE* no qual o usuário está desenvolvendo um software para extrair informações que servem como entrada ou que contribuem para a entrada, a fim de localizar ou sugerir um trecho de código ou um componente de uma *API*, p. ex. as ferramentas *CodeBroker* (YE; FISCHER; REEVES, 2000), *XSnippet* (SAHAVECHAPHAN; CLAYPOOL, 2006) e *Rascal* (MCCAREY; CINNÉIDE; KUSHMERICK, 2005). Mais detalhes de como as ferramentas integradas a uma *IDE* utilizam informações no contexto de programação serão apresentados na Subseção 1.3.3.

Uma outra vantagem das ferramentas *web* é que para o usuário acessar uma ferramenta *web* este utiliza um navegador *web* e estes fornecem um ambiente rico de interação proporcionado pela possibilidade de navegação entre as páginas. O usuário pode, através de *hiperlinks*, acessar páginas relacionadas com uma página sendo visualizada e também pode voltar para as páginas acessadas anteriormente. Além disso, navegadores *web* são customizados e configurados de acordo com o perfil do usuário, pois possuem recursos como os favoritos, os *cookies*, os certificados de segurança e o histórico de navegação (GOLDMAN; MILLER, 2008, p. 65). Logo, há um dilema, ao escolher integrar uma ferramenta à *IDE*, o projetista da ferramenta *CSACT* está abrindo mão, além da independência da *IDE*, de um ambiente de interação rico proporcionado pelos navegadores *web*, a favor de um ambiente (*IDE*) com uma interface de interação mais acoplada e próxima da tarefa que o programador esteja trabalhando no momento (BRANDT et al., 2010, p. 514). Essa proximidade com a tarefa que o usuário está trabalhando no momento evita o problema da troca de contexto entre o ambiente de programação e o navegador *web*. Essa troca de contexto pode provocar uma distração para o usuário e diminuir sua produtividade (YE; FISCHER; REEVES, 2000, p. 62). Em uma ferramenta integrada à *IDE*, o usuário não necessita abrir o navegador *web* e suas ações são realizadas dentro da própria *IDE* (SAWADSKY; MURPHY, 2011, p. 49).

Com o objetivo de utilizar as informações presentes no contexto de programação na *IDE* e ao mesmo tempo aproveitar os recursos de interação fornecidos por um navegador *web*, a ferramenta *Codetrail* (GOLDMAN; MILLER, 2008) implementa um canal de comunicação entre a *IDE Eclipse* e o navegador *Firefox*[8]. Entre outras funcionalidades disponibilizadas pelo *Codetrail*, a ferramenta disponibiliza uma aba no navegador *Firefox*. Esta aba é automaticamente atualizada para que o usuário navegue pela documentação dos elementos (pacotes, classes, métodos, etc) no contexto de programação na *IDE Eclipse* com os quais o programador esteja trabalhando no momento. Se o programador está editando um código fonte na *IDE Eclipse* e o cursor aponta para um elemento, por exemplo para um método, a documentação do elemento é automaticamente exibida no navegador *Firefox*. Embora seja importante citar a ferramenta, devido à sua particularidade em integrar

[8] <http://www.mozilla.org/firefox/products/>

informações presentes em ambientes diferentes (navegador *web* e *IDE*), *Codetrail* não se enquadra em uma ferramenta *CSACT*, pois seu foco é apresentar itens de documentação dos componentes das *APIs* e não os componentes propriamente dito ou trechos de código.

1.1.2 Dependência arquitetural com uma *IDE*

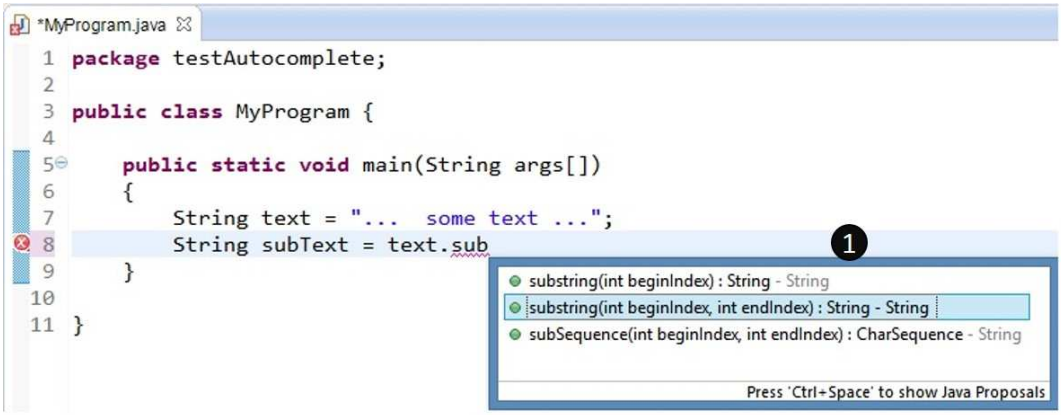
Sobre a dependência das ferramentas integradas a uma *IDE*, seria interessante a possibilidade que as ferramentas integradas a uma *IDE* fossem facilmente migradas para outras *IDEs*. Do ponto de vista acadêmico, isso seria útil para realizar estudos exploratórios que atingissem uma gama maior de usuários e ajudaria a compreender as ferramentas para estilos de programação diferentes e em arquiteturas de *IDEs* diferentes. Uma estratégia a fim de aumentar a portabilidade das ferramentas seria a criação de uma arquitetura modularizada. Módulos que não necessitassem serem acoplados à *IDE* poderiam ter o código fonte reutilizado entre *IDEs* diferentes. Somente módulos que possuíssem recursos dependentes da *IDE* seriam acoplados à *IDE* e, portanto, teriam uma versão para cada *IDE*, por exemplo um módulo que exibisse resultados em componentes gráficos específicos de uma *IDE*. Módulos não acoplados à *IDE* poderiam ser implementados em uma camada de software que seria executada em um processo do sistema operacional separado do processo da *IDE* a fim de desacoplá-los da arquitetura da *IDE*. Os módulos acoplados à *IDE*, então, se comportariam como um adaptador e se comunicariam com esta camada de software, rodando em um processo separado do sistema operacional, através de algum protocolo de comunicação padrão, como por exemplo o *TCP/IP*. Tal arquitetura é discutida em (SAWADSKY; MURPHY, 2011, p. 50), no entanto a construção de uma ferramenta implementando a arquitetura foi deixada para trabalhos futuros.

1.1.3 Autocompletadores de código

Um subconjunto especial de ferramentas *CSACTs* integradas a uma *IDE* são os *autocompletadores de código*. *Autocompletadores de código* sugerem componentes de uma *API*, como métodos, atributos de classe e construtores, baseado no que o usuário já tenha digitado no conteúdo da linha atualmente em edição em um código fonte. Nas Figuras 9 e 10, é apresentado um cenário de uso do *autocompletador de código* nativo da *IDE Eclipse* (ECLIPSE FOUNDATION, 2015). O usuário está editando um código fonte *Java* e, na linha *7*, é declarada uma variável local do tipo *java.lang.String* com o identificador *text*. Na linha *8*, com a intenção de extrair uma parte do valor da variável *text*, o usuário digita *"text.sub"*, pois acredita que há algum método da classe *java.lang.String* cujo nome tenha o prefixo *sub* e que retorne uma *substring* do valor de *text*. A fim de ser auxiliado, o usuário aciona o *autocompletador de código* nativo do *Eclipse* através do atalho de teclado *CONTROL + ESPAÇO*. O *autocompletador* sugere três métodos (item *1*). Os três métodos sugeridos são todos os métodos da classe *java.lang.String* cujo nome tenha o prefixo *sub*.

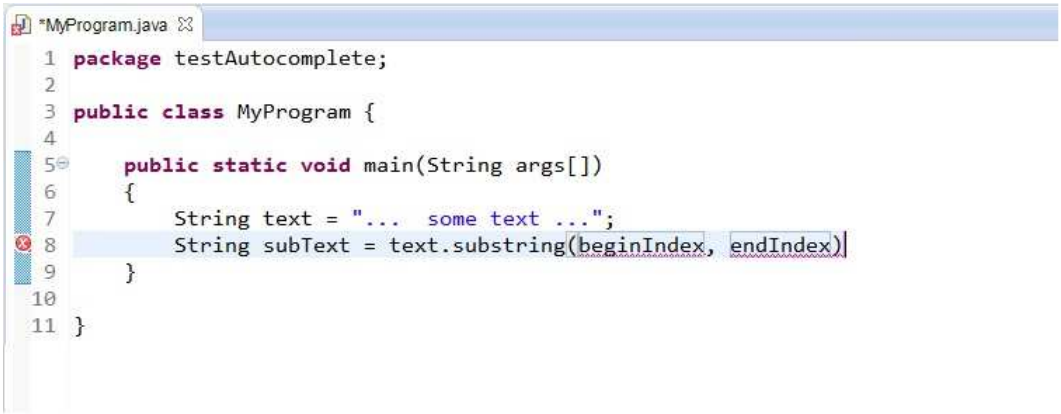
O usuário, então, seleciona o segundo método, método com a assinatura *substring(int,int)*, e o *autocompletador* completa a assinatura do método desejado na linha sendo editada (linha *8* da Figura 10).

Figura 9 – *Autocompletador* nativo do *Eclipse*



Fonte: Eclipse Foundation (2015)

Figura 10 – *Autocompletador* do *Eclipse* após seleção de um elemento

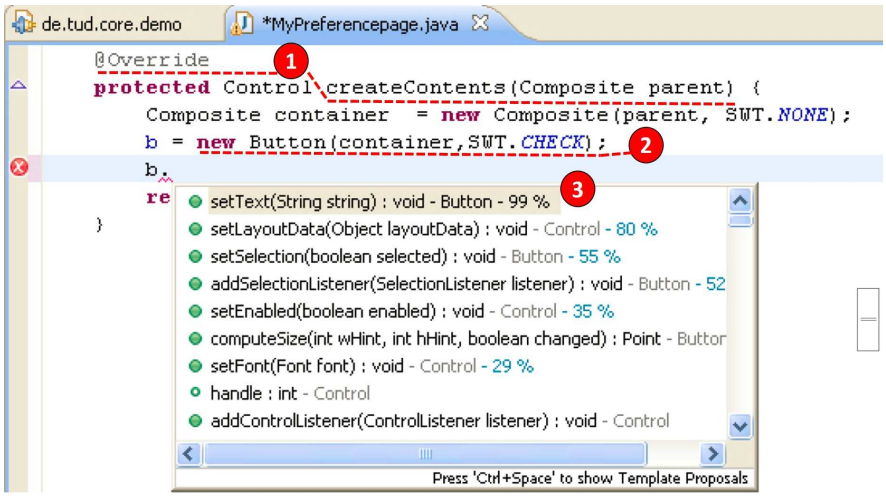


Fonte: Eclipse Foundation (2015)

Uma limitação dos *autocompletadores de código* é que eles se baseiam exclusivamente no sistema de tipos estático da linguagem de programação e as sugestões são ordenadas em ordem alfabética, ao invés de serem ordenadas baseando-se em informações no contexto de programação. Bruch, Monperrus e Mezini (2009) apresenta e compara três algoritmos para suprir tais deficiências. Um dos algoritmos apresentados, denominado *BMNCCS - **B**est **M**atching **N**eighbors **C**ode **C**ompletion **S**ystem* e que obteve a melhor performance, ordena os métodos apresentados ao usuário baseando-se nos métodos e nos construtores chamados anteriormente para o objeto que o usuário está solicitando auxílio e baseado em qual método é sobrescrito pelo método que está sendo editado no código fonte em edição. Na

Figura 11 é apresentado um cenário de uso do *autocompletador* utilizando esse algoritmo. O *autocompletador* ordena os métodos de acordo com as probabilidades do usuário utilizar o método. O método *setText(String)* é o primeiro da lista e apresenta uma probabilidade de 99% de ser chamado para o objeto *b* (item *3*). Essa probabilidade é calculada baseando-se no fato que o método sendo editado sobrescreve o método *createContents* (item *1*) e para o objeto *b*, que é do tipo *Button*, foi chamado o construtor *Button(Composite,int)* (item *2*). Baseando-se em outros métodos que também sobrescrevem o método *createContents* e que possuem um objeto do tipo *Button* para o qual também foi chamado o construtor *Button(Composite,int)* é calculada a probabilidade de ser chamado o método *setText(String)* para o objeto *b*. Esses outros métodos que são utilizados para calcular as probabilidades são métodos presentes em um repositório da ferramenta extraídos de outros projetos de *software*.

Figura 11 – *Autocompletador* utilizando *BMNCCS*



Fonte: adaptado de (BRUCH; MONPERRUS; MEZINI, 2009, p. 219)

Robbes e Lanza (2010) apresentaram um conjunto de algoritmos utilizados pelos *autocompletadores de código* a fim de ordenar os elementos exibidos ao usuário. Os algoritmos foram comparados e um dos algoritmos que obtiveram uma boa performance na comparação realizada foi escolhido para implementar um *autocompletador* denominado *OCompletion*, que foi integrado na *IDE Squeak*[9]. O algoritmo escolhido prioriza os métodos apresentados ao usuário baseado na data da última modificação do método. Métodos recentemente modificados ou métodos chamados por métodos recentemente modificados são melhores ranqueados.

O *autocompletador de código* é recurso muito utilizado pelos desenvolvedores de software (MURPHY; KERSTEN; FINDLATER, 2006, p. 80), no entanto não é autossuficiente. Muitas vezes um desenvolvedor necessita entender detalhes não esclarecidos

[9] *IDE* para o desenvolvimento de projetos utilizando a linguagem *Smalltalk* <http://squeak.org/>.

por um *autocompletador de código*. Um *autocompletador* sugere componentes de uma *API*, no entanto não esclarece como os componentes de uma *API* podem ser utilizados juntos a fim de realizar uma operação. Por exemplo, um método pode retornar um valor dependendo do resultado do processo executado pelo método, um método pode alterar os valores dos argumentos passados pelos métodos e tem métodos que dependem da execução de outros métodos antes de serem chamados (BHARDWAJ; LUCIANO; KLEMMER, 2011). Ferramentas que apresentam trechos de código também são necessárias durante o desenvolvimento de software para que o usuário possa entender como utilizar uma *API*.

1.2 Fonte de dados

A fonte de dados é de onde a ferramenta *CSACT* extrai os elementos, trechos de código ou componentes de uma *API*, os quais são apresentados ao usuário.

1.2.1 Próprio projeto de *software*

Algumas ferramentas que apresentam componentes de uma *API* para o usuário utilizam como fonte de dados o próprio projeto de *software* sendo desenvolvido. De acordo com as ferramentas *CSACTs* apresentadas nos trabalhos analisados, tal estratégia é exclusiva dos *autocompletadores de código*, os quais foram apresentados na Subseção anterior (Subseção 1.1.3). *Autocompletadores de código* de linguagens estaticamente tipadas confiam no sistema de tipos estático da linguagem para apresentar componentes relevantes ao usuário de acordo com o contexto presente no código fonte em edição. Por exemplo, no cenário de uso apresentado na Figura 9, o usuário está editando um código fonte *Java* na *IDE Eclipse* (ECLIPSE FOUNDATION, 2015) e solicita auxílio para chamar um método para o identificador de objeto *text*, que é uma variável local do tipo *java.lang.String*. O *autocompletador* sugere os métodos *substring(int)*, *substring(int, int)* e *subSequence(int,int)* (item *1*) verificando quais métodos da classe *java.lang.String* possuem o nome com o prefixo *sub*. A classe *java.lang.String* é uma classe da *API* padrão da plataforma *Java*, portanto é uma classe que automaticamente faz parte do contexto do projeto de *software* sendo desenvolvido pelo usuário.

Embora confiar em informações do próprio projeto de software em desenvolvimento seja uma estratégia exclusiva dos *autocompletadores de código*, há *autocompletadores de código* que confiam em outras fontes de dados. Por exemplo, o *autocompletador de código* apresentado por Bruch, Monperrus e Mezini (2009), descrito na Subseção anterior (Subseção 1.1.3), utiliza informações presentes em um repositório da ferramenta construído a partir de trechos de código de outros projetos de *software* (fonte de dados abordado na próxima Subseção, 1.2.3) e o *autocompletador Calcite* (MOOTY et al., 2010) utiliza informações de páginas *web*.

Calcite é um *autocompletador de código* que apresenta formas de instanciar uma determinada classe. As maneiras de instanciar uma classe são extraídas de trechos de código de páginas *web*. A ferramenta utiliza o buscador do *Yahoo* a fim de procurar por páginas *web* contendo trechos de código os quais apresentam maneiras de instanciar classes. Por exemplo, um trecho de código em uma página *web* contendo uma linha que começa com *"SSLSocketFactory factory ="* indica que a linha contém uma maneira de instanciar a classe *SSLSocketFactory*. As formas de instanciar classes são armazenadas em um repositório que é posteriormente utilizado para responder as requisições dos usuários. Mais sobre ferramentas que extraem informações das páginas *web* é discutido na Subseção 1.2.3.

1.2.2 Outros projetos de *software*

Uma outra fonte de dados utilizada por algumas *CSACTs* são outros projetos de software e não o próprio projeto de *software* sendo desenvolvido pelo usuário da ferramenta conforme abordado na Subseção anterior (BRUCH; MONPERRUS; MEZINI, 2009; BRUCH; SCHÄFER; MEZINI, 2006; SAHAVECHAPHAN; CLAYPOOL, 2006; MCCAREY; CINNÉIDE; KUSHMERICK, 2005). Esses outros projetos de *software* são projetos de *software* finalizados e que contém trechos de código que demonstram a utilização das *APIs* de interesse. Essa estratégia baseia-se na concepção que os códigos fontes presentes em projetos de aplicações já existente apresentam a utilização correta das *APIs* de interesse, visto que o resultado do projeto é um *software* completo e funcional.

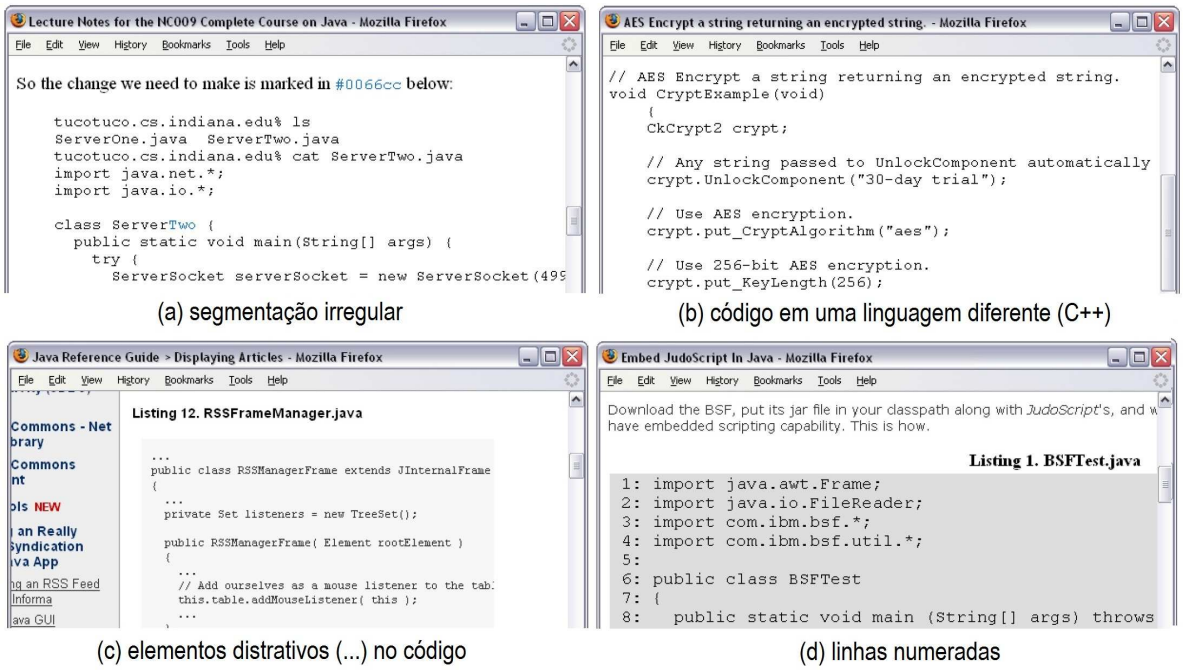
Um conjunto de projetos de *software* disponíveis na *Internet* populares são os projetos de código aberto (DESHPANDE; RIEHLE, 2008), os quais são utilizados por algumas ferramentas *CSACTs* apresentadas em trabalhos anteriores a fim de apresentar trechos de códigos (BAJRACHARYA et al., 2007; BLACK DUCK SOFTWARE, 2015; ZHONG et al., 2009) ou componentes de uma *API* (YE; FISCHER; REEVES, 2000; LEMOS et al., 2007) ao usuário. Um grande desafio ao coletar trechos de código de projetos de código aberto pela *Internet* é que não há um padrão na distribuição dos códigos fontes presentes nesses projetos. Cada repositório de projetos de código aberto utiliza um controlador de versão diferente e um protocolo de *download* diferente. Bajracharya, Ossher e Lopes (2014) amenizam esse problema através de uma arquitetura baseada em módulos, que é utilizada para implementar um sistema de busca de trechos de código denominado *Sourcerer*. Para cada projeto de software é criado um módulo, denominado *Code Crawler*, que trata as particularidades para extrair as informações do projeto de software. Os módulos devem seguir um padrão de implementação para que possam ser facilmente integrados ao sistema quando há a necessidade de extrair trechos de código de um novo projeto de código aberto.

1.2.3 Páginas *web*

Ferramentas que consideram apenas os códigos fontes de projetos de código aberto ignoram trechos de código presentes em páginas *web*, como páginas de documentação, fóruns, blogs e páginas de tutoriais das *APIs*. Tais páginas, além de apresentarem trechos de código como exemplos para ensinar a utilização da *API*, contém textos explicativos que contribuem para o entendimento desses trechos de código. Os termos presentes em uma página *web* contendo um trecho de código colabora com a eficiência da busca realizada pela ferramenta *CSACT* ao permitir que o usuário busque pelo trecho de código utilizando um termo (uma palavra chave) presente na página, não limitando a busca somente pelos termos presentes no trecho de código (HOFFMANN; FOGARTY; WELD, 2007; BRANDT et al., 2010). Além disso, os trechos de código contido em páginas *web* geralmente são cuidadosamente confeccionados por um especialista da *API* com a intenção de demonstrar o uso da *API*, logo há uma tendência de serem mais concisos (HOFFMANN; FOGARTY; WELD, 2007, p. 15). Aproveitando o potencial de tais trechos de código presentes nas páginas *web*, trabalhos anteriores apresentaram ferramentas *CSACTs* que extraem trechos de código de páginas *web* para serem apresentados como resultados das buscas realizadas pelos usuários (STYLOS; MYERS et al., 2006; HOFFMANN; FOGARTY; WELD, 2007; BRANDT et al., 2010; WANG et al., 2011). Tais ferramentas utilizam um buscador *web*, como o *Google*, a fim de buscar por páginas *web* que contenham trechos de código. Os principais desafios enfrentados por essas ferramentas são como identificar quais páginas *web* possuem trechos de código e como isolar e extrair segmentos da página que correspondam aos trechos de código. Na Figura 12 são apresentadas algumas dificuldades encontradas ao extrair trechos de código a partir de páginas *web*: um segmento de texto de uma página *web* pode conter elementos sem uma separação clara entre qual parte representa o trecho de código de interesse e qual parte é apenas um texto (item *a*); uma página *web* pode conter um trecho de código em uma linguagem de programação semelhante a linguagem de interesse, porém diferente e irrelevante para o usuário, por exemplo, uma ferramenta com o objetivo de apresentar trechos de código *Java* poderia encontrar trechos de código *C++* (item *b*); trechos de código podem apresentar elementos distrativos, como *"..."*, e que não fazem parte da linguagem de programação (item *c*); outro elemento distrativo que pode aparecer em trechos de código em páginas *web* são as numerações das linhas (item *d*).

As principais técnicas para identificar trechos de código em páginas *web* envolvem a aplicação de métodos de mineração de dados, mais especificamente métodos de classificação de dados (HAN; KAMBER, 2006, p. 285-382), os quais consistem em extrair características de um segmento de texto, como a presença ou não de determinadas palavras reservadas em uma linguagem de programação (*if*, *else*, *for*, *while*, etc), presença ou não de linhas terminadas com ponto e vírgula e a presença ou não de chaves e colchetes. Tais características são então utilizadas como entrada para um algoritmo de classificação que retorna se o segmento

Figura 12 – Dificuldades de extração em páginas *web*



Fonte: adaptado de (HOFFMANN; FOGARTY; WELD, 2007, p. 18)

de texto é um trecho de código ou não. Esses métodos são baseados em heurísticas e, portanto, não são livres de falhas. Por exemplo, se o objetivo da ferramenta é identificar trechos de código *Java* em páginas *web*, um algoritmo de classificação poderia classificar de forma errônea um trecho de código escrito em C++ por possuírem características semelhantes, ou ainda, um segmento de texto poderia ser classificado como trecho de código por fazer muitas referências para elementos de trechos de código (BRANDT et al., 2010, p. 517). Uma técnica utilizada principalmente para diminuir os falsos positivos, segmentos de texto classificados como trechos de código quando na realidade não são, é a utilização de um *parser* adaptativo. Um *parser* adaptativo é um *parser* com o objetivo de compilar trechos de código, no entanto não é muito rigoroso em relação à sintaxe da linguagem de programação. Ele é capaz de realizar pequenas correções no trecho de código a fim de obter uma compilação bem sucedida. Tais modificações podem incluir, por exemplo, a remoção de caracteres ilegais, a remoção de uma linha identificada como uma linha não contendo sentenças válidas, a remoção de números presentes no início das linhas e a inserção de trechos de código soltos (fora de um método e fora de uma classe) para dentro de um método e para dentro de uma classe para que possam ser compilados (WANG et al., 2011, p. 594).

1.2.4 Testes unitários

Métodos baseados na mineração de repositórios de projetos de código aberto ou de páginas *web* tendem a produzir muitos códigos irrelevantes (falsos positivos), principalmente pelo fato de tentarem extrair elementos relevantes a partir da análise de uma grande quantidade de informações irrelevantes. Há um dilema na escolha do tamanho da fonte de dados escolhida para extrair as informações. Se a fonte de dados é muito pequena, elementos relevantes podem deixar de serem encontrados. Se a fonte de dados for muito extensa e diversificada, como páginas *web*, a ferramenta poderá produzir muitos resultados irrelevantes. Ghafari et al. (2014) sugere que trechos de código possam ser extraídos dos teste unitários do próprio projeto de desenvolvimento da *API* de interesse a fim de contornar essa deficiência. Um teste unitário é um trecho de código com o objetivo de testar um componente de uma *API*, como classes e métodos (OSHEROVE, 2013). O conjunto de testes unitários de um projeto de software é um conjunto conciso e relevante a respeito da *API* sendo testada. Eles apresentam não somente os cenários de uso de uma classe, mas também as melhores práticas para utilizar a classe, pois são normalmente confeccionados pela mesma organização que projetou e implementou a classe. Além disso, nem sempre há códigos fontes disponíveis na *Web* de projetos que utilizem a *API* de interesse. Por exemplo, há casos que a *API* de interesse foi recentemente publicada ou os projetos de software que utilizam a *API* podem ser projetos privados de empresas que não possuem o código aberto por interesses comerciais.

O potencial da utilização dos teste unitários como fonte de trechos de código como exemplos de utilização de uma *API* foi explorado por Nasehi e Maurer (2010). No trabalho foi realizado um experimento envolvendo tarefas de programação que utilizavam componentes de uma determinada *API* e foram disponibilizados teste unitários do próprio projeto de desenvolvimento da *API* para os participantes. Foi observado que a utilização dos testes unitários como exemplos pode ser útil para enfrentar dificuldades na utilização da *API*, no entanto o desenvolvedor pode ter dificuldades para encontrar o cenário de uso correto de uma *API* dentro dos testes unitários e, portanto, é necessário ferramentas para apoiar esse processo.

Embora haja muitos pontos a favor da utilização dos teste unitários, muitos são os desafios para extrair os trechos de código a partir dos testes unitários (GHAFARI et al., 2014; GHAFARI, 2014). Teste unitários são conceitualmente divididos em três fases: configuração, execução e verificação. A configuração contém sentenças para criar um objeto da classe a ser testada e colocar esse objeto em um estado inicial. A execução chama métodos para o objeto criado na configuração a fim de alterar o estado do objeto. E, por fim, a verificação contém sentenças para verificar se o objeto se encontra no estado esperado. Na Figura 13 é apresentado um exemplo de teste unitário. As linhas marcadas com *S1* e *S2* são linhas de configuração. Linhas marcadas com *E1* e *E2* são linhas de

execução e linhas marcadas com *O1* e *O2* são linhas de verificação. O teste unitário testa a classe *ArrayList*, no entanto apresenta, no mesmo teste unitário, dois cenários de uso de *ArrayList*, um cenário para adicionar um elemento (linhas marcada com *S1*, *E1* e *O1*) e outro para remover um elemento (linhas marcadas com *S2*, *E2* e *O2*). Para que esse trecho de código possa ser utilizado como um exemplo de utilização da classe *ArrayList*, seria necessário separar estes dois cenários. Assim, teria se dois trechos, um que mostrasse como adicionar um elemento e outro que mostrasse como remover um elemento. Além disso, certas sentenças são relevantes apenas nos testes unitários, por exemplo as sentenças de verificação. Seria interessantes que tais sentenças não fossem exibidas em um trecho de código apresentado como exemplo de utilização da classe *ArrayList*. Porém, separar os cenários e remover sentenças irrelevantes, não é uma tarefa trivial. Linhas podem pertencer ao mesmo cenário, como as linhas *1* e *2*. A divisão configuração, execução e verificação é apenas conceitual, não há uma regra que garanta que uma determinada linha realmente pertença à fase de configuração, à fase de execução ou à fase de verificação.

Figura 13 – Exemplo de um teste unitário

```
1  S1 S2 List array = new ArrayList();
2  S1 S2 String item = "bob";
3  E1 array.add(item);
4  O1 int size = array.size();
5  O1 assertEquals(1, size);
6  O1 assertTrue(array.contains(item));
7  E2 array.remove(item);
8  O2 assertFalse(array.contains(item));
```

Fonte: Ghafari (2014)

A fim de estabelecer padrões de como os testes unitários são codificados, um conjunto de testes unitários foram analisados por Zhu et al. (2014) . Dessa análise foram definidas algumas heurísticas a fim de separar as fases dos testes unitários (configuração, execução e verificação) e separar os cenários de uso dos componentes testados. As heurísticas foram então utilizadas para implementar uma ferramenta denominada *UsETeC - Usage Examples from Test Code*, que é uma ferramenta baseada em documentação, que aceita como entrada a documentação da *API* no formato *Javadoc* e produz como resultado a documentação complementada com trechos de código que demonstram a utilização da *API* (veja sobre ferramentas baseadas em documentação na Subseção 1.3.5).

1.2.5 Trechos de código cadastrados manualmente

Há ferramentas que não extraem trechos de código automaticamente de uma fonte de dados, mas confia em um repositório de trechos de código cadastrados manualmente

(ONEY; BRANDT, 2012; WIGHTMAN et al., 2012). Um tipo especial de trechos de código cadastrados manualmente são os *templates de código*. *Templates de código* são trechos de código contendo diretivas de integração (WIGHTMAN et al., 2012), as quais são utilizadas por uma ferramenta, normalmente um *plug-in* ou uma funcionalidade nativa da *IDE*, para realizar a integração do trecho de código no código fonte em edição durante a ativação do *template* pelo usuário. Um exemplo de *IDE* com suporte aos *templates de código* é o *Eclipse* (ECLIPSE FOUNDATION, 2015). O *Eclipse* disponibiliza um conjunto de interfaces gráficas a fim de facilitar a manutenção desses *templates* (criação, edição, remoção e migração). Uma vez que um *template de código* esteja cadastrado na IDE do desenvolvedor, é possível que o desenvolvedor ative o *template* durante a edição de um código fonte Java. Mais sobre *templates de código* será discutido na Seção 1.5.

Trechos de código cadastrados manualmente possuem a mesma vantagem dos trechos de código extraídos em páginas *web* mencionados anteriormente, tais trechos de código (cadastrados manualmente) são confeccionados a fim de ensinar e facilitar a utilização de uma *API* e, portanto, há uma tendência de possuir uma qualidade melhor e serem mais concisos. No entanto, diferente dos trechos de código extraídos de páginas *web*, trechos de código cadastrados manualmente não possuem os riscos associados a extração automática de uma fonte de dados. Em trechos de código extraídos de páginas *web*, o algoritmo de extração baseia-se em heurísticas (WANG et al., 2011, p. 594). Não há uma garantia que um trecho de código prático e útil será extraído da página *web*. O mesmo risco existe ao extrair trechos de código a partir dos testes unitários (GHAFARI et al., 2014). Por outro lado, a desvantagem é que, para os trechos de códigos cadastrados manualmente, existe um custo envolvido tanto na criação como na manutenção dos trechos de código (WIGHTMAN et al., 2012, p. 219-220). Logo, o conjunto de trechos de código cadastrados manualmente possuem uma tendência de serem incompletos. Ou seja, para uma determinada *API*, há o risco de não haver trechos de código confeccionados para a *API* ou o conjunto de trechos de código confeccionados para a *API* poderá não cobrir todos os cenários de uso.

1.3 Entrada

Nesta Seção, é apresentado os diferentes tipos de entrada utilizados pelas ferramentas *CSACTs* - ***C**ode **S**nippet or **A**PI **C**omponents **T**ools* - apresentadas nos trabalhos analisados. A entrada é utilizada pela ferramenta como critério para decidir quais elementos são relevantes para o usuário.

1.3.1 *Query* baseada em palavras-chaves

Um tipo de entrada que é utilizado pelas ferramentas *CSACTs* são as palavras-chaves. Como um exemplo, há a ferramenta *Assieme* (HOFFMANN; FOGARTY; WELD, 2007), que aceita como entrada palavras-chaves e retorna componentes de uma *API* (pacotes, classes e métodos) associados às palavras-chaves digitadas pelo usuário. *Assieme* infere a relação de uma palavra-chave e um componente de uma *API* baseada nos textos de páginas *web* contendo trechos de código que referenciam o componente da *API*. Por exemplo, para a *query "output acrobat"*, a ferramenta retorna o pacote *com.lowagie.text.pdf*, pacote da *API iText*[10] para manipulação de arquivos *PDF*, desde que a ferramenta tenha encontrado alguma página *web* cujo conteúdo possua os termos *output* e *acrobat* e cujo conteúdo contenha um trecho de código que faça referência ao pacote *com.lowagie.text.pdf*. Uma vez que a ferramenta tenha exibido os componentes da *API* (pacotes, classes e métodos) baseada nas palavras-chaves digitadas, o usuário pode selecionar um dos componentes para que a ferramenta exiba as páginas *web* contendo os trechos de códigos que façam referência para o componente selecionado. Outra ferramenta que utiliza palavras-chaves é a ferramenta *Mica* (STYLOS; MYERS et al., 2006), que redireciona uma *query* digitada pelo usuário contendo palavras-chaves para o *Google* e processa as páginas retornadas a fim de apresentar ao usuário somente aquelas que possuam trechos de código *Java*. A vantagem de utilizar uma entrada baseada em palavras-chaves é que é um método simples e intuitivo. Não há a necessidade do usuário possuir informações detalhadas de forma antecipada. Ele precisa apenas conhecer algum termo que semanticamente esteja relacionado com o elemento desejado (HENNINGER, 1991, p. 253).

CodeFinder (HENNINGER, 1991), uma ferramenta *CSACT* para localizar funções *Lisp*, também utiliza palavras-chaves, no entanto se diferencia por implementar um mecanismo que expande o resultado da busca através de uma técnica denominada *Spreading Activation*, que modela a associação entre termos (palavras-chaves) e documentos (funções *Lisp*) através de um grafo. Termos e documentos são representados por nós no grafo, enquanto a associação entre um termo e um documento é representado por uma aresta no grafo. Um nó é capaz de ativar seus nós vizinhos através de um processo iterativo. O processo se inicia com o nó representado o termo presente em uma *query* como um nó ativado e com um valor de ativação igual a 1. No passo seguinte, o nó ativado (no caso representando o termo presente na *query*) ativa os nós vizinhos passando o valor de ativação, no entanto o valor de ativação sofre um decaimento por um fator. Por exemplo, se o fator de decaimento for igual 0.5, então o valor de ativação passado para o nó vizinho é a metade do valor de ativação do nó que está realizando a ativação. Esse processo continua até que os valores de ativação atinjam um limite inferior. Os documentos com um valor de ativação maior que zero são considerados associados ao termo da *query* e o valor de

[10] <http://itextpdf.com/api>

ativação de um documento exprime a magnitude da associação entre o documento e o termo presente na *query*, o que é então utilizado para ranquear os documentos no resultado da *query*. Detalhes desse processo são apresentados em (HENNINGER, 1991, p. 255), que prevê também a ocorrência de ciclos e ativações provenientes de mais de um nó do grafo.

CodeFinder possui também suporte a recuperação por reformulação. Recuperação por reformulação é o processo através do qual *queries* são melhoradas pelo usuário até que este encontre uma *query* que produza o resultado desejado (YE; FISCHER; REEVES, 2000, p. 61). Na ferramenta *CodeFinder*, após o usuário digitar uma *query* e solicitar a busca, além da ferramenta apresentar as funções *Lisp* de acordo com a *query*, ela retorna um conjunto de palavras-chaves relacionadas com as palavras-chaves presentes na *query*. Essas palavras-chaves relacionadas podem ser adicionadas na *query* original de forma iterativa até que o resultado desejado seja encontrado. Uma estratégia semelhante é encontrada na ferramenta *Mica*. A ferramenta, além de apresentar as páginas *web* contendo os trechos de código, exibe uma barra lateral contendo palavras-chaves. As palavras-chaves exibidas nessa barra lateral são nomes de componentes de *APIs* (nomes de classes, interfaces e métodos) encontradas nas páginas *web* retornadas no resultado da busca. O usuário tem a opção de selecionar uma das palavras-chaves exibidas na barra lateral a fim de executar um nova *query* baseada na palavra-chave selecionada.

1.3.2 *Query* baseada em informações específicas do domínio

Query baseada em palavras-chaves é um tipo de entrada utilizada por sistemas de busca de um domínio mais genérico, como por exemplo sistemas de busca de páginas *web* (*Google*[11] e *Bing*[12]). Utilizar uma *query* baseada somente em palavras-chaves em uma ferramenta específica de um domínio, como as ferramentas *CSACTs*, é uma estratégia que ignora o potencial de se utilizar critérios mais detalhados e mais próximos do domínio considerado, no caso o domínio de desenvolvimento de software. Logo, algumas *CSACTs* utilizam informações mais específicas do que as palavras-chaves. Por exemplo, a ferramenta *APIExample* (WANG et al., 2011) aceita como entrada o identificador de uma classe e retorna trechos de código que façam referência para a classe especificada.

Um outro exemplo de ferramenta que utiliza *queries* mais específicas, é o sistema de busca de trechos de código denominado *Sourcerer* (BAJRACHARYA et al., 2006). *Sourcerer* aceita *queries* em diferentes modos. As *queries* são baseadas em palavras-chaves, no entanto o resultado depende do modo escolhido. Seis modos de pesquisa são definidos: *C (Component), C-use (Component USE), F (Function), F-use (Function USE), Fi-C (Control Structure Fingerprint) e Fi-T (Type Fingerprint)*. Os modos *C (Component)* e *F (Function)* são utilizados para buscar por trechos de código que declaram, respectivamente,

[11] <http://www.google.com>
[12] <http://www.bing.com>

a classe ou o método relacionados com as palavras-chaves digitadas na *query*. Os modos *C-use (Component USE)* e *F-use (Function USE)* são utilizados para buscar por trechos de código que utilizam, respectivamente, a classe ou o método relacionados com as palavras-chaves digitadas na *query*. *Fi-C (Control Structure Fingerprint)* e *Fi-T (Type Fingerprint)* são modos utilizados para recuperar trechos de código baseado em critérios estruturais mais detalhados. *Fi-C (Control Structure Fingerprint)* permite criar *queries* com restrições de acordo com as estruturas de controle presentes nos trechos de código. Por exemplo, através deste modo é possível especificar uma *query* com a seguinte semântica: "procure por trechos de código de métodos contendo mais de 50 sentenças *switch*". *Fi-T (Type Fingerprint)* possibilita criar *queries* com restrições de acordo com a estrutura de classes, métodos, atributos, construtores, etc. Através deste modo é possível, por exemplo, especificar uma *query* com a seguinte semântica: "procure por trechos de código que definem classes que possuam campos do próprio tipo e que não tenham métodos declarados".

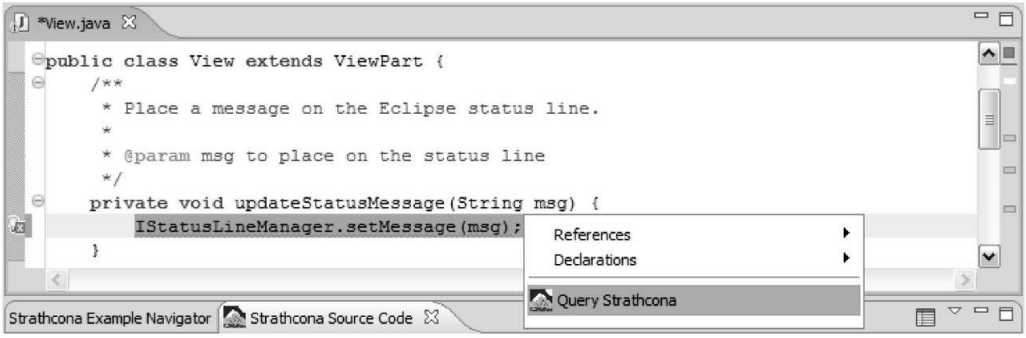
1.3.3 Utilizando informações no contexto de programação

Uma *query* mais específica, com informações detalhadas do trecho de código ou do componente de uma *API* desejado, pode produzir resultados mais precisos, comparado com *queries* menos específicas, como as *queries* baseadas somente em palavras-chaves. No entanto, o usuário pode não possuir as informações necessárias para formular uma *query* mais específica. Além disso, uma ferramenta que depende de *queries* mais específicas está subordinada a disposição do usuário em formular *queries* que normalmente são maiores e mais complexas (HENNINGER, 1991, p. 253).

A fim de poupar o usuário de formular *queries* longas e complexas, algumas ferramentas integradas à *IDE* beneficiam-se do acesso às informações presentes no contexto de programação para automaticamente formular *queries* com informações específicas do domínio de desenvolvimento de software. Por exemplo, a ferramenta *Stratchona* (HOLMES; WALKER; MURPHY, 2006) permite que o usuário selecione um trecho de código em um código fonte em edição na *IDE Eclipse* e solicite que ela realize uma busca por trechos de código estruturalmente semelhantes ao trecho de código selecionado. Ela considera informações estruturais do trecho de código selecionado e informações estruturais da classe que contém o trecho de código selecionado. As informações estruturais consideradas pela ferramenta são quais métodos o trecho de código selecionado chama, quais tipos e quais atributos de classe são referenciados pelo trecho de código e qual classe é estendida pela classe que contém o trecho de código selecionado. Por exemplo, na Figura 14 é exibido um cenário de uso da ferramenta, no qual o usuário seleciona um trecho de código como entrada para uma *query*. Do trecho de código selecionado são extraídas as seguintes informações estruturais: é chamado o método *IStatusLineManager.setMessage(String)*, os tipos *IStatusLineManager* e *String* são referenciados e a classe que contém o trecho de código

selecionado estende a classe *ViewPart*. Para buscar trechos de código presentes em seu repositório estruturalmente semelhantes ao trecho de código selecionado, *Stratchona* define quatro heurísticas de busca: *INHERITS*, *CALLS*, *USES* e *REFERENCES*. *INHERITS* procura por trechos de código de métodos cuja classe estenda a mesma classe estendida pela classe que contém o trecho de código selecionado (no exemplo apresentado, é a classe *ViewPart*). *CALLS* procura por trechos de código que chamam os mesmos métodos. *USES* procura por trechos de código que fazem referência para os mesmos tipos. E *REFERENCES* procura por trechos de código que fazem referência para os mesmos atributos de classes. Após o conjunto de trechos de código serem recuperados para cada heurística, eles são ranqueados pela quantidade de heurísticas que os retornaram. Por exemplo, um trecho de código retornado por ambas heurísticas *INHERITS* e *CALLS* é melhor ranqueado do que um trecho de código retornado somente pela heurística *USES*.

Figura 14 – Uma *query* na ferramenta *Stratchona*



Fonte: adaptado de (HOLMES; WALKER; MURPHY, 2006, p. 954)

Outras informações utilizadas como entrada por ferramentas integradas a uma *IDE* apresentadas nos trabalhos analisados incluem: termos presentes em comentários do código fonte em edição (YE; FISCHER; REEVES, 2000); assinatura de um método recém declarado no código fonte em edição (YE; FISCHER; REEVES, 2000); o nome do método e da classe que estão sendo editados (ZHONG et al., 2009); nomes dos elementos (classes e métodos) do projeto de software que o usuário tenha recentemente interagido (editado ou acessado) (SAWADSKY; MURPHY, 2011; ROBBES; LANZA, 2010); interface de uma classe ainda não implementada extraída de um teste unitário (LEMOS et al., 2007; HUMMEL; JANJIC; ATKINSON, 2008); tipos das variáveis no escopo do código fonte em edição (MANDELIN et al., 2005); o conjunto de métodos utilizados pela classe em edição (MCCAREY; CINNÉIDE; KUSHMERICK, 2005); propriedades detalhadas da classe em edição, como quais interfaces ela implementa, quais métodos ela sobrescreve, quais métodos ela chama e quais outras classes ela instancia (BRUCH; SCHÄFER; MEZINI, 2006). Há ferramentas que utilizam informações no contexto de programação a fim de complementar uma *query* baseada em palavras-chaves, por exemplo, a ferramenta *Blueprint* (BRANDT

et al., 2010, p. 515) aceita *queries* baseadas em palavras chaves e complementa a *query* com o nome e a versão da linguagem de programação do código fonte em edição na *IDE*. A ferramenta *SnipMatch* (WIGHTMAN et al., 2012, p. 220) utiliza os tipos e os nomes das variáveis no escopo do código fonte em edição na *IDE* a fim de ranquear e filtrar trechos de código retornados por uma *query* baseada em palavras-chaves.

1.3.4 Ativação automática da ferramenta

Embora algumas ferramentas explorem as informações presentes no contexto de programação, muitas dessas ferramentas exigem uma ação explícita por parte do usuário. Apenas para citar alguns exemplos, temos: a ferramenta *Strathcona*, mencionada na Subseção anterior, exige que o usuário selecione um trecho de código e solicite que a ferramenta execute uma *query* através de um menu suspenso (ver Figura 14); *autocompletadores de código* (descritos na Subseção 1.1.3) normalmente exigem que o usuário utilize um atalho de teclado para ativar a ferramenta, por exemplo, o *autocompletador de código* nativo do *Eclipse* é ativado ao pressionar *CONTROL+ESPAÇO* durante a edição de um código fonte *Java*; a ferramenta *Blueprint* (BRANDT et al., 2010) exige que o usuário digite algumas palavras-chaves e ative a ferramenta através de um atalho de teclado.

Exigir que usuário realize uma ação explícita para ativar a ferramenta possui três desvantagem principais. Primeiro, o usuário precisa decidir quais informações serão utilizadas como entrada. Por exemplo, no caso de ferramentas que utilizam *queries* baseadas em palavras-chaves (apresentadas na Subseção 1.3.1) há a necessidade de decidir quais palavras-chaves deverão ser utilizadas na *query*. Em uma ferramenta como a ferramenta *Stratchona*, o usuário necessita decidir qual trecho de código será utilizado como entrada. Essa necessidade representa um custo. O usuário poderá deixar de utilizar a ferramenta caso julgue que este custo não compense o valor do resultado produzido pela ferramenta (YE; FISCHER; REEVES, 2000, p. 62). Segundo, o usuário pode desconhecer que a ferramenta exista ou que esteja disponível no ambiente de programação e, neste caso, mesmo que a ferramenta fosse útil a ele, ela estaria sendo subtilizada (ROBBES; LANZA, 2010, p. 202). Terceiro, a ferramenta poderá deixar de recomendar um elemento relevante ao usuário, pois o elemento pode ser um elemento desconhecido a ele. O usuário pode desconhecer que exista uma determinada *API* que resolva um determinado problema, consequentemente, não irá nem mesmo utilizar uma entrada para a ferramenta (como palavras-chaves) relacionada a *API* (YE; FISCHER; REEVES, 2000, p. 61).

A fim de poupar o usuário de tomar decisões relacionadas a entrada, diminuir o risco dele deixar de utilizar a ferramenta e diminuir o risco dele deixar de utilizar um elemento relevante simplesmente por desconhecer que o elemento exista, algumas ferramentas apresentam resultados ao usuário sem a necessidade de uma ação explícita. Por exemplo, a ferramenta *Rascal* (MCCAREY; CINNÉIDE; KUSHMERICK, 2005),

de forma automática e constante, recomenda métodos para o usuário utilizar baseada no conjunto de métodos já utilizados pela classe atualmente em edição na *IDE*. Outros exemplos incluem: *CodeBroker* (YE; FISCHER; REEVES, 2000), que recomenda métodos automaticamente assim que o usuário finaliza a edição de um bloco de comentário em um código fonte *Java* e também logo após o usuário declarar a assinatura de um novo método; *Fishtail* (SAWADSKY; MURPHY, 2011), que exibe uma lista de *links* para páginas *web*, supostamente contendo exemplos de trechos de códigos relevantes ao usuário, atualizada automaticamente e periodicamente baseada nos elementos (classes, métodos ou atributos de classes) que o usuário tenha recentemente interagido (acessado ou editado); *Code Conjurer* (HUMMEL; JANJIC; ATKINSON, 2008), que automaticamente torna-se ativo assim que um teste unitário é parcialmente finalizado e sugere classes para a implementação completa do teste unitário; e a ferramenta *OCompletion* (ROBBES; LANZA, 2010), um *autocompletador de código* que automaticamente sugere métodos e classes enquanto o desenvolvedor está editando um código fonte.

Uma desvantagem das ferramentas que automaticamente apresentam elementos ao usuário é que elas correm o risco de distrair o usuário de forma desnecessária, principalmente se ela apresentar elementos irrelevantes para a tarefa que ele esteja trabalhando (ROBBES; LANZA, 2010, p. 202).

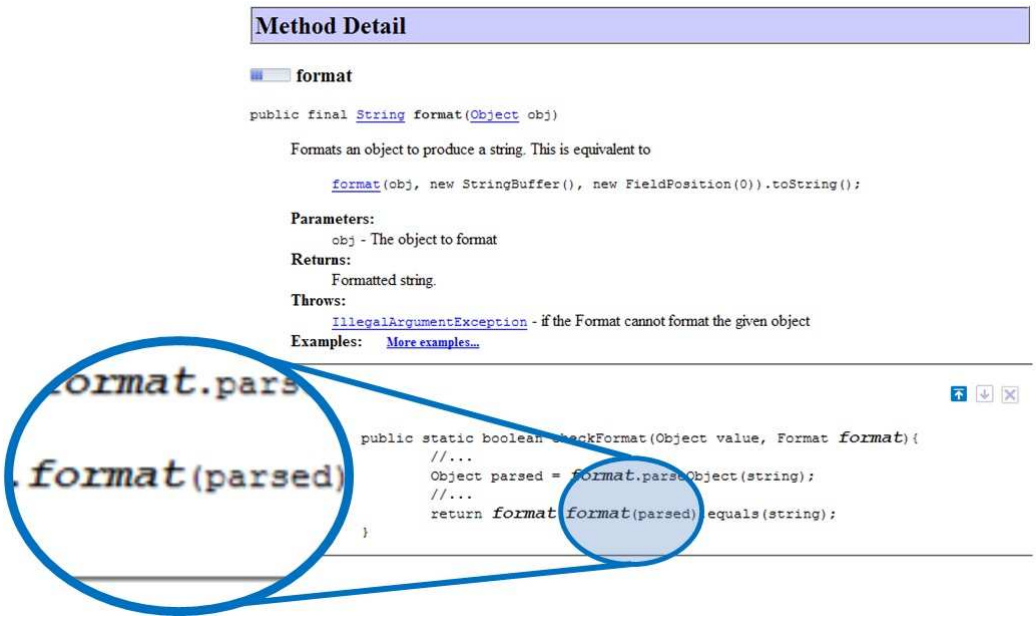
1.3.5 Ferramentas baseadas em documentação

Há ferramentas que têm o objetivo de complementar a documentação de uma *API*. Tais ferramentas não recebem um entrada específica do usuário, mas recebe a documentação completa da *API* no formato *Javadoc* e produz uma documentação complementada com trechos de código. Os trechos de código são extraídos de fonte de dados, como a *Web* (KIM et al., 2010), os projetos de código aberto (MONTANDON et al., 2013) ou os testes unitários do próprio projeto de desenvolvimento da *API* (ZHU et al., 2014). Para cada método presente na documentação da *API*, a documentação do método é automaticamente complementada com trechos de código que demonstram a utilização do método. Na Figura 15, é apresentado um trecho da documentação do método *format(Object)*, método da classe *java.text.Format* da *API* padrão da plataforma *Java*, complementada através da ferramenta proposta por Kim et al. (2010).

1.4 Resultado

Após receber uma entrada e decidir quais elementos são relevantes, a ferramenta *CSACT* deve apresentar o resultado ao usuário através de elementos gráficos. Nesta Seção, são apresentadas as estratégias adotadas pelas ferramentas em relação a apresentação do resultado para o usuário. São discutidas as estratégias sob três perspectivas: o tipo do

Figura 15 – *Javadoc* complementado automaticamente



Fonte: adaptado de (KIM et al., 2009, p. 542)

resultado (Subseção 1.4.1), o suporte ao entendimento do resultado (Subseção 1.4.2) e o ranqueamento dos elementos do resultado (Subseção 1.4.3).

1.4.1 Tipo de resultado

Em relação ao tipo de resultado apresentado, as ferramentas *CSACTs* são divididas em dois grupos: um grupo de ferramentas que apresentam trechos de código e outro grupo de ferramentas que apresentam componentes de uma *API*. Ferramentas que apresentam componentes fornecem um suporte a reutilização de software que procura reutilizar elementos de software prontos e bem definidos de uma *API*, os quais são denominados de componentes, como por exemplo, funções *Lisp* (YE; FISCHER; REEVES, 2000), métodos (ROBBES; LANZA, 2010) e classes *Java* (HUMMEL; JANJIC; ATKINSON, 2008). Por sua vez, ferramentas que apresentam trechos de código oferecem um suporte maior a uma reutilização baseada na copia e na cola de partes de trechos de código (KIM et al., 2004; BRANDT et al., 2009b). É necessário que o usuário copie partes do trecho de código apresentado pela ferramenta, cole em um código fonte no projeto de software em desenvolvimento e o altere a fim de adaptá-lo ao contexto do software em desenvolvimento.

Embora um componente de uma *API* possa ser utilizado diretamente, nem sempre o que o programador deseja pode ser implementado através da utilização de um único componente ou através da utilização de uma sequência bem definida de componentes. Por exemplo, em muitos casos, a fim do programador implementar determinada característica do software é necessário utilizar vários métodos e classes de uma *API* através de um

protocolo específico, ou seja, os objetos das classes da *API* devem interagir entre si através de chamadas de métodos em uma ordem específica e passando argumentos específicos (KIM et al., 2004). Tal protocolo de uso somente pode ser reutilizado pelo programador através de trechos de código. Por exemplo, na Figura 16 é apresentado um trecho de código na linguagem *C++* que demonstra um protocolo de uso de uma *API* de manipulação de documentos *DOM*[13]. O trecho de código demonstra como realizar uma iteração através dos elementos *DOM* filhos de um documento *DOM*. Muitas informações são apresentadas nesse curto trecho de código a fim de esclarecer o usuário como realizar uma iteração nos elementos *DOM* de um documento: é necessário obter a lista de nós filhos do documento (linha *1*); obter a quantidade de nós filhos do documento (linha 2); realizar uma iteração através de um laço (linha *4*) a fim de se obter cada nó filho do documento (linha *6*); é necessário, por fim, verificar se o nó filho do documento é de fato um elemento *DOM* (linha 9).

Figura 16 – Exemplo de protocolo de uso de uma *API*

```
1 DOMNodeList *children = doc->getChildNodes();
2 int numChildren = children->getLength();
3
4 for (int i=0; i<numChildren; ++i)
5    {
6       DOMNode *child = (children->item(i));
7       if (child->getNodeType() == DOMNode.ELEMENT_NODE)
8       {
9          DOMElement *element = (DOMElement*)child
```

Fonte: adaptado de Kim et al. (2004, p. 88)

Um aspecto a ser observado em relação as ferramentas *CSACTs* que apresentam trechos de código é o tamanho ou o escopo do trecho de código apresentado. O tamanho e o escopo varia de uma ferramenta para outra. Há ferramentas que apresentam como resultado todo o conteúdo de arquivos de código fonte (BLACK DUCK SOFTWARE, 2015; THE SPARS PROJECT & OSAKA UNIVERSITY, 2015; ARAGON CONSULTING GROUP, 2015), há ferramentas que apresentam trechos de código extraídos de métodos (HOLMES; WALKER; MURPHY, 2006; ZHONG et al., 2009; MAR; WU; JIAU, 2011), enquanto outras apresentam trechos de código mais concisos, como trechos de código sumarizados, com linhas ou sentenças eliminadas do trecho de código original (BUSE; WEIMER, 2012; ZHU et al., 2014; KIM et al., 2010; MONTANDON et al., 2013) ou trechos de código que são inerentemente concisos, por exemplo, trechos de código extraídos de páginas *web* (HOFFMANN; FOGARTY; WELD, 2007; BRANDT et al., 2010) ou

[13] *DOM*, *Document Object Model*, modelo padronizado pela *W3C* utilizado como uma representação intermediária, em memória, do conteúdo de arquivos *HTML* e *XML* <http://www.w3.org/DOM/Overview.html>.

trechos de código confeccionados manualmente (ONEY; BRANDT, 2012; WIGHTMAN et al., 2012; Snip2Code, 2014). Um programador que esteja procurando por um trecho de código para realizar determinada tarefa precisará de um trecho de código com um escopo diferente dependendo do contexto (OHTANI et al., 2015, p. 22). Algumas vezes precisará de um trecho de código que demonstre a implementação completa de uma classe ou de um método, em outras situações precisará de um trecho de código com poucas linhas que apenas ilustre alguma operação.

Em relação às ferramentas que apresentam componentes de *APIs*, é possível encontrar ferramenta que apresentam diversos tipos de resultados. Os *autocompletadores de código*, apresentados na Subseção 1.1.3, apresentam um método que possa ser chamado para um determinado objeto. A ferramenta *CodeFinder* (HENNINGER, 1991) apresenta funções *Lisp* (HENNINGER, 1991). A ferramenta *CodeBroker* (YE; FISCHER; REEVES, 2000) apresenta métodos que possuam determinada assinatura. As ferramentas *Code Conjurer* (HUMMEL; JANJIC; ATKINSON, 2008) e *Codegenie* (LEMOS et al., 2007) apresentam classes que implementam determinado cenário de teste. A ferramenta *Calcite* (MOOTY et al., 2010) apresenta maneiras de instanciar uma determinada classe e a ferramenta *Fruit* (BRUCH; SCHÄFER; MEZINI, 2006) apresenta recomendações de determinadas tarefas relacionadas ao desenvolvimento orientado a objetos (estender uma determinada classe, sobrescrever determinado método, chamar determinado método, instanciar determinada classe e implementar determinada interface).

1.4.2 Suporte ao entendimento do resultado

Após uma ferramenta *CSACT* apresentar o resultado ao usuário, é necessário que o usuário selecione um elemento entre os elementos apresentados no resultado para posteriormente integrá-lo ao contexto de programação (HOLMES et al., 2009). No entanto, para o usuário julgar qual é o elemento relevante para a tarefa de programação, antes é necessário entendê-los. Esse processo de tentar entender os elementos do resultado apresentado pode ser uma tarefa não trivial (WU; MAR; JIAU, 2010; MAR; WU; JIAU, 2011). Por exemplo, para uma ferramenta que apresenta um determinado método de uma *API*, é necessário entender se o serviço disponibilizado pelo método corresponde ao serviço desejado e é necessário entender como os argumentos devem ser passados ao método para que o método comporte-se da maneira desejada. A fim de auxiliar o entendimento do resultado apresentado, ferramentas apresentam informações adicionais no resultado. Entre as informações adicionais apresentadas pelas ferramentas apresentadas nos trabalhos analisados, temos: uma descrição textual que informa o que o trecho de código apresentado realiza (BRANDT et al., 2010; ONEY; BRANDT, 2012; WANG et al., 2011; WIGHTMAN et al., 2012), documentação do componente (classe ou método) apresentado (YE; FISCHER; REEVES, 2000; BRUCH; SCHÄFER; MEZINI, 2006), documentação de

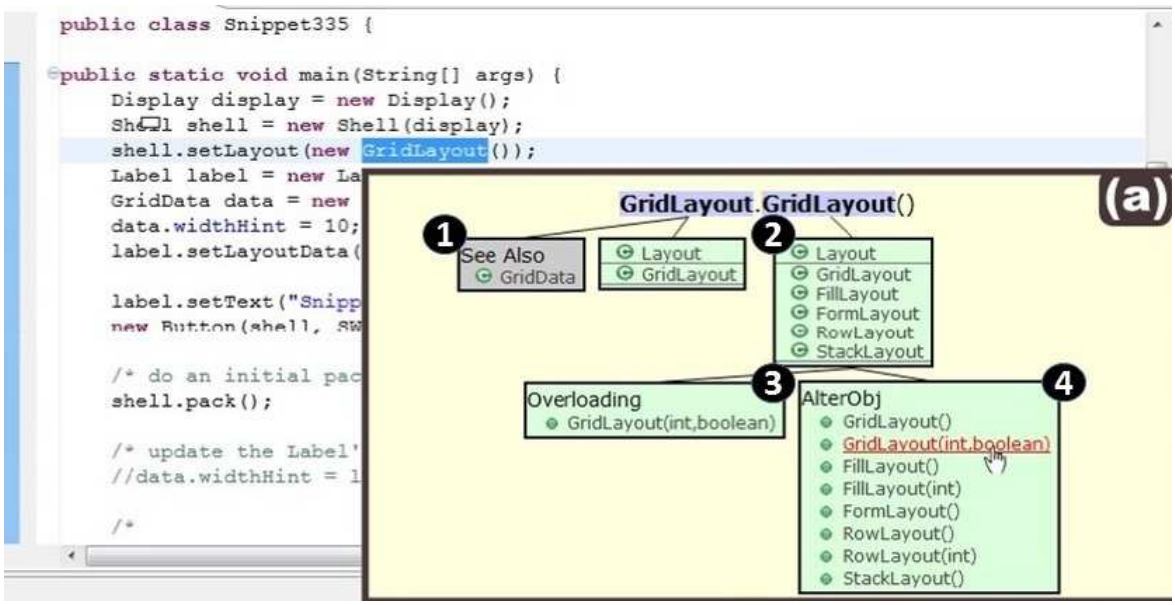
componentes de uma *API* referenciados no trecho de código apresentado (HOFFMANN; FOGARTY; WELD, 2007) e um exemplo da execução do trecho de código apresentado (BRANDT et al., 2010).

Quando a ferramenta apresenta trechos de código, muitas vezes também é necessário entender como componentes referenciados no trecho de código, como classes e métodos, se relacionam com outros componentes da *API*. A fim de fornecer um suporte maior para esse processo, a ferramenta *CoDocent* (WU; MAR; JIAU, 2010) exibe elementos gráficos que ilustram ao usuário o relacionamento de um componente referenciado no trecho de código com outros componentes. Na Figura 17, é ilustrado um cenário de uso da ferramenta. Um trecho de código, que demonstra a utilização da *API SWT*[14,15], é apresentado ao usuário. O trecho de código demostra a criação de uma janela e a inserção de elementos gráficos a serem exibidos nessa janela. Na *API SWT*, os elementos gráficos são posicionados na janela através de um objeto do tipo *GridLayout*, que é configurado na janela através do método *setLayout*. Durante a inspeção do trecho de código apresentado, o usuário seleciona um componente da *API* referenciado no trecho de código, no caso um construtor da classe *GridLayout*. É, então, exibido ao usuário um assistente com elementos gráficos a fim de apresentar informações a respeito dos relacionamentos do componente selecionado com outros componentes da *API* (item *a*). Nas informações apresentadas no assistente estão: o construtor selecionado é declarado na classe *GridLayout*; a classe *GridData* é referenciada na documentação da classe *GridLayout* (item *1*); a classe *GridLayout* estende a classe *Layout* e há outras classes que também estendem a classe *Layout*, como a classe *FillLayout*, *FormLayout*, etc (item *2*); há um outro construtor, com um parâmetro do tipo *int* e um parâmetro do tipo *boolean*, que sobrecarrega o construtor padrão selecionado (item *3*) e outros construtores poderiam ser utilizados em substituição do construtor selecionado (item *4*). O usuário tem a opção de selecionar algum componente apresentado no assistente a fim de exibir informações presentes na documentação. Por exemplo, o usuário poderia clicar no construtor da classe *GridLayout* com os parâmetros do tipo *int* e do tipo *boolean* (item destacado em vermelho no bloco marcado com o número *4*) a fim de obter mais informações a respeito do construtor. Todas essas informações apresentadas no assistente contribuem para o usuário entender o trecho de código e entender como adaptá-lo. O usuário poderia, por exemplo, entender que é possível utilizar um construtor da classe *FillLayout* a fim de posicionar os elementos da janela de uma maneira diferente da maneira proporcionada pela classe *GridLayout*.

Outra ferramenta que utiliza elementos gráficos a fim de fornecer um suporte melhor ao entendimento de trecho de código apresentado ao usuário é a ferramenta *PropER-Doc* (MAR; WU; JIAU, 2011). *ProperER-Doc* exibe um diagrama que representa a relação entre as classes referenciadas no trecho de código. No diagrama, as classes referenciadas

[14] *API* para exibição de elementos gráficos, como janelas, botões e áreas de texto.
[15] <https://www.eclipse.org/swt/>

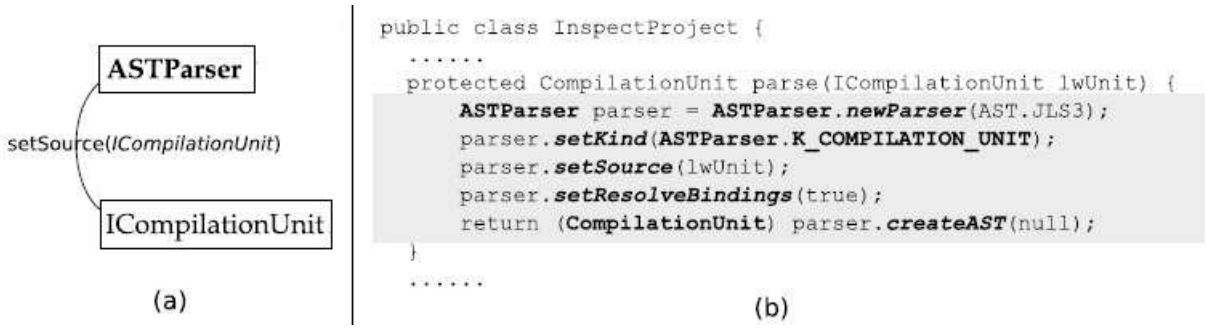
Figura 17 – Tela capturada da ferramenta *CoDocent*



Fonte: adaptado de (WU; MAR; JIAU, 2010, p. 138)

são representadas através de um retângulo e os relacionamentos entre as classes são representadas através de arestas rotuladas com o nome do método que relaciona uma classe com outra. Por exemplo, na Figura 18, no item *b* é apresentado um trecho de código e no item *a* é apresentado o diagrama correspondente. A classe *ASTParser* se relaciona com a classe *ICompilationUnit*, pois *ASTParser* possui um método *setSource* e um objeto do tipo *ICompilationUnit* é passado como argumento do método *setSource*.

Figura 18 – Diagrama apresentado pela ferramenta *PropER-Doc*

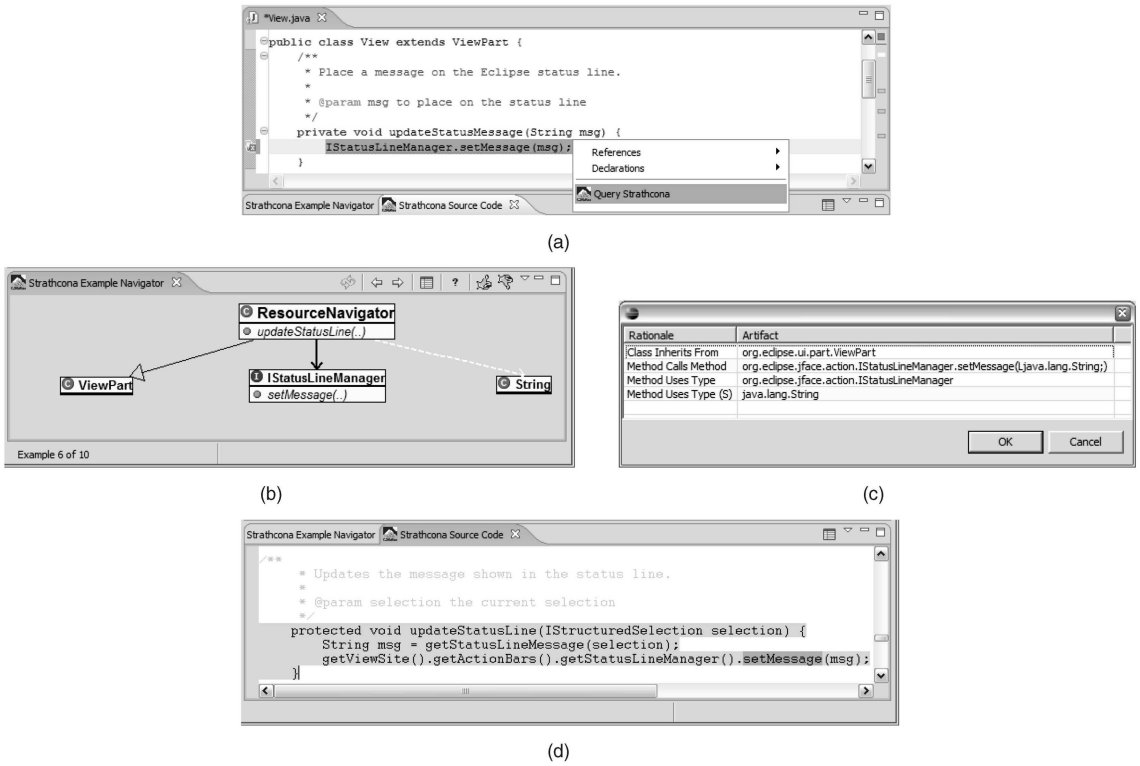


Fonte: adaptado de (MAR; WU; JIAU, 2011, p. 335)

Uma outra preocupação identificada entre as ferramentas analisadas é que algumas ferramentas apresentam informações que explicam o motivo pelo qual determinado elemento foi apresentado no resultado. Por exemplo, na Figura 19 são apresentadas telas da ferramenta *Stratchona* (HOLMES; WALKER; MURPHY, 2006). A ferramenta *Stratchona* é descrita na Subseção 1.3.3 e define quatro heurísticas a fim de buscar por trechos de

código em um repositório semelhantes a um trecho de código selecionado como entrada (item *a* da Figura 19). A ferramenta apresenta junto com um trecho de código retornado na busca (item *d*) um resumo do motivo pelo qual aquele trecho de código foi retornado (item *c*). O resumo é apresentado tanto através de uma representação textual como também através de uma representação gráfica (item *b*). Como as heurísticas de busca definidas pela ferramenta baseiam-se em informações estruturais, como qual classe é estendida pela classe de onde o trecho de código foi extraído, a representação gráfica dos motivos pelo qual um determinado trecho de código foi retornado é um digrama de classe da notação *UML*[16] (item *b*).

Figura 19 – Telas da ferramenta *Stratchona*



Fonte: (HOLMES; WALKER; MURPHY, 2006, p. 954)

Uma outra estratégia para facilitar a inspeção por parte do usuário do resultado apresentado, utilizada comumente por ferramentas que apresentam trechos de código (WANG et al., 2011; BLACK DUCK SOFTWARE, 2015; ARAGON CONSULTING GROUP, 2015; THE SPARS PROJECT & OSAKA UNIVERSITY, 2015; BRANDT et al., 2010), é destacar partes do trecho de código relacionados com a entrada. Por exemplo, na Figura 20 é apresentado um cenário de uso da ferramenta *Blueprint* (BRANDT et al., 2010). Os termos *load* e *image* são utilizados como entrada para a ferramenta (item

[16] *UML*, *Unified Modeling Language*, uma linguagem de modelagem de sistemas padronizada pela *OMG* <http://www.uml.org/>.

A). Um trecho de código é apresentado como resultado (item *B*) e as partes do trecho apresentado relacionadas com os termos digitados na entrada são destacadas (item *C*) a fim de exibir o motivo pelo qual o trecho de código foi retornado e facilitar a inspeção por parte do usuário.

Figura 20 – Tela da ferramenta *Blueprint*

Fonte: (BRANDT et al., 2010, p. 513)

1.4.3 Organização do resultado

Normalmente um resultado é uma coleção de elementos que são apresentados ao usuário e, conforme mencionado na Subseção anterior, o processo de investigar se um elemento é relevante entre os elementos apresentados é um processo oneroso. É esperado que um usuário consiga explorar apenas os dez primeiros elementos apresentados em um resultado de uma pesquisa (SAHAVECHAPHAN; CLAYPOOL, 2006, p. 421). Logo, é recomendável que os elementos apresentados no resultado sejam ordenados através de algum critério de relevância. No trabalho de Sahavechaphan e Claypool (2006), é apresentado a ferramenta *XSnippet* e investigado o uso de três critérios a fim de ranquear os trechos de código apresentados ao usuário: similaridade com o contexto de programação, frequência e tamanho. *XSnippet* é um ferramenta integrada à *IDE Eclipse* que apresenta ao usuário trechos de código de um repositório de arquivos de código fonte *Java*.

O objetivo da ferramenta *XSnippet* é apresentar trechos de código que demonstrem como instanciar determinado tipo. O usuário, ao editar um código fonte *Java* na *IDE Eclipse*, pode criar uma *query* para que a ferramenta apresente trechos de código que demonstram como obter um objeto de um determinado tipo T (tipo especificado na *query*) a partir dos objetos disponíveis no escopo do código fonte em edição. Os trechos de código apresentados pela ferramenta são trechos de código contendo uma sequência bem definida (sem estruturas de controle) de chamadas de métodos, atribuições e conversão de tipos (*downcasts* e *upcasts*) que parte de um objeto de um determinado tipo e produz um objeto do tipo desejado. Por exemplo, o trecho de código da Figura 21 é um trecho de código apresentado pela ferramenta *XSnippet* a fim de demonstrar como obter um objeto do tipo *ICompilationUnit* a partir de um objeto do tipo *ISelection*.

Figura 21 – Exemplo de trecho de código na ferramenta *XSnippet*

```
ISelection selection;
IStructuredSelection ss = (IStructuredSelection) selection;
Object obj = ss.getFirstElement();
IJavaElement je = (IJavaElement) obj;
IJavaElement ije = je.getAncestor(IJavaElement.COMPILATION_UNIT);
ICompilationUnit cu = (ICompilationUnit) ije
```

Fonte: adaptado de (SAHAVECHAPHAN; CLAYPOOL, 2006, p. 423)

Em relação aos três critérios utilizados pela ferramenta *XSnippet* para ordenar os trechos de código apresentados (similaridade com o contexto de programação, frequência e tamanho), para critério da similaridade com o contexto de programação, a similaridade é calculada baseando-se em dois aspectos: a similaridade do contexto parental e a similaridade do contexto de tipos. Para um trecho de código S que pertence a classe CS extraído do repositório de códigos fontes e uma *query* Q executada durante a edição de um método M pertencente à classe C no código em edição no ambiente de programação, temos que a similaridade do contexto parental é calculada baseando-se na similaridade entre a classe estendida por C e a classe estendida por CS e a similaridade entre as interfaces implementadas por C as interfaces implementadas por CS. A similaridade do contexto de tipos é calculada baseando-se na similaridade entre os tipos disponíveis no escopo de M e os tipos referenciados em S. A similaridade entre o trecho de código S e o contexto de programação é então calculada através da média entre a similaridade do contexto parental e a similaridade do contexto de tipos. Os cálculos das similaridades entre as classes e entre as interfaces são baseados em um algoritmo que recebe dois tipos (os tipos das classes ou os tipos das interfaces sendo comparadas) e retorna um valor numérico representando a similaridade entre os dois tipos. Trechos de código mais similares ao contexto de programação são melhores ranqueados dos que trechos de código menos similares.

O critério da frequência considera a frequência que um determinado trecho de código aparece no repositório. Um módulo denominado *PruningAgent* evita que trechos de código redundantes sejam apresentados no resultado de uma busca, no entanto a ferramenta registra para um determinado trecho de código a quantidade de trechos de código idênticos identificados no repositório. Essa quantidade é considerada como um critério para o ranqueamento, ou seja, trechos de código que são mais frequentes no repositório são melhores ranqueados do que trechos de código incomuns. Já o critério do tamanho considera o tamanho do trecho de código (quantidade de linhas). Trechos de código menores (com menos quantidade de linhas) são melhores ranqueados do que trechos de código maiores.

No trabalho de Sahavechaphan e Claypool (2006) os três critérios foram implementados para a ferramenta *XSnippet*. Além dos três critérios, foi implementado também um critério que combina os três critérios de forma hierárquica: os trechos de código são primeiramente ranqueados considerando a similaridade com o contexto de programação, trechos de código igualmente similares ao contexto de programação são, então, ranqueados considerando o critério da frequência e, por fim, trechos de código que são igualmente similares ao contexto de programação e igualmente frequentes são ranqueados considerando o critério do tamanho. Os quatro critérios (similaridade com o contexto de programação, frequência, tamanho e a combinação dos três) foram avaliados através de uma validação experimental que consistiu em comparar os critérios baseado em quais critérios retornaram trechos de código mais relevantes para determinadas tarefas de programação. O critério que considera a combinação dos três critérios apresentou a melhor performance seguido do critério da frequência.

A frequência também foi utilizada em outras ferramentas apresentadas nos trabalhos analisados como critério para ranquear os elementos apresentados (HOFFMANN; FOGARTY; WELD, 2007; MOOTY et al., 2010), assim como o tamanho (MANDELIN et al., 2005; KIM et al., 2010). Outros critérios que foram identificados para ordenar os elementos apresentados no resultado incluem: grau de relevância do trecho de código apresentado em relação a uma determinada classe utilizada como entrada na *query* (MAR; WU; JIAU, 2011); similaridade entre o nome da classe onde o trecho de código foi extraído e o nome da classe sendo editada no ambiente de programação e a similaridade entre o nome do método onde o trecho de código foi extraído e o nome do método sendo editado (ZHONG et al., 2009); quantidade de *commits* do arquivo de código fonte de onde o trecho de código apresentado foi extraído (MONTANDON et al., 2013); similaridade entre os termos digitados na entrada e termos relacionados com o trecho de código apresentado (WIGHTMAN et al., 2012); similaridade entre os termos digitados na entrada e os termos relacionados com a função *Lisp* apresentada (HENNINGER, 1991); data da última modificação do método apresentado ou data da última modificação de um método que chama o método apresentado (ROBBES; LANZA, 2010); similaridade entre termos presentes em

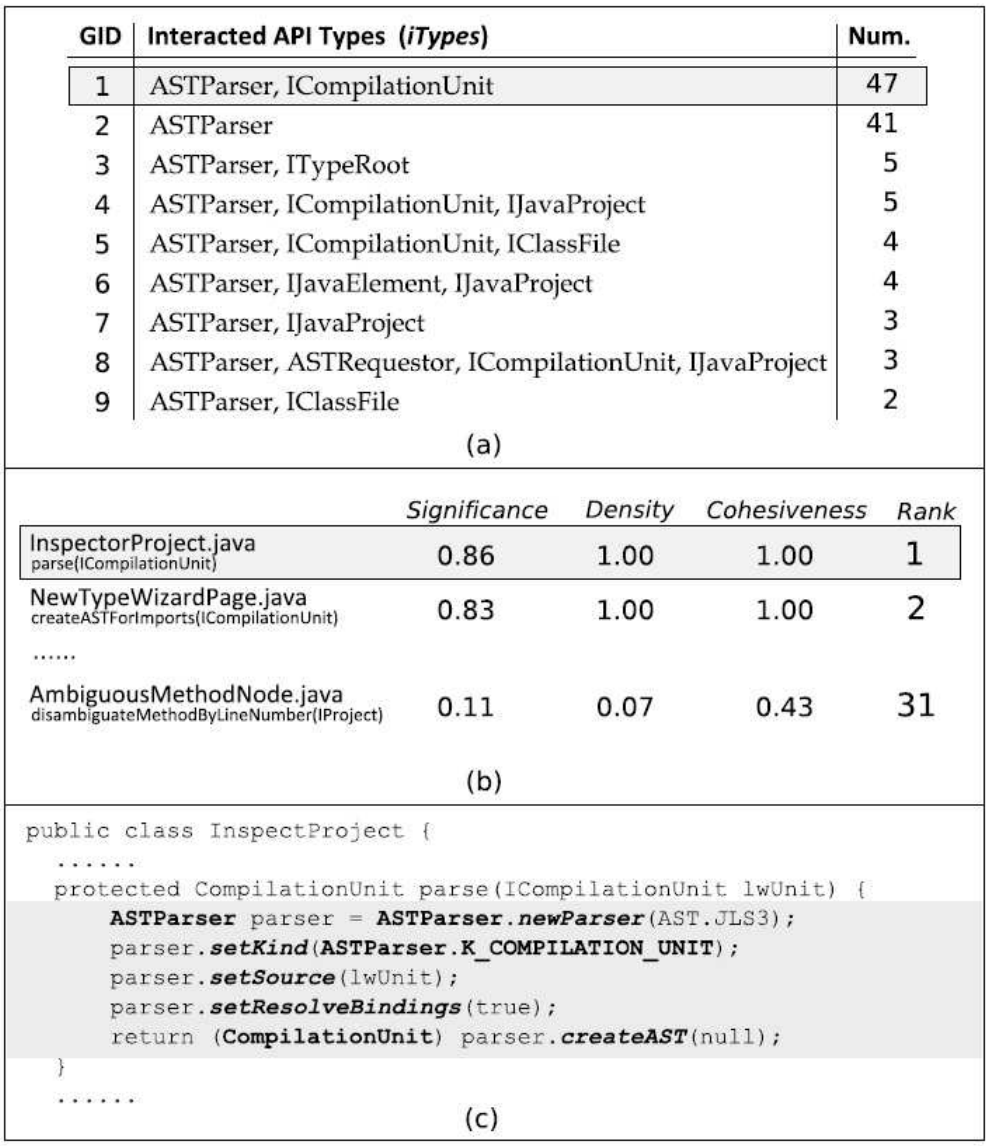
um bloco de comentário no código fonte em edição e termos presentes na documentação do método apresentado e a similaridade entre a assinatura de um método recentemente declarado no código fonte em edição e a assinatura do método apresentado (YE; FISCHER; REEVES, 2000).

Outra estratégia a fim de facilitar o processo de investigação por parte do usuário do resultado apresentado pela ferramenta é agrupar os elementos similares. Agrupar elementos similares evita que o usuário perca-se em uma grande quantidade de elementos que não trazem nenhuma informação nova no processo de investigação do resultado apresentado. Se um determinado elemento é avaliado irrelevante pelo usuário, ele pode passar a avaliar elementos em um outro grupo, pois outros elementos no grupo do elemento considerado irrelevante muito provavelmente também serão irrelevantes. Um exemplo de ferramenta que agrupa os elementos apresentados no resultado é a ferramenta *PropER-Doc* (MAR; WU; JIAU, 2011). A ferramenta aceita como entrada o nome de uma classe *Java* e retorna trechos de código extraídos a partir do *Google Code Search*[17] a fim de apresentar cenários de uso da classe. Na apresentação do resultado os trechos de código são agrupados de acordo com o conjunto de classes que colaboram com a classe especificada na entrada. Na Figura 22, é apresentado um cenário de uso da ferramenta. Após o usuário ter solicitado uma busca por trechos de código que utilizam a classe *ASTParser*, a ferramenta retorna os trechos de código de forma agrupada. O primeiro grupo contém trechos de código que utilizam a classe *ASTParser* em colaboração com a classe *ICompilationUnit* (item *a*). O grupo pode ser selecionado a fim de exibir os arquivos código fonte *Java* que contém os trechos de código pertencentes ao grupo (item *b*), no caso trechos de código que demonstram como a classe *ASTParser* é utilizada em colaboração com a classe *ICompilationUnit*. Um arquivo pode ser selecionado a fim de exibir o trecho de código que utiliza a classe de interesse (item *c*).

Outras ferramentas que agrupam o resultado são: a ferramenta *Assieme* (HOFFMANN; FOGARTY; WELD, 2007), que agrupa trechos de código pelos componentes da *API* que são utilizados no trecho de código, por exemplo, trechos de código que utilizam determinados pacotes, determinadas classes e determinados métodos; a ferramenta *MAPO*, (ZHONG et al., 2009) que agrupa trechos de código pela sequência de métodos que o trecho de código chama; e a ferramenta *APIExample* (WANG et al., 2011), que aceita como entrada o nome de uma classe e retorna trechos de código que façam referência para a classe agrupados pelo conjunto de métodos chamados para a classe.

[17] *Google Code Search* foi um sistema mantido pela empresa *Google* que realizava a busca de trechos de código extraídos de projetos de código aberto. A empresa deixou de manter o sistema em 2012 (GOOGLE, 2011).

Figura 22 – Trechos de código agrupados na ferramenta *PropER-Doc*



GID	Interacted API Types (*iTypes*)	Num.
1	ASTParser, ICompilationUnit	47
2	ASTParser	41
3	ASTParser, ITypeRoot	5
4	ASTParser, ICompilationUnit, IJavaProject	5
5	ASTParser, ICompilationUnit, IClassFile	4
6	ASTParser, IJavaElement, IJavaProject	4
7	ASTParser, IJavaProject	3
8	ASTParser, ASTRequestor, ICompilationUnit, IJavaProject	3
9	ASTParser, IClassFile	2

(a)

	Significance	*Density*	*Cohesiveness*	*Rank*
InspectorProject.java parse(ICompilationUnit)	0.86	1.00	1.00	1
NewTypeWizardPage.java createASTForImports(ICompilationUnit)	0.83	1.00	1.00	2
.....				
AmbiguousMethodNode.java disambiguateMethodByLineNumber(IProject)	0.11	0.07	0.43	31

(b)

```
public class InspectProject {
  ......
  protected CompilationUnit parse(ICompilationUnit lwUnit) {
      ASTParser parser = ASTParser.newParser(AST.JLS3);
      parser.setKind(ASTParser.K_COMPILATION_UNIT);
      parser.setSource(lwUnit);
      parser.setResolveBindings(true);
      return (CompilationUnit) parser.createAST(null);
  }
  ......
```

(c)

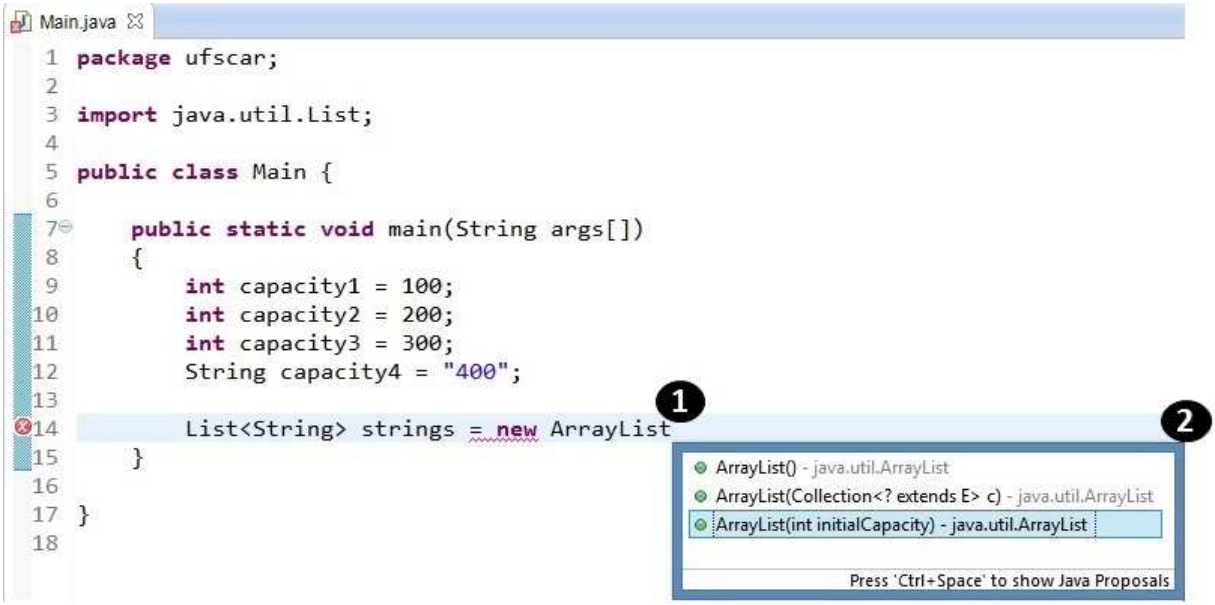
Fonte: adaptado de (MAR; WU; JIAU, 2011, p. 332)

1.5 Integração do elemento selecionado

Após o usuário julgar que determinado elemento apresentado no resultado é útil para a tarefa de programação que ele está realizando, é necessário adaptá-lo ao contexto do software que está sendo desenvolvido (HOLMES et al., 2009). Adaptações necessárias podem incluir a renomeação de variáveis, inclusão de dependências (WIGHTMAN et al., 2012; MOOTY et al., 2010) e eliminação de sentenças do trecho de código original (BRANDT et al., 2009a; KIM et al., 2004). Para componentes de uma *API*, como métodos e classes, por vezes é necessário passar os argumentos para se chamar um método (ZHANG et al., 2012) ou para se instanciar uma classe (MOOTY et al., 2010). Por exemplo, nas Figuras 23 e 24, é apresentado um cenário de uso do *autocompletador de código* nativo

da *IDE Eclipse* (os *autocompletadores de código* são apresentados na Subseção 1.1.3). O usuário está editando um arquivo código fonte *Java* e, após digitar o conteúdo da linha 14 (Figura 23), o qual declara uma variável do tipo *List* e contém o início de uma sentença de um possível construtor (item 1 da Figura 23), ele pressiona o atalho de teclado *CONTROL + ESPAÇO*. O *autocompletador de código* é ativado e recomenda três construtores, todos para a classe *java.util.ArrayList* (item 2 da Figura 23). O usuário seleciona o construtor com um parâmetro do tipo *int*. Assim que o construtor é selecionado, automaticamente a classe *java.util.ArrayList* é importada no código fonte em edição (linha 3 na Figura 24). Além disso, baseado nas variáveis no escopo do código fonte em edição, são sugeridos argumentos para o construtor (item 1 da Figura 24).

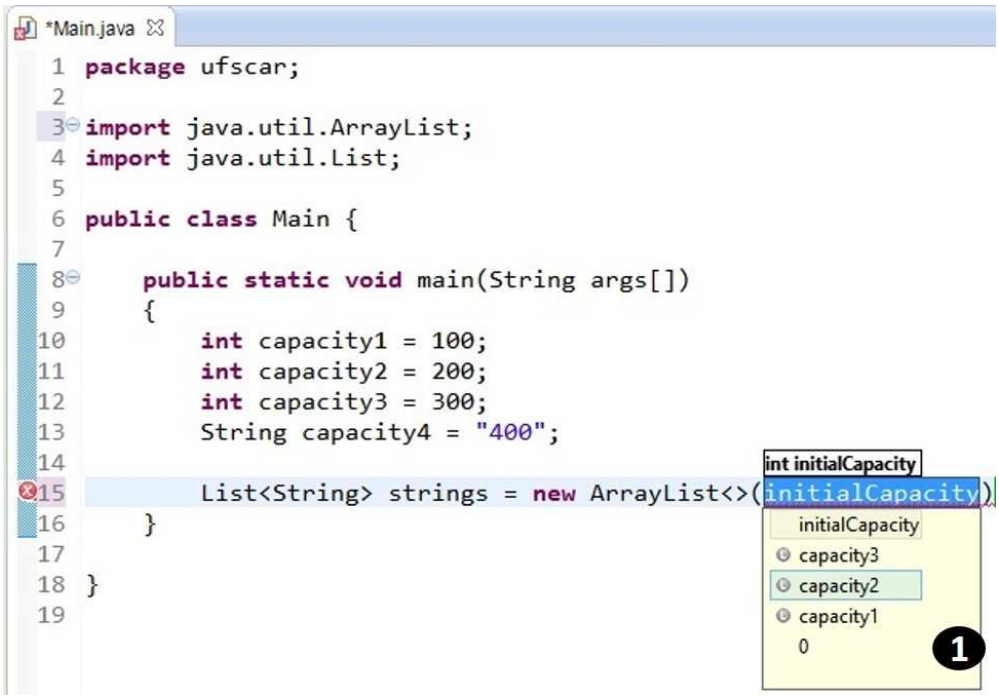
Figura 23 – Sugestão de construtores no *autocompletador* do *Eclipse*



Fonte: (ECLIPSE FOUNDATION, 2015)

Ferramentas *CSACTs* que apresentam componentes de uma *API*, como métodos e classes, em geral apresentam algum tipo de facilidade para adaptar o elemento apresentado no contexto de programação do usuário, como é o caso por exemplo dos *autocompletadores de código* nativos das *IDEs Netbeans 8.0* (ORACLE CORPORATION, 2014) e *Visual Studio 2013* (MICROSOFT CORPORATION, 2014), os quais também realizam o *import* necessário ao selecionar uma classe apresentada. Entre os trabalhos acadêmicos analisados, é possível citar a ferramenta *Calcite* (MOOTY et al., 2010), um *autocompletador de código* que apresenta maneiras de instanciar uma determinada classe de interesse. As maneiras de instanciar as classes podem ser através de um construtor da própria classe como também através de métodos de uma outra classe que retornem um objeto da classe de interesse. A ferramenta, após o usuário ter escolhida uma maneira de instanciar a classe, é capaz de declarar variáveis de uma outra classe necessária para instanciar a classe de interesse

Figura 24 – Sugestão de argumentos no *autocompletador* do *Eclipse*



Fonte: (ECLIPSE FOUNDATION, 2015)

e realiza tanto a importação da classe de interesse como a importação da outra classe. Outra ferramenta que apresenta funcionalidades para integrar o resultado apresentado é a ferramenta *Code Conjurer* (HUMMEL; JANJIC; ATKINSON, 2008). *Code Conjurer* implementa uma busca por classes baseada nos testes unitários do projeto de software em desenvolvimento no ambiente de programação. Após o usuário ter definido um teste unitário, a ferramenta extrai a interface da classe testada no teste unitário e, baseada na interface da classe, gera uma *query* para uma outra ferramenta denominada *Merobase*, que é um sistema de busca de trechos de código de projetos de código aberto e que retorna para o *Code Conjurer* classes prontas e compatíveis com a *interface* definida na *query*. As classes retornadas pelo *Merobase* são automaticamente testadas de acordo com o teste unitário definido e classes que não passam no teste não são apresentadas ao usuário. Se o usuário selecionar alguma classe entre as classes apresentadas, *Code Conjurer* encarrega-se de integrar a classe no projeto resolvendo as dependências necessárias. Se a classe selecionada depender de outras classes, outras classes dentro do mesmo pacote da classe selecionada são também integradas ao projeto. O processo não é estendido a outros pacotes. Outra ferramenta que utiliza os testes unitário a fim de apresentar classes ao usuário e capaz de integrar as classes apresentadas é a ferramenta *CodeGenie* (LEMOS et al., 2007).

Em relação as ferramentas que apresentam trechos de código, entre as *CSACTs* analisadas, as que extraem trechos de código automaticamente de alguma fonte de dados, como a *Web* (Subseção 1.2.3) e os projetos de código aberto (Subseção 1.2.2), não apresentam

funcionalidades que facilitam a adaptação do trecho de código no contexto de programação (HOFFMANN; FOGARTY; WELD, 2007; WANG et al., 2011; BAJRACHARYA et al., 2006). Mesmo quando tais ferramentas são integradas a uma *IDE* (ZHONG et al., 2009; HOLMES; WALKER; MURPHY, 2006; BHARDWAJ; LUCIANO; KLEMMER, 2011), o usuário necessita copiar o trecho de código apresentado pela ferramenta, colá-lo e adaptá-lo manualmente. A exceção é a ferramenta *Blueprint* (BRANDT et al., 2010), que automaticamente cola um trecho de código apresentado no código fonte em edição, no entanto também não apresenta funcionalidades para auxiliar a adaptação do trecho de código, como a renomeação de variáveis, a inclusão automática de dependências ou o auxílio na configuração de parâmetros.

Por sua vez, ferramentas baseadas em *templates de código* apresentam funcionalidades que facilitam a adaptação dos trechos de código. *Templates de código* são trechos de código cadastrados manualmente contendo diretivas de integração. As diretivas de integração são sentenças que informam a ferramenta como o trecho de código deverá ser adaptado no contexto do código fonte em edição ao ativar o *template* (após o usuário ter realizado uma busca pelo *template* desejado). Um cenário de uso dos *templates de código* na *IDE Eclipse* (ECLIPSE FOUNDATION, 2015) é apresentado na introdução deste trabalho de mestrado. A *IDE Eclipse* possui suporte nativo aos *templates de código*. Entre as principais diretivas de integração suportadas pelos *templates de código* do *Eclipse*, é possível citar as diretivas *newName*, *import* e *var*. A diretiva *newName* possibilita declarar uma variável no trecho de código definido no *template* e instruir o *Eclipse* a renomear a variável caso outra variável com o mesmo nome esteja presente no escopo do código fonte em edição durante a ativação do *template*. A diretiva *import* instrui o *Eclipse* a importar determinados pacotes e determinadas classes e a diretiva *var* declara um parâmetro para o *template*. Parâmetros de *templates* são substituídos por seus respectivos argumentos durante a ativação do *template* e tem como função reaproveitar o trecho de código em contexto diferentes. Um parâmetro de *template* pode ser associado a um tipo. A associação de um parâmetro de um *template* a um determinado tipo significa que o parâmetro do *template* aceitará como argumento identificadores de objetos (variáveis locais ou atributos de classe) do tipo associado que estiverem disponíveis no escopo do código fonte em edição durante a ativação do *template*.

Wightman et al. (2012) apresenta uma ferramenta denominada *SnipMatch*. A ferramenta estende a ideia utilizada no *Eclipse* e propõe um sistema de busca de *templates de código* baseado em palavras-chaves, diferente da funcionalidade nativa do *Eclipse*, que busca os *templates de código* através somente de um nome. A ferramenta também propõe novas diretivas de integração. Uma dentre as diretivas de integração propostas é a diretiva *helper*, que possibilita declarar classes utilitárias para um *template de código*. Classes utilitárias são classes contendo métodos que oferecem alguma funcionalidade ao trecho de código definido pelo *template*. Na Figura 25, é apresentado um exemplo de *template de*

código da ferramenta, que declara uma classe utilitária de nome *FileLineReader*, declarada na linha *8* e utilizada da linha *26*. A classe utilitária é delimitada pelas diretivas de integração *helper*, na linha *7*, e *endHelper*, na linha *24*. Durante a ativação do *template de código*, a ferramenta adiciona a classe utilitária no projeto do usuário, caso a classe ainda não tenha sido adicionada. Isso garante que o trecho gerado pelo *template*, que depende da classe utilitária, funcione corretamente.

Figura 25 – Exemplo de *template* da ferramenta *SnipMatch*

```
1  ${import(java.io.File)}
2  ${import(java.io.BufferedReader)}
3  ${import(java.io.FileReader)}
4  ${import(java.io.IOException)}
5  ${import(java.util.ArrayList)}
6
7  ${helper}
8  class FileLineReader {
9     public static String[] readAllLines(File file) {
10       ArrayList<String> lines = new ArrayList<String>();
11       try {
12          BufferedReader br = new BufferedReader(new FileReader(file));
13          String line = null;
14
15          while ((line = br.readLine()) != null) {
16             lines.add(line);
17          }
18          br.close();
19       }
20       catch (IOException e) { return null; }
21       return lines.toArray(new String[lines.size()]);
22    }
23 }
24 ${endHelper}
25
26 String[] ${freeName(lines)} = FileLineReader.readAllLines(${fileObject});
```

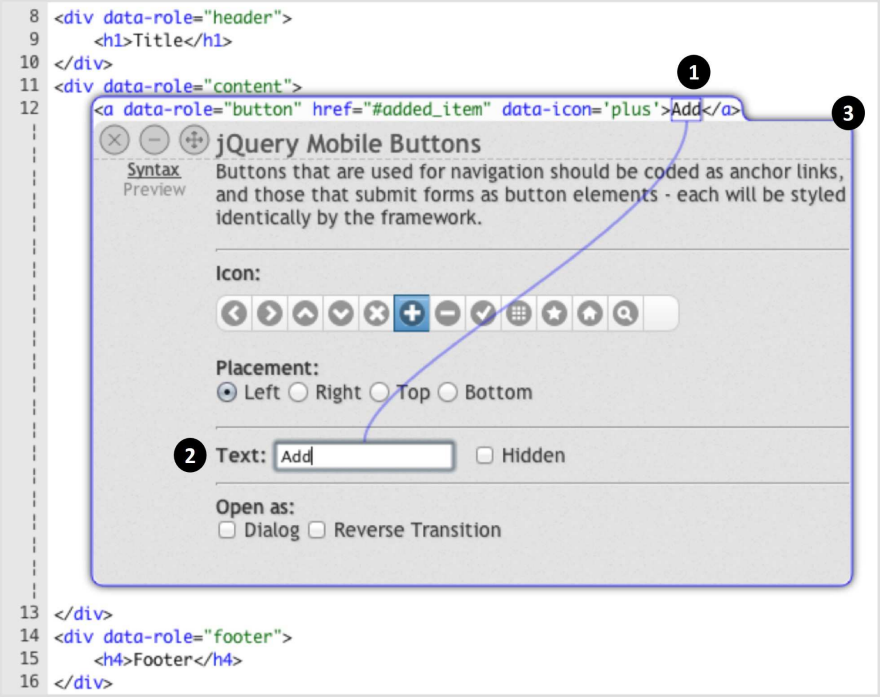
Fonte: adaptado de (WIGHTMAN et al., 2012, p. 223)

Oney e Brandt (2012) propõem a utilização de *templates de código* denominados *Codelets*. *Codelets* são *templates de código* associados a uma interface gráfica com componentes para editar os valores dos parâmetros do *template*. A interface gráfica é construída pelo próprio criador do *Codelet*. No trabalho dos autores, uma ferramenta com suporte aos *Codelets* é implementada e integrada à *IDE ACE*[18,19]. Os *Codelets* são associados a palavras-chaves e, durante a edição de um arquivo código fonte, o usuário pode procurar, através das palavras-chaves, por um *Codelet* desejado e acioná-lo. Ao acionar o *Codelet*, o editor gráfico de parâmetros associado é apresentado ao usuário para

[18] *ACE - **A**jax.org **C**loud9 **E**ditor*, *IDE* para desenvolvimento de arquivos *JavaScript*, *HTML* e *CSS*.
[19] <https://ace.c9.io/>

que ele possa configurar os valores dos parâmetros do *Codelet*. De acordo com os valores configurados, o trecho de código é gerado e colado no código fonte em edição. O editor gráfico de parâmetros associado ao *Codelet* continua associado ao trecho de código gerado pelo *Codelet*, ou seja, mesmo após a ativação do *Codelet* e após o trecho de código ter sido já colado em um arquivo, o usuário pode retornar ao trecho de código a fim de alterar os valores dos parâmetros através do editor gráfico de parâmetros. Na Figura 26, é apresentado um cenário de uso dos *Codelets* durante a edição de um arquivo *HTML*. A linha 12 é um trecho de código gerado por um *Codelet*. O trecho contém uma região cujo conteúdo depende do valor de um parâmetro cujo nome é *Text*, região com o conteúdo *"Add"* (item *1*). O usuário pode acionar o editor gráfico de parâmetros (item *3*) a fim de alterar o valor do parâmetro. Se o valor do parâmetro é alterado no editor gráfico de parâmetros, o conteúdo da região do trecho de código que depende do valor do parâmetro é automaticamente alterado.

Figura 26 – Utilização dos *Codelets*



Fonte: adaptado de (ONEY; BRANDT, 2012, p. 2697)

1.6 Conclusão

Neste Capítulo foi discorrido sobre as características das ferramentas apresentadas nos trabalhos analisados. As características foram abordadas em relação a cinco aspectos: o *contexto de uso*, que é o ambiente no qual a ferramenta está inserida; a fonte de dados da qual a ferramenta extrai as informações a serem apresentadas ao usuário; a entrada

utilizada pela ferramenta para decidir quais elementos são relevantes ao usuário; o resultado, que pode apresentar trechos de códigos ou componentes de uma *API* ao usuário, e as facilidades oferecidas pelas ferramentas a fim de adaptar um elemento selecionado ao contexto do projeto do *software* em desenvolvimento. Para o *contexto de uso*, para a fonte de dados e para a entrada foram apresentadas para cada aspecto as diferentes estratégias adotadas pelas ferramentas. Cada estratégia possui vantagens e desvantagens, as quais estão sumarizadas nas Tabelas 4, 5 e 6.

Em relação ao resultado, foi possível identificar algumas práticas adotadas pelas ferramentas para facilitar o entendimento e a inspeção do resultado apresentado ao usuário. Entre tais práticas é possível citar: a utilização de representações gráficas a fim de tornar a apresentação do resultado mais intuitiva; a apresentação de informações que esclarecem o motivo pelo qual determinado elemento tenha sido retornado no resultado; a organização dos elementos apresentados no resultado em grupos de elementos similares e a o ordenação dos elementos apresentados no resultado através de algum critério de relevância.

Para a integração do elemento selecionado no contexto do projeto do *software* sendo desenvolvido, foi possível identificar que não são todas as ferramentas que apresentam facilidades para a adaptação do elemento no contexto de programação. Adaptações necessárias podem incluir a renomeação de variáveis, inclusão de dependências e a configuração de parâmetros. Ferramentas que apresentam componentes de uma *API* em geral apresentam facilidades para adaptar o componente ao contexto do usuário, como a inclusão de dependências necessárias para utilizar o componente e a sugestão de argumentos. Ferramentas que apresentam trechos de código e extraem esses trechos automaticamente de uma fonte de dados, como a partir de páginas *web* ou a partir de projetos de código aberto disponíveis na *Internet*, não possuem facilidades para adaptar os trechos de código apresentados. Por outro lado, ferramentas que trabalham com *templates de código*, trechos de código cadastrados manualmente contendo diretivas de integração, apresentam funcionalidades para auxiliar o usuário, como a inclusão automática de dependência, a renomeação de variáveis já declaradas e a sugestão de argumentos para os parâmetros do *template*.

O objetivo desta análise bibliográfica foi compreender o desenvolvimento atual das ferramentas, posicionar a ferramenta *IPS*, desenvolvida neste trabalho de mestrado, em relação às outras ferramentas no que tange as similaridades e os diferenciais funcionais e derivar possíveis trabalhos futuros na área. A ferramenta *IPS* será detalhada no próximo Capítulo e as características da ferramenta *IPS* em relação às outras ferramentas serão explanadas na Seção sobre trabalhos relacionados (Seção 2.4). Possíveis trabalhos futuros na área serão discutidos na conclusão desta dissertação.

Tabela 4 – *Contexto de uso*: vantagens e desvantagens

Estratégia	**Vantagens**	**Desvantagens**
Integrada a uma *IDE*	Pode explorar o contexto de programação da *IDE* no qual o usuário está desenvolvendo um software para extrair informações que servem como entrada. Interface de interação mais próxima da tarefa que o programador esteja trabalhando no momento, evitando a troca de contexto. As ações do usuário são realizadas dentro da própria *IDE*.	Dependência arquitetural com a *IDE*. Experiência de interação proporcionado para o usuário limitado aos recursos de interação com o usuário disponíveis na *IDE*. Não possui um ambiente tão rico de interação com o usuário como em um navegador *web*.
Ferramenta *web*	Independência com a *IDE*. Pode se aproveitar de todo um ambiente rico de interação proporcionado pelos navegadores *web*.	Necessidade da troca de contexto. O usuário necessita abrir o navegador *web* (um ambiente diferente da *IDE*) para realizar as ações na ferramenta.

Fonte: próprio autor

Tabela 5 – Entrada: vantagens e desvantagens

Estratégia	**Vantagens**	**Desvantagens**
Palavras-chaves	Um tipo de entrada simples e intuitivo. Não há a necessidade do usuário possuir informações detalhadas, como informações estruturais, do elemento desejado de forma antecipada. O usuário precisa apenas conhecer algum termo que semanticamente esteja relacionado ao elemento desejado.	Ignora o potencial de se utilizar critérios mais detalhados e mais próximos do domínio considerado, no caso do domínio de desenvolvimento de *software*.
Informações específicas do domínio	Utiliza critérios mais detalhados e mais próximos do domínio considerado, no caso do domínio de desenvolvimento de software, logo há uma tendência de produzir resultados mais precisos.	Há uma tendência da *query* ser longa e complexa ou exigir que o usuário possua informações detalhadas de forma antecipada, como características estruturais do elemento desejado.
Informações no contexto de programação	Poupa o usuário de formular *queries* longas e complexas. Utiliza informações presentes no contexto de programação na *IDE* para automaticamente formular *queries* com informações específicas do domínio de desenvolvimento de *software*.	Como as informações são extraídas do contexto de programação na *IDE* utilizada pelo usuário, ferramentas que utilizam essa estratégia necessariamente são dependentes da arquitetura da *IDE*.

Fonte: próprio autor

Tabela 6 – Fonte de dados: vantagens e desvantagens

Estratégia	**Vantagens**	**Desvantagens**
Próprio projeto de *software*	Simplicidade. Não necessita lidar com grande quantidade de informação e sempre está disponível.	Quantidade de informações limitada. Pode não haver informações suficientes somente no projeto do software em desenvolvimento.
Outros projetos de *software*	Exploram a concepção que os códigos fontes presente em projetos de aplicações já existente apresentam a utilização correta das *APIs* de interesse, visto que o resultado do projeto é um *software* completo e funcional. Podem explorar a grande quantidade de projetos de código aberto presentes na *Internet*.	Explorar projetos de código aberto pela *Internet* apresenta desafios devido a falta de um padrão na distribuição os códigos fontes presentes nesses projetos. Cada repositório de projetos de código aberto utiliza um controlador de versão diferente e um protocolo de *download* diferente.
Páginas *web*	Páginas *web* contém textos explicativos que contribuem para o entendimento. Os termos presentes em uma página *web* contendo um trecho de código colabora com a eficiência da busca ao permitir que o usuário busque pelo trecho de código utilizando um termo presente na página. Trechos de código contido em páginas *web* geralmente são cuidadosamente confeccionados por um especialista da *API* com a intenção de demonstrar o uso da *API*, logo há uma tendência de serem mais concisos.	Para extrair os trechos de código das páginas *web* são utilizados métodos baseados em heurísticas. Não há uma garantia absoluta que sempre um trecho de código prático e útil será extraído da página *web*. É uma fonte bastante extensa e diversificada, o que pode contribuir para produzir falsos positivos (segmentos de textos classificados como trechos de código quando na realidade não são).
Testes unitários	O conjunto de testes unitários do próprio projeto de desenvolvimento da *API* de interesse é um conjunto conciso e relevante a respeito da *API*. Eles apresentam não somente os cenários de uso das classes da *API*, mas também as melhores práticas para utilizar as classes, pois são normalmente confeccionados pela mesma organização que projetou e implementou as classes. Nem sempre há códigos fontes disponíveis na *Web* de projetos que utilizem a *API* de interesse. Por exemplo, há casos que a *API* de interesse foi recentemente publicada ou os projetos de software que utilizam a *API* podem ser projetos privados de empresas que não possuem o código aberto por interesses comerciais. Utilizar os testes unitários do próprio projeto de desenvolvimento da *API* de interesse evita essa situação.	Um teste unitário pode apresentar mais de um cenário de uso de uma classe e conter sentenças que são somente relevantes no contexto do teste unitário. Há muitos desafios que envolve a separação de cenários de uso e remoção de sentenças irrelevantes a fim de extrair trechos de código a partir dos teste unitários para servirem de exemplo de utilização de uma *API*. Métodos baseado em heurísticas para lidar com esses desafios não possui uma garantia absoluta que sempre um trecho de código prático e útil será extraído de um teste unitário.
Trechos de código cadastrados manualmente	São confeccionados com o objetivo exclusivo de de ensinar e facilitar a utilização de uma *API* e, portanto, há uma tendência de possuir uma qualidade melhor e serem mais concisos. Não possuem os riscos associados a extração automática de uma fonte de dados. Em trechos de código extraídos de páginas *web*, o algoritmo de extração se baseia em heurísticas. Não há uma garantia que um trecho de código prático e útil será extraído da página *web*, o mesmo risco existe ao extrair trechos de código a partir dos testes unitários. Ferramentas que trabalham com trechos de código cadastrados manualmente, como os *templates de código*, apresentam funcionalidades que auxiliam o usuário a adaptar o trecho de código selecionado, após o usuário ter realizado uma busca pelo trecho de código desejado.	Há um custo tanto na criação como na manutenção dos trechos de código. Devido ao custo de criação e manutenção, o conjunto de trechos de código cadastrados manualmente possuem uma tendência de serem incompletos. Ou seja, para uma determinada *API* há o risco de não haver trechos de código confeccionados para aquela *API* ou o conjunto de trechos de código confeccionados para uma *API* pode não cobrir todos os cenários de uso da *API*.

Fonte: próprio autor

2 Solução proposta

A solução proposta nesta dissertação é a ferramenta *IPS — Intuitive Programming System*. O *IPS* é uma ferramenta *CSACT* disponibilizada como um plug-in da *IDE Eclipse* e trabalha com *templates de código*. Ferramentas *CSACTs* são abordadas no Capítulo 1 desta dissertação e os *templates de código* são trechos de código cadastrados manualmente contendo diretivas de integração (abordados nas Seções 1.2.5 e 1.5).

Com o objetivo de apresentar as funcionalidades do *IPS*, na próxima Seção (Seção 2.1) é apresentado um cenário de uso hipotético da ferramenta. Algumas funcionalidades não apresentadas no cenário de uso são apresentadas na Seção 2.2. Na Seção 2.3 são apresentados a arquitetura da ferramenta e os detalhes de implementação. A fim de esclarecer as novidades do *IPS* em relação às outras ferramentas, um comparativo é apresentado na Seção 2.4.

2.1 Cenário de uso

1. O cenário de uso envolve dois atores: *DataAPIOrg* e *John*. *DataAPIOrg* é uma organização hipotética responsável pela criação, manutenção e distribuição de uma *API* com o nome *SQLDataAPI* cujo objetivo é fornecer uma interface para os usuários (programadores) acessarem informações de banco de dados relacionais. *John* é um programador usuário da *API SQLDataAPI* que está desenvolvendo um software que realiza acessos a banco de dados relacionais.

2. A organização *DataAPIOrg* distribui a biblioteca *SQLDataAPI* através de um site disponível na Internet. É uma biblioteca para plataforma *Java* empacotada em um arquivo cujo nome é *SQLDataAPI.jar*. Neste arquivo, estão empacotados os arquivos com extensão *class*, os quais possuem os *bytecodes* das classes *Java* que fazem parte da *API*.

3. Com o objetivo de catalogar os cenários de uso da *API SQLDataAPI*, no arquivo *SQLDataAPI.jar*, junto com os arquivos com a extensão *class*, é empacotado também um conjunto de arquivos com a extensão *xcspt*. Cada arquivo com extensão *xcspt* é um arquivo *XML* que define um conjunto de *templates de código*, os quais são denominados no *IPS* de *code snippets*. Detalhes da estrutura de um arquivo *XCSPT* são apresentados na Seção 2.3.2.

4. Os *code snippets* servem como um recurso de aprendizagem da utilização da *API* e são confeccionados pelos próprios desenvolvedores da *API*. Como os desenvolvedores

da *API* possuem o conhecimento de como a *API* foi projetada, eles são as pessoas mais capacitadas para demostrarem como a *API* deve ser utilizada, ou seja, como as classes utilizadoras da *API* devem colaborar com as classes pertencentes à *API* (ROBILLARD, 2009, p. 32).

5. Um *code snippet* tem: um nome, uma descrição, palavras-chaves associadas, uma lista de *imports*, uma lista de parâmetros e um corpo. O nome é utilizado para identificar um *code snippet*. A descrição é um texto breve descrevendo o objetivo do *code snippet*. As palavras-chaves permitem ao usuário realizar buscas não necessitando lembrar o nome exato do *code snippet* desejado. As palavras-chaves podem ser palavras que transmitem a ideia de comando. Exemplo, para um *code snippet* com o objetivo de conectar em uma base *MySQL* as seguintes palavras-chaves podem ser associadas: "*connect to MySQL database*". O corpo do *code snippet* define o trecho de código *Java* que é gerado durante o acionamento do *code snippet*. A lista de *imports* é uma lista de identificadores de classes *Java* que devem ser importadas ao acionar o *code snippet* para que o trecho de código *Java* gerado pelo *code snippet* funcione corretamente. Os parâmetros permitem o reaproveitamento do corpo em contextos diferentes, eles podem ser referenciados no corpo e, durante o acionamento do *code snippet*, cada referência do parâmetro no corpo do *code snippet* é substituído pelo seu respectivo argumento.

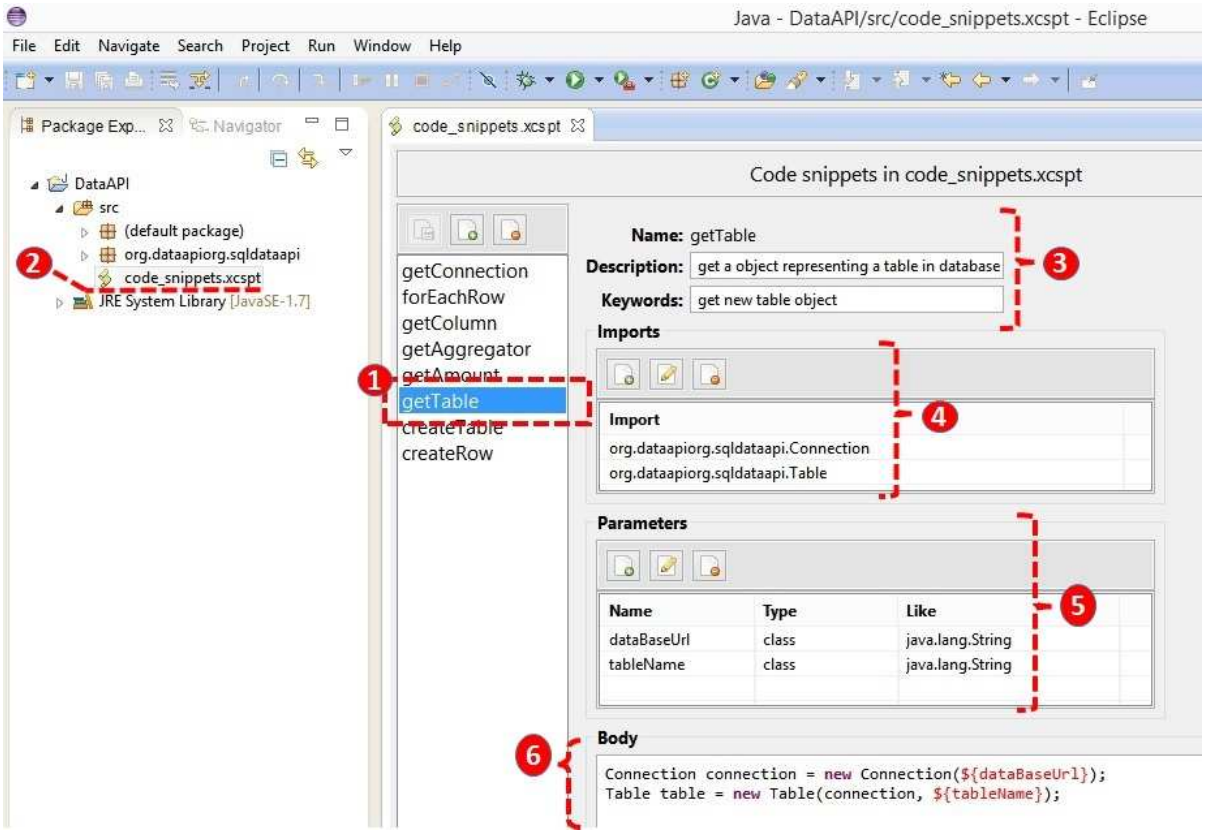
6. A fim de ilustrar a utilização dos *code snippets* através da ferramenta *IPS*, entre os *code snippets* empacotados na *API* hipotética *SQLDataAPI*, vamos tomar como exemplo o *code snippet* de nome *getTable* (Figura 27). *getTable* é um *code snippet* cujo objetivo é criar uma instância de um objeto do tipo da classe *Table*, classe da *API SQLDataAPI* do pacote *org.dataapiorg.sqldataapi* que representa uma tabela em um banco de dados relacional.

7. Na Figura 27, é ilustrado o *code snippet getTable* (item 1), que é declarado no arquivo de nome *code_snippets.xcspt* (item 2). Um *code snippet* é definido em um arquivo *XCSPT* através de uma interface gráfica que é utilizada pelo desenvolvedor da *API* para editar a descrição e as palavras-chaves (item 3), a lista de *imports* (item 4), a lista de parâmetros (item 5) e o corpo (item 6).

8. *John*, um programador usuário da *API SQLDataAPI*, realiza o *download* do arquivo *SQLDataAPI.jar*, que contém os *bytecodes* das classes *Java* que fazem parte da *API* (arquivos *class*) junto com os arquivos *XCSPT*, que contém os *code snippets* que auxiliam a utilização da *API*.

9. *John* adiciona o arquivo *SQLDataAPI.jar* (item 1 destacado na Figura 28) como uma referência em um projeto *Java* na *IDE Eclipse* (projeto de nome *SQLDataAPIClient* destacado como item 2 na Figura 28). Assim que *John* referencia o

Figura 27 – *Code snippet getTable*



Fonte: próprio autor

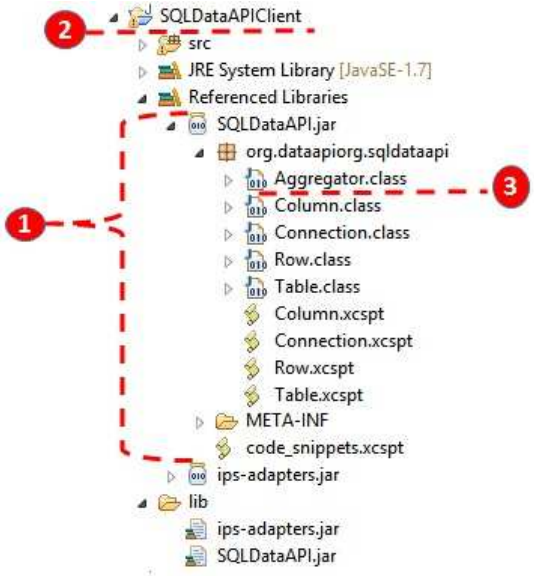
arquivo *SQLDataAPI.jar* no projeto *SQLDataAPIClient*, os *code snippets* definidos nos arquivos *XCSPT* presentes no arquivo *SQLDataAPI.jar* são carregados pela ferramenta *IPS* e ficam disponíveis para uso no ambiente de programação para o projeto *SQLDataAPIClient*.

10. Durante o desenvolvimento do projeto *SQLDataAPIClient*, *John* se depara com uma tarefa que ele necessita calcular a média aritmética dos valores presentes na coluna *grade* de uma tabela relacional de nome *Student* em um banco de dados. Para isso, ele inicia a implementação de um método com o nome *getMeanGrade*, que deverá retornar a média aritmética dos valores na coluna *grade*. Mentalmente, *John* constrói um modelo e divide a tarefa em alguns passos (Figura 29).

11. Para o primeiro passo, *John* precisa escrever um trecho de código para obter uma instância de um objeto representando a tabela *Student* de um banco de dados relacional. Para isso, ele, com o intuito de procurar por um *code snippet* que possa auxiliá-lo na tarefa, aciona o *IPS* digitando o caractere # em uma linha vazia no código fonte (item 1 da Figura 30).

12. O *IPS* apresenta uma campo de texto para que *John* possa digitar palavras-chaves

Figura 28 – Referenciando a biblioteca *SQLDataAPI*



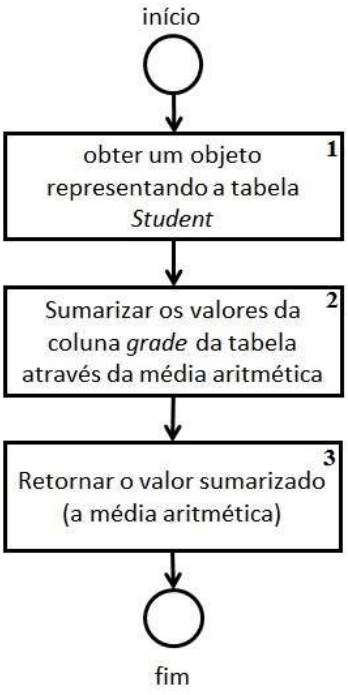
Fonte: próprio autor

relacionadas ao *code snippet* desejado. Ele digita as palavras-chaves *"get table object"* (item 2 da Figura 30) e o *IPS* retorna os *code snippets* disponíveis no ambiente de programação que sejam associados a pelo menos uma palavra-chave entre as palavras-chaves digitadas na área de texto (item 3 da Figura 30). Os *code snippets* disponíveis no ambiente de programação são os *code snippets* declarados nos arquivos *XCSPT* presentes nas bibliotecas *Java* (arquivos com a extensão *jar*) referenciados no projeto *Java* na *IDE Eclipse* (veja Figura 28). Para cada *code snippet* retornado é apresentado o nome do *code snippet* e as palavras-chaves associadas. Os *code snippets* são listados em ordem de relevância e são apresentados no máximo dez *code snippets* ao usuário. Os *code snippets* que são associados a mais palavras-chaves entre as palavras-chaves digitadas são considerados mais relevantes.

13. Através do nome e das palavras-chaves de cada *code snippet* apresentado, *John* investiga se algum dos *code snippets* é o *code snippet* desejado. *John* julga que o primeiro *code snippet* retornado é provavelmente o *code snippet* desejado, portanto, ele seleciona a primeira linha da listagem (item 4 da Figura 30). O *IPS* apresenta uma área (item 5 da Figura 30) que exibe as informações do *code snippet* selecionado: o nome, a descrição, as palavras-chaves, a lista de parâmetros e o corpo. Para cada parâmetro, uma lista de possíveis argumentos é apresentada para que *John* selecione o argumento desejado (item 6 na Figura 30).

14. Os argumentos recomendados para os parâmetros são variáveis presentes no escopo do código fonte sendo editado. Por exemplo, o parâmetro *dataBaseUrl* do *code snippet getTable* (item 6 na Figura 30) aceita argumentos do tipo da classe

Figura 29 – Tarefa a ser realizada por *John*

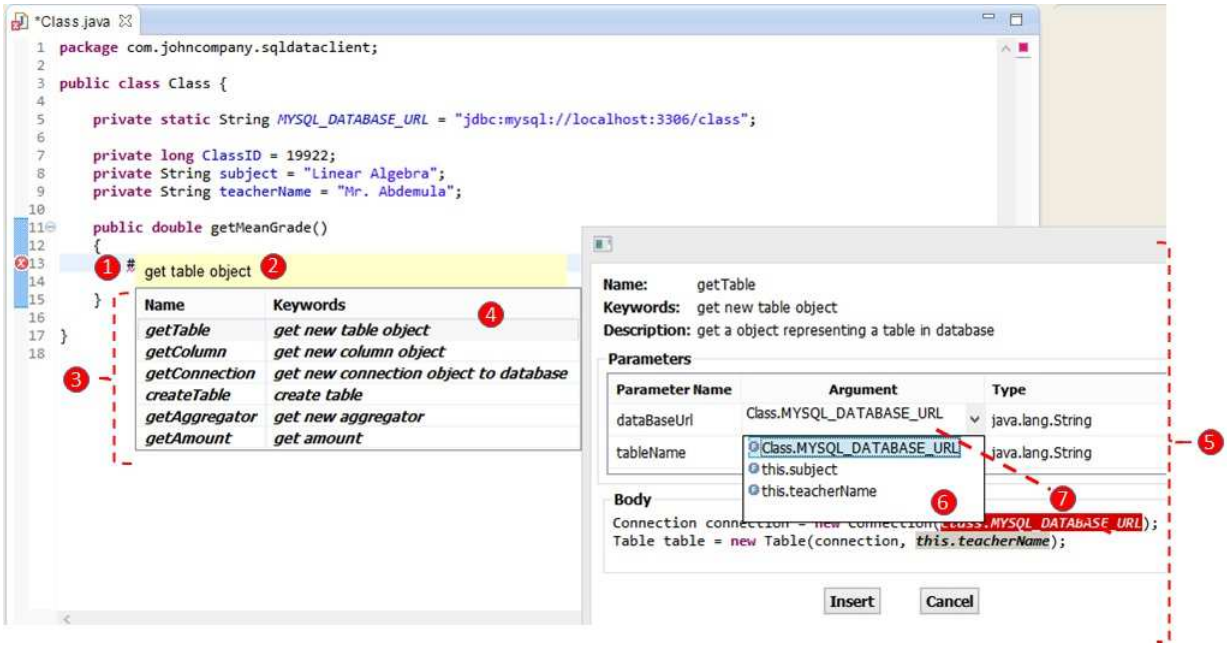


Fonte: próprio autor

java.lang.String e, portanto, são recomendados para o parâmetro os argumentos *MYSQL_DATABASE_URL* (campo de classe declarado na linha 5 do arquivo *Class.java*), *subject* (variável de instância declarada na linha 8) e *teachName* (variável de instância declarada na linha 9). As sugestões de argumentos apresentadas são ordenadas pela similaridade entre o nome da variável e o nome do parâmetro e, por padrão, o primeiro argumento sugerido é automaticamente selecionado para o parâmetro. No exemplo apresentado, o argumento *MYSQL_DATABASE_URL* é o primeiro da lista (veja item 6 na Figura 30) e, caso nenhum outro argumento seja selecionado, ele é automaticamente selecionado para o parâmetro *dataBaseUrl*, pois, dentre os argumentos sugeridos, *MYSQL_DATABASE_URL* é o que possui o nome mais similar a *dataBaseUrl*. A similaridade entre os nomes é calculada utilizando a distância de *Levenshtein* (GILLELAND; SOFTWARE, 2016).

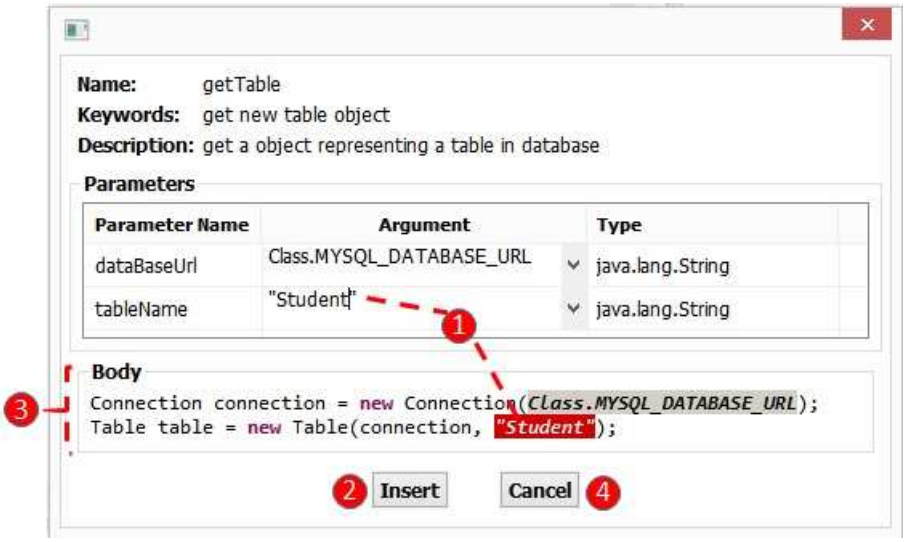
15. Enquanto *John* está selecionando algum argumento para um parâmetro e o componente gráfico contendo os argumentos está com o foco, as referências do parâmetro no corpo do *code snippet* são destacadas com um fundo na cor vermelha para que *John* visualize a localização das referências para o parâmetro no corpo do *code snippet* (veja item 7 na Figura 30). Além disso, conforme algum argumento é selecionado, o corpo do *code snippet* é automaticamente atualizado substituindo todas as referências do parâmetro no corpo do *code snippet* pelo argumento selecionado.

Figura 30 – Utilizando o *IPS*



Fonte: próprio autor

Figura 31 – Setando parâmetros do *code snippet*

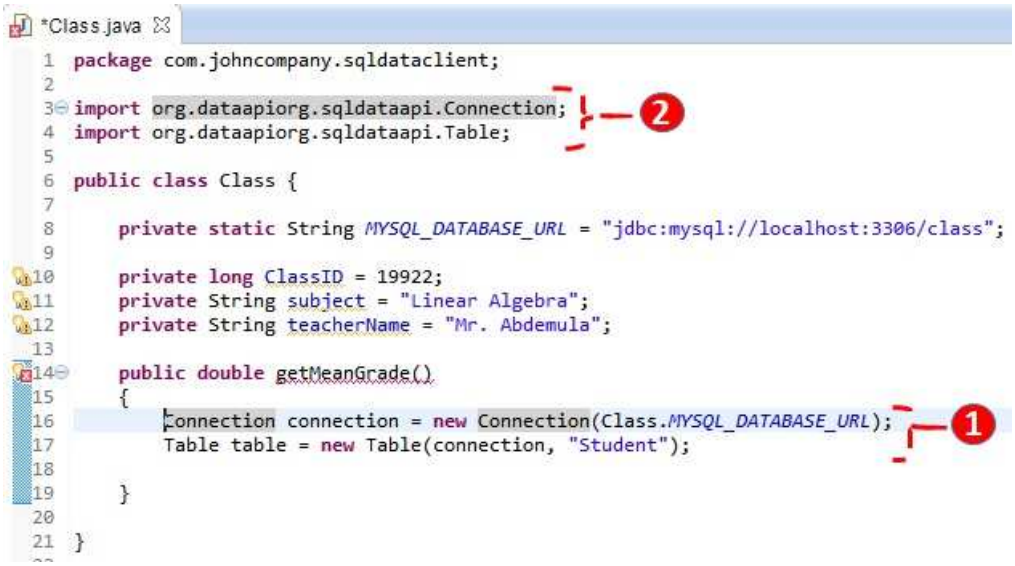


Fonte: próprio autor

16. Um argumento de um parâmetro pode ser, além de um variável no escopo do código fonte em edição, uma literal. No exemplo, para o parâmetro *tableName*, *John*, ao invés de escolher umas das variáveis do tipo *java.lang.String* disponíveis no escopo do código fonte em edição, digita a literal *"Student"* como argumento (item 1 destacado na Figura Figura 31).

17. Após *John* escolher os argumentos para cada parâmetro do *code snippet*, o botão *Insert* (item 2 destacado na Figura 31) é acionado e, nesse caso, o corpo do *code snippet* (item 3 destacado na Figura 31) substitui a linha de acionamento do *IPS* (a linha 13 do arquivo *Class.java* apresentado na Figura 30 é substituída pelo trecho de código compreendendo a linha 16 e a linha 17 apresentadas na Figura 32). Além disso, é inserido no código fonte os *imports* que o trecho de código inserido depende (item 2 da Figura 32). *John* poderia ainda, caso não quisesse acionar o *code snippet* selecionado, selecionar outro *code snippet* dentre os *code snippets* apresentados na listagem (item 4 destacado na Figura 30) ou ele poderia cancelar a busca e fechar o *IPS* pressionando o botão *Cancel* (item 4 da Figura 32).

Figura 32 – *Code snippet* inserido

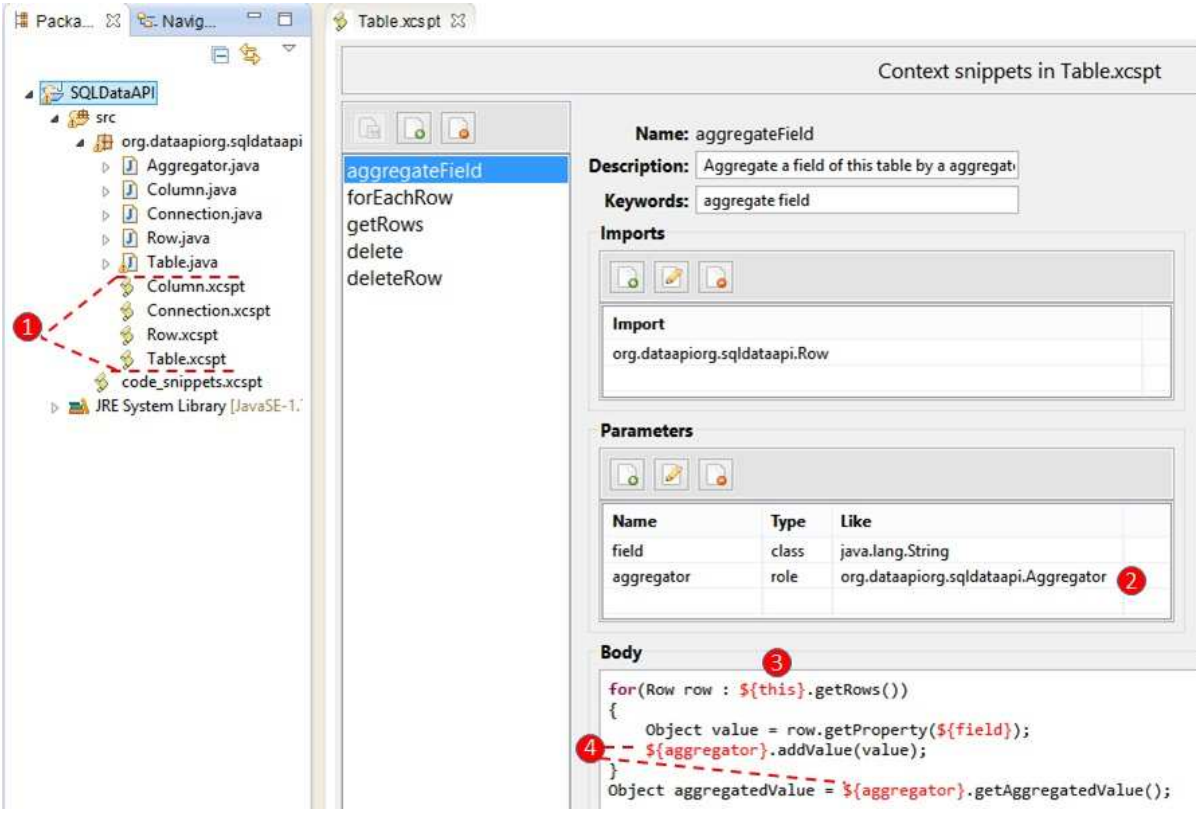


Fonte: próprio autor

18. A fim de ilustrar outras funcionalidades do *IPS* e dar continuidade a apresentação do cenário de uso, vamos tomar como exemplo um outro *code snippet*, cujo nome é *aggregateField* (veja Figura 33) e que tem o objetivo de agregar os valores de uma coluna de uma tabela relacional. O *code snippet aggregateField* é associado à classe *Table* do pacote *org.dataapiorg.sqldataapi*. No *IPS*, um *code snippet* associado a uma classe *Java* pode ser acionado para um identificador de objeto (variável local ou atributo de classe) do tipo da classe associada. O usuário aciona o *IPS* para um identificador de objeto, pois ele deseja realizar alguma operação com o objeto. Neste caso, ao apresentar os *code snippets* ao usuário, o *IPS* somente considera os *code snippets* associados à classe do objeto. Para associar um *code snippet* a uma classe *Java*, os *code snippets* devem estar definidos em um arquivo com a extensão *xcspt* com o mesmo nome e no mesmo pacote da classe associada. Por exemplo, o *code snippet aggregateField* é associado à classe *Table* do pacote *org.dataapiorg.sqldataapi*,

pois ele é definido no arquivo *Table.xcspt* que se encontra no mesmo pacote da classe *Table* (veja item 1 destacado na Figura 33).

Figura 33 – *Code snippet aggregateField*



Fonte: próprio autor

Figura 34 – *Adapter Aggregator*

```
1 package org.dataapi.adapters;
2
3 import org.ufscar.ppgccs.ips.adapters.RoleAdapter;
4
5 public interface Aggregator extends RoleAdapter
6 {
7     public void addValue(Object value);
8     public Object getAggregatedValue();
9 }
```

Fonte: próprio autor

19. O *code snippet aggregateField* possui dois parâmetros: *field* e *aggregator* (veja item 2 destacado na Figura 33). O parâmetro *field* é do tipo *class*. Parâmetros do tipo *class* aceitam como argumento identificadores de objetos do tipo definido por uma classe ou uma interface *Java* associada ao parâmetro. A classe ou a interface *Java*

associada ao parâmetro é definida através do atributo *like* na definição do parâmetro. No exemplo, o parâmetro *field* é associado à classe *java.lang.String*, portanto, aceita como argumento identificadores de objetos (variável local ou atributo de classe) do tipo *java.lang.String*. Por outro lado, o parâmetro *aggregator* é do tipo *role*. Para os parâmetros do tipo *role*, a seleção dos possíveis argumentos do parâmetro é realizada através dos *adapters*. Um *adapter* é definido declarando-se uma interface *Java*. Para que uma interface *Java* seja considerada um *adapter*, como regra, deverá estender a interface *RoleAdapter* do pacote *br.ufscar.ppgccs.ips.adapters* (veja a Figura 34, que apresenta a definição de um *adapter* com o nome *Aggregator*). Um parâmetro do tipo *role* tem um *adapter* associado, que é definido através do atributo *like* na definição do parâmetro. No exemplo apresentado na Figura 33, o parâmetro *aggregator* é associado ao *adapter org.dataapiorg.sqldataapi.Aggregator* (veja item 2 destacado na Figura 33). É possível também associar uma classe *Java* a um *adapter*. A associação de uma classe *Java* a um determinado *adapter* é realizada através da *annotation ClassLike*.

Figura 35 – Classe *ArithmeticMeanAggregator*

```
1  package com.johncompany.sqldataclient;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import br.ufscar.ppgccs.ips.adapters.ClassLike;
7  import br.ufscar.ppgccs.ips.adapters.MethodMappedTo;
8
9  import org.dataapiorg.sqldataapi.Aggregator;
10
11 @ClassLike(Aggregator.class) (1)
12 public class ArithmeticMeanAggregator
13 {
14     List<Double> values = new ArrayList<>();
15
16     @MethodMappedTo("addValue(Object)")
17     public void insertValue(Object value) {
18         values.add((Double) value);
19     }
20
21     @MethodMappedTo("getAggregatedValue()")
22     public Object getMean(){
23         Double sum = 0D;
24         for(Object value : this.values)
25         {
26             sum += (Double) value;
27         }
28         return sum / 3D;
29     }
30 }
```

(2) (3)

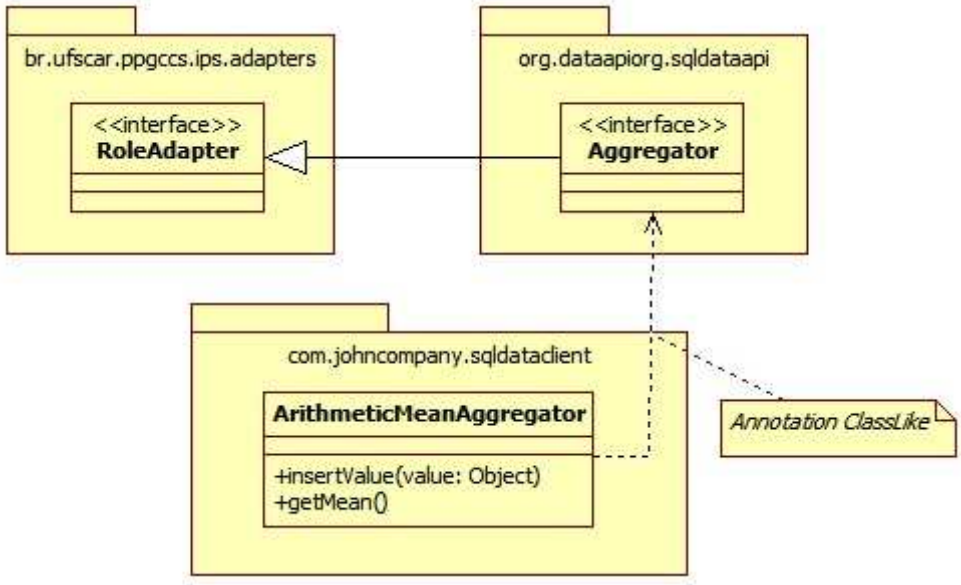
Fonte: próprio autor

20. Na Figura 35, é apresentada a classe *ArithmeticMeanAggregator*, que é associada ao *adapter Aggregator* (item 1) e, na Figura 36, é apresentado um diagrama *UML*[1] que

[1] *UML*, *Unified Modeling Language*, uma linguagem de modelagem de sistemas padronizada pela *OMG*

ilustra a relação entre a classe, o *adapter Aggregator* e a interface *RoleAdapter*. Se uma classe *X* é anotada através da *annotation ClassLike* tendo como parâmetro o *adapter Y*, significa que identificadores de objetos do tipo da classe *X* podem ser aceitos como argumentos de parâmetros do tipo *role* que sejam associados ao *adapter Y*. No exemplo, identificadores de objetos da classe *ArithmeticMeanAggregator* (Figura 35) podem ser utilizados como argumento do parâmetro *aggregator* (item 2 da Figura 33), pois a classe *ArithmeticMeanAggregator* é anotada através da *annotation ClassLike* tendo como parâmetro o *adapter Aggregator* e o parâmetro *aggregator* é associado ao *adapter Aggregator*.

Figura 36 – Diagrama de classe de *ArithmeticMeanAggregator*



Fonte: próprio autor

21. O *code snippet aggregateField* é confeccionado por um desenvolvedor da *API SQL-DataAPI* com o objetivo de disponibilizar um trecho de código que demonstre ao usuário da *API* como agregar os valores de uma coluna de uma tabela. No entanto, para tornar o *code snippet* mais genérico, o desenvolvedor do *code snippet* utiliza um parâmetro do tipo *role* (parâmetro *aggregator* no item 2 destacado na Figura 33). Isso possibilita que no corpo do *code snippet* sejam feitas referências para um objeto que desempenha um determinado papel, no entanto sem conhecer a classe *Java* do objeto. No caso, essas referências são referências para um parâmetro do tipo *role* (veja item 4 destacado na Figura 33). Durante a utilização do *code snippet*, as referências no corpo do *code snippet* para o parâmetro do tipo *role* são então adaptadas para um objeto escolhido como argumento do parâmetro pelo usuário do

<http://www.uml.org/>.

code snippet. A classe *Java* do objeto escolhido como argumento do parâmetro do tipo *role* será uma classe *Java* confeccionada pelo usuário do *code snippet*, portanto, uma classe desconhecida durante a confecção do *code snippet*.

22. O *adapter Aggregator* é uma interface *Java* empacotada e distribuída na *API SQLDataAPI* (arquivo *SQLDataAPI.jar*), junto com os *code snippets* e outras classes e interfaces *Java* que fazem parte da *API* (veja item 3 destacado na Figura 28).

23. Continuando o cenário de uso, a fim de completar o segundo passo do fluxograma apresentado na Figura 29, *John* precisa sumarizar os valores da coluna *grade* da tabela *Student* (representada pela variável *table* declarada na linha 17 do arquivo *Class.java* apresentado na Figura 32). A sumarização precisa ser realizada através da média aritmética dos valores da coluna. Para isso, *John* confecciona a classe *ArithmeticMeanAggregator* (Figura 35), uma classe responsável por calcular a média aritmética de um conjunto de valores e que possui dois métodos: um método com o nome *insertValue* (item 2 destacado na Figura 35), que adiciona um valor numérico a uma lista de valores, e outro método com o nome *getMean* (item 3 destacado na Figura 35), que obtém a média dos valores adicionados na lista. Para informar o *IPS* que a classe *ArithmeticMeanAggregator* é compatível com o *adapter Aggregator* (*adapter* apresentado na Figura 34), *John* anota a classe *ArithmeticMeanAggregator* com a *anottation ClassLike* utilizando como parâmetro a interface *Aggregator* do pacote *org.dataapiorg.sqldataapi* (item 1 da Figura 35). Além disso, *John* anota o método *insertValue* com a *anottation MethodMappedTo* utilizando o parâmetro *addValue(Object)*, isto informa que o método *insertValue* da classe *ArithmeticMeanAggregator* (item 2 destacado na Figura 35) é mapeado para o método *addValue* do *adapter Aggregator* (item 1 destacado na Figura 34). O método *getMean* é anotado com a *anottation MethodMappedTo* utilizando o parâmetro *getAggregatedValue*, isto informa que o método *getMean* da classe *ArithmeticMeanAggregator* (item 3 destacado na Figura 35) é mapeado para o método *getAggregatedValue* do *adapter Aggregator* (item 2 destacado na Figura 34).

24. *John* prossegue a implementação do método *getMeanGrade* da classe *Class*. Ele declara uma variável do tipo da classe *ArithmeticMeanAggregator* (linha 18 na Figura 37) e, a fim de completar o segundo passo do fluxograma apresentado na Figura 29, ele ativa o *IPS* para o identificador de objeto *table* (variável local declarada na linha 17 na Figura 37). Para isso, ele digita *table* seguido do caractere # em uma linha vazia do código fonte sendo editado (veja linha 19 na Figura 37). O *IPS* apresenta uma campo de texto para que *John* possa digitar palavras-chaves relacionadas ao *code snippet* desejado (item 1 da Figura 37). Ele digita as palavras-chaves *"aggregte fied"* e o *IPS* retorna os *code snippets* disponíveis no ambiente de programação que sejam associados à classe *Table* do pacote *org.dataapiorg.sqldataapi* e que sejam

associados a pelo menos uma palavra-chave entre as palavras-chaves digitadas na área de texto (item 2 da Figura 37). Na busca de *code snippets* somente são considerados os *code snippets* associados à classe *Table* do pacote *org.dataapiorg.sqldataapi*, pois a variável local *table*, utilizada para ativar o *IPS*, é do tipo *Table*. *John* cometeu erros de digitação e embora tenha digitado as palavras-chaves *"aggregte fied"* a intenção dele era ter digitado *"aggregate field"*. Porém, o *IPS* considera palavras-chaves com pequenos erros de digitação e, portanto, é retornado um *code snippet* associado às palavras-chaves *"aggregate field"*.

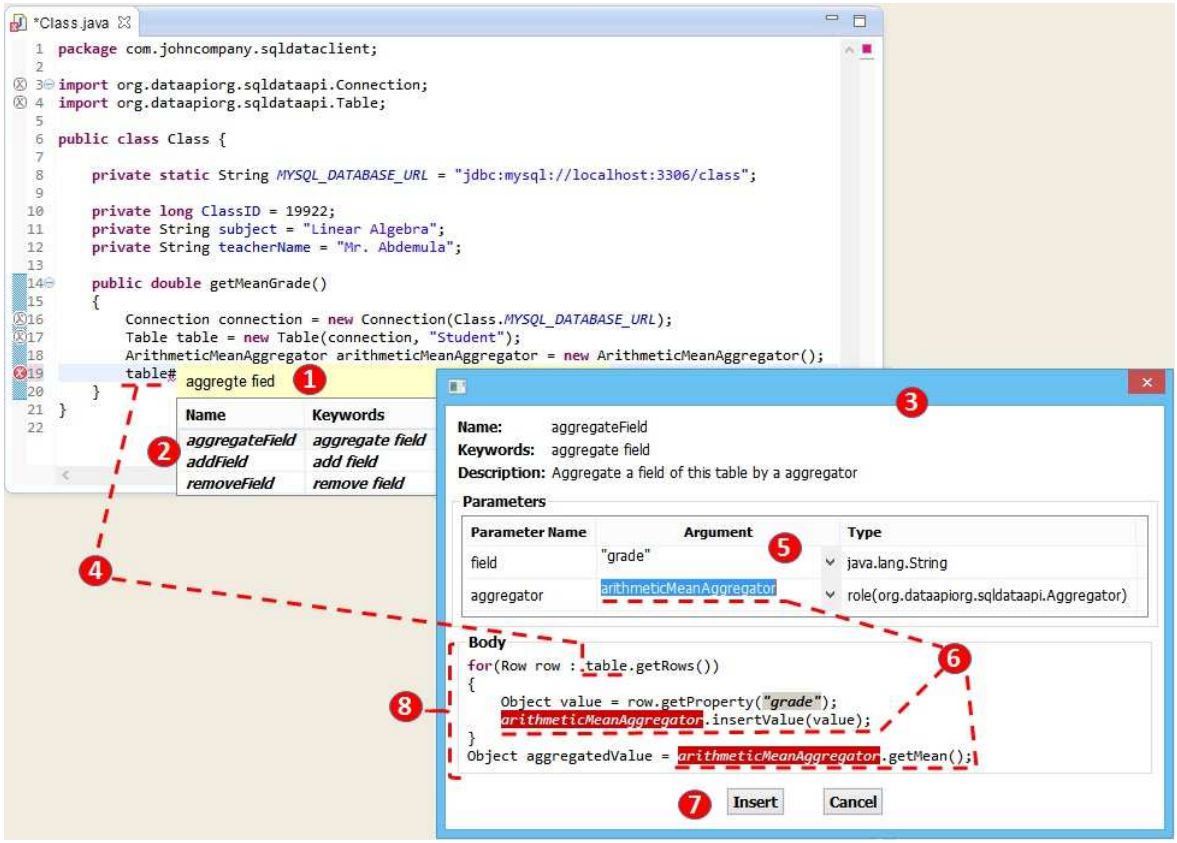
25. *John* seleciona o *code snippet* retornado de nome *aggregateField*, e o *IPS* apresenta uma área que exibe as informações do *code snippet* (item 3 na Figura 37). No corpo do *code snippet* a diretiva *${this}* utilizada na definição do *code snippet* (item 3 destacado na Figura 33) é substituída pelo identificador de objeto que ativa o *IPS*, no caso a variável local *table* (veja item 4 destacado na Figura 37).

26. *John* seleciona os argumentos para os parâmetros do *code snippet*. Para o parâmetro *field*, ele digita a literal *"grade"* (item 5 na Figura 37), isto informa que ele deseja agregar os valores da coluna de nome *grade* da tabela representada pelo objeto *table* . O parâmetro *aggregator* é um parâmetro do tipo *role* associado ao *adapter Aggregator* do pacote *org.dataapiorg.sqldataapi* e *John* seleciona a variável local *arithmeticMeanAggregator*, declarada na linha 18 do arquivo *Class.java* exibido na Figura 37, como argumento do parâmetro (item 6 na Figura 37). É possível escolher a variável *arithmeticMeanAggregator* como argumento do parâmetro *aggregator*, pois o parâmetro é associado ao *adapter Aggregator* do pacote *org.dataapiorg.sqldataapi* e a variável *arithmeticMeanAggregator* é do tipo da classe *ArithmeticMeanAggregator*, que é uma classe anotada com a *annotation ClassLike* tendo como parâmetro o *adapter Aggregator* (veja item 1 destacado na Figura 35).

27. Assim que *John* seleciona a variável *arithmeticMeanAggregator* como argumento do parâmetro *aggregator*, as referências para o parâmetro *aggregator* no corpo do *code snippet* são automaticamente adaptadas para a variável *arithmeticMeanAggregator* (item 6 na Figura 37). O método *addValue* do *adapter Aggregator* é substituído pelo método *insertValue* da classe *ArithmeticMeanAggregator* e o método *getAggregatedValue* do *adapter Aggregator* é substituído pelo método *getMean* da classe *ArithmeticMeanAggregator*, o que segue o mapeamento de métodos definido na classe *ArithmeticMeanAggregator* (veja itens 2 e 3 destacados na Figura 35).

28. Após *John* escolher os argumentos para cada parâmetro do *code snippet*, ele aciona o botão *Insert* (item 7 destacado na Figura 37) e o corpo do *code snippet* (item 8 destacado na Figura 37) substitui a linha de acionamento do *IPS* (a linha 19 do

Figura 37 – Ativando o *code snippet aggregateField*



Fonte: próprio autor

arquivo *Class.java* apresentado na Figura 37 é substituída pelo trecho de código da linha 20 até a linha 25 apresentado na Figura 38).

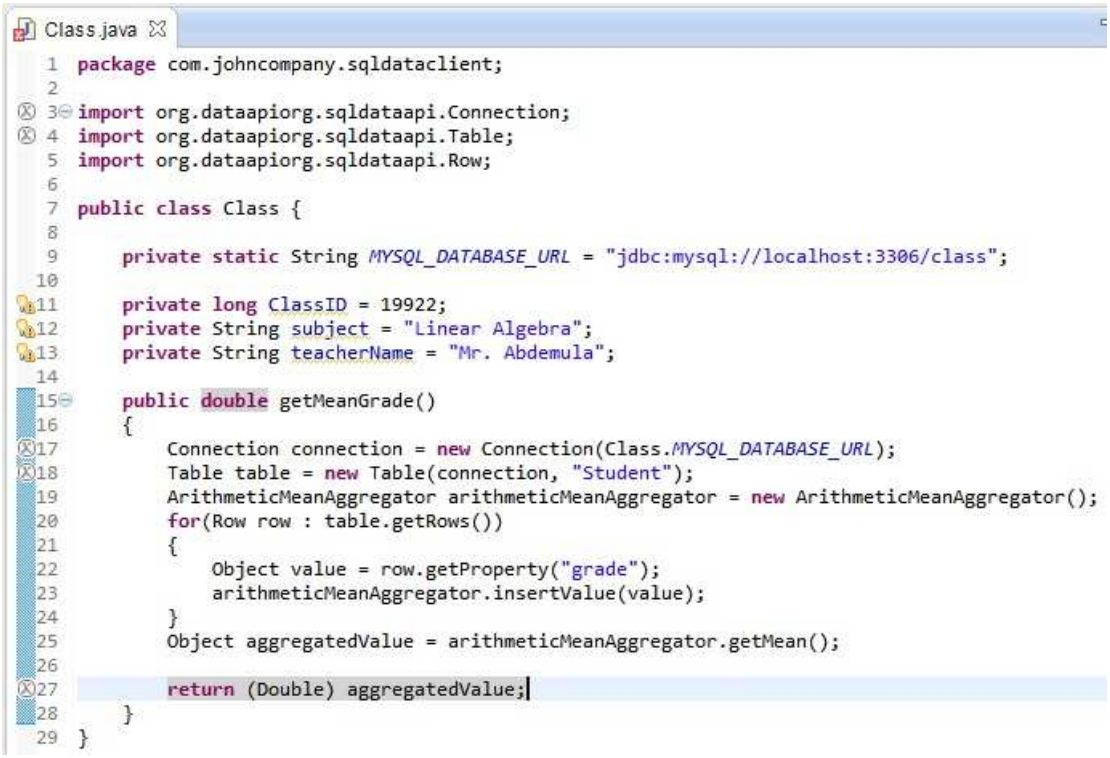
29. Para completar o terceiro passo do fluxograma apresentado na Figura 29 e finalizar a implementação do método *getMean*, *John* retorna o objeto *aggregatedValue* (linha 27 do arquivo *Class.java* apresentado na Figura 38).

2.2 Funcionalidades adicionais

2.2.1 *Recomendador de code snippets*

O *recomendador de code snippets* é uma funcionalidade do *IPS* que recomenda *code snippets* ao usuário enquanto ele está editando algum código fonte na *IDE Eclipse*, sem a necessidade de uma ação explícita. O *recomendador* utiliza informações do contexto do código fonte em edição a fim de recomendar uma lista de *code snippets* possivelmente úteis para o usuário. A lista de *code snippets* recomendados é constantemente atualizada conforme o conteúdo do código fonte em edição é alterado.

Figura 38 – *Code snippet aggregateField* após ativação

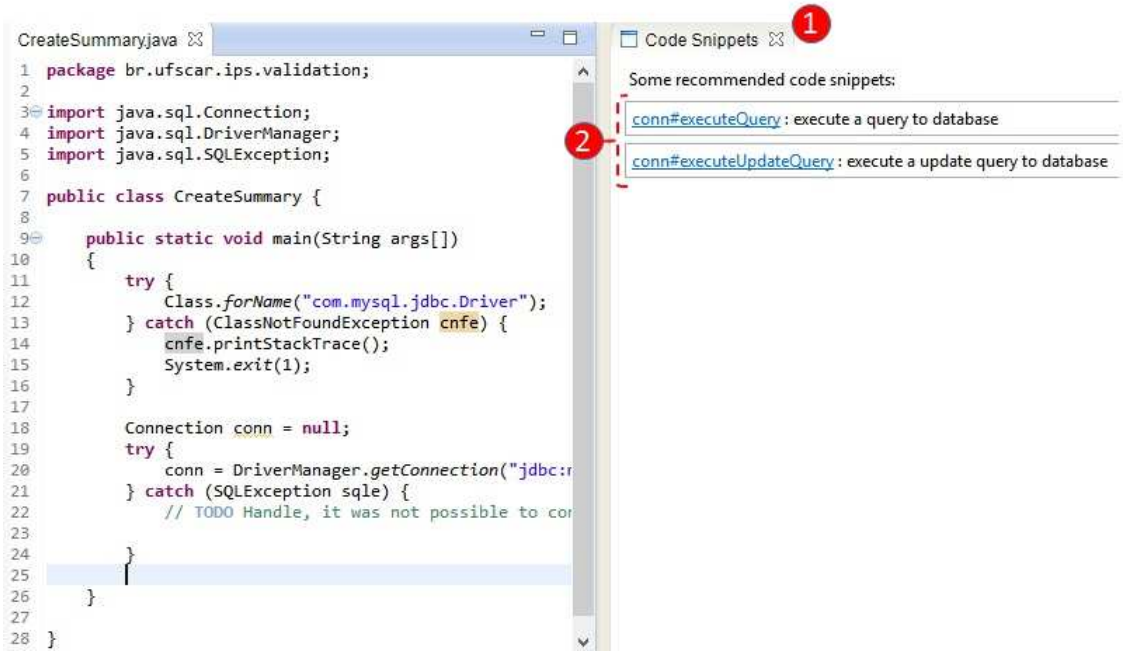


Fonte: próprio autor

Na Figura 39, como exemplo, é apresentado um cenário de uso onde o usuário (um programador) está editando o conteúdo de um código fonte *Java*. Após o usuário ter digitado um trecho de código (trecho de código compreendido entre as linhas 11 e 24 do código apresentado na Figura 39), o cursor de texto encontra-se na linha 25. O *recomendador de code snippets*, que é uma aba dentro da *IDE Eclipse* que pode ser posicionada ao lado do editor de códigos fontes *Java* (item 1 destacado na Figura 39), recomenda a ativação de dois *code snippets*: o *executeQuery* e o *executeUpdateQuery* (veja item 2 destacado na Figura 39). Ambos os *code snippets* são associados à classe *java.sql.Connection*. Como a variável *conn*, declarada na linha 18, é do tipo da classe *java.sql.Connection* e ela é acessível na linha 25 (onde se encontra o cursor de texto), são recomendados a ativação desses dois *code snippets* para a variável *conn*. As recomendações de ativação dos *code snippets* são para identificadores de objetos (variáveis locais ou atributos de classe). Por exemplo, a recomendação *conn#executeQuery* está recomendando a ativação do *code snippet* de nome *executeQuery* para a variável local *conn*. À frente de cada recomendação é exibida a descrição do *code snippet*.

O usuário percebe que na linha 23 (veja código fonte exibido na Figura 40) há a necessidade de tratar a exceção do tipo *SQLException* através da variável *sqle*. Portanto, ele posiciona o cursor de texto na linha 23 e, a fim de refletir a nova posição do cursor, o

Figura 39 – *Recomendador*: passo 1 do cenário de uso

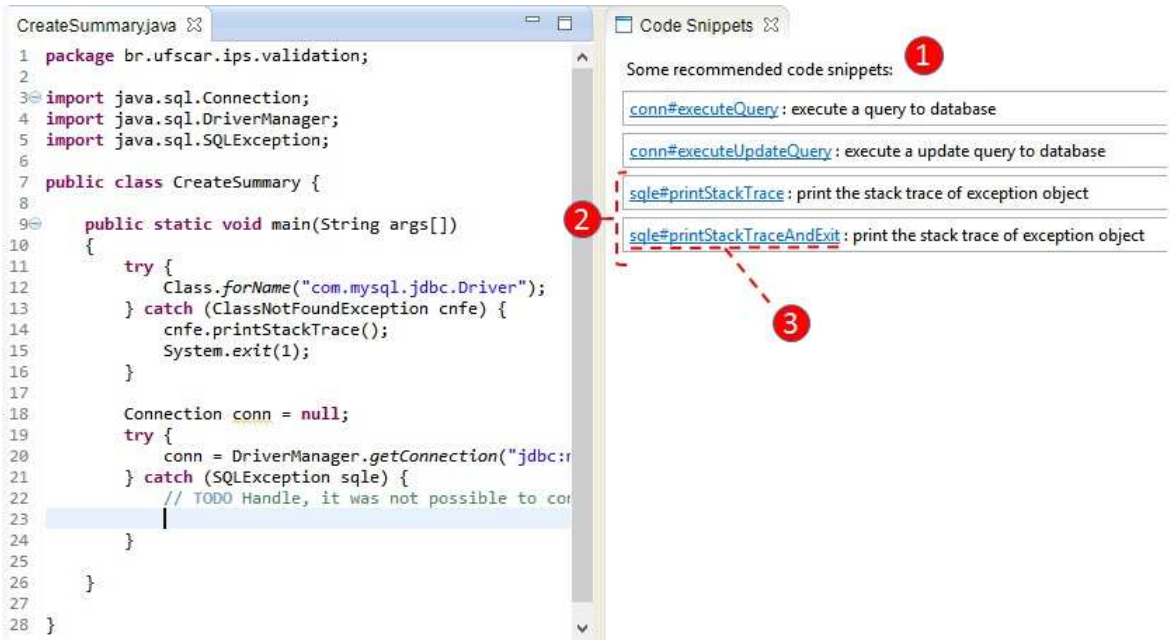


Fonte: próprio autor

recomendador de code snippet atualiza a lista de recomendações (veja item 1 destacado na Figura 40). Com isso, o *recomendador* passa a recomendar dois *code snippets* que antes não estavam sendo recomendados: o *code snippet printStackTrace* e o *code snippet printStackTraceAndExit* (veja item 2 destacado na Figura 40), ambos associados à classe *java.sql.SQLException*. Esses dois *code snippets* são recomendados, pois, na linha 23 (onde se encontra o cursor de texto), a variável *sqle* é acessível e ela é do tipo *java.sql.SQLException*. A variável *conn* declarada na linha 18, que é do tipo da classe *java.sql.Connection*, também é acessível na linha 23. Logo, os *code snippets executeQuery* e *executeUpdateQuery* continuam sendo recomendados.

A fim de tratar a exceção do tipo *SQLException*, o usuário resolve ativar o *code snippet printStackTraceAndExit* para a variável *sqle* através da recomendação fornecida pelo *recomendador* de *code snippet*. Para isso, ele pressiona o botão esquerdo do mouse com o cursor do mouse sobre a recomendação *sqle#printStackTraceAndExit* (item 3 destacado na Figura 40) e é aberta uma janela que exibe as informações do *code snippet* (item 1 destacado na Figura 41). O usuário pressiona o botão *Insert* (item 2 destacado na Figura 41) e o trecho de código gerado pelo *code snippet* (item 3 destacado Figura 41) é inserido na linha 23 (veja Figura 42).

Figura 40 – *Recomendador*: passo 2 do cenário de uso



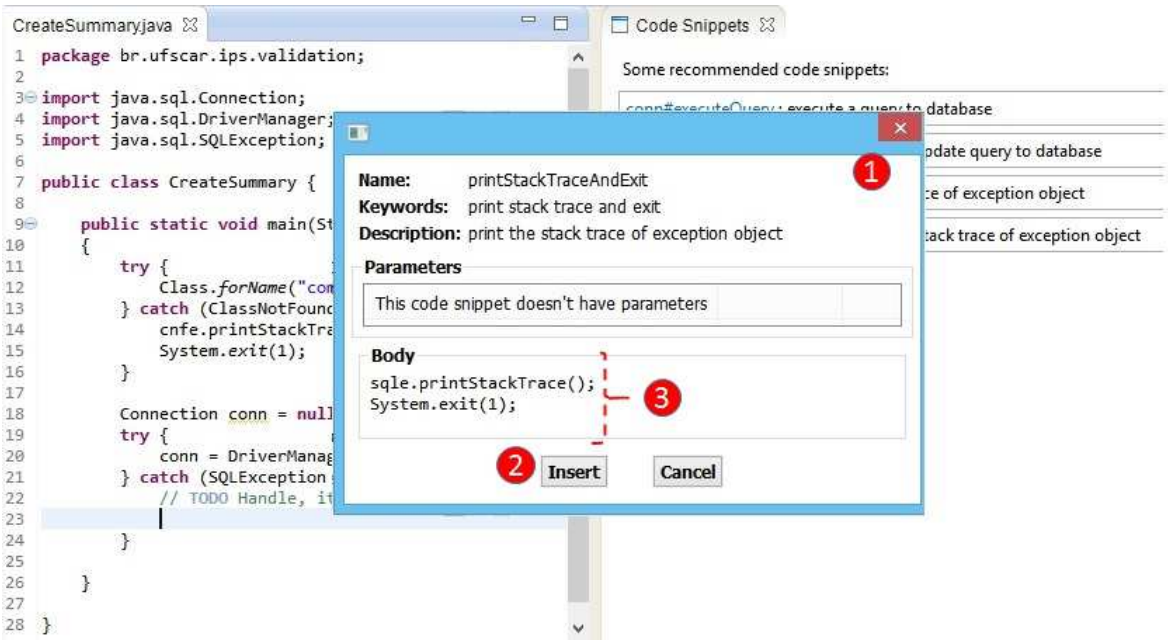
Fonte: próprio autor

2.2.2 Renomeação automática de variáveis

Um trecho de código gerado por um *code snippet* pode declarar uma variável com o nome igual ao nome de uma variável já declarada no escopo do código fonte sendo editado. Para evitar que um trecho de código gerado por um *code snippet* insira erros de compilação, o *IPS* renomeia variáveis cujo nome seja igual ao nome de alguma variável no escopo do código fonte sendo editado. Para exemplificar essa situação, na Figura 43 é apresentado um cenário de uso no qual o usuário está implementando um método denominado *search* que possui dois parâmetros: *elements* do tipo *ArrayList* e *element* do tipo *Object* (item 1 destacado na Figura 43). A fim de obter auxílio na tarefa de implementar o método, o usuário procura pelo *code snippet forEachElement*, que é um *code snippet* associado à classe *ArrayList* e cujo corpo possui um trecho de código que realiza uma iteração para cada elemento de uma lista representada por um objeto do tipo *ArrayList*. O corpo do *code snippet forEachElement* declara uma variável cujo nome é *element* (item 2 destacado na Figura 43). Como no escopo do código fonte sendo editado já existe uma variável com o nome *element* (parâmetro *element*, item 1 destacado na Figura 43), a variável é renomeada no trecho de código gerado pelo *code snippet* (item 3 destacado na Figura 43), evitando dessa forma o conflito de nomes entre as variáveis.

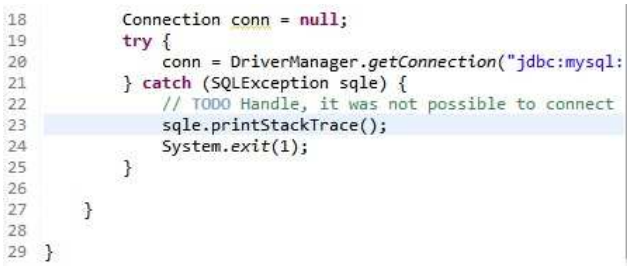
O *IPS* renomeia uma variável que tenha um conflito de nomes concatenando o nome original da variável com um inteiro. Inicialmente esse inteiro possui o valor 1. No entanto, esse inteiro é incrementado até que seja formado um nome que não possua um

Figura 41 – *Recomendador*: passo 3 do cenário de uso



Fonte: próprio autor

Figura 42 – *Recomendador*: passo 4 do cenário de uso



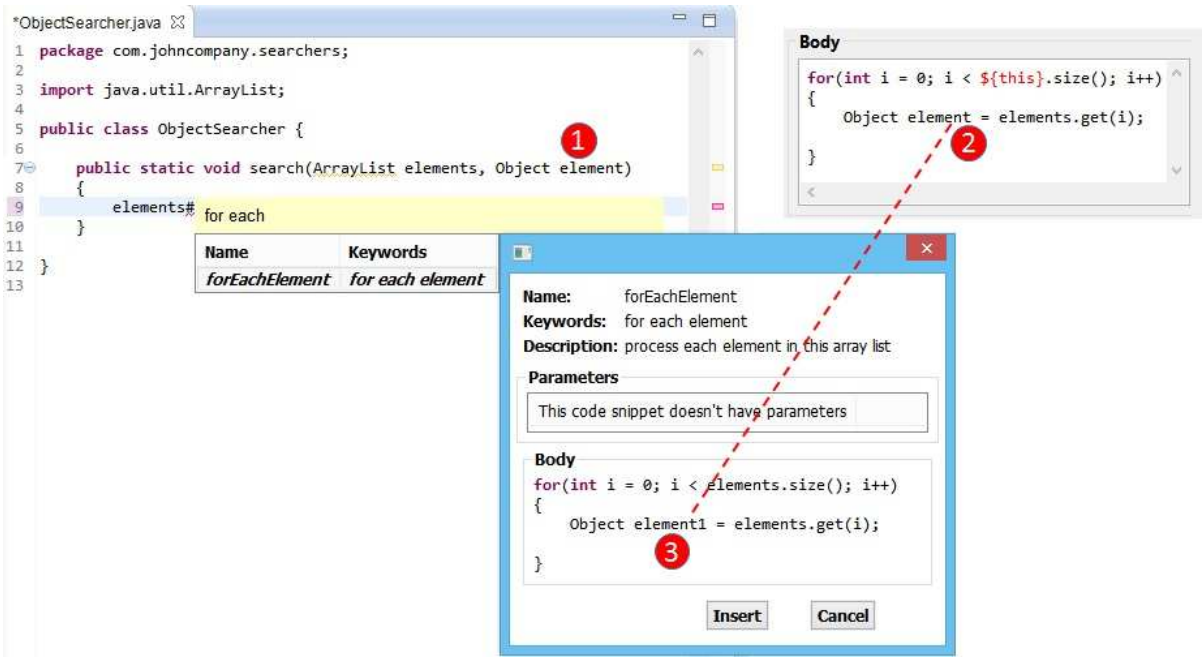
Fonte: próprio autor

conflito com o nome de alguma variável já declarada no escopo. Por exemplo, se o *code snippet* declara uma variável de nome *var* e no escopo existem variáveis com os nomes *var*, *var1* e *var2*; o novo nome da variável declarada no *code snippet* será *var3*.

2.2.3 Herança

Se uma classe *java* A estende uma classe B, A herda todos os *code snippets* de B. Ou seja, os *code snippets* associados a B também são associados a A. Essa herança é transitiva, ou seja, se A estende B e B estende C, A herda os *code snippets* de B e de C, da mesma forma que ocorre com os métodos em *java*. Formalmente temos a seguinte definição:

Figura 43 – Renomeação de variável



Fonte: próprio autor

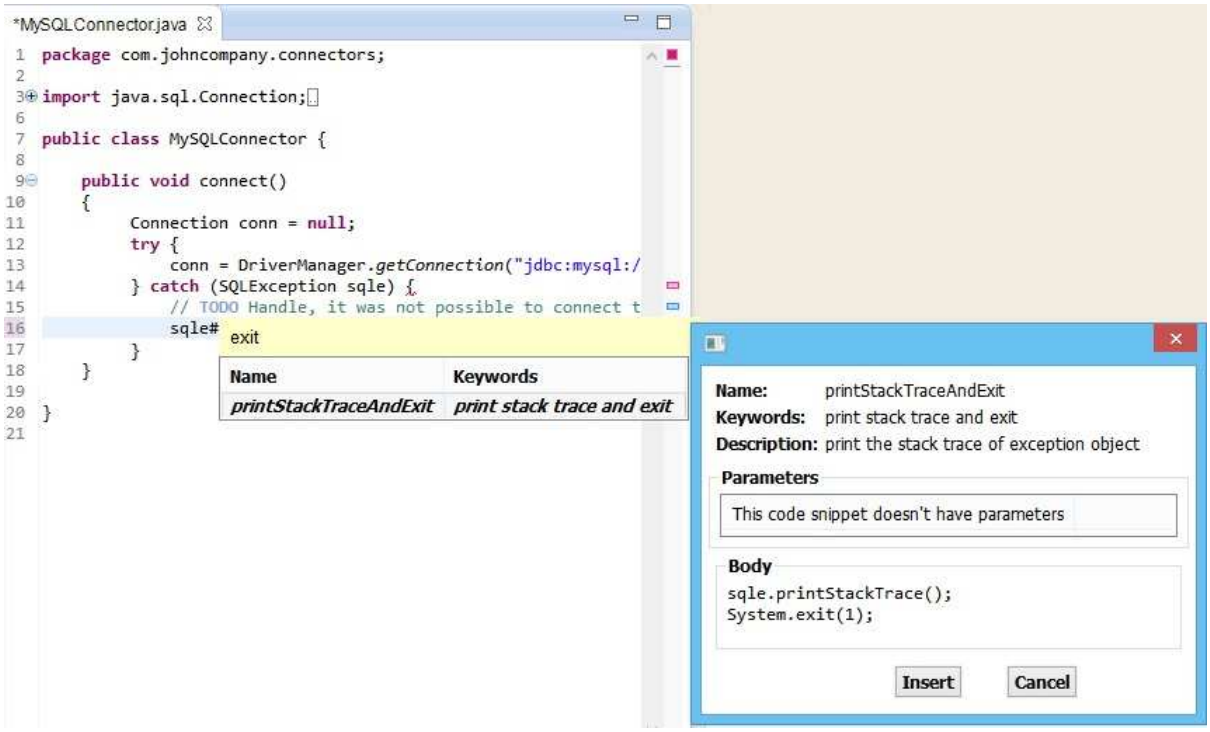
$$cspt(A) = \begin{cases} csptd(A), & \text{se } A = \ java.lang.Object \text{ (caso base)} \\ csptd(A) \ \cup \ cspt(\ parent(A)\ ), & \text{caso contrário} \end{cases}$$

Onde:

- *cspt* é uma função que aceita uma classe *java* A e retorna um conjunto de *code snippets* associados a A;
- *csptd* é uma função que aceita uma classe *java* A e retorna um conjunto de *code snippets diretamente associados* a A. Um *code snippet* é diretamente associado a uma classe *java* A, se o *code snippet* está declarado no arquivo de extensão *XCSPT* de mesmo nome e no mesmo pacote da classe A;
- *parent* é uma função que aceita uma classe *java* A e retorna uma classe B tal que A estende B.

Essa funcionalidade permite que um *code snippet* associado diretamente a uma determinada classe possa ser ativado para um objeto cujo tipo seja de uma outra classe mais específica. Por exemplo, na Figura 44 é apresentado um cenário de uso no qual o usuário, na linha 16 do código do arquivo *MySQLConnector.java*, precisa tratar uma exceção do tipo *SQLException* representada pela variável *sqle*. Para isso, o usuário procura

Figura 44 – Herança de *code snippets*



Fonte: próprio autor

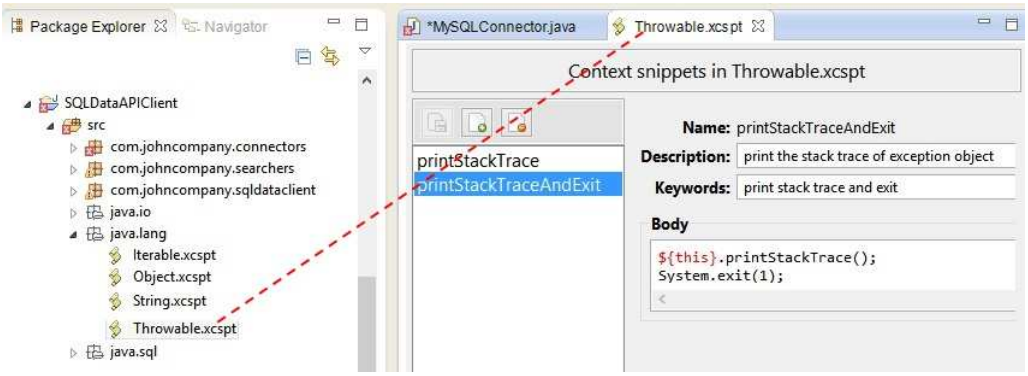
por um *code snippet* associado à classe *java.sql.SQLException*[2]. Ele utiliza a palavra-chave *exit* e é retornado um *code snippet* denominado *printStackTraceAndExit*. O *code snippet* contém um trecho de código que imprime o conteúdo da pilha de execução no momento do lançamento da exceção e depois encerra o programa que está sendo executado. O *code snippet printStackTraceAndExit* é definido no arquivo *Throwable.xcspt* no pacote *java.lang*, portanto é associado diretamente à classe *java.lang.Throwable*[3]. *java.lang.Throwable* é uma classe pertencente à *API* padrão da plataforma *Java* e é a superclasse de todos os *erros* e de todas as *exceções* da linguagem *Java*. Como a classe *java.sql.SQLException* estende a classe *java.lang.Exception* que por sua vez estende a classe *java.lang.Throwable*, o *code snippet printStackTraceAndExit* associado à classe *java.lang.Throwable* é retornado para a variável *sqle* que é do tipo da classe *java.sql.SQLException*.

2.2.4 Mapeamento de argumentos dos *adapters*

Em alguns casos, uma classe associada a um *adapter* pode possuir métodos cuja ordem dos parâmetros não seguem a mesma ordem definida no método correspondente no *adapter*. Por exemplo, na Figura 46 é apresentada a definição de um *adapter* denominado *PointDistanceCalculator*, que é utilizado no *code snippet calcTrianglePerimeter*

[2] <http://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html>
[3] <http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>

Figura 45 – Definição do *code snippet printStackTraceAndExit*



Fonte: próprio autor

apresentado na Figura 47. O *code snippet calcTrianglePerimeter* possui um parâmetro denominado *distanceCalculator* do tipo *role* associado ao *adapter PointDistanceCalculator* (item 1 destacado na Figura 47). O corpo do *code snippet* (*body*) possui um trecho de código que calcula o perímetro de uma triângulo. O perímetro de um triângulo é calculado por meio da soma da distância de cada par de pontos que forma o triângulo. Se um triângulo é formado pelos pontos A, B e C; seu perímetro é a soma da distância entre os pontos A e B, mais a distância entre os pontos B e C, mais a distância entre os pontos C e A. A distância de cada par de pontos do triângulo é calculada através de alguma classe associada ao *adapter PointDistanceCalculator*, que possui um método denominado *calcDistance*. No corpo do *code snippet*, é referenciado o método *calcDistance* (item 2 destacado na Figura 47), que possui quatro parâmetros do tipo *double* : *x1*, *y1*, *x2* e *y2* (veja item 1 destacado na Figura 46). *x1* e *y1* são as dimensões x e y do primeiro ponto; *x2* e *y2* são as dimensões x e y do segundo ponto. O método *calcDistance* deve retornar a distância entre o primeiro e o segundo ponto. Na Figura 48, é apresentada a definição da classe *MyDistanceCalculator*, que é associada ao *adapter PointDistanceCalculator*. O método *calcDistance* da classe *MyDistanceCalculator* (item 1 destacado na Figura 48) é mapeado para o método *calcDistance* do *adapter PointDistanceCalculator* (item 1 destacado na Figura 46) através da *annotation MethodMappedTo*. No entanto, a ordem dos parâmetros do método *calcDistance* da classe *MyDistanceCalculator* é diferente da ordem dos parâmetros do método *calcDistance* do *adapter PointDistanceCalculator*. Os quatro parâmetros do método *calcDistance* da classe *MyDistanceCalculator* são *x1*, *x2*, *y1* e *y2* (item 1 destacado na Figura 48). *x1* e *x2* são respectivamente a dimensão x do primeiro ponto e a dimensão x do segundo ponto. *y1* e *y2* são respectivamente a dimensão y do primeiro e a dimensão y do segundo ponto. Ou seja, enquanto para o método *calcDistance* do *adapter PointDistanceCalculator* a ordem dos parâmetros é as dimensões x e y do primeiro ponto seguidas das dimensões x e y do segundo ponto, para o método *calcDistance* da classe *MyDistanceCalculator* a ordem dos parâmetro é as dimensões x do primeiro e do

segundo ponto seguidas das dimensões *y* do primeiro e do segundo ponto. Dessa forma, é necessário realizar também o mapeamento entre os parâmetros dos dois métodos. Isso é realizado através da anotação *ParameterMappedTo* (item 2 destacado na Figura 48). A anotação *ParameterMappedTo* possui um parâmetro denominado *argumentsIndexes* que mapeia os parâmetros do método na classe para os parâmetros do método no *adapter*. O parâmetro aceita uma *string* formada por inteiros separados por vírgula. Cada inteiro informa a posição do parâmetro no método do *adapter*. Para um método *M* anotado com a anotação *ParameterMappedTo* e mapeado através da anotação *MethodMappedTo* para um método *MA* em um adapter *A*, o valor *j* do *i-ésimo* inteiro em *argumentsIndexes* indica que o *i-ésimo* parâmetro no método *M* é mapeado para o *j-ésimo* parâmetro do método *MA*. No exemplo apresentado na Figura 48, a anotação *ParameterMappedTo* (item 2 destacado) mapeia respectivamente o primeiro, o segundo, o terceiro e o quarto parâmetro do método *calcDistance* na classe *MyDistanceCalculator* para o primeiro, para o terceiro, para o segundo e para o quarto parâmetro do método *calcDistance* no *adapter PointDistanceCalculator*.

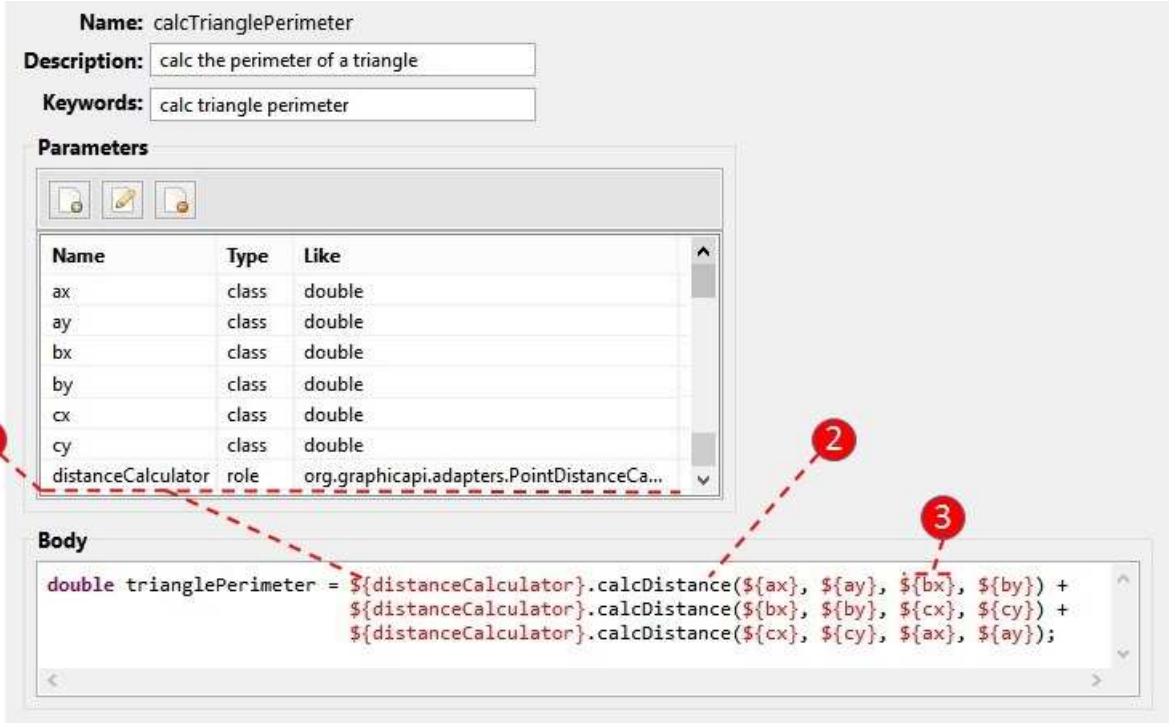
Figura 46 – Definição do *adapter PointDistanceCalculator*

```
1  package org.graphicapi.adapters;
2
3  import br.ufscar.ppgccs.ips.adapters.RoleAdapter;
4
5  public interface PointDistanceCalculator extends RoleAdapter {
6
7      public double calcDistance(double x1, double y1, double x2, double y2);  1
8
9  }
```

Fonte: próprio autor

Na Figura 49, é apresentado um cenário de uso onde o usuário está editando o código fonte *Main.java* e na linha 11 o *code snippet calcTrianglePerimeter* é ativado. Para o parâmetro *distanceCalculator* do *code snippet calcTrianglePerimeter*, o usuário seleciona a variável local *distanceCalculator*, variável do tipo *MyDistanceCalculator* declarada na linha 10 do código fonte *Main.java*. Com isso, as referências para o método *calcDistance* do *adapter PointDistanceCalculator* no corpo do *code snippet calcTrianglePerimeter* (veja item 2 destacado na Figura 47) são adaptadas para o método *calcDistance* da classe *MyDistanceCalculator* seguindo o mapeamento de parâmetros definido através da anotação *ParameterMappedTo* (item 2 destacado na Figura 48). Por exemplo, perceba que o parâmetro *bx* é utilizado como terceiro argumento do método *calcDistance* referenciado no corpo do *code snippet calcTrianglePerimeter* (veja item 3 destacado na Figura 47), enquanto que o parâmetro *bx* na ativação do *code snippet*, que recebe como argumento a variável local *pBX*, é utilizado como segundo argumento do método *calcDistance* (veja

Figura 47 – Definição do *code snippet calcTrianglePerimeter*



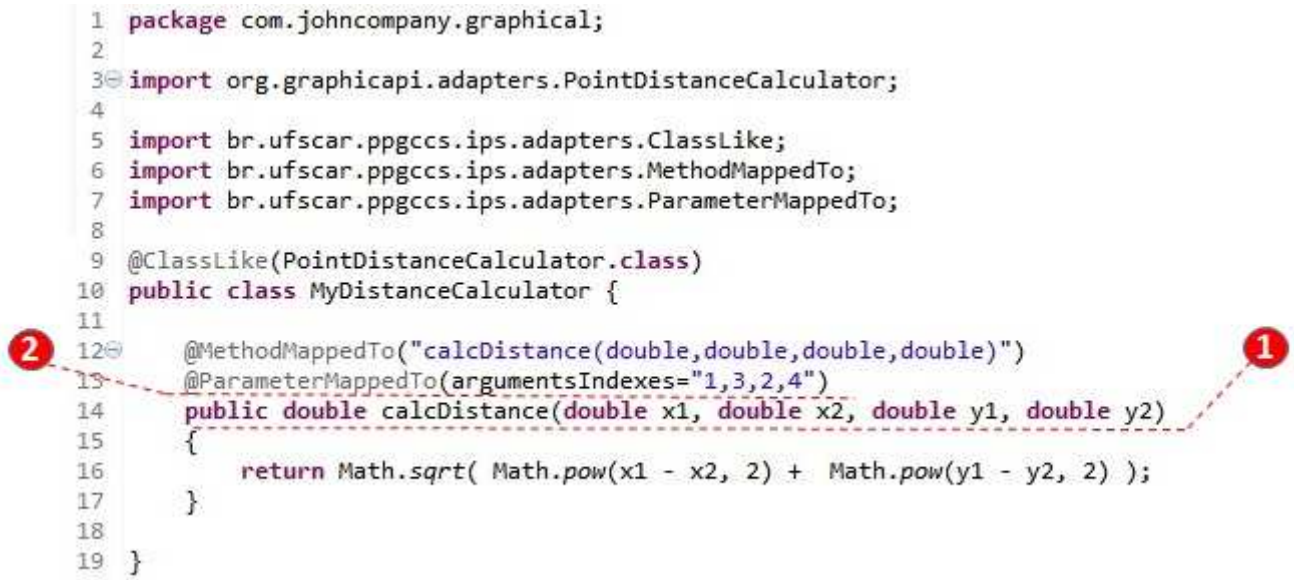
Fonte: próprio autor

item 1 destacado na Figura 49), pois o segundo parâmetro do método *calcDistance* da classe *MyDistanceCalculator* é mapeado para o terceiro parâmetro do método *calcDistance* do *adapter PointDistanceCalculator* (veja item 2 destacado na Figura 48).

2.3 Arquitetura e implementação

O *IPS*, *Intuitive Programming System*, é um plug-in para o *Eclipse*. Um *plug-in* é desenvolvido utilizando a própria *IDE Eclipse* através de um tipo especial de projeto denominado *Plug-in*. Grande parte da implementação do *IPS* foi desenvolvida a partir de informações de Blewitt (2013) e de Vogella Company (2016), que descrevem os detalhes necessários para implementar um *plug-in* para o *Eclipse*. O *IPS* consiste de dois módulos: um editor de *code snippet* denominado *CSPTE*, *Code SniPpet Editor*, e um assistente de código denominado *IPSCA*, *Intuitive Programming System Code Assistant*. O módulo *CSPTE* (item 1 destacado na Figura 50) consiste de uma interface gráfica que auxilia o desenvolvedor de uma *API* a criar e a editar os arquivos *XCSPT* (item 2 destacado na Figura 50). Cada arquivo *XCSPT* é um arquivo *XML* que define um conjunto de *code snippets*. Os *code snippets* são utilizados para catalogar os padrões de uso das classes da

Figura 48 – Classe *MyDistanceCalculator*




```java
package com.johncompany.graphical;

import org.graphicapi.adapters.PointDistanceCalculator;

import br.ufscar.ppgccs.ips.adapters.ClassLike;
import br.ufscar.ppgccs.ips.adapters.MethodMappedTo;
import br.ufscar.ppgccs.ips.adapters.ParameterMappedTo;

@ClassLike(PointDistanceCalculator.class)
public class MyDistanceCalculator {

    @MethodMappedTo("calcDistance(double,double,double,double)")
    @ParameterMappedTo(argumentsIndexes="1,3,2,4")
    public double calcDistance(double x1, double x2, double y1, double y2)
    {
        return Math.sqrt( Math.pow(x1 - x2, 2) +  Math.pow(y1 - y2, 2) );
    }

}
```

Fonte: próprio autor

API, são definidos nos arquivos *XCSPT* e são empacotados em arquivos *JAR*[4] (item 3 destacado na Figura 50). Um arquivo *JAR* é utilizado pelo desenvolvedor de uma *API* para distribuir a *API* aos usuários. O arquivo *JAR* utilizado para distribuir a *API* pode conter, além das classes *Java* da *API* [5] (item 4 destacado na Figura 50), os arquivos *XCSPT* que definem os *code snippets* que auxiliam a utilização das classes da *API*. O usuário de uma *API* pode importar o arquivo *JAR* para utilizar a *API* em seu projeto *Java* na *IDE Eclipse* (item 5 destacado na Figura 50). Durante a edição de arquivos código fonte *Java* na *IDE Eclipse*, o módulo *IPSCA* (item 6 destacado na Figura 50) auxilia o usuário da *API* a encontrar e a utilizar os *code snippets* desejados. O *IPSCA* auxilia o usuário a configurar os argumentos para os parâmetros do *code snippet* e insere o trecho de código *Java* gerado a partir do *code snippet* no código fonte sendo editado (veja item 7 destacado na Figura 50).

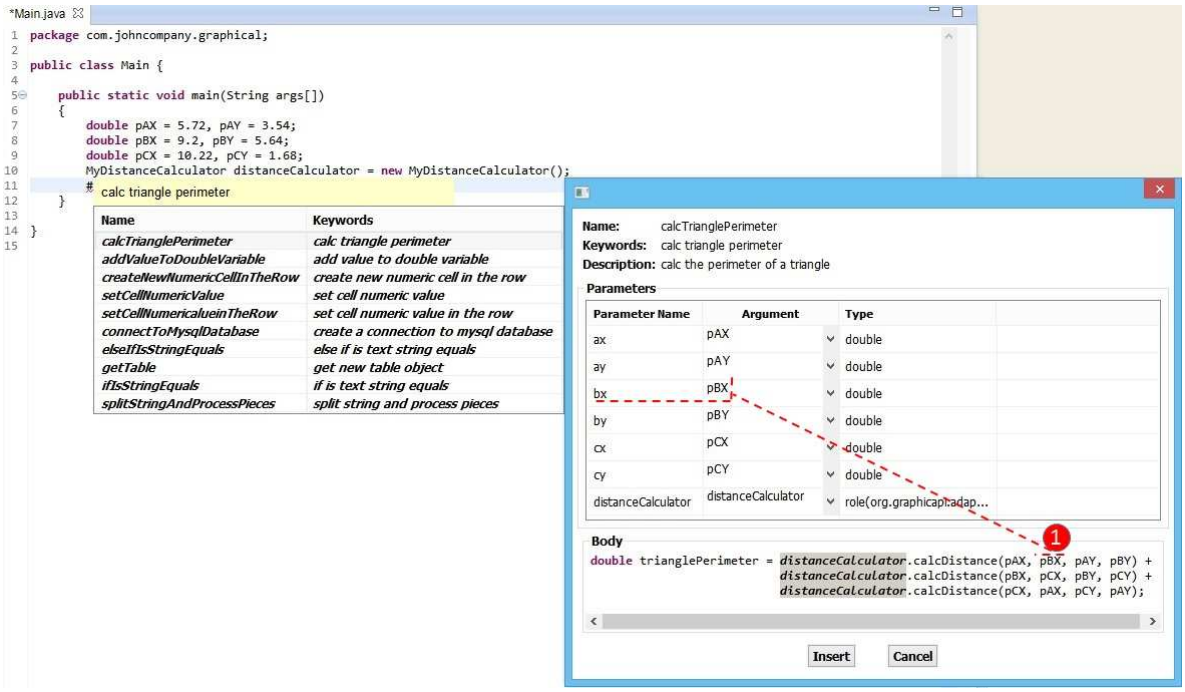
2.3.1 *Code snippets*

Os *templates de código* do *IPS* são denominados simplesmente de *code snippets*. O desenvolvedor de uma *API* pode utilizar os *code snippets* para catalogar os padrões

[4] *JAR — Java ARchive*. É um formato de arquivo baseado no formato de arquivo *ZIP*. Na plataforma *Java* é utilizado para empacotar componentes de uma biblioteca *Java* em único arquivo <http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jarGuide.html>.

[5] Na plataforma *Java* o códigos das classes e das interfaces após compiladas (*bytecodes*) são armazenados em arquivos com a extensão *class*, que é um formato de arquivo utilizado pela máquina virtual *Java*. Cada arquivo *class* contém a definição de uma única classe ou interface *Java* <https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html>.

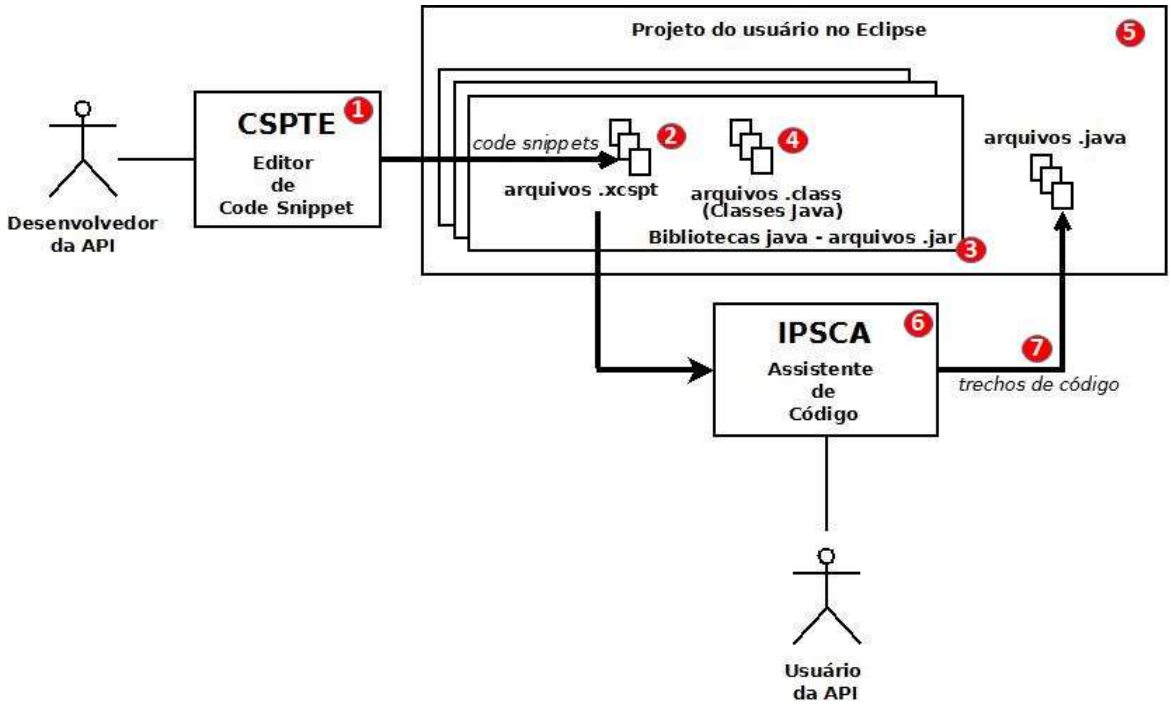
Figura 49 – Uso do *code snippet calcTrianglePerimeter*



Fonte: próprio autor

de uso das classes da *API*. Os *code snippets* podem ser associados a classes *Java*. A associação entre uma classe *Java* e os *code snippets* é realizada através de arquivos com a extensão *xcspt*, que contem a definição de um conjunto de *code snippets* associados a uma determinada classe. Por exemplo, para realizar a associação de um conjunto de *code snippets* a uma classe pública *Java* denominada *MinhaClasse* definida no arquivo *MinhaClasse.java*, basta criar um arquivo com a extensão *xcspt* com o nome *MinhaClasse.xcspt*. O arquivo deve estar no mesmo pacote da classe. A estrutura do arquivo é apresentada na próxima Subseção (Subseção 2.3.2). Um *code snippet* tem: um nome, uma descrição, palavras-chaves associadas, uma lista de *imports*, uma lista de parâmetros e um corpo. O nome é utilizado para identificar o *code snippet*. A descrição é um texto breve descrevendo o objetivo do *code snippet*. As palavras-chaves permitem ao usuário realizar buscas não necessitando lembrar o nome exato do *code snippet* desejado. As palavras-chaves podem ser palavras que transmitem a ideia de comando. Por exemplo, para um *code snippet* com o objetivo de conectar em uma base *MySQL*, as seguintes palavras-chaves podem ser associadas: "*connect to MySQL database*". O corpo do *code snippet* define o trecho de código *Java* que é gerado durante o acionamento do *code snippet*. A lista de *imports* é uma lista de identificadores de classes *Java* que devem ser importadas ao acionar o *code snippet* para que o trecho de código *Java* gerado pelo *code snippet* funcione corretamente. Os parâmetros permitem o reaproveitamento do corpo em contextos diferentes, eles podem ser referenciados no corpo e, durante o acionamento do *code snippet*, cada referência do parâmetro no corpo do *code*

Figura 50 – Componentes do *IPS*



Fonte: próprio autor

snippet é substituído pelo seu respectivo argumento.

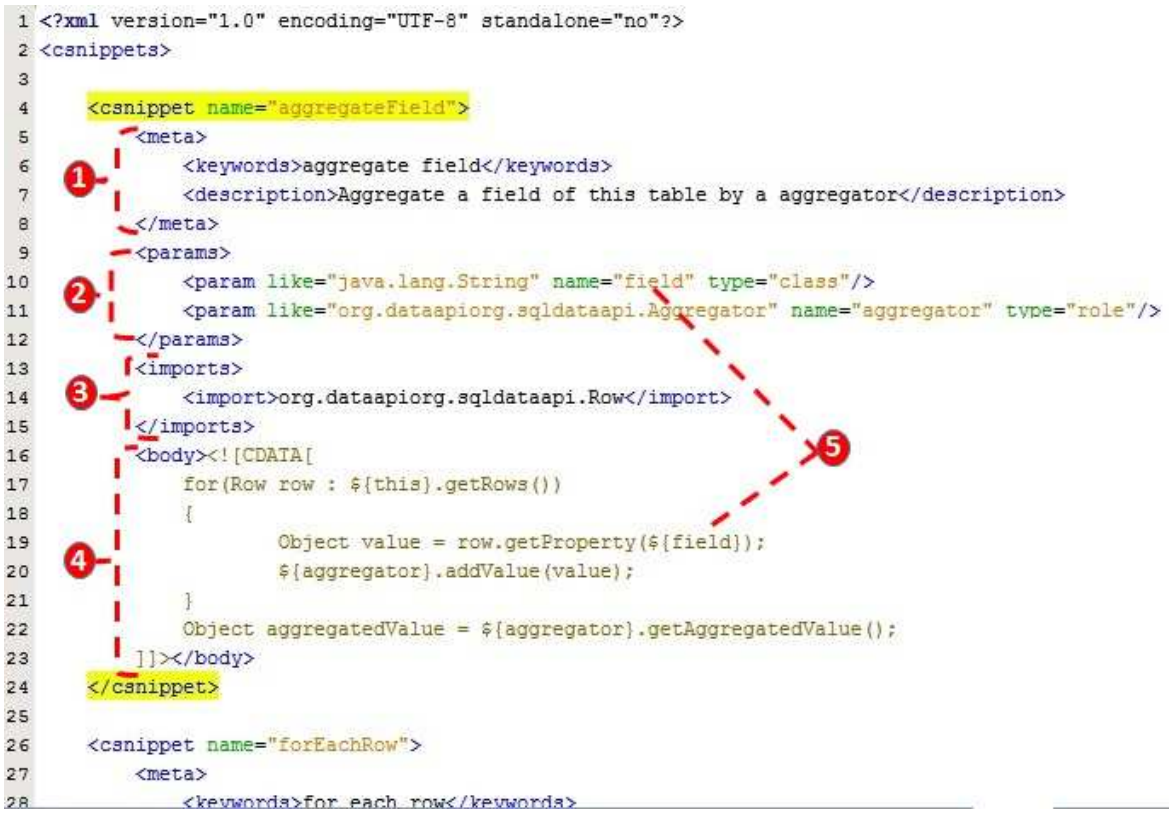
2.3.2 Estrutura do arquivo *XCSPT*

Um arquivo *XCSPT*, *XML Code SniPpet*, é um arquivo no formato *XML*[6]. A *tag* raiz do arquivo é a *tag csnippets* (veja linha 2 no trecho exibido na Figura 51) e cada *code snippet* é definido pela *tag csnippet* (linha 4). O atributo *name* da *tag csnippet* define o nome do *code snippet*. A *tag csnippet* possui quatro seções: *meta* (item 1 destacado na Figura 51), *params* (item 2), *imports* (item 3) e *body* (item 4). *meta* define a descrição do *code snippet* e as palavras-chaves associadas, que podem ser utilizadas para pesquisar pelo *code snippet*. A seção *params* define os parâmetros do *code snippet*. *body* contém o corpo do *code snippet* e define, através de um trecho de código *Java*, um uso particular da *API* que o *code snippet* pertence.

Um *code snippet* pode ter um conjunto de parâmetros e cada parâmetro é definido através da *tag param* na seção *params* (item 2 na Figura 51). Há dois tipos de parâmetros possíveis: *class* e *role*. Conforme ilustrado no cenário de uso apresentado na Seção 2.1 deste Capítulo, os parâmetros aceitam como argumento identificadores de objetos (variáveis locais ou atributos de classes) no escopo do código fonte *Java* em edição no ambiente de desenvolvimento *Eclipse*. Parâmetros do tipo *class* aceitam como argumento identificadores

[6] Formato de texto flexível padronizado pela *W3C*. <http://www.w3.org/XML/>.

Figura 51 – Arquivo *XCSPT*



Fonte: próprio autor

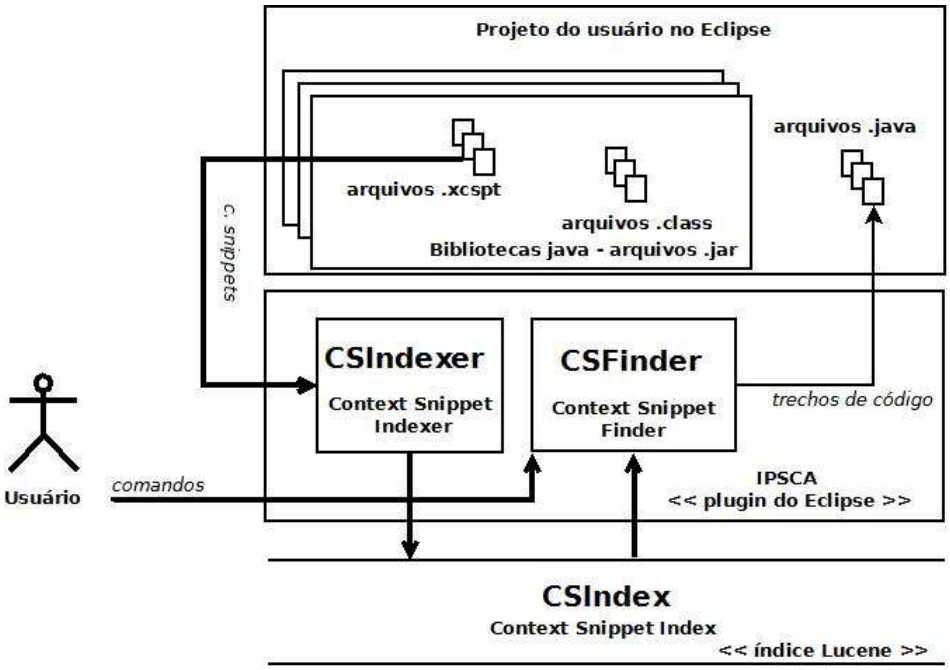
de objetos do tipo definido por uma classe ou uma interface *Java* associada ao parâmetro. A classe ou a interface *Java* associada ao parâmetro é definida pelo atributo *like* na definição do parâmetro. Para os parâmetros do tipo *role*, a seleção dos possíveis argumentos do parâmetro é realizada através dos *adapters*. Um parâmetro do tipo *role* tem um *adapter* associado. O *adapter* associado ao parâmetro é definido através do atributo *like* na definição do parâmetro. Os parâmetros são referenciados no corpo do *code snippet* através do caractere *$* seguido do nome do parâmetro delimitado pelos caracteres *{* e *}*. Por exemplo, no corpo do *code snippet aggregateField* apresentado na Figura 51, o parâmetro *field* é referenciado através da diretiva *${field}* (veja item 5 destacado), e o parâmetro *aggregator* é referenciado através da diretiva *${aggregator}* (nas linhas 20 e 22). Durante o acionamento do *code snippet*, cada referência do parâmetro no corpo do *code snippet* é substituído pelo seu respectivo argumento.

2.3.3 Módulo *IPSCA*

O módulo *IPSCA*, *Intuitive Programming System Code Assistant*, auxilia o usuário na localização e na utilização dos *code snippets*. Ele é composto de três componentes:

- um índice de *code snippets* denominado *CSIndex — Code Snippet Index*;
- um indexador de *code snippets* denominado *CSIndexer — Code Snippet Indexer*;
- e um localizador de *code snippets* denominado *CSFinder — Code Snippet Finder*.

Figura 52 – Arquitetura do *IPSCA*



Fonte: próprio autor

2.3.3.1 *CSIndex*

O índice *CSIndex* é implementado através da *API* de busca e indexação de documentos *Lucene* (APACHE SOFTWARE FOUNDATION, 2016a). O *Lucene* possui um modelo de dados tal que cada registro no índice é representado por uma entidade denominada *Document*[7]. Cada *Document* possui um conjunto de campos e cada campo é representado por uma entidade denominada *Field*[8]. Um *Field* possui um nome e um conjunto de valores. Os *code snippets* indexados em *CSIndex* são mapeados para o modelo de dados do *Lucene*. Cada *code snippet* indexado é um *Document* no índice contendo nove campos (*Fields*), os quais são descritos na Tabela 7. Supondo que o *code snippet aggregateField*, apresentado na Figura 51, esteja presente em um arquivo *XCSPT* com o nome *Table.xcspt*, no pacote *org.dataapiorg.sqldataapi*, em um arquivo de nome *SQLDataAPIClient.jar*, em um projeto

[7] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Document.html>
[8] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/document/Field.html>

Java de nome *SQLDataAPIClient* e no *workspace*[9] `<C:\eclipse\workspace>`, o *code snippet* seria mapeado em um registro no índice *CSIndex* com as colunas contendo valores conforme apresentado na Tabela 8.

Tabela 7 – Colunas no índice *CSIndex*

Nome da coluna	**Descrição**
workspace	Contém o caminho do diretório utilizado pelo *Eclipse* para armazenar os projetos do usuário.
project	Contém o nome do projeto *Java* no *workspace* onde se encontra o *code snippet* indexado.
name	Contém o nome do *code snippet* indexado.
associated_class	Identificador da classe *Java* associada ao *code snippet*.
keywords	Contém as *keywords* associadas ao *code snippet* indexado.
param	Contém uma lista de *strings*. Cada *string* representa um parâmetro do *code snippet* indexado. Se o parâmetro for do tipo *class*, a *string* que representa o parâmetro é formado pela palavra *class* seguida do qualificador completo da classe ou da interface Java associada ao parâmetro delimitado por parênteses. Por exemplo, para um parâmetro do tipo *class* associado à classe *String* do pacote *java.lang*, a *string* que representa o parâmetro é a *string class(java.lang.String)*. Se o parâmetro for do tipo *role*, a *string* que representa o parâmetro é formado pela palavra *role* seguida do qualificador completo do *adapter* associado ao parâmetro delimitado por parênteses. Por exemplo, para um parâmetro do tipo *role* associado ao *adapter Aggregator* do pacote *org.dataapiorg.sqldataapi*, a *string* que representa o parâmetro é a *string "role(org.dataapiorg.sqldataapi.Aggregator)"*.
import	Contém uma lista de *strings*. Cada *string* representa um *import* do *code snippet* indexado.
xml_content	Contém o conteúdo *XML* que define o *code snippet* indexado. Corresponde ao trecho do arquivo *XCSPT* de onde o *code snippet* foi extraído.

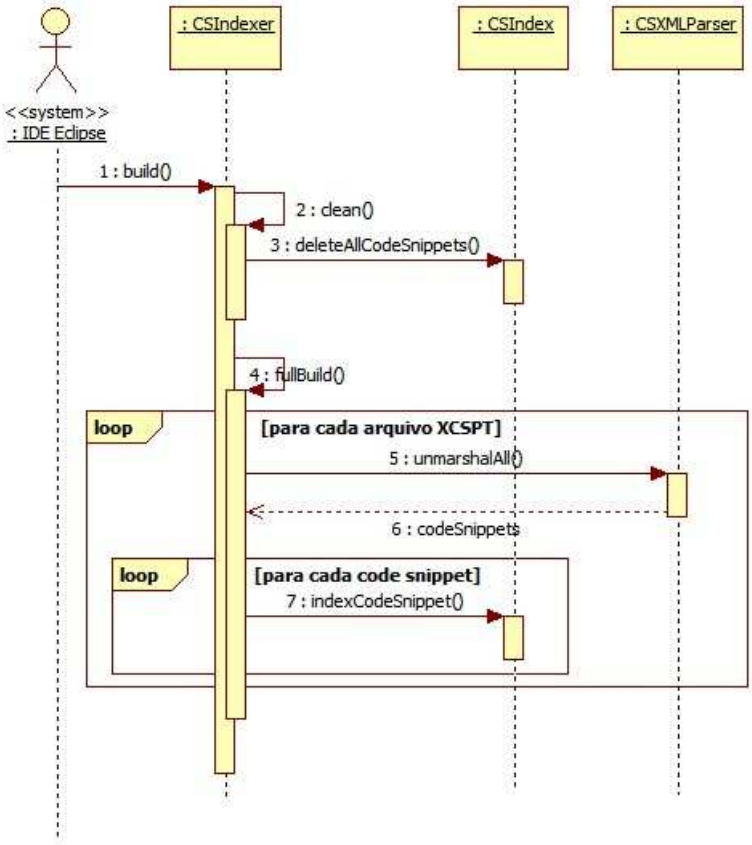
Fonte: próprio autor

2.3.3.2 *CSIndexer*

O módulo *CSIndexer* possui dois modos de indexação: uma indexação completa e uma indexação incremental. A indexação completa ocorre quando um *build* dos projetos

[9] *Workspace* é um termo utilizado no *Eclipse* para designar um diretório, cuja localização é configurável, utilizado para armazenar os arquivos dos projetos do usuário e também onde são armazenados arquivos que contém as preferências do usuário <http://www.vogella.com/tutorials/Eclipse/article.html#workspace>.

Figura 53 – Diagrama de sequência do *CSIndexer* — indexação completa



Fonte: próprio autor

na *IDE Eclipse* é solicitado pelo usuário (veja item 1 no diagrama de sequência *UML* apresentado na Figura 53). A primeira ação que o módulo *CSIndexer* realiza na indexação completa é remover todos os *code snippets* já presentes no índice (itens 2 e 3 na Figura 53). Logo após, para cada projeto *Java* na *IDE Eclipse*, o módulo procura em todas as bibliotecas do projeto (arquivos *JAR*) por arquivos *XCSPT*. Para cada arquivo *XCSPT* encontrado, é indexado em *CSIndex* todos os *code snippets* do arquivo. Cada arquivo *XCSPT* é um arquivo *XML* e, para obter os *code snippets*, o *CSIndexer* solicita a um objeto da classe *CSXMLParser* a análise sintática do arquivo *XCSPT* (item 5 na Figura 53). A classe *CSXMLParser* é uma classe da implementação do *IPS* e ela utiliza a *API* padrão da plataforma *Java* para manipulação de arquivos *XML*, cujo nome é *JAXP*[10], para obter o conteúdo do arquivo *XCSPT* através do modelo *DOM*[11]. Baseado no modelo *DOM* do conteúdo do arquivo *XCSPT*, a classe *CSXMLParser* cria um objeto que representa cada *code snippet* definido no arquivo (item 6 na Figura 53). Cada *code snippet* é indexado em

[10] <https://docs.oracle.com/javase/tutorial/jaxp/>.

[11] *DOM*, *Document Object Model*, modelo padronizado pela *W3C* utilizado como uma representação intermediária, em memória, do conteúdo de arquivos *HTML* e *XML* <http://www.w3.org/DOM/Overview.html>.

Tabela 8 – Colunas do *code snippet aggregateField* no índice

Nome da coluna	**Valor(es)**
workspace	*C:\eclipse\workspace*
project	*SQLDataAPIClient*
name	*aggregateField*
associated_class	*org.dataapiorg.sqldataapi.Table*
keywords	*aggregate field*
param	*class(java.lang.String)* *role(org.dataapiorg.sqldataapi.Aggregator)*
import	*org.dataapiorg.sqldataapi.Row*
xml_content[12]	*<csnippet name="aggregateField">* *...* *</csnippet>*

Fonte: próprio autor

CSIndex (item 7 na Figura 53). De outra maneira, a indexação incremental ocorre quando uma nova biblioteca (arquivo *JAR*) é adicionada em algum projeto *Java*. Nesse caso, o *CSIndexer* indexa somente os *code snippets* presentes nos arquivos *XCSPT* da biblioteca adicionada e não ocorre a remoção de todos os *code snippets* já presentes no índice antes de iniciar indexação. No entanto, durante a indexação de um *code snippet*, caso exista no índice um *code snippet* com o mesmo nome e para o mesmo projeto *Java*, o *code snippet* presente no índice é removido antes que o *code snippet* seja indexado.

2.3.3.3 *CSFinder*

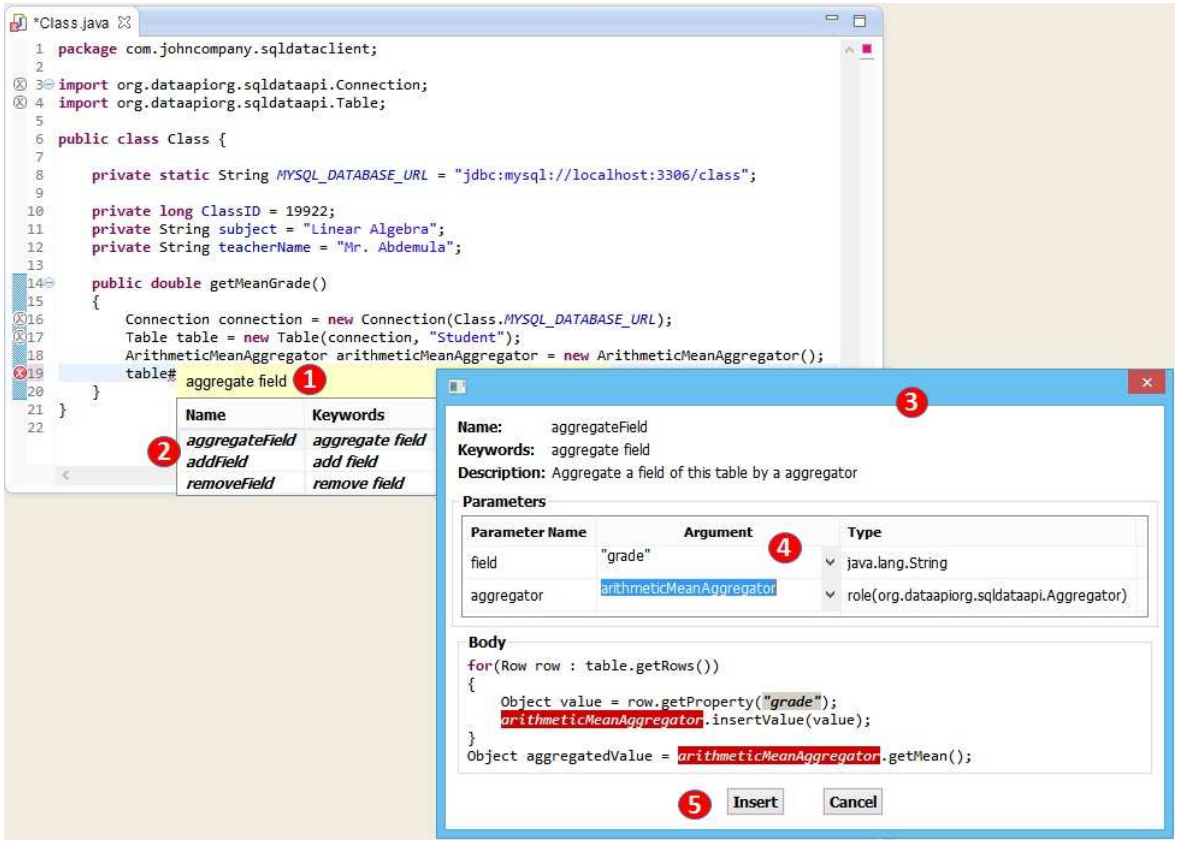
O *CSFinder* é responsável pelas interações com o usuário durante a utilização dos *code snippets*. Ao editar um código fonte *Java* na *IDE Eclipse*, se o usuário digitar um identificador de objeto (variável local ou atributo de classe) seguido do caractere #, o *CSFinder* apresenta um campo de texto para que o usuário digite as palavras-chaves do *code snippet* desejado (veja linha 19 do trecho de código apresentado na Figura 54). Para que isso fosse possível, na implementação do *IPS* foi necessário definir *extensões*. A *IDE Eclipse* define para seus *plug-ins pontos de extensão*[13]. Os *pontos de extensão* são definidos para que os *plug-ins* possam alterar ou adicionar determinados comportamentos em certas funcionalidades da *IDE*. Através de um arquivo denominado *plugin.xml*, um *plug-in* pode definir uma ou mais *extensões* para um determinado *ponto de extensão*. O

[12] O conteúdo da coluna *xml_content* no índice, apresentado na última linha da Tabela 8, está abreviado. O valor dessa coluna contém todo o trecho *XML* que define o *code snippet*. Para o *code snippet aggregateField*, o valor da coluna conteria o trecho compreendido entre as linhas 4 e 24 do código apresentado na Figura 51.

[13] <http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html>

arquivo *plugin.xml* é um arquivo *XML* que pertence ao projeto de implementação do *plug-in* e é utilizado para realizar a instalação do *plug-in* em uma instância da *IDE Eclipse*.

Figura 54 – Interações com o usuário



Fonte: próprio autor

Na Figura 55, é apresentado o trecho do arquivo *plugin.xml* do projeto de implementação do *IPS* que define *extensões* para que seja realizada uma ação sempre que o usuário digitar o caractere #. Uma *extensão* é definida através da *tag extension* e o *ponto de extensão* que a *extensão* faz referência é especificada pelo atributo *point*. Na linha 86, é definida uma *extensão* para o *ponto de extensão* cujo identificador é *org.eclipse.ui.bindings*. O *ponto de extensão org.eclipse.ui.bindings* é definido pelo *Eclipse* para que os desenvolvedores de *plug-ins* possam adicionar ações que sempre são executadas quando o usuário pressionar uma determinada tecla ou uma sequência de teclas[14]. A ação executada e a tecla ou a sequência de teclas que dispara a ação são definidas pela *tag key* (linha 87) e ela possui três atributos: *commandId* (linha 87), *contextId* (linha 88) e *sequence* (linha 89). *commandId* contém o identificador de um *comando*. No *Eclipse*, um *comando* é a abstração de uma ação a ser executado em resposta a algum evento e é definido através de uma *extensão* para o *ponto de extensão org.eclipse.ui.commands* (linha 78). O identificador do comando é definido através

[14] Outros *pontos de extensão* definidos pelo *Eclipse* podem ser encontrados em (ECLIPSE FOUNDATION, 2016c)

Figura 55 – Definição de uma extensão

```
77 </extension>
78 <extension point="org.eclipse.ui.commands">
79    <command id="br.ufscar.ppgccs.ips.commands.ActiveUI"/>
80 </extension>
81 <extension point="org.eclipse.ui.handlers">
82    <handler class="br.ufscar.ppgccs.ips.ActiveUI"
83             commandId="br.ufscar.ppgccs.ips.commands.ActiveUI">
84    </handler>
85 </extension>
86 <extension point="org.eclipse.ui.bindings">
87    <key  commandId="br.ufscar.ppgccs.ips.commands.ActiveUI"
88          contextId="org.eclipse.jdt.ui.javaEditorScope"
89          sequence="#">
90    </key>
91 </extension>
92 <extension point="org.eclipse.ui.editors">
```

Fonte: próprio autor

do atributo *id* da *tag command* (linha 79). No trecho compreendido entre as linhas 78 e 80, é definido um *comando* cujo identificador é *br.ufscar.ppgccs.ips.commands.ActiveUI*, que é referenciado no atributo *commandId* da *tag key* na linha 87. A ação que um comando deve executar é definida por um *handler* que deve ser associado ao *comando*. Um *handler* é definido através de uma extensão para o *ponto de extensão org.eclipse.ui.handlers* (linha 81) e é implementado através de uma classe *Java*. A classe *Java* que implementa um *handler* deve estender a classe *org.eclipse.core.commands.AbstractHandler* e sobrescrever o método *execute*, o qual implementa a ação executada pelo *handler*. A associação entre um *handler* e um comando é realizada através da *tag handler* (linha 82) e ela possui dois atributos: *class* e *commandId*. O atributo *class* deve conter o qualificador completo da classe que implementa o *handler* e *commandId* deve conter o identificador do comando que se deseja associar ao *handler*. No trecho compreendido entre as linhas 81 e 85, é definido um *handler* implementado pela classe *Java br.ufscar.ppgccs.ips.ActiveUI*, classe que pertence ao projeto de implementação do *CSFinder*. Esse *handler* é associado ao comando cujo identificador é *br.ufscar.ppgccs.ips.commands.ActiveUI*, o qual é definido no trecho compreendido entre as linhas 78 e 80. O atributo *contextId* da *tag key* na linha 88, define em qual *contexto* o evento deve ser considerado. Um *contexto*[15] é uma abstração utilizada pelo *Eclipse* para restringir quando um comando associado a um evento deve ser executado. No caso, é utilizado o valor *org.eclipse.jdt.ui.javaEditorScope*, que define que somente é para ser considerado o evento (pressionamento da tecla #) durante a edição de arquivos código fonte *Java*. O atributo *sequence* da *tag key* (linha 89) define qual tecla ou qual sequência de teclas deve ser pressionada pelo usuário para

[15] Mais sobre os *contextos* do *Eclipse* pode ser encontrado em (ECLIPSE FOUNDATION, 2016b)

considerar que o evento ocorreu. No caso, é definido que o evento ocorre quando o usuário pressionar a tecla #. Com essas configurações, é definido que o método *execute* da classe *br.ufscar.ppgccs.ips.commands.ActiveUI* é executado sempre que o usuário pressionar a tecla # durante a edição de um arquivo de código fonte *Java* na *IDE Eclipse*.

O método *execute* da classe *ActiveUI* é responsável por identificar a necessidade da exibição da área de texto para que o usuário digite as palavras-chaves do *code snippet* desejado (item 1 destacado na Figura 54). A área de texto somente é exibida ao usuário somente quando o caractere # é digitado em uma linha vazia e dentro de um método. Essas verificações são necessárias para evitar que o *CSFinder* seja ativado de forma indesejada. Por exemplo, fora de um método, onde não há a necessidade da utilização de *code snippets*. Ou ainda, por exemplo, quando o usuário estiver digitando o conteúdo de uma *String* contendo o caractere #. Opcionalmente, o caractere # pode estar precedido de um identificador de objeto (variável local ou atributo de classe). Se o caractere estiver precedido de um identificador de objeto, o método *execute* da classe *ActiveUI* verifica também se o identificador de objeto está devidamente declarado e se é possível determinar seu tipo. Caso não seja possível identificar o tipo do identificador, a área de texto não é exibida e o evento é ignorado. Essas verificações são realizadas utilizando a *AST* (ECLIPSE FOUNDATION, 2016a) — *Abstract Syntax Tree* — do código fonte *Java* sendo editado pelo usuário e considerando a posição do cursor de texto. Através da *AST*, é possível identificar se o identificador de objeto que precede o caractere # é um identificador de objeto disponível no escopo do código fonte sendo editado na posição onde se encontra o cursor de texto e é possível também obter o tipo do identificador. A *AST* é obtida através da classe *ASTParser* da *API JDT Core*[16], que é uma *API* que integra o *JDT*[17] — *Eclipse Java Development Tools*. O *JDT* é um conjunto de *plug-ins* da *IDE Eclipse* responsável por disponibilizar um conjunto de ferramentas que suporta o desenvolvimento de projetos *Java*. Entre as principais ferramentas do *JDT*, é possível citar um depurador, um editor de códigos fontes *Java* e um assistente de código que sugere elementos (métodos, construtores e campos de classes) ao programador durante a edição de códigos fontes. O *JDT* disponibiliza *APIs* que permitem acessar as funcionalidades dessas ferramentas. Entre essas *APIs*, está a *API JDT Core*, que contém uma série de classes que permitem acessar e fazer a manipulação de um código fonte *Java* sendo editado pelo usuário na *IDE Eclipse*.

Logo que as verificações são realizadas pelo método *execute* da classe *ActiveUI* e é verificado que o caractere # digitado refere-se a um comando de ativação do *CSFinder*, é apresentado um campo de texto para que o usuário digite as palavras-chaves do *code snippet* desejado (item 1 destacado Figura 54). O campo de texto é implementado através de uma classe *Java* denominada *SearchBoxUI*, que pertence ao projeto de implementação

16 <http://www.eclipse.org/jdt/core/>
17 <http://www.eclipse.org/jdt/>

do módulo *CSFinder* e que encapsula um objeto da classe *StyledText*[18]. A classe *StyledText* é uma classe da biblioteca de componentes gráficos *SWT*[19] e implementa um componente gráfico que representa um campo de texto.

Assim que o usuário digita as palavras-chaves no campo de texto implementado pela classe *SearchBox*, uma busca no índice *CSIndex* é realizada. Para realizar buscas no índice, o *Lucene* utiliza uma abstração denominada *termo*, que é representado pela classe *Term*[20]. Um *termo* é um par formado pelo nome de uma coluna e um valor (normalmente uma palavra) que pode ser encontrado em algum registro no índice na coluna. Por exemplo, o registro apresentado na Tabela 8 possui o *termo* <*keywords, aggregate*>, pois na coluna *keywords* há a palavra *aggregate* (assim como também há a palavra *field*). A *API Lucene* possui também uma classe denominada *TermQuery*[21], que estende a classe *Query*[22] e que representa uma *query* para um determinado *termo*. Por exemplo, para criar uma *query* que retorne todos os registros contendo o *termo* <*keywords, aggregate*>, basta criar um objeto do tipo *Term* que represente o *termo* <*keywords, aggregate*> e depois utilizar esse objeto para criar um objeto do tipo *TermQuery*. *Queries* para *termos*, representadas por objetos do tipo *TermQuery*, podem ser combinadas a fim de criar *queries* mais complexas. Essa combinação é realizada criando um objeto da classe *BooleanQuery*[23], que também é um classe da *API Lucene* que estende a classe *Query*. A busca realizada pelo *CSFinder*, após o usuário digitar as palavras-chaves no campo de texto (item 1 destacado Figura 54), ocorre através da criação de um objeto do tipo *BooleanQuery* que combina objetos do tipo *TermQuery*. Os objetos do tipo *TermQuery* representam *queries* para os seguintes *termos*: um *termo* para a coluna *workspace* contendo o valor correspondente ao diretório do *workspace* da instância da *IDE Eclipse* que o usuário esteja utilizando; um *termo* para a coluna *project* contendo o valor correspondente ao nome do projeto *Java* que contém o código fonte que o usuário esteja editando; um *termo* para a coluna *keywords* para cada palavra-chave digitada pelo usuário; um *termo* para a coluna *import* para cada *import* no código fonte *Java* sendo editado pelo usuário; um *termo* para a coluna *param* para cada tipo de identificador de objeto disponível no escopo do código fonte sendo editado e um *termo* para a coluna *associated_class* contendo o tipo do identificador de objeto que precede o caractere # na ativação do *CSFinder*.

Tomando como exemplo o cenário de uso apresentado na Figura 54, que o usuário ativa o *CSFinder* para a variável local *table* na linha 19 durante a edição do código fonte *Class.java*, supondo que o *workspace* da *IDE Eclipse* esteja configurado para o diretório

[18] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/custom/StyledText.html>
[19] <https://www.eclipse.org/swt/>
[20] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/index/Term.html>
[21] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/TermQuery.html>
[22] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Query.html>
[23] <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/BooleanQuery.html>

`<C:\eclipse\workspace>` e que o arquivo *Class.java* pertença a um projeto *Java* de nome *SQLDataAPIClient*, para realizar a busca dos *code snippets* no índice, são criados objetos do tipo *TermQuery* conforme listados na Tabela 9.

Tabela 9 – Objetos do tipo *TermQuery*

#	Coluna	Valor	Obrigatório
1	*workspace*	*C:\eclipse\workspace*	Sim
2	*project*	*SQLDataAPIClient*	Sim
3	*keywords*	*aggregate*	Não
4	*keywords*	*field*	Não
5	*import*	*org.dataapiorg.sqldataapi.Table*	Não
6	*import*	*org.dataapiorg.sqldataapi.Row*	Não
7	*param*	*class(long)*	Não
8	*param*	*class(java.lang.String)*	Não
9	*param*	*class(org.dataapiorg.sqldataapi.Connection)*	Não
10	*param*	*class(org.dataapiorg.sqldataapi.Table)*	Não
11	*param*	*class(com.johncompany.sqldataclient.ArithmeticMeanAggregator)*	Não
12	*param*	*role(org.dataapiorg.sqldataapi.Aggregator)*	Não
13	*associated_class*	*org.dataapiorg.sqldataapi.Table*	Sim

Fonte: próprio autor

Para a coluna *keywords*, é criado um objeto *TermQuery* para cada palavra-chave digitada pelo usuário. No exemplo considerado são criados dois objetos *TermQuery*: um para a palavra-chave *aggregate* (item 3 da Tabela 9) e outro para a palavra-chave *field* (item 4 da Tabela 9). É criado um objeto *TermQuery* para a coluna *import* para cada *import* declarado no código fonte sendo editado (itens 5 e 6 na Tabela 9), um para o *import* declarado na linha 3 e outro para o *import* declarado na linha 4 do arquivo *Class.java* (ver Figura 54). É criado um *TermQuery* para a coluna *param* para cada identificador de objeto presente no escopo do código fonte sendo editado (itens 7, 8, 9, 10 e 11 da Tabela 9). Para cada tipo, é criado um único objeto *TermQuery*. Por exemplo, na Tabela 9 para a coluna *param*, é listado apenas um objeto *TermQuery* para o tipo *java.lang.String* (item 8), embora haja três identificadores de objetos do tipo *java.lang.String* disponíveis no escopo do código do arquivo *Class.java* apresentado na Figura 54 (os campos de classe *MYSQL_DATABASE_URL*, *subject* e *teacherName* declarados nas linhas 8, 11 e 12 respectivamente). Os identificadores de objetos (variáveis locais ou atributos de classes) e seus respectivos tipos são obtidos através da análise da *AST* do código fonte *Java* sendo editado pelo usuário, que é obtida através da classe *ASTParser* da *API JDT Core*. Os itens 11 e 12 da Tabela 9 referem-se a variável local *arithmeticMeanAggregator* declarada na linha 18 do arquivo *Class.java*. A variável é do tipo da classe *com.johncompany.sqldataclient.ArithmeticMeanAggregator*, que é uma classe associada ao *adapter org.dataapiorg.sqldataapi.Aggregator*. Por esse motivo, é criado o

TermQuery referente ao item 12 da Tabela 9. Para a coluna *associated_class* (item 13 na Tabela 9), é criado um objeto *TermQuery* contendo o tipo do identificador de objeto que precede o caractere # na ativação do *CSFinder*, que no exemplo é a variável local *table* cujo tipo é *org.dataapiorg.sqldataapi.Table* (declarada na linha 17 no arquivo *Class.java* apresentado na Figura 54 e utilizada na linha 19 para ativar o *CSFinder*). Esses objetos do tipo *TermQuery*, listados na Tabela 9, são então combinados através de um objeto do tipo *BooleanQuery* para a execução da *query* que retorna os *code snippets* indexados em *CSIndex*.

Na combinação dos objetos do tipo *TermQuery* através de um objeto *BooleanQuery*, é possível especificar para cada objeto *TermQuery* se ele é obrigatório ou não (veja a terceira coluna da Tabela 9). Se um objeto *TermQuery* for obrigatório, somente são retornados registros no índice contendo o *termo* especificado pelo objeto. Objetos do tipo *TermQuery* para as colunas *workspace*, *project* e *associated_class* são obrigatórios. Ou seja, os registros retornados na busca devem conter os termos especificados pelos itens 1, 2 e 13 da Tabela 9. Embora outros termos especificados na busca não sejam obrigatórios (para as colunas *keywords*, *import* e *param*), eles contribuem para o ranqueamento dos registros retornados. Esse ranqueamento é realizado pelo *Lucene* através de um algoritmo baseado no modelo de ranqueamento *VSM — Vector Space Model* (MANNING; RAGHAVAN; SCHÜTZE, 2008, p. 100), que considera um registro no índice como um vetor. Cada dimensão do vetor corresponde a um *termo* que pode estar presente no registro ou não. Se o *termo* estiver presente no índice, o valor do vetor na dimensão correspondente ao *termo* possui um valor positivo, caso contrário possui o valor zero. No *VSM*, *queries* são também consideradas vetores. Se uma *query* especifica um determinado *termo* (através de um objeto do tipo *TermQuery*), o valor do vetor na dimensão correspondente ao *termo* possui um valor positivo. Caso a *query* não especifique o termo (não contenha um objeto do tipo *TermQuery* para o *termo*), o valor do vetor na dimensão correspondente ao *termo* possui o valor zero. Os registro retornados na busca são então ranqueados através da similaridade entre o vetor que representa o registro no índice e o vetor que representa a *query*. O valor utilizado para representar a similaridade entre dois vetores é o cosseno do ângulo formado entre os vetores e os registros cujos vetores são mais similares ao vetor que representa a *query* são melhores ranqueados. Mais detalhes sobre o algoritmo utilizado pelo *Lucene* para ranquear os registros retornados em uma busca podem ser encontrados em (APACHE SOFTWARE FOUNDATION, 2016b).

Para os dez primeiros registros retornados na busca, cada registro é convertido para um objeto do tipo da classe *CodeSnippet*, que é uma classe pertencente ao projeto de implementação do *IPS* e que contém todas as informações de um *code snippet*. A conversão de um registro retornado na busca para um objeto do tipo *CodeSnippet* é realizada através de uma análise sintática do conteúdo do campo *xml_content* do registro, que contém o trecho do arquivo *XCSPT* que define o *code snippet* (veja Tabela 7). Essa análise sintática

é realizada pela classe *CSXMLParser* que, baseada no modelo *DOM* do conteúdo do campo *xml_content*, cria um objeto do tipo *CodeSnippet*. Os dez primeiros *code snippets* retornados na busca são apresentados ao usuário através de uma tabela (item 2 destacado Figura 54). A tabela é implementada pela classe *ProposalsListUI*, que pertence ao projeto de implementação do módulo *CSFinder* e que encapsula um objeto da classe *Table*[24]. *Table* é uma classe da *API SWT* que implementa um componente gráfico para apresentação de tabelas. Cada linha da tabela (item 2 destacado Figura 54) é associada a um objeto do tipo *CodeSnippet* e é exibido na linha o nome do *code snippet* e as palavras-chaves associadas ao *code snippet*.

Assim que usuário seleciona um dos *code snippets* apresentados na tabela, é exibido uma janela contendo todas as informações do *code snippet* selecionado (item 3 destacado na Figura 54). A janela é implementada pela classe *CodeSnippetActivatorUI*, que pertence ao projeto de implementação do módulo *CSFinder* e que estende a classe *Composite*[25]. *Composite* é uma classe da *API SWT* que representa um componente gráfico que pode conter outros componentes gráficos, como campos de texto, rótulos, listas, tabelas e botões. Além de exibir as informações do *code snippet* selecionado pelo usuário, a janela implementada pela classe *CodeSnippetActivatorUI* exibe para cada parâmetro do *code snippet* uma lista para que o usuário selecione o argumento do parâmetro (item 4 destacado na Figura 54). Cada lista é populada com identificadores de objetos disponíveis do escopo do código fonte sendo editado pelo usuário que sejam compatíveis com o parâmetro. Para os parâmetros do tipo *class*, são exibidos identificadores disponíveis no escopo do código sendo editado que sejam do tipo da classe associada ao parâmetro. Para os parâmetros do tipo *role*, são exibidos identificadores disponíveis no escopo do código sendo editado que sejam do tipo de uma classe associada (através da *annotation ClassLike*) ao *adapter* associado ao parâmetro do *code snippet*. Os identificadores de objetos (variáveis locais ou atributos de classes) e seus respectivos tipos são obtidos através da análise da *AST* do código fonte *Java* sendo editado pelo usuário, que é obtida através da classe *ASTParser* da *API JDT Core*.

O usuário seleciona os argumentos para os parâmetros do *code snippet* (item 4 destacado Figura 54). Conforme o usuário seleciona os argumentos, um método da classe *CodeSnippetActivatorUI* atualiza o corpo do *code snippet*, substituindo as referências do parâmetro no corpo pelo argumento selecionado. Após o usuário selecionar os argumentos para os parâmetros, ele pressiona o botão *Insert* (item 2 destacado Figura 54) e a classe *CodeSnippetActivatorUI* substitui a linha que ativou o *IPS* (linha 19 no trecho de código apresentado na Figura 54) pelo trecho de código gerado (resultado do corpo do *code*

[24] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Table.html>

[25] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Composite.html>

snippet com as referências dos parâmetros substituídos pelos respectivos argumentos). Essa substituição ocorre através de um objeto do tipo *IDocument*[26] que representa o conteúdo textual do arquivo código fonte sendo editado. *IDocument* é uma interface da *API JFace*[27], que é uma *API* de componentes gráficos construída sobre a *API SWT*. Um editor de código fonte *Java* na *IDE Eclipse* é representado por um objeto do tipo da classe *JavaEditor*[28], que pertence ao *JDT*. Cada arquivo código fonte *Java* aberto na *IDE Eclipse* possui um objeto do tipo *JavaEditor* correspondente. A classe *JavaEditor* por sua vez encapsula um objeto do tipo *SourceViewer* que pode ser obtido através do método *getViewer*. *SourceViewer* é uma classe pertencente à *API JFace* que implementa um componente gráfico que pode apresentar e editar códigos fontes. O conteúdo do código fonte apresentado por um objeto do tipo *SourceViewer* é representado por um objeto do tipo *IDocument* encapsulado pela classe *SourceViewer* e que pode ser obtido através do método *getDocument*. Portanto, uma vez obtido o objeto do tipo *JavaEditor* correspondente ao arquivo código fonte *Java* sendo editado na *IDE Eclipse*, é possível obter o objeto do tipo *IDocument* para manipular o conteúdo textual do código fonte.

Na implementação do *CSFinder*, para obter o objeto do tipo *JavaEditor* correspondente ao arquivo código fonte *Java* sendo editado pelo usuário, é utilizada uma classe denominada *HandlerUtil*[29] que pertence à *API Eclipse Platform*[30], *API* da plataforma padrão do *Eclipse*. Essa classe possui um método estático denominado *getActiveEditor*, que retorna um objeto que representa o editor ativo no momento na *IDE Eclipse*. Um editor ativo na *IDE Eclipse* é o editor que está com o foco do cursor de texto e, portanto, o usuário está o utilizando para editar algum arquivo. Como mencionado no início desta Seção, assim que o usuário digita o caractere # durante a edição de um código fonte *Java* na *IDE Eclipse*, é executado o método *execute* da classe *ActiveUI*, classe que pertence ao projeto de implementação do *CSFinder*. O método *execute* aceita como parâmetro um objeto do tipo *ExecutionEvent*, que representa o evento que aciona o método *execute*. O método *getActiveEditor* da classe *HandlerUtil* também aceita como parâmetro um objeto do tipo *ExecutionEvent* e retorna um objeto que representa o editor de onde o evento representado pelo objeto do tipo *ExecutionEvent* foi disparado. Se o evento foi disparado de um editor de código fonte *Java*, o método *getActiveEditor* retorna um objeto do tipo *JavaEditor*. Dessa forma, é possível obter o objeto do tipo *JavaEditor* correspondente ao arquivo código fonte *Java* sendo editado onde o usuário tenha digitado o caractere #.

[26] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/text/IDocument.html>
[27] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/guide/jface.htm>
[28] <https://www.cct.lsu.edu/~rguidry/ecl31docs/api/org/eclipse/jdt/internal/ui/javaeditor/JavaEditor.html>
[29] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/handlers/HandlerUtil.html>
[30] <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/overview-summary.html>

Para isso, basta passar o objeto do tipo *ExecutionEvent* recebido pelo método *execute* da classe *ActiveUI* para o método *getActiveEditor* da classe *HandlerUtil*. O objeto do tipo *JavaEditor* retornado pelo método *getActiveEditor* é então utilizado para obter o objeto do tipo *SourceViewer*, que por sua vez é utilizado para obter o objeto do tipo *IDocument* para realizar a manipulação do texto do código fonte *Java* sendo editado. Na Figura 56, é exibido um trecho de código que exemplifica a obtenção do objeto do tipo *IDocument* a partir do objeto do tipo *ExecutionEvent* passado como parâmetro do método *execute*.

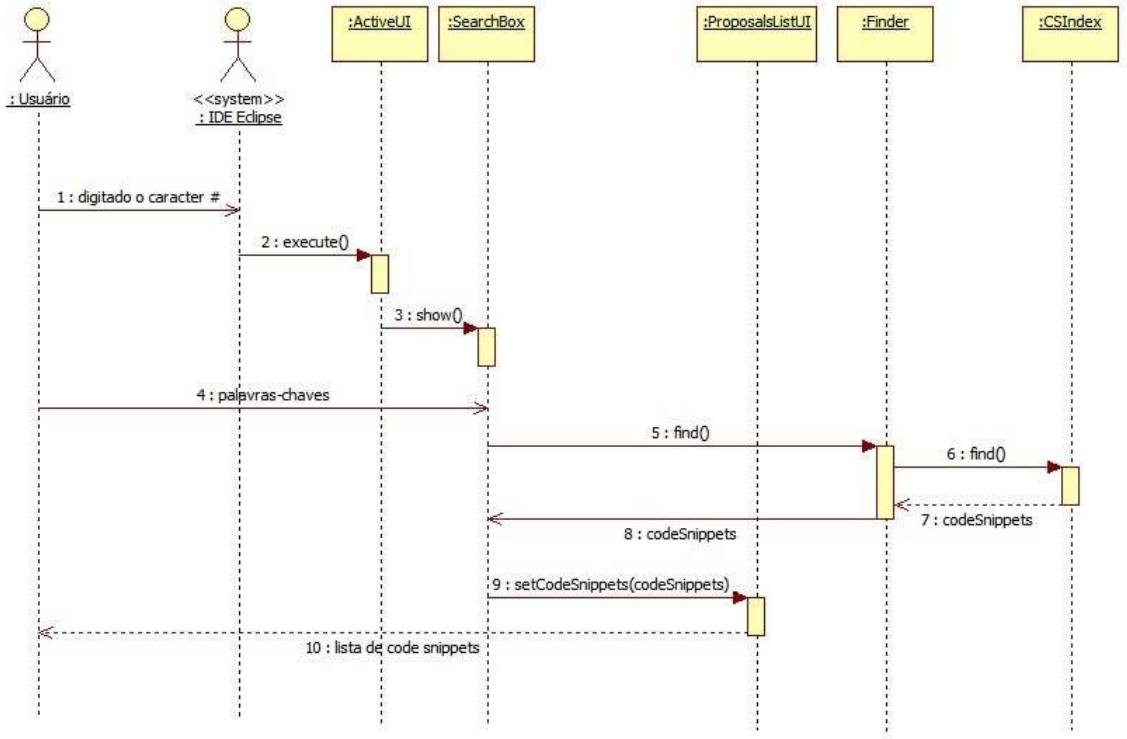
No *CSFinder*, após o usuário pressionar o botão *Insert* (item 5 da Figura 54) da janela implementada pela classe *CSActivatorUI*, janela que apresenta todas as informações do *code snippet* selecionado (item 3 da Figura 54), o método *replace* do objeto do tipo *IDocument* relacionado ao código fonte sendo editado pelo usuário é utilizado para substituir a linha que aciona o *CSFinder* (linha 19 do código fonte *Class.java* apresentado na Figura 54 onde o usuário digita o caractere #) pelo trecho de código de código gerado pelo *code snippet* (corpo do *code snippet* com as referências dos parâmetros no corpo substituídos pelos respectivos argumentos selecionados pelo usuário). O método *replace* de *IDocument* substitui uma determinada região do texto por um novo conteúdo. Ele possui três parâmetros: *offset* e *length* do tipo *int*; e *text* do tipo *String*. *text* define o novo conteúdo, *offset* e *length* definem a região no texto que deve ser substituída pelo novo conteúdo. A posição do início da região no texto é definida pelo parâmetro *offset* e *lenght* define o tamanho da região.

Figura 56 – Trecho de código para obter objeto do tipo *IDocument*

```
1  package br.ufscar.ppgccs.ips;
2
3  import org.eclipse.core.commands.AbstractHandler;
4  import org.eclipse.core.commands.ExecutionEvent;
5  import org.eclipse.core.commands.ExecutionException;
6  import org.eclipse.jdt.internal.ui.javaeditor.JavaEditor;
7  import org.eclipse.jface.text.IDocument;
8  import org.eclipse.jface.text.source.SourceViewer;
9  import org.eclipse.ui.handlers.HandlerUtil;
10
11 public class ActiveUI extends AbstractHandler {
12
13     @Override
14     public Object execute(ExecutionEvent event) throws ExecutionException {
15         JavaEditor editorPart = (JavaEditor) HandlerUtil.getActiveEditor(event);
16         SourceViewer sourceViewer = (SourceViewer) editorPart.getViewer();
17         IDocument document =  sourceViewer.getDocument();
18
19     }
20
21 }
```

Fonte: próprio autor

Figura 57 – Diagrama de sequência do *CSFinder* — listagem



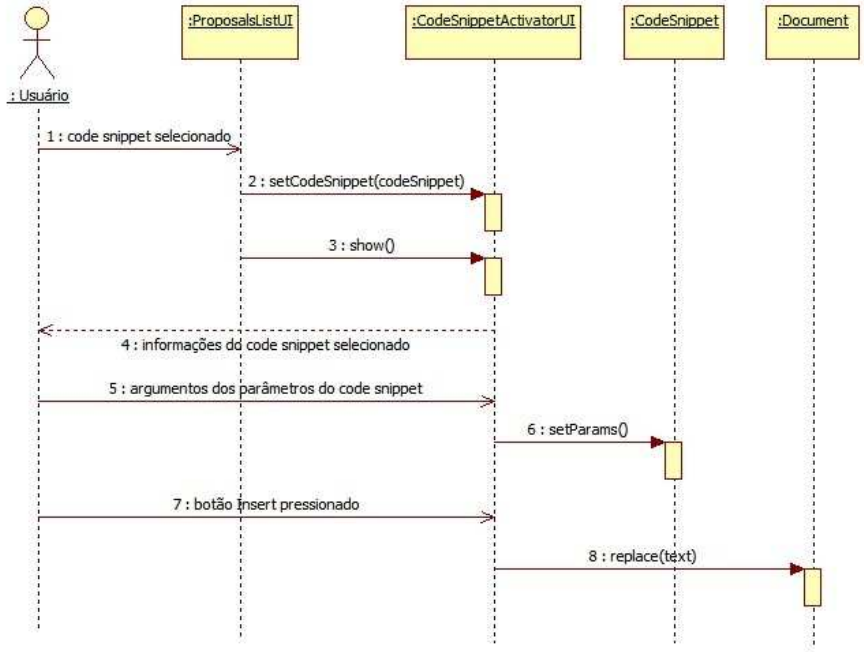
Fonte: próprio autor

A fim de sumarizar as etapas descritas nesta Subseção, nas Figuras 57 e 58 são apresentados diagramas de sequência *UML*. Na Figura 57, são apresentadas as interações entre os componentes até a etapa que lista para o usuário os dez primeiros *code snippets* retornados na busca, apresentados na lista implementada pela classe *ProposalsListUI* que é destacada como item 2 na Figura 54. Já na Figura 58, são apresentadas as interações entre os componentes a partir do momento que o usuário seleciona um dos *code snippets* listados até o momento que o usuário pressiona o botão *Insert* da janela implementada pela classe *CodeSnippetActivatorUI* (item 5 destacado na Figura 54), e a linha que aciona o *CSFinder* é substituída pelo trecho de código gerado pelo *code snippet*.

2.3.4 Módulo *CSPTE*

O *CSPTE*, *Code SniPpet Editor*, é o módulo responsável por auxiliar a edição de arquivos *XCSPT*. Constitui-se de uma interface gráfica integrada à *IDE Eclipse* que torna a edição dos arquivos *XCSPT* mais intuitiva. Durante a abertura de um arquivo *XCSPT* para edição, o módulo realiza a análise sintática do arquivo *XCSPT* e apresenta as informações dos *code snippets* definidos no arquivo (veja Figura 59). A interface gráfica é implementada pela classe *CodeSnippetsEditorView*, classe que pertence ao projeto de

Figura 58 – Diagrama de sequência do *CSFinder* — acionamento



Fonte: próprio autor

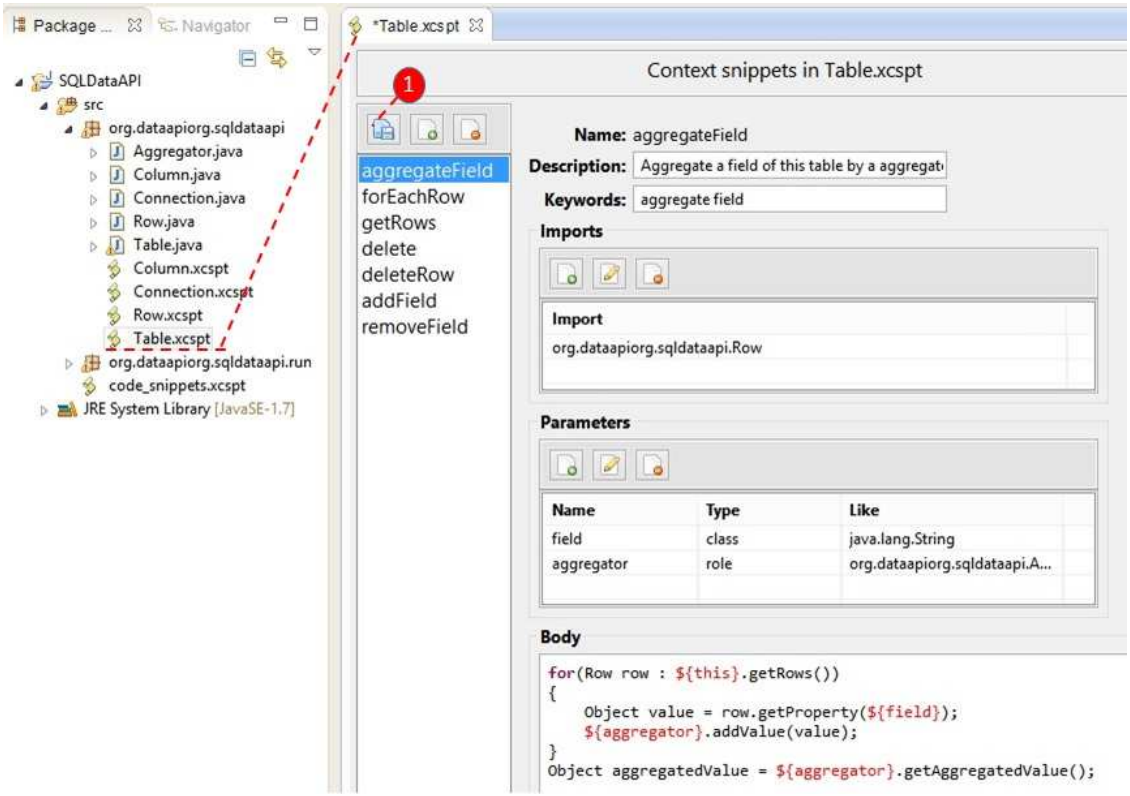
implementação do *CSPTE* e que estende a classe *Composite*[31]. *Composite* é uma classe da *API SWT* que representa um componente gráfico que pode conter outros componentes gráficos, como campos de texto, rótulos, listas, tabelas e botões. A análise sintática do arquivo *XCSPT*, realizada na abertura do arquivo para edição, é realizada pela classe *CSXMLParser*, classe pertencente ao projeto de implementação do *IPS* e que utiliza a *API* padrão da plataforma *Java* para manipulação de arquivos *XML*, *JAXP*[32]. A classe *CSXMLParser* obtém o conteúdo do arquivo *XCSPT* através do modelo *DOM*[33] e, baseado no modelo *DOM*, ela cria, para cada *code snippet* definido no arquivo *XCSPT*, um objeto do tipo da classe *CodeSnippet*. A classe *CodeSnippet* é uma classe pertencente ao projeto de implementação do *IPS* e que contém todas as informações de um *code snippet*. Os objetos do tipo *CodeSnippet* representando cada *code snippet* definido no arquivo *XCSPT* são inseridos em uma lista em memória e suas informações são apresentadas na janela implementada pela classe *CodeSnippetsEditorView*. As informações na janela implementada pela classe *CodeSnippetsEditorView* (Figura 59) são sincronizadas com essa lista de objetos do tipo *CodeSnippet* em memória. Se o usuário altera, através da janela, alguma informação

31 <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Composite.html>

32 <https://docs.oracle.com/javase/tutorial/jaxp/>.

33 *DOM*, *Document Object Model*, modelo padronizado pela *W3C* utilizado como uma representação intermediária, em memória, do conteúdo de arquivos *HTML* e *XML* <http://www.w3.org/DOM/Overview.html>.

Figura 59 – Janela do *CSPTE*



Fonte: próprio autor

de algum *code snippet*, o objeto do tipo *CodeSnippet* correspondente na lista é também alterado. Se um novo *code snippet* é criado, baseado nas informações do novo *code snippet* preenchidas na janela pelo usuário, um novo objeto do tipo *CodeSnippet* é adicionado à lista em memória. Se um *code snippet* é removido, o *code snippet* correspondente é removido da lista em memória. Quando o usuário pressiona o botão salvar (item 1 destacado na Figura 59), o conteúdo da lista em memória é transformado em uma representação *XML* e esse conteúdo é salvo no arquivo *XCSPT* em edição. Essa transformação da lista de objetos do tipo *CodeSnippet* em memória para uma representação em *XML* é realizada pelo método *marshalAll* da classe *CSXMLParser*, que recebe como parâmetro uma lista de objetos do tipo *CodeSnippet* e retorna uma *String* com o conteúdo *XML* representando os *code snippets* na lista.

2.4 Trabalhos relacionados

Ferramentas *CSACT*

O *IPS* é uma ferramenta *CSACT*, que são ferramentas que localizam ou sugerem trechos de código ou componentes de uma *API* para o usuário e são abordadas no Capítulo

1 desta dissertação. As Ferramentas *CSACTs* são divididas em dois grandes grupos. Um grupo de ferramentas que utilizam técnicas de busca e mineração de dados (ZHONG et al., 2009; BRUCH; MONPERRUS; MEZINI, 2009; KIM et al., 2010; MONTANDON et al., 2013) e outro grupo de ferramentas que utilizam *templates de código* (WIGHTMAN et al., 2012; ONEY; BRANDT, 2012), padrões de código cadastrados manualmente muitas vezes denominados também de *code snippets*. A principal diferença é que os *templates de código* são codificados por especialistas de uma determinada *API* com o propósito de ensinar e auxiliar a utilização da *API*, enquanto que ferramentas que utilizam técnicas de busca e mineração de dados extraem trechos de código automaticamente de repositórios de códigos fontes que não são codificados com o propósito de ensinar a utilização da *API*. Os repositórios de códigos fontes, utilizados pelas ferramentas baseadas em busca e mineração de dados, podem ser desde de toda a *Web*, através da utilização de um provedor de busca como o *Google*, ou um repositório mais restrito como uma base de códigos fontes privada de uma empresa. Por sua vez, os trechos de código cadastrados manualmente, como os *templates de código*, são confeccionados exclusivamente para ensinar e auxiliar a utilização de uma *API*, logo eles possuem uma tendência de ter uma qualidade maior; com menos erros de compilação, com menos erros de execução, são mais concisos e mais fáceis de serem entendidos. No entanto, nem sempre os *templates de código* estão disponíveis, pois, como são manualmente confeccionados, há um custo envolvido tanto na criação como na manutenção.

Ferramentas *CSACTs* podem ser integradas a uma *IDE*, como um *plug-in*, ou serem independentes de *IDE*. Ferramentas integradas a uma *IDE* possuem a vantagem de poderem sugerir a utilização de trechos de códigos mais relevantes ao contexto do código fonte em edição (ZHONG et al., 2009; BRUCH; MONPERRUS; MEZINI, 2009; BRANDT et al., 2010; WIGHTMAN et al., 2012), por tanto são orientadas à tarefa que o programador esteja trabalhando. Por outro lado, ferramentas não integradas a uma *IDE* são orientadas à documentação, ou seja, têm como objetivo estender a documentação da *API* e sua vantagem é justamente serem independentes de *IDE* (KIM et al., 2010; MONTANDON et al., 2013).

Ferramentas de *templates de código*

Templates de código são atualmente suportados nativamente pela *IDE Eclipse* (ECLIPSE FOUNDATION, 2015). Ao digitar uma palavra, durante a edição de um código fonte *Java*, e acionar o *autocompletador de código*, o *Eclipse* sugere *templates de código* com nomes que começam com a palavra digitada. Ao acionar um *template de código*, o *Eclipse* auxilia a adaptação do trecho de código do *template* sugerindo identificadores no escopo do código fonte em edição compatíveis com os respectivos parâmetros do *template*. O sistema de *template* do *Eclipse* também realiza de forma automática as importações de pacotes

necessários para executar o trecho de código gerado. Wightman et al. (2012) apresenta uma ferramenta denominada *SnipMatch*. O *SnipMatch* estende a ideia utilizada no *Eclipse* e propõe um sistema de busca de *templates* baseado em palavras-chaves. A ferramenta filtra os *templates* mais relevantes e propõe argumentos para os parâmetros. Oney e Brandt (2012) propõem a utilização de *templates de código* denominados de *Codelets*. Os *Codelets* são *templates de código* associados a uma interface gráfica com componentes gráficos que auxiliam o usuário a editar os argumentos dos parâmetros do *template de código*.

O *IPS* é uma ferramenta de *templates de código* integrada à *IDE Eclipse*. Uma das novidades apresentadas pelo *IPS* é a possibilidade de associar um *template de código* a uma classe *Java*, o que possibilita localizar um *template de código* a partir de um identificador de objeto disponível no escopo do código fonte sendo editado. Essa funcionalidade foi influenciada principalmente pela existência dos *autocompletadores de código*, que são apresentados na Seção 1.1.3. Um *autocompletador de código* possibilita localizar um método a partir de um identificador de objeto disponível no escopo do código fonte sendo editado. Esse recurso é bastante utilizado pelos programadores (MURPHY; KERSTEN; FINDLATER, 2006, p. 80). A hipótese que motivou a implementação de *templates de código* associados a classes *Java* foi que, para economizar tempo e ser mais produtivo, um programador pode querer que, a partir de um identificador de objeto no escopo do código fonte sendo editado, seus códigos fontes sejam completados com trechos de código e não somente por métodos.

Ferramentas de *templates de código* possuem suporte a parâmetros. Os parâmetros permitem que os *templates de código* sejam reaproveitados e adaptados em contextos diferentes. Os parâmetros são definidos na criação dos *templates de código* e são substituídos pelos argumentos passados durante a ativação. Os sistemas nativos de *templates de código* das *IDEs* atuais, como as *IDEs Eclipse* (ECLIPSE FOUNDATION, 2015), *Visual Studio* (MICROSOFT CORPORATION, 2014) e o *Netbeans* (ORACLE CORPORATION, 2014), possuem suporte a parâmetros associados a um tipo. A associação de um parâmetro de um *template de código* a um determinado tipo significa que o parâmetro do *template* aceita como argumento identificadores de objetos (variáveis locais ou atributos de classe) do tipo associado que estiverem disponíveis no escopo do código fonte em edição durante a ativação do *template*. O *IPS* introduz uma nova abordagem denominada de "parâmetros do tipo *role*", que estende o conceito dos parâmetros associados a um tipo presente nas ferramentas de *templates de código* atuais. Os parâmetros do tipo *role* têm como objetivo aceitar argumentos que seriam a princípio incompatíveis com uma determinada classe, porém que sigam um determinado padrão esperado. Parâmetros do tipo *role* auxiliam a integração entre objetos de classes pertencentes a uma determinada *API* com objetos de classes pertences ao projeto do usuário da *API*. Um desenvolvedor de uma *API* pode, através dos parâmetros do tipo *role*, escrever um *template de código* com um parâmetro que aceitará como argumento identificadores de objetos (variáveis locais e atributos de

classes) cujo tipo seja uma classe desconhecida pelo autor do *template*. Por exemplo, um especialista de uma *API* para processamento de grafos pode escrever um *template* que implemente um algoritmo de busca em grafos que dependa de uma estrutura de dados do tipo pilha. A classe que implementará a estrutura de dados do tipo pilha não precisa ser conhecida para confeccionar o *template*. Durante a ativação do *template*, as referências no corpo do *template* para a pilha são então adaptadas para uma pilha implementada pelo usuário do *template*.

O *IPS* suporta também a renomeação automática de variáveis. Embora no *Eclipse* (ECLIPSE FOUNDATION, 2015) exista a diretiva *${newName}*, que pode ser utilizada para gerar um nome para uma variável sem conflito de nomes, no *IPS* o criador do *template de código* não precisa preocupar-se em utilizar nenhuma diretiva de integração. O *IPS* encarrega-se de verificar os nomes e de renomear as variáveis automaticamente. Outro ponto que é possível destacar no *IPS* é a distribuição dos *templates de código* através de arquivos *JAR*. Na plataforma *Java*, um arquivo *JAR* é utilizado para empacotar componentes de uma *API* em único arquivo e é utilizado pelo desenvolvedor de uma *API* para distribuir a *API* aos usuários. O *IPS* define seus *templates de código* em arquivos com a extensão *XCSPT*, que são arquivos com conteúdo *XML*. Um conjunto de arquivos *XCSPT* contendo os *templates de código* que auxiliam a utilização de uma *API* podem ser distribuídos no mesmo arquivo *JAR* que distribui a *API*. Isso possibilita que criadores de uma *API* distribuam, juntamente com a *API*, *templates de código* que auxiliem a utilização da própria *API*. No trabalho apresentado por Oney e Brandt (2012), os *templates de código*, os quais são denominados no trabalho de *Codelets*, também são definidos em arquivos *XML*, no entanto não é abordado como esses *templates de código* seriam distribuídos entre os usuários, apenas é citado que implementações futuras da ferramenta apresentada no trabalho poderá distribuir esses arquivos *XML* para os programadores de um projeto de software através de um *sistema de controle de versão*. Já no trabalho de Wightman et al. (2012), que apresenta a ferramenta *SnipMatch*, é proposto uma arquitetura cliente-servidor. Os *templates de código* são armazenados em um servidor e o *plug-in* que é executado na *IDE* do usuário no lado cliente é responsável por recuperar os *templates de código* armazenados no servidor.

Recomendador

O *recomendador de templates de código* é uma funcionalidade do *IPS* que recomenda *templates de código* ao usuário enquanto este está editando algum código fonte na *IDE*, sem a necessidade de uma ação explícita por parte do usuário. São recomendados *templates de código* para os identificadores de objetos (variáveis locais ou campos de classes) disponíveis no escopo do código fonte em edição. A lista de *templates de código* recomendados é constantemente atualizada conforme o conteúdo do código fonte em edição é alterado.

Outros trabalhos apresentaram ferramentas *CSACTs* que exploram a recomendação de elementos ao usuário sem uma ação explícita por parte do usuário. Por exemplo, *CodeBroker* (YE; FISCHER; REEVES, 2000) recomenda métodos *Java* baseado nas palavras presentes em um bloco de comentários de um código fonte sendo editado. *Rascal* (MCCAREY; CINNÉIDE; KUSHMERICK, 2005) recomenda métodos *Java* baseado no conjunto de métodos já utilizados na classe atualmente em edição. *OCompletion* é um *autocompletador de código* da linguagem *Smalltalk* que recomenda métodos para um identificador de objeto sem a necessidade da utilização de um atalho de teclado (ex., *CONTROL + ESPAÇO*). Essas ferramentas citadas diferenciam-se do *IPS* por apresentarem componentes de uma *API*, mais especificamente métodos. O *IPS* recomenda trechos de código. Em relação a este aspecto, de recomendar elementos ao usuário sem a necessidade de uma ação explícita, a ferramenta que mais assemelha-se ao *IPS* é a ferramenta *Redprint* (BHARDWAJ; LUCIANO; KLEMMER, 2011), pois também recomenda trechos de códigos. No entanto, o *IPS* diferencia-se por recomendar trechos de código, no caso *templates de código*, que são associados às classes dos identificadores de objetos disponíveis no escopo do código fonte sendo editado. Enquanto o *Redprint* realiza uma busca por trechos de código baseado nas palavras presentes na linha onde se encontra o cursor de texto, o *IPS* utiliza os tipos dos identificadores de objetos (variáveis locais ou campos de classes) disponíveis no escopo do código fonte. No *IPS*, cada trecho de código é recomendado para um identificador de objeto e, se o usuário seleciona o trecho de código recomendado para um determinado identificador de objeto, o trecho de código é automaticamente adaptado para aquele identificador de objeto. Além disso, os trechos de código são *templates de código* e certas adaptação são realizadas assim que o usuário ativa um *template de código* recomendado, como a inclusão de certos *imports* que o *template de código* depende. No *Redprint*, o trecho de código recomendado é estático e há a necessidade de adaptações manuais por parte do usuário. Outras ferramentas de *templates de código*, como o sistema de *templates de códigos* nativo da *IDE Eclipse* (ECLIPSE FOUNDATION, 2015), o *SnipMatch* (WIGHTMAN et al., 2012) e a ferramenta apresentada por Oney e Brandt (2012), não apresentam um *recomendador de templates de código*.

3 Teste com programadores

A fim de obter um entendimento melhor dos benefícios que algumas funcionalidades da ferramenta pode trazer aos programadores, foi realizado um teste experimental envolvendo programadores profissionais. O objetivo principal do teste foi obter informações em relação à experiência que um programador pode ter com a ferramenta, como quais dificuldades enfrentadas e o grau de satisfação em realizar uma tarefa de programação através da ferramenta.

3.1 Metodologia

Treze programadores participaram do teste. Onze programadores foram contactados através do site *Freelancer*[1], que é um site para a contratação de programadores que trabalham com projetos de software precificados por hora de trabalho, e dois programadores foram contactados pessoalmente. Um site[2] foi confeccionado contendo as instruções necessárias para o participante realizar o teste e consistia de uma tarefa de programação que o participante tinha que cumprir utilizando a ferramenta *IPS*.

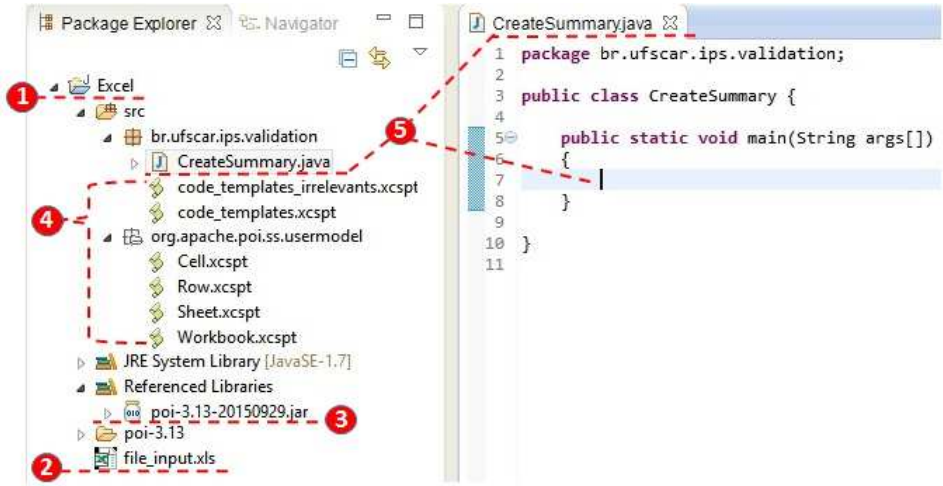
Foi disponibilizado um conjunto de *templates de código* contendo tanto *templates de código* associados a classes *Java* como também templates de códigos não associados a classes *Java*. Além disso, foi disponibilizado um ambiente *Eclipse* com o *IPS* previamente instalado e com a aba do *recomendador de templates de código* ativada.

Os participantes tiveram que realizar uma tarefa de programação que consistia em implementar o método estático *main* de uma classe *Java* denominada *CreateSummary* com o objetivo de sumarizar valores de uma planilha no formato *Excel*. Os participantes tiveram que acessar os dados da planilha e manipulá-la através da *API Apache POI*[3], *API* para manipulação de arquivos *Excel*. No ambiente *Eclipse*, foi disponibilizado aos participantes um projeto *Java* (item 1 destacado na Figura 60) contendo todos os artefatos necessários para realizar a tarefa: o arquivo no formato *Excel* contendo a planilha a ser sumarizada (item 2 destacado na Figura 60); a *API Apache POI* (versão 3.13) devidamente importada no projeto *Java* (item 3 destacado na Figura Figura 60); os arquivos *XCSPT* contendo os *code snippets* (item 4 destacado na Figura Figura 60); e o código fonte da classe *CreateSummary* contendo o método estático *main* para o participante implementar (item 5 destacado na Figura Figura 60).

A planilha no formato *Excel* (arquivo *file_input.xls* destacado como item 2 na

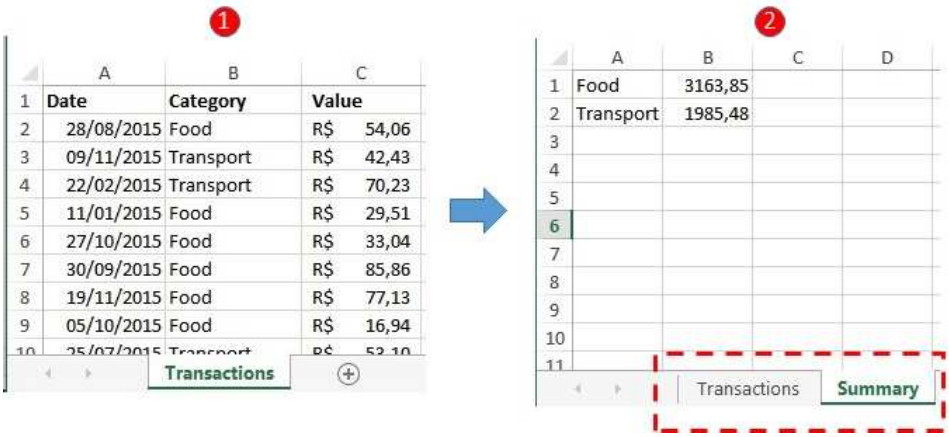
[1] <https://www.freelancer.com/>
[2] <https://ipstool.wordpress.com/>
[3] <https://poi.apache.org/spreadsheet/index.html>

Figura 60 – Projeto *Java* para o teste



Fonte: próprio autor

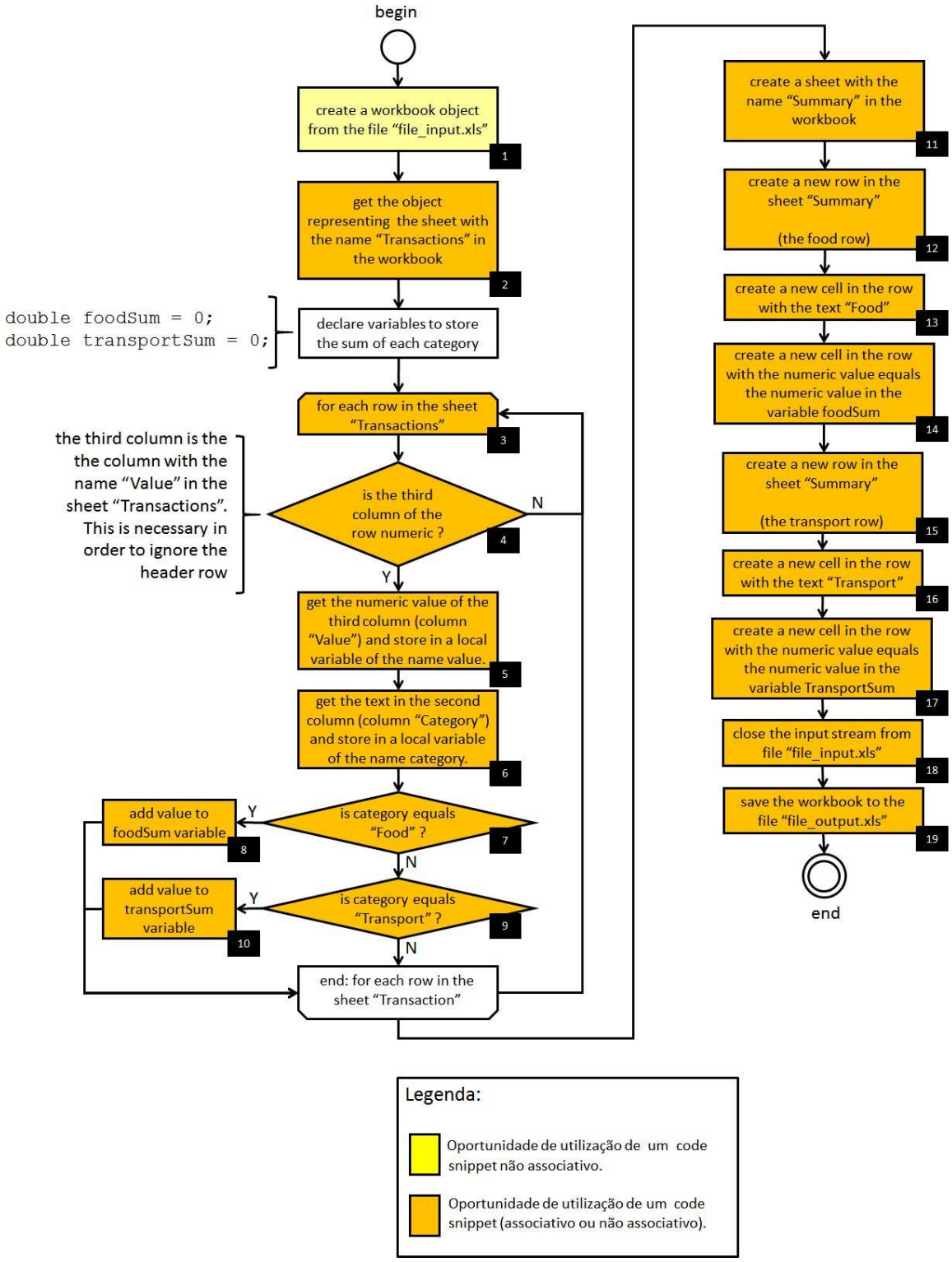
Figura 61 – Planilha *file_input.xls*



Fonte: próprio autor

Figura 60) continha uma aba denominada *Transactions* (item 1 destacado na Figura 61) com três colunas: *Date*, *Category* e *Value*. As linhas da planilha representavam transações. A coluna *Date* representava a data da transação, a coluna *Category* representava a categoria da transação e a coluna *Value* representava o valor da transação. Havia somente duas categorias de transação: *Food* e *Transport*. O objetivo da tarefa dada ao participante era criar uma nova aba com o nome *Summary* contendo a sumarização dos valores das transações para a categoria *Food* e para a categoria *Transport* (item 2 destacado na Figura 61).

Figura 62 – Fluxograma da tarefa



Fonte: próprio autor

Como não foi o objetivo do experimento registrar informações relacionadas à capacidade dos participantes em solucionar o problema proposto na tarefa em relação às questões de decisões de projeto, foi entregue um fluxograma aos participantes que representava quais os passos eram necessários serem executados pelo método *main* a fim de atingir o objetivo da tarefa proposta (fluxograma apresentado na Figura 62, com os passos não destacados com cores diferentes). Era esperado, portanto, que o participante conseguisse implementar cada passo do fluxograma através do auxílio dos *templates de código* fornecidos e utilizando a *API Apache POI* para acessar e manipular a planilha.

Antes da tarefa iniciar, foi solicitado ao participante que ele assistisse a um vídeo que apresentava como utilizar o *IPS*. Foi apresentado no vídeo conceitos como: o que é um *template de código*, como utilizá-los e o que são os parâmetros dos *templates de código*. O vídeo também continha informações sobre como utilizar os *templates de código* associados a uma classe *Java* e sobre como utilizar o *recomendador de templates de código*. Foi solicitado ao participante que ele gravasse a interação dele com a ferramenta *IPS* durante toda a realização da tarefa. Os vídeos foram gravados utilizando a ferramenta *CamStudio*[4]. Após a conclusão da tarefa, o participante teve que: preencher e entregar um questionário (em inglês), entregar o vídeo da interação dele com a ferramenta e entregar o código fonte da classe *CreateSummary* com o método *main* implementado. Uma versão em português do questionário que os participantes tiveram que preencher é apresentado no Apêndice A.

3.2 Resultados

Treze programadores participaram do teste, no entanto não foi possível obter o vídeo da interação com a ferramenta *IPS* de três participantes e, portanto, as informações desses três participantes foram desconsideradas.

Os perfis dos participantes de acordo com as respostas fornecidas no questionário (Apêndice A) estão sumarizados na Figura 64. Todos os participantes declararam possuir experiência com *Java* e com a *IDE Eclipse*. Alguns participantes declararam possuir experiência também com as linguagens *C++* (3), *C#* (3), *C* (2), *JavaScript* (2), *PHP* (2), *Groovy* (2), *Python* (1), *Perl* (1) e *Delphi* (1), e com as *IDEs Netbeans* (5), *Visual Studio* (2), *IntelliJ* (2), *Android Studio* (2), *JDeveloper* (1), *RSA* (1), *Turbo C* (1) e *Borland Delphi* (1). Em relação ao tempo de experiência com programação, cinco participantes responderam possuir mais de 5 anos, três participantes responderam possuir entre 3 e 5 anos, dois responderam possuir entre 1 e 3 anos. Quatro participantes declararam que não conheciam nenhuma ferramenta de *templates de código*, três declararam que conheciam mas que nunca tinham utilizado, dois declararam que utilizam raramente e um declarou

[4] <http://camstudio.org/>

que utiliza frequentemente.

Figura 63 – Perfis dos participantes

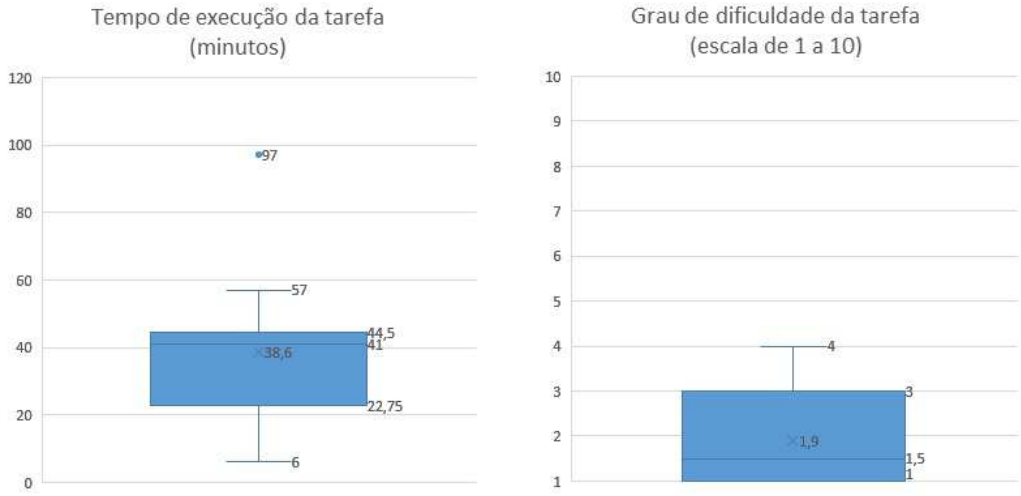
Fonte: próprio autor

Dos dez participantes, oito participantes entregaram um código com exatamente a funcionalidade solicitada. Um dos participantes entregou um código com a funcionalidade solicitada, no entanto modificando o arquivo de entrada. Ou seja, a aba com o título *"Summary"* foi criada no próprio arquivo de entrada de nome *"file_input.xls"*, ao invés de ser criado um novo arquivo com o nome *"file_output.xls"* contendo a aba *"Summary"*. Outro participante, apesar de ter seguido os passos do fluxograma apresentado, entregou

um código que cria a aba *"Summary"* vazia devido a um erro no código entregue.

Na Figura 64, estão sumarizados os tempos e os graus de dificuldades declarados pelos participantes na realização da tarefa. Para codificar o método *main* da classe *CreateSummary*, os participantes levaram uma mediana de 41 minutos, sendo que o tempo mínimo foi de 6 minutos e o tempo máximo foi de 97 minutos. Em relação ao grau de dificuldade na realização da tarefa para implementar o método *main* da classe *CreateSummary*, em uma escala de um (muito fácil) a dez (muito difícil), cinco consideraram que o grau de dificuldade foi um, dois consideraram que o grau de dificuldade foi dois, dois consideram que o grau de dificuldade foi três e um considerou que o grau de dificuldade foi quatro.

Figura 64 – Tempo e grau de dificuldade da tarefa



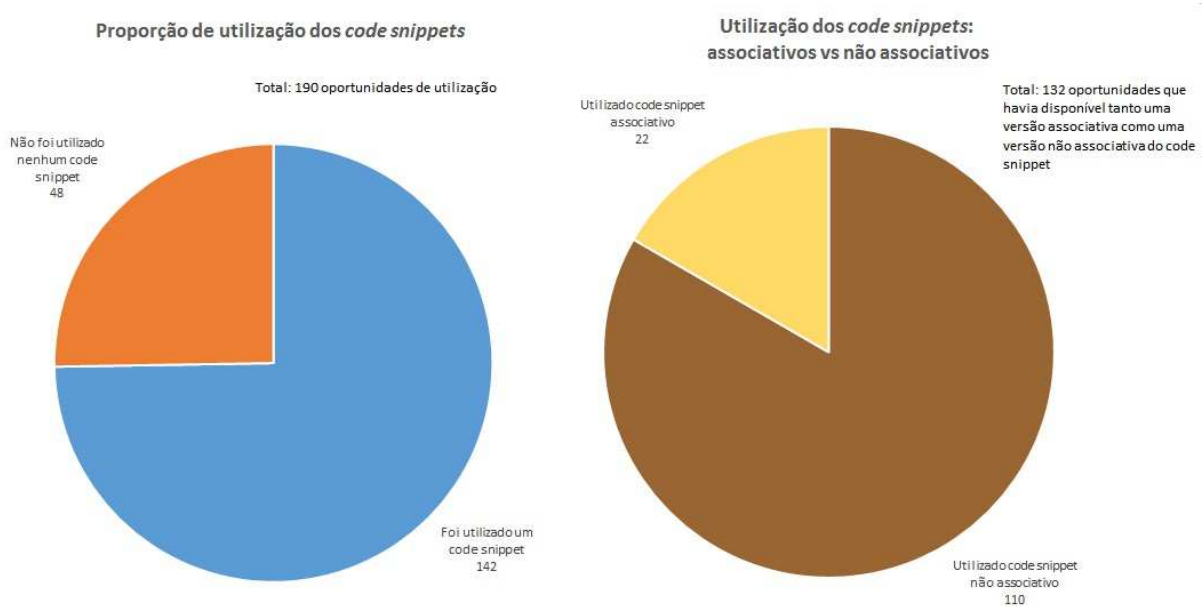
Fonte: próprio autor

3.2.1 Utilização dos *code snippets*

No fluxograma apresentado na Figura 62, estão destacados passos em que havia a oportunidade de utilizar algum *code snippet* disponível no *IPS* durante a realização da tarefa pelos participantes. Para o primeiro passo do fluxograma, em cor amarela, havia disponível um *code snippet* relevante não associativo que o participante poderia utilizar para implementar o passo. Para os demais passos, em cor laranja, foi disponibilizado um *code snippet* associativo e uma versão não associativa do mesmo *code snippet* relevante para o passo. Além disso, foram disponibilizados mais 32 *code snippets* não associativos e 32 *code snippets* associativos irrelevantes para a tarefa. Um *code snippet* associativo relevante para implementar um determinado passo poderia ser acionado pelo participante tanto digitando um identificar de objeto seguido do carácter # como também através do *recomendador de code snippets*.

No total, considerando os dez participantes, houve 190 oportunidades de utilizar um *code snippet* para implementar um passo do fluxograma. Em 190 dessas oportunidades, um *code snippet* acionado através do *IPS* foi utilizado em 142 (74,74%) oportunidades para implementar um passo do fluxograma. Dessas 142 vezes que algum participante recorreu ao *IPS*, 132 vezes era uma oportunidade que o participante tinha disponível tanto um *code snippet* associativo como um versão não associativa. Dessas 132 oportunidades, algum participante preferiu utilizar a versão não associativa em 110 (83,33%) oportunidades e algum participante preferiu utilizar a versão associativa em 22 (16,67%) oportunidades. Das 22 oportunidades que algum participante preferiu utilizar um *code snippet* associativo, somente duas vezes o *recomendador de code snippets* foi utilizado.

Figura 65 – Utilização dos *code snippets*



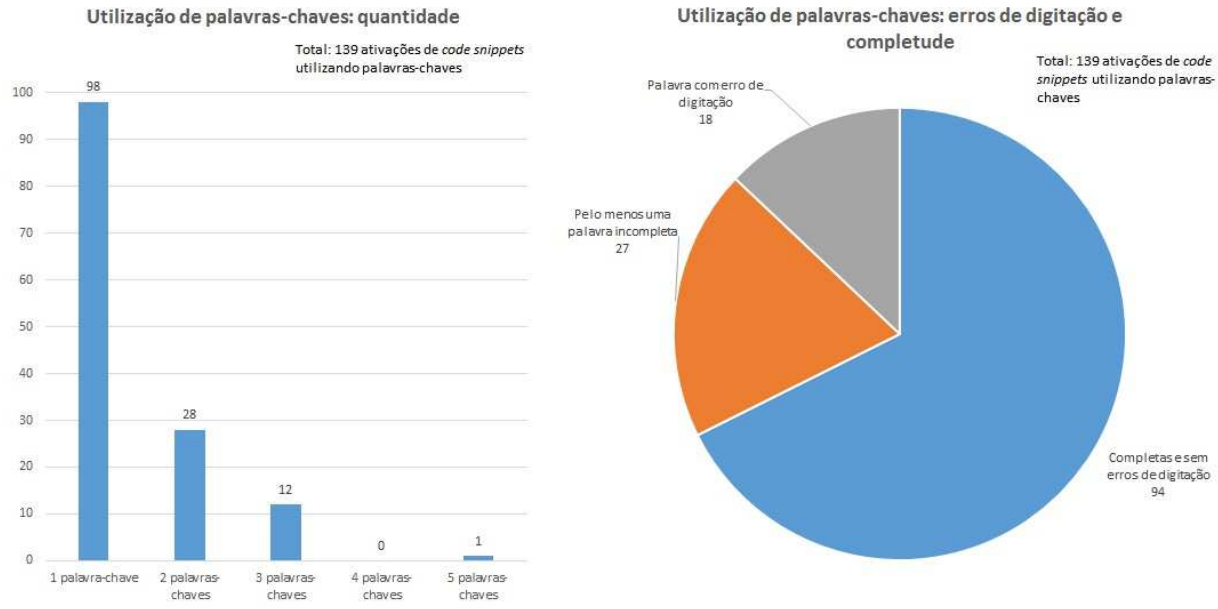
Fonte: próprio autor

3.2.2 Palavras-chaves

Uma das observações realizadas durantes o teste com os programadores foi como os participantes utilizaram as palavras-chaves para procurar pelos *code snippets* desejados. Houve um preferência para a utilização de poucas palavras-chaves. Frequentemente, foi possível observar que os participantes voluntariamente tentaram minimizar a quantidade de palavras-chaves na busca, o *code snippet* desejado era selecionado assim que este aparecia entre os primeiros da listagem e consequentemente o participante não digitava mais palavras-chaves no campo de busca. De 139 ativações de *code snippets* que algum participante utilizou palavras-chaves para a busca do *code snippet*, 98 (70,50%) vezes foi utilizado uma palavra-chave, 28 (20,14%) vezes foi utilizado duas palavras-chaves,

12 (8,63%) vezes foi utilizado três palavras-chaves e em uma situação um participante utilizou cinco palavras-chaves. Além disso, das 139 ativações de *code snippets* que algum participante utilizou palavras-chaves para a busca do *code snippet*, 94 (67,63%) vezes algum participante utilizou palavras-chaves completas e sem erros de digitação. Vinte e sete vezes (19,42%) algum participante utilizou pelo menos uma palavra-chave incompleta na busca, ou seja, utilizou somente os primeiros caracteres da palavra-chave. Dezoito vezes (12,95%) algum participante cometeu algum erro de digitação.

Figura 66 – Utilização de palavras-chaves



Fonte: próprio autor

3.2.3 Renomeação de variáveis

Após a ativação de um *code snippet*, para uma maior clareza do código fonte sendo editado, é recomendado que as variáveis declaradas pelos *code snippets* sejam renomeadas. No entanto, durante os testes com os programadores, alguns participantes preferiram não renomear todas as variáveis. Por exemplo, durante os testes realizados com os programadores, do passo 12 até o passo 17 do fluxograma apresentado na Figura 62, o participante teve que realizar a ativação da seguinte sequência de *code snippets*: *code snippet createNewRowInTheSheet* para criar uma linha na planilha, *code snippet createNewTextCellInTheSheet* para criar uma célula com conteúdo textual, *code snippet createNewNumericCellInTheSheet* para criar uma célula com um conteúdo numérico, novamente o *code snippet createNewRowInTheSheet* para criar outra linha na planilha, novamente o *code snippet createNewTextCellInTheSheet* para criar outra célula com conteúdo textual e novamente o *code snippet createNewNumericCellInTheSheet* para criar

outra célula com um conteúdo numérico. Essa sequência, sem renomear as variáveis, gera o código apresentado na Figura 67.

Figura 67 – Renomeação de variáveis

```
1  Sheet summarysheet = wb.createSheet("Summary");
2
3  Row newRow = summarysheet.createRow(summarysheet.getPhysicalNumberOfRows());
4  Cell newTextCell = newRow.createCell(newRow.getLastCellNum() < 0 ? 0 : newRow.getLastCellNum());
5  newTextCell.setCellValue("Food");
6  Cell newTextCell1 = newRow.createCell(newRow.getLastCellNum() < 0 ? 0 : newRow.getLastCellNum());
7  newTextCell1.setCellValue(foodSum);
8
9  Row newRow1 = summarysheet.createRow(summarysheet.getPhysicalNumberOfRows());
10 Cell newTextCell2 = newRow1.createCell(newRow1.getLastCellNum() < 0 ? 0 : newRow1.getLastCellNum());
11 newTextCell2.setCellValue("Transport");
12 Cell newTextCell3 = newRow1.createCell(newRow1.getLastCellNum() < 0 ? 0 : newRow1.getLastCellNum());
13 newTextCell3.setCellValue(transportSum);
```

Fonte: próprio autor

Alguns participantes (três dos dez participantes) preferiram renomear somente as variáveis do tipo *Row* declaradas nas linhas 3 e 9 do trecho de código apresentado acima e deixaram de renomear as variáveis do tipo *Cell* declaradas nas linhas 4, 6, 10 e 12. Provavelmente, nesse caso, os participantes acharam que renomear essas quatro variáveis traria um grau de detalhamento semântico desnecessário. É interessante notar que a funcionalidade da renomeação automática do *IPS* gerou um código sem erros de compilação e, portanto, desobrigou o participante a renomear tais variáveis. Embora não nomear variáveis com um nome relevante seja uma má prática de programação, deixar a possibilidade do programador simplesmente utilizar os nomes gerados automaticamente por uma ferramenta de *template de código* como o *IPS* pode trazer uma maior agilidade durante a programação. Isso é particularmente útil em situações como a apresentada nesta Seção, quando o programador julga que renomear certas variáveis possa ser um grau de detalhamento desnecessário, ou durante uma prototipação, quando o código fonte é utilizado somente para testar uma ideia e é posteriormente descartado.

3.2.4 Parâmetros

Para configurar os argumentos dos parâmetros dos *code snippets*, o *IPS* oferece algumas facilidades. Ele recomenda identificadores de objetos (variável local ou atributo de classe) disponíveis no escopo do código fonte. As sugestões de argumentos apresentadas são ordenadas pela similaridade entre o nome do identificador e o nome do parâmetro e, por padrão, o primeiro argumento sugerido é automaticamente selecionado para o parâmetro. Em muitos passos do fluxograma foi utilizado *code snippets* cujos parâmetros tinham somente um argumento compatível no escopo do código fonte sendo editado pelo participante e, nesses casos, o *IPS* automaticamente selecionou a argumento compatível. Logo, foi possível observar durante o teste com os programadores que essa funcionalidade traz um agilidade maior na utilização dos *code snippets*. No entanto, é necessário ressaltar

que para dois participantes a seleção automática de argumentos acabou introduzindo erro no código. Por exemplo, para um dos participantes no passo 12 do fluxograma apresentado na Figura 62, o participante ativou o *code snippet createNewRowInTheSheet* para criar uma nova linha na planilha com o nome *"Summary"*. O *code snippet* aceita como parâmetro um objeto do tipo *Sheet* e, no escopo do código fonte sendo editado pelo participante, havia duas variáveis do tipo *Sheet*. Uma variável tinha o nome *sheet* e representava a aba da planilha com o nome *"Transactions"*. A outra variável tinha o nome *sheet1* e representava a aba da planilha com o nome *"Summary"*. O participante tinha que selecionar a variável *sheet1* como argumento do *code snippet createNewRowInTheSheet*, no entanto o usuário deixou selecionado o argumento padrão sugerido pelo *IPS*, que foi a variável *sheet*. Isso gerou um erro no código que foi somente corrigido pelo participante após ele depurar o código.

Houve 36 casos que algum participante teve que escolher um identificador de objeto (variável local) como argumento para um parâmetro diferente do identificador de objeto automaticamente selecionado pelo *IPS*. Desses 36 casos, 22 vezes algum participante utilizou o teclado para digitar o nome do identificador de objeto desejado, e 14 vezes foi utilizado a lista contendo as sugestões de argumentos, nestes casos o participante realizou a seleção do identificador de objeto desejado através do *mouse*.

3.2.5 Comentários dos participantes

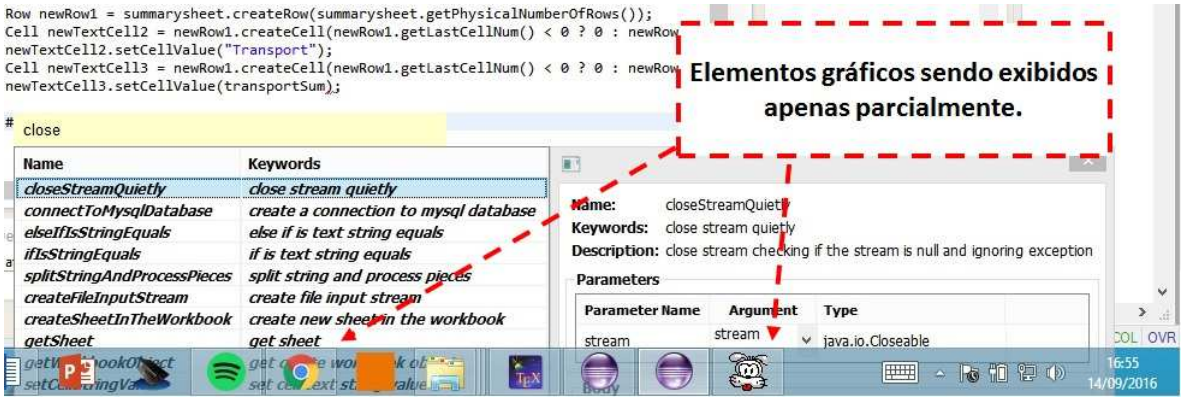
Através do formulário entregue aos participantes, eles puderam deixar comentários sobre a experiência que tiveram durante o teste ao utilizar a ferramenta *IPS*. No geral, os comentários foram bastante positivos e foram recebidos comentários como: "Essa ferramenta é fantástica, espero que no futuro eu possa utilizá-la em muitos *frameworks*", "É uma ótima ferramenta", "Eu realmente gostei desta ferramenta", "É uma ferramenta muito legal", "Parabéns pelo trabalho". Um dos participantes afirmou que a ferramenta auxiliou bastante na realização da tarefa principalmente pelo fato que havia um tempo que ele não programava em *Java*. Outro participante enfatizou que ele achou a tarefa realmente fácil com todas as sugestões de *code snippets* apresentadas.

Um dos participantes fez uma reflexão mais aprofundada sobre a ferramenta e comentou que a ferramenta auxilia muito na realização de uma tarefa de programação, principalmente quando o programador não tiver experiência na utilização de uma *API*, e que os *code snippets*, quando bem empregados, podem contribuir para diminuir o tempo de desenvolvimento de um *software* e ao mesmo tempo criar códigos mais otimizados e com um estilo mais padronizado. Ainda, ele comentou que os *code snippets* poderiam ser utilizados por estudantes de qualquer idade em cursos de programação a fim de facilitar a aprendizagem.

Três participantes apontaram um problema de usabilidade da ferramenta. Quando

o cursor de texto está localizado próximo do limite inferior da área de trabalho do sistema operacional, a listagem de *code snippets*, que é apresentada após uma busca, e a janela contendo as informações de um *code snippet* selecionado são exibidas apenas parcialmente. Esse problema, embora ainda possibilitasse o usuário de utilizar o *code snippet*, em algumas situações obrigava o usuário a arrastar a janela que exibe as informações do *code snippet* selecionado mais para cima para que ele pudesse escolher os argumentos dos parâmetros.

Figura 68 – Problema de usabilidade



Fonte: próprio autor

Uma nova versão da ferramenta *IPS* foi confeccionada corrigindo o problema mencionado e disponibilizada no site <https://ipstool.wordpress.com/>. Nesta nova versão, caso o cursor de texto encontrar-se próximo do limite inferior da área de trabalho do sistema operacional, a listagem dos *code snippets* e a janela que exibe as informações do *code snippet* selecionado são exibidas acima da área de texto na qual o usuário digita as palavras-chaves.

3.3 Análise

Era esperado que os participantes utilizassem os *code snippets* associativos de forma mais frequente. Das 132 oportunidades que o participante teve a opção de utilizar ou a versão associativa ou a versão não associativa, algum participante preferiu utilizar a versão associativa somente em 22 (16,67%) oportunidades. É possível que esse comportamento tenha ocorrido principalmente por falta de um treinamento com uma ênfase maior nas funcionalidades mais inovadoras da ferramenta *IPS*. Conforme mencionado anteriormente, somente foi apresentado um vídeo ao participante de 10 minutos englobando tanto conceitos gerais dos *templates de código* como as funcionalidades mais específicas do *IPS*, como a utilização dos *code snippets* associativos e do *recomendador de code snippets*. Trabalhos futuros deverão realizar um estudo empírico mais detalhado para comparar o desempenho de dois grupos, um grupo utilizando somente *code snippets* associativos, que receberia um

treinamento mais aprofundado para utilizar os *code snippets associativos*, e outro grupo utilizando somente *code snippets* não associativos. Este último grupo também receberia um treinamento, no entanto que não abordasse os *code snippets* associativos.

Outra hipótese é que para o usuário utilizar um *code snippet* associativo, sem utilizar o *recomendador de code snippets*, é necessário que ele pense, escolha e digite para qual identificador de objeto (variável local ou atributo de classe) ele deseja acionar um *code snippet*. Quando o usuário utiliza um *code snippet* não associativo, essa necessidade de ter que pensar, escolher e digitar o identificador de objeto não existe, ao custo de ter que selecionar um parâmetro adicional. Devido as facilidades que o *IPS* proporciona na seleção de parâmetros, o usuário pode preferir ter um parâmetro adicional para selecionar, que é o caso quando se utiliza um *code snippet* não associativo, do que ter que digitar um identificador de objeto para acionar o *IPS*. As facilidades que o *IPS* proporciona durante a seleção de parâmetro é que ele além de recomendar argumentos para os parâmetros baseado no tipo do parâmetro também seleciona automaticamente um argumento cujo nome seja o mais semelhante ao nome do parâmetro. Todavia, ainda era esperado que essa predileção do usuário aos *code snippets* não associativos não existisse, pois há opção do usuário utilizar o *recomendador de code snippets*, que automaticamente sugere a ativação dos *code snippets* para os identificadores de objetos disponíveis no escopo do código fonte sendo editado. Porém, o *recomendador de code snippets* também foi pouco utilizado pelos participantes no teste. Das 22 oportunidades que algum participante preferiu utilizar um *code snippet* associativo, somente duas vezes foi através do *recomendador de code snippets*. É possível que isso tenha ocorrido devido a localização da aba com as recomendações. Uma hipótese é que os usuários prestam pouca atenção na aba, pois a aba não está próxima o suficiente da área onde o usuário está escrevendo o código fonte. Trabalhos futuros poderão experimentar fornecer menos recomendações de *code snippets* associativos ao usuário, porém através de elementos gráficos suspensos próximos ao cursor de texto enquanto o usuário estiver digitando.

Em relação a utilização das palavras-chaves, em 32,37% das ativações de *code snippets* que algum participante utilizou palavras-chaves, foram utilizadas palavras-chaves incompletas. Nestes casos, o participante preferiu digitar somente uma parte do início da palavra-chave ou o participante cometeu erros de digitação da palavra-chave. Além disso, a quantidade de palavras-chaves utilizadas nas buscas foi minimizada pelos participantes, 70,50% das vezes foi utilizada somente uma palavra-chave. Esses fatos evidenciam a importância que o suporte a buscas mais relaxadas, quando os critérios utilizados pelo usuário não casam exatamente com as propriedades do elemento desejado na busca, tem em ferramentas de trechos de código. O *IPS*, utilizando funcionalidades disponíveis na *API Lucene*, considera que uma palavra-chave foi utilizada mesmo quando o usuário comete pequenos erros de digitação na palavra-chave ou quando o usuário digita somente o início de uma palavra-chave. Além disso, o *IPS* realiza uma busca que prioriza *code snippets* cujos

parâmetros são compatíveis com identificadores de objetos (variáveis locais e atributos de classe) disponíveis no escopo do código fonte sendo editado. Essa priorização, em algumas situações, tem um efeito de priorizar *code snippets* relevantes ao usuário mesmo quando ele não tenha utilizado palavras-chaves adequadas para a busca. Dessa forma, apesar de 32,37% das vezes que um *code snippet* foi ativado no teste não foram utilizadas palavras-chaves completas e sem erros de digitação, o *IPS* foi capaz de apresentar os *code snippets* desejados aos participantes. Um fato interessante é que de dezoito vezes que algum participante cometeu algum erro de digitação, treze vezes algum participante aglutinou duas ou mais palavras-chaves. Por exemplo, ao invés do participante ter utilizado as palavras-chaves "get sheet" para obter o *code snippet* que obtém um objeto representando a aba da planilha, o participante utilizou a palavra-chave "getsheet". Este caso não foi previsto durante o desenvolvimento do *IPS*. Mesmo quando isso ocorreu, o *IPS* apresentou o *code snippet* desejado ao usuário, no entanto não apresentou o *code snippet* desejado entre os três primeiros resultados da listagem, e sim entre os três últimos. Versões futuras do *IPS* poderão implementar um preprocessamento do conteúdo digitado pelo usuário a fim de identificar e separar palavras-chaves aglutinadas e priorizar melhor os *code snippet* quando o usuário tiver este comportamento.

Em relação à configuração dos parâmetros dos *code snippets*, em *61,11%* dos casos que algum participante teve que escolher um identificador de objeto (variável local) como argumento de um parâmetro, o participante preferiu utilizar o teclado. Portanto, versões futuras do *IPS* poderão implementar uma funcionalidade que complete o nome de um identificador de objeto sendo utilizado como argumento de um parâmetro enquanto o usuário estiver digitando o início do nome do identificador.

Embora o teste realizado com os programadores tenha sido importante para levantar evidências, é necessário ressaltar aqui as limitações deste estudo. A amostra de programadores foi bastante reduzida e pouco representa a população de programadores como um todo. O meio utilizado para contactar os participantes, que foi o site *Freelancer*[5], embora a principio possa ser considerado um meio para acessar profissionais de diferentes perfis, foi possível perceber que há uma concentração muito grande de profissionais de poucos países, como Índia, Paquistão e Egito. O site *Freelancer* também apresenta uma tendência de selecionar os membros do site mais experientes e assíduos para tarefas maiores e complexas. Tarefas menores e simples, como o teste realizado nesta dissertação, são sugeridas para membros menos experientes no site. Outro ponto é que o teste realizado não foi um teste comparativo. Trabalhos futuros deverão realizar testes comparando resultados de grupos diferentes. Por exemplo, um grupo utilizando somente *code snippets* associativos e outro utilizando somente *code snippets* não associativos. Além disso, futuras versões do *IPS* poderão ter seu código disponível em um projeto de código aberto para obter

[5] <https://www.freelancer.com/>

uma grande quantidade de usuários. Isto permitiria realizar estudos exploratórios que atingissem uma gama maior de usuários e ajudaria a compreender os conceitos empregados na ferramenta para estilos de usuários diferentes e em ambientes de trabalho diferentes.

Conclusão

Esta dissertação apresentou a ferramenta *IPS*. Foi apresentada também uma análise bibliográfica que possibilitou compreender o desenvolvimento atual das ferramentas, posicionar a ferramenta *IPS* em relação às outras ferramentas no que tange as similaridades e os diferenças funcionais e derivar possíveis trabalhos futuros para o tema. Por último, foi apresentado um teste com programadores profissionais que foi realizado com o objetivo de levantar informações em relação à experiência que um programador pode ter com a ferramenta *IPS*, como quais dificuldades enfrentadas e o grau de satisfação em realizar uma tarefa de programação através da ferramenta.

A ferramenta *IPS* traz conceitos inovadores em relação às outras ferramentas de trechos de código, como os parâmetros do tipo *role*, os *code snippets* associados a classes *Java* e o recomendador de *templates*. É possível destacar também algumas funcionalidades do *IPS*, que embora não tão inovadoras, são importantes serem consideradas para facilitar o cotidiano dos programadores, como a renomeação automática de variáveis, a aceitação de palavras-chaves incompletas na busca, a sugestão de argumentos para os parâmetros dos *code snippets* ordenadas pela similaridade com o nome do parâmetro e a seleção automática do primeiro argumento sugerido para o parâmetro.

Em relação ao teste com os programadores, o teste evidenciou a importância que um suporte a buscas relaxadas, quando os critérios utilizados pelo usuário não casam exatamente com as propriedades do elemento desejado na busca, possui em uma ferramenta de *templates de código*. Nesse quesito, a ferramenta *SnipMatch* (WIGHTMAN et al., 2012) estendeu a ideia utilizada no *Eclipse* e propôs um sistema de busca de *templates de código* baseado em palavras-chaves, diferente da funcionalidade nativa do *Eclipse* que busca os *templates de código* através somente do nome do *template*. Neste trabalho de mestrado, através da ferramenta *IPS* é proposto um suporte a busca por palavras-chaves que considera também palavras-chaves com pequenos erros de digitação.

Embora era esperado que os participantes utilizassem os *code snippets* associativos e o recomendador de *code snippets* de forma mais frequente durante o teste com os programadores, esses conceitos não podem ser descartados. Futuros testes com usuários devem ser executados e outras estratégias devem ser testadas. Para os *code snippets* associativos, é necessário realizar um teste que compare o desempenho de dois grupos, um grupo utilizando somente *code snippets* associativos e outro grupo utilizando somente *code snippets* não associativos. É possível testar também a possibilidade de integrar o conceito dos *code snippets* associativos no próprio *autocompletador de código* nativo do *Eclipse*. Isso possibilitaria testar o conceito dentro de uma interface mais familiar para os

usuários do *Eclipse*. Já para o recomendador de *code snippets*, há a possibilidade de testar uma estratégia diferente em relação à exibição das recomendações. Na versão atual dez recomendações são exibidas em uma aba separada da área onde o usuário edita o código fonte na *IDE Eclipse*. Uma outra estratégia seria exibir poucas recomendações (duas ou três), no entanto próximas e abaixo do cursor de texto enquanto o usuário estiver editando um código fonte *Java*. Isso provavelmente despertaria uma atenção maior do usuário. Ainda assim, esta estratégia deverá ser validada quanto a sua importância em relação ao ganho de produtividade ou de satisfação por parte do programador. Como essas recomendações estarão próximas da área onde o programador edita o código fonte, ele deverá sentir-se à vontade com essas recomendações. Além disso, é possível investigar e testar estratégias diferentes para a seleção dos *code snippets* que são recomendados ao usuário. Na versão atual do *IPS*, são recomendados *code snippets* associados às classes *Java* que sejam do mesmo tipo dos identificadores de objetos (variáveis locais e atributos de classes) disponíveis no escopo do código fonte sendo editado pelo usuário. Por exemplo, uma alternativa seria utilizar *filtragem colaborativa* (ROBILLARD, 2014, p. 16) para recomendar *code snippets* para um determinado usuário com um determinado perfil baseado no conjunto de *code snippets* utilizados por outros usuários do mesmo perfil.

Através da análise bibliográfica foi possível realizar um levantamento das características das ferramentas *CSACTs - **C**ode **S**nippets and **A**PI **C**omponents **T**ools*, que é o conjunto de ferramentas que auxiliam a localização de trechos de código ou de componentes de uma *API*, como classes e métodos, e de ferramentas que sugerem trechos de código ou componentes de uma *API* relevantes para uma determinada tarefa de programação. As características foram apresentadas no Capítulo 1 através de cinco perspectivas: o *contexto de uso*, que é o ambiente no qual a ferramenta está inserida; a fonte de dados da qual a ferramenta extrai as informações a serem apresentadas ao usuário; a entrada utilizada pela ferramenta para decidir quais elementos são relevantes ao usuário; o resultado, que pode apresentar trechos de códigos ou componentes de uma *API* ao usuário, e as facilidades oferecidas pelas ferramentas a fim de adaptar um elemento selecionado ao contexto do projeto do *software* em desenvolvimento, após o usuário ter realizado uma busca pelo elemento desejado. Esse levantamento permitiu destacar as similaridades e as principais contribuições do *IPS*. Além disso, foi possível levantar aspectos sobre o tema que não foram cobertos na solução proposta nesta dissertação e que podem ser investigados por trabalhos futuros. Esses aspectos são discutidos a seguir.

Para o *contexto de uso*, trabalhos futuros poderão explorar técnicas a fim de reduzir a dependência arquitetural das ferramentas *CSACTs* integradas a uma *IDE*, como por exemplo a solução sugerida por Sawadsky e Murphy (2011, p. 50), que é a criação de uma arquitetura modularizada tal que os módulos que não necessitam serem acoplados à *IDE* podem ter o código fonte reutilizado entre *IDEs* diferentes e somente módulos que são acoplados à *IDE* possuem uma versão para cada *IDE*. Ainda, a estratégia adotada por

Goldman e Miller (2008), que implementa um canal de comunicação entre a *IDE Eclipse* e o navegador *Firefox*, também poderá ser explorada para as ferramentas *CSACTs*, a fim de utilizar as informações presentes no contexto de programação na *IDE* e ao mesmo tempo aproveitar os recursos de interação fornecidos pelo navegador *web*.

A utilização dos testes unitários como uma fonte de dados para extrair trechos de código para uma ferramenta *CSACT* é uma estratégia promissora, sobretudo se for considerado os argumentos apresentados por Ghafari (2014) e os resultados empíricos apresentados por Nasehi e Maurer (2010). No entanto, muitos são os desafios para extrair trechos de códigos que sirvam de exemplos da utilização de uma *API*. Embora heurísticas tenham sido criadas por Zhu et al. (2014) e utilizadas para implementar a ferramenta *UsETeC*, trabalhos futuros poderão explorar novas heurísticas com o objetivo de realizar estudos comparativos.

Foi possível identificar que as ferramentas de *templates de código*, comparadas com outras ferramentas de trechos de código, possuem um suporte maior para adaptar o trecho de código selecionado pelo usuário no contexto de programação (após o usuário ter realizado uma busca pelo trecho de código desejado). No entanto, como os *templates de código* são cadastrados manualmente, o custo de criação e manutenção desses *templates* pode não justificar a vantagem de utilizá-los. Dessa forma, trabalhos futuros poderão explorar estratégias que auxiliem a criação e a manutenção desses trechos de código. Apenas como um exemplo, uma ferramenta poderia, a partir de páginas *web* ou a partir de projetos de código aberto, extrair e apresentar trechos de código ao usuário. O usuário poderia, então, através do auxílio da ferramenta, adicionar diretivas de integração a um determinado trecho de código selecionado, o qual tornaria-se um *template de código* cadastrado em um repositório a fim de utilizá-lo no ambiente de programação. Uma outra solução para os trechos de códigos cadastrados manualmente seria recorrer ao *Crowdsourcing* (BRABHAM, 2013). Uma ferramenta *CSACT* poderia possibilitar a criação de trechos de código públicos em uma arquitetura cliente-servidor. Um trecho de código público cadastrado na ferramenta por um usuário seria acessível a todos os outros usuários da ferramenta, maximizando dessa forma a quantidade de trechos de código disponíveis para cada usuário. Embora essa estratégia tenha sido explorada por Wightman et al. (2012), trabalhos futuros poderão explorar essa estratégia com o objetivo de apresentar resultados de estudos de casos que tragam um entendimento melhor das vantagens de utilizar o compartilhamento e a criação colaborativa de *templates de código* entre os usuários de uma ferramenta *CSACT* integrada a uma *IDE*.

Muitos trabalhos analisados que tratam sobre o tema procuraram sobretudo apresentar soluções para facilitar a utilização das *APIs*. No entanto, há outros cenários de uso das ferramentas de trechos de código que merecem serem investigados por trabalhos futuros. Um exemplo é no ensino de linguagens de programação. Como ponderado por

um dos participantes durantes os testes: “com o advento cada vez maior dos cursos de programação para pessoas de diversas idades (inclusive crianças), a ferramenta (*IPS*) pode fornecer uma forma fácil de programar e aprender”. Um outro cenário de uso que pode ser explorado é a utilização das ferramentas de *template de código* por programadores com deficiência física. Devido às facilidades fornecidas por essas ferramentas, há uma diminuição do uso do teclado, o que pode facilitar a atividade de programação para portadores de certas deficiências.

Referências

APACHE SOFTWARE FOUNDATION. *Apache Lucene Core*. 2016. Disponível em: <http://lucene.apache.org/core/>. Citado na página 99.

APACHE SOFTWARE FOUNDATION. *Lucene: Similarity*. 2016. Disponível em: <https://lucene.apache.org/core/3_6_0/api/core/org/apache/lucene/search/Similarity.html>. Citado na página 108.

ARAGON CONSULTING GROUP. *Krugle*. 2015. Disponível em: <http://www.krugle.com/>. Citado 4 vezes nas páginas 32, 36, 55 e 59.

BAEZA-YATES, R.; RIBEIRO-NETO, B. *Modern Information Retrieval: the concepts and technology behind search*. [S.l.]: Addison Wesley, 2011. ISBN 9780321416919. Citado na página 33.

BAJRACHARYA, S. et al. Sourcerer: a search engine for open source code supporting structure-based search. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 681–682. ISBN 1-59593-491-X. Citado 4 vezes nas páginas 32, 34, 49 e 67.

BAJRACHARYA, S. et al. Sourcerer: a search engine for open source code. In: *International Conference on Software Engineering (ICSE 2007)*. [S.l.: s.n.], 2007. Citado 2 vezes nas páginas 22 e 42.

BAJRACHARYA, S.; OSSHER, J.; LOPES, C. Sourcerer: an infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, v. 79, p. 241 – 259, 2014. ISSN 0167-6423. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). Citado na página 42.

BHARDWAJ, A. P.; LUCIANO, D.; KLEMMER, S. R. Redprint: integrating API specific instant example and instant documentation display interface in IDEs. In: ACM. *Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology*. [S.l.], 2011. p. 21–22. Citado 4 vezes nas páginas 32, 41, 67 e 118.

BLACK DUCK SOFTWARE. *Open Hub Code Search*. 2015. Disponível em: <http://code.openhub.net/>. Citado 5 vezes nas páginas 32, 37, 42, 55 e 59.

BLEWITT, A. *Eclipse 4 Plug-In Development by Example Beginner's Guide*. [S.l.]: Packt Publishing, 2013. (Open source : community experience distilled). ISBN 9781782160335. Citado na página 94.

BRABHAM, D. *Crowdsourcing*. [S.l.]: MIT Press, 2013. ISBN 9780262518475. Citado na página 135.

BRANDT, J. et al. Example-centric programming: integrating web search into the development environment. In: ACM. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [S.l.], 2010. p. 513–522. Citado 15 vezes nas páginas 22, 32, 33, 36, 37, 43, 44, 52, 55, 56, 57, 59, 60, 67 e 115.

BRANDT, J. et al. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: ACM. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [S.l.], 2009. p. 1589–1598. Citado 2 vezes nas páginas 21 e 64.

BRANDT, J. et al. Writing code to prototype, ideate, and discover. *Software, IEEE*, IEEE, v. 26, n. 5, p. 18–24, 2009. Citado na página 54.

BRUCH, M.; MONPERRUS, M.; MEZINI, M. Learning from examples to improve code completion systems. In: ACM. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.], 2009. p. 213–222. Citado 6 vezes nas páginas 32, 39, 40, 41, 42 e 115.

BRUCH, M.; SCHÄFER, T.; MEZINI, M. FrUiT: IDE support for framework understanding. In: ACM. *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*. [S.l.], 2006. p. 55–59. Citado 4 vezes nas páginas 32, 42, 51 e 56.

BUSE, R. P.; WEIMER, W. Synthesizing API usage examples. In: IEEE. *Software Engineering (ICSE), 2012 34th International Conference on*. [S.l.], 2012. p. 782–792. Citado na página 55.

DESHPANDE, A.; RIEHLE, D. The total growth of open source. In: *Open Source Development, Communities and Quality*. [S.l.]: Springer, 2008. p. 197–209. Citado na página 42.

ECLIPSE FOUNDATION. *Eclipse Kepler*. 2015. Disponível em: <http://www.eclipse.org/ide/>. Citado 19 vezes nas páginas 22, 24, 25, 26, 27, 28, 29, 36, 38, 39, 41, 47, 65, 66, 67, 115, 116, 117 e 118.

ECLIPSE FOUNDATION. *Eclipse Corner Article: Abstract Syntax Tree*. 2016. Disponível em: <http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html>. Citado na página 105.

ECLIPSE FOUNDATION. *Platform Plug-in Developer Guide: contexts*. 2016. Disponível em: <http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/guide/workbench_advext_contexts.htm>. Citado na página 104.

ECLIPSE FOUNDATION. *Platform Plug-in Developer Guide: extension points reference*. 2016. Disponível em: <http://help.eclipse.org/kepler/nav/2_1_1>. Citado na página 103.

GHAFARI, M. Extracting code examples from unit test cases. In: IEEE. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. [S.l.], 2014. p. 667–667. Citado 3 vezes nas páginas 45, 46 e 135.

GHAFARI, M. et al. Mining unit tests for code recommendation. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 142–145. ISBN 978-1-4503-2879-1. Citado 2 vezes nas páginas 45 e 47.

GILLELAND, M.; SOFTWARE, M. P. *Levenshtein distance, in three flavors*. 2016. Disponível em: <http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Fall2006/Assignments/editdistance/Levenshtein%20Distance.htm>. Citado na página 77.

GOLDMAN, M.; MILLER, R. Codetrail: connecting source code and web resources. In: *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*. [S.l.: s.n.], 2008. p. 65–72. ISSN 1943-6092. Citado 2 vezes nas páginas 37 e 135.

GOOGLE. *A fall sweep*. 2011. Disponível em: <https://googleblog.blogspot.com.br/2011/10/fall-sweep.html>. Citado na página 63.

HAN, J.; KAMBER, M. *Data Mining: concepts and techniques*. Second. 500 Sansome Street, Suite 400, San Francisco, CA 94111: Morgan Kaufmann Publishers, 2006. ISBN 1-5860-901-6. Citado na página 43.

HENNINGER, S. Retrieving software objects in an example-based programming environment. In: ACM. *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*. [S.l.], 1991. p. 251–260. Citado 7 vezes nas páginas 32, 34, 48, 49, 50, 56 e 62.

HOFFMANN, R.; FOGARTY, J.; WELD, D. S. Assieme: finding and leveraging implicit references in a web search interface for programmers. In: ACM. *Proceedings of the 20th annual ACM symposium on User interface software and technology*. [S.l.], 2007. p. 13–22. Citado 12 vezes nas páginas 22, 32, 33, 36, 43, 44, 48, 55, 57, 62, 63 e 67.

HOLMES, R. et al. The end-to-end use of source code examples: an exploratory study. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 555–558. ISSN 1063-6773. Citado 2 vezes nas páginas 56 e 64.

HOLMES, R.; WALKER, R.; MURPHY, G. Approximate structural context matching: an approach to recommend relevant examples. *Software Engineering, IEEE Transactions on*, v. 32, n. 12, p. 952–970, Dec 2006. ISSN 0098-5589. Citado 7 vezes nas páginas 32, 50, 51, 55, 58, 59 e 67.

HUMMEL, O.; JANJIC, W.; ATKINSON, C. Code Conjurer: pulling reusable software out of thin air. *Software, IEEE*, IEEE, v. 25, n. 5, p. 45–52, 2008. Citado 6 vezes nas páginas 32, 51, 53, 54, 56 e 66.

KIM, J. et al. Adding examples into java documents. In: IEEE. *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. [S.l.], 2009. p. 540–544. Citado na página 54.

KIM, J. et al. Towards an intelligent code search engine. In: *AAAI*. [S.l.: s.n.], 2010. Citado 6 vezes nas páginas 32, 34, 53, 55, 62 e 115.

KIM, M. et al. An ethnographic study of copy and paste programming practices in OOPL. In: *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*. [S.l.: s.n.], 2004. p. 83–92. Citado 3 vezes nas páginas 54, 55 e 64.

LEMOS, O. A. L. et al. CodeGenie: using test-cases to search and reuse source code. In: ACM. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. [S.l.], 2007. p. 525–526. Citado 5 vezes nas páginas 32, 42, 51, 56 e 66.

MANDELIN, D. et al. Jungloid mining: helping to navigate the API jungle. *ACM SIGPLAN Notices*, ACM, v. 40, n. 6, p. 48–61, 2005. Citado 3 vezes nas páginas 32, 51 e 62.

MANNING, C.; RAGHAVAN, P.; SCHÜTZE, H. *Introduction to Information Retrieval*. [S.l.]: Cambridge University Press, 2008. ISBN 978-0521865715. Citado na página 108.

MAR, L. W.; WU, Y.-C.; JIAU, H. Recommending proper API code examples for documentation purpose. In: *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. [S.l.: s.n.], 2011. p. 331–338. ISSN 1530-1362. Citado 8 vezes nas páginas 32, 55, 56, 57, 58, 62, 63 e 64.

MCCAREY, F.; CINNÉIDE, M. Ó.; KUSHMERICK, N. Rascal: a recommender agent for agile reuse. *Artificial Intelligence Review*, Springer, v. 24, n. 3-4, p. 253–276, 2005. Citado 6 vezes nas páginas 32, 37, 42, 51, 52 e 118.

MCLELLAN, S. et al. Building more usable APIs. *Software, IEEE*, v. 15, n. 3, p. 78–86, 1998. ISSN 0740-7459. Citado na página 21.

MICROSOFT CORPORATION. *Visual Studio 2013*. 2014. Disponível em: <www.visualstudio.com/>. Citado 3 vezes nas páginas 22, 65 e 116.

MONTANDON, J. E. et al. Documenting APIs with examples: lessons learned with the APIMiner platform. In: IEEE. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. [S.l.], 2013. p. 401–408. Citado 5 vezes nas páginas 32, 53, 55, 62 e 115.

MOOTY, M. et al. Calcite: completing code completion for constructors using crowds. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*. [S.l.: s.n.], 2010. p. 15–22. ISSN 1943-6092. Citado 6 vezes nas páginas 32, 41, 56, 62, 64 e 65.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are Java software developers using the Eclipse IDE ? *Software, IEEE*, IEEE, v. 23, n. 4, p. 76–83, 2006. Citado 2 vezes nas páginas 40 e 116.

NASEHI, S.; MAURER, F. Unit tests as API usage examples. In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. [S.l.: s.n.], 2010. p. 1–10. ISSN 1063-6773. Citado 2 vezes nas páginas 45 e 135.

NYKAZA, J. et al. What programmers really want: results of a needs assessment for SDK documentation. In: ACM. *Proceedings of the 20th annual international conference on Computer documentation*. 2002. p. 133–141. Disponível em: <https://gordonandgordon.com/downloads/What-Programmers-Really-Want.doc>. Citado na página 21.

OHTANI, A. et al. On the level of code suggestion for reuse. In: IEEE. *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*. [S.l.], 2015. p. 26–32. Citado na página 56.

ONEY, S.; BRANDT, J. Codelets: linking interactive documentation and example code in the editor. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2012. (CHI '12), p. 2697–2706. ISBN 978-1-4503-1015-4. Citado 10 vezes nas páginas 22, 32, 47, 56, 68, 69, 115, 116, 117 e 118.

ORACLE CORPORATION. *Netbeans 8.0*. 2014. Disponível em: <http://netbeans.org>. Citado 4 vezes nas páginas 22, 36, 65 e 116.

ORENSTEIN, D. *QuickStudy: Application Programming Interface (API)*. 2000. Disponível em: <http://www.computerworld.com/s/article/43487/Application_Programming_Interface>. Citado na página 21.

OSHEROVE, R. *The Art of Unit Testing: with examples in C#*. [S.l.]: Manning Publications Company, 2013. ISBN 9781617290893. Citado na página 45.

ROBBES, R.; LANZA, M. Improving code completion with program history. *Automated Software Engineering*, Springer US, v. 17, n. 2, p. 181–212, 2010. ISSN 0928-8910. Citado 7 vezes nas páginas 32, 40, 51, 52, 53, 54 e 62.

ROBILLARD, M. *Recommendation Systems in Software Engineering*. [S.l.]: Springer, 2014. (SpringerLink : Bücher). ISBN 9783642451355. Citado na página 134.

ROBILLARD, M. P. What makes APIs hard to learn ? Answers from developers. *Software, IEEE*, IEEE, v. 26, n. 6, p. 27–34, 2009. Citado 2 vezes nas páginas 21 e 74.

SAHAVECHAPHAN, N.; CLAYPOOL, K. XSnippet: mining for sample code. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 413–430. ISBN 1-59593-348-4. Citado 6 vezes nas páginas 32, 37, 42, 60, 61 e 62.

SAWADSKY, N.; MURPHY, G. C. Fishtail: from task context to source code examples. In: *Proceedings of the 1st Workshop on Developing Tools As Plug-ins*. New York, NY, USA: ACM, 2011. (TOPI '11), p. 48–51. ISBN 978-1-4503-0599-0. Citado 6 vezes nas páginas 32, 37, 38, 51, 53 e 134.

SHULL, F.; LANUBILE, F.; BASILI, V. R. Investigating reading techniques for object-oriented framework learning. *Software Engineering, IEEE Transactions on*, IEEE, v. 26, n. 11, p. 1101–1118, 2000. Citado na página 21.

Snip2Code. 2014. Disponível em: <https://www.snip2code.com>. Citado na página 56.

STYLOS, J.; MYERS, B. et al. Mica: a web-search tool for finding API components and examples. In: IEEE. *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. [S.l.], 2006. p. 195–202. Citado 3 vezes nas páginas 32, 43 e 48.

THE SPARS PROJECT & OSAKA UNIVERSITY. *SPARS-J*. 2015. Disponível em: <http://sel.ist.osaka-u.ac.jp/SPARS/>. Citado 3 vezes nas páginas 32, 55 e 59.

VOGELLA COMPANY. *Eclipse, RCP, Plugin and OSGi Development*. 2016. Disponível em: <http://www.vogella.com/tutorials/eclipse.html>. Citado na página 94.

WANG, L. et al. APIExample: an effective web search based usage example recommendation system for Java APIs. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2011. p. 592–595. Citado 11 vezes nas páginas 22, 32, 36, 43, 44, 47, 49, 56, 59, 63 e 67.

WIGHTMAN, D. et al. SnipMatch: using source code context to enhance snippet retrieval and parameterization. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2012. (UIST '12), p. 219–228. ISBN 978-1-4503-1580-7. Citado 15 vezes nas páginas 22, 32, 47, 52, 56, 62, 64, 67, 68, 115, 116, 117, 118, 133 e 135.

WU, Y.-C.; MAR, L. W.; JIAU, H. CoDocent: support API usage with code example and API documentation. In: *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*. [S.l.: s.n.], 2010. p. 135–140. Citado 3 vezes nas páginas 56, 57 e 58.

YE, Y.; FISCHER, G.; REEVES, B. Integrating active information delivery and reuse repository systems. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 2000. v. 25, n. 6, p. 60–68. Citado 13 vezes nas páginas 32, 33, 35, 37, 42, 49, 51, 52, 53, 54, 56, 63 e 118.

ZHANG, C. et al. Automatic parameter recommendation for practical API usage. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 826–836. ISBN 978-1-4673-1067-3. Citado na página 64.

ZHONG, H. et al. MAPO: mining and recommending API usage patterns. In: *ECOOP 2009–Object-Oriented Programming*. [S.l.]: Springer, 2009. p. 318–343. Citado 8 vezes nas páginas 32, 42, 51, 55, 62, 63, 67 e 115.

ZHU, Z. et al. Mining API usage examples from test code. In: IEEE. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. [S.l.], 2014. p. 301–310. Citado 6 vezes nas páginas 22, 32, 46, 53, 55 e 135.

APÊNDICE A – Questionário Utilizado no Teste Experimental

Questionário:

Nome : _____

Hora de início: _____ Hora de término: _____

1) **Há quanto tempo você possui experiência com programação?**

- □ Há menos de um ano.
- □ Entre um e três anos.
- □ Entre três e cinco anos.
- □ Há mais de cinco anos.

2) **Quais linguagens de programação você mais utiliza? (cite no máximo três linguagens e em ordem de preferência)**

1. _____
2. _____
3. _____

3) **Quais ambientes de programação (*IDEs*) você mais utiliza? (cite no máximo três e em ordem de preferência)**

1. _____
2. _____
3. _____

4) **Você utiliza os *templates* de código (conhecidos também como *code snippets*) disponíveis no ambiente de programação?**

- □ Não conheço esse recurso.
- □ Nunca utilizei, no entanto sei que o recurso está disponível em alguns ambientes de programação.
- □ Raramente utilizo.
- □ Utilizo frequentemente.

5) **Em uma escala de 1 a 10 (sendo 1 muito fácil e 10 muito difícil), como você classificaria o grau de dificuldade que você teve para realizar a tarefa?**

□ 1. □ 2. □ 3. □ 4. □ 5. □ 6. □ 7. □ 8 □ 9. □ 10.

6) **Caso desejar, deixe aqui suas críticas, sugestões e comentários. Comente as principais dificuldades que você teve ao realizar a tarefa.**