DANIEL SCHREIBER

TAGGINGSENSE: MÉTODO BASEADO EM SENSEMAKING PARA COMPREENSÃO DE CÓDIGOS FONTE ORIENTADOS A OBJETOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Curitiba
2012

DANIEL SCHREIBER

TAGGINGSENSE: MÉTODO BASEADO EM SENSEMAKING PARA COMPREENSÃO DE CÓDIGOS FONTE ORIENTADOS A OBJETOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de Concentração: Ciência da Computação

Orientador: Prof. Dra Andreia Malucelli

Curitiba
2012

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor, com anuência de seu orientador.

Curitiba, 01 de Agosto de 2012.

Assinatura do Autor

Assinatura do Orientador

FICHA CATALOGRÁFICA

Schreiber, Daniel
Taggingsense: Método baseado em sensemaking para compreensão de códigos fonte orientados a objetos / D. Schreiber. -- Curitiba, 2012.
Número de páginas 139.

Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná. Curitiba. Programa de Pós-Graduação em Informática.

DEDICATÓRIAS

Aos meus amigos, que sempre acreditaram em mim.

AGRADECIMENTOS

Ao meu melhor amigo, que esteve, e ainda está, presente nos piores e melhores momentos da minha vida. Seu nome, Jesus Cristo.

A minha professora orientadora Dra Andreia Malucelli, que me incentivou, ajudou, contribuiu e me aceitou como orientando para o desenvolvimento deste trabalho.

A professora Dra Sheila Reinehr, por ter ajudado na pesquisa e no desenvolvimento deste trabalho.

Ao Dionei Domingos, por ceder seu tempo e seu conhecimento para o desenvolvimento e pesquisa deste trabalho.

Aos meus pais, Arno e Ivone Schreiber, pelo apoio, amor e exemplo de vida.

Aos meus amigos do grupo de pesquisa de engenharia de software, por compartilhar ideias, sugestões e criticas a pesquisa.

“Critique o tolo, e ele te odiará.
Critique o sábio, e ele te amará.”
(Provérbios 9:8)

RESUMO

Todo software produzido requer manutenção, tanto para correção de erros quanto para aplicação de melhorias. No entanto, a manutenção é uma das atividades da Engenharia de Software que mais está presente no ciclo de desenvolvimento, representando 90% de participação. Um dos principais problemas do alto custo da manutenção é a dificuldade de entender o código fonte de outra autoria. Estima-se que programadores gastam em média até 90% do esforço da manutenção na compreensão do código fonte. Conforme pesquisa de campo realizada nesta pesquisa, 52% dos participantes responderam que a documentação existente dos projetos de software não auxiliam no processo de compreensão do código, e 41% acreditam que um repasse pessoal de quem desenvolveu o código fonte auxilia significativamente. No entanto, nem sempre é possível ter o apoio do autor do código, por se tratar de um sistema legado ou um projeto *open source*. Desta forma, este trabalho tem por objetivo desenvolver um método baseado em sensemaking para reduzir o tempo de compreensão do código fonte orientado a objetos para manutenção de sistemas. Foi utilizada a pesquisa de desenvolvimento formada pelas etapas de análise de mercado, análise de objeto, preparação e desenvolvimento. Para avaliar o método desenvolvido, denominado Taggingsense, foi realizado um experimento prático com programadores identificados como juniores e seniores. Com a aplicação do método TaggingSense um programador júnior conseguiu realizar a mesma manutenção que um programador sênior, com um desempenho superior a 33% quando comparado a um programador sênior sem a utilização do método. Com isso é possível concluir que o método proposto, utilizando a extração do conhecimento, seu processamento e compartilhamento, auxilia positivamente no processo de manutenção e compreensão do código fonte, trazendo benefícios como diminuição do tempo dispendido, qualidade e maior segurança nas alterações realizadas.

Palavras chaves: Conhecimento, Sensemaking, Manutenção, Código Fonte, Compreensão, Ontologia.

ABSTRACT

Every produced software requires maintenance, either for errors correction or implementation of improvements. However, maintenance is one of the activities of software engineering that is most present in the software development cycle, representing 90% of participation. A major problem of the high cost of software maintenance is the difficulty to understand the source code produced by another person. It's estimated that programmers spend up to 90% of maintenance effort in understanding the source code. According to a field research realized, 52% of participants reported that the documentation of software project does not help in the process of understanding, and 41% believes that a knowledge transfer from who developed the source code improves significantly this task. However, the author's source code may not be available, due to the fact that software could be a legacy system or an open source project. Thus, this research aims to develop a method based on sensemaking to reduce object oriented source code comprehension time on systems maintenance. The development research was used, composed of by market analysis stage, object analysis, preparation and development. To evaluate the method, TaggingSense, a practical experiment has been conducted with juniors and seniors developers. As result, the juniors developers could accomplish a 33% of better performance on software comprehension, compared with seniors developers without the use of proposed method. By this, it is possible to conclude that the proposed method, used on knowledge extrication, processing and sharing, helps positively on the process of maintenance and comprehension of source code, bringing benefits such the reduce of time spent, quality and safety on the changes made.

Keywords: Knowledge, Sensemaking, Maintenance, Source Code, Comprehension, Ontology.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ADS	*Abstract Description System*
CMMI	*Capability Maturity Model Integration*
CWA	*Closed-World Assumption*
DAML	*Agent Markup Language*
DDL	*Distributed Description Logics*
DL	*Description Logic*
FOAF	*Friend of a Friend*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
IEE	Instituto de Engenheiros Eletricistas e Eletrônico
ISO	*International Organization for Standardization*
KBS	*Knowledge-based System*
LOC	*Lines of Code*
OIL	*Ontology Inference Layer*
OO	*Object Oriented*
OWA	*Open-World Assumption*
OWL	*Web Ontology Language*
RAM	*Randomic Access Memory*
RDF	*Resource Description Framework*
RDFS	*Resource Description Framework Schema*
RSS	*Really Simple Syndication*
SKOS	*Simple Knowledge Organization System*
SPARQL	*SPARQL Protocol and RDF Query Language*
TI	Tecnologia da Informação

UML	*Unified Modeling Language*
URI	*Uniform Resource Identifiers*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
XML	*Extensible Markup Language*

CAPÍTULO 1 - INTRODUÇÃO

O termo Engenharia de Software surgiu em 1968, com o objetivo de transformar o desenvolvimento de software em uma disciplina de engenharia, como a engenharia elétrica e mecânica. O desenvolvimento de software era uma arte, onde os profissionais da área eram vistos como artistas que usavam a criatividade para construir e “manutenir”, e não como engenheiros, cujo foco está em custos, reuso, estimativa de esforços e na qualidade (SCHNEIDER, 2009) .

A partir desta data, muitos aspectos e processos de construção de software foram criados a aperfeiçoados. Dentre estes processos está a construção de software, que envolve as atividades de análise, projeto, codificação, testes e manutenção (IEEE, 2004). Tais atividades foram elaboradas e desenvolvidas por meio de estudos sobre a engenharia de software que, até nos dias atuais, ultrapassam os limites de atuação nos campos da ciência e engenharia da computação, e herdam de outras disciplinas, como Estatística, Matemática, Economia, Sistemas de Informação Gerenciais, Sistemas de Engenharia, Ciência Cognitiva, Sociologia e Antropologia, Processos e Tecnologia (ERDOGMUS; TANIR, 2002).

Mesmo ultrapassando fronteiras, algumas das atividades da engenharia de software ainda apresentam pontos de deficiência que chamam a atenção. Um destes pontos é a manutenção de software propriamente dita. Conforme dados levantados por este trabalho, o aumento significativo do custo proveniente da manutenção de software deixa em evidência o quão imatura está a construção de software. Conforme pesquisa realizada por (LIENTZ; SWANSON, 1980), o custo total aplicado na manutenção e melhorias, em meados de 1980, correspondia a mais de 50% do custo total de um sistema de software. (ERLIKH, 2000) demonstra que, na década de 2000, o total de esforço na manutenção passou para mais de 90%.

A preocupação com o desenvolvimento de novos sistemas é crescente, pois já ultrapassa a capacidade das empresas de manter os sistemas existentes. Como a grande parte dos orçamentos de software estão direcionados à manutenção, poucos

recursos permanecem para novos desenvolvimentos. (SEACORD et al., 2003) enfatiza que se esta tendência continuar, eventualmente não serão destinados recursos para desenvolvimento de novos sistemas, e, consequentemente, dará início a idade média da era da informação, também conhecida como crise dos sistemas legados.

Para enfatizar a importância da crise dos sistemas legados, vários autores estudaram e levantaram dados relacionados à manutenção. Em 1979 o custo em manutenção correspondia a 67% dos custos relacionados ao desenvolvimento (ZELKOWITZ et al, 1979). De 1981 à 1990 o custo atingiu o patamar de 90% (LIENTZ; SWANSON, 1980), (MCKEE, 1984), (PORT, 1988), (HUFF, 1990), (MOAD, 1990).

Todos os esforços na manutenção se justificam na medida que o software é utilizado pelos usuários. Todo software que está em plena atividade precisa sofrer manutenções com certa frequência, seja para realizar ajustes pós-implantação, melhorias substanciais em desempenho, correções de bugs ou adaptação ao ambiente (ZAIN et al., 2011).

A manutenção é inevitável. É preciso manter os softwares atualizados e eficientes. Qualquer sistema de software reflete o mundo exterior, e quando o mundo muda, o software precisa ser alterado conforme as mudanças ocorridas (SOUZA et al., 2005). A manutenção é realizada devido às seguintes razões: alterações nos requisitos, correções de problemas, mudança no hardware, melhoria de desempenho, modificações nos componentes, aperfeiçoamento no software, otimização do código fonte, melhoria da eficiência, organização do código fonte e eliminação de qualquer desvio de implementação relacionado aos requisitos (AGARWAL; TAYAL, 2007).

Comparando-se a evolução do software com o custo da manutenção, conclui-se que os modelos e métodos existentes ainda não estão completamente maduros, abrindo espaço para novas pesquisas exploratórias.

Dentre métricas e processos propostos para melhorar a manutenção, alguns estudos cognitivos relacionados a compreensão de software começaram a surgir. A compreensão está diretamente relacionada ao código fonte da aplicação. Sendo o código fonte o principal componente da manutenção, a compreensão do software é fator predominante para fornecer uma manutenção de software efetiva, permitindo a evolução dos sistemas informatizados (MAYRHAUSER; VANS, 1995).

A compreensão do software corresponde às atividades que as pessoas realizam para entender, conceitualizar e raciocinar sobre o software (MENG et al., 2006). É o processo de dar sentido a códigos fonte complexos, sendo que a compreensão tem sido considerada como conceitualmente complexa e computacionalmente difícil. A compreensão pode ser considerada como um processo de construir mapeamentos entre o conhecimento existente e artefatos (YANG et al., 2005).Estima-se que desenvolvedores gastam em média de 40% a 90% do esforço de manutenção no processo de compreensão do software (LUCIA et al., 1996),(TELEA; LUCIAN, 2011). Uma das possíveis razões para a dificuldade na compreensão do código fonte é a falta de conhecimento por parte das pessoas sem experiência, assim como programadores de outras áreas. Outras razões que afetam o impacto da compreensão são os fatores relacionados ao conhecimento, auto eficácia e organização do conhecimento, que estão inter-relacionados e afetam o desempenho dos programadores durante a tarefa de compreensão (WIEDENBECK, 2005).

Uma das formas de construir o conhecimento e dar sentido às coisas é por meio do processo de *sensemaking*. Define-se *sensemaking* como um processo de transformar circunstâncias em situações que sejam compreendidas explicitamente em palavras e que sirva como um local de partida para agir (WEICK et al, 2005). Weick considera a rotulagem (atribuição de nomes explícitos) como um dos passos essenciais no *sensemaking*.

A rotulagem é realizada por várias pessoas, com o objetivo de colocar ordem e categorizar, sendo esta rotulagem conhecida como folksonomia. Por meio de rotular, ou seja, realizar anotação a partir de um determinado objeto (STURTZ, 2004), é possível extrair e processar o conhecimento com base em artefatos existentes , em especial, no código fonte propriamente dito.

Sendo o conhecimento um dos fatores decisivos, é de suma importância que este seja disseminado e preservado pelos envolvidos na compreensão e manutenção. Engenheiros de software são colaboradores do conhecimento, visto que as tarefas desempenhadas nesta área requerem conhecimento e experiência. Cada experiência está relacionada a um fato ou experiência de projeto passado.

1.1 Motivação

Uma das exigências do mercado é responder e atuar de forma rápida e coerente as mudanças. Não basta ter profissionais experientes e especialistas em uma determinada área, é preciso interligar áreas e trabalhar em conjunto para atender um determinado objetivo. Atualmente, trabalham nas organizações, profissionais polivalentes com conhecimento generalizáveis, motivados pela grande mudança célere das necessidades do mercado (ROCHA-PINTO et al, 2009).

Para responder de forma rápida às exigências do mercado é preciso descentralizar o conhecimento. O conhecimento não está localizado e incorporado no processo produtivo, está em constante transformação e nas cabeças das pessoas, o que evidencia a necessidade de investimento em gestão do conhecimento (COCCO et al., 2003). A tarefa de gestão do conhecimento tem por objetivo tornar acessível os conhecimentos existentes, assim como apoiar no desenvolvimento de novos conhecimentos, arquivando-os em uma base de conhecimento.

Define-se gestão do conhecimento como um método que simplifica o processo de compartilhar, distribuir, criar, capturar e entender o conhecimento de uma corporação (DAVENPORT; PRUSAK, 1998). No entanto, o conhecimento precisa ser disseminado entre todas as partes envolvidas de um determinado projeto. Atualmente, a Tecnologia da Informação (TI) tem sido usada para apoiar a gestão do conhecimento, dando suporte à comunicação e troca de ideias e experiências (E-CONSULTING CORP, 2004). De vários meios e ferramentas de comunicação e troca de conhecimento, a Web Semântica pode facilitar a disseminação do conhecimento (ZHAO et al, 2009). Trata-se da evolução da web tradicional, correspondendo a uma rede de dados que pode ser processada direta e indiretamente por máquinas (BERNERS-LEE, 1999), provendo um framework que permite que os dados sejam compartilhados e reutilizados por aplicativos, empresas e comunidades em geral (GALL; REIF, 2008).

Nas atividades de manutenção, o conhecimento do domínio está incluído nos artefatos relacionados ao software. É na etapa de análise e compreensão que os envolvidos atuam para extrair este conhecimento e utilizá-lo para prosseguir com a manutenção. Durante esta atividade, o conhecimento adquirido é armazenado na memória das pessoas, se dividindo em duas classes (SHNEIDERMAN, 1992):

- **conhecimento semântico**: conhecimento de conceitos, como por exemplo, a operação de uma declaração de atribuição e a noção de uma classe de objeto. Este conhecimento é adquirido por meio da experiência e do aprendizado ativo;
- **conhecimento sintático**: conhecimento da representação detalhada, como por exemplo, descrever detalhadamente um objeto em UML *(Unified Modeling Language*), funções padrões de uma determinada linguagem de programação. Este conhecimento é retido de maneira não processada e a sua estrutura é arbitrária e desorganizada, sendo adquirido por meio da memorização.

Tanto o conhecimento semântico como o sintático estão diretamente ou indiretamente relacionados à compreensão do código fonte. Muitos estudos e modelos de compreensão identificaram diferentes tipos de conhecimento, dentre eles: conhecimento da programação, conhecimento da situação do mundo real abordado pelo software e conhecimento do domínio da aplicação (YANG et al., 2005).

A complexidade envolvida na codificação, processo pelo qual os desenvolvedores passam suas intenções para o computador, em especial no paradigma Orientado a Objetos (OO), implica no alto poder de processamento e armazenamento na memória das pessoas, pois além do domínio, é preciso mentalizar a organização dos objetos e do fluxo dos dados (CORRITORE; WIEDENBECK, 1998). A capacidade de trabalho com estruturas complexas de informação dificulta gradativamente a forma de como se codifica e gerencia os fontes e rotinas, principalmente a codificação de regras de negócio complexas altamente mutáveis. A dificuldade da organização de rotinas complexas juntamente com a grande quantidade de esforço aplicada na manutenção, aliada à ausência de uma solução ideal para o problema, motivou o desenvolvimento deste trabalho.

Acredita-se que um método de compreensão aplicado ao código fonte, relacionado à extração e disseminação do conhecimento poderá auxiliar no processo de compreensão, diminuindo as incertezas e tempo dedicados a estas tarefas.

Sendo assim, a questão principal que se pretende responder é: **“É possível diminuir o tempo de esforço da compreensão do código fonte e com isso aumentar a qualidade e a eficiência da manutenção de software?”.**

1.2 Objetivos

Este trabalho tem por objetivo geral **desenvolver um método baseado em sensemaking para reduzir o tempo de compreensão do código fonte orientado a objetos para manutenção de sistemas.**

O objetivo geral deste trabalho será atingido seguindo os seguintes objetivos específicos:

(i) Pesquisar a percepção dos programadores em relação às dificuldades para realizar a manutenção e compreensão de código fonte.

(ii) Conceber um método baseado em sensemaking.

(iii) Desenvolver um ambiente para apoiar o método proposto.

(iv) Avaliar o método proposto.

1.3 Estrutura do documento

O Capítulo 1, aqui apresentado, visa oferecer ao leitor um panorama geral sobre o contexto no qual se insere este trabalho de pesquisa.

O Capítulo 2 aprofunda o referencial teórico inicial descrito no Capítulo 1, focando especialmente os temas compreensão e manutenção, *sensemaking*, folksonomias, conhecimento, ontologia e Web Semântica.

O Capítulo 3 apresenta um posicionamento metodológico, caracteriza a pesquisa e define as estratégias para execução.

Os Capítulos 4, 5 e 6 descrevem, respectivamente, o método proposto, a seleção e caracterização do ambiente com a folksonomia, a implementação do método e os resultados obtidos de acordo com os objetivos definidos.

O Capítulo 7 apresenta as considerações finais da pesquisa, descrevendo a relevância do estudo e as contribuições da pesquisa.

Os Apêndices A e B descrevem as questões e o resultado da pesquisa de campo realizada com programadores.

1.4 Considerações sobre o capítulo

Este capítulo apresentou a necessidade de mais estudos no quesito compreensão de software para manutenção. Apresentou a importância da manutenção e as principais atividades envolvidas, assim como as motivações para a realização deste trabalho, os objetivos e a delimitação e estrutura do trabalho.

CAPÍTULO 2 - REVISÃO DA LITERATURA

A manutenção de software, considerada a atividade predominante na engenharia de software, é definida como qualquer modificação realizada no sistema após sua entrega (SEACORD et al., 2003).

Conforme o padrão IEEE 1219 (IEEE, 2004), manutenção de software é compreendida como uma modificação de um produto de software após sua entrega, com objetivo de corrigir falhas, adaptar o produto a um ambiente modificado, aperfeiçoar o desempenho ou outros atributos.

Existem quatro categorias fundamentais no processo de manutenção de software (SEACORD et al., 2003):

- **corretiva**: mudanças são feitas para realizar reparos de defeitos no sistema, ocasionados pela falha da atividade de testes.
- **adaptativa**: realizadas para manter o ritmo das mudanças ocorridas no ambiente, como por exemplo, um novo sistema operacional, linguagem de compiladores e ferramentas, sistemas de gerenciamento de banco de dados e demais componentes comerciais e hardwares.
- **perfectiva**: mudanças realizadas para melhorar o produto, como adicionar novos requisitos ou melhorar a performance, usabilidade ou novos atributos do sistema. Este tipo de manutenção também é conhecida como mudança de aprimoramento (*enhancement*).
- **preventiva**: ocorre com o objetivo de melhorar a confiabilidade ou manutenibilidade futura, antevendo transformações no software.

As manutenções corretivas, adaptativas, perfectivas ou preventivas são ocasionadas devido às alterações em nível de requisitos do software, alterações a nível de especificação funcional do sistema de software, alterações de interface a nível de especificação de performance, mudanças a nível do ambiente do sistema, a disparidade entre a especificação e a implementação do software e mudanças estratégicas necessárias (BADAREEN et al., 2011) .

Contudo, as mudanças ocasionadas tanto por erros quanto por melhorias apresentam custos significativos para o ciclo do desenvolvimento do software,

excedendo os custos iniciais de desenvolvimento ao longo do ciclo de vida, impactando no custo total. A Figura 2-1 ilustra o aumento do custo relativo a manutenção e evolução nas últimas décadas. Nota-se que o custo total além de aumentar gradativamente a cada década, já alcançou o patamar de 90% (ERLIKH, 2000)..

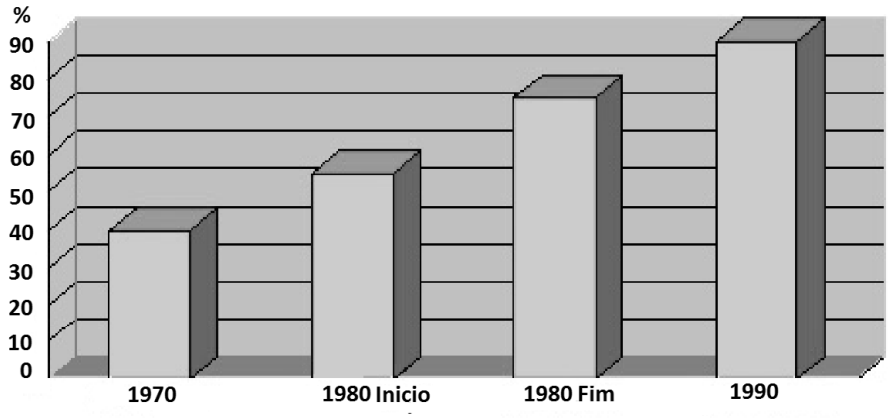

Figura 2-1 Evolução do custo da manutenção, adaptado de (ERLIKH, 2000)

Alguns dos problemas de manutenção relacionados ao aumento gradativo do custo da manutenção podem ser identificados. De acordo com (PIGOSKI, 1997), o custo de manutenção é muito alto, a velocidade de serviço de manutenção é muito lenta e há dificuldade de gerir a prioridade das solicitações de mudança. Entretanto, pela perspectiva interna, o ambiente de trabalho força os mantenedores a trabalhar com *design* de código de software mal projetados.

Muitos outros problemas podem ser citados, conforme estudos levantados por (DEKLEVA, 1992), dentre os mais importantes estão: gestão da rápida mudança das prioridades, técnicas de testes inadequadas, dificuldade de medir o desempenho, documentação de software em falta ou incompleta, adaptação as rápidas mudanças nas organizações do usuário, grande número de solicitações dos usuários em espera, dificuldade em medir/demonstrar a contribuição da equipe de manutenção, baixa estima devido a falta de reconhecimento, falta de profissionais na área (especialmente os experientes), pouca metodologia, poucos padrões, procedimentos ou ferramentas específicas para realizar a manutenção, integração de código fonte complexo e não estruturado (sobrepondo a incompatibilidade dos sistemas), pouca informação disponível ao pessoal da manutenção, inexistência de planos estratégicos para a manutenção, dificuldade em cumprir as expectativas do usuário, falta de compreensão e apoio dos gerentes de TI, softwares manutenidos rodando

em sistemas e tecnologias obsoletas e perda da expertise quando o funcionário deixa a empresa. Os problemas destacados envolvem todo o processo de manutenção, e não somente a codificação.

O processo de manutenção de software engloba várias tarefas, as quais são apresentadas na Figura 2-2 (BADAREEN et al., 2011) e explicadas a seguir:

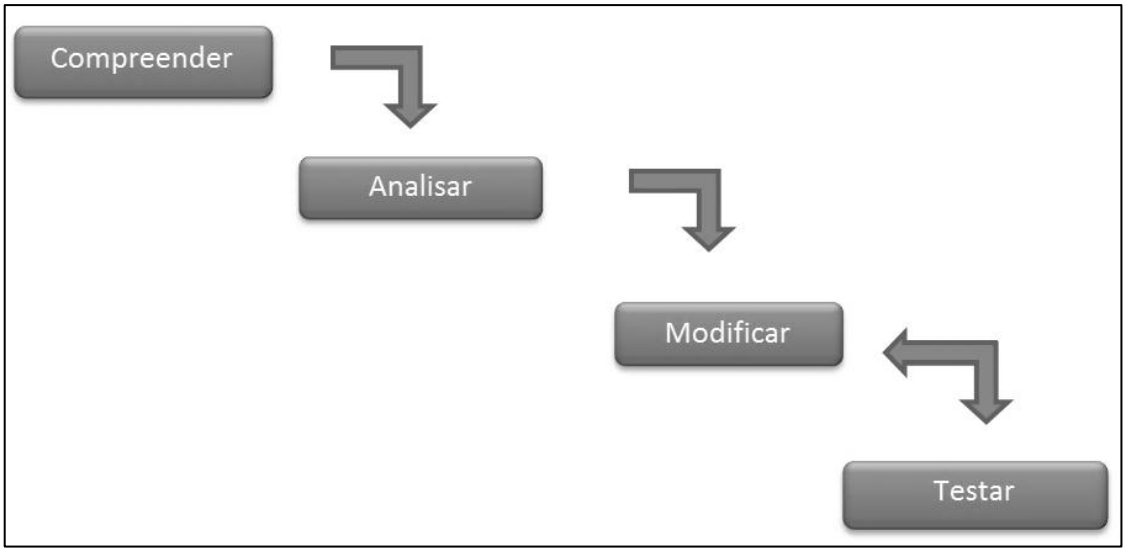

Figura 2-2 - Processo de manutenção de software, adaptado de (BADAREEN et al., 2011)

- **compreender**: processo de perceber, entender as funções do sistema e seus relacionamentos. É considerado como uma base essencial para inspecionar e modificar um produto de software. Os responsáveis pela manutenção precisam compreender toda a estrutura do sistema de software, identificando os componentes principais e suas propriedades, assim como relacionar os componentes identificados com seus níveis mais baixos (WITTE et al., 2007).
- **análise**: tarefa essencial, que tem por objetivo identificar a modificação necessária para corrigir, melhorar, ou adotar o sistema.
- **modificar**: tarefa de alterar e corrigir as funções inspecionadas dentro do sistema.
- **testar**: avalia se as alterações realizadas atingiram os objetivos das manutenções, considerando apenas as alterações que foram realizadas durante a tarefa de modificação.

De todas as atividades envolvidas no processo de manutenção, a compreensão é a mais importante, pois é considerada a base essencial para modificar um produto de software (WITTE et al., 2007). Estudos demonstram que os esforços aplicados na manutenção estão direcionados principalmente a parte de compreensão (CORRITORE; WIEDENBECK, 1999). Para (CORBI, 1989), a tarefa de compreensão corresponde a mais de 40% do total de esforço aplicado na manutenção.

O que faz com que os programadores tenham dificuldades para entender o software é resultado de pesquisas na complexidade do software, por meio da medição da dificuldade em respeito a compreensão do programa como uma complexidade de software (YANG et al., 2005). Basili (BASILI, 1980) define que a complexidade do software é uma medida dos recursos despendidos por um sistema interagindo com um pedaço de software para executar uma determinada tarefa. Dentre algumas medidas criadas, com ênfase na dificuldade de *debugar*, codificar, testar, atender requisitos não funcionais, alguns autores separaram em dois grupos em termos de complexidade: complexidade computacional e complexidade psicológica (YANG et al., 2005).

A complexidade computacional está relacionada a aspectos técnicos, como aspectos quantitativos referentes a uma solução desejada à resolução de problemas computacionais. A complexidade psicológica refere-se as características do software que o tornam difícil de entender e trabalhar.

Um estudo realizado por (MATHIAS et al, 1999) demonstrou que os programadores tendem a mudar suas estratégias de codificação a medida que o número de linhas de código (LOC) aumenta, pois a complexidade de compreensão está relacionada ao número de LOC do código fonte. (YANG et al., 2005) estudou a complexidade de entender um programa no momento de manutenção, para efeitos de cálculos e métricas de estimativas de esforço.

A compreensão do código é uma atividade ligada a questões psicológicas, que estudam como as pessoas montam e abstraem modelos mentais para trabalhar com grande volume de dados para a escrita e entendimento do código fonte e está relacionada à recuperação de informações a partir do código fonte. Estas atividades, que correspondem a modelos de compreensão, podem ser categorizadas em três modelos: *top down, bottom up* e ambos (MATHIAS et al, 1999).

O modelo *top down*, proposto por (SOLOWAY et al, 1988), tem origem no alto nível de abstração. Conforme pesquisa realizada pelos autores, programadores preferem estratégias para certos loops, pois requer menos esforço cognitivo para entender, ocasionando menos erros. As habilidades adquiridas conforme o tempo, nomeadas de “objetivos e planos” (*goals and plans*), correspondem a um “enlatado”, utilizado para resolver um certo tipo de problema. Na medida que o programador desenvolve habilidades, ele desenvolve um repositório de planos, possibilitando adicionar e mesclar com outros para resolver o problema.

O modelo *bottom up*, desenvolvido por (PENNINGTON, 1987), é composto por dois modelos que representam o conhecimento: *program model* e *situation model*. *Program model* explica como a construção de um código específico trabalha, enquanto o *situation model* explica o por que do código fazer tal trabalho.

Nos trabalhos realizados por (MAYRHAUSER; VANS, 1995) conclui-se que a compreensão de programas envolve a integração de quatro modelos: *top down domain model, program model, situation model e knowledge base*, ou seja, a integração do modelo *top down* e *bottom up*.

Apesar de certas informações serem possíveis de obter por meio da engenharia reversa, o documento final gerado não é tão rico quanto aquele obtido por meio da extração do conhecimento utilizando outros métodos (RILLING et al, 2006). É trabalhoso transformar o código fonte em visão de programa e semântica de forma automática. As variações do código, como por exemplo nos diagramas de classes gerados a partir do código, são difíceis de entender, pois o código está sempre organizado envolto de funções específicas, ao invés de conceitos de domínios específicos. Extrair um diagrama de classes do código não introduzirá nenhum conhecimento inferido (*inferred knowledge*), acarretando na falta do contexto e do conhecimento explícito do domínio (ZHOU et al, 2008).

Para contornar este problema, o trabalho de (WITTE et al., 2007) propôs conectar o código fonte com artefatos de documentação de software, utilizando ontologia, *Web Ontology Language* (OWL), população de ontologia, Web Semântica e mineração de texto, com objetivo de manter a rastreabilidade do código com artefatos para realizar uma manutenção melhor.

O acesso a artefatos e demais informações relevantes ao software auxilia no processo de compreensão. No entanto, nem sempre estas informações estão acessíveis.

Ao se deparar com uma nova aplicação, a pessoa que irá realizar a manutenção não estará familiarizada com a organização dos fontes, das classes, métodos, nomenclaturas, pacotes e arquitetura. Para prover um melhor entendimento e compreensão de algo novo e desconhecido, o *sensemaking*, ou seja, o modelo de fazer sentido as coisas, pode ser utilizado como um *framework* de apoio às atividades de compreensão.

Sensemaking é uma tentativa de lidar com a ambiguidade e a incerteza (WEICK et al, 2005). A manutenção de software se enquadra nestas situações, cuja metodologia de trabalho se adequa como uma instância do *sensemaking*.

2.1 Sensemaking

Sensemaking envolve circunstâncias, transformando-se em situações que são explicitamente compreendidas em palavras e que servem como local de partida para tomada de ação. Corresponde a uma construção teórica que contém os mecanismos cognitivos e sociais para lidar com ambiguidade e a incerteza (WEICK et al, 2005).

Os mecanismos cognitivos e sociais se relacionam ao comportamento humano, sendo que o comportamento é como um processo em curso, de forma indeterminada e menos intencional. Neste aspecto, o *sensemaking* idealiza o comportamento como algo mais indeterminado e menos intencional, minimizando as pessoas como atores racionais (WEICK, 1995). A ideia básica de *sensemaking* para (WEICK, 1993) é que a realidade é uma realização contínua que surge dos esforços para criar ordem e sentido.

(DERVIN, 1992) define *sensemaking* como "uma rede teórica, um conjunto de pressupostos e proposições, e um conjunto de métodos que têm sido desenvolvidos para estudar a produção de sentido que as pessoas fazem em suas experiências cotidianas*".*

O processo de *sensemaking* foi estruturado e desenvolvido a partir do campo da teoria da dissonância cognitiva (FESTINGER, 1957). Esta teoria diz respeito à racionalização retrospectiva que indivíduos comprometem-se a tentar restaurar as consistências em seus modelos mentais quando confrontado com a dissonância “pós decisão”.

Geralmente, os modelos mentais criados por pessoas que estão prestes a tomar alguma decisão, são modelados em favor a uma escolha, dentre várias alternativas. Este comportamento de decisão, conforme (WEICK, 1995), é visto

quando as pessoas se deparam com um choque em suas experiências da realidade e isto corresponde ao processo de *sensemaking*.

Desta forma, o *sensemaking* se inicia pelo caos, onde vários dados, muitos destes sem sentido, são acatados para posterior processamento. Para indivíduos, trata-se da construção de um modelo mental da situação vigente. Para a organização, é um processo de criação colaborativa e compartilhada de consciência e de compreensão de perspectivas e interesses variados. O *sensemaking* não começa novamente depois de novos fatores, mas tudo ocorre em meio a organizar um fluxo de potenciais antecedentes e consequências (CHIA, 2000).

Estes antecedentes são fatos e acontecimentos ocorridos durante o processo de organização do *sensemaking*, que venham a ocasionar a invenção de novos significados para algo que já tenha ocorrido no passado, mas que nunca fora reconhecido como um objeto, evento ou como um processo autônomo (MAGALA, 1997).

Os eventos descobertos pelas pessoas são processados e, consequentemente, passíveis de discussão com demais pessoas, de diferentes níveis de conhecimento e experiências. As pessoas discutem eventos por meio de uma mistura de vocabulários de diferentes níveis. (WEICK, 1995) define seis vocabulários que agem como substância para o *sensemaking*: ideologias em nível social, controle de terceira ordem em nível organizacional, paradigmas de ocupação, teorias-de-ação de indivíduos, tradições como vocabulários de antecessores, e histórias como sequência de vocabulários e experiências. Com palavras as pessoas colocam ordem no mundo desenhado a partir de diversas fontes.

Quando uma história, que corresponde a fatos, acontecimentos e observações, é retida pela pessoa, tende-se a tornar mais substancial porque está relacionada com experiências passadas, conectada a identidades significativas, sendo utilizada como uma fonte de guia para mais ações e interpretações (WEICK et al, 2005). Desta forma, não se tem por objetivo obter a resposta certa, nem a verdadeira, mas a história é constantemente reformulada, de forma a se tornar mais compreensiva possível.

Os oito estágios do *sensemaking,* idealizados por (WEICK et al, 2005) sob o ponto de vista descritivo, abrangem tanto aspectos individuais como organizacionais. Estes estágios dão continuidade à definição e caracterização do *sensemaking*, com exemplos de aplicação no processo de manutenção e compreensão:

- **Sensemaking organiza o fluxo**

O processo inicial do *sensemaking* é o caos. Desta forma, *sensemaking* é tido como um processo sistemático de organização, confrontando o caos, reparando, e destacando observações para tentar encaixar estes fenômenos dentro de uma estrutura que permite ao observador lidar e dar sentido a sua experiência vivida (WEICK et al, 2005). O tema central do *sensemaking* é organizar para fazer sentido as entradas equivocadas, tornando-as mais organizadas.

- **Sensemaking inicia com a observação e suporte (*bracketing*)**

Para (KOLB, 1984), a aprendizagem a partir de observações e experiências concretas se resume a conclusões, que, quando tal conclusão é validada e privada de seu aspecto emocional, pode-se, então, se transformar gradualmente em conhecimento.

De forma geral, os primeiros estágios do *sensemaking* devem ser esculpidos de forma forçada do fluxo indiferenciado de experiências brutas e conceitualmente fixado e rotulado para que possa se tornar uma espécie de “moeda comum” na troca comunicacional (CHIA, 2000).

- **Sensemaking está relacionado à rotulagem**

Uma vez escolhido o estímulo do fluxo de experiências, o *sensemaker* procede com a rotulagem e categorização destes estímulos. O processo de rotular não é uma característica exclusiva e pessoal. O fato de rotular os estímulos colabora com a atribuição de atributos ao evento (estímulo), o que permite que outras pessoas envolvidas no processo de *sensemaking* reconheçam e reajam a estes estímulos.

Este é o estágio que segue após o reconhecimento de um processo autônomo. A análise e consideração de fatores passados, ou seja, a retrospectiva, além de ser importante, pode também, ser rejeitada e ignorada, devido ao tipo do acontecimento relacionado ao estímulo (CHIA, 2000), (WEICK et al, 2005).

- **Sensemaking é retrospectivo**

As conclusões dos atos são estabelecidas na medida que as pessoas olham para trás em suas respectivas observações e percebem um certo tipo de padrão.

Isto pode ser visto a partir dos rótulos e das categorizações realizadas. No entanto, a rotulagem em si não consegue captar a dinâmica do que está acontecendo (WEICK et al, 2005).

Erros e diagnósticos são conhecidos no momento depois da atividade, descritos como “cognições complexas da experiência de agora e depois. Eles identificam o atraso da compreensão humana”(PAGET, 1988).

- **Sensemaking está relacionado a presunção**

O mecanismo usado pelo *sensemaker* para iniciar a fase de interpretação é chamado de presunção (WEICK et al, 2005). Em essência, o *sensemaker* observa um fenômeno, seleciona-o, rotula-o e então prossegue para o estado de afirmar algumas hipóteses possíveis quanto à relevância e ao efeito da observação.

- **Sensemaking é social e sistêmico**

O processo de *sensemaking* é influenciado pelos vários fatores sociais, como discussões em grupo e contatos com sistemas externos (WEICK et al, 2005). A observação de um fenômeno e a discussão com outras pessoas influencia o *sensemaker* a catalogar e classificar tal acontecimento, a dar significado as ações e tirar conclusões referentes às hipóteses formuladas.

- **Sensemaking está relacionado a ação**

Uma pergunta que um *sensemaker* poderia fazer ao observar um fenômeno é “O que está acontecendo aqui?” e, consequentemente, a segunda pergunta de igual importância seria “O que eu devo fazer depois?”, diretamente relacionada à tomada de uma ação. Uma ação, assim como uma conversa, é tratada como ciclos, ao invés de uma sequência linear (WEICK et al, 2005). A discussão ocorre de forma precoce e tardia, assim como a ação e discussão podem ser designados como “um ponto de partida para o destino”. Para (WEICK et al, 2005), a ignorância e o conhecimento coexistem, o que significa que o *sensemaking* adaptativo honra e rejeita o passado.

- **Sensemaking está relacionado a organização através da comunicação**

Comunicação é o componente central do *sensemaking*. A comunicação é vista como um processo contínuo de fazer sentido as circunstâncias em que as pessoas “se encontram” e aos eventos que as afetam (WEICK et al, 2005).

De forma interativa e envolvente, a comunicação é articulada pelos envolvidos. Define-se articulação como um processo social que faz o conhecimento tácito mais explícito ou usável (OBSTFELD, 2004). (TAYLOR, VAN EVERY, 2000) complementa como uma situação é conversada dentro do ser por meio da troca interativa dos membros organizacionais para produzir uma visualização das circunstâncias, incluindo as pessoas, seus objetos, suas instituições, história e suas localizações em um tempo e lugar finito. Um exemplo é como uma pessoa transmite seu conhecimento, por meio da comunicação oral, para outra pessoa de maior conhecimento sobre o assunto, que, em seguida, rearticula com termos relevantes à pessoa de interesse (terceira pessoa), absorvendo a complexidade da situação por tratar da perspectiva da primeira e terceira pessoas envolvidas.

A imagem do *sensemaking* que fala de eventos e organizações sugere que os padrões de organização estejam localizados nas ações e conversas e nos textos destas atividades que são preservadas em estruturas sociais. O que ocorre neste contexto é o processo social no qual o conhecimento tácito é transformado em explícito e usável.

As atividades não só da manutenção e compreensão, mas da engenharia de software, demandam muita informação e conhecimento. Para organizações de software, os principais ativos não são fábricas, prédios e máquinas, mas sim o conhecimento guardado pelos funcionários (BJORNSON, DINGSOYR, 2008).

A Figura 2-3 ilustra o modelo mental adotado pelo *sensemaking*, com os fatores externos que influenciam no processo como um todo. Conforme ilustra a Figura 2-3, o *framework* mental é alimentado a partir de estímulos de ações. Por meio da retrospecção, o *framework* mental é realimentado, por incertezas e estranhezas. Desta forma, através de treinamento e implementação, o *framework* mental se altera constantemente, substituindo percepções por outras percepções (SELIGMAN, 2000).

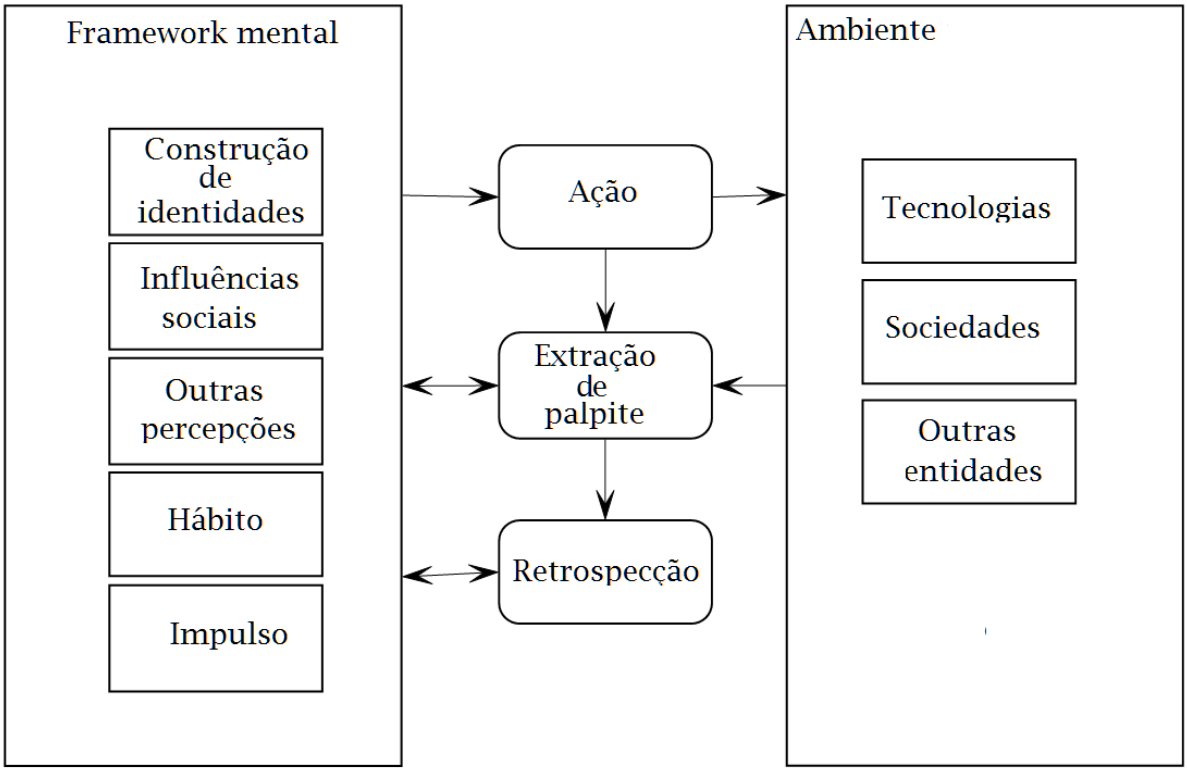

Figura 2-3 - Ciclo do Sensemaking, adaptado de (SELIGMAN, 2000)

Com base nos estágios do processo do *sensemaking*, é possível fazer uma analogia das atividades que estão relacionadas a manutenção e compreensão de software. Porém, é importante destacar como e por que as pessoas têm dificuldades para trabalhar com grande estruturas de dados contidas no código fonte.

2.2 Limitação da aquisição do conhecimento na manutenção de software

Sabe-se que uma pessoa motivada é o fruto para o sucesso de qualquer atividade a ser desempenhada. A motivação está ligada a fatores cognitivos e sociais. Para (HAZZAN, 2003),, a Engenharia de Software é uma atividade cognitiva e social, destacando a importância da compreensão de como as pessoas escrevem software.

É por meio das habilidades cognitivas que cada indivíduo expressa sua inteligência, educação e experiências. No entanto, a maioria do conhecimento armazenado na memória do ser humano apresenta restrições básicas, em decorrência de como essas informações são modeladas e armazenadas no cérebro das pessoas.

Sommerville (Sommerville, 2003) classifica a memória humana em três áreas distintas:

- **memória de capacidade limitada, acesso rápido e de curto prazo**: utilizada para processamento das informações e não para armazenar. Tipo de memória comparável aos registros de um computador;
- **memória de trabalho, com maior capacidade**: utilizada para o processamento das informações, com período de retenção mais longo do que a memória de curto prazo. Tipo de memória comparável a memória RAM (*Randomic Access Memory*) de um computador;
- **memória de longo prazo**: memória de grande capacidade de armazenamento, porém, tempo de acesso lento e não confiável. Utilizada para o armazenamento permanente das informações. Tipo de memória comparável ao disco de um computador.

Quando ocorre a entrada de mais informações que a memória de curto prazo pode lidar, poderá acarretar em erros e perda de informação, devido à necessidade da transferência de informações durante o processo de entrada. A Figura 2-4 ilustra o processo de percepção de novas informações e a assimilação com o conhecimento existente, nas memórias de curto e longo prazo.

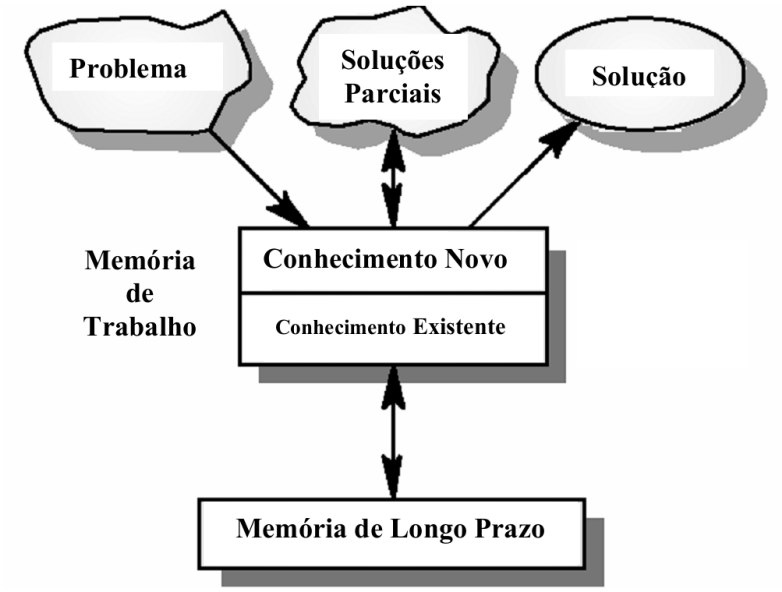

Figura 2-4 - Processo de aquisição de conhecimento para resolução de novos problemas. Adaptado de (SOMMERVILLE, 2003)

Os leitores de programas abstraem, em alto nível, as informações por meio da integração semântica interna, sendo compreendidos apenas quando a declaração representar um aglomerado lógico para que, somente assim, seja transferido para a memória de longo prazo.

A maneira como os programadores experientes aprendem uma nova linguagem de programação e abstraem os conceitos está ligada aos dois tipos de conhecimentos. Os conceitos de programação, como *loops*, declarações condicionais, atribuições, por exemplo, são aprendidos sem dificuldades, devido aos conhecimentos destes conceitos a partir de outras linguagens de programação. Neste caso, ocorre o conhecimento semântico. Já a sintaxe de programação da nova linguagem se torna um pouco mais difícil de abstrair, pois neste caso o programador tende a misturar as sintaxes de outras linguagens de programação que se está familiarizado, por meio do conhecimento sintático, conforme demonstrado na Figura 2-5 (SOMMERVILLE, 2003).

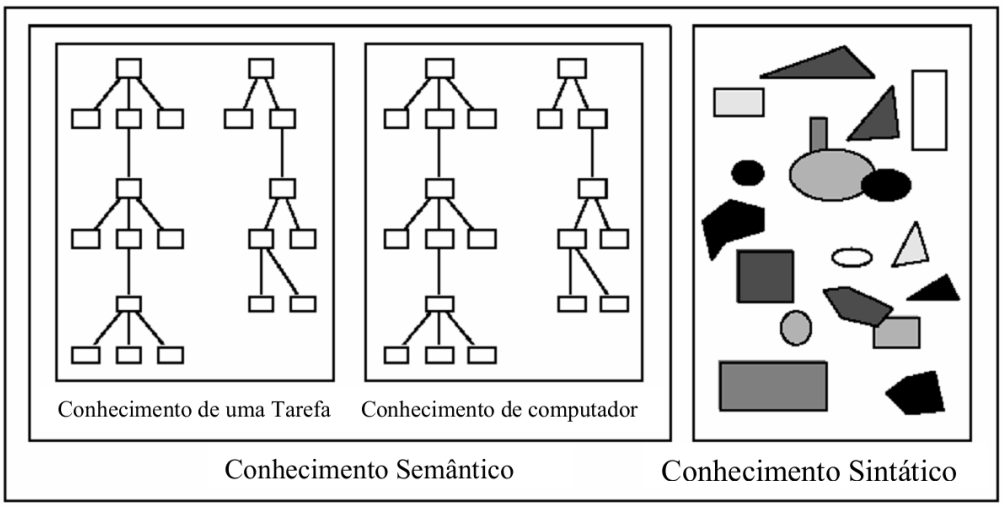

Figura 2-5 - Organização do conhecimento semântico e sintático. Adaptado (SOMMERVILLE, 2003)

A forma de escrever um programa se assemelha com a de resolver algum tipo de problema. O primeiro estágio envolve a declaração do problema na memória de trabalho a partir da memória de curto prazo, integrando-se com o conhecimento existente na memória de longo prazo. Como resultado, tem-se um programa executável. A solução desenvolvida (programa executável) é fruto de um modelo semântico interno do problema e da solução. Este modelo semântico é livre de erros se a linguagem de programação incluir construções que combinem com as

estruturas semânticas de nível inferior, que o desenvolvedor compreende. Desta forma, a linguagem de alto nível Java, deve conter menos erros do que os códigos escritos em baixo nível, como o assembler (SOMMERVILLE, 2003).

Desta forma, a programação estruturada está sujeita a menos erros do que a orientada a objetos. A programação estruturada tem como base conceitos semânticos, além de não sobrecarregar a memória de curto prazo do programador. Programas estruturados podem ser lidos de cima para baixo e de forma sequencial, o que facilita seu entendimento. As abstrações são feitas sequencialmente. A memória de curto prazo pode se dedicar a uma única seção de código, evitando a necessidade de recuperar informações sobre outras partes do programa que interfiram com aquela seção.

2.2.1 Sensemaking e Manutenção

No entendimento da codificação, certos trechos de código, como regras de negócio complexas, exigem uma melhor atenção do programador, assim como funções e classes que idealizam o sistema, de forma a se tornarem essenciais para a execução do software. Esta percepção inicia o estado do *sensemaking*.

O programador é guiado a partir de modelos mentais que são adquiridos durante seu trabalho atual, como experiências passadas. Estes modelos servirão de apoio para ajudá-lo e guiá-lo durante suas atividades para fazer sentido.

No âmbito da análise do fonte, funções, classes, variáveis e pacotes podem ser rotulados e classificados conforme sua funcionalidade, seu significado ou valor percebido ao projeto, auxiliando a criar os primeiros significados aos pedaços de código.

Um significado de uma função ou rotina pode não ser descoberto numa primeira tentativa de análise do código, mas sim, numa atividade relacionada à compreensão que fez com que o programador analisasse a mesma rotina, num momento posterior, percebendo seu real significado. Suponha-se que o autor analisou um trecho de código e mapeou sua funcionalidade; algumas horas mais tarde, o mesmo autor voltou a analisar o mesmo trecho e descobriu um certo padrão.

Descobrir o significado é a mesma coisa que fazer sentido. Fazer sentido é conectar o abstrato com o concreto. Na prática de programação, e consequentemente, na manutenção e compreensão, enquanto classes são instâncias do concreto, o conhecimento envolvido pela classe é abstrato.

Interpretação e experimentação envolve o concreto. Já a idiossincrasia e o pessoal relacionam com o abstrato e impessoal (PAGET, 1988).

Durante os trabalhos, o programador troca experiências com outros profissionais que, consequentemente, o auxiliam em suas tarefas. Se o conhecimento sobre uma correção ocorre aos poucos, então o conhecimento descoberto não está localizado apenas dentro da cabeça do especialista. Em vez disso, está em todo o sistema e é realizado em coordenação e distribuição de informações entre os profissionais envolvidos (WEICK et al, 2005).

Programadores, assim como qualquer outro profissional, criam sentido pensando, o que significa que eles interpretam, simultaneamente, seus conhecimentos com *frameworks* (modelos) confiáveis. Mas a desconfiança destes *frameworks* é testada com novos *frameworks* e novas interpretações.

Sendo a classificação e a rotulação (assimilação de nomes explícitos) os dois passos essenciais no processo de *sensemaking*, os programadores não só comentam o código, mas também “tagueam” e adicionam significado a trechos de código, com o objetivo de expressar suas compreensões ao referido código. Neste cenário, o mecanismo de taguear[1], identificar e destacar trechos do código, se assemelha ao processo da folksonomia.

2.3 Folksonomia

Folksonomia é a junção das palavras *folk* (pessoa) e *taxonomy* (taxonomia). Entende-se como “classificação do povo”. É uma forma relacional de categorizar e classificar coisas na web (PEREIRA, 2006). Exemplos da folksonomia podem ser vistos nos sites Flickr, Del.icio.us, Technnorati, Writely e Amazon (Figura 2-6).

(WAL, 2005) define folksonomia como o resultado de uma marcação pessoal livre de informações e objetos (qualquer coisa com uma URL) para uma recuperação. A marcação é realizada em um ambiente social, compartilhado e aberto para outros. O ato da marcação é desempenhado pelas próprias pessoas que consomem a informação. O valor na marcação é derivado de pessoas usando seus próprios vocabulários, adicionando significado explícito, que pode ter origem de uma compreensão inferida da informação/objeto.

[1] Palavra comumente utilizada na língua portuguesa para traduzir o termo *tagging*, que corresponde a rotular, etiquetar, identificar.

Figura 2-6 - Tags apresentadas com a opção de adicionar uma nova tag a um item apresentado no site www.amazon.com

A coleção de todas as categorizações realizadas por um mesmo usuário é conhecida como sua personomia. O conjunto de todas as personomias, quando compartilhadas com outros usuários do sistema, constitui uma folksonomia (WAL, 2007), (HOTHO et al, 2006) E (JÄSCHKE et al, 2008). De maneira dinâmica, a personomia evolui na medida que o usuário revisa e aprende com seus conhecimentos.

Uma das características da folksonomia é a liberdade oferecida para os usuários anotarem recursos com diferentes *tags*. No entanto, essa liberdade acarreta no problema de ambiguidade, como o uso de acrônimos, sinônimos, sinonímia (diferentes etiquetas com o mesmo significado) e polissemia (mesma etiqueta com significados diferentes) (HOTHO et al, 2006), (WU et al, 2006).

A folksonomia se caracteriza em dois grupos (WAL, 2005): extensa (*broad*) e limitada (*narrow*). A característica extensa compreende o resultado de vários usuários anotando o mesmo recurso, observando as anotações realizadas em comum. A característica limitada é o resultado de um usuário anotar um recurso, ou um número menor de anotadores.

Dentre as características citadas é possível resumir as vantagens e desvantagens da utilização da folksonomia. O Quadro 2-1 identifica estes principais pontos positivos e negativos, conforme (KROSKI, 2005), (QUINTARELLI, 2005), (MATHES, 2004) e (LIMPENZ et al, 2008).

Quadro 2-1 Pontos positivos e negativos da folksonomia

Pontos positivos	***Pontos negativos***
Vocabulário não controlado, representando vocabulário genérico. Não são inclusivas.	Imprecisas, devido à existência de *tags* ambíguas, personalizadas e inexatas.

Criação de *tags* e anotações de forma rápida.	Possuem termos compostos e inúteis.
Tags aplicáveis em várias categorias.	Falta de controle para criação de sinônimos e homônimos.
Pelo fato das *tags* serem consideradas subjetivas é possível observar o comportamento dos usuários.	*Tags* mais especializadas e mais gerais podem se referir ao mesmo objeto.

Conforme sua característica, a folksonomia é tida como abordagem *bottom-up*, onde o domínio não é criado por especialistas e seu modelo não segue uma hierarquia explícita nem vocabulários específicos, conforme as taxonomias (SHIRKY, 2005) e (QUINTARELLI, 2005). Devido a isto, a folksonomia surgiu com o propósito de diminuir o problema na aquisição de conhecimentos (WAL, 2007),(HOTHO et al, 2006). Os resultados da aplicação da folksonomia são meta-informações visuais geradas pelos usuários. Estas informações, gerenciadas, podem ser guardadas e disseminadas posteriormente.

O conhecimento adquirido pode ser guardado e caracterizado a partir do uso de ontologias. A utilização de ontologias em conjunto com a folksonomia, contorna os pontos negativos apresentados, além de possibilitar o emprego de características semânticas para a evolução da folksonomia.

2.4 Ontologias

Nas últimas décadas na Ciência da Computação, emergiu o conceito de ontologias, sendo utilizada em três importantes áreas: sistemas de informação e base de dados; engenharia de software; e inteligência artificial (SMITH, WELTY, 2001).

Para o campo da ciência da computação, ontologia é utilizada como contexto de compartilhamento de informação para troca de informação entre aplicações. Para a comunidade de Inteligência Artificial a ontologia é empregada para o compartilhamento de conceitos e reúso (LACY, 2005).

Ontologia é um catálogo de tipo de coisas sobre um domínio (SOWA, 2000). Corresponde a “uma teoria que diz respeito a tipos de entidades e, especificamente, a tipos de entidades abstratas que são aceitas em um sistema como uma linguagem”.

Uma das definições mais difundidas é apresentada por (GRUBER, 1993) como “... uma especificação explícita de uma conceitualização”. O termo conceitualização corresponde a uma coleção de conceitos, objetos e demais entidades existentes em um domínio e os relacionamentos entre eles (GENESERETH, NILSSON, 1987).

A definição de Gruber, pelo fato de ser a mais explorada, é discutida por (GUARINO, 1998). Guarino discute a interpretação da conceitualização adotada por Gruber, afirmando não corresponder à intuição desejada. “Uma conceitualização é um grupo de relações extensionais (enumeração de aspectos de todas as espécies que são do mesmo nível de abstração) descrevendo um ‘estado das coisas’ particular, enquanto a noção que temos em mente é uma relação intencional (lista de características do conceito), nomeando algo como uma rede conceitual a qual se superpõe a vários possíveis ‘estados das coisas’”.

A revisão por (GUARINO, 1998) da definição de conceitualização considera o aspecto intencional para obter uma interpretação mais satisfatória: “ontologia se refere a um artefato constituído por um vocabulário usado para descrever uma certa realidade, mais um conjunto de fatos explícitos e aceitos que dizem respeito ao sentido pretendido para as palavras do vocabulário”. As palavras aparecem como predicados unários ou binários, por meio da teoria da lógica de primeira ordem. Os vocabulários originados por predicados lógicos formam a rede conceitual que define o caráter intencional às ontologias.

Para (BORST, 1997), uma ontologia é a especificação formal de uma conceitualização compartilhada. Formal pelo fato de uma ontologia ser processada por máquinas, e compartilhada por refletir o conhecimento consensual de um grupo de especialistas.

Para (WEISS, 1999), ontologia é um objeto de especificação de conceitos e relações em uma área de interesse, onde ontologia é mais do que uma simples taxonomia de classes.

Diferentes definições são elaboradas sob diversos pontos de vistas e áreas. Algumas definições são criadas para serem independentes do processo, enquanto outras são influenciadas pelo processo de desenvolvimento (CORCHO et al, 2003) .

Alguns dos problemas relatados da folksonomia podem ser contornados com o emprego da ontologia, associando um ou mais conceito às *tags*. Desta forma, a

folksonomia é estruturada, organizada e desenvolvida conforme os conceitos mapeados na ontologia.

2.4.1 Folksonomia e Ontologias

Segundo (BERNERS-LEE et al, 2001) a folksonomia falha quando os usuários precisam de processamento mais complexo. Para isto, são utilizadas representações semânticas e as ontologias são a chave para preencher o vazio deixado pela folksonomia (HAMMOND et al, 2005), permitindo classificar o conteúdo referenciando a um vocabulário controlado (idealizado por especialistas) e os metadados fornecidos pelo uso das ontologias são interpretáveis pela máquina. A Figura 2-7 ilustra a organização e classificação das tags por meio da folksonomia e ontologia.

A folksonomia não representa explicitamente uma conceitualização compartilhada, devido à presença de ambiguidade e de uma organização plana do relacionamento de anotações. Por outro lado, as ontologias são inequívocas, claras e sua relação é semanticamente rica. Desta forma, (DOTSIKA, 2009) afirma que a folksonomia e a ontologia desempenham um papel semelhante ao de sistemas de classificação de conteúdo Web, apesar de serem formalizadas de diferentes maneiras, conforme visto na Figura 2-7.

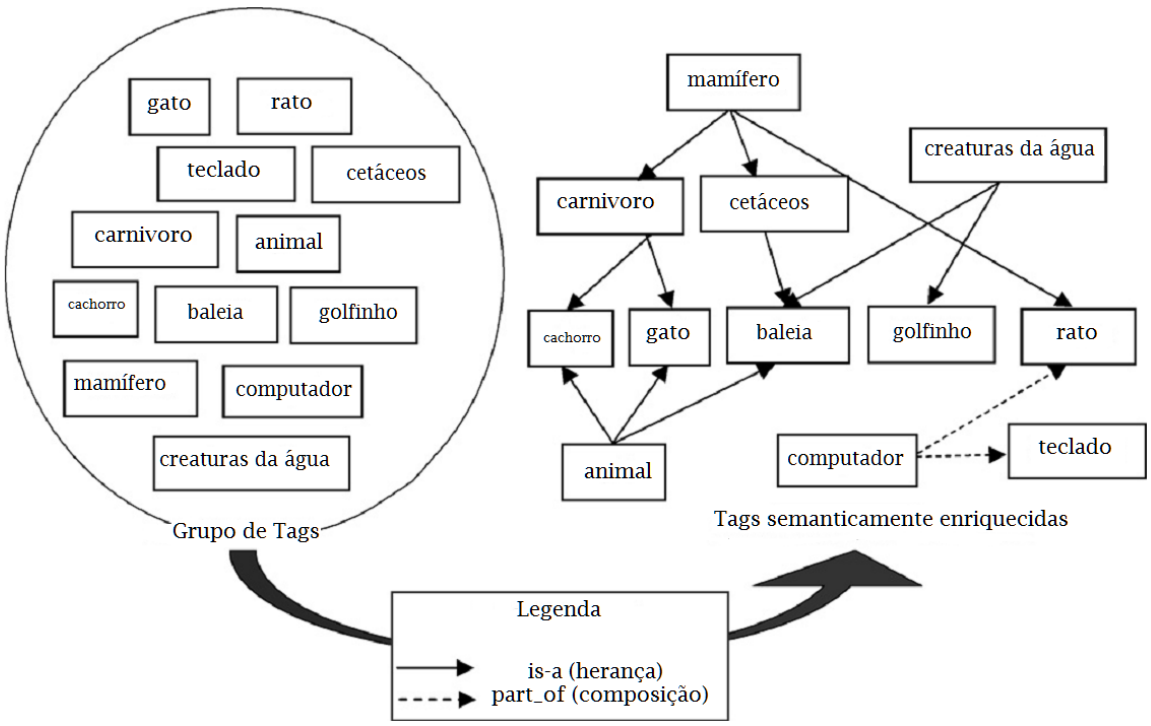

Figura 2-7 - Enriquecimento semântico da folksonomia. Adaptado de (DOTSIKA, 2009)

Um dos problemas da folksonomia é a falta de revocação. A revocação reflete a habilidade do sistema de localizar todos os registros relacionados a um tópico. Devido à falta de controle de sinônimos, a folksonomia é quase inutilizável para buscar informações específicas (HSIEH et al, 2008). Este é um dos problemas que os ontologistas devem contornar. Um sistema pode utilizar o conceito *tag-on-tag* para destacar que "esta *tag* é sinônimo daquela *tag*" e um outro sistema pode possuir o conceito de "esta *tag* representa um conjunto de outras *tags*". Não há exigências de que todos os sistemas compartilhem o mesmo conceito. O compartilhamento bem sucedido do conhecimento requer apenas que os sistemas claramente identifiquem as diferenças quando eles compartilharem dados (GRUBER, 2007).

2.4.2 Ontologias para o processo de tagueamento

O processo de identificação de um objeto é caracterizado como etiquetar, ou taguear um conteúdo criado com um ou mais rótulos ou *tags* (GRUBER, 2007). Para que seja possível taguear, é preciso distinguir o sujeito e o objeto. Sujeito é a pessoa, ou qualquer outro agente, que está identificando algo com uma *tag*. Objeto é qualquer item eletrônico (texto, documento, e-mail, palavra, etc) que esteja acessível ao sujeito.

Alguns autores (MIKA, 2005) (HALPIN et al, 2006) sugerem que o processo de tagueamento deva ser representado pela tripla *Tagging (U, T, R)*, onde *U* representa o conjunto de usuários, *T* o conjunto de *Tags* e *R* o conjunto de itens/objetos (*resource*) sendo tagueados. Neste modelo, também chamado de "*tri-concepts*", é possível utilizar a mesma *tag* para identificar diferentes objetos, por diferentes usuários. Enquanto que Newman (NEWMAN, 2005) (Figura 2-8) descreve o item como resource (*Thing*), Gruber (GRUBER, 2007) classifica-o como *object*, adicionando o item *source* como o espaço onde a ação de tagueamento ocorreu.

Enquanto que *resource* idealiza objetos, pois corresponde a uma única fonte, o *source* corresponde a fontes ou aplicações que utilizam a folksonomia (ECHARTE et al, 2007). O mesmo se aplica a usuário e a um conjunto de usuários. Desta forma, a evolução da tripla anterior pode ser idealizada adicionando mais informações para atender os pré-requisitos da folksonomia: *source* e *userGroup.*

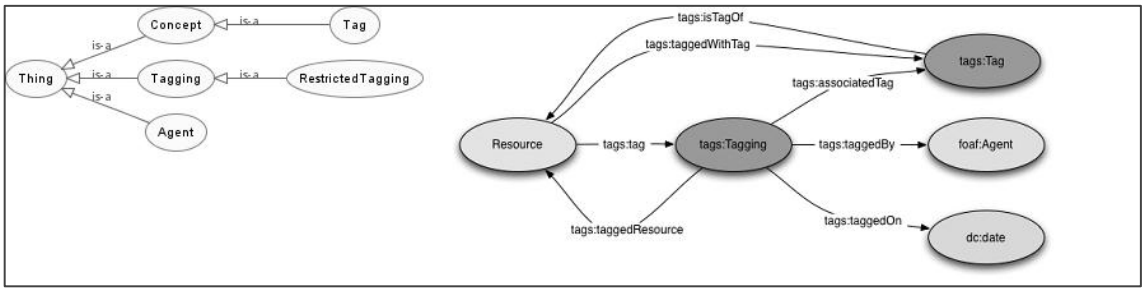

Figura 2-8 - TagOntology de Newman

Gruber (GRUBER, 2007) adiciona a propriedade *source*, que corresponde ao espaço onde a ação de tagueamento foi realizada. Alguns autores seguiram esta ideia, aperfeiçoando para atender os requisitos da folksonomia. Moat (PASSANT, 2007) estende a proposta de Newman e adiciona a classe "*Meaning*", com objetivo de agregar mais valor as *tags* por meio de significados (KIM et al, 2008). Por meio desta classe é possível controlar os problemas relacionados a sinônimos e ambiguidades.

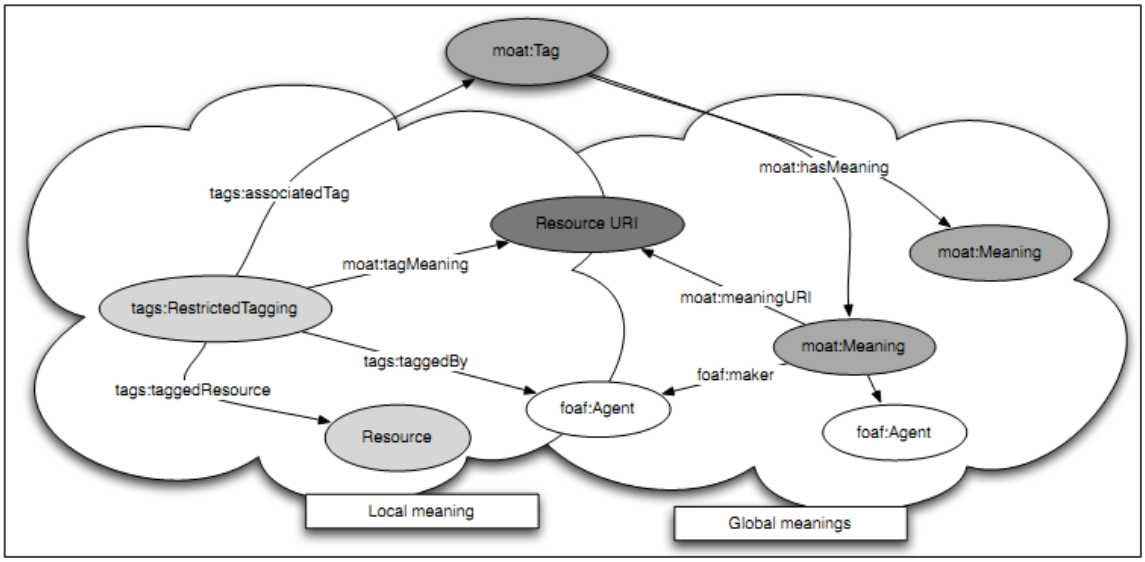

Figura 2-9 - Ontologia de Moat. Adaptado de (KIM, 2008)

A ontologia SCOT (Social Semantic Cloud of Tag), tendo como base a ontologia de Newman, tem sua estrutura baseada no conceito de uma *tag* representar um conjunto de outras *tags*. Para isso, implementa a classe *Tagcloud*, que liga um ou vários processos e ambientes de tags, auxiliando na comunicação e troca de *tags* entre sistemas heterogêneos (KIM et al, 2008), conforme ilustrado na Figura 2-10.

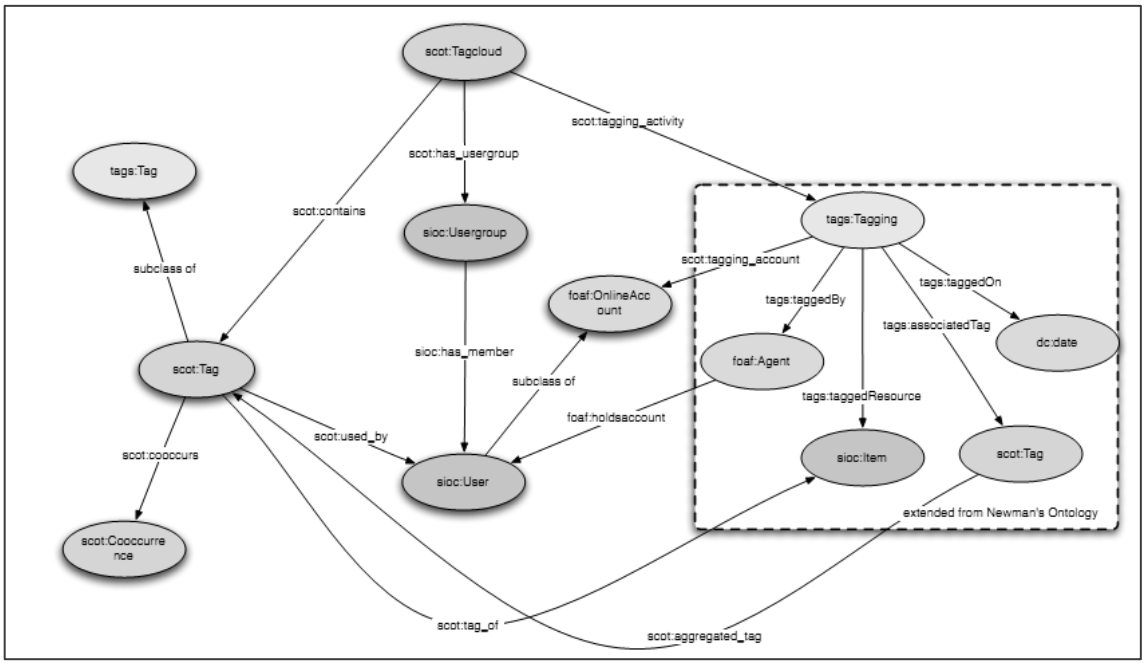

Figura 2-10 - Modelo de representação de folksonomia SCOT. Adaptado de (KIM et al, 2008)

Knerr (KNERR, 2006), implementou a classe *ServiceDomain* e *Source* para melhor representar uma folksonomia. *SerciveDomain* corresponde ao domínio, ou comunidade, onde o processo de tagueamento ocorreu (Figura 2-11). Desta forma, é possível converter dados de tagueamento de aplicações sem perder seu contexto original.

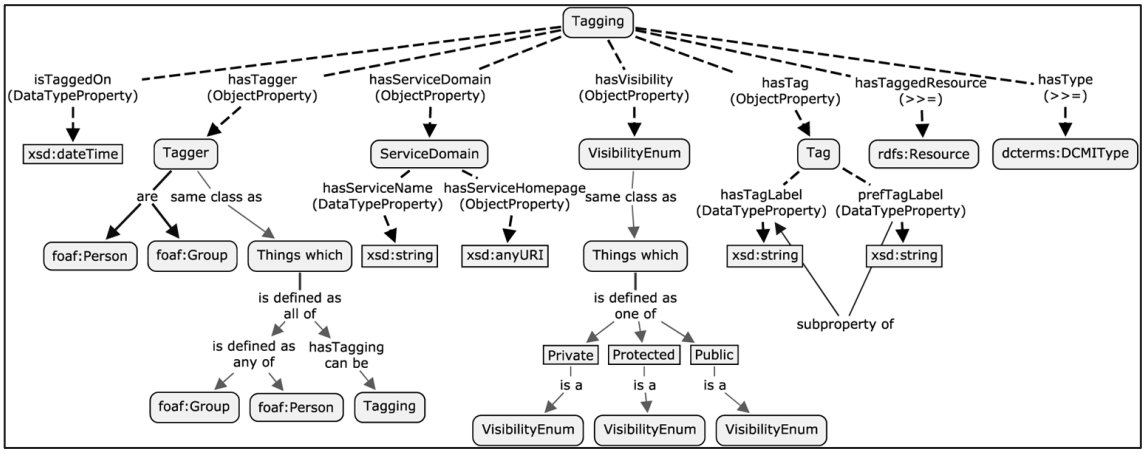

Figura 2-11 - Ontologia de Knerr. Adaptado de (KNERR, 2006)

Echarte propôs o modelo de folksonomia baseado nos princípios de Gruber e Newman. Seu grande diferencial está na implementação das classes *Annotation* e

AnnotationTag, que permite enriquecer o processo de tagueamento com mais informações sobre o *resource* (ECHARTE et al, 2007).

Kim (KIM et al, 2008) propõe que a folksonomia deve ser representada pela tripla "*Folksonomy: (tag set, user group, source, occurrence, Tagging+)",* onde *tag set* indica o conjunto de *tags* sendo empregado, *user group* o conjunto de usuários que participaram na atividade de tagueamento, *source* o local onde a folksonomia é utilizada, *ocurrence* a popularidade da mesma e o último parâmetro reflete o processo coletivo de tagueamento realizado pelos usuários da folksonomia, representado pela tripla *Tagging (U, T, R)*.

A ontologia com a Web Semântica, podem ser aplicadas com a folksonomia para obter melhores benefícios na aquisição, manipulação e disseminação do conhecimento. Além disso, têm por objetivo integrar dados heterogêneos e estabelecer a interoperabilidade entre sistemas diferentes.

Por meio da Web Semântica é possível publicar recursos de conhecimento reutilizáveis na engenharia de software, superando as desvantagens da linguagem natural em uma forma de representar o conteúdo de maneira formal e precisa e sem ambiguidades (ZHAO et al, 2009).

2.5 Web Semântica

A evolução dos dados fez emergir da rede mundial de computadores um novo conceito de compartilhamento de informações. Até então, apenas informações destinadas a seres humanos eram compartilhadas na rede. Qualquer aplicação destinada à Web, como os motores de busca, realizavam a conversão de palavras para outros tipos de dados destinados à interpretação exclusiva pelo computador, o que ocasionava uma fraca aderência entre termos e significados (THURAISINGHAM, 2002).

Defronte a uma nova necessidade, Tim Berners-Lee projetou um novo conceito de compartilhamento de informações, a Web Semântica, onde os dados são processados diretamente ou indiretamente por máquinas (BERNERS-LEE, 1999). O propósito da Web Semântica não é criar, ou recriar, uma infraestrutura Web que seja mais inteligente, mas idealizar uma infraestrutura que seja mais apropriada ao trabalho de integração da informação na Web (ALLEMANG; HENDLER, 2008).

A Web tradicional foi criada para compartilhar documentos e não dados contidos dentro destes documentos (POLLOCK, 2009). A nova internet propõe dar

significado às informações que antes eram mantidas apenas como dados isolados de aplicativos/sites proprietários, por motivos estratégicos e competitivos. Documentos com significado tornam a rede mais inteligente simplesmente pelo fato destes estarem organizados no lugar certo (ALLEMANG; HENDLER, 2008).

Na Web, qualquer pessoa expressa seus comentários e opiniões sobre qualquer tema em páginas HTML. O problema da Internet é que esta foi construída apenas para humanos, sendo que a máquina precisa usinar as informações, transformando-as em dados de máquina. Aplicações, como *Scraper*, transformam o HTML em representação RDF (*Resource Description Framework*), para ser passível de ter as informações integradas (POLLOCK, 2009).

Na Web Semântica cada indivíduo expressa suas opiniões a respeito de uma entidade que pode ser combinada com informações de outras fontes, gerando dados não só para o próprio ser humano, mas para a máquina, evitando que motores de busca ou qualquer outra aplicação Web tenham que “adivinhar” o significado dos dados.

A ideia principal da Web Semântica é suportar uma Web distribuída ao nível de dados, e não ao nível de apresentação, ou seja, ao invés de uma página Web apontar para uma outra página, um dado pode apontar para outro, por meio de referências globais, chamados de *Uniform Resource Identifiers* (URI) (ALLEMANG; HENDLER, 2008). Tais dados são publicados na Web e processados por meio de metadados.

Criado por Berners-Lee e sua equipe, os metadados realçam o conceito do dado propriamente dito, com informações adicionais, melhorando a sua interpretação e utilização pela máquina, originando, consequentemente, o *Resource Description Framework* (RDF).

2.5.1 Resource Description Framework

(DACONTA; OBRST; SMITH, 2003) definem *Resource Description Framework* (RDF) como uma linguagem baseada em XML para descrever recursos. (POLLOCK, 2009) aufere como um padrão de especificação de dados e modelagem utilizado para codificar metadados e informação digital. O consórcio W3C (*World Wide Web Consortium*) define RDF como uma linguagem para suportar a Web Semântica, da mesma forma que o HTML, considerado a linguagem que ajudou a iniciar a Web. RDF é um framework de apoio à descrição de recursos (*resource*

description), ou metadados (dados sobre dados), para a Web, oferecendo estruturas em comum que podem ser utilizadas para interoperar a troca de dados em XML (POWERS, 2003).

Como recurso, proveniente da sigla RDF, pode ser entendido como qualquer artefato ou entidade disponibilizado na Web, desde que possua uma identificação única, podendo ser um ponteiro, referência de objeto ou até mesmo um valor no formato *string* (POLLOCK, 2009). O identificador único corresponde ao URI, mecanismo para identificação de um objeto na Web. Por meio deste mecanismo, cada item é interligado à demais itens, formando um único conjunto de dados.

A descrição, da sigla RDF, caracteriza o tipo de relacionamento entre os recursos formando um modelo de dados gráfico (*graph data model*). O framework (RDF) é uma combinação de protocolos baseados na Web (URI, HTTP, XML, por exemplo) que definem, pelo modelo semântico, as relações permitidas entre os itens de dados no RDF.

Por meio do relacionamento entre as entidades descritas pelo RDF, é possível capturar dados de diversas fontes e unificá-las formando um único *datasource*. Sendo uma entidade, uma tabela composta por várias linhas, cada linha deve possuir três atributos chaves do RDF: sujeito, predicado e objeto. Os três elementos formam uma *triple* composta por um identificador único URI (ALLEMANG; HENDLER, 2008). Na frase “Lula governou o Brasil”, por exemplo, tem-se:

- **sujeito**: http://www.brasil.gov.br/presidente#Lula.
- **predicado**: http://www.brasil.gov.br/presidente#governou.
- **objeto**: Brasil.

Para o sujeito e predicado é aplicado um identificador único global. Por meio desta estratégia adotada pelo RDF, é possível descrever uma única entidade separadamente em diversos servidores, onde cada parte da informação é complementada por outro autor de outra fonte, resultando em uma única entidade.

O RDF foi projetado para trabalhar com informações de qualquer tipo de fonte, sendo convertido em conjuntos de triplas (planilhas, XML, tabelas de banco de dados e webpages), como um simples mecanismo, convertendo informações de variadas fontes em um formato único, combinando tudo em um *single store* (*data federation*) (ALLEMANG; HENDLER, 2008).

Informações de diversas fontes convertidas em triplas são chamadas de *data federation*. Dados heterogêneos e de múltiplas fontes são convertidas em um

formato único, agrupados em um *single store*. Os dados são unidos e “misturados” por meio de padrões de linguagens de consultas, como o SPARQL, buscando informações através de banco de dados RDF distribuídos na Web (POLLOCK, 2009), (ALLEMANG; HENDLER, 2008). SPARQL corresponde a padronização da *query language* pela W3C, especificamente para busca de informações em locais diferentes, formando uma única fonte de dados *dataset* (conjunto de dados).

A consulta dos dados mantidos pelo RDF é realizada a partir do SPARQL. Sendo o SPARQL semelhante a *queries* de banco de dados tradicionais, existe uma grande diferença de integração da informação semântica RDF *Store* com a integração tradicional. Os bancos de dados tradicionais não possuem padrões, o que dificulta o intercâmbio de dados entre diferentes bancos. Em contraste, a Web Semântica foi desenvolvida justamente para este fim, ou seja, inter-relação das informações é realizada de forma natural.

A linguagem de consulta facilita o desenvolvimento de portais Web com RDF, se assemelhando a banco de dados relacionais tradicionais, devido a arquitetura em camadas adotada. Uma consulta faz junção em várias tabelas, resultando em uma única tabela.

Um dos problemas do RDF é a sua carência na especificação de relacionamentos semânticos complexos, como a especificação de lógica para um relacionamento específico. Uma das alternativas para atender essa lacuna é a utilização da *Web Ontology Language* (OWL). OWL é uma extensão do modelo de dados RDF, oferecendo suporte ao desenvolvimento de modelo de dados complexos, com melhores vocabulários e lógica de software (POLLOCK, 2009).

Para completar a falta de recurso do RDF, assim como adicionar e agregar maior referência à linguagem de conhecimento, utilizam-se *schemas*, declarados da forma de *namespaces* no topo do arquivo RDF.

Assim sendo, é possível usufruir da semântica por meio do *Resource Description Framework Schema* (RDFS, ou, RDF-Schema), onde vários *schemas* definidos em conjunto podem proporcionar ontologias sob medida para bases de conhecimentos customizadas, oferecendo significado aos elementos das afirmações contidas no RDF, além de servir como metamodelo para a definição de conceitos de domínio específico.

2.5.2 Web Ontology Language

Originado pelo departamento de defesa norte americano e europeu, a OWL corresponde a junção da *Ontology Inference Layer* (OIL) e DARPA *Agent Markup Language* (DAML).

A OWL corresponde a uma linguagem de ontologia para a Web Semântica. Provê classes, propriedades, indivíduos e valores de dados armazenados como documentos da Web Semântica, sendo que as informações contidas na OWL podem ser usadas em conjunto com informações escritas em RDF (W3C, 2009).

Classe, proveniente da OWL, também conhecida como coleção, é um conjunto de membros que compartilham propriedades ou características semelhantes (POLLOCK, 2009). Classes são os blocos básicos que constituem domínio. Todas as classes na Web Semântica são subclasses da classe *Thing*. Indivíduos (*individual*), também chamados de instâncias, correspondem aos membros das classes, representando conceitos físicos ou virtuais da ontologia. Os indivíduos são associados, ou relacionados, com outros indivíduos por meio de propriedades do objeto (*Object properties*) (LACY, 2005).

Propriedade, conceito originado pelo RDF, tem como objetivo relacionar as instâncias do sujeito a qualquer tipo de dado ou outro objeto de instância. Propriedades são utilizadas para fazer declarações de todos os membros ou de uma determinada classe ou instância específica em uma classe (LACY, 2005).

Na OWL, a forma como as informações são tratadas e processadas se diferencia de outras bases de informação. Em sistemas tradicionais, uma informação tida como não verdadeira, é considerada falsa. Isto implica no *closed-world assumption* (CWA). Comparando a linguagens de programação ou banco de dados tradicionais, um campo lógico, por exemplo, é considerado como falso se o seu valor não for atribuído. Ao contrário desta analogia, a OWL assume a analogia *open-world assumption*, (OWA) onde uma informação é considerada falsa somente quando esta pode ser provada que é falsa. Desta forma, a OWL se torna um sistema monotônico, ou seja, de lógica dedutiva, onde uma nova informação adicionada à uma base de conhecimento nunca falsificará (negará) uma conclusão anterior (POLLOCK, 2009).

A principal vantagem da OWL é a maneira como a modelagem de dados é tratada. Uma simples modelagem de dados pode apresentar múltiplos sentidos, o que a torna dinâmica. Quando uma pergunta é feita sobre o que poderia e o que não poderia ser verdadeiro sobre certos indivíduos pertencerem a uma mesma classe, e

não se tem todas as informações no momento, a OWL se torna uma ferramenta poderosa de raciocínio, pois ao contrário de uma modelagem de dados para banco de dados tradicionais, as afirmações não são compreendidas com mais de um sentido, ou seja, mesmo não tendo a resposta, a ontologia tenta “adivinhar” a melhor resposta.

É importante que as afirmações sejam especificadas com clareza, caso contrário a base se tornará inconsistente. As principais afirmações, em nível de classe, são: *class equivalence, property equivalence, individual equivalence, class disjointness, individual disjointness* e *subsumption* (POLLOCK, 2009). As propriedades também possuem características relacionadas às afirmações. No entanto, estas são consideradas mais complexas, pois além de exigirem mais planejamento, devem estar em conformidade com a principal característica da OWL: ser monotônico. As principais características das propriedades da OWL são:

- **funcional**: indica que uma instância não possui mais de um tipo de relacionamento com outra instância.
- **inversa**: indica que um determinado relacionamento é bidirecional, ou seja, o sujeito e o objeto são invertidos quando se relacionam de maneira oposta ao relacionamento definido.
- **simétrica**: indica que o relacionamento entre sujeito e objeto são iguais também na direção oposta.
- **transitiva**: indica se uma instância faz parte de um determinado relacionamento. Se um objeto A for igual a B, e B for igual a C, então A é igual a C.
- **interseção**: indica que os membros de uma instância correspondem aos mesmos membros de duas ou mais outras instâncias.
- **união**: indica que todos os membros de duas ou mais classes somados fazem parte de uma nova classe especificada.
- **complemento**: indica que todos os membros que pertencem a uma classe não pertencem a nenhuma outra classe.
- **classe** restrição: corresponde as restrições, ou seja, valores que podem ser associados a uma propriedade. Existem inúmeros tipos de restrições, destacando *owl:allValuesFrom*, *owl:someValuesFrom* e *owl:hasValue*. Cada restrição descreve como uma nova classe deve

ser limitada com possíveis propriedades de afirmação (ALLEMANG; HENDLER, 2008).

- **domínio** e **escopo**: domínio infere a adesão de um sujeito em uma classe e, escopo infere a adesão de um objeto.

Sendo a OWL uma linguagem para representação de ontologias, e a ontologia uma forma de representação do conhecimento, é imprescindível que este conhecimento possa ser criado, armazenado e compartilhado, por meio da gestão do conhecimento.

2.6 Gestão do Conhecimento

Em um novo projeto de software, o gerente de projetos pode não possuir conhecimentos suficientes sobre a nova empreitada, como regras de negócio, fatores e indicadores relacionados ao cliente. Em contrapartida, o arquiteto de software precisa ter ciência dos requisitos, principalmente dos não funcionais, para estruturar a arquitetura de acordo com as novas necessidades; e os desenvolvedores precisam conhecer ferramentas e técnicas de desenvolvimento antes de iniciar seus trabalhos.

Engenheiros de software são colaboradores do conhecimento, visto que as tarefas desempenhadas nesta área requerem conhecimento e experiência. Cada experiência está relacionada a um fato ou experiência de projeto passado.

Cada participante deve colaborar com seus conhecimentos, para que juntos, o projeto como um todo se torne inteligente, e não apenas um agregado de dados. No entanto, para que um empreendimento e, consequentemente, uma organização se torne mais inteligente, as aprendizagens arquivadas devem ser de fácil acesso para reúso, além de resultados anteriores serem validados, organizados (em estrutura ou por links) e, em alguns casos, se tornarem recomendações para novos projetos (SCHNEIDER, 2009) .

Gestão do conhecimento não deve somente “disponibilizar” o conhecimento, mas também disseminá-lo. É importante validar o conhecimento antes de reutilizá-lo, comparando o conhecimento de diversas fontes.

O conhecimento está relacionado com a criatividade. Um novo conhecimento é gerado quando uma nova perspectiva ou um novo relacionamento é estabelecido,

conduzindo a uma nova conclusão que não se tinha anteriormente (SCHNEIDER, 2009) .

No entanto, para que as pessoas possam usufruir deste conhecimento, é preciso, transformá-los de tácito para explícito. Para (NONAKA; TAKEUCHI, 1995), o conhecimento tácito e explícito interagem entre si, por meio da socialização, externalização, combinação e internalização.

Externalização é o conhecimento tácito quando convertido em conhecimento explícito. O oposto, quando o conhecimento explícito é convertido para o implícito, ocorre o que se chama de internalização e, consequentemente, em conhecimento tácito. Entende-se por socialização quando ocorre a transferência de conhecimento de uma pessoa para outra, sem intermédios. Quando um conhecimento explícito for convertido em outro conhecimento explícito, ocorre uma combinação (NONAKA; TAKEUCHI, 1995).

O modelo clássico de raciocínio define três padrões de abstração e aplicação do conhecimento: indução, dedução e abdução. Pela indução chega-se a conclusão partindo de um caso específico para o mais genérico. Ao contrário, conclui-se, por meio da dedução, o princípio do geral ao específico. Por fim, abdução consiste na invenção de novos princípios genéricos por meio da decorrência de hipóteses de um caso especial (SHANK, 1998) .

Em âmbito geral, a maioria das atividades de diferentes áreas demandam conhecimento, raciocínio e distribuição. Sendo que cada atividade possui suas particularidades.

As atividades da engenharia de software estão totalmente relacionadas à gestão do conhecimento, desde as atividades iniciais de entrevistas com o usuário até a fase pós-entrega, que correspondem às manutenções e melhorias do produto final.

A Engenharia de Software está associada não apenas à semântica de codificação, mas sim, à documentação e gestão envolvida para chegar ao produto final. Toda documentação, assim como arquitetura e algoritmos aplicados ao projeto, dependem de experiências e conhecimentos passados. A Engenharia de Requisitos, por exemplo, demanda conhecimento e compreensão do domínio da aplicação, assim como habilidades de se comunicar com pessoas experientes, além de ter habilidades de mapear os requisitos em possíveis soluções (SCHNEIDER, 2009), (ZHAO et al, 2009).

Engenheiros de software tomam decisões o tempo todo, sendo que na maioria das vezes essas decisões são tomadas em cenários incertos e inseguros, devido à falta de informação. O acesso a fontes de conhecimento não só auxilia, mas melhora a confiança em respeito à decisão tomada.

O fato de uma organização de software que aprende (*learning software organization*) demandar muito conhecimento em sua forma de trabalho, em específico no desenvolvimento de software, além da utilização mais madura da tecnologia da informação e melhores usos de ferramentas disponíveis (e-mail, fórum, ferramentas específicas), faz com que esta se diferencia de outras organizações que aprendem.

Mentzas (MENTZAS et al., 2003) compara e classifica as ferramentas disponíveis em um organização em duas famílias, do ponto de vista da gestão do conhecimento: gestão do conhecimento como um produto, e gestão do conhecimento como um processo.

A abordagem do conhecimento como um produto se compara a um supermercado. A logística do conhecimento, embalar, armazenar, e classificar um pedaço desse conhecimento, da mesma forma que um produto que se expõe na prateleira, é uma das preocupações. Enquanto que administradores do supermercado se concentram na organização das propostas e na qualidade dos produtos, os trabalhadores do conhecimento se responsabilizam em selecionar os produtos, conforme o apoio da estrutura da administração. Quando um “pedaço” do conhecimento chega de “algum lugar”, estes são verificados, reembalados, etiquetados e colocados nas prateleiras.

A abordagem do conhecimento como um processo aborda a questão do gerenciamento do conhecimento no aprendizado individual e coletivo. O conhecimento está na pessoa, enfatizando os grupos sociais, a psicologia e o cognitivo envolvido, destacando, desta forma, a importância da experiência tácita que as pessoas carregam como fruto de experiências e convivência passadas.

No campo da Ciência da Computação, a inteligência artificial tem seguido os passos da abordagem do conhecimento como um produto, tentando resolver o problema de como oferecer, buscar e catalogar o conhecimento conforme a demanda dos usuários. Informações em metadados, ontologias, e a Web Semântica estão sendo utilizados para automatizar, classificar e minerar tais informações.

O aprendizado constante sem dúvida é a chave para se manter atualizado e pronto para atender novas demandas e necessidades do mercado. Contudo, o conhecimento precisa estar bem documentado e "configurado" na base de conhecimento da organização. Os benefícios são percebidos durante os processos da Engenharia de Software. Um deles é a formalização do conhecimento tácito (SCHNEIDER, 2009).

Quando um membro da equipe deixar a organização, ou se submeter a outras responsabilidades, seu conhecimento e experiência ficam arquivados na empresa, com o propósito de reuso. Sabe-se que tudo o que é implícito e tácito é facilmente esquecido, devendo assim, ser disseminado à equipe e, consequentemente, à organização.

2.7 Processo de Compreensão aplicado à manutenção de software

Vários trabalhos foram desenvolvidos relacionados à manutenção e compreensão de software, sendo que nem todos estão focados em atender o mesmo objetivo, porém, utilizam técnicas semelhantes, atuando no código fonte.

O Quadro 2-2 apresenta o resumo dos principais trabalhos que utilizaram técnicas que exploraram conceitos que estavam relacionados, além do objetivo deste trabalho, ao mesmo objeto fim de estudo: o código fonte e a tentativa de aperfeiçoar a manutenção e compreensão.

Quadro 2-2 Principais trabalhos relacionados à compreensão e manutenção de software

Artigo	Descrição
An Empirical Validation of Complexity Profile Graph (YANG et al., 2005)	Estudou a complexidade de entender um programa no momento de manutenção, para efeitos de cálculos e métricas de estimativas de esforço.
That is Not My Program Investigating the Relation between Program Comprehension and Program Authorship (DASGUPTA, 2010)	Por meio de pesquisas, identificou os níveis de compreensão: sintático e semântico. Conclusão dos maiores fatores decisivos na compreensão são: acesso a comunidades experientes, aspectos culturais, própria eficácia, domínio, organização do conhecimento e a existência de modelos mentais.
The Role of Software Measures and Metrics in Studies of Program Comprehension (MATHIAS et al, 1999)	O artigo apresentou métricas de compreensão da complexidade do software. Programadores tendem a mudar suas estratégias na medida em que o LOC aumenta. Quando a medição resulta em maiores complexidades, os programadores têm maiores dificuldades de entender o código.

Identifying Domain Expertise of Developers from Source Code (SINDHGATTA, 2008)	Por meio de catalogação do código fonte descobre o conhecimento dos programadores sobre o domínio da aplicação. Aplica métricas de "aproximação" do código com o conhecimento.
Empowering Software Maintainers with Semantic Web Technologies (WITTE et al., 2007)	Explorou uma forma de manter "conectado" por meio de conexões semânticas artefatos da engenharia de software partindo do código fonte, por meio de ontologias. Como resultado da mineração de texto no código fonte, populou uma ontologia a partir da criação de instâncias, possibilitando a realização de consultas para busca de dados, assim como descobrir *design patterns* no código fonte.
Ontology Classification for Semantic-Web-Based Software Engineering (ZHAO et al, 2009)	Demonstrou a aplicação da Web Semântica juntamente com diversas ontologias aplicadas ao suporte das atividades de processos, especificação de requisitos, arquitetura, padrões de software (*design patterns*), implementação, qualidade e manutenção da engenharia de software.
Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology (ZHOU et al, 2008)	Propõe a união da ontologia de conhecimento do código com a de conhecimento do domínio.
A formalism of ontology to support a software maintenance knowledge-based system (APRIL et al, 2008)	Com base no Software Maintenance Maturity Model e no KBS (knowledge-based system) formalizou-se uma ontologia para alcançar as melhores práticas na manutenção.
A Unified Ontology-Based Process Model for Software Maintenance and Comprehension (RILLING et al, 2006)	Apresenta um modelo de compreensão (Comprehension Process Meta-Model). Técnica de gerenciamento de processos que estabelece comunicação e interação entre usuários, processos e gestores de ontologia, auxiliando o usuário durante as diferentes fases do processo de manutenção.
Software Maintenance Maturity Model The software maintenance process model (APRIL et al, 2005)	Propôs um modelo que complementa o CMMi (*Capability Maturity Model Integration*) e as normas ISO (International Organization for Standardization) para manutenção.
A Context-Driven Software Comprehension Process Model (MENG et al., 2006)	Aplica ontologia e DL (*Description Logic*) no modelo dirigido a histórias para apoiar processo de compreensão desenvolvido pelos autores.
Shared Waypoints and Social Tagging to Support Collaboration in Software	Realiza o tagueamento do código por meio de anotações, com o objetivo de guiar os desenvolvedores por meio de *waypoints*. Esse trabalho não usa ontologias nem princípios semânticos.

Development (STOREY et al, 2006)	Tem-se por objetivo auxiliar no processo de desenvolvimento a partir da navegação, coordenação e captura do conhecimento.
An ontology-based program comprehension model (ZHANG, 2007)	Desenvolveu uma ontologia de código fonte e documentação para auxiliar no processo de compreensão, por meio de buscas complexas inferidas na ontologia populada a partir da mineração de texto aplicada no código fonte.

A compreensão do código é uma atividade ligada às questões psicológicas, que estudam como as pessoas montam modelos mentais para trabalhar com grande volume de dados para a escrita e entendimento do código fonte. Existem alguns modelos mentais e formas de organizar os pensamentos para extração, compreensão e escrita estritamente relacionados ao código fonte que foram estudados por vários pesquisadores. No entanto, o trabalho "*Industrial experience with an integrated code comprehension model*", de (MAYRHAUSER; VANS, 1995), foi o alicerce de vários estudos seguintes, por apresentar conceitos e estudos psicológicos relevantes ao tema, utilizado no desenvolvimento e implementação de novos modelos para auxiliar no processo de compreensão de código fonte.

A utilização de ontologias foi muito explorada na atividade de manutenção de software por grande parte dos trabalhos aqui destacados. Uma das propostas foi a sua utilização para complementar a modelagem de software, usando-a como mais um "artefato" de modelagem para representação do conhecimento. Outras formas de utilização são destacadas, como: uso para descoberta de falhas de implementação dos requisitos, segurança (brechas), automatização de testes contínuos por meio de sistemas multi-agentes "guiados" por ontologia, construção de casos de testes automaticamente a partir da captura do domínio do código, procura de erros/falhas, descoberta de *design patterns,* extração do conhecimento do domínio da aplicação e rastreabilidade dos artefatos de engenharia de software e código.

Dentre as técnicas de aplicação da ontologia com o código fonte, este trabalho propõe uma nova abordagem: utilizar a ontologia como consequência do conhecimento extraído do código fonte por meio do modelo de sensemaking.

O modelo sensemaking é muito explorado no campo da medicina, principalmente para a descoberta de doenças a partir de sintomas observados, onde os profissionais são constantemente confrontados com o desconhecido em suas experiências cotidianas. No entanto, seu emprego ainda não é muito explorado na área de Engenharia de Software, em especial na compreensão de códigos fonte.

Como diferencial, este trabalho abordou e aplicou o modelo sensemaking para apoiar e desenvolver novas descobertas e tratar do desconhecido no ato da compreensão do código fonte por parte dos programadores.

Com base no sensemaking foi desenvolvido e implementado um método, tendo como princípio formalizar e implementar uma folksonomia dentro do código fonte, para que seja possível extrair o conhecimento e mantê-lo em uma base de conhecimento, com o objetivo de extrair e disseminar tanto o conhecimento de domínio como de funcionalidades contidas no código fonte.

2.8 Considerações sobre o capítulo

Este capítulo apresentou os principais modelos empregados no processo de compreensão, e manipulação do código fonte, tanto para aquisição do conhecimento dentro do código fonte como para sua disseminação.

A Web Semântica pode ser utilizada para distribuir o conhecimento, a folksonomia para criar e desenvolver o conhecimento e o sensemaking como modelo para o processo da folksonomia, para consequentemente, modelar o conhecimento adquirido com o apoio de ontologias.

Pelo fato de a memória do ser humano ser limitada ao analisar, interpretar e entender o código fonte, a busca do conhecimento previamente adquirido a partir da análise do código fonte pode ajudar os programadores a melhorar a performance de compreensão do código fonte.

CAPÍTULO 3 - ESTRUTURAÇÃO DA PESQUISA

O objetivo deste capítulo é descrever alguns conceitos importantes referentes à metodologia de pesquisa, identificar as características metodológicas deste trabalho e detalhar as estratégias de pesquisa que foram adotadas para atingir o objetivo geral do trabalho.

São várias as definições de pesquisa. Para (ANDRADE, 2009), pesquisa é o conjunto de conhecimento sistemático, com base no raciocínio lógico, com objetivo de encontrar soluções para problemas propostos, por meio da utilização de métodos científicos.

(GIL, 2002) define pesquisa como procedimento racional e sistemático que tem como objetivo proporcionar respostas a problemas propostos. (CERVO, BERVIAN, 1996) completam que a solução de problemas ocorre através do emprego de processos científicos. Por fim, (SALOMON, 1993) complementa que a pesquisa é uma investigação e o tratamento por escrito de questões abordadas metodologicamente.

A metodologia da pesquisa deste trabalho segue as orientações da Pesquisa de Desenvolvimento proposta por (VAN DER MAREN, 1996). Segundo (VAN DER MAREN, 1996) este tipo de pesquisa pode tomar três formas: desenvolvimento de conceito, desenvolvimento de objeto e desenvolvimento ou aperfeiçoamento de habilidade pessoal.

Devido ao estudo desta pesquisa propor um método e envolver o desenvolvimento de um ambiente de apoio à manutenção e compreensão do código fonte, esta pesquisa se enquadra no desenvolvimento de um objeto, que visa solucionar problemas formulados a partir da prática cotidiana, por meio de teorias elaboradas a partir da investigação nomotética (VAN DER MAREN, 1996).

A pesquisa de desenvolvimento envolve quatro etapas principais, conforme ilustra a Figura 3-1, as quais serão detalhadas e discutidas a seguir, no âmbito desta pesquisa.

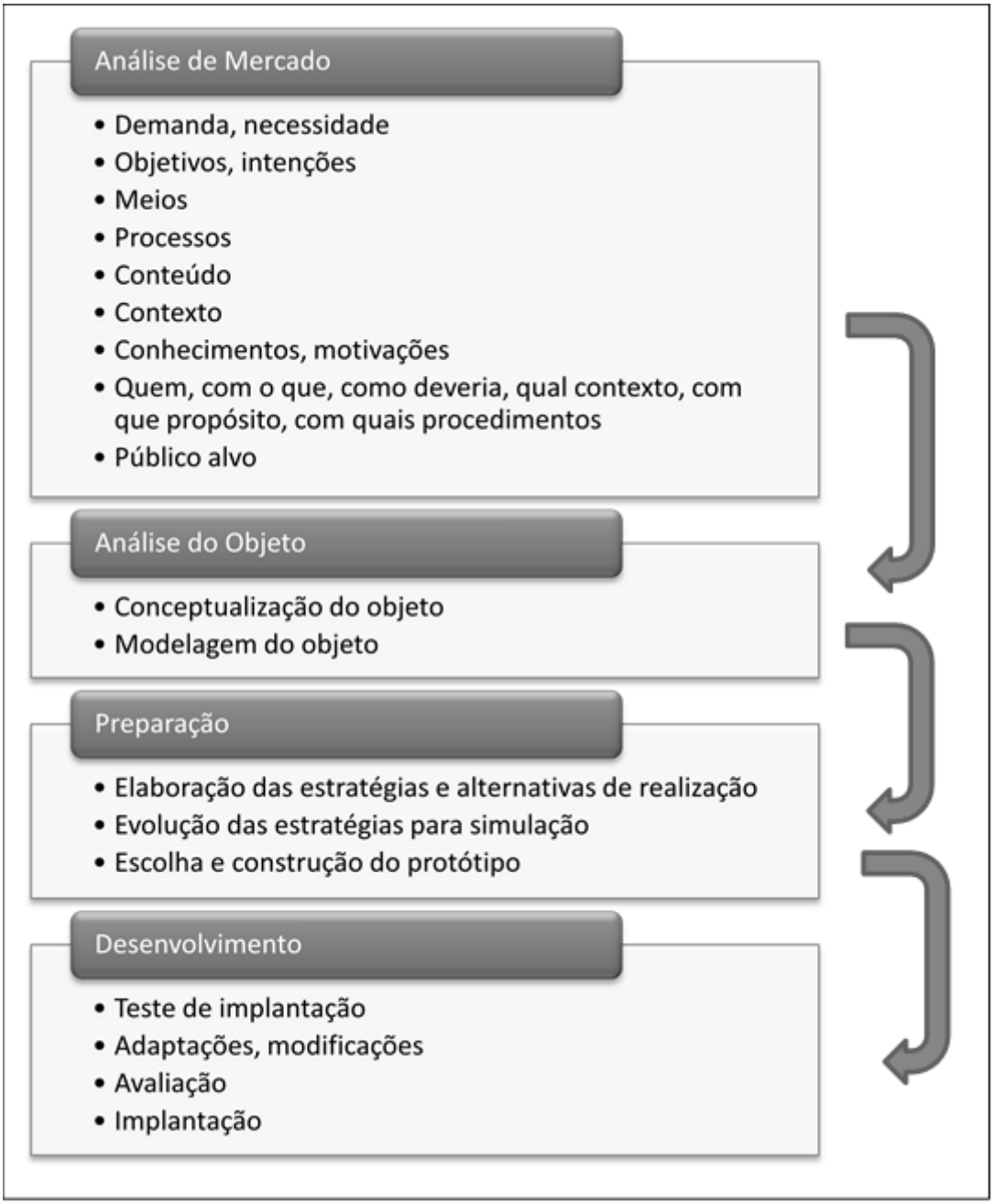

Figura 3-1 - Etapas da pesquisa desenvolvimento de objeto. Adaptado de (VAN DER MAREN, 1996)

3.1.1 Etapa 1 – Análise de Mercado

Nesta fase foi realizada uma pesquisa bibliográfica com o objetivo de encontrar as abordagens já propostas para apoiar a tarefa de manutenção e compreensão de código fonte, assim como foi elaborada uma pesquisa de campo com profissionais que atuam no desenvolvimento de software (programadores), com o objetivo de avaliar os problemas existentes no processo de manutenção e compreensão de códigos fonte.

A pesquisa bibliográfica foi realizada nas bases científicas ACM Digital Library, ScienceDirect, SpringerLink, IEEE e Google Scholar, considerando publicações em conferências e journals, sem limitação do ano de publicação. Dos trabalhos encontrados e relacionados ao tema, todos tinham suas pesquisas embasadas em estudos psicológicos e cognitivos realizados entre meados de 1980 e 1995. As principais propostas relacionadas a compreensão do código fonte, assim como o resultado desta pesquisa, estão inclusos no capítulo 2.

Para a pesquisa de campo com programadores, foi elaborado um instrumento de pesquisa, contendo questões sobre compreensão e manutenção de código fonte; métodos utilizados pelos programadores para realizar manutenção e compreensão em códigos desconhecidos; ferramentas utilizadas, técnicas e fatos que auxiliam, ou não, na manutenção. Foi realizada uma pesquisa piloto para avaliação do instrumento, com o objetivo de marcar o tempo de resposta e avaliar se as questões estavam claras e objetivas. Após o piloto, foram realizados os ajustes necessários no instrumento, o qual foi disponibilizado no Google Docs[2], sendo a solicitação da participação dos profissionais encaminhada por e-mail.

No total 119 programadores participaram da pesquisa de campo, onde 54% dos respondentes correspondiam a programadores seniores, pois tinham mais de 5 anos de experiência com programação, e 46% correspondiam a juniores, apresentando tempo menor de experiência. Dos programadores seniores, 41% tinham mais de 5 anos de experiência em orientação a objetos, enquanto que dos programadores juniores, 17% apresentaram ter menos de 1 ano de experiência em orientação a objetos.

3.1.2 Etapa 2 – Análise do Objeto

Com base na análise de mercado é possível planejar os procedimentos para elaboração do método e do ambiente para apoiar a sua avaliação.

3.1.2.1 Planejar os procedimentos para concepção do método

Tem-se como objetivo desenvolver um método que atenda as características do processo de compreensão, catalogação e busca do conhecimento na manutenção de software. Deve-se permitir adicionar informações ao código fonte,

[2] O formulário está descrito no APÊNDICE A

que nesta proposta será por meio de *tags*, entre trechos de códigos importantes que contêm dados relacionados às regras de negócio ou execução do software.

Para atender os estágios dos processos de sensemaking, que correspondem aos 8 estágios identificados por Weick (WEICK et al., 2005), e contemplar as características do conhecimento sintático para semântico, conforme (SOMMERVILLE, 2003), o método deverá incluir alguns requisitos, mapeados no Quadro 3-1, de acordo com cada estágio do sensemaking.

Quadro 3-1 - Requisitos do método conforme os estágios do sensemaking

Estágio	Requisito
Organização do fluxo	Permitir criar e relacionar *tags*.
Observação e suporte	Permitir disponibilizar dados sobre a *tag*.
Rotulagem	Permitir identificar as *tags* como privadas e públicas.
Retrospecção	Permitir visualizar e consultar as *tags* existentes e suas respectivas características.
Presunção	Permitir relacionar a *tag* ao item de domínio, que corresponde ao código fonte.
Social e sistêmico	Permitir compartilhar as *tags* e analisar suas utilizações, por meio de pesos e frequências associados.
Ação	Permitir associar semanticamente a *tag* com o objeto de domínio, possibilitando buscas bidirecionais.
Organização através da comunicação	Possibilitar inferir e compartilhar informações a partir de técnicas da Web Semântica.

As tags que serão criadas a partir da folksonomia, que correspondem aos resultados do processo de sensemaking, serão os alicerces do processo de aquisição de conhecimento, realizado de forma manual pelo programador responsável pela manutenção.

Para que seja possível suportar os processos do sensemaking, a partir da criação, manipulação e associação das tags ao código fonte, faz-se necessário o uso de ontologia para folksonomia e ontologia de código fonte orientado a objetos. Sendo assim, faz-se necessário analisar a possibilidade de reúso ou a necessidade de criação das ontologias.

3.1.2.2 Avaliar ontologia para folksonomia

Como a folksonomia é uma das estratégias definidas a ser utilizada no método, então é necessário defini-la e implementá-la. No entanto, para que uma folksonomia possa realizar processamentos mais complexos por meio de representações semânticas, além de possibilitar gerenciar o vocabulário controlado gerado e compartilhado pelos programadores de um mesmo projeto, a ontologia se torna a ferramenta complementar à folksonomia. Além do mais, a representação do conhecimento extraído, a partir da folksonomia, é melhor representada com base em uma ontologia.

Foi realizada uma pesquisa em bases científicas, repositórios de ontologias e sites de universidades, com o objetivo de encontrar ontologias desenvolvidas para folksonomia e que tivessem sido implementadas e disponíveis para o reúso. A pesquisa em bases científicas eletrônicas foi realizada na ACM Digital Library, ScienceDirect, SpringerLink e IEEE. Foram utilizadas as palavras chaves *folksonomy* e *ontology*. O período de busca compreendeu até fevereiro de 2012, sendo consideradas publicações de periódicos e *journal*. Em relação as repositórios pesquisados, foram considerados: Protégé Ontology Library, Swoogle, Daml, Ontolp, SchemaWeb e OntoSelect. Com base nos resultados obtidos, chegou-se a alguns projetos de ontologias hospedados em algumas universidades, pois a implementação das ontologias encontradas com base nos artigos e repositórios encontrados faziam partes de projetos e estudos de estudantes.

A pesquisa nas bases científicas retornou 1.202 artigos. Por meio do título, foram classificados aqueles mais importantes e relevantes. Tendo uma pré classificação realizada, foram analisados os resumos destes artigos, sendo eliminados aqueles que não tratavam a ontologia como apoio à folksonomia. Os artigos restantes, 32, foram lidos e então foram descartados aqueles que não apresentavam claramente uma ontologia já implementada. Foram analisados dois artigos chaves (KIM et al, 2008) e (KIM et al, 2008), que apresentavam o resultado de um estudo de mercado contendo as principais ontologias existentes e implementadas. Com base nestes artigos, chegou-se a lista das principais ontologias existentes e que apresentavam ontologia desenvolvida para folksonomia, sendo cada uma das ontologias analisadas seguindo os seguintes critérios estabelecidos para seleção da mais apropriada para esta pesquisa. A ontologia deve:

- preferencialmente estar implementada como ontologia *heavyweight* e disponível para reuso;
- auxiliar na construção e manutenção da folksonomia;
- resolver questões de ambiguidades, sinônimos, acrônimos, sinonímia e polissemia: estas são as principais deficiências da folksonomia;
- possibilitar separar as *tags* pessoais das *tags* de utilização comum;
- taguear e descrever recursos;
- possibilitar associar recursos por inter-relacionamentos das tags; e
- realizar semântica eficaz para as necessidades de inferência de conhecimento.

Como foi encontrada a ontologia de Knerr (KNERR, 2006) que atende as necessidades desta pesquisa, não foi necessário desenvolver uma nova ontologia.

3.1.2.3 Avaliar ontologia para código fonte

Como o propósito desta ontologia é apoiar a ontologia da folksonomia, seu critério de seleção não foi tão criterioso quanto ao adotado na análise da ontologia de folksonomia.

Desta forma, foi realizada uma pesquisa apenas em motores de busca e repositórios de ontologia. O objetivo foi avaliar as ontologias existentes e que poderiam ser utilizadas para armazenar instâncias de conceitos de código fonte orientado a objetos. Os critérios utilizados na seleção destas ontologias baseadas nas classes e definições existentes, foram:

- possuir classes que definem pacotes, classes, métodos e atributos;
- possuir propriedades relevantes às definições de domínio, como nome, tipo e modificadores;
- especificação coesa dos relacionamentos entre as classes ontológicas de definições de domínio. Relacionamento de métodos com classes, atributos com classes, pacotes com classes e parâmetros de entrada e saída de métodos.

Dentre as ontologias avaliadas, SWONTO, SOCON, SEC, SOURCECODE e SCRO, a última foi selecionada para armazenar as instâncias relacionadas ao código fonte. SCRO é uma ontologia OWL-DL, criada para suportar as principais tarefas de compreensão de software através da representação explícita do conhecimento conceitual encontrado no código fonte (ALNUSAIR, 2010).

3.1.2.4 Refinar ontologia de folksonomia e alinhar com ontologia do código fonte

Para ser possível realizar inferências melhores sobre o conhecimento adquirido e buscas semânticas no conteúdo do projeto, as ontologias de folksonomia e código fonte foram integradas (*merge*). Com isso foi possível adquirir o conhecimento inerente às pessoas e ao código fonte de forma organizada e ordenada. Na medida que o conhecimento do código fonte for catalogado via tags pelos usuários, a ontologia da folksonomia será populada, tendo como resultado o processo de aquisição do conhecimento.

Além da integração entre as ontologias, foi adaptada a ontologia de código fonte selecionada, para atender alguns conceitos levantados anteriormente, como a criação da classe *Chunk*[3] e a propriedade *hasChunkBlock.* Na ontologia de folksonomia foi criada a propriedade *frequency* associada a classe Tag, com objetivo de guardar o número de vezes que uma determinada instância de uma Tag foi utilizada.

3.1.3 Etapa 3 – Preparação

A partir da análise do problema e da projeção do método proposto, a etapa de preparação consiste no desenvolvimento do método e implementação do ambiente para aplicação do método proposto.

3.1.3.1 Concepção do método

Conforme os estudos desenvolvidos na literatura referente ao desenvolvimento e manutenção de software, as atividades cognitivas envolvidas e os processos do sensemaking, foi proposto um método para apoiar a atividade de compreensão e, consequentemente, a manutenção de códigos fonte.

O método foi concebido a partir dos estágios do sensemaking, expandindo gradativamente e consequentemente, para a disciplina de manutenção da engenharia de software. A partir destes estágios, foram elaboradas etapas e atividades que contemplam o método, com objetivo de guiar e auxiliar o programador durante a compreensão do código fonte.

Com base nas etapas do método um ambiente foi especificado e implementado, para que fosse possível aplicar o método e avaliá-lo.

---

[3] Porções do código fonte que programadores reconhecem.

3.1.3.2 Implementação do plugin de apoio

Para o desenvolvimento e implementação do ambiente, os seguintes requisitos foram definidos:

- ser totalmente integrado a IDE de desenvolvimento Eclipse;
- possibilitar popular a ontologia de código fonte a partir da seleção do arquivo fonte Java;
- identificar o usuário;
- permitir adicionar *tags* no código fonte;
- permitir reutilizar as *tags*;
- permitir consultar as *tags* criadas, a frequência de utilização e o autor criador;
- permitir tornar uma *tag* pública para os demais programadores, possibilitando-os a reutilizá-las;
- permitir consultar, a partir do código fonte, as *tags* associadas aos métodos, classes e atributos;
- permitir exportar as ontologias populadas para serem analisadas e processadas por outras ferramentas.

Foram utilizadas as seguintes ferramentas e tecnologias para implementação: Eclipse SDK 3.6, MySQL, Protégé e Java 6.

3.1.4 Etapa 4 – Desenvolvimento

A etapa final consiste na concretização desta pesquisa, aplicando-a e discutindo os resultados obtidos.

3.1.4.1 Realizar experimentos

O objetivo desta etapa foi avaliar o método proposto. Foi proposto para uma equipe de desenvolvedores realizar duas melhorias em um sistema existente e desconhecido pelos integrantes. O sistema consiste em um projeto de automação de força de vendas, desenvolvida em linguagem Java para dispositivos móveis. Sua versão inicial foi projetada para ser executada em dispositivos PAML OS, Windows Mobile e Android. O sistema utiliza o banco de dados *Litebase* e possui uma arquitetura simples de 3 camadas: *model, view* e *control*.

O projeto original foi alterado, onde foram removidas as funcionalidades propostas pelos experimentos que serão executados. Também foram realizadas

pequenas modificações no fonte, como por exemplo, foi criado na classe “FrmCadastroPedido” do projeto, o método “validarDesconto”, com o objetivo de distrair a atenção do programador. Apesar de ter um nome sugestivo, este método não realiza a validação do desconto, como indicado, além de este método não ser chamado pela aplicação.

A equipe que participou do experimento era composta por 4 pessoas, sendo 2 profissionais juniores e 2 profissionais seniores, classificados da seguinte forma:

- **júnior**: programador que possui menos de 5 anos de experiência em programação em orientação a objetos, estruturação, arquiteturas de software, *design patterns*, organização e boas práticas de codificação. Para a execução dos experimentos, serão denominados como JA e JB.
- **sênior**: programador que possui mais de 5 anos de experiência em desenvolvimento de sistemas, já trabalhou em projetos grandes e complexos e que possui, mas não necessariamente, conhecimento sobre o domínio do software que está em manutenção. Para a execução dos experimentos, serão denominados como SA e SB.

Foram realizados três experimentos no total, conforme descritos no Quadro 3-2, Quadro 3-3 e Quadro 3-4. Cada experimento possui um objetivo, procedimento e proposta em específico.

Os experimentos foram realizados dentro do ambiente de trabalho de uma empresa fábrica de software de médio porte, por profissionais que detinham experiências em mais de uma linguagem de programação, além de serem formados na área, executavam a tarefa de desenvolvimento há mais de um ano. O Quadro 3-5 apresenta uma breve descrição dos envolvidos nos experimentos.

Quadro 3-2 - Descrição do experimento 1

	Programadores	**Procedimento**	**Descrição**
Experimento 1	Junior A. Sênior B.	Passar a mesma atividade para uma pessoa sem experiência e outra com experiência. A atividade consiste em realizar uma melhoria em um sistema existente. No contexto geral, será solicitado que seja adicionado uma	Ao informar o desconto validar se o desconto informado é maior que o permitido. Se for maior informar o usuário que o desconto não pode ser maior
	Objetivo		
	Avaliar a dificuldade de entender e compreender um código fonte de outra		

	autoria **Avaliação** Tempo e o local da melhoria.	regra de negócio, onde espera-se que esta alteração seja realizada na classe correta, pois o projeto está estruturado em camadas, e cada classe tem sua responsabilidade, além da implementação correta da validação. Neste experimento não será disponibilizado os recursos de utilização de *tags*, apenas os recursos oferecidos pela própria IDE.	que o valor cadastrado na base de dados e configurar automaticamente o valor do desconto com o valor máximo permitido. O valor desconto máximo está armazenado na variável "descMax", na classe de tela do pedido.

Quadro 3-3 - Descrição do experimento 2

	Programadores	**Procedimento**	**Descrição**
Experimento 2	Junior B. Sênior B. **Objetivo** Avaliar a compreensão de um código fonte de outra autoria realizando tagueamento. **Avaliação** Tempo, o local da melhoria e o nome e número de novas tags criadas durante o processo.	Passar a mesma atividade para ambos, semelhante ao experimento 1. O código fonte não estará tageado, mas será permitido que os desenvolvedores adicionem, compartilhem e utilizem as *tags* para auxiliar o processo de manutenção.	Não permitir remover pedidos que já foram enviados ao servidor. Na tentativa de excluir um pedido já enviado a operação deverá ser cancelada, e o usuário deverá ser informado.

Quadro 3-4 - Descrição do experimeto 3

	Programadores	**Procedimento**	**Descrição**
Experimento 3	Junior A. Junior B. Senior B. **Objetivo** Avaliar a compreensão de um projeto já tagueado por alguém que conheça o projeto.	Repetir o experimento 1, mas desta vez com o projeto já tagueado. A atividade consiste em corrigir um erro existente no código, onde algumas informações estão sendo mostradas erroneamente na tela. Os envolvidos nos testes deverão utilizar as *tags* como guias para chegar ao ponto crítico do sistema, realizando a	Idem ao experimento 1.

	Avaliação	manutenção no local correto. Para esta atividade, um programador sênior irá realizar a compreensão e disponibilizará o projeto já tagueado para os envolvidos na manutenção.	
	Tempo e a qualidade da manutenção, comparar a melhoria desenvolvida entre os programadores júniores e seniores, avaliar o número de novas tags criadas.		

Para cada programador foram repassados individualmente conceitos básicos de folksonomia e de tagueamento no código fonte. Foi instalado em cada computador o Eclipse com o *plugin* desenvolvido e as configurações necessárias para acessar a rede e o banco de dados, que estava localizado em um computador específico da empresa.

Os experimentos foram executados considerando duas situações:

- **base de conhecimento vazia**: neste contexto, contemplados pelos experimentos 1 e 2, não existiam *tags* associadas ao código fonte. As *tags*, que correspondem a base de conhecimento, foram desenvolvidas pelos programadores na medida que estes realizaram o processo do sensemaking no código fonte, associando-as ao código e disponibilizando para as demais pessoas do grupo;
- **base de conhecimento previamente populada**: as *tags* da base de conhecimento já estavam criadas e disponíveis aos programadores que iriam realizar a manutenção, que no caso corresponde ao experimento 3. O sensemaking teve início com base nas *tags* já identificadas.

Quadro 3-5 - Profissionais que participaram dos experimentos

Profissional	Descrição
JA	Possui pouca experiência em programação. Nunca trabalhou com Java, mas conhece linguagens semelhantes, como C#.NET e C. Atua como desenvolvedor ASP.NET e PHP.
JB	Possui experiência moderada em programação. Trabalha atualmente com programação Java no desenvolvimento de um produto específico da empresa.
SA	Possui experiência em desenvolvimento Java, além de outras linguagens de orientação a objetos. Já trabalhou em projetos de grande porte. Hoje atua como programador Java, Flex e realiza atividades de gerenciamento de projetos com a equipe.
SB	Possui larga experiência em desenvolvimento, modelagem e processos da engenharia de software. Domina Java e C++. Atua como desenvolvedor Java e arquiteto.

Foi estipulado que cada experimento seria executado em um tempo limite de 30 minutos, prazo estipulado para não atrapalhar o resto da equipe, muito menos a desconcentração das atividades dos envolvidos, pois o objetivo era aplicar os experimentos durante o trabalho, e não em um ambiente caracterizado como laboratório de testes.

A condução dos experimentos ocorreu de forma individual. Para cada profissional o autor da pesquisa realizou o repasse do projeto, assim como a melhoria desejada que fosse executada no ambiente. Ao decorrer dos experimentos foram levantadas observações e questões relacionadas a navegação, dificuldades e até sugestões dadas pelos profissionais.

Após a execução de cada experimento, de forma individual, os resultados foram relacionados e analisados, conforme descritos no Quadro 3-6.

O objetivo da comparação dos resultados dos experimentos é poder comparar o desempenho do programador júnior, utilizando os conceitos da folksonomia no código fonte por meio do ambiente implementado, com um programador sênior, sem utilizar o recurso proposto por este trabalho.

Quadro 3-6 - Comparativo dos experimentos

Relacionamento	Avaliação - Objetivo
Experimento 1 x Experimento 2	• Verificar o desempenho da manutenção sem o uso de tags (experimento 1) e com o uso de tags (experimento 2); • Avaliar o desempenho entre manutenções realizadas entre programadores juniores, entre programadores seniores, e entre programadores juniores e seniores.
Experimento 1 x Experimento 3	• Analisar o desempenho da manutenção realizada por um programador sênior sem o uso de tag e por um programador júnior com o uso de tag.
Experimento 2 x Experimento 3	• Avaliar os impactos causados na manutenção da melhoria quando não existem tags, ou seja, a compreensão é iniciada sem auxílio de conceitos de domínios previamentes criados (experimento 2) • Avaliar os impactos causados na manutenção da melhoria quando as tags forem previamente (experimento 3) e estão disponibilizadas para auxiliar no processo de compreensão.

3.1.4.2 Avaliar o método proposto

O objetivo da avaliação é responder a questão inicial desta pesquisa: "É possível diminuir o tempo de esforço da compreensão do código fonte e com isso aumentar a qualidade e a eficiência da manutenção de software?".

Com base na execução dos experimentos foi analisado o comportamento dos programadores avaliados, através da observação[4] das atitudes, das dúvidas questionadas pelos programadores durante as atividades e a desconfiança e incerteza ao lidarem com o código fonte desconhecido.

Além do comportamento, o tempo também foi considerado, pois é a partir dele que será possível concluir e medir a eficiência do método.

Por fim, a qualidade da manutenção executada foi avaliada se a partir do método proposto as melhorias solicitadas foram executadas no local correto do código fonte, evitando, desta forma, implementações em locais errôneos, que poderia acarretar em um comportamento inadequado ou até falha de execução do sistema.

[4] Foi acompanhado a execução dos experimentos lado a lado com cada programador.

3.2 Considerações sobre o capítulo

Este capítulo apresentou a caracterização da pesquisa, a estratégia e as etapas. Para cada etapa detalhou-se os respectivos procedimentos e premissas para execução. As etapas macro definidas, conforme o modelo de Pesquisa e Desenvolvimento adotado, foram: análise de mercado, análise do objeto, preparação e desenvolvimento.

CAPÍTULO 4 - MÉTODO TAGGINGSENSE

Com base nos estágios do sensemaking, foi proposto um método para dar suporte as etapas e aos processos intrínsecos envolvidos na compreensão de um código fonte durante a manutenção.

O método é denominado “TaggingSense”, pois reúne, em sua essência, os conceitos de tagueamentos da folksonomia, e os estágios e processos identificados pelo sensemaking.

O método foi desenvolvido com o objetivo de orientar as atividades das pessoas durante a compreensão de software, por meio de etapas que supostamente deveriam ser seguidas para padronizar e aperfeiçoar a atividade de compreensão e manutenção de software, além de possibilitar, por meios formais, arquivar o conhecimento extraído, reutilizá-lo, aperfeiçoá-lo e compartilhá-lo, com objetivo de acelerar e aperfeiçoar o processo de compreensão de códigos fonte desconhecidos.

A seguir será apresentado o método proposto, demonstrando suas atividades e etapas. O capítulo 6 apresentará um ambiente desenvolvido também no âmbito desta pesquisa para apoiar o método proposto.

4.1 Desenvolvimento do método

Com base nos estágios e estudos do sensemaking realizados, foi elaborado, como ponto de partida para o desenvolvimento do método, o Quadro 4-1. Este quadro apresenta, de forma resumida, os principais pontos dos estágios do sensemaking, conforme idealizado por (WEICK et al, 2005), para o desenvolvimento do método TaggingSense.

Quadro 4-1 - Estágios do Sensemaking

Organização	• Observação do caos • Organização e estruturação do caos
Observação	• Assimilação • Validação • Transformação do conhecimento

Rotulagem	• Processo autônomo • Experiências e fatos passados • Categorização • Atribuições
Retrospectivo	• Análise do passado • Percepção de padrão
Presunção	• Interpretação do fenômeno • Identificação do fenômeno • Afirmação de hipóteses
Social e Sistêmico	• Comunicação • Catalogação do acontecimento • Classificação do acontecimento • Conclusões preliminares
Ação	• Avaliação do fato atual • Avaliação do fato futuro
Comunicação	• Comunicação • Transformação do conhecimento tácito em explícito • Análise das circunstâncias

Com base nos pontos abordados um método macro foi criado para início do estudo do desenvolvimento do sensemaking durante a percepção dos códigos desconhecidos, conforme Figura 4-1.

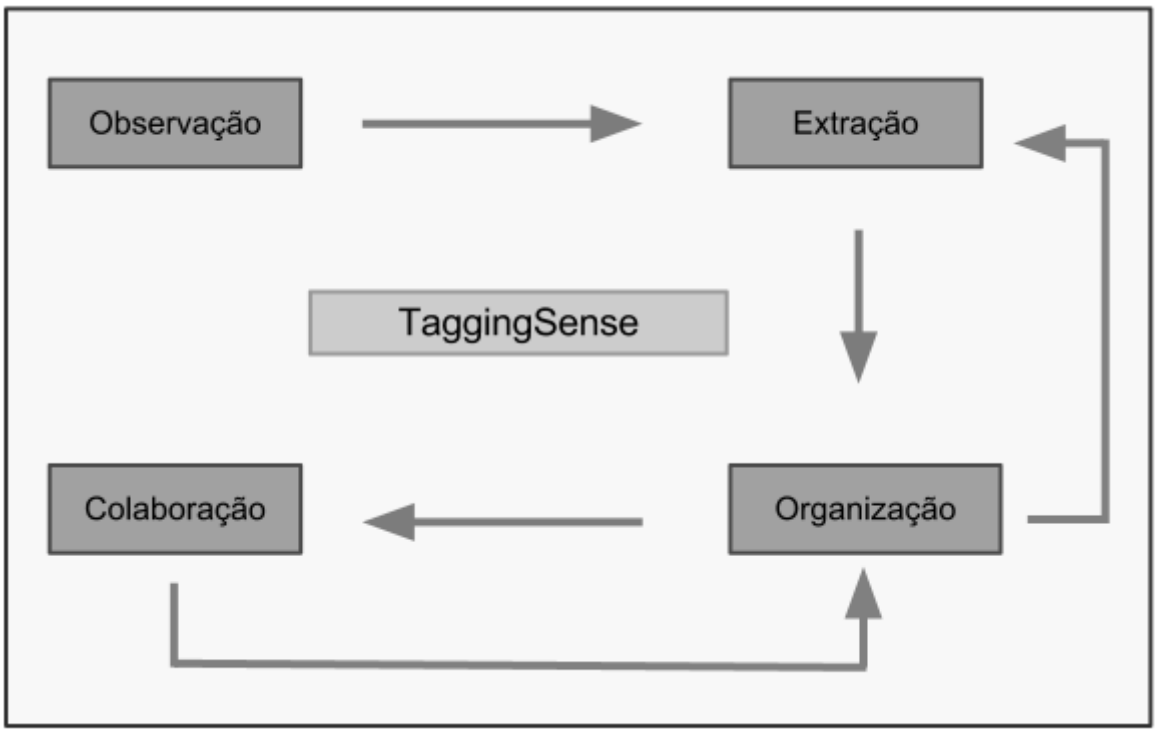

Figura 4-1 - Método TaggingSense: Visão macro

Observação: esta atividade consiste na análise superficial que o programador realiza ao iniciar a atividade de manutenção. Como ponto de partida, ocorre uma

avaliação superficial da forma como está estruturado o código fonte, as classes e demais arquivos relacionados ao projeto. Também é considerada como análise inicial da arquitetura, padrões existentes e demais conceitos técnicos relevantes. Como consequência das observações, o programador, nesta atividade, formula ideias e as estrutura, tendo como base experiências de projetos passados.

Extração: atividade relacionada a extração e desenvolvimento do conhecimento contido no código fonte. É nesta atividade que inicia-se o sensemaking. O conhecimento é formalizado e arquivado, pelo programador, junto ao código fonte.

Organização: organização e estruturação do conhecimento extraído. Esta atividade consiste em dar suporte e apoiar ou rejeitar ideias e hipóteses levantadas, de forma a aperfeiçoar a estruturação do conhecimento. É nesta atividade que o programador identifica fenômenos e padrões observados, aperfeiçoa a externalização e a catalogação do conhecimento adquirido.

Colaboração: o principal componente desta atividade é a comunicação. Nesta atividade ocorre o compartilhamento e o desenvolvimento do conhecimento com o grupo de pessoas envolvidas, por meio da troca de experiências e o refinamento do aprendizado.

4.1.1 Estrutura do método proposto

Com base no método preliminar proposto (Figura 4-1), além dos estágios do sensemaking (Quadro 4-1), desenvolveu-se o detalhamento das etapas do método, apresentados na Figura 4-2, nos Quadro 4-2, Quadro 4-3, Quadro 4-4 e Quadro 4-5.

Figura 4-2 - Etapas do Método TaggingSense

Quadro 4-2 - Atividade 'Observação' do método TaggingSense

Nome da Atividade:	**Observação**
Tarefas:	**Descrição:**
Análise da estrutura	Estudo preliminar da estruturação do código fonte. Avaliação do número de classes, arquivos, pacotes e bibliotecas do projeto.
Busca de conhecimento técnico e domínio	Aperfeiçoamento do conhecimento técnico em relação à estruturação do código fonte, como linguagem de programação, paradigmas, arquitetura e padrões. Breve busca em documentação, fóruns, equipe de trabalho e sites de Internet questões relevantes ao domínio da aplicação. Estudo preliminar de conceitos de domínio e regras de negócio mais relevantes.
Saída	Formulação e estruturação de ideias e hipóteses.

Quadro 4-3 - Atividade 'Extração' do método TaggingSense

Nome da Atividade:	**Extração**
Tarefas:	**Descrição:**
Extração do conhecimento	Desenvolvimento dos conceitos de domínio. Assimilação entre questões de domínio e questões técnicas relacionadas ao código fonte. Exemplo: analisar e identificar no código fonte classes e métodos que implementam uma lógica de negócio específica.
Tagging	Marcação do código fonte por meio de tags. Utilização da folksonomia para apoiar, suportar e organizar as tags

	criadas durante a extração do conhecimento. Corresponde ao início da formalização da extração do conhecimento, onde as ideias são postas em palavras e assimiladas com trechos de códigos relevantes, ou nome de variáveis, métodos ou classes. Exemplo: destacar métodos, classes ou trechos de código e atribuir uma palavra chave que ajude a identificar o real significado destes pontos no código fonte. Estas palavras chaves serão identificadas como *tags*.
Externalização	Ocorre a articulação do conhecimento, ou seja, transformação do conhecimento tácito em conhecimento explícito ou usável. Nesta tarefa o programador já possui o conhecimento sintático e semântico desenvolvido, a partir da execução das etapas anteriores. É nesta etapa que ocorre a externalização das ideias para dentro da estrutura do ambiente. Esta tarefa representa a continuidade da tarefa de *Tagging*.
Guias	Aprimoramento da marcação do código fonte. A marcação é estruturada como meio de auxiliar o programador a se localizar no código fonte, por meio de *waypoints*. A navegabilidade é aprimorada, com objetivo de auxiliar em novas descobertas no código fonte. Exemplo: a partir das palavras chaves identificadas, o programador realiza buscas de trechos de códigos que são relevantes a palavra chave, além de se guiar por trechos de códigos já previamente analisados e estudados.
Saída	Conhecimento formalizado.

Quadro 4-4 - Atividade 'Organização' do método TaggingSense

Nome da Atividade:	**Organização**
Tarefas:	**Descrição:**
Enriquecimento tags	Desenvolvimento de novos conceitos relacionados aos conceitos já desenvolvidos e identificados, por meio das *tags*. Associações entre *tags* e entre novos conceitos de domínios com as *tags* já existentes.
Refinamento do conhecimento	Refinamento dos pontos relacionados ao domínio da aplicação. Ocorre a reavaliação e a continuação da tarefa “Extração do conhecimento” da atividade anterior.

	Com base nas *tags* já criadas novas *tags* são criadas, com o objetivo de apoiar e aperfeiçoar o relacionamento semântico entre as *tags* (conhecimento) Esta tarefa pode ser executada por outro usuário, quando as *tags* já estiverem sido liberadas/publicadas para o grupo com a finalidade de reuso.
Reavaliação das tags	Validação da importância com o projeto e com o domínio. As *tags* redundantes são eliminadas, e as comuns são reutilizadas no projeto.
Catalogação / padronização	Ocorre a padronização entre os termos já criados. Termos semelhantes ou que são sinônimos são reajustados. As tags desenvolvidas são constantemente organizadas e catalogadas conforme o conceito extraído do código fonte. Tarefa ocorre durante toda a atividade de organização.
Liberação	Toda nova *tag* criada tem como propriedade sua visibilidade configurada como privada, pois é neste estágio que a mesma está sendo desenvolvida, podendo sofrer alterações e até mesmo ser eliminada pelo usuário criador. Durante a fase inicial do levantamento do conhecimento, as *tags* não estão preparadas para serem reutilizadas, devido ao seu estágio precoce para com o ambiente. Após sua liberação, ou seja, alterada sua visibilidade para pública, demais pessoas do grupo poderão utilizá-las e reutilizá-las para novas assimilações de conceito de domínio com o código fonte do mesmo projeto em questão. Uma vez tornada pública, uma tag não pode se tornar privada novamente, pois isso implicaria negativamente nas tags derivadas e já desenvolvidas.
Saída	Conhecimento reformulado e organizado.

Quadro 4-5 - Atividade 'Colaboração' do método TaggingSense

Nome da Atividade:	**Colaboração**
Tarefas:	**Descrição:**
Armazenamento	Durante todo o processo, o conhecimento extraído é armazenado em um banco de dados denominado base de conhecimento, por meio de ontologias. A ontologia populada corresponde à ontologia de folksonomia com as *tags* criadas e relacionadas entre si e entre os conceitos de domínio identificados no código fonte.

	Exemplo: armazenar todas as tags criadas com base em trechos do código no banco de dados MySQL.
Compartilhamento	O banco de dados deve ser compartilhado com todos os envolvidos no processo de manutenção do projeto em específico. Este compartilhamento consiste nos indivíduos criados na ontologia da folksonomia e que devem estar disponíveis no momento da manutenção, para pessoas que não irão desenvolver o conhecimento, mas apenas usá-lo como ferramenta de apoio, assim como para aqueles que irão continuar o processo de compreensão.
Refinamento	Aprimoramento e melhorias nas *tags* criadas por outros programadores. A partir do momento que as *tags* são compartilhadas com o grupo, novos conhecimentos são desenvolvidos e extraídos, com base nas *tags* existentes. Este processo ocorre de forma cíclica, e em constante transformação durante toda a atividade de manutenção.
Reutilização	Reutilização de *tags* criadas por outros programadores. Ao reutilizar uma nova *tag* sua frequência de utilização aumenta e, consequentemente, sua importância e relevância no contexto do conhecimento. Exemplo: utilizar uma *tag* criada por uma outra pessoa para identificar outro trecho de código localizado no mesmo projeto.
Saída	Banco de dados de conhecimento.

4.1.2 Considerações do método TaggingSense

Com base nos estágios do sensemaking propostos por (WEICK et al, 2005), apresentada na revisão da literatura deste documento, um método para auxiliar a manutenção de códigos fonte foi proposto, com objetivo de guiar a execução desta tarefa, assim como para a implementação para uma futura ferramenta, que neste trabalho será desenvolvido um ambiente para apoiar este método, além de avalia-lo.

Foram criadas 4 atividades, contendo no total 15 tarefas, que se inter-relacionam. Cada atividade tem um propósito como entradas, gerando uma saída específica. As saídas geradas pelas atividades são:

- **formulação e estruturação de ideias e hipóteses**: desenvolvimento das ideias durante o estudo preliminar do código fonte pelo programador. As ideias estão formuladas e estruturadas tacitamente, onde sua externalização ocorrerá na execução da próxima atividade;
- **conhecimento formalizado**: transformação do conhecimento tácito em explícito, por meio da execução das tarefas desta atividade;
- **conhecimento reformulado e organizado**: a atividade anterior tem como objetivo formalizar o conhecimento. No entanto, o conhecimento formalizado não está estruturado e organizado. Esta atividade organiza o conhecimento de forma estruturada, além de enriquecer o conhecimento existente com mais informações; e
- **base de conhecimento**: corresponde ao local onde todo o conhecimento extraído do código fonte por um ou vários programadores está armazenado.

O método TaggingSense propõe a utilização de uma ontologia com base na folksonomia para a organização e manipulação das *tags*. Para contemplar o método, foram avaliadas as principais ontologias desenvolvidas para a folksonomia, selecionando a mais adequada para compor e método proposto.

4.2 Características da folksonomia para o método TaggingSense

É imprescindível que haja um reaproveitamento de *tags*. Desta forma, os problemas de ambiguidade são amenizados e a estrutura da folksonomia estará mais organizada. Após o usuário validar as entradas e associar o objeto a tag, o mesmo deve validar o relacionamento entre as *tags* e conceitos, que correspondem a instâncias de classes da ontologia. Este processo não é automático, porém, ao se deparar com a necessidade de criar uma nova *tag*, não se faz necessário contatar o administrador para criar uma nova instância na ontologia, pois o ambiente de folksonomia será controlado e restrito.

Em um ambiente aberto, onde centenas de milhares de usuários criam e associam *tags*, pode-se tornar um problema, devido ao consenso exigido na manipulação de ontologias de domínio. Porém, quando o grupo de usuários é limitado e suas atividades são controláveis, como, por exemplo, em um ambiente

corporativo, ou um departamento de desenvolvimento de software de uma determinada empresa, é possível evitar a variabilidade de *tags* e manter os conceitos e instâncias de ontologias criadas de forma organizada, evitando o papel do administrador (PASSANT, 2007). No meio colaborativo é possível criar normas e regras para tagueamento, o que não é possível em um ambiente totalmente aberto. Desta forma, o método não-supervisionado é a melhor opção para o método que está sendo proposto.

Grande parte das propostas e resultados apresentados da geração de ontologias a partir da folksonomia não levam em consideração a forma de como os seres humanos pensam, descartando o processo cognitivo envolvido na geração do conhecimento (CHEN, 2010). Processos disparados por agentes que buscam o conhecimento de forma automática no fonte, por meio de técnicas de mineração de textos, não consideram o fator cognitivo humano da catalogação do código fonte.

A utilização de *tags* por meio da folksonomia se enquadra melhor ao fator de demonstrar o pensamento humano, comparado com aqueles relacionados a extração automática de textos (AL-KHALIFA, DAVIS, 2007). Pelo processo manual, o usuário desenvolverá o sensemaking do código fonte e manualmente irá identificar um tópico/conhecimento por meio do tagueamento. Neste processo, a folksonomia será o resultado do processo do sensemaking idealizado pelo usuário.

Um dos pontos fortes da folksonomia é a livre atribuição de palavras a recursos. Anotar um recurso com várias palavras chaves exige menos esforço cognitivo do que escolher uma única categoria (SINHA, 2006). Para que essa livre atribuição se torne vantajosa, é interessante que seja possível identificar as *tags* destinadas a uso pessoal para com aquelas de uso coletivo e para outros fins. Conforme (SEN et al, 2006), a folksonomia pode ser classificada em 3 grupos:

- **tags pessoais**: relacionam a intenção do criador, sendo utilizado muitas vezes para referências próprias ou para organizações pessoais, como organização de tarefas e gerenciamento, por exemplo;
- **tags subjetivas**: expressa a opinião das pessoas a respeito do recurso; e
- **tags fatuais**: identificam fatos a respeito do recurso.

É fundamental que a implementação da folksonomia atenda os requisitos dos grupos anteriores de tal forma que seja possível separar e identificar as *tags*

pessoais com as *tags* de comum utilização, pois pela identificação pessoal será possível aplicar os processos de sensemaking, enquanto que as globais (comum) auxiliam no processo de população, aquisição e disseminação do conhecimento.

4.2.1 Ontologia para folksonomia

Há duas formas básicas de integrar a folksonomia e ontologia. A primeira corresponde a geração e população da ontologia com base no resultado da folksonomia, guardando todos os conhecimentos auferidos nos processos intrínseco da folksonomia. A segunda forma tem por objetivo usar a ontologia como base de apoio aos processos da folksonomia. É pela segunda forma que se resolve os principais problemas da folksonomia, como sinônimos, ambiguidades e buscas. O método proposto tem como relevância harmonizar a folksonomia e ontologia pela segunda forma.

Embora sejam abordagens diferentes, ambas se complementam devido a suas características. Para (MIKA, 2005) folksonomias são ontologias. Apesar de ser uma afirmação unilateral, Mika argumenta que a folksonomia não é tão dinâmica. Conhecimento altamente formal requer um compromisso que é difícil de obter em grande escala. Enquanto que as ontologias que são amplamente compartilhadas (*lightweight web ontologies*, como RSS e FOAF) serem basicamente superficiais devido ao seu escopo de compartilhamento, sistemas de grande escala como a Web é dinâmico, sendo que para alcançar um compromisso ontológico significante requereria estabilidade. A folksonomia se posiciona em vocabulários informais e não controlados comparado com o praticado pela comunidade da Web Semântica, o que a torna inutilizada para aplicação de raciocínio automatizado usando abordagens lógicas. No entanto, a folksonomia é repleta de semântica, sendo que a extração não é baseada na lógica, mas sim, emprestada da análise de rede. *Lightweight*, dinâmica e limitada no escopo de compartilhamento, folksonomias correspondem a um tipo diferente de ontologias, comparado com aquelas que apresentam mais formalidade nos vocabulários, estáveis, controladas e defendidas pela comunidade da Web Semântica.

Considerando os modelos já propostos por Gruber, Newman e estendidos por Kim, a ontologia a ser selecionada para atender o método proposto por este trabalho deve atender a alguns critérios, para que seja possível sua implementação durante o curso deste trabalho. Os critérios levantados sugerem que:

(i) **a ontologia exista e esteja implementada**: pretende-se adotar uma ontologia existente devidamente construída para ser utilizada no desenvolvimento deste trabalho. Ontologias que existem a nível conceitual e não estão implementadas, ou estão parcialmente implementadas, não serão consideradas;

(ii) **auxilie na construção e manutenção da folksonomia**: a ontologia deverá dar suporte as principais características da folksonomia, como *source*, *user group*, *tag*, *tagging* e frequência/ocorrência;

(iii) **resolver questões de ambiguidades, sinônimos, acrônimos, sinonímia e polissemia**: estas são as principais deficiências da folksonomia. A ontologia deve prever estas questões, oferecendo alternativas para a ocorrência destes casos;

(iv) **possível de separar as tags pessoais com as tags de comum utilização**: apesar de o uso da folksonomia ser coletivo, é importante saber e separar as *tags* pessoais com aquelas que se originaram no grupo. Manter a autoria e a classificação ajuda a desenvolver o processo de sensemaking;

(v) **esteja implementada como *heavyweight* ontology**: espera-se que a ontologia seja *heavyweight e* representada em OWL-DL, sendo passível de ser raciocinada de forma automática;

(vi) **taguear e descrever recursos**: este é o requisito essencial. A ontologia deve apresentar conceitos básicos de *tags* e recursos, como item e objeto e suas correlações.

(vii) **possível de ligar recursos por inter-relacionamentos das tags**: as *tags* especificadas pela ontologia devem prever a rastreabilidade entre recursos associados pelas *tags*. Deve ser possível taguear e descrever um recurso a partir de uma ou mais *tags*. Não considerar as *tags* como textos livres somente, mas como objetos. Assim será possível adicionar propriedades as *tags* e aos relacionamentos;

(viii) **realização semântica eficaz para as necessidades de inferência de conhecimento**: possibilitar identificar elementos, buscar e compartilhar por meio de recursos da Web Semântica. Descrever os recursos por meio da URI. Espera-se que seja possível aplicar buscas de conceitos

e conhecimentos por meio do SPARQL. Para que seja possível, a ontologia deverá ser implementada utilizando o formato RDF.

Antes de avaliar e selecionar as ontologias existentes, faz-se necessário analisá-las também sob o ponto de vista do sensemaking, conforme os estágios caracterizados por (WEICK et al, 2005) e descritos no capítulo 2.1 deste trabalho. O Quadro 4-6 aborda os estágios do sensemaking e as observações relacionadas.

Quadro 4-6 Estágios do sensemaking comparados com a implementação da ontologia e o método proposto

Estágio do sensemaking	Na ontologia
Organização do fluxo	Durante o caos, o usuário se confronta com o desconhecido. Pelo fato de observar o novo, surgem várias ideias. Cada ideia corresponde a uma *tag*, onde várias *tags* descrevem um objeto. Neste estágio, uma *tag* pode se inter-relacionar semanticamente por meio de metadatas e *annotations*.
Observação e suporte	A partir do resultado da organização, ocorre o processo de conclusão e validação das *tags*, com objetivo de transformá-las em conhecimento. Os relacionamentos são refinados e melhorados.
Rotulagem	Neste estágio o processo do sensemaking já está evoluído, assim como as tags e seus relacionamentos. As *tags* (*taggings*) privadas se tornam públicas e visíveis aos demais envolvidos, permitindo que as pessoas envolvidas no processo reconheçam e reajam as *tags* identificadas.
Retrospecção	Olhar as *tags* já criadas e descobrir um padrão. Para isso, a *tag* deve possuir data e hora de criação e atributos como ocorrência, frequência e/ou polaridade, pois de acordo com o resultado da análise, as *tags* receberão pesos, a fim de identificar aquelas que são mais importantes e plausíveis.
Presunção	Neste ponto a pessoa parte para o estado de afirmação, por meio de hipóteses. Neste estágio, a *tag* selecionada, juntamente com seus objetos, são relacionados ao conhecimento do domínio do problema.
Social e sistêmico	Compartilhamento das *tags* chaves identificadas, que correspondem aquelas que possuem os maiores pesos, com os demais, por meio de um banco de dados de conhecimento acessível de forma coletiva.
Ação	Retomada do código fonte, adaptando trechos menos importantes com as *tags* chaves identificadas. A identificação deste trechos ajudará a responder a pergunta “o que está acontecendo aqui?”, e apoiará a decisão do *sensemaker* na continuidade do processo, respondendo a segunda pergunta “o que devo fazer depois?”.
Organização através da comunicação	Compartilhamento e organização das *tags*. Este passo é a base para a implementação dos anteriores. A junção e compartilhamento poderá ser realizado por meio de conceitos de agrupamento (*TagCloud*, *ServiceDomain*, *Site*, etc) , e disponibilizados por meio da infraestrutura da Web Semântica.

4.2.2 Ontologias avaliadas

Foram avaliadas as principais ontologias desenvolvidas para apoiar o processo de tagemanto. Dentre elas, estão:

Gruber: Apesar de não possuir implementação, muitos autores utilizaram seu modelo como base. Gruber descreveu os quatro elementos do processo de tagueamento: *object*, *tag*, *tagger* e *source*. O elemento *source* corresponde a um escopo de *namespaces* ou universo de quantificação para objetos, permitindo diferenciar-se entre dados de marcação (*tagging*) a partir de diferentes (KIM et al, 2008).

Newman: Descreve relacionamento entre um agente, recurso arbitrário e uma ou mais tags. Utiliza como base três conceitos essenciais: *Tagger*, *Tagging* e *Tag (User, Resource e Tag)*, que correspondem as atividades de tagueamento. Implementa as classes *RestrictedTagging* (uma tag por recurso) e *Tagging* (uma ou mais tags por recurso). Não representa o fonte (*souce*) da ação de tagueamento. As tags são apresentadas como instâncias de *tags:Tag*. Pelo fato de ser instâncias de tags é possível usar URI, ponto chave para a Web Semântica. Não define regras de cardinalidade sobre os números dos *labels* que uma tag pode ter. Aplica o padrão SKOS para relação entre *tags* e *foaf:Agent* para identificar quem tageou (NEWMAN, 2005).

SCOT: Apesar do foco ser para atividades de tagueamento colaborativas, também se encaixa para folksonomia. Usa conceitos e propriedades herdadas do modelo de Newman, como o conceito de *tagging*. Possibilita a interoperabilidade de dados de tag entre fontes heterogêneas. Possui representação dos elementos *source*, *user group* e *tag set* (conjunto de *tags*). Não possui propriedade direta para descrever quem estava envolvido na atividade de tagueamento. Descreve quem usa a tag, por meio da propriedade *scot:usedBy*. Possui atributos explícitos para tratamento das principais deficiências da folksonomia, como *scot:acronym, scot:plural, scot:singular, scot:sinonimo*. Permite a troca de *tags* metadatas semânticas entre aplicações sociais e interoperação entre fonte de dados, serviços ou agentes no *tagspace*. Emprega o padrão SIOC para descrever informações de sites e relacionamentos entre recursos de sites, FOAF para representar maquinas ou homens como agente e SKOS para caracterizar os relacionamentos entre *tags* (KIM et al, 2008)(KIM et al, 2008).

MOAT: Voltado para a anotação semântica, provendo significado ao tagueamento de texto (*free-text tagging*). Além da proposta por Newman (*Tag, Tagging* e *Tagger*), apresenta a classe *Meaning*, que representa significados customizados e escritos pelos usuários, não deixando o significado da tag ser ambíguo. Pode-se descrever o processo de *tagging* por meio da ligação da *tags:Tagging* proposta na ontologia de Newman. Não possui classe para descrever grupos, e utiliza o padrão SIOC (PASSANT, 2007).

Knerr: Foi estendido para suportar folksonomia, pelas classes *ServiceDomain* e *Source*. Permite usar *foaf:Group* e *foaf:Person* para descrever os grupos de usuários. É o mais completo para representar tagueamento e Folksonomia. Criou mais conceitos como o tempo, domínio, visibilidade e tipo para as *tags*. Utiliza o padrão SKOS para relacionar semanticamente uma Tag com outra, e FOAF para representar as pessoas e seus relacionamentos (KNERR, 2006).

Echarte: Propôs o modelo de folksonomia baseado nas ideias de Gruber e Newman. Prove a classe *Annotation*, que representa uma atividade de tagueamento.

NAO: *Nepomuk Annotation Ontology* (SCERRI et al, 2007) corresponde a uma ontologia desenvolvida voltada para anotação de recursos para semântica aplicada em desktop (Social semantic Desktop). Apesar de não ter como foco o processo de tagueamento como os demais, apresenta grande relevância para o tagueamento em sistemas sociais (KIM et al, 2008).

Analisando as ontologias por meio de artigos publicados, sites e a partir do próprio arquivo da ontologia disponibilizada no formato *rdfs* e *owl*, foi elaborado o Quadro 4-7. Apesar de algumas divergências, com exceção ao Gruber, devido estar definida apenas em nível conceitual, todas apresentaram grandes contribuições significativas. As questões na qual a ontologia atendia aos critérios de seleção, mas não completamente, foram caracterizadas pelo meio termo “parcial”. Este critério foi elaborado tendo como base o resultado da comparação entre a melhor ontologia com a menos satisfatória para o requisito. Esta caracterização não foi utilizada como seleção criteriosa, mas sim, apenas como comparativo.

Quadro 4-7 Comparação das ontologias com os critérios elaborados

Querstão	Gruber	MOAT	SCOT	Newman	Knerr	NAO	Echarte
1. A ontologia existe e está implementada	Não	Sim	Sim	Sim	Sim	Sim	Sim
2. Auxilie na construção e manutenção da folksonomia	Sim	Sim	Sim	Não	Sim	Não	Sim
	Propõe *object*, *tag*, *tagger* e *source*.	Mas não implementa UserGroup	Implementa a TagCloud, o que torna possível unir duas persnomias, formando a folksonomia	Descreve relacionamento entre agente, recursos arbitrários e um ou mais tags. Mas seu foco está na personomia.	Estende conceitos de tempo, dominio, visibilidade e tipo.	Nao possui os elementos básicos para caracterizar uma tripla (triple) de Folksonomia	Mas não implementa UserGroup.
3. Resolver questões de ambiguidades, sinônimos, acrônimos, sinonímia e polissemia	Não	Sim	Sim	Parcial	Sim	Não	Parcial
		Pela classe *Meaning* adiciona relacionamentos extras a Tag, oferecendo mais significados a mesma Tag	Possui a melhor implementação, para resolver questões de ambiguidade. Oferece o Object Property base *spelling_variant*	Ambiguidade - equivalentTag Acrônimo e sinonímia - relatedTag	Sim. Implementa a propriedade sameAs		Adiciona descrições Texto auxiliares a Tag, mas não é possíel realizar um relacionamento semântico com outra Tag.
4. Possível de separar as tags pessoais com as tags de comum utilização	Não	Sim	Não	Não	Sim	Não	Sim
	As tags são criadas por um único usuário apenas.	Define Global Meaning e Local Meaning	Cada pessoa tem sua própria pesronomia, e por meio da TagCloud ocorre a junção dessas personomias, formando a folksonomia.	Não é possível. Teria que adaptar, criando innstancias de agentes separados, sendo que cada agente corresponderia a uma tag classificada localmente e comumente.	Pode ser implementado pelas classes de visibiliadde: Private, Protected e Public.		Possui as clsses TagCommon e TagPersonal que herdam da Tag
5. Esteja implementada como lightweight ontology	Nao	Sim	Sim	Sim	Sim	Sim	Sim
7. Possível de linkar recursos por inter-relacionamentos das tags	Não	Sim	Sim	Sim	Sim	Não	Não
	Não está claro no modelo conceitual os tipos de relacionamentos entre as tags.		Inclusive com recursos de fontes heterogeneas	Descreve relacionamento entre agente, recursos arbitrários e um ou mais tags.	Tag pode ter vários labels. Uma Tag pode se relacionar semanticamente com outras tags porque a Tag é uma subclasse de skos:Concept e pelo Object Propertie *hasTaggedResource*		Não há relacinamento de tags equivalentes especificadas
6. Tagear e descrever recursos: este é o requisito essencial	Não	Sim	Sim	Não	Sim	Sim	Sim
		Possui Classe Item. Uma simples Tag pode se relacionar a diferentes URIs. Também é possível descrever por meio da classe moat:Meaning	Por meio do relacionamento da tag com o Item, é possível caracterizar bem o objeto sendo tageado	Não existe classe recurso especificada. A ontologia deverá ser adaptada. O relacionamento de associação é diretamente com a super classe Thing	Apresenta várias tags para caracterizar os relacionamentos. Caracteriza até o "o que" e o "tipo" do relacionamento, por meio de classes.		Pela classe AnnotationTag
8. Realização semântica eficaz para as necessidades de inferência de conhecimento	Não	Parcial	Sim	Parcial	Sim	Não	Parcial
		Os metadados existentes são unicamente voltados para as tags.		Não há metadados suficientes para um relacionamento adequado. Teria que criar mais classes e Object Properties para enriquecer semanticamente.			Não há metadados suficientes. Seria necessário acrescentar novas propriedades (Object Properties)

No entanto, além de a ontologia de Knerr atender melhor aos requisitos, a mesma foi escolhida não pela sua implementação, mas sim pela disponibilização e fácil acesso à documentação da ontologia, das classes e das propriedades. Contudo, além deste diferencial, vale citar que a ontologia de Moat disponibiliza em seu site um servidor e um cliente que podem ser baixados e instalados. Trata-se de uma infraestrutura pré montada e pronta para utilização, sendo um servidor que armazena os diferentes significados das tags e um cliente, que interage com o servidor, permitindo que o usuário facilmente realize anotações.

4.3 Considerações do capítulo

Este capítulo apresentou o método proposto para a extração do conhecimento do código fonte. Método este proposto com base na revisão da literatura, enfatizando o sensemaking. Foram descritos as atividades e tarefas compostas pelo método, assim como a descrição de cada item.

Para complementar o método, uma análise sobre as ontologias existentes foi realizada. Dentre as ontologias analisadas, a que satisfez melhor o atendimento dos requisitos foi a ontologia desenvolvida de Knerr, devido a sua implementação, facilidade de entendimento e acesso a documentação.

Os capítulos seguintes correspondem a extensão do método proposto, focalizando com conceitos, técnicas e demais pontos essenciais para a implementação e execução do método.

CAPÍTULO 5 - AMBIENTE TAGGINGSENSE

Este capítulo tem por objetivo apresentar o ambiente desenvolvido para apoiar a aplicação do método proposto baseado em sensemaking. Primeiro será apresentada a arquitetura do *ambiente* e sua estruturação, desde a organização das ontologias, população e interação com o código fonte, e depois a sua respetiva implementação, na forma de um plugin para a IDE de desenvolvimento Eclipse.

5.1 Introdução

Para idealizar o projeto a fim de capturar o conhecimento dos desenvolvedores e popular uma ontologia de conhecimento, foi desenvolvido um *plugin* para a IDE Eclipse, com foco para projetos escritos na linguagem Java.

Eclipse é uma comunidade *open source* cujo foco está no desenvolvimento de uma plataforma de desenvolvimento extensível, *runtimes* e *frameworks* de aplicação para a construção, implementação e gerenciamento de software e todo o ciclo de vida (ECLIPSE, 2012).

A lógica do ambiente consiste em analisar os arquivos de código fonte do projeto Java, realizar o parser das classes, métodos e atributos e, a partir do resultado, popular a ontologia SCRO, específica de orientação a objetos. A população desta ontologia não é um fator crucial para o processo como um todo, pois a utilização desta ontologia servirá apenas de apoio ao processo de extração e organização do conhecimento.

A partir da ontologia SCRO populada é disponibilizado ao usuário o mecanismo de taguear o código fonte. O processo de taguear consiste em adicionar informações aos indivíduos da ontologia de orientação a objetos. Estas informações correspondem a palavras chaves da *tag*, data e hora e criador, podendo ser adicionada a mesma *tag* criada para demais indivíduos.

No final do processo é apresentado ao programador as *tags* criadas e compartilhadas por mais usuários, com objetivo de guiá-los e auxiliá-los na manutenção e compreensão do código fonte.

A seguir serão descritas as etapas do processo, bem como sua arquitetura, implementação e questões relevantes.

5.2 Arquitetura

O desenvolvimento do ambiente está estruturado em camadas, conforme ilustra a Figura 5-1, com o objetivo de melhor adequar ao método, além de facilitar o desenvolvimento e os testes.

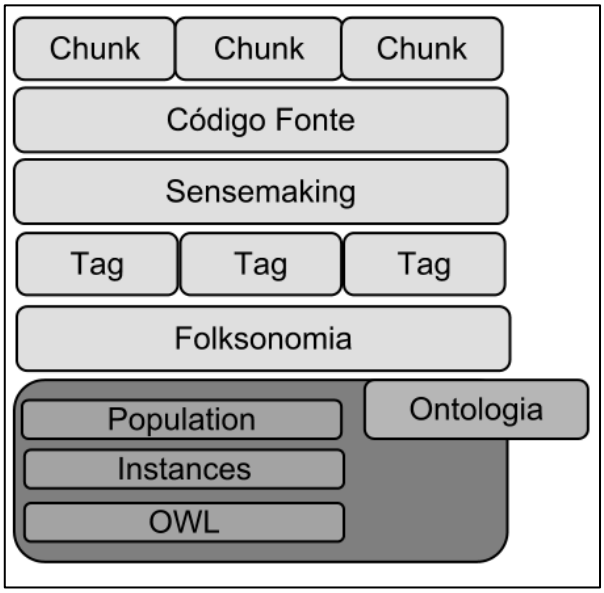

Figura 5-1 - Processos para extração do conhecimento

A camada de código fonte, identificada na Figura 5-1 pelos “Chunk”, e pelo “Código Fonte”, corresponde à estruturação do código fonte e das informações contidas nestes artefatos de software. Esta camada engloba a ontologia de código fonte, o processo de extração das informações do código e a população da respectiva ontologia.

A camada “Sensemaking” está relacionada com a ação direta do usuário, na qual tem-se como resultado a população da ontologia de folksonomia, por meio da extração manual do conhecimento realizada pelo programador no código fonte.

Tanto as camadas relacionadas ao código fonte quanto a camada de folksonomia resultam na população da ontologia, que corresponde a instâncias criadas e mantidas no padrão OWL.

A seguir serão detalhados os principais aspectos de cada camada da arquitetura do projeto.

5.2.1 Ontologia de Código Fonte

A ontologia de SCRO especifica os principais conceitos de orientação a objetos (OO). Cada conceito criado na ontologia é interpretado como um conjunto que define um grupo de instâncias que compartilham propriedades em comum. Classes disjuntas foram modeladas para caracterizar diferentes tipos de instâncias, como por exemplo, definir que todas as instâncias que são membros da classe *InstanceMethod* são necessariamente membros da classe *Method*, mas nenhuma delas pode ser membro de outras subclasses de *Method* (ALNUSAIR, 2010).

A Figura 5-2 apresenta a ontologia de código fonte. À esquerda estão as classes que fazem parte da taxonomia da ontologia, enquanto que à direita estão as propriedades de objetos associadas às classes.

Como o objetivo é utilizar esta ontologia como apoio ao processo de sensemaking, apenas alguns conceitos e classes presentes na ontologia foram utilizados. A seguir serão descritas as classes da ontologia e sua utilização no processo de população da ontologia.

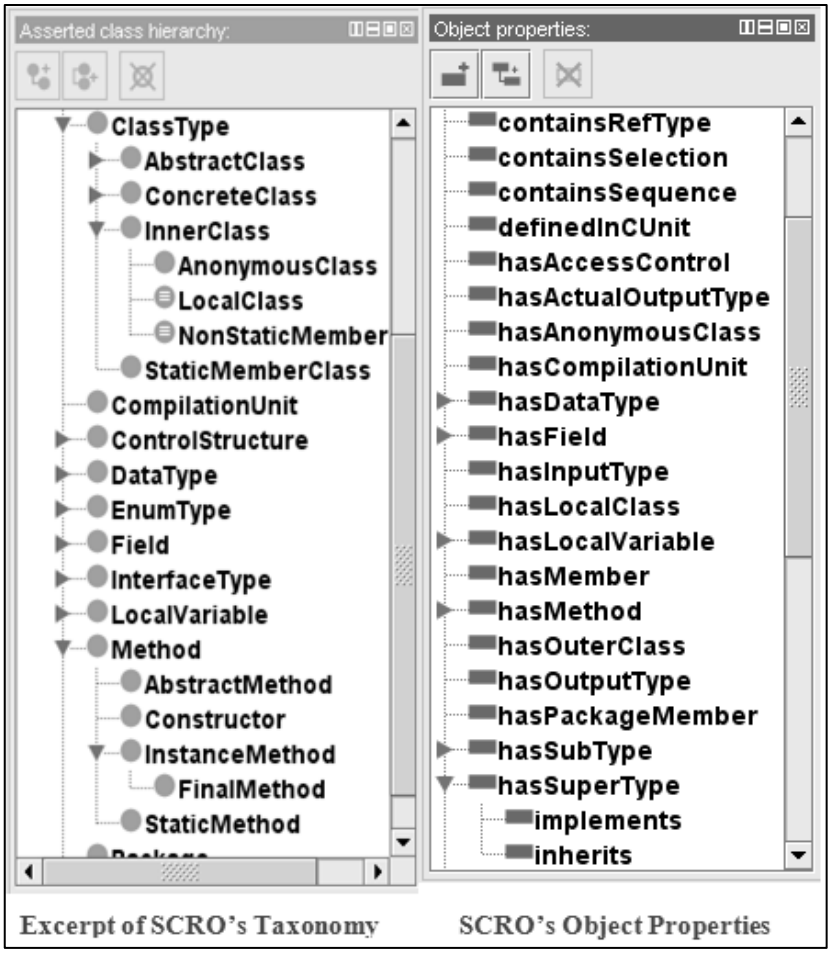

Figura 5-2 - Classes e propriedades da ontologia de código fonte. Adaptado de (ALNUSAIR, 2010)

Package: armazena o nome completo do pacote, por exemplo: br.com.xiraba.bo.

ClassType: corresponde a classe genérica que especifica os tipos de classes na linguagem OO. Os tipos de classes que são herdados da *ClassType* são: *AbstractClass, ConcreteClass* e *InnerClass.* Conforme o tipo de classe lido no código fonte Java, uma instância da ontologia é criada, relacionando-se com a *Package* por meio da propriedade *isPackageMemberOf*.

InterfaceType: semelhante ao *ClassType*, também se relacionando ao pacote por meio da propriedade *isPackageMemberOf*.

AccessControl e *AccessModifier:* correspondem aos modificadores genéricos da linguagem de programação: *private*, *protected* e *public*. São as classes bases para as classes que identificam os modificadores. As classes que herdam são: *PublicModifier*, *ProtectedModifier* e *PrivateModifier*. Para cada classe, interface, método e atributo lido do arquivo de código fonte, um dos modificadores são associados às respectivas instâncias, por meio da propriedade *hasAccessControl* disponível nas classes *ClassType, Field, Method* e *InterfaceType*.

Method: armazena dados referentes ao método da classe. Um método pode ser caracterizado como abstrato, construtor, final, estático ou padrão. As características são detalhadas pelas classes *AbstractMethod*, *Constructor, FinalMethod, StaticMethod* e *Method* respectivamente. De acordo com o tipo lido, a ontologia é populada, além das propriedades que as contemplam: *hasAccessControl, isMemberOf*, que mantêm referência à classe ou interface ao qual o método pertence, além dos parâmetros de entrada e saída.

Parameter: classe que representa os parâmetros de entrada de um método. Armazena o nome e a relação com o método, a partir da propriedade *hasParameter* da classe *Method.*

PrimitiveDataType: classe que representa o tipo primitivo do dado. Os tipos primitivos básicos suportados são: *boolean, byte, char, double, float, int, long* e *short.*

Field: classe que representa um atributo de uma classe Java. Um atributo é criado com base no seu nome, modificador e a classe que pertence, através das propriedades *hasAccessControl* e *IsMemberOf*.

5.2.2 Leitura Código Fonte

Sob a perspectiva de interação do usuário com o *plugin*, os *Chunks*, assim como o código fonte, correspondem ao primeiro estágio do contato entre o programador e o ambiente.

A interação inicia a partir da compreensão do código fonte pelo programador de forma *bottom-up*, ou seja, de baixo, representado pelas linhas do código fonte, para cima, representando o conhecimento do domínio, através da identificação dos *chunks* relevantes. *Chunks* são porções do código que programadores reconhecem. Grandes *chunks* contêm vários outros pequenos *chunks* (ZHANG, 2007) (LETOVSKY, 1986). Durante a compreensão, tags são criadas e associadas, pelo programador, ao código fonte. Estas tags criadas correspondem a instâncias criadas na ontologia de folksonomia.

Mas antes da criação manual das tags, é necessário que o código fonte esteja previamente processado e sincronizado com a ontologia de código fonte. Esta sincronização consiste na extração das informações dos arquivos Java do projeto, como métodos, valores de entradas e saídas de cada método, atributos da classe e a própria classe, e populadas na ontologia de código fonte, conforme ilustrado pela Figura 5-3.

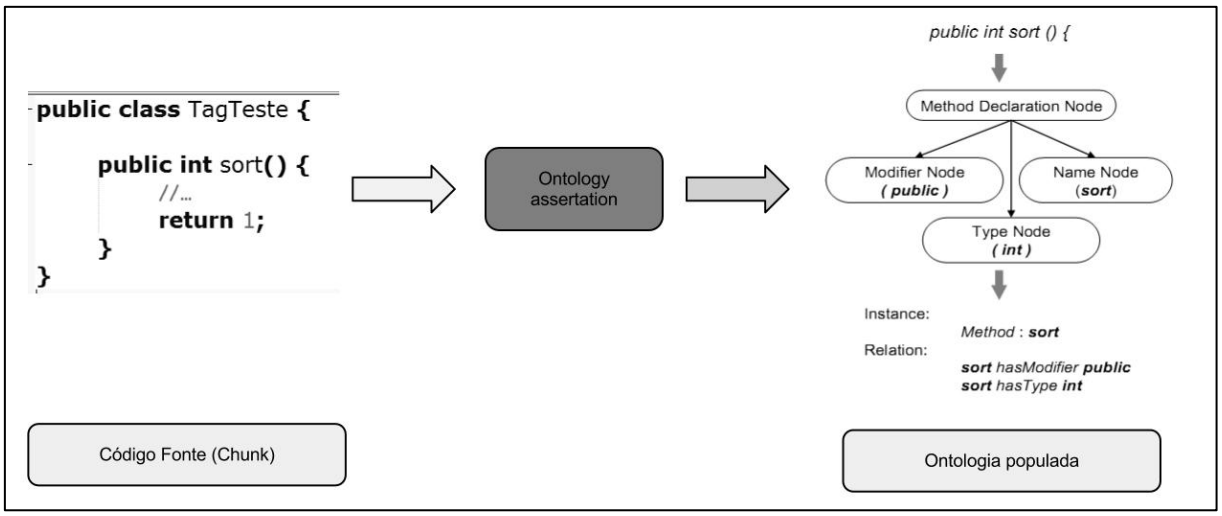

Figura 5-3 - População da ontologia de orientação a objetos a partir do código fonte

A realização da extração destas informações ocorre de forma automática pelo *plugin*, por meio da classe *SourceCodeExtractor*. Esta classe recebe como parâmetro de entrada o arquivo Java ou o diretório do código fonte do projeto, que

por meio do processamento recursivo, analisa cada arquivo Java e realiza a população da ontologia de código fonte.

Por meio da biblioteca QDox que a classe *SourceCodeExtractor* realiza a leitura dos arquivos fontes. QDox é um analisador robusto de alta velocidade para extração de definições de classes, interfaces e métodos do código fonte. Foi projetado para ser usado por geradores de códigos ativos ou ferramentas de documentação (QDOX, 2012).

A análise de cada arquivo fonte Java é realizada a partir das leituras dos pacotes. Para cada pacote são lidas todas as classes. Em seguida, para cada classe são lidos os atributos e métodos, assim como o tipo, os parâmetros e o retorno. Tendo toda a estrutura lida, as informações são armazenadas na OWL de código fonte.

A lógica da leitura e, consequentemente, da população da ontologia de código fonte pela classe *SourceCodeExtractor*, é realizada a partir da adaptação de (GANAPATHY, SAGAYARAJ, 2011), conforme demonstrado abaixo:

```
1   Lê os pacotes analisados pelo QDox.
2       Para cada pacote lido salva sua definição na ontologia OWL de código fonte.
3       Lê as classes contidas no pacote.
4           Para cada classe lida salva sua definição na ontologia OWL.
5           Guarda o modificador da classe na ontologia OWL.
6           Associa a classe ao pacote.
7           Lê os atributos da classe
8               Para cada atributo lido salva seu nome e sua definição na ontologia OWL.
9               Salva o tipo do atributo.
10          Lê os métodos da classe
11              Guarda o modificador do método na ontologia OWL.
12              Associa o método a classe
13              Guarda o tipo de retorno do método na ontologia OWL.
14              Para cada parâmetro de entrada contido no método
15                  Guarda o nome do parâmetro na ontologia OWL.
16                  Guarda o tipo do parâmetro na ontologia OWL.
17              Fim
18          Fim
19      Fim
20  Fim
```

A leitura e escrita na ontologia é realizada por meio da biblioteca *semantic web* (OWL-API). Esta biblioteca é uma *interface* Java, implementada pela W3C *Web Ontology Language*, permitindo trabalhar com recursos da Web Semântica dentro dos projetos Java, suportando realizar integração com os raciocinadores *FaCT*++ e *Pellet*. Seu foco está direcionado para OWL-DL e OWL 2, sendo largamente utilizado por diversas instituições (OWLAPI, 2012).

Uma das vantagens da utilização da biblioteca *semantic web* é a sua integração com a plataforma de desenvolvimento de ontologia *Protégé*. Por meio do *plugin* Protégé-OWL *Code Generator* é possível gerar código Java a partir de uma ontologia específica. Desta forma, a manipulação dos *beans, data objects* e *properties* são gerados e encapsulados por classes *wrappers*, o que torna o desenvolvimento e a manipulação de recursos da Web Semântica em projetos Java mais fácil e segura (REDMOND, 2012).

A Figura 5-4 apresenta o código fonte Java gerado a partir do Protégé-OWL *Code Generator*. A partir do código gerado é possível interagir com a ontologia por meio de métodos de acessos, da mesma forma que se acessa e interage com métodos de classes criadas em Java.

Durante o processo de leitura das classes Java pelo *SourceCodeExtractor*, os *beans* gerados da ontologia de código fonte são instanciados e acessadas conforme o andamento da leitura da classe Java, e suas propriedades valorizadas e salvas no ontologia. A Figura 5-5 demonstra os passos executados entre leitura do código fonte pela classe *SourceCodeExtractor* e a população da ontologia de código fonte.

```
package edu.uwm.cs;

import java.util.Collection;

/**
 *
 * <p>
 * Generated by Protege (http://protege.stanford.edu). <br>
 * Source Class: Package <br>
 * @version generated on Sat Apr 14 16:19:48 BRT 2012 by Daniel
 */
public interface Package extends WrappedIndividual {

    /* ************************************************
     * Property http://www.cs.uwm.edu/~alnusair/ontologies/scro.owl#annotatedBy
     */

    /**
     * Gets all property values for the annotatedBy property.<p>
     *
     * @returns a collection of values for the annotatedBy property.
     */
    Collection<? extends AnnotationType> getAnnotatedBy();

    /**
     * Checks if the class has a annotatedBy property value.<p>
     *
     * @return true if there is a annotatedBy property value.
     */
    boolean hasAnnotatedBy();

    /**
     * Adds a annotatedBy property value.<p>
     *
     * @param newAnnotatedBy the annotatedBy property value to be added

    ...
```

Bean que representa uma instância da classe Package

Propriedades da ontologia geradas

Figura 5-4 - Código Java gerado a partir do Protégé-OWL *Code Generator*

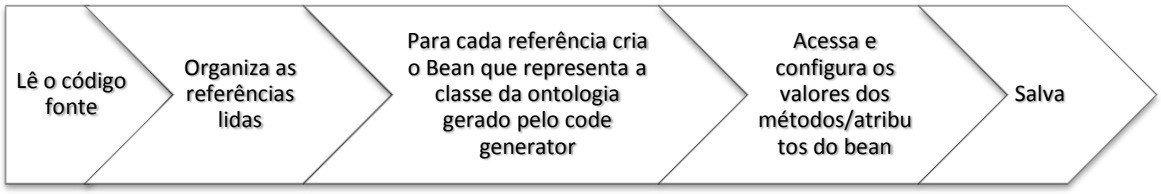

Figura 5-5 - Etapas executadas para ler o código fonte e popular a ontologia

Ao salvar um *bean*, suas propriedades são automaticamente transformados em instâncias de indivíduos e persistidos no banco de dados MySQL.

A biblioteca OWL-API se integra facilmente com o banco de dados. Devido à um número considerável de acessos, busca e persistência de dados na ontologia por vários usuários simultaneamente, torna-se mais vantajoso e produtivo persistir a

ontologia em um banco de dados. Aquém a estes fatores, um sistema baseado em banco de dados melhora a funcionalidade do sistema, além do gerenciamento escalável e controle de transações (AUER, IVES, 2007). Para contemplar esse requisito, a biblioteca OWL-DB foi adotada para persistir as ontologias em banco de dados, tanto as definições *T-Box* quanto *A-Box*.

A biblioteca OWL-DB, desenvolvida baseada no padrão OWL-API, atua entre a camada de gerenciamento e persistência, sobrescrevendo as rotinas básicas de manipulação de dados externos. Desta forma, sua utilização é quase que transparente ao projeto, pois sua chamada é configurada apenas na classe base da OWL-API, a *OWLOntologyManager*.

Junto a biblioteca OWL-API atua o motor de inferência Pellet. Pellet (PELLET, 2012) fornece serviços de raciocínios para ontologias OWL 2. Sua utilização, neste ambiente, ocorre sempre que uma nova instância na ontologia é criada em tempo de execução, com objetivo de avaliar as consistências.

Uma das desvantagens da OWL-API é a falta de suporte nativo a consultas SPARQL. Para suprir esta necessidade a biblioteca SPARQL-DL *Query Engine* foi acoplada ao projeto, atuando na camada mais alta da OWL-API. A linguagem de consulta SPARQL-DL é um subconjunto distinto da SPARQL, corresponde a uma linguagem expressiva e permite misturar particularmente consultas *T-Box, R-Box* e *A-Box* (DERIVO, 2012).

Tendo a ontologia de código fonte populada, o próximo passo é realizar a interação com a ontologia da folksonomia, possibilitando que novos indivíduos criados nesta ontologia, por meio da criação de *tags* a partir da ação do programador, sejam associados conforme instâncias de indivíduos da ontologia de código fonte.

5.2.3 Integração da Ontologia Código Fonte com a Ontologia de Folksonomia

Para integrar a ontologia de código fonte com a ontologia de folksonomia foram avaliadas algumas técnicas existentes de alinhamento.

A técnica *Distributed Description Logics* (DDL) é um formalismo para a combinação de diferentes bases de conhecimento DL em um sistema de informação flexível, preservando a “identidade” e independência de cada ontologia. A partir desta técnica é possível criar uma classe na ontologia A, e configurá-la para ser uma subclasse da ontologia B, por exemplo. Esta técnica se torna interessante para a

Web Semântica, pois facilita o desenvolvimento de uma rede de distribuição ligadas e independente entre ontologias (GRAU et al, 2004).

A técnica de ligação *E-Connection* é um método para combinar formalismos lógicos que são expressáveis no *Abstract Description System (ADS).* Corresponde a uma linguagem de representação do conhecimento definida como a combinação de outros formalismos lógicos. Esta técnica permite fornecer, de forma expressiva, a combinação de bases de conhecimento escritos em uma variedade de linguagens lógicas, além do baixo acoplamento entre a combinação dos formalismos lógicos existentes (GRAU et al, 2004).

Outra técnica de integração entre ontologias ocorre a partir da *Ontology merging*. Esta técnica corresponde a criação de uma ou mais ontologias com base em duas ou mais ontologias. A nova ontologia unificará e, em geral, substituirá a ontologia base (BRUIJIN et al, 2006). Esta técnica foi utilizada para a integração das ontologias neste trabalho por causa da persistência em banco de dados. Desta forma, se torna mais fácil e coerente trabalhar com um único banco de dados, que representar as ontologias de folksonomia e código fonte em ontologias diferentes e, consequentemente, banco de dados distintos.

A interação é um fator crucial para realizar a extração do conhecimento e seu arquivamento, pois todo conhecimento extraído pelo programador está relacionado a algum item de código fonte. Desta forma, uma instância de uma *tag* criada estará, obrigatoriamente, relacionada a uma instância de um item de código fonte.

O resultado da integração será um link entre uma instância da classe *tagging/resource* e uma instância de um item de código fonte, conforme visto na Figura 5-6.

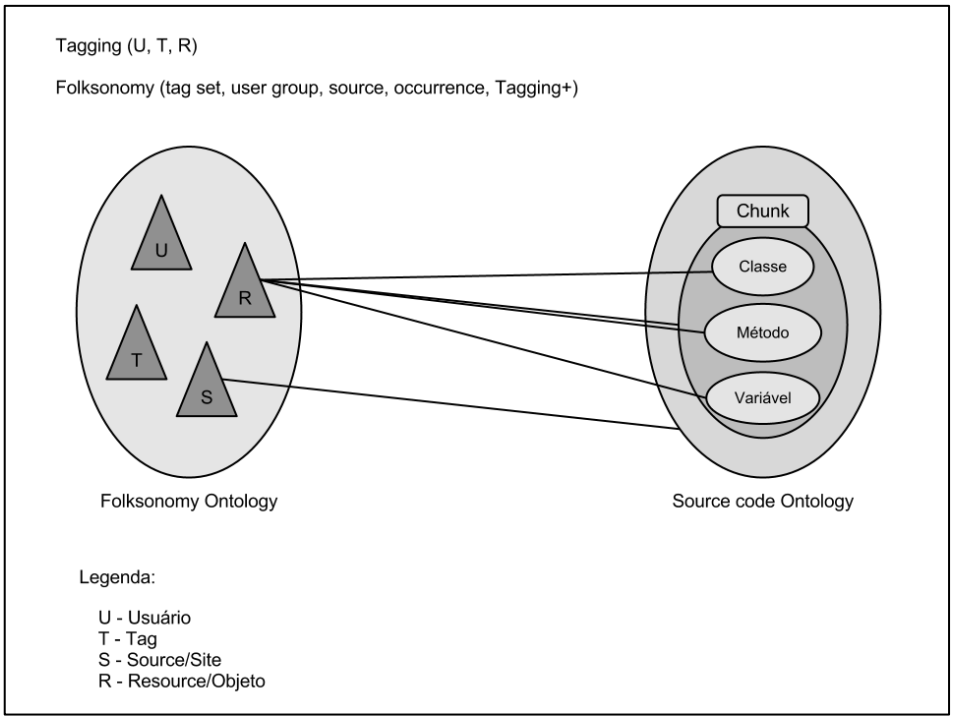

Figura 5-6 - Relacionamento entre as ontologias da folksonomia e código fonte

A ligação entre ambas é realizada por meio das instâncias criadas na ontologia *Source Code*, que correspondem a classes, métodos, variáveis (atributos) e *chunk*. A classe *Resource*, da ontologia de folksonomia, guarda referência às instâncias no momento de criar uma nova *Tagging*. Independentemente da *Tag* associada, o *Resource* será sempre associado novamente ao processo de *Tagging*.

Por fim, o processo resultará em uma base de conhecimento contendo todas as tags criadas e suas respectivas associações, derivadas do conhecimento do domínio auferidos a partir do código fonte. A base de conhecimento consiste na própria ontologia da folksonomia populada e inferida pelos mecanismos de inferência. A Figura 5-7 descreve este processo de forma macro.

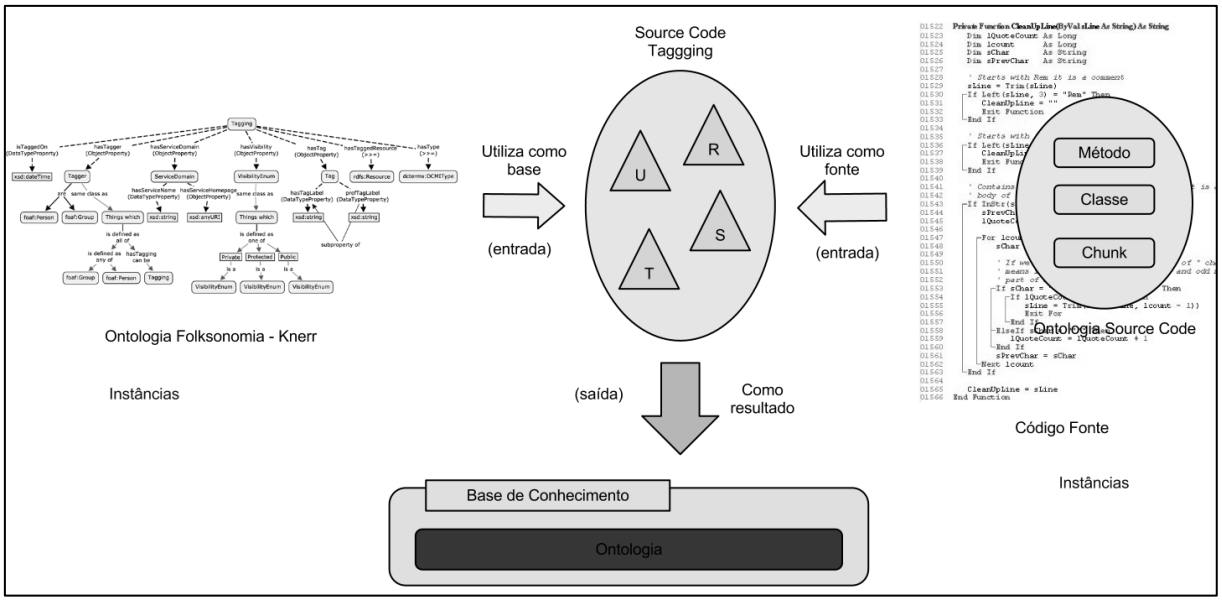

Figura 5-7 – Geração da base de conhecimento a partir da união das ontologias de código fonte e folksonomia

O próximo passo apresenta o desenvolvimento do ambiente, tendo como consideração a arquitetura e modelos adotados durante a fase de estruturação da arquitetura.

5.3 Implementação do Ambiente

Primeiramente serão detalhados os requisitos para o desenvolvimento do ambiente, baseado nas premissas de sensemaking, folksonomia e base de conhecimento. Em seguida serão ilustrados os detalhes de utilização do ambiente desenvolvido.

5.3.1 Requisitos

Baseado nos estudos desenvolvidos e no detalhamento do ambiente, foram levantados os requisitos de implementação para apoiar o processo de compreensão de código fonte.

Requisito 1: Para capturar e armazenar o conhecimento de domínio do código fonte é preciso que o *plugin* seja modelado e preparado para consultar e gravar informações de domínio na ontologia de folksonomia. As informações referem-se ao conhecimento adquirido durante o processo de compreensão, e devem ser semanticamente interligadas para permitir a realização de consultas e inferências (*reasoning)*.

Requisito 2: População da ontologia de código fonte. Como é muito vulnerável popular uma ontologia manualmente, e devido a necessidade de lidar com grande quantidade de código e informações de sistemas grandes e complexos, não é garantido que a população manual seja eficaz e segura, além de se tornar uma tarefa repetitiva. Desta forma, o *plugin* deve fornecer um método para a extração das informações semânticas do código fonte e popular a ontologia de código fonte automaticamente.

Requisito 3: População da ontologia de folksonomia. A população da ontologia de domínio, que corresponde às *tags* criadas, deve-se realizar de forma manual. Ao contrário das ferramentas e trabalhos desenvolvidos para auxiliar a compreensão do código fonte TagSEA (STOREY et al, 2006) e SOUND (ZHANG, 2007), onde ocorre uma mineração de texto com base no código fonte e extração dos tópicos e *tags*, o processo de compreensão do código fonte, em consequência do sensemaking, é melhor desenvolvido de forma manual, pois é neste momento em que o usuário estará assimilando e entendendo o código fonte. O resultado do tópico/conhecimento manualmente extraído do código fonte resultará na população da ontologia de domínio. Desta forma, o sistema deverá possibilitar criar *tags* com base em trechos de código fontes selecionados e popular a ontologia específica.

Requisito 4: Buscas de instâncias na ontologia. Artefatos de softwares são, frequentemente, auto consistentes (ZHANG, 2007), permitindo manter as integridades estruturais e semânticas por meio de ferramentas específicas, como, por exemplo, um compilador. No entanto, a representação do conhecimento, na forma de ontologia, deve lidar com grandes quantidades de informações provenientes do código fonte e do usuário (realização do sensemaking). Grandes quantidades de instâncias e seus relacionamentos exigirá a aplicação de raciocínios na ontologia que suporte buscas de instâncias complexas (A-Box).

Requisito 5: Com base nos requisitos 1, 2, 3 e 4, nos processos do sensemaking e na estrutura da folksonomia, o *plugin* deverá permitir criar, relacionar, disponibilizar, identificar, consultar e compartilhar as *tags* durante o processo de compreensão do código fonte.

Requisito 6: Integração com o ambiente de trabalho. Todo o processo de aquisição de conhecimento e buscas deve ser integrado à IDE de desenvolvimento, para que o usuário não perca o foco durante o processo de compreensão do código fonte.

5.3.2 Visão geral do Ambiente

Para que seja possível extrair automaticamente o código fonte e permitir a interação direta com o usuário, o sistema foi projetado e desenvolvido baseado na plataforma Eclipse 3.6 e Java 6. A Figura 5-8 ilustra o ambiente e os seus componentes.

A manipulação do código fonte, de característica automática, popula a ontologia de código fonte, por meio do QDox, enquanto o resultado das manipulações das *tags* são manuais, conforme a ação do usuário. O código fonte é o único artefato de software de entrada, visto que as demais entradas no sistema são por meio da intervenção manual.

As consultas por *tags* criadas e populadas ocorrem por meio do SPARQL-DL, biblioteca de apoio a OWL-API, visto que a mesma não tem suporte nativo a consultas SPARQL.

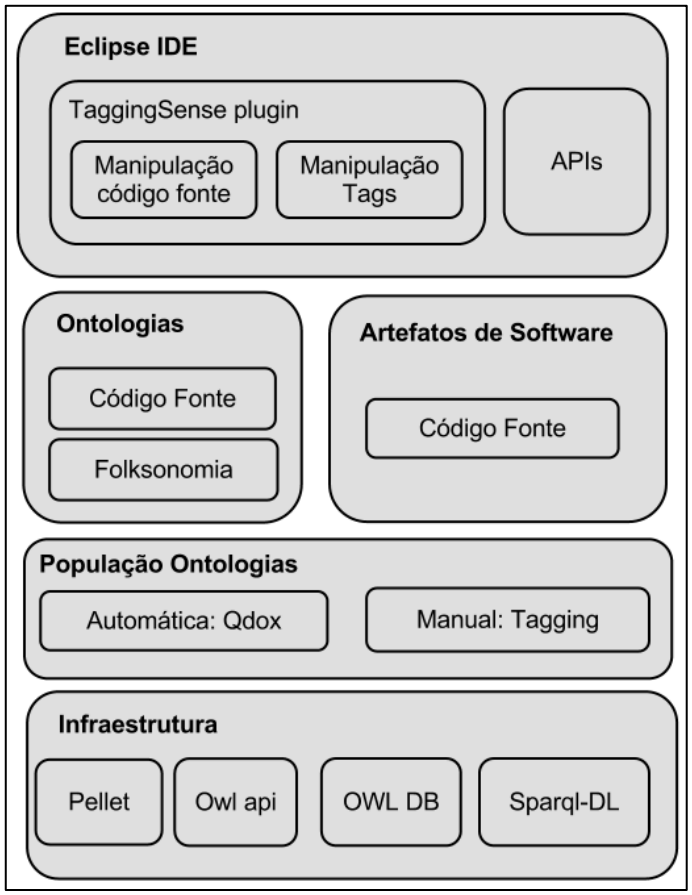

Figura 5-8 - Visão Geral: componentes do ambiente

5.3.3 Características do Ambiente

Baseado nos requisitos de extração e manipulação do conhecimento discutidos anteriormente foi desenvolvido o TaggingSense *Plugin* para manipulação das ontologias e tagueamento no código fonte. Suas funcionalidades estão descritas a seguir:

- **visualizar tags**: lista todas as *tags* públicas criadas por qualquer pessoa, além das *tags* privadas que são de autoria do usuário atual (Figura 5-9 – 1).
- **adicionar novas tags**: por meio do trecho do código fonte selecionado adiciona nova *tag*, com a possiblidade de criar ou reutilizar uma *tag* já existente (Figura 5-9 - 2).
- **visualizar tags relacionados ao código selecionado**: lista todas as *tags* associadas ao código selecionado e/ou posicionado pelo usuário. A partir desta janela é possível analisar a relação do objeto relacionado a programação com o conceito de domínio (*tag*) associado (Figura 5-10 – 1 e Figura 5-10 - 2).
- **visualizar tags em formato de árvore**: lista todas as *tags* e as instâncias dos itens da ontologia de programação associadas a cada *Tag*. É possível navegar no código fonte a partir da interação dos itens da árvore (Figura 5-10 – 3).
- **visualizar utilização de todas as tags**: lista todas as *tags* públicas criadas por qualquer pessoa, além das *tags* privadas que são de autoria do usuário atual. São detalhadas a data e hora de criação, o responsável pela criação e a frequência de utilização da *tag* no projeto (Figura 5-10 – 4).

Também é possível, a partir das janelas de visualização das *tags*, tornar as *Tags* e as *Taggings* públicas, possibilitando que outros usuários possam ver as *tags* e usá-las cooperativamente.

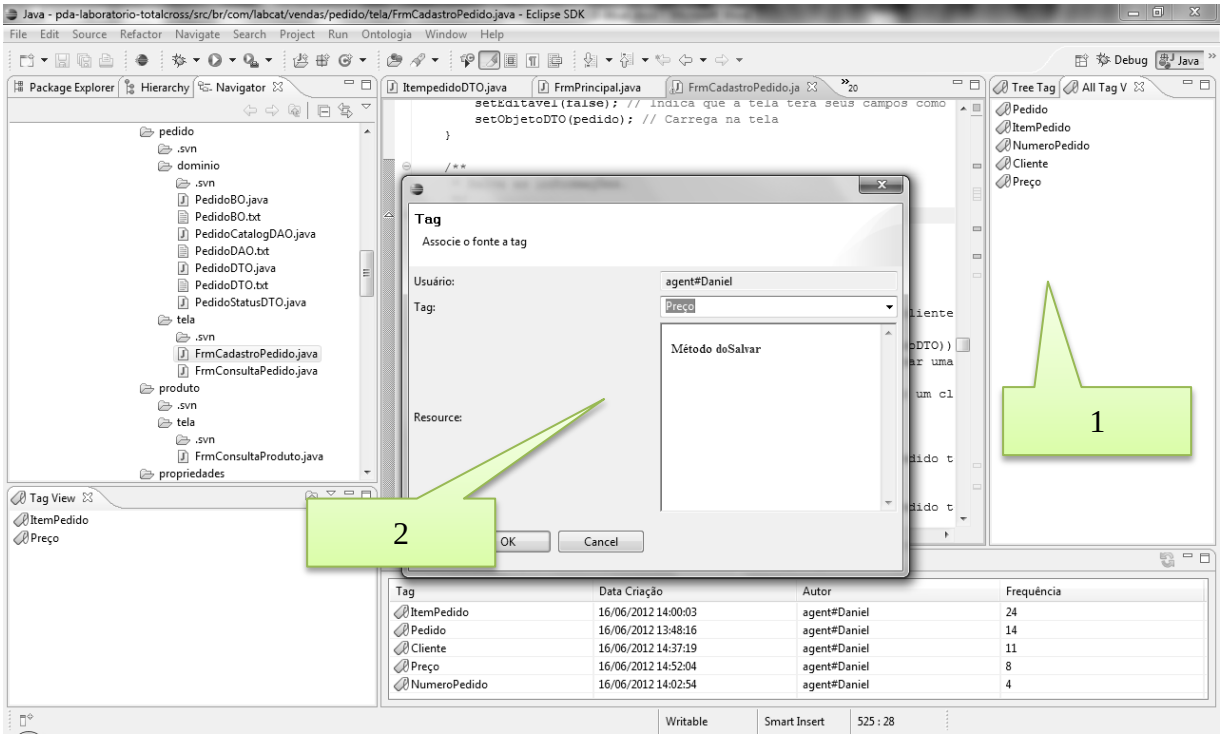

Figura 5-9 – Tela do *Plugin* para Sensemaking: (1) Visualizar *Tags*, (2) Adicionar novas *Tags*

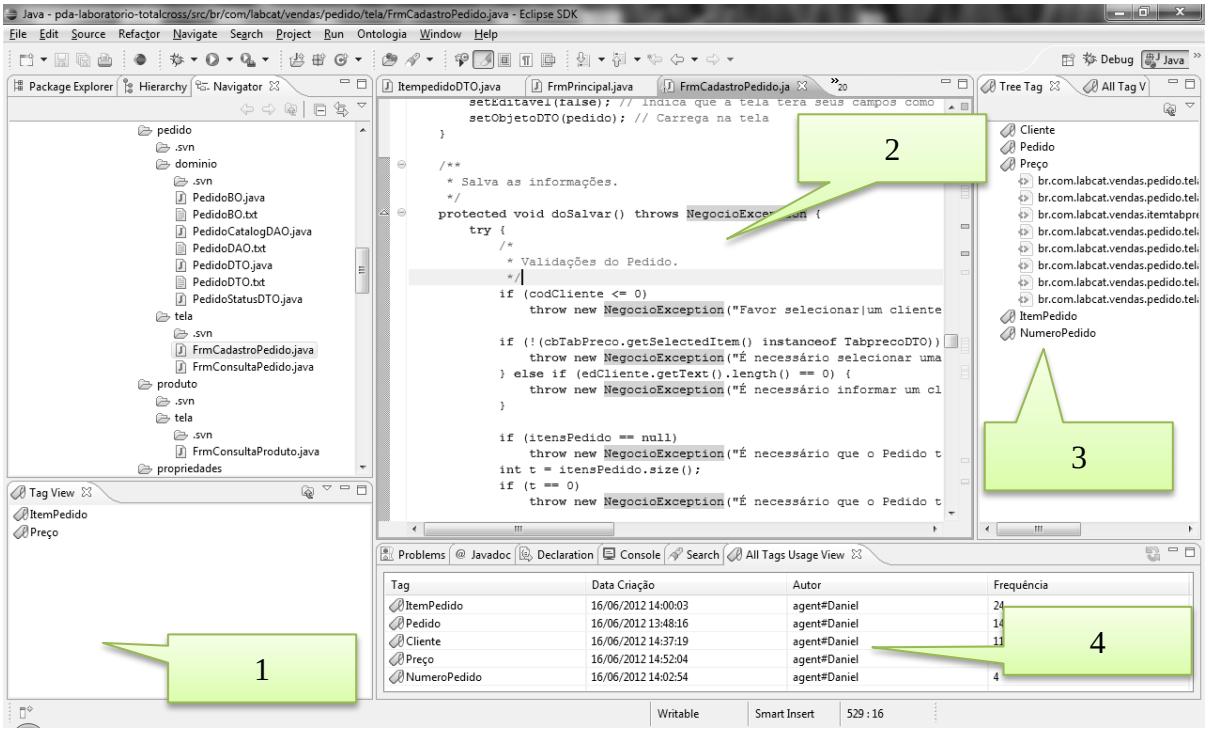

Figura 5-10 – Tela do *Plugin* para Sensemaking plugin: (1)(2) Visualizar *Tags* do código selecionado, (3) Visualizar *tags* em árvore, (4) Visualizar todas as *tags*

A partir das *tags* é possível gerar um grafo, contendo os detalhes de relacionamento e utilização de cada *tag*, conforme demonstra a Figura 5-11.

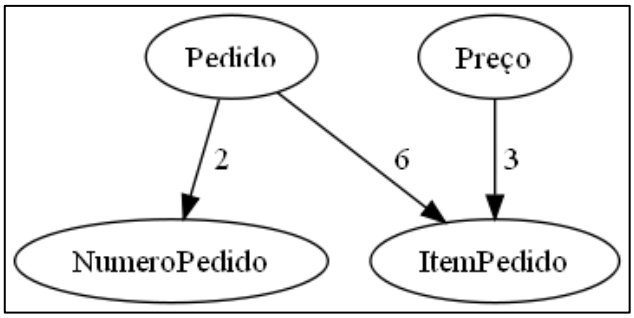

Figura 5-11 - Grafo das *tags* e seus relacionamentos

Cada círculo corresponde a uma *tag* criada pelo programador. As setas indicam que existe relacionamento entre uma *tag* e outra. No caso da Figura 5-11, o "Pedido" possui relação com "NumeroPedido", por meio de um mesmo trecho de código fonte ser tagueado por ambas as *tags*.

Os números contidos nas setas identificam o peso do relacionamento, calculado com base na frequência de utilização das *tags*. No exemplo da Figura 5-11, a *tag* "Pedido" e "NumeroPedido" são utilizadas 2 vezes para identificar um mesmo trecho de código.

5.3.4 Considerações sobre o capítulo

Este capítulo apresentou os detalhes da preparação, integração e implementação das ontologias de código fonte e folksonomia. Apresentou as técnicas, arquitetura utilizada, processos e detalhes das bibliotecas utilizadas na implementação do *plugin* TaggingSense, ferramenta de extração do conhecimento no código fonte.

A partir do desenvolvimento deste *plugin* foi possível realizar experimentos com programadores que atuam na manutenção de código fontes, com objetivo de avaliar o método proposto.

CAPÍTULO 6 - AVALIAÇÃO DO MÉTODO

Neste capítulo são apresentados e discutidos os experimentos realizados para avaliar o método proposto baseado em sensemaking. Com o uso do sensemaking será avaliada também a aplicabilidade do uso de *tags* dentro da IDE de desenvolvimento, com ênfase à manutenção do código fonte. Com base no trabalho realizado por (SHARMA, 2011) referente ao estudo do desenvolvimento do sensemaking a partir de recursos oferecidos, disponíveis e não disponíveis, que correspondem à informações referente ao tema ao qual as pessoas foram convidadas a desenvolver, as seguintes questões foram levantadas para serem avaliadas durante os experimentos:

- **impacto no processo de sensemaking quando alguns recursos são oferecidos e outros não:** avaliar como as *tags* (recursos) disponíveis, assim como os não disponíveis, afetam os processos do sensemaking.
- **a organização das tags no momento do sensemaking:** a partir da extração do conhecimento, novas *tags* serão criadas pelos programadores. Neste quesito, será avaliado o número de *tags*, estruturação e sua coerência com o contexto do ambiente e do domínio. A avaliação levará em consideração ambientes onde não existam *tags*, ou seja, o desenvolvimento das *tags* ocorrerá do início, e onde as *tags* já foram criadas e estão disponíveis para utilização, apoiando e influenciando os programadores no desenvolvimento de novas *tags*.
- **avaliar a estruturação e o desenvolvimento de uma manutenção complexa e uma manutenção simples**: analisar o número de *tags* criadas, sua coerência com o contexto do problema e tempo decorrido em uma manutenção simples, que gera pouco esforço mental para compreender, com outra manutenção onde é exigido maior atenção e compreensão para resolução do problema.

- **avaliar a quantidade de tags e o número de utilizações para os itens fáceis e já conhecidos, comparados com itens difíceis e desconhecidos**: Um item corresponde a uma característica da melhoria, como por exemplo, manipular e alterar um botão de tela (item fácil), ou alterar uma parte de cálculo de uma regra de negócio específica (item complexo). Nesta situação, será avaliado e comparado a criação e a frequência de utilização de *tags* na manutenção de itens fáceis e difíceis.
- **tempo despendido para realizar o sensemaking, conforme o número de tags:** Analisar o tempo utilizado pelos participantes em cada etapa do processo: sem *tags*, poucas *tags* e muitas *tags*.
- **número de tags novas adicionadas à tópicos fornecidos por programadores seniores:** avaliar se novas *tags* serão criadas ou utilizadas para fazer novas descobertas no código fonte a partir de *tags* existentes, analisando os números para cada situação.

6.1 Experimentos

Durante os experimentos foram observados e anotados como os programadores realizaram as manutenções propostas. Além da solução técnica proposta, também foram observadas as reações relacionadas ao comportamento de cada um, quando deparados com situações complexas e que exigia maior esforço e concentração.

As subseções seguintes apresentam, além destas observações, o resultado de cada experimento. Após a apresentação destes resultados, serão discutidos o que foi observado com as questões previamente levantadas no capítulo 3, correlacionando com os processos intrínsecos do sensemaking.

6.1.1 Experimento 1

No experimento 1, que foi realizado pelo programador SA e JA, não foi permitido utilizar nenhum recurso de tagueamento. A manutenção foi proposta para que os programadores a realizassem na maneira usual, com os recursos da própria IDE. O problema a ser solucionado correspondia à perda do foco do campo “desconto”, o qual deveria ser alterado.

Programador SA: o processo de manutenção, pelo programador SA, iniciou a partir da análise da execução da aplicação. O programador executou a aplicação, e conforme sua interação tentou entender o comportamento das ações executadas por ele em resposta aos eventos disparados pela aplicação a partir da ação do programador. O programador assimilou os eventos gerados pela aplicação, que no caso correspondia à perda do foco do campo “desconto”, que deveria ser alterado. O programador SA utilizou o método de depuração para auxiliar no entendimento do código fonte. A partir desta técnica, sua hipótese do evento selecionado anteriormente para realizar melhoria foi confirmada. Também foi a partir da característica do componente de digitação do desconto apresentado em tela que o programador conseguiu localizar o método. No entanto, apesar de ter encontrado o método, o SA não “confiou” em sua escolha, e começou a analisar o código fonte da classe que continha regras de tela, na busca por outros métodos mais sugestivos. Um dos métodos encontrados foi o “validarDesconto”, criado propositalmente para distrair a sua atenção, visto que este método, apesar de ter um nome sugestivo, não realiza a validação em si, além de ser um método que está implementado, mas não é utilizado pelo sistema. De fato a distração ocorreu, fazendo com que o programador gastasse tempo desnecessário tentando descobrir onde o referido método era chamado. Algum tempo depois o programador SA voltou à sua análise inicial, focando no método que tinha descoberto inicialmente. Consequentemente, descobriu o local exato do problema dentro do método. No entanto, o programador SA apresentou dificuldades em descobrir em que lugar (classe/método) originava o valor máximo de desconto permitido na digitação do pedido, no qual serve de parâmetro para realizar a validação solicitada neste experimento. O experimento foi executado com sucesso, em 16 minutos, sendo implementado no método e classe esperados.

Programador JA: o processo de manutenção, pelo programador JA, iniciou a partir da análise das estruturas dos pacotes e classes no projeto, com o objetivo de descobrir a lógica e padrão adotado na organização dos fontes. Sem utilizar recursos de busca de classes oferecidos pela IDE, o programador chegou à classe do pedido responsável em criar e organizar os componentes e comportamentos de eventos em tela. Em nenhum momento foi depurada a aplicação para auxiliar na compreensão do código fonte. Apenas buscas manuais, realizadas pelo próprio

usuário, e automáticas, aonde a própria IDE busca no arquivo fonte o termo informado. Mais de um terço do tempo foi aplicado na busca do atributo que representava o desconto máximo permitido. Praticamente todas as classes *beans* foram analisadas e estudadas. O programador descobriu um atributo que indicava o desconto máximo, mas não conseguiu deduzir a lógica de como este valor era calculado e lido. Esperava-se que o programador descobrisse o atributo que continha o desconto máximo, onde a mesma era valorizada no evento de perda de foco do campo “tabela de preço”, após informada pelo usuário na capa do pedido. Após 25 minutos corridos foi repassado ao programador esta variável e sua localização, indicando que ele deveria utilizá-la para aplicar a regra de validação do desconto. Os 5 minutos restantes foram gastos tentando achar o local da melhoria. Depois de muitas tentativas, a melhoria foi realizada no método “validarDesconto”, método que não é utilizado pelo sistema. O experimento ocorreu em 30 minutos, tempo máximo estipulado. No entanto, não foi concluído com êxito pelo programador JA. Apesar da realização correta da validação, o código não foi implementado no local válido, visto que o método alterado não fazia parte do sistema.

6.1.2 Experimento 2

Programador JB: a manutenção desenvolvida pelo programador JB iniciou a partir do desenvolvimento as tarefas descritas no experimento 2. Como ponto de partida iniciou a manutenção procurando no fonte pelas palavras “delete” e “remover”, para achar o local exato da melhoria. Gastou esforço tentando achar o método correto, pois no fonte haviam vários métodos que não estavam relacionados à solicitação de melhoria proposta no experimento, mas que tinham nomes indicativos semelhantes. Apesar de antes do início do experimento ter sido apresentada a ferramenta desenvolvida para taguear e auxiliar no processo de compreensão, o JB não utilizou o *plugin* como ferramenta de apoio. Em virtude desta escolha pelo JB, o mesmo apresentou dificuldade e muita insegurança em achar o local exato onde se esperava que a melhoria fosse implementada. Mais de 3 minutos foram gastos na análise da classe e nos fontes em geral. Após 15 minutos corridos, o programador apresentou características de cansaço mental e estresse. No intuito de continuar com os experimentos, foi passado ao programador deste experimento a dica do método correto onde deveria ser implementada a melhoria, devido ao fato de o tempo estimado ter atingido a metade do tempo estipulado e a não execução, até

então, da primeira etapa do experimento, a de localizar o local correto no código fonte. Tendo a localização exata, o programador iniciou a pesquisa do atributo onde continha a informação do status do pedido. Propositalmente o atributo para esta regra não foi desenvolvido de forma intuitiva. Esperava-se encontrar o atributo status na classe *bean* Pedido, o que de fato não existiu. Novamente, muito esforço foi gasto para descobrir o atributo correto. Após 22 minutos do início deste experimento o programador descobriu que o atributo responsável em armazenar o valor do estado do pedido era “sync”. No entanto, o programador não concretizou a melhoria proposta dentro do prazo de 30 minutos. O mesmo não conseguiu localizar o valor que caracterizava o código do status “enviado” para a realização da validação. Código que estava armazenado em uma constante dentro da própria aplicação, ou seja, o valor não era dinâmico, mas sim estático, e de fácil localização, além de sua constante possuir a nomenclatura indicativa. O programador JB, neste experimento, apresentou pouca lógica de programação e raciocínio lógico. Fator observado em suas variadas tentativas e erros em achar o local exato, ao invés de tentar estudar e compreender o fluxo do código dentro do fonte.

Programador SB: foi designado o programador SB a realizar a mesma tarefa executada por JB. Ao contrário do JB onde suas tentativas iniciais em localizar o método correto ocorreu pela procura semelhante à ação esperada, SB iniciou a compreensão executando a aplicação. A partir da execução da tela de consultas do pedido, descobriu-se com muita facilidade o campo de tela onde detalhava todos os *status* existentes no sistema. Abrindo o código fonte da tela de consulta, chegou com facilidade ao trecho do código onde o componente de lista era criado, e descobriu as constantes designadas a cada código do status do pedido. A partir dali, descobriu o nome da constante e seu valor para o *status* “enviado”. Notoriamente, após localizado o trecho, o programador SB iniciou o processo de criar *tags*. Criou-as com nomes sugestivos, caracterizadas como “ação + substantivo”, de forma não genéricas, como pode ser visto na Figura 6-1. A forma de como as *tags* foram criadas por um lado descrevia o que o trecho fazia, mas não permitia reutilizá-las em outros trechos. Em seguida, por meio da análise de chamadas e relacionamentos entre métodos e classes, descobriu que o atributo “sync” era o que caracterizava o *status* do pedido. Ao contrário do programador JB, o SB não usou a técnica de nomenclatura indicativa para achar o atributo. Sua localização ocorreu em virtude do

entendimento e estruturação do código fonte. O SB conseguiu realizar essa melhoria em 12 minutos, sendo que não precisou receber ajuda ou quaisquer tipos de conselhos. No entanto, nem o JB como o SB implementaram a melhoria no método e classe desejada, que correspondia à classe *Business Object* do pedido. A verificação em si para permitir remover ou não o pedido ocorreu no método remover implementado na classe destinada à camada de apresentação. Apesar de ser válido, não é o local indicado pela arquitetura adotada no projeto.

Figura 6-1 - Tags criadas por SB

6.1.3 Experimento 3

Programador JB: iniciou a manutenção utilizando massivamente as *tags* disponíveis, conforme visto na Figura 6-2. Assimilou-as com facilidade a relação entre “Preço” e “Pedido”. Como a alteração proposta estava diretamente relacionada a regra de desconto, o programador JB começou a navegar por todos os trechos de códigos que estavam marcados com a *tag* “Preço”. Por meio da *tag* deduziu com facilidade o atributo que guardava o valor do desconto máximo permitido para o item da tabela de preço do pedido. Após a localização, procurou no código fonte todos os lugares que chamavam o atributo em questão. Para cada trecho de código localizado, conferiu item a item com a *tag* que estava relacionada, observando

atentamente se o trecho que chamava o atributo também estava associado a *tag* “Preço”.

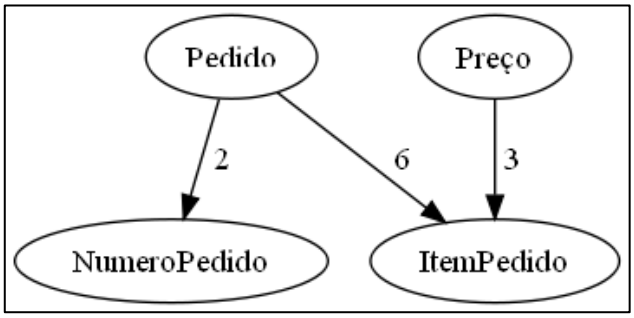

Figura 6-2 - *Tags* fornecidas aos programadores

Apesar de o local onde a melhoria deveria ser implementada não estar localizado em um método intuitivo e indicativo, o programador o achou com facilidade, estudando apenas qual seria o local mais indicativo dentro do método para implementar a condição de validação do desconto. Um fato curioso observado foi a criação, por parte de JB, da *tag* “Desconto” no método “validarDesconto” na classe de apresentação da tela do pedido, conforme visto na Figura 6-3. Este método, apesar de ter um nome sugestivo, não é utilizado por nenhuma parte do software. O mesmo foi criado com a intenção de confundir o programador, com objetivo de simular uma situação real. No entanto, apesar deste empecilho no caminho, este método não estava inicialmente tagueado. Devido a isto, quando guiado pelas *tags*, o programador não gastou energias em trechos desnecessários e irrelevantes.

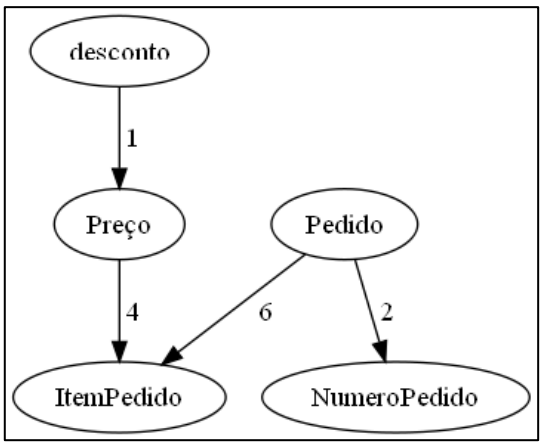

Figura 6-3 - Tag "desconto" criada no ambiente pelo programador JB

O experimento 3 foi executado por JB em apenas 8 minutos, sem precisar de ajuda ou qualquer outro tipo de suporte, comparado a SA, que sem auxílio das *tags*, executou a mesma manutenção no experimento 1 em 16 minutos.

Programador SB: o processo de manutenção iniciado pelo programador SB ocorreu da mesma forma que o programador JB neste mesmo experimento, a partir da análise das *tags* disponíveis. Avaliou as *tags* “Preço” e “ItemPedido”. Achou em menos de um minuto o local e a variável que deveria ser utilizada na lógica de validação. No entanto, gastou mais tempo para achar o local exato dentro do método, visto que o mesmo era complexo e extenso. Devido à facilidade oferecida pelas *tags*, o programador SB desta vez não iniciou o entendimento a partir da depuração da aplicação. Ao final, SB fez o seguinte comentário: “Ficou mais fácil achar o lugar da melhoria no código fonte”. No total foram gastos 4 minutos por parte de SB para a implementação da melhoria proposta no experimento 3, metade do tempo apresentado pelo programador JB.

6.1.4 Resultados obtidos

Para cada experimento realizado, foi analisado e cronometrado o tempo que cada programador precisou para executar as melhorias propostas. O Quadro 6-1 apresenta o total de tempo gasto por cada programador por experimento.

Quadro 6-1 - Resultados obtidos na execução dos experimentos

Programador	Experimento	Tempo (min.)
JA	1	30
JB	2	30
	3	8
SA	1	16
SB	2	12
	3	4

A partir dos tempos por programador, foi observada a relação de tempo por grupo de programadores, para a realização de uma mesma manutenção, quando disponíveis e utilizados recursos de tag, para quando não disponíveis, conforme demonstra o Quadro 6-2.

Quadro 6-2 – Comparativo entre o tempo da mesma manutenção com *tag* e sem *tag*

	Sem tags	Com tags
Grupo Júnior	30 minutos	8 minutos
Grupo Sênior	16 minutos	4 minutos

Outro fator importante observado foi a criação de novas *tags* nas circunstâncias onde não estavam disponíveis, apenas permitia-se criar novas *tags*, com a situação onde o código já estava tagueado e disponível ao programador. O Quadro 6-3 apresenta o número de novas tags criadas pelos programadores entre o experimento 2, onde não era disponibilizado nenhuma *tag*, e o experimento 3, onde as tags já estavam desenvolvidas e disponíveis.

Quadro 6-3 - Número de novas *tags* criadas por grupo

	Experimento 2 - Sem tags	Experimento 3 - Com tags
Grupo Júnior	0	1
Grupo Sênior	5	0

O Quadro 6-3 apresenta o número de *tags* entre dois tipos de experimentos, o experimento 2, considerado simples, e o 3, considerado complexo. Transformando estes números em tempo, com base nos dados já levantados, foi possível extrair e comparar o tempo de desenvolvimento de um item simples com um item complexo, conforme demonstra o Quadro 6-4.

Quadro 6-4- Comparativo item simples versus item complexo

	Simples Experimento 2	Complexo Experimento 3
Grupo Júnior	30 minutos	8 minutos
Grupo Sênior	12 minutos	4 minutos

6.1.5 Discussão

A seguir serão discutidos os pontos de avaliação destacados no quadros anteriores, percorrendo por pontos importantes dos experimentos. Em seguida será

realizado um breve comparativo dos resultados obtidos com os resultados dos estudos realizados por (SHARMA, 2011). Por último, as considerações do capítulo.

Nos experimentos 1 e 2 não estava disponível aos programadores recursos de tags para serem utilizadas no processo de compreensão. No entanto, no experimento 2 era possível criar novas *tags*, enquanto que isso não era possível no experimento 1. Já para o experimento 3 as *tags* foram disponibilizadas para auxiliar no processo de compreensão.

Conforme o Quadro 6-2, concluiu-se que de fato o desenvolvimento do sensemaking foi altamente influenciado conforme a disponibilidade dos recursos. O grupo de programadores juniores sem utilização das *tags* gastou em média 30 minutos para realizar a manutenção proposta. Já por meio das *tags*, este tempo caiu para 8 minutos, demonstrando um desempenho de produtividade de 74%.

Na mesma percepção, o grupo de seniores realizou a mesma manutenção em 12 minutos, enquanto que com o intermédio das *tags*, esse tempo caiu para 4 minutos, apresentando um ganho de 75%.

Um fato curioso foi observado durante a avaliação do número de *tags* criadas. No experimento 2, onde as *tags* não estavam disponíveis, mas foi oferecido a possibilidade de criá-las e utilizá-las, apenas o grupo de programadores seniores usufruiu desta possibilidade. No entanto, as *tags* criadas foram utilizadas como *waypoints* e tópicos de memorização extraídos do código fonte. *Waypoints* são utilizados para salvar locais de interesse por meio de sistemas de posicionamento geográfico, servido como pontos de referência para orientar a navegação (STOREY et al, 2006). Desta forma, as *tags* criadas auxiliaram na navegação do código fonte, auxiliando os programadores a se localizarem entre as inúmeras classes e métodos.

Em contrapartida, no ambiente onde as *tags* já estavam desenvolvidas e disponíveis, apenas o grupo de juniores adicionou uma nova *tag*, com o mesmo objetivo do outro grupo, a utilização como *waypoint*.

Conclui-se que em ambientes desconhecidos os programadores mais experientes possuem mais facilidade em extrair o conhecimento do código fonte, devido ao fato deles apresentarem maior experiência. Também foi observado que em ambientes onde o conhecimento do código já estava presente, os seniores não processaram novos conhecimentos, enquanto que os juniores foram conduzidos pelas tags existentes e adicionaram uma nova *tag* relacionada. O fato de não processarem novos conhecimentos coloca em prova a conclusão do estudo

realizado por (RAMAL et al, 2002), que demonstra que não há interesse por parte dos engenheiros de software em estudar o conhecimento do domínio da aplicação na realização de manutenções pontuais, onde apenas os conhecimentos relacionados à engenharia de software (programação, ambiente de desenvolvimento e implementação da aplicação) são consideradas. (RAMAL et al, 2002) concluiu em seu estudo que os programadores cultivam seus conhecimentos passados, e procurar buscar novos conhecimentos é um processo custoso, que somente é realizado quando há uma necessidade clara para o programador e que não exista um caminho mais fácil previsto. Para Singer apud (RAMAL et al, 2002), engenheiros de softwares tentam entender somente o necessário de um sistema para resolver o problema corrente e depois tendem a esquecer os detalhes do que aprenderam.

Os programadores seniores no experimento 2 apresentaram uma média de desempenho 60% superior (Quadro 6-4) no mesmo experimento realizado pelo grupo de juniores. Neste experimento, apenas os seniores utilizaram o recurso de extração do conhecimento do código fonte, o que justifica que o sensemaking é melhor desenvolvido quando já se tem bases e experiências passadas (WEICK et al, 2005).

No entanto, como já discutido, em um ambiente onde o conhecimento contido no código fonte já esteja previamente extraído por um especialista de maior conhecimento e disponibilizado, via *tags*, para aqueles que possuem menos experiência, demonstrou um ganho significativo de desempenho. Acessando os conhecimentos extraídos do código fonte apenas pelos juniores, os mesmos conseguiram realizar a mesma melhoria, comparada com o grupo de seniores, em 33,34% de tempo a menos. Desta forma, concluiu-se que o método proposto para extração e compartilhamento de conhecimento do código fonte apresentou-se eficaz o suficiente para melhorar o desempenho geral da equipe de desenvolvimento. No entanto, é preciso realizar mais experimentos com um grupo maior de programadores, com o objetivo de averiguar o quão melhor este método apresenta ser.

Um experimento semelhante intitulado “*Role of available and provided resources in sensemaking*” realizado por Nikhil Sharma (SHARMA, 2011) avaliou a forma e o comportamento das pessoas durante a realização de sensemaking. O experimento foi conduzido com alunos de diversos cursos de uma universidade, na qual os alunos foram convidados a montar a estrutura do sensemaking para dois

tópicos: chá e bebidas para terceira idade. Conforme estudos do autor, tanto recursos disponíveis quanto recursos oferecidos afetam as atividades de sensemaking. Pontos levantados pela pesquisa demonstraram que a forma de como os recursos são disponibilizados, desde oferecido por outros profissionais, quanto acessados na Internet para desenvolver e aperfeiçoar uma estrutura/tema, influenciaram diretamente no processo de *sensemaking*. Artefatos antigos e maduros fornecidos foram classificados e melhor utilizados, porém resultou em menor estrutura e informação, enquanto os não fornecidos foram mais estruturados. O Quadro 6-5 apresenta as questões levantadas no experimento de (SHARMA, 2011), com seus respectivos resultados comparativamente em relação aos resultados obtidos com os experimentos desta pesquisa.

Quadro 6-5 - Comparativo do experimento de Sharma (SHARMA, 2011) com os obtidos

Questão	Resultado obtido na pesquisa de Sharma	Resultado obtido
Atividades do sensemaking irão variar de acordo com os recursos disponíveis	Tópicos mais fáceis de entender e que são mais comuns são trabalhados muito mais intensamente do que aqueles incomuns.	Conforme avaliado, as tags que expressavam uma melhor lógica e coerência com a manutenção solicitada aos programadores foram mais acessadas e avaliadas, comparadas àquelas que não estavam totalmente relacionadas ao problema.
Atividades de sensemaking variam conforme a presença de artefatos fornecidos	Sim, no entanto participantes que tinham acesso a artefatos fornecidos acabaram adicionando poucas informações, seções e bookmarks.	Em ambientes já tageados, novas tags não foram criadas. Em resumo, os programadores tentaram reaproveitar as tags existentes.
Utilização de artefatos fornecidos variam na presença de estruturas facilmente disponíveis	As pessoas utilizaram em maior quantidade marcadores (bookmark) fornecidos para a tarefa de bebidas (menos conhecidas) comparado com a tarefa de chá (mais conhecida).	Nas melhorias complexas os programadores usufruíram das tags já existentes, sendo utilizadas como guias para compreender o código fonte.
Atividades de sensemaking e utilização de artefatos variam com base na maturidade dos artefatos fornecidos	Aqueles artefatos que foram fornecidos em um estágio posterior foram adicionados menos número de pastas em seus esboços e marcações (*outline* e *bookmark*).	Quando as tags já estavam presentes no ambiente, desenvolvidas por outro programador experiente, foram adicionadas em um número expressivamente menor, comparado nos experimentos onde não haviam tags disponíveis.

Como a pesquisa realizada de Sharma avaliou o desenvolvimento do *sensemaking* nas pessoas, seu trabalho foi comparado com este, com o objetivo de analisar os resultados e suportar suas afirmações, mas no ambiente de desenvolvimento de software.

Sua pesquisa, assim como os resultados, auxiliaram a suportar o desenvolvimento deste trabalho. Desta forma, foi possível, a partir de seu trabalho, trazer para o ambiente de engenharia de software seus resultados, e provar que o que foi realizado em seus experimentos, também se aplicam ao processo de manutenção de software, conforme visto no resumo de ambos os trabalhos descritos no Quadro 6-5.

6.2 Considerações sobre o capítulo

Este capítulo discutiu e apresentou os experimentos realizados com um grupo de programadores com objetivo de avaliar o método proposto pelo trabalho, assim como os resultados obtidos e as discussões.

Foram comparados os resultados obtidos entre os programadores de mesmo grupo e de grupos diferentes. Os resultados foram analisados com base em três experimentos. Como principal resultado, o experimento concluiu que é possível melhorar o desempenho dos programadores juniores, além de aperfeiçoar as técnicas de programação já dominadas pelos programadores seniores.

.

CAPÍTULO 7 - CONSIDERAÇÕES FINAIS

Este capítulo finaliza o projeto de pesquisa, destacando os principais pontos levantados, a relevância do estudo, suas contribuições e sugestões, assim como as perspectivas para trabalhos futuros.

7.1 Relevância do estudo

Os números apresentados no capítulo 1 e 2 demonstram a deficiência na área de manutenção de software, com ênfase especificamente no processo de compreensão de código fonte. Vários estudos foram desenvolvidos para tentar amenizar esta situação, porém, conforme os resultados apresentados, ainda ‘e possível estudar e melhorar este processo.

Dentre as atividades de manutenção, a compreensão é a mais abrangente e importante, pois envolve maior esforço cognitivo das pessoas. A compreensão está ligada a recuperação de informações a partir do código fonte. Estas informações são documentos do projeto, ricos em detalhes que ajudam a compreender o código fonte, é o conhecimento extraído a partir do código fonte.

Apesar de alguns destes artefatos serem possíveis de obter a partir de uma ferramenta *case*, por meio da engenharia reversa diretamente aplicada ao código fonte, o documento final não é tão rico quanto aquele obtido por meio de extração do conhecimento a partir de outros métodos.

Um exemplo de extração do conhecimento é aplicar a engenharia reversa através de diagramas de classe (UML) e de banco de dados (ER). O problema nestas duas abordagens é o foco. Ambas estão direcionadas para a comunicação e a estruturação lógica e física dos componentes de software, respondendo “o que”, e ignorando “o por que”, que só é possível responder incorporando informações de domínio no processo de compreensão (KELLER et al, 1999). Pelo domínio da ontologia é possível recuperar mais conceitos que no diagrama de classe UML, facilitando o entendimento do domínio da aplicação pelos programadores, devido as especificações de associação e dependências. Além do mais, um diagrama de

classes de código não introduzirá nenhum conhecimento inferido, acarretando na falta do contexto e no conhecimento explícito do domínio (ZHOU et al, 2008).

O conhecimento extraído diretamente do código fonte, por meio do processo de sensemaking, é rico em detalhes importantes e valiosos para aplicar na compreensão do código fonte. Este conhecimento pode ser melhor aproveitado quando armazenado, através de ontologias, e disseminado a mais pessoas, através de técnicas da Web Semântica.

Com base na pesquisa realizada, como mostram os resultados do Apêndice B, 70% dos entrevistados concordaram que ter um repasse pessoal da pessoa que criou o código fonte facilita a manutenção, e 68% afirmaram que tendo um repasse pessoal de alguém que já tem conhecimento do código fonte também auxilia na manutenção. Desta forma, conclui-se que não apenas extrair o conhecimento, mas compartilhar com demais pessoas envolvidas, beneficia a equipe e desempenho de trabalho.

Os resultados dos experimentos aplicados na extração do conhecimento no código fonte demonstraram um ganho de 33,34% de desempenho em produtividade, por parte dos programadores juniores, quando o conhecimento é compartilhado e utilizado. Desta forma, foi possível alavancar o processo de compreensão de um programador júnior, fazendo com que este consiga realizar manutenções de qualidade semelhantes a um programador sênior em um tempo inferior. Também, como consequência, os programadores seniores também se beneficiaram com este método, visto que o mesmo se aplica a ambos os grupos de programadores.

Desta forma, é possível concluir que a extração do conhecimento, seu processamento de forma adequada e o compartilhamento, auxiliam positivamente no processo de manutenção e compreensão do código fonte, trazendo benefícios como diminuição do tempo dispendido, qualidade e maior segurança nas alterações realizadas, por guiar o programador no local exato da melhoria, fazendo com que a manutenção não ocorra em lugares errôneos que poderiam afetar a qualidade do programa, além de abrir possibilidades de brechas de seguranças. Sendo assim, a questão principal desta pesquisa pode ser respondida, é possível diminuir o tempo de esforço para a compreensão do código fonte durante a sua manutenção.

7.2 Contribuições da pesquisa

Com base nos estudos e experimentos realizados durante a execução deste trabalho, as seguintes contribuições foram identificadas:

- identificação de técnicas de extração do conhecimento a partir do código fonte. Armazenamento, processamento e disseminação do conhecimento por meio da utilização de ontologias.
- identificação de ontologias para apoiar os processos da folksonomia e compartilhamento dos conhecimentos identificados na engenharia de software.
- método para permitir auxiliar os programadores na manutenção de códigos fonte desconhecidos, com o propósito de ajudar na compreensão a partir da extração e do armazenamento do conhecimento.
- aplicação do sensemaking no contexto da engenharia de software, nos processos que envolvem a compreensão do código fonte.
- percepção dos programadores em relação a códigos fonte desconhecidos. Avaliação de suas capacidades psicológicas para tratamento de questões desconhecidas, e a forma de organização e desenvolvimento do conhecimento.
- ambiente para extração e compartilhamento de conhecimento a partir do código fonte.

7.3 Limitações da pesquisa

A pesquisa realizada com profissionais mostrou-se satisfatória, com a participação de 119 programadores. Mesmo apresentando um resultado que justifica a importância deste trabalho em meio ao mercado, e apoiando o resultado dos experimentos realizados, o total de participantes nos experimentos práticos não foi tão expressivo. Novas conclusões sobre a compreensão poderiam ser tiradas se o número de participantes nos experimentos fosse maior. Também seria interessante avaliar a reação dos programadores com formações distintas, além de avaliar a questão do impacto da cultura e costumes pessoais e organizacionais.

Outra limitação apresentada foi a implementação do *plugin*. Como o objetivo do trabalho era estudar e avaliar as questões de compreensão, o produto gerado a

partir deste estudo, que corresponde ao *plugin* desenvolvido para a IDE Eclipse, não foi implementado da melhor forma possível. Sua usabilidade acabou comprometida, o que poderia ajudar os programadores a entender mais rapidamente sua utilização e seus benefícios.

Por fim, para possibilitar a execução dos experimentos dentro do ambiente de trabalho, os experimentos foram limitados ao tempo máximo de 30 minutos para cada execução. Com isso, os resultados finais refletiram o tempo máximo estipulado, e não o tempo total real levado pelos envolvidos nos experimentos.

7.4 Trabalhos futuros

Como continuidade deste trabalho, as seguintes sugestões são apresentadas:

- desenvolver técnicas de extração e integração do conhecimento com outros artefatos de software, como a documentação técnica e de negócio. Aplicar conceitos de *sensemaking* e folksonomia nestes documentos, expandido a base de conhecimento para além dos limites específicos do código fonte;
- expandir o método proposto para novos desenvolvimentos de software. Utilizar os conceitos aplicados de extração, disseminação e compartilhamento do conhecimento em todas as etapas do ciclo de vida do software, e não apenas na manutenção. Desta forma, o método proposto poderia ser utilizado também para, além da criação de um novo software, para seu gerenciamento e manutenções futuras, além de integrar e auxiliar na realização de testes automatizados mais elaborados e, de certa forma, mais inteligentes;
- unir técnicas de extração do conhecimento automatizadas e guiadas por agentes, juntamente com o processo manual e conciso do *sensemaking*, com objetivo de melhorar o conhecimento extraído, além de enriquecer a semântica e as informações contidas nos artefatos de software;
- aperfeiçoar o *plugin*, tornando-o mais *user-friendly* para extração e busca de *tags* dentro do código fonte. Tornar o ambiente desenvolvido em um produto mais dinâmico, e disponibilizar o projeto para a comunidade.

REFERÊNCIAS BIBLIOGRÁFICAS

(AGARWAL; TAYAL, 2007) ARGAWAL, B. B.; TAYAL, S. P. **Software Engineering.** Laxmi Publications: New Delhi, 2007.

(AL-KHALIFA, DAVIS, 2007) AL-KHALIFA, H,; DAVIS H. **Exploring the value of folksonomies for creating semantic metadata.** In International Journal on Semantic Web &Information Systems, n. 3 v. 1, p. 13–39, 2007.

(ALLEMANG; HENDLER, 2008) ALLEMANG, D.; HENDLER, J. **Semantic Web for the Working Ontologist. Effective Modeling in RDFS and OWL.** Elsevier: Estados Unidos, 2008, 330 p.

(ALNUSAIR, 2010) ALNUSAIR, A. **SCRO – Source-code Ontology.** Disponível em <http://www.indiana.edu/~awny/index.php/research/projects-tools/15-research/ontologies/10>. Acesso em 22 fev. 2012.

(ANDRADE, 2009) ANDRADE, M. M. **Introdução a metodologia do trabalho científico**. Atlas: São Paulo, 2009, 162 p.

(ANTONIOU; HARMELEN, 2008) ANTONIOU, G.; HARMELEN, F. **Semantic Web primer.** Massachusetts Institute of Technology: Estados Unidos, ed. 2, 2008, 264 p.

(APRIL et al, 2005) APRIL, A., HAYES, J.H., ABRAN, A., DUMKE, R.R. **Software Maintenance Maturity Model (SMmm): the software maintenance process model**. In Proceedings of Journal of Software Maintenance, p. 197-223, 2005.

(APRIL et al, 2008) APRIL, A., DESHARNAIS, J., DUMKE, R. **A formalism of ontology to support a software maintenance knowledge-based system.** In Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering Conference (SEKE06), p. 331 – 336, 2006.

(ARGYRYS; SCHÖN, 1996) ARGYRYS, C.; SHCÖN, D. **Organizational Learning II: Theory, Method and Practice.** Addison-Wesley: Estados Unidos, 1996.

(AUER, IVES, 2007) AUER, S.; IVES, Z. **Integrating Ontologies and Relational Data.** Technical report, , Universidade da Pensilvânia, Estados Unidos, 2007.

(BADAREEN et al., 2011) BADAREEN, A. B. AL.; SELAMAT, M. H.; JABAR, M. A.; DIN, J.; TURAEV, S. **The Impact of Software Quality on Maintenance Process. International Journal of Computers.** International Journal of Computers, Issue 1, ACM, v. 5, p. 191 – 1999, 2011.

(BASILI, 1980) BASILI, V. R. **Quantitative software complexity models: A panel summary.** In Basili VR (ed) tutorial on models and methods for software management and engineering, IEEE, 1980.

(BENNER, 1994) BENNER, P. **The role of articulation in understanding practices and experience as sources of knowledge in clinical nursing.** Tully, ed. Philosophy In An Age of Pluralism, Cambridge University Press: Nova Iorque, p. 136-155, 1994.

(BERNERS-LEE et al, 2001) BERNERS-LEE, T., HENDLER, J., LASSILA, O. **The Semantic Web. Scientific American Magazine.** Disponível em <http://www.sciam.com/article.cfm?id=the-semantic-web&print=true>. Acesso em 16 nov. 2011.

(BERNERS-LEE, 1999) BERNERS-LEE, T. **Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.** Publishers Inc: Nova Iorque, 1999.

(BJORNSON, DINGSOYR, 2008) BJORNSON, T., DINGSOYR, T. **Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used.** In Information and Software Technology, Elsevier, p. 1055 – 1068, 2008.

(BOOCH et al., 1999) BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The unified modeling language user guide.** Massachusetts: Addison Wesley, 1999. 482 p.

(BORST, 1997) BORST W. N. **Construction of Engineering Ontologies.** Technical Report, University of Tweenty: Holanda, 1997.

(BRUIJIN et al, 2006) BRUIJIN, J.; EHRIG, M.; FEIER, C.; FEIER, C.; RECUERDA, F. M.; SCHARFFE, F.; WEITEN, M. **Ontology mediation, merging, and alining.** In Semantic Web Technologies: Trends and Research in Ontology-based Systems, John Wiley & Sons, 2006

(CATTUTO et al, 2007) CATTUTO, C.; SCHMITZ, C.; BALDASSARRI, A.; SERVEDIDO, V. D. P.; LORETO, V.; HOTHO, A.; GRAHL, M.; STUMME, G. **Network Properties of Folksonomies**. AI Communications v. 4, n. 20, p. 245 – 262, 2007.

(CERVO, BERVIAN, 1996) CERVO, A L., BERVIAN, P.A **Metodologia científica**. Makron Books: São Paulo, 4 ed., 1996, 90 p.

(CHEN, 2010) CHEN, W.; CAI, Y.; LEUNG, H.; LI, Q. **Generating ontologies with basic level concepts from folksonomies.** In International Conference on Computational Science, ICCS, v. 1, n. 1, p. 573 – 581, 2010.

(CHIA, 2000) CHIA, R. **Discourse analysis as organizational analysis. Organization.** Organization, v. 7 n. 3, p. 513-518, 2000.

(CIRIBELLI, 2003) CIRIBELLI, M. C. **Como Elaborar uma Dissertação de Mestrado através da pesquisa científica**. 7Letras:Rio de Janeiro, 2003, 227 p.

(COCCO et al., 2003) COCCO, G.; GALVÃO, A. P.; SILVA, G. **Capitalismo cognitivo: trabalho, redes e inovação.** Rio de Janeiro: DPA, 2003.

(CORBI, 1989) CORBI, T. **Program understanding: challenge for the 1990s.** IBM Sytem Journal, 28(2), p. 294 – 306, 1989.

(CORCHO et al, 2003) CORCHO, O.; LOPEZ, M. F.; PEREZ, A. G. **Methodologies, tools and languages for building ontologies. Where is their meeting point?** Data & Knowledge Engineering, v. 46, n. 1, p. 41-64, 2003.

(CORRITORE; WIEDENBECK, 1998) CORRITORE, C. L.; WIEDENBECK, S. **Mental representations of expert procedural andobject-oriented programmers in a softwaremaintenance task.** In. J. Hum.-Comput. Stud, p..61-83, 1998.

(CORRITORE; WIEDENBECK, 1999) CORRITORE, C. L.; WIEDENBECK, S. **Mental representations of expert procedural and object-oriented programmers in a software maintenance task.** In International Journal of Human-Computer Studies, v. 50, n. 1, p. 61 – 83, 1992.

(DACONTA; OBRST; SMITH, 2003) DACONTA, M. C.; OBRST, L. J.; SMITH, K. T. **The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management.** John Wiley & Sons: Indianapolis, Estados Unidos, 2003

(DASGUPTA, 2010) DASGUPTA, C. **That is Not My Program Investigating the Relation between Program Comprehension and Program Authorship.** In ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference, ACM, n. 103, 2010.

(DAVENPORT; PRUSAK, 1998) DAVENPORT, T..; PRUSAK, L.. **Working Knowledge: How Organizations Manage What They Know**. Harvard Business School Press: Boston, 1998.

(DEKLEVA, 1992) DEKLEVA, S. M. **Delphi Study of Software Maintenance Problems.** In Conference on Software Maintenance, IEEE, p. 10 – 17, 1992.

(DERIVO, 2012) DERIVO, Semantic Systems.: **SPARQL-DL API** Disponível em <http://www.derivo.de/en/resources/sparql-dl-api/>. Acesso em: 31 mai. 2012.

(DERVIN, 1992) DERVIN, B. **From the mind's eye of the user: The sense-making qualitative-quantitative methodology.** In Qualitative research in information management, Englewood, p. 61 – 84, 1992.

(DOTSIKA, 2009) DOTSIKA, F. **Uniting formal and informal descriptive power: Reconciling ontologies with folksonomies**. In International Journal of Information Management, v. 9, n. 5, p. 407-415, 2009.

(ECHARTE et al, 2007) ECHARTE, F.; ASTRAIN, J. J.; CÓRDOBA, A.; VILADANGOS, J. **Ontology of Folksonomy: A New Modeling Method.** In Semantic Authoring, Annotation and Knowledge Markup, Whistler: Canada, 2007

(ECLIPSE, 2012) Eclipse: **ID de Desenvolvimento**. Disponível em <http://www.eclipse.org>. Acesso em: 02 jun. 2012.

(E-CONSULTING CORP, 2004) E-CONSULTING CORP. **A gestão do conhecimento na prática.** HSM Management 42, Janeiro-Fevereiro. p. 53-59, 2004.

(ERDOGMUS; TANIR, 2002) ERDOGMUS, H.; TANIR, O. **Advances in Software Engineering: comprehension, evaluation, and evolution.** Springer: Nova Iorque, 2002, 495 p.

(ERLIKH, 2000) ERLIKH, L. **Leveraging legacy system dollars for E-business.** IT Professional, IEEE, v. 2, n. 3, p. 17 – 23, 2000.

(FESTINGER, 1957) FESTINGER, L. **A theory of cognitive dissonance.** Stanford University Press: Estados Unidos, 1957.

(GALL; REIF, 2008) GALL, H.; REIF, G. **Semantic Web Technologies in Software Engineering.** In: 30th International Conference on Software Engineering (ICSE 2008), Universidade de Zurich, Alemanha, 2008.

(GANAPATHY, SAGAYARAJ, 2011) GANAPATHY, G.; SAGAYARAJ, S. **To Generate the Ontology from Java Source Code**. In International Journal of Advanced Computer Science and Applications, v. 2, n. 2, 2011.

(GENESERETH, NILSSON, 1987) GENESERETH, M. R., NILSSON, N. J. **Logical Foundation of Artificial Intelligence.** Morgan Kaufmann Publisher Inc: Califórna, Estados Unidos, 1987.

(GIL, 2002) GIL, A. C. **Como elaborar projetos de pesquisa.** Atlas: São Paulo, 4 ed., 2006. 175 p.

(GRAU et al, 2004) GRAU, B. C.; PARSIA, B.; SIRIN, E. **Working with multiple ontologies on the semantic web**. In International Semantic Web Conference, Springer, v. 3298, p. 620 – 234, 2004.

(GRUBER, 1993) GRUBER, T. R. **Toward Principles for the Design of Ontologies Used for Knowledge Sharing.** International Journal of Human and Computer Studies, ACM, v. 43, n. 5-6, p. 907-928, 1993.

(GRUBER, 2007) GRUBER, T. **Ontology of Folksonomy: A Mash-up of Apples and Oranges.** Disponível em <http://tomgruber.org/writing/ontology-of-folksonomy.htm>. Acesso em 27 mar. 2012.

(GUARINO, 1998) GUARINO, N. **Formal Ontology and Information Systems.** In the Proceedings of Formal Ontology in Information Systems, p. 3-15, 1998.

(HALPIN et al, 2006) HALPIN, H.; ROBU, V.; SHEPARD, H. **The Dynamics and Semantics of Collaborative Tagging.** In Proceedings of the 1st Semantic Authoring and Annotation Workshop (SAAW06), Georgia, Estados Unidos, 2006.

(HAMMOND et al, 2005) HAMMOND, T., HANNAY,T., LUND, B., SCOTT, J. **Social Bookmarking Tools (I): A General Review**. Disponível em < http://www.dlib.org/dlib/april05/hammond/04hammond.html> Acesso em 16 nov. 2011.

(HAZZAN, 2003) HAZZAN, O. **Cognitive and social aspects of software engineering: a course framework.** In Proceeding ITiCSE '03 Proceedings of the 8th annual conference on Innovation and technology in computer science education, ACM, p. 3 – 6, 2003.

(HOTHO et al, 2006) HOTHO, A., JÄSCHKE, R., SCHMITZ, C., STUMME, G. **Information retrieval in folksonomies: Search and ranking.** In: Sure, Y. and Domingue, J., editors, The Semantic Web: Research and Applications, Springer Berlin, v. 4011 cap. 31, p. 411-426, 2006.

(HSIEH et al, 2008) HSIEH, W.; STU, J.; CHEN, Y.; CHOU, S. T. **A collaborative desktop tagging system for group knowledge management based on concept space.** In Expert Systems with Application, Elsevier, v. 36, I 5, p. 9513 – 9523, 2009.

(HUFF, 1990) HUFF, S. **Information systems maintenance.** The Business Quarterly 55, p. 30-32, 1990.

(HYPERDICTIONARY, 2011) **Dicionário online.** Disponível em <http://www.hyperdictionary.com/>. Acesso em 14 nov. 2011.

(IEEE, 2004) IEEE Computer Society. **Guide to The Software Engineering Body of Knowledge - SWEBOOK**. Los Alamitos: CS Press, 2004, 204 p.

(JÄSCHKE et al, 2008) JÄSCHKE, R., HOTHO, A., SCHMITZ, C., GANTER, B. STUMME, G. **Discovering shared conceptualizations in folksonomies.** In: Web Semantics: Science, Services and Agents on the World Wide Web, v. 6, n. 1, p. 38-53, 2008.

(KAHLMEYER-MERTENS et al, 2007) KAHLMEYER-MERTENS, R. S.; FUMANGA, M.; TOFFANO, C. B.; SIQUEIRA, F. **Como elaborar projetos de pesquisa.** Editora FVG: Rio de Janeiro, 1 ed. , 2007. 140 p.

(KELLER et al, 1999) KELLER, R. K.; SCHAUER, R.; ROBITAILLES, S.; PAGE, P. **Pattern-based Reverse Engineering of Design Components**. In Proc. 21 Int'l Conf. Software Engineering, California, Estados Unidos, p. 226 – 235, 1999.

(KIM et al, 2008) KIM, H. L.; PASSANT, A.; BRESLIN, J. G.; SCERRI, S.; DECKER, S. **Review and Alignment of Tag Ontologies for Semantically-Linked Data in Collaborative Tagging Spaces.** In Proceeding of the 2nd International Conference on Semantic Computing, IEEE, p. 315 – 322, 2008.

(KIM et al, 2008) KIM, H. L.; SCERRI, S.; BRESLIN, J. G.; DECKER, S.; KIM, H. G. **The State of the Art in Tag Ontologies: A Semantic Model for Tagging and Folksonomies.** In Proceeding of the Int. Conference on Dublin Core and Metadata Application, Berlin, Alemanha, 2008.

(KNERR, 2006) KNERR, T. **Tagging ontology- towards a common ontology for folksonomies**. Disponível em <http://tagont.googlecode.com/files/TagOntPaper.pdf>. Acesso em 27 mar. 2012.

(KOLB, 1984) KOLB, D. **Experiential Learning Experience as a Source of Learning and Development.** Prentice Hall: Englewood Cliffs, Estados Unidos, 1984.

(KROSKI, 2005) KROSKI, E. **The Hive Mind: Folksonomies and User-Based Tagging.** Disponível em <http://infotangle.blogsome.com/category/folksonomies>. Acesso em 10 nov. 2011.

(LACY, 2005) LACY, L. W. **OWL: Representing Information Using the Web Ontology Language.** Trafford Publishing: Victoria, Canada, 2005, 284 p.

(LEAHY, 2012) LEAHY, P. **Using Java Naming Conventions**. Disponível em <http://java.about.com/od/javasyntax/a/nameconventions.htm/>. Acesso em: 14 jul. 2012.

(LETOVSKY, 1986) LETOVSKY, S. **Cognitive Processes in Program Comprehension**. In Empirical Studies of Programmers, Ablex Publishing, p. 58 – 79, 1986.

(LIENTZ; SWANSON, 1980) LIENTZ, B. P.; SWANSON, E. **Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organization.** Addison-Wesley: Estados Unidos, 1980, , 214 p.

(LIENTZ; SWANSON, 1980) LIENTZ, B. P.; SWANSON, E.B. **Software Maintenance Management.** Los Angeles: Addisson-Wesley Publishing Company, 1980.

(LIMPENZ et al, 2008) LIMPENZ, F., GANDON, F., BUFFA, M. **Bridging Ontologies and Folksonomies to Leverage Knowledge Sharing on the Social Web: a Brief Survey.** In Proc. 1st International Workshop on Social Software Engineering and Applications (SoSEA), 2008.

(LUCIA et al., 1996) LUCIA, A. De.; FASOLINO, A.R.; MUNRO, M. **Understanding Function Behaviours Through Program Slicing**. In 4th IEEE Workshop on Program Comprehension, IEEE, p. 9-18, 1996.

(MAGALA, 1997) MAGALA, S. J. **The making and unmaking of sense.** Organization Studies, v. 2, n.18, p. 317-338, 1997.

(MATHES, 2004) MATHES, A. **Folksonomies - Cooperative Classification and Communication Through Shared Metadata.** Disponível em <http://www.adammathes.com/academic/computer-mediated-communication/folksonomies.html>. Acesso em 10 nov. 2011.

(MATHIAS et al, 1999) MATHIAS, K, S., CROSS, J. H., DEAN T. H., Barowski, L. A. **The role of software measures and metrics in studies of program comprehension.** In ACM Southeast Regional Conference, ACM, 1999.

(MAYRHAUSER; VANS, 1995) MAYRHAUSER, A.; VANS, A. **Program Comprehension During Software Maintenance and Evolution**. IEEE Computer 28, IEEE, p. 44-55, 1995.

(MCKEE, 1984) MCKEE, J. **Maintenance as a function of design.** Proceedings of the AFIPS National Computer Conference, ACM, p. 187-193, 1984.

(MENG et al., 2006) MENG, Wen.; RILLING, Juergen.; ZHANG, Yonggang.; WITTE, René.; MUDUR, Sudhir.; CHARLAND, Philippe. **A Context-Driven Software Comprehension Process Model.** IEEE Software Evolvability Workshop, 2006.

(MENTZAS et al., 2002) MENTZAS, G.; APOSTOLOU, D.; EBECKER, A.; YOUNG, R. **Knowledge Asset Management – Beyond the Process-centered and Product-centered Approaches.** Springer: Londres, 2002.

(MENTZAS et al., 2003) MENTZAS, G.; APOSTOLOU, D.; EBECKER, A.; YOUNG, R. **Knowledge Asset Management – Beyond the Process-centered and Product-centered Approaches.** Springer-Verlag, 2003.

(MERRIAM-WEBSTER, 2011) **Dicionário online.** Disponível em <www.m-w.com>. Acesso em 16 nov. 2011.

(MIKA, 2005) MIKA, P. **Ontologies Are Us: A Unified Model of Social Networks and Semantics**. In International Semantic Web Conference, Springer, p. 522-536.

(MOAD, 1990) MAOD, J. **Maintaining the competitive edge.** Datamation 61-62, 64, 66.

(NEWMAN, 2005) NEWMAN, R. **Tag ontology design**. Disponível em <http://www.holygoat.co.uk/projects/tags/>. Acesso em 27 mar. 2012.

(NONAKA; TAKEUCHI, 1995) NONAKA, I.; TAKEUCHI, H. **The Knowledge Creating Company.** Oxford University Press: Japão, 1995.

(NOY, STUCKENSCHMIDT, 2005) NOY, N. F.; STUCKENSCHMIDT, H. **Ontology Alignment: An annotated Bibliography.** In Semantic Interoperability and Integration**.** Schloss Dagstuhl, Alemanha, 2005.

(OBSTFELD, 2004) OBSTFELD, D. **Saying more and less of what we know: The social processes of knowledge creation, innovation, and agence.** Universidade da Califórnia, 2004.

(OLIVEIRA, 2004) OLIVEIRA, L. R. M. **A comunicação educativa em ambientes virtuais: um modelo de *design* de dispositivos para o ensino-aprendizagem na universidade.** Universidade do Minho: Braga, 2004, 68 p.

(OWLAPI, 2012) OWLAPI: **Implementação Java para a Web Ontology Language (OWL)**. Disponível em <http://semanticweb.org/wiki/OWL_API>. Acesso em: 31 mai. 2012.

(PAGET, 1988) PAGET, M. A. **The Unity of Mistakes.** Temple University Press: Philadelphia, Estados Unidos, 1988.

(PASSANT, 2007) PASSANT, A. **Using Ontologies to Strengthen Folksonomies and Enrich Information Retrieval in Weblogs: Theoretical background and corporate use-case.** In International Conference on Weblogs and Social Media, Boulder, Estados Unidos, 2007.

(PELLET, 2012) Pellet: **OWL 2 Reasoner for Java**. Disponível em <http://clarkparsia.com/pellet>. Acesso em: 02 jun. 2012.

(PENNINGTON, 1987) PENNINGTON, N. **Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs.** Cognitive Psychology, v. 19, p. 295 - 341, 1987.

(PEREIRA, 2006) PEREIRA, H. C. **Folksonomia e a maneira com que nós colocamos ordem nas coisas.** Disponível em <http://revolucao.etc.br/archives/folksonomia-e-a-maneira-com-que-nos-colocamos-ordem-nas-coisas/>. Acesso em 10 nov. 2011.

(PIGOSKI, 1997) PIGOSKI, T. M. **Practical software maintenance: Best practice for managing your software investment.** John Wiley & Sons: Nova Iorque, Estados Unidos, 1997, 384 p.

(POLLOCK, 2009) POLLOCK, J. T. **Semantic Web for Dummies.** Wiley Publishing: Indianapolis, Estados Unitos, 2009. 412 p.

(PORT, 1988) PORT, O. **The software trap – automate or else.** Business Week 3051 (9), p. 142-154, 1988.

(POWERS, 2003) POWERS, S. **Practical RDF.** O'Reilly & Associates: Estados Unidos, 2003, 337 p.

(QDOX, 2012) QDox: **QDox Java Parser Extractor**. Disponível em <http://qdox.codehaus.org/>. Acesso em: 31 mai. 2012.

(QUINTARELLI, 2005) QUINTARELLI, E. **Folksonomies: power to the people.** Disponível em <http://www.iskoi.org/doc/folksonomies.htm>. Acesso em 10 nov. 2011.

(RAMAL et al, 2002) RAMAL, M. F..; MENESES, R. M.; ANQUETIL, N. **A Disturbing Result on the Knowledge Used During Software Maintenance.** In 9th Working Conference on Reverse Engineering, IEEE, p. 277 - 286, 2002.

(REDMOND, 2012) REDMOND, T.: **Protege-OWL Code Generator**. Disponível em <http://protegewiki.stanford.edu/wiki/Protege-OWL_Code_Generator>. Acesso em: 31 mai. 2012.

(REZENDE, 2005) REZENDE, Denis. **Engenharia de Software e Sistemas de Informação.** Brasport: Rio de Janeiro, 2005. 313 p.

(RILLING et al, 2006) RILLING, J.; YONGGANG, Z.; MENG, W. J.; WITTE, R.; HAARSLEV, V.; CHARLAND, P. **A Unified Ontology-Based Process Model for Software Maintenance and Comprehension.** In MoDELS'06 Proceedings of the 2006 international conference on Models in software engineering, Springer-Verlag, p. 56 – 65, 2006.

(ROCHA-PINTO et al, 2009) ROCHA-PINTO, S. R.; PEREIRA, C. S.; COUTINHO, M. T. C.; JOHANN, S. L. **Dimensões funcionais da gestão de pessoas**. FGV: Rio de Janeiro, 2009. 146 p.

(SALOMON, 1993) SALOMON, D. V. **Como fazer uma Monografia**. Martins Fontes: São Paulo, 1993, 54 p.

(SCERRI et al, 2007) SCERRI, S., SINTEK, M., VAN ELST. L., HANDSHUCH, S. **NEPOMUK Annotation Ontology**. Disponível em <http://www.semanticdesktop.org/ontologies/nao/>. Acesso em 28 mar. 2012.

(SCHNEIDER, 2009) SCHNEIDER, K. **Experience and Knowledge Management in Software Engineering.** Springer: Alemanha, 2009. 235 p.

(SEACORD et al., 2003) SEACORD, R.C.; PLAKOSH, D., LEWIS, G. A. **Modernizing Legacy Systems – Software technologies, engineering processes, and business practices**. Addison-Wesley: Boston, 2003, 332 p.

(SELIGMAN, 2000) SELIGMAN, L. S. **Adoption as sensemaking: toward an adopter-centered process model of IT adoption**. In Proceedings of ICIS, p. 361-370, 2000.

(SEN et al, 2006) SEN, S.; LAM, S. K.; RASHID, A. M.; COSLEY, D.; FRANKOWSKI, D.; OSTHERHOUSE, J.; HARPER, F. M.; RIEDL, J. **Tagging, communities, vocabulary, evolution.** In CSCW '06 Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, ACM, p. 181 – 190, 2006.

(SHANK, 1998) SHANK, G. **The extraordinary powers of abductive reasoning. Theory and Psychology.**, 8, 6, p. 841-60.

(SHARMA, 2011) SHARMA, N. **Role of Available and Provided Resources in Sensemaking**. In Proceedings of the 2011 annual conference on Human factors in computing systems CHI 2011, ACM, p. 1807 – 1816, 2011.

(SHIRKY, 2005) SHIRKY, C. **Ontology is Overrated: Categories, Links, and Tags.** Disponível em < http://shirky.com/writings/ontology_overrated.html>. Acesso em 16 nov. 2011.

(SHNEIDERMAN, 1992) SHNEIDERMAN, B. **Designing the user interface: Effective strategies for effective human-computer interaction.** Addison Wesley, 2 ed, 1992.

(SINDHGATTA, 2008) SINDHGATTA, R. **Identifying Domain Expertise of Developers from Source Code.** In Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, p. 981 – 989, 2008.

(SINHA, 2006) SINHA, R. **Tagging from Personnal to Social : Observations and Design Principles**. In Tagging Workshop, World Wide Web Int. Conf., 2006.

(SMITH, WELTY, 2001) SMITH, B., WELTY, C. **Ontology: Towards a new synthesis.** In Formal Ontology in Information Systems, ACM, p. iii-x, 2001.

(SOLOWAY et al, 1988) SOLOWAY, E., ADELSON, B., EHRLICH, K. **Knowledge and Processes in the Comprehension of Computer Programs.** In The Nature of Expertise, p. 129 – 152, 1988.

(SOMMERVILLE, 2003) Sommerville, I. **Engenharia de Software.** Addison Wesley: São Paulo, 2003. 592 p.

(SOUZA et al., 2005) SOUZA, S. C. B.; ANQUETIL, N.; OLIVEIRA, K. M. **A Study of the Documentation Essential to Software Maintenance.** In Proceedings of the

23rd Annual international Conference on Design of Communication, Coventry, United Kingdom, 2005.

(SOWA, 2000) SOWA, J. F. **Ontology, metadata and semiotics.** In: Ganter, B. Mineau G., editors, Conceptual structures: Logic, linguistic and Computational issues, Springer-Berlin, p. 55 – 81, 2000.

(STOREY et al, 2006) STOREY, M.; CHENG, L.; RIGBY, I. **Shared Waypoints and Social Tagging to Support Collaboration in Software Development.** In CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, ACM, p. 195 – 198, 2006.

(STURTZ, 2004) STURTZ, D. **Comunal Categorization: The Folksonomy.** Disponível em: <http://www.davidsturtz.com/drexel/622/communal-categorization-the-folksonomy.html>. Acesso em 05 nov. 2011.

(TAYLOR, VAN EVERY, 2000) TAYLOR, J. R., VAN EVERY, E. J. **The Emergent Organization: Communication as its Site and Surface.** Erlbaum: Mahwah, 2000.

(TELEA; LUCIAN, 2011) TELEA, Alexandru.; LUCIAN, Voinea. **Visual software analytics for the build optimization of large-scale software systems.** Computational Statistics 26 (4), p. 635–654, 2011.

(THURAISINGHAM, 2002) THURAISINGHAM, B. M. **XML databases and the semanci web.** CRC Press : Estados Unidos, 2002, 302 p.

(VAN DER MAREN, 1996) VAN DER MAREN, J. **Méthodes de recherché pour l'Éducation.**, De Boeck: Bélgica, 1996.

(W3C, 2009) MATHES, A. **OWL 2 Web Ontology Language Document Overview.** Disponível em <http://www.w3.org/TR/owl2-overview/>. Acesso em 16 nov. 2011.

(WAL, 2005) WAL, V. **Explaining and Showing Broad and Narrow Folksonomies.** Disponível em: <http://www.personalinfocloud.com/2005/02/explaining_and_.html>. Acesso em 10 nov. 2011.

(WAL, 2007) WAL, V. **Folksonomy.** Disponível em <http://vanderwal.net/folksonomy.html>. Acesso em 10 nov. 2011.

(WEICK et al, 2005) WEICK, K. E., SUTCLIFFE, K. M., OBSTFELD, D. **Organizing and the Process of Sensemaking.** Organization Science, p. 409-421, 2005.

(WEICK, 1993) WEICK, K. E. **The collapse of sensemaking in organizations: The Mann Gulch disaster.** Administrative science quarterly , 38, n. 4, p. 628-52, 1993.

(WEICK, 1995) WEICK, K. E. **Sensemaking in organizations.** Stage Publications: Thousand Oaks, Estados Unidos, p. 61-62, 1995.

(WEISS, 1999) WEISS, G. **Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence**, The MIT Press Cambridge: Londres, Inglaterra, 1999, 648 p.

(WIEDENBECK, 2005) WIEDENBECK, S. **Factors affecting the success of non-majors in learning to program.** In Proceedings of the First international Workshop on Computing Education Research, ACM, p. 13 – 24, 2005.

(WITTE et al., 2007) WITTE, R.; ZHANG, Y., RILLING, J. **Empowering Software Maintainers with Semantic Web Technologies.** Springer-Verlag Berlin, n. 4519, p. 37 – 52, 2007.

(WOODS; YANG, 1996) WOODS, S.; YANG, Q. **The Program Understanding Problems: Analysis and a Heuristic Approach.** In IEEE Proceedings of ICSE-18, IEE, p. 6-15, 1996.

(WU et al, 2006) WU, X., ZHANG, L., YU, Y. **Exploring social annotations for the semantic web**. In WWW '06 Proceedings of the 15th international conference on World Wide Web, ACM, p. 417 – 426, 2006.

(YANG et al., 2005) YANG, Jeong.; HENDRIX, Dean.; CHANG, Kai.; UMPHRESS, David. **An Empirical Validation of Complexity Profile Graph.** In Proceedings of the 43rd annual Southeast regional conference, ACM, v. 1, p. 143 – 149, 2005.

(ZAIN et al., 2011) ZAIN, J. M.; MOHD, W. M. b. W.; EL-QAWASMEH, E. **Software Engineering and Computer Systems.** Springer-Verlag: Alemanha, 2011, 1 ed, p. 155 – 167.

(ZELKOWITZ et al, 1979) ZELKOWITZ, M.; SHAW, A.; GANNON, J.. **Principles of Software Engineering and Design.** Prentice Hall Inc , p. 157 – 178, 1979.

(ZHANG, 2007) ZHANG, Y. **An ontology-based program comprehension model**. Tese doutorado, Universidade de Concordia, Canada, 2007.

(ZHAO et al, 2009) ZHAO, Y., DONG, J., PENG, T. **Ontology Classification for Semantic-Web-Based Software Engineering**. In IEEE Transactions on Services Computing, IEE, v. 2, n. 4, p. 303-317, 2009.

(ZHOU et al, 2008) ZHOU, H., CHEN, F., YANG, H. **Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology**. In Quality Software, 2008. QSIC '08. The Eighth International Conference, IEEE, p. 225 – 234, 2008.

APÊNDICE A – FORMULÁRIO DE PESQUISA ACADÊMICA

Esta pesquisa tem como objetivo realizar a avaliação de um método para melhorar a manutenção e a compreensão de código fonte orientado a objetos. Esta é uma das etapas da dissertação de mestrado de Daniel Schreiber, que está sendo desenvolvida sob orientação da Dra Andreia Malucelli, no Programa de Pós-Graduação em Informática (PPGIa) da Escola Politécnica da Pontifícia Universidade Católica do Paraná (PUCPR). O título deste trabalho é: “MÉTODO BASEADO EM SENSEMAKING PARA COMPREENSÃO DE CÓDIGOS FONTE ORIENTADOS A OBJETOS”.

Desde já agradeço pela sua atenção e colaboração.

Daniel Schreiber
xiraba@gmail.com
Curitiba, 8 de junho de 2012.

ESCLARECIMENTOS:

a) O tempo total previsto para responder esta pesquisa é de 6 minutos. A pesquisa está organizada da seguinte forma:

Parte 1 - Caracterização do Profissional

Parte 2 – Questionário geral relacionado à manutenção de software

b) O Apêndice A apresenta o Acordo de Confidencialidade

*Obrigatório

Parte 1 - Caracterização do Profissional

Nome: *

E-mail: *

Instituição/empresa (não obrigatório):

Quais atividades você desempenha atualmente?
É possível selecionar mais de uma opção

- [ ] Programador
- [ ] Analista de Sistemas
- [ ] Arquiteto de Software
- [ ] Outro:

Há quantos anos trabalha como desenvolvedor de software?

- ( ) Trabalho há menos de 1 ano
- ( ) Trabalho há um ano ou mais
- ( ) Trabalho há mais de cinco anos

Com quais linguagens de programação você trabalha ou já trabalhou?
É possível selecionar mais de uma opção

- [ ] Java
- [ ] C#
- [ ] Flex
- [ ] Cobol
- [ ] Visual Basic
- [ ] Pascal
- [ ] PHP
- [ ] Asp
- [ ] Progress
- [ ] Outro:

Há quanto tempo você trabalha com programação orientada a objetos?

- ( ) Trabalho há menos de 1 ano
- ( ) Trabalho há um ano ou mais
- ( ) Trabalho há mais de cinco anos

De que forma você iniciou a programação:

- ◯ Diretamente com o paradigma de orientação a objetos
- ◯ Inicialmente a partir do paradigma estruturado

Parte 2 – Manutenção de Código Fonte

Como você realiza a estruturação e organização do código fonte? *

- ◯ Por diretório, agrupando por funcionalidades semelhantes
- ◯ Por diretório, agrupando por unidade de negócio/domínio
- ◯ Por diretório, agrupando por camada da arquitetura (lógica de negócio / lógica visual)
- ◯ Não estruturo o código fonte
- ◯ Outro:

Em que tipo de projeto você atua? *

- ◯ Desenvolvimento: crio novos projetos
- ◯ Suporte: realizo manutenção em projetos existentes
- ◯ Ambos
- ◯ Outro:

Como você avalia a importância de adicionar comentários no código fonte? *

- ◯ Nada importante
- ◯ Pouco importante
- ◯ Importante
- ◯ Muito importante
- ◯ Imprescindível

Você costuma adicionar comentários em trechos de CÓDIGO COMPLEXOS? *

- ◯ Nunca
- ◯ Raramente
- ◯ Frequentemente
- ◯ A maioria das vezes
- ◯ Sempre

Você costuma adicionar comentários em trechos de CÓDIGO SIMPLES? *

- ◯ Nunca
- ◯ Raramente
- ◯ Frequentemente
- ◯ A maioria das vezes
- ◯ Sempre

Qual o seu critério para dar nome às classes? *

- ◯ De acordo com a funcionalidade da classe
- ◯ De acordo com a sua definição de negócio
- ◯ Concatenando a funcionalidade e definição de negócio no nome
- ◯ Não utilizo critério para dar nome à classe
- ◯ Outro:

Qual o seu critério para dar nome aos métodos? *

- ◯ De acordo com a funcionalidade do método
- ◯ De acordo com a sua definição de negócio
- ◯ Concatenando a funcionalidade e definição de negócio no nome
- ◯ Não utilizo critério para dar nome aos métodos
- ◯ Outro:

Qual o seu critério para dar nome aos atributos da classe? *

- ◯ De acordo com o tipo e a funcionalidade da classe
- ◯ De acordo com a sua definição de negócio
- ◯ Concatenando a funcionalidade e definição de negócio no nome
- ◯ Não utilizo critério para dar nome à classe
- ◯ Outro:

Os códigos provenientes de outros desenvolvedores e que você realiza manutenção são fáceis de entender? *

- ◉ Nunca
- ◉ Raramente
- ◉ Frequentemente
- ◉ A maioria das vezes
- ◉ Sempre
- ◉ Não realizo manutenções de código de outros desenvolvedores
- ◉ Outro:

Quando você está realizando uma manutenção em um código que não conhece, quais estratégias você utiliza para facilitar a compreensão e registrar questões de descoberta a partir do código fonte? *

É possível selecionar mais de uma opção

- ☐ Anotações pessoais diretamente no código fonte, por meio de comentários
- ☐ Anotações pessoais em papel
- ☐ Anotações pessoais em um editor de texto
- ☐ Anotações pessoais usando mapas mentais
- ☐ Anotações pessoais usando diagramas (UML, por exemplo)
- ☐ Conversas diretamente com o autor do código
- ☐ Conversas diretamente com outras pessoas que conhecem o código
- ☐ Leitura da documentação do software
- ☐ Utilizo recursos disponíveis na IDE
- ☐ Não utilizo nenhuma estratégia
- ☐ Outro:

Com que frequência você recorre à documentação do projeto para compreender o código fonte? *

- ◉ Nunca
- ◉ Raramente
- ◉ Frequentemente
- ◉ A maioria das vezes
- ◉ Sempre

Com base na sua experiência, você considera que a documentação disponível nos projetos que participou auxiliou na manutenção? *

- ( ) Nunca
- ( ) Raramente
- ( ) Frequentemente
- ( ) A maioria das vezes
- ( ) Sempre

Quais ferramentas ou recursos que a IDE oferece e você utiliza para auxiliar na compreensão do código fonte? *

É possível selecionar mais de uma opção

- [ ] Não utilizo nenhuma ferramenta ou recurso da IDE
- [ ] Outline browser
- [ ] Ferramenta Case
- [ ] UML designer
- [ ] Agrupadores
- [ ] Outro:

Quais técnicas de marcação ou anotações no código fonte você utiliza? *

É possível selecionar mais de uma opção

- [ ] Técnicas que a própria IDE disponibiliza
- [ ] Plugins na IDE para marcar e anotar o código fonte
- [ ] Ferramentas de terceiros para realizar marcações e anotações
- [ ] Nenhuma
- [ ] Outro:

Você considera que uma explicação pessoal do autor do código fonte facilita a manutenção? *

- ( ) Nunca
- ( ) Raramente
- ( ) Frequentemente
- ( ) A maioria das vezes
- ( ) Sempre

Você considera que explicação pessoal de um desenvolvedor que realiza manutenção frequente no código fonte facilita a manutenção? *

- ○ Nunca
- ○ Raramente
- ○ Frequentemente
- ○ A maioria das vezes
- ○ Sempre

Comentários:

TERMO DE CONFIDENCIALIDADE

Este Termo de Confidencialidade visa estabelecer um acordo entre os pesquisadores Daniel Schreiber e Andreia Malucelli, doravante denominados Pesquisadores e o profissional desenvolvedor doravante denominado Participante, a respeito da confidencialidade das informações coletadas durante o processo de pesquisa da dissertação de mestrado do primeiro, sob orientação do segundo, intitulado: “MÉTODO BASEADO EM SENSEMAKING PARA COMPREENSÃO DE CÓDIGOS FONTE ORIENTADOS A OBJETOS”.

Por meio deste Termo de Confidencialidade, os Pesquisadores se comprometem a:

- portar-se com discrição em todos os momentos da pesquisa acadêmica, não comentando ou divulgando qualquer tipo de informação que tenha sido repassada de forma oral ou escrita;
- não divulgar o nome do Participante, em qualquer meio, a menos que expressamente autorizado por este;
- não divulgar, em qualquer meio, os dados e informações individualizadas coletados durante o processo de pesquisa com o Participante;

- divulgar, em formato de dissertação, artigos e apresentações, apenas os dados agregados, dos quais não se possa retirar ou inferir a identificação do Participante;
- retornar para o Participante as informações coletadas e analisadas, em formato individualizado dos seus próprios dados e em formato agregado com os dados de todos os demais Participantes.

As assinaturas abaixo expressam a concordância quanto ao cumprimento deste Termo de Confidencialidade, por prazo indeterminado.

Curitiba, 8 de junho de 2012.

Daniel Schreiber Andreia Malucelli

APÊNDICE B – ANÁLISE DA PESQUISA DE CAMPO

Este apêndice apresenta a pesquisa de campo realizada com programadores que atuam na atividade de manutenção de software. Tem por objetivo apoiar a investigação referente a extração e desenvolvimento do conhecimento durante as manutenções em códigos fonte e averiguar a dificuldade, barreiras e oportunidades que podem ser exploradas para auxiliar na subatividade de compreensão do código fonte.

Pesquisa de Campo

A pesquisa de campo tem por objetivo observar os fatos diretamente no local onde ocorreram os fatos ou fenômenos. Os dados podem ser obtidos por meio de entrevistas, questionários, consultas, depoimentos e registro de ocorrências de determinados fenômenos (CIRIBELLI, 2003).

Pesquisa de campo é muito utilizada, porque leva o pesquisador a entrar em contato direto com a realidade. Sua pesquisa é utilizada com a finalidade de recolher e registrar ordenadamente os dados relativos aos assuntos escolhidos como objeto de estudo (CIRIBELLI, 2003).

Apresenta-se como investigação empírica diretamente no local onde estão ocorrendo os fenômenos ou que dispõe de elementos para a realização da investigação (KAHLMEYER-MERTENS et al, 2007). Conforme Kahlmeyer-Mertens, os tipos de pesquisas de campo enquadram-se como: experimental, pesquisa-ação, estudo de caso, pesquisa etnográfica e fenomenológica.

A pesquisa de campo desenvolvida, sob o caráter experimental, tem por objetivo avaliar a complexidade e dificuldade envolvidas no entendimento e execução de manutenção de código fonte de outras autorias. Tem-se como premissa testar as seguintes questões:

- **técnicas**: avaliar se os programadores possuem dificuldades em realizar manutenção de código fonte de outra autoria; quais as técnicas que os programadores utilizam para ajudar no processo de manutenção; e como é realizada a estruturação do código.
- **documentação**: Qual o papel real da documentação? A documentação é útil e utilizada? Como a técnica de adicionar comentário no código fonte é vista e utilizada.

- **entendimento**: com base nas técnicas e documentação, averiguar a questão do conhecimento envolvido. Como os programadores avaliam a questão do entendimento e repasse de conhecimento para melhorar o processo de compreensão.

Resultados e Análise da Pesquisa de Campo

No total 119 profissionais responderam a pesquisa de campo realizada. Destes, 54% correspondiam a programadores seniores e 46% a programadores juniores.

A pesquisa foi dividida em duas partes. A primeira era composta por perguntas relacionadas a características do respondente, como tempo de atuação, atividades que desempenha, conhecimentos gerais sobre linguagens de programação e orientação a objetos. A segunda parte correspondia a pesquisa propriamente dita, com questões referentes ao tema manutenção e compreensão, abordando hábitos de programação, dificuldades e técnicas utilizadas. O objetivo era analisar como os programadores trabalham em seus projetos, desde a estruturação do código até a codificação propriamente dita.

A primeira pergunta correspondia a questão sobre estruturação e organização do código fonte. Dentre as repostas (Figura B-1), nota-se que é costume agrupar o código fonte com base na arquitetura adotada pelo projeto. Desta forma, toda nova manutenção por profissionais que não conhecem o projeto será baseada na arquitetura do projeto, sendo possível concluir que o conhecimento da arquitetura do projeto é um fator importante, pois este auxilia no processo inicial do *sensemaking*.

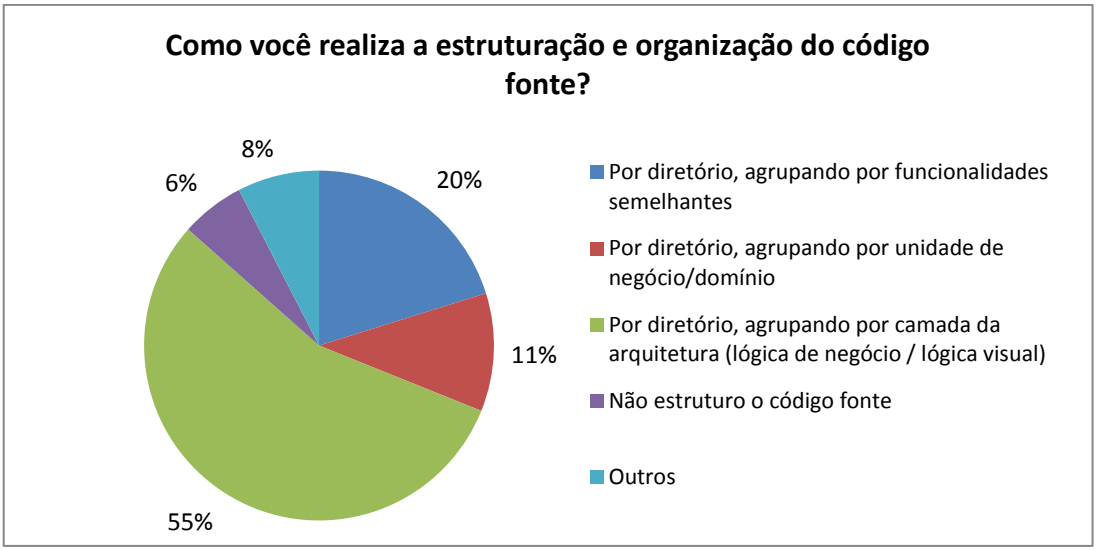

Figura B-1 - Como os programadores realizam a estruturação e organização do código fonte

As perguntas “Qual o seu critério para dar nome às classes?”; “Qual o seu critério para dar nome aos métodos?” e “Qual o seu critério para dar nome aos métodos?” avaliam outro fator fundamental que afeta diretamente na compreensão: a nomenclatura.

A nomenclatura adotada corretamente ajuda a manter o código fácil de ler para outros programadores. A legibilidade do código fonte é importante porque contribui para diminuir o tempo gasto pelos programadores a tentar descobrir o que o código faz, deixando mais tempo para correção ou modificação (LEAHY, 2012).

A nomeação para as classes, conforme os resultados apresentados na Figura B-2, demonstraram que 93% dos respondentes aplicam critérios de nomenclaturas relacionados a funcionalidade que a classe desempenha, com base na funcionalidade da classe, na definição de negócio e/ou ambos. Também deve-se considerar que 5,88% responderam que utilizam outras técnicas de nomenclatura. Desta forma, conclui-se que mais de 98% dos programadores estão conscientes da importância de manter o código coeso e organizado por meio da nomenclatura.

A mesma avaliação foi realizada para métodos e atributos de classe, levando em consideração o fator de negócio e a funcionalidade. Observou-se que a mesma proporção da classificação foi obtida, conforme demonstra a Figura B-3, Figura B-4.

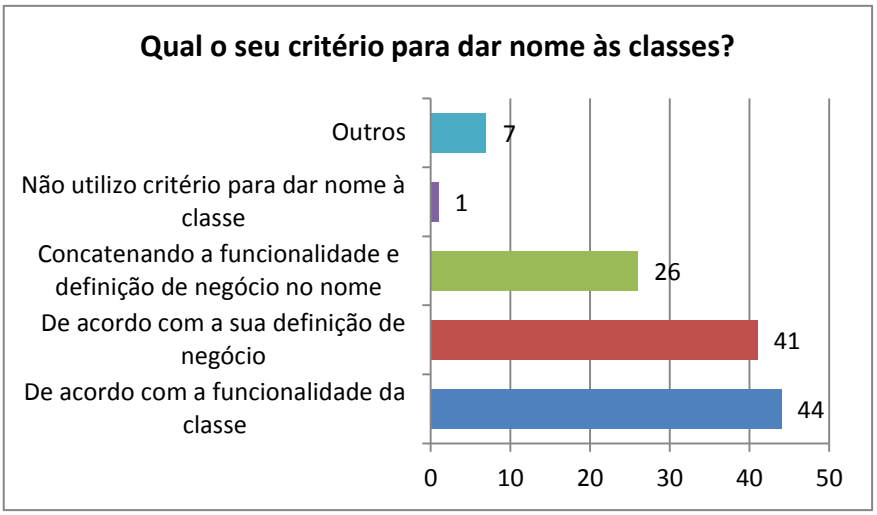

Figura B-2 - Critérios utilizados pelos programadores para dar nome às classes

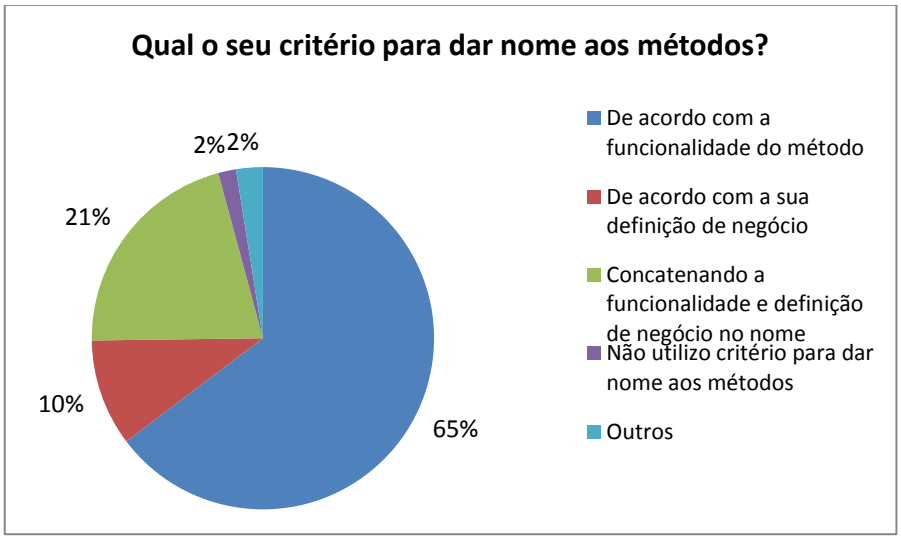

Figura B-3 - Critérios utilizados pelos programadores para dar nomes aos métodos

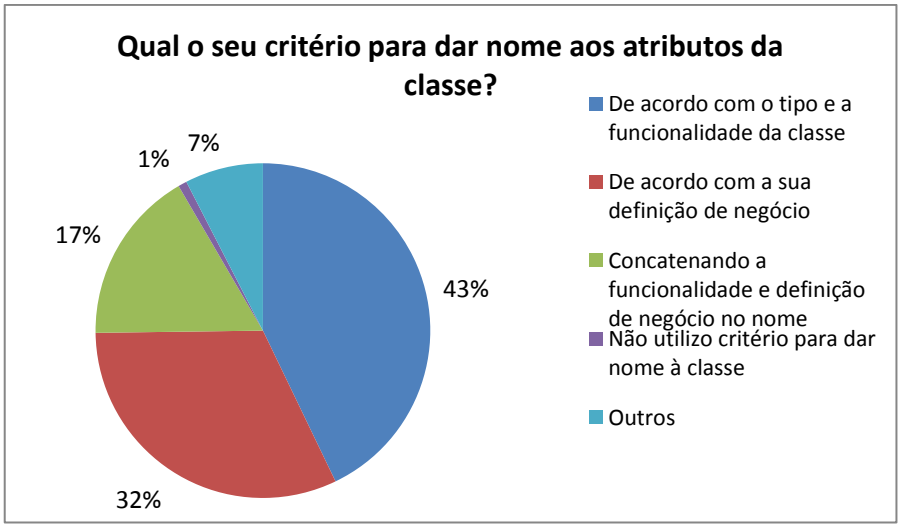

Figura B-4 - Critérios utilizados pelos programadores para dar nome aos atributos da classe

Outro ponto importante avaliado na pesquisa é a forma como os programadores realizam os comentários. Enquanto a linguagem de programação está relacionada diretamente a interpretação da máquina, o comentário está direcionado a interpretação humana, o que a torna essencial no entendimento do código fonte.

Foi questionado aos programadores se eles realizavam a documentação do código fonte por meio de comentários. A questão seguinte teve como objetivo averiguar o grau de importância da documentação no código fonte pelos programadores. A Figura B-5 ilustra o resultado deste questionamento.

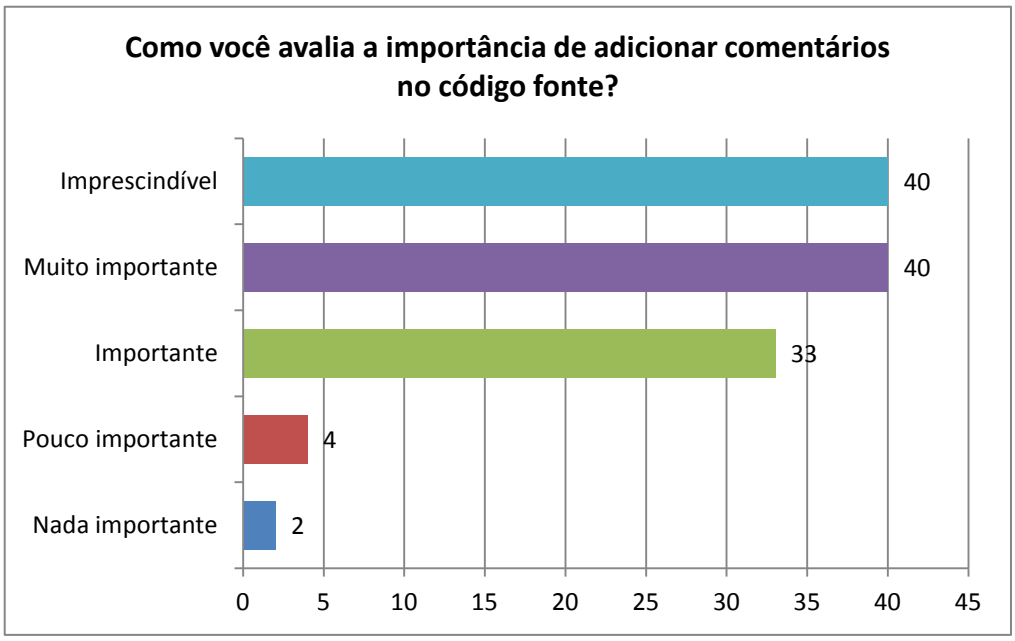

Figura B-5 – Como os programadores avaliam a importância de adicionar comentários no código fonte

95% dos programadores entrevistados concordam que a documentação é um fator de suma importância para o código fonte. Apesar de sua importância, há profissionais que durante a aplicação do experimento para avaliação do ambiente proposto neste trabalho defenderam que o código deva ser “auto documentável”, passando a dar menos importância a documentação em códigos simples. Também durante o experimento alguns programadores relataram considerar que comentários em excesso pode dificultar a leitura do código, principalmente aqueles que não agregam valor significativo, conforme demonstra a Figura B-6.

```
...
//Verifica a ordem de corte
if ( ordem_corte == 3){
        //se entrou aqui é porque será finalizada
        finalizar = true;
} else {
        //se entrou aqui é porque não será finalizada
        finalizar = false;
}
...
```

Figura B-6 - Demonstração de comentários desnecessários

Os programadores que participaram do experimento sugeriram que os comentários sejam mais elaborados e realizados em trechos de códigos complexos, onde a regra é muito complexa e/ou há itens subjetivos relacionados à regra de negócio implementada. Com base nesta sugestão, foi questionado a prática de documentar códigos simples e complexos, e observou-se realmente uma forte tendência a documentar apenas trechos de códigos complexos, conforme visto na Figura B-7 e Figura B-8.

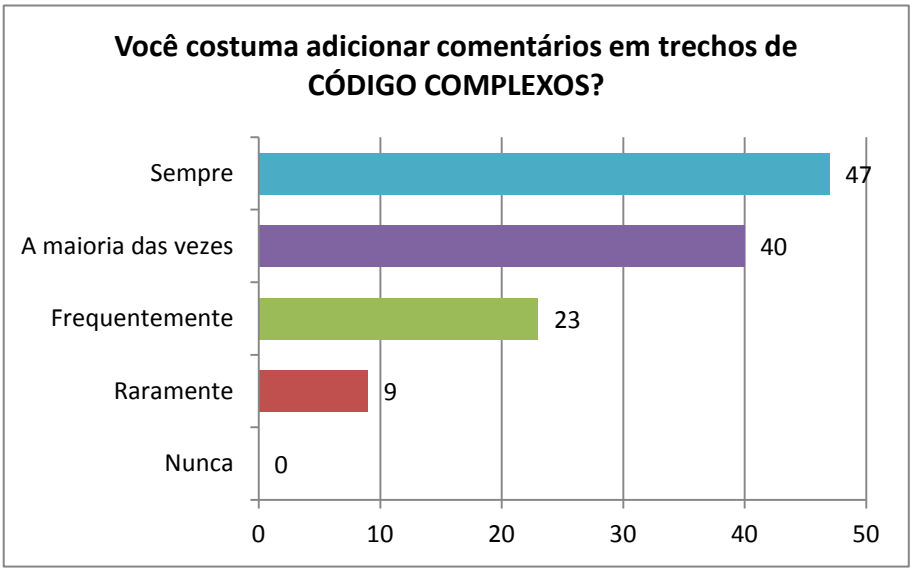

Figura B-7 – Frequência de documentar via comentários códigos fonte complexos

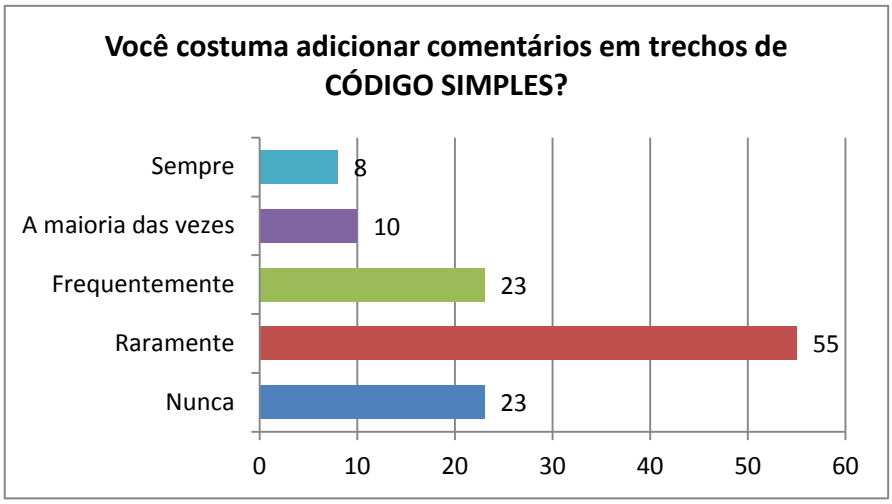

Figura B-8 – Frequência de documentar via comentários códigos fonte simples

No entanto, a realização ou não de comentários em trechos de códigos pode ser questionável. Independente da realização ou não da prática de documentação, o que pode ser um trecho de código complexo para uma pessoa pode ser simples para outra. Ou seja, não é possível concluir, com base nesta pesquisa, se a prática da documentação é realizada de forma correta e consciente. Porém, concluiu-se que os participantes estão cientes da sua importância durante o processo de codificação e manutenção.

Uma questão importante que merece especial atenção é a documentação existente. Dos entrevistados, 52% afirmaram que a documentação raramente ajuda na manutenção. Ou seja, a sua existência não tem valor significativo para a manutenção, pois independente de sua existência, os programadores são obrigados a gastar esforços na compreensão e entendimento para que, somente assim, seja realizada a manutenção programada (Figura B-9).

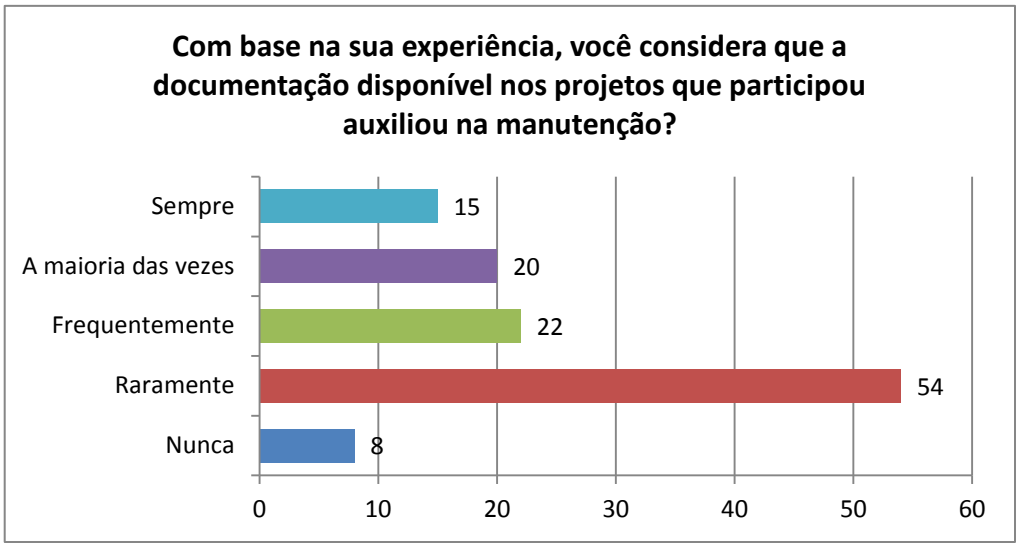

Figura B-9 – Como os programadores avaliam o uso da documentação disponível nos projetos como auxílio na manutenção

Em contraste, a cultura de buscar na documentação fatos que possam auxiliar a manutenção, também pode ser questionada. A questão “Com que frequência você recorre à documentação do projeto para compreender o código fonte?”, Figura B-10, demonstrou que 39% responderam que raramente procuram a documentação, e 36% frequentemente fazem isso.

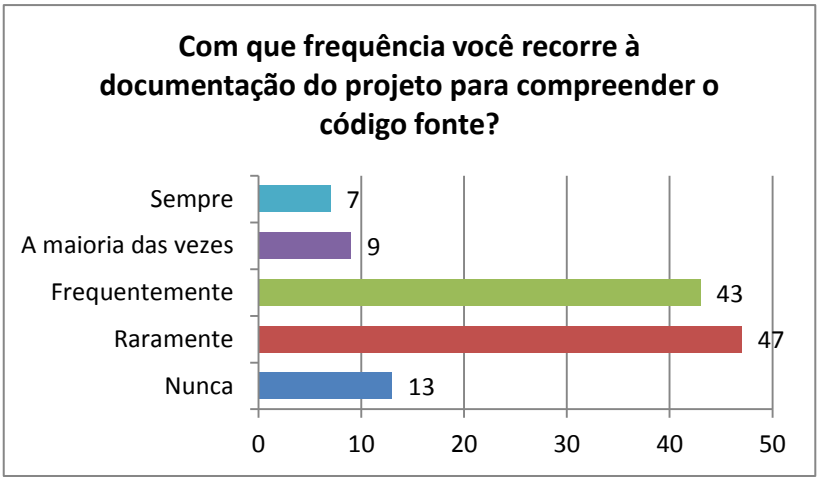

Figura B-10 - Com que frequência os programadores recorrem à documentação do projeto para compreender o código fonte

Engenheiros de software não costumam buscar conhecimento em documentos externos durante a atividade de manutenção, preferem trabalhar com o que já sabem (RAMAL et al, 2002). Com base na pesquisa realizada, não é possível concluir o real motivo. Uma das hipóteses é a formação profissional e experiência. No entanto, quanto a esta questão, observou-se que:

- mais de 55% dos programadores seniores afirmaram que a documentação auxilia positivamente a manutenção (opções *frequentemente, A maiora das vezes e Sempre*). E neste mesmo grupo, 52% possuem o costume de buscar ajuda na documentação (opções *frequentemente, A maiora das vezes e Sempre*), ao invés de compreender o código fonte de forma *bottom-up*.
- os programadores juniores possuem menos a cultura de buscar suporte na documentação. Apenas 46% responderam que praticam esta atividade (opções *frequentemente, A maiora das vezes e Sempre*). No entanto, 40% não acham plausível que a documentação ajude na manutenção.

Devido a grande discrepância e a observação de não haver um padrão entre os programadores juniores e programadores seniores, pode-se concluir, também, que a qualidade da documentação e a capacidade de interpretação dos documentos podem interferir diretamente nesta questão. Se um programador sênior sabe, por experiência, da importância da documentação, busca informações nestes documentos, mas considera-o, de certa forma, inútil para a manutenção e compreensão, pode-se concluir que haja algum tipo de problema na qualidade das documentações de projetos existentes.

Outra possível interpretação, com base na importância da documentação para os programadores juniores, e sua não utilização, seria a possível dificuldade apresentada para entender e interpretar estes documentos.

Porém, quando se realiza manutenção em sistemas legados, de outros autores e de outras empresas, por exemplo, a documentação, apesar de importante, pode não estar disponível. Neste caso, a única solução é iniciar a manutenção e compreensão a partir do início.

As próximas questões foram elaboradas com objetivo de extrair dos respondentes as atividades diretas e indiretas utilizadas durante a compreensão do código fonte.

A pergunta “Os códigos provenientes de outros desenvolvedores e que você realiza manutenção são fáceis de entender?” demonstrou que existem dificuldades de entender códigos de outra autoria. Apesar de existir um certo equilíbrio entre o fácil e o difícil, 34% opinaram como “frequentemente”, demonstrando a existência de

um certo tipo de desconforto por parte das pessoas que realizam manutenção, conforme pode ser visto na Figura B-11.

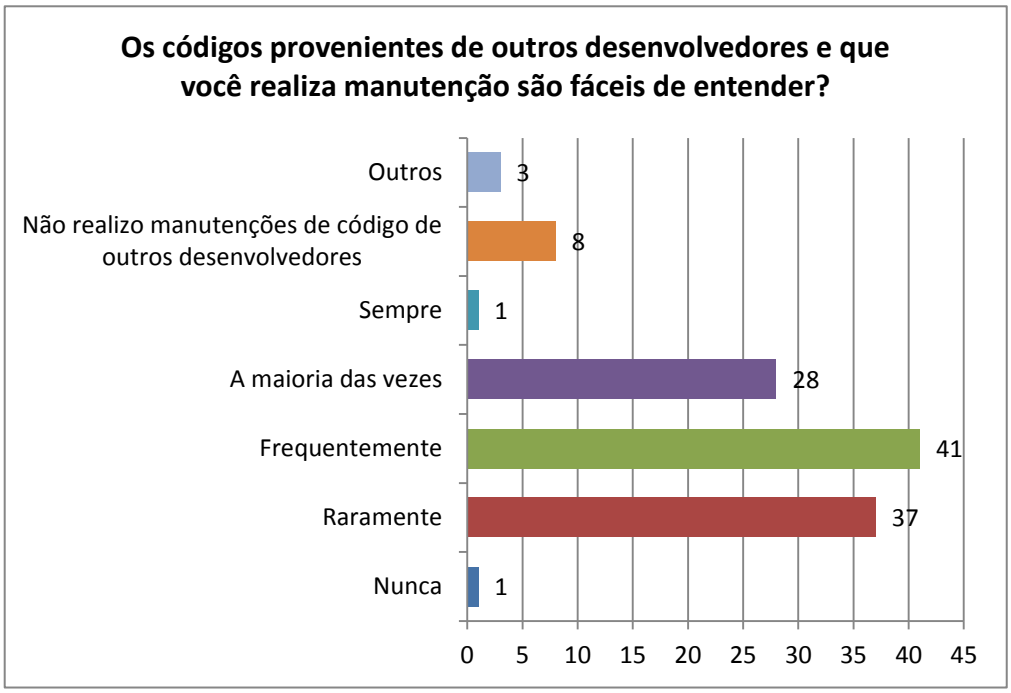

Figura B-11 – Como os programadores avaliam o entendimento de códigos em manutenção provenientes de outros desenvolvedores

Nota-se que 66% dos programadores demonstraram uma tendência para a dificuldade de entender o código fonte de terceiros, respondendo como “nunca”, “raramente” ou “frequentemente”. Podem existir vários fatores que levam a esta tendência: falta de padronização, falta de conhecimento técnico e de negócio, desinteresse em manter o código organizado, dentre outros. No entanto, existem algumas técnicas e ferramentas que podem ser utilizadas como apoio durante a manutenção para diminuir a complexidade de trabalhar com código fonte de terceiros, desde plug-ins e ferramentas disponibilizadas pela própria IDE de desenvolvimento, quanto outras ferramentas existentes.

Com base nisso, uma questão foi elaborada para demonstrar a utilização destas ferramentas pelos programadores. A Figura B-12 mostra o resultado.

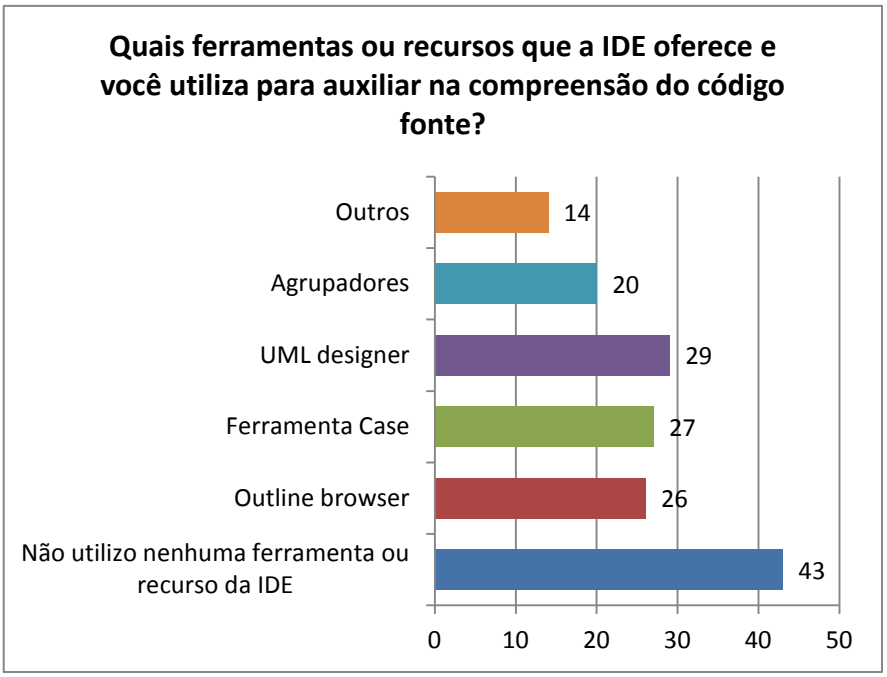

Figura B-12 – Ferramentas/recursos que os desenvolvedores utilizam para auxiliar na compreensão do código fonte

Observou-se que 43 pessoas não utilizam ferramentas para apoiar no entendimento do código fonte. Não é possível concluir qual a razão deste resultado observado. Uma das possíveis hipóteses seria o relacionamento com o fator psicológico, visto que a compreensão está relacionado com o conhecimento existente no código fonte, e a sua dificuldade de extrair e reutilizar este conhecimento.

A pergunta “Quando você está realizando uma manutenção em um código que não conhece, quais estratégias você utiliza para facilitar a compreensão e registrar questões de descoberta a partir do código fonte?” pode dar um peso maior a esta hipótese, conforme visto na Figura B-13. Dentre as técnicas sugeridas nesta questão que tratam sobre a extração e manipulação do conhecimento de forma direta e indiretamente, a realização de uma conversa pessoal com o autor ou com uma pessoa que detém maior conhecimento sobre tal código, foi uma das técnicas de maior preferência observadas.

Não resta dúvida, conforme a pesquisa realizada, que este repasse do autor do código facilita o processo de manutenção pelas pessoas que estão iniciando naquele código. A maioria dos entrevistados, representando 41%, afirmaram que o repasse sempre ajuda na manutenção. A maioria das vezes, representado por 29% e frequentemente, por 21% ajudam a suportar esta afirmação. A mesma relação é

visível ao recebimento de repasse por outra pessoa que não é autor, mas possui um conhecimento maior sobre o código fonte (Figura B-14 e Figura B-15).

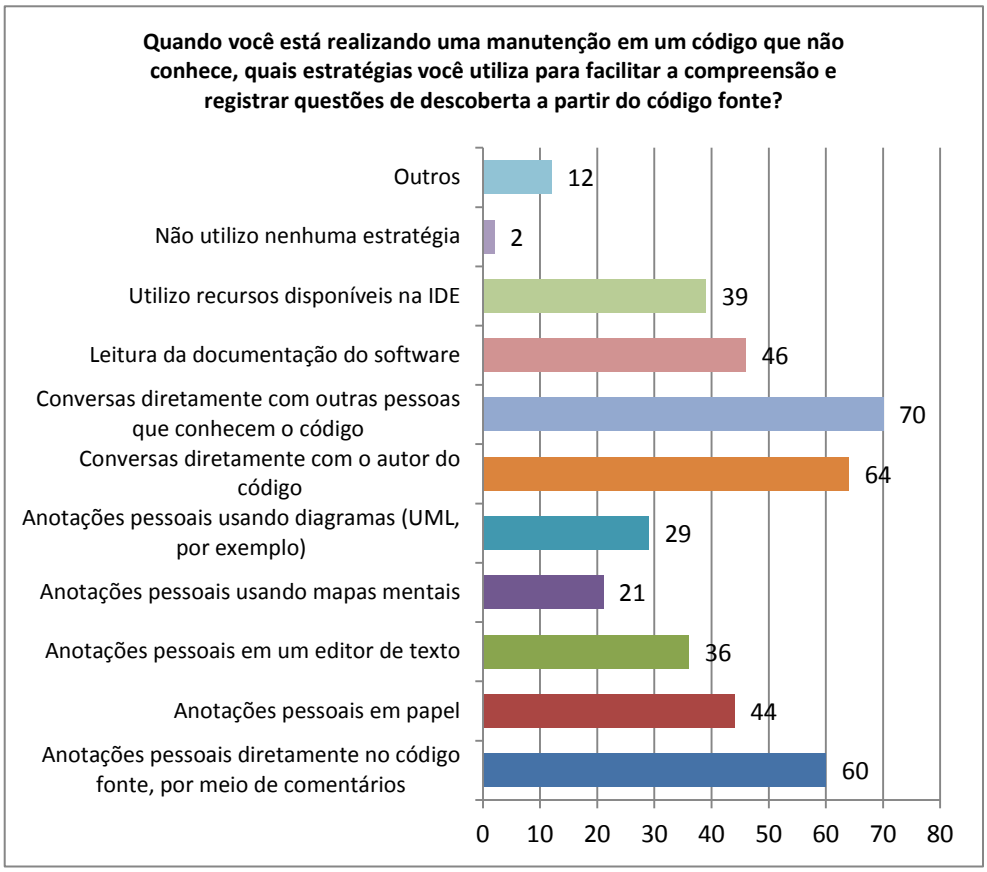

Figura B-13 – Estratégias adotadas pelos desenvolvedores adotam para auxiliar no entendimento de códigos fonte desconhecidos

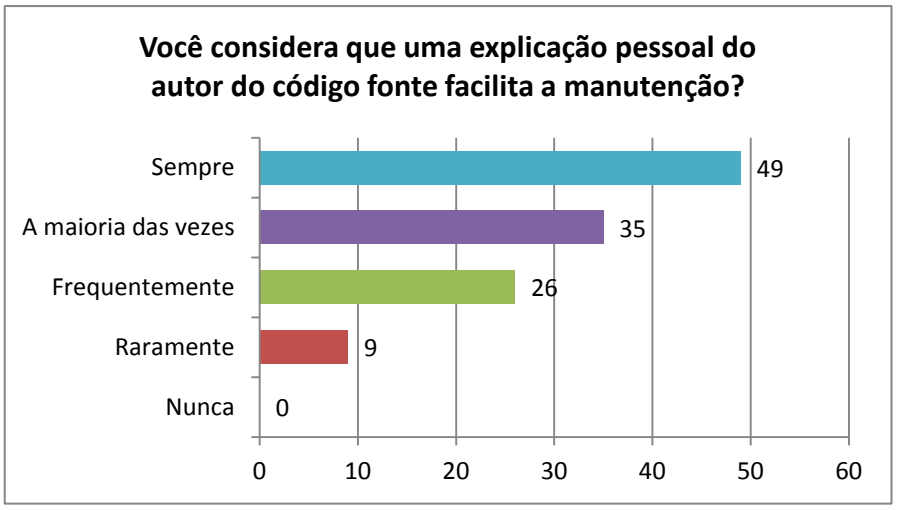

Figura B-14 - Importância da explicação pessoal do próprio autor do código fonte

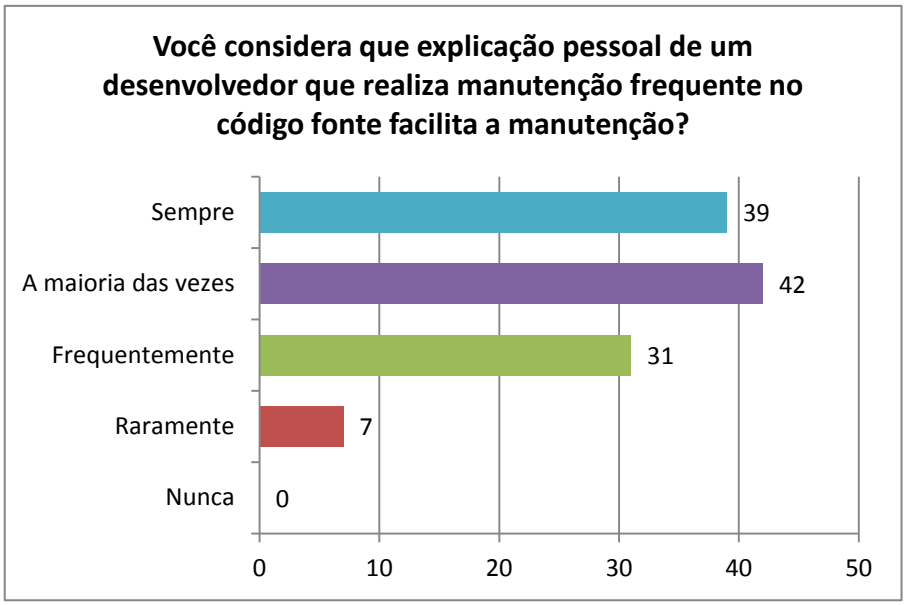

Figura B-15 - Importância da explicação pessoal de alguém que conhece o código fonte

Considerações da Pesquisa de Campo

Com base na pesquisa realizada observou-se um tipo de “fragilidade” na documentação. Apesar de sua importância e seu papel fundamental na engenharia de software, sua utilização, assim como os benefícios ganhos em virtude de sua usabilidade, não desencadeia uma melhor eficiência na atividade de compreensão.

Sabe-se que é fundamental documentar e manter estes artefatos atualizados. No entanto, a manutenção, em especial o entendimento, exige maior concentração e dedicação dos programadores, o que a torna um pouco menos essencial para esta etapa do processo de software.

Com base nos resultados obtidos, observou-se uma certa convicção quanto ao compartilhamento de conhecimento. É fundamental e crucial para o sucesso da manutenção ter um repasse inicial de conhecimento, e não realizar o processo de entendimento e levantamento a partir do início. Ter uma explicação de alguém que detém um conhecimento ajuda significativamente na tarefa de manutenção, pois o sensemaking não ocorre no início, mas sim, a partir de um ponto de partida já estruturado.

Conforme já estudado anteriormente, o conhecimento está em constante transformação e nas cabeças das pessoas (COCCO et al., 2003). No entanto, o

repasse nem sempre pode ser viável, por questões estratégicas, de negócio, logística ou financeiras.

Desta forma, este trabalho tem como fundamento a necessidade do mercado em melhorar e aperfeiçoar o processo de compreensão da manutenção de software através da extração, desenvolvimento e compartilhamento do domínio do conhecimento com o código fonte, de forma a transferi-lo da pessoa que domina o código fonte para uma base de conhecimento e, consequentemente, a realização de repasses de forma automática para aqueles que estão iniciando a manutenção.