Revelando Extensões de Conceitos no Código Fonte

Assis S. Vieira[1], **Bruno C. da Silva**[2], **Cláudio Sant'Anna**[1]

[1]Instituto de Matemática – Universidade Federal da Bahia (UFBA)
Cep 40170-110 – Salvador, BA – Brasil

[2]Universidade Salvador (UNIFACS)
Cep 41770-235 – Salvador, BA – Brasil

assis.sv@gmail.com, bruno.carreiro@pro.unifacs.br, santanna@dcc.ufba.br

Resumo. *Na modificação de software, a Localização de Conceitos é a etapa em que o desenvolvedor localiza no código-fonte os trechos de código que devem ser modificados. Nesta etapa, um método frequentemente usado pelos desenvolvedores é o CLDS – Concept Location by Dependence Search. As atuais técnicas que se baseiam neste método deixam a cargo do desenvolvedor a missão de identificar e delimitar os fluxos de controle e de dados presentes em um software. Tal missão, consome tempo e esforço do desenvolvedor. Para tratar este problema, desenvolvemos parcialmente um modelo heurístico que identifica os diferentes fluxos e seus elementos mais significativos. Com base neste modelo, propomos uma nova técnica para o método CLDS.*

1. Introdução

A localização de conceitos faz parte do processo de compreensão de software [Biggerstaff et al. 1993, Rajlich and Wilde 2002]. Esta tarefa é responsável em média por 62% do esforço e por 70% do tempo dos engenheiros durante uma atividade de modificação de software [Soh et al. 2013, Minelli et al. 2015].

Nas últimas décadas, pesquisadores têm desenvolvido modelos cognitivos para tentar explicar o processo de compreensão de software realizado por programadores [Hansen et al. 2012, Mayrhauser and Vans 1995]. Nesse contexto, Rajlich introduz na compreensão de software a noção de *conceito*, utilizado pela linguística e outras áreas do conhecimento [Rajlich and Wilde 2002, Rajlich 2009, Rajlich 2011]. Cada conceito é constituído por *nome*, *intenção* e *extensão*: o nome identifica o conceito; a intenção é uma sugestão de significado; e a extensão de um conceito é qualquer coisa no mundo real que reflete a intenção deste conceito. Para os autores, no contexto da engenharia de software as extensões no código-fonte podem ganhar diferentes formas. Em um código orientado a objeto, por exemplo, uma extensão pode ganhar a forma de um pacote, classe, método ou um conjunto destes. Assim, eles apontam que a localização de conceitos pode ser vista como um processo de localizar no código-fonte a extensão dos conceitos que se tem interesse.

Durante a leitura do código-fonte não há nada que indique ao programador que ele está passando de um conceito para outro, a não ser os nomes usados nos identificadores de classes e métodos. Entretanto, nem sempre os nomes usados são significativos o suficiente para tal tarefa [Eaddy et al. 2007]. Rajlich [2009] relata que um dos fatores para a dificuldade na recuperação do significado (intenção), considerando apenas um nome, é a

natural relação de muitos-para-muitos entre nome, intenção e extensão que constituem os conceitos. Exemplificando: dado isoladamente o identificador do método *dividir()*, não é possível distinguir se sua intenção refere-se a "dividir uma estrutura de dados para um algoritmo de busca" ou "inserir um divisor de ícones na barra de ferramentas de interface gráfica".

Portanto, para recuperar apropriadamente a intenção de um conceito é preciso antes identificar o contexto no qual ele está inserido. Considerando que o contexto de muitos conceitos que constituem um software encontra-se no próprio código-fonte, temos a seguinte questão de pesquisa: **como identificar os limites entre a extensão de um determinado conceito e as extensões dos demais conceitos que compreendem seu contexto?**

É fundamental conhecer os limites das extensões que consistem os conceitos que devem ser modificados, caso contrário, a modificação pode invadir inapropriadamente as extensões dos conceitos alheios à necessidade de modificação. Além disso, na localização de conceitos, conhecer tais limites é importante para concluir se uma extensão específica reflete de fato as propriedades contidas na intenção, ou se a localização de parte dessa extensão é apenas uma coincidência, tal como foi dado no exemplo do método *dividir()*. Considera-se que estes são alguns dos fatores que fazem os desenvolvedores se esforçarem tanto na recuperação do propósito dos elementos de código, elevando assim os custos de manutenção [Sillito et al. 2005, LaToza et al. 2006, Roehm et al. 2012].

Pesquisas sobre o processo cognitivo realizado por programadores durante atividades de compreensão de software apontam que, durante a leitura de um código-fonte desconhecido os programadores constantemente tentam compreendê-lo identificando os fluxos de controle e de dados [Mayrhauser and Vans 1995]. A nossa hipótese é de que os significados dos fluxos são mais facilmente alcançados pela leitura dos elementos de código que dão início a esses fluxos. Tais elementos tendem estar mais próximos do contexto do domínio da aplicação e dos termos conhecidos pelo desenvolvedor do que os demais elementos. Portanto, os demais elementos que compõe os fluxos podem ser inicialmente ignorados pelo desenvolvedor até que, em um momento posterior, a compreensão destes elementos periféricos se torne necessária.

Um método que apoia os desenvolvedores durante a leitura de um código-fonte é o CLDS (*Concept Location by Dependence Search*) [Chen and Rajlich 2000, Chen and Rajlich 2010]. As ferramentas que dão suporte a este método podem ser encontradas em ambientes de desenvolvimento, tal como o Eclipse[1]. Uma técnica específica para o CLDS também já foi proposta, o Ripples [Chen and Rajich 2001]. Contudo, não foi encontrada nenhuma abordagem que informa qual elemento origina os fluxos nos quais um específico elemento participa. Deste modo, as atuais técnicas deixam a cargo do desenvolvedor a missão de conhecer os limites dos diferentes fluxos de um código-fonte.

Neste cenário, propomos uma nova técnica que possibilita ao desenvolvedor conhecer os limites entre os fluxos de controle presentes no código-fonte, priorizando e destacando os elementos mais significativos de cada fluxo. Para verificar a viabilidade de tal proposta, apresentamos também como tal técnica pode ser aplicada em um sistema real.

[1] Ambiente de Desenvolvimento Integrado usado pela comunidade Java: http://www.eclipse.org.

Assim, para melhor compreensão, o presente artigo segue estruturado em quatro seções. A Seção 2 apresenta métodos e técnicas existentes de localização de conceitos. A Seção 3 descreve em alto nível a técnica proposta, destacando a diferença entre ela e as atuais técnicas. A Seção 4 demonstra a aplicação deste modelo em um software específico, o JHotDraw 7.6[2]. Por fim, na Seção 5, resume-se o que tem sido feito até o momento e os próximos passos para a continuidade da pesquisa.

2. Métodos e Técnicas de Localização de Conceitos

Um método de localização de conceitos amplamente conhecido é o *Concept Location by Dependence Search* (CLDS) [Chen and Rajlich 2000, Chen and Rajlich 2010]. Neste método os desenvolvedores leem o código-fonte e seguem as dependências estáticas que julgarem relevantes. Os ambientes de desenvolvimento têm fornecido recursos para auxiliar o desenvolvedor nesse sentido, tal como a funcionalidade de *Call Hierarchy* do Eclipse, além de *Plugins* como o Ripples [Chen and Rajich 2001]. Entretanto, essas iniciativas ainda não conseguem resolver o problema de compreensão das extensões dos conceitos no código-fonte.

Adicionalmente, outras técnicas baseadas em análise estática também foram propostas, porém não há evidências de que elas tenham mitigado o problema de compreensão de conceitos no código-fonte. No estudo realizado por De Alwis et al. [2007], os autores reportaram que durante as tarefas de modificação de software, não houve diferença significativa nos esforços dos desenvolvedores ao usar três ferramentas (JQuery, Ferret, Suade) que apoiam diferentes técnicas de localização de conceitos e as ferramentas convencionais disponibilizadas pelo ambiente de desenvolvimento Eclipse. Entre outras hipóteses, os autores explicaram que esse resultado pode ter ocorrido porque as técnicas testadas simplesmente re-empacotavam as informações, disponibilizadas pela IDE (*Integrated Development Environment*), em novas formas em vez de fornecer novos meios de resolver os problemas de compreensão. Esta hipótese converge com as observações de Sillito et al. [2005] e Ko et al. [2006] em outros estudos exploratórios em que os participantes utilizavam uma IDE amplamente conhecida.

No estudo realizado por Sillito et al. [2005], os autores observaram que, um modelo integrado que colocasse em evidência as relações entre as entidades identificadas como relevantes seria de grande valia. Os autores ainda destacaram que, mesmo quando lápis e papel foram usados, a integração ainda apresentou-se como um importante e difícil desafio. Os autores também puderam observar que as re-visitas de entidades e relações foram comuns, mas nem sempre simples. De fato, 57% das visitas observadas para os elementos de código foram re-visitas e esta proporção aumentava ao longo do tempo.

No estudo realizado por Ko et al. [2006], os pesquisadores relataram que as ferramentas do Eclipse utilizadas para a navegação do código causaram uma sobrecarga nos desenvolvedores. De acordo com a razão apresentada, sempre que uma dependência era analisada, o resultado da dependência anterior era perdido, forçando os desenvolvedores a repetir a busca. Os autores apontaram que estes problemas levaram os desenvolvedores a gastarem em média 35% do tempo em mecanismos de navegação.

[2]Framework para aplicações gráficas escritas em Java: https://sourceforge.net/projects/jhotdraw/.

3. Uma Nova Técnica para Localização de Conceitos

Neste cenário, propomos uma nova técnica baseada no método CLDS que apoia o desenvolvedor durante a localização de conceitos no código-fonte. Nesta técnica, um elemento de código inicial, chamado de *semente*, é selecionado pelo desenvolvedor. Uma ferramenta, por sua vez, apresenta um grafo de elementos relacionados à semente selecionada. Entretanto, diferente das atuais técnicas baseadas no CLDS, a referida proposta parte da perspectiva de que cada um dos elementos de código representa pequenos conceitos que se relacionam estaticamente para compor outros conceitos mais abstratos. A premissa é de que, dado um fluxo de controle ou de dados, o conceito mais significativo deste fluxo é representado pelo elemento de código que o inicia. Os demais elementos do fluxo são portanto coadjuvantes, pois possuem importância secundária na compreensão do fluxo. Os elementos apresentados pela ferramenta são então organizados em superconceitos e subconceitos da semente selecionada.

Os superconceitos da semente são os elementos que dão origem aos fluxos em que a semente participa. A semente tem um único superconceito quando ela é referenciada por um único elemento, ou quando ela é referenciada por dois ou mais elementos e estes forem referenciados, direta ou indiretamente, por um único elemento em comum. A semente pode ter dois ou mais superconceitos quando os elementos que a referenciam não possuem um elemento em comum que os referencie. Logo, os elementos que originam os distintos fluxos são os superconceitos da semente. Os subconceitos da semente são todos os elementos que, para serem alcançados, seja direta ou indiretamente, é necessário passar pela semente. Isso significa que a semente é o superconceito destes elementos.

A Figura 1 ilustra um possível grafo aplicando essa abordagem em um sistema real. Neste cenário, o desenvolvedor está tentando compreender o método *ApplicationModel.createActionMap(Application, View)* da biblioteca gráfica JHotDraw. Ao selecionar este método como semente, a técnica apresenta o grafo ilustrado pela referida figura.

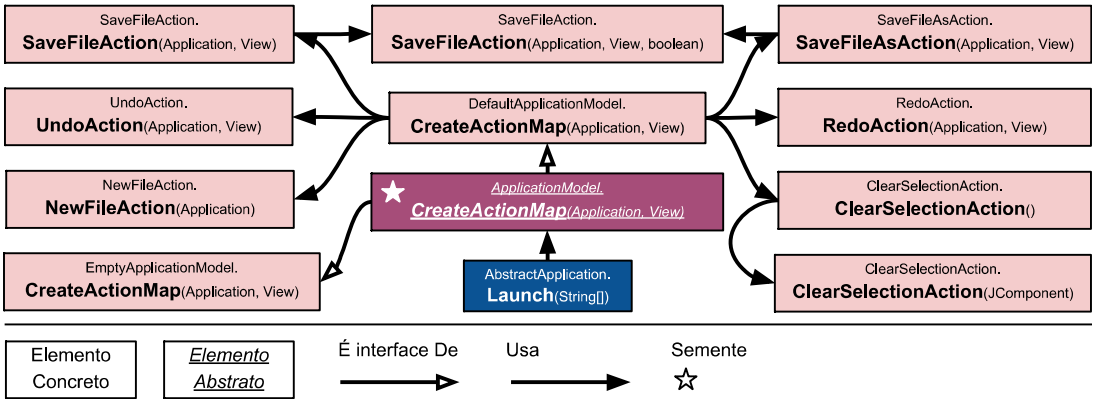

Figura 1. Grafo de conceitos apresentado ao desenvolvedor.

Neste grafo, todos os vértices são métodos. O elemento com uma estrela é a semente selecionada. O elemento logo abaixo da semente é o seu superconceito. Os demais são os subconceitos da semente. Este superconceito é o elemento que origina todo o fluxo no qual a semente participa. Coincidentemente, ele referencia a semente de modo indireto, através de outros 7 elementos que não são apresentados neste grafo (veja a Seção 4 para mais detalhes). Neste primeiro momento é possível que o desenvolvedor

conclua que a semente selecionada participa exclusivamente da inicialização da aplicação gráfica, uma vez que o identificador do superconceito - *AbstractApplication.launch(..)* - remete a isso. Porém, na utilização das demais técnicas da CLDS, no mínimo, vinte e uma dependências devem ser visitadas para chegar a esta mesma conclusão, interagindo com diversos mecanismos de interface da IDE. Estas dependências podem ser conferidas em detalhes na Seção 4.

Para implementar a técnica proposta, iniciamos o desenvolvimento de um modelo heurístico com um conjunto de regras que definem a delimitação entre diferentes extensões presentes no código-fonte. Considerando as restrições de espaço, apresenta-se a seguir algumas das regras desenvolvidas até o presente momento, omitindo a descrição detalhada do procedimento de aplicação. As regras omitidas referem-se principalmente as questões como dependências circulares. Contudo, as regras apresentadas são suficientes para compreender os resultados obtidos na aplicação deste modelo no *JHotDraw 7.6* (Seção 4).

1. **Regra da Composição:** a extensão de um conceito representado por um elemento de código *A* se estende unicamente aos elementos que são usados exclusivamente para defini-lo. Os demais elementos de código referenciados por *A*, porém de modo não exclusivo, não compõem a extensão do conceito composto por *A*. Por exemplo, se um método *m1()* compõe a extensão de um conceito C1, todos os elementos de código referenciados de modo exclusivo por *m1()* também farão parte da extensão do conceito C1. Os demais elementos referenciados por *m1()*, mas que também são referenciados por outros elementos de código não fazem parte da sua extensão.
2. **Regra da Referência Implícita:** quando um método concreto é invocado polimorficamente, o fluxo de controle passa do método chamador para o método abstrato, e este, por sua vez, decide então qual das suas especializações executar. Deste modo, em vez de considerar a relação direta do método chamador para o método concreto, considera-se primeiramente que, o método chamador faz referência ao método abstrato, e este, então, a uma referência implícita ao específico método concreto, como se o método abstrato possuísse uma implementação e nela tivesse referências de todas as suas especializações (métodos concretos). Assim, em tempo de execução, o método abstrato decide pelo método concreto que deve ser executado.

Algumas técnicas de localização de conceitos utilizam outras abordagens para organizar o código-fonte em superconceitos e subconceitos [Dit et al. 2013]. Tais técnicas se fundamentam na AFC - Análise Formal de Conceitos. A AFC é um modelo matemático que formaliza a noção de conceito através da teoria dos conjuntos [Wille 1992]. Diferente do modelo que está sendo proposto, o grafo de conceitos resultante da AFC tende a ser mais complexo do que o grafo de dependências estáticas fornecido como entrada, dificultando a compreensão dos conceitos presentes no código-fonte [Anquetil 2000].

4. Aplicação da Técnica no JHotDraw

Para verificar a viabilidade do modelo foi aplicado manualmente as regras mencionadas em parte do código-fonte do *JHotDraw 7.6*, que é um projeto real de código-fonte aberto. No escopo dessa avaliação, foram considerados apenas os elementos

de código do tipo método. A semente selecionada foi o método *ApplicationModel.createActionMap(Application, View)*. A escolha desta semente teve como critério métodos em que fosse possível exercitar as regras apresentadas. Ao examinar a semente, aplicou-se a *Regra da Referência Implícita*, uma vez que este era um método abstrato com duas especializações (métodos concretos). Em seguida, foi descoberto que este método abstrato era a única referência para seus dois métodos concretos. Logo, aplicou-se a *Regra de Composição* nos três elementos, resultando na relação superconceito e subconceito, respectivamente entre o método abstrato e suas especializações. Deste modo, foi dada continuidade a exploração do código até que todos os elementos da extensão do conceito, representado pela semente, foram encontrados. O resultado desta investigação pode ser conferido na Figura 2. O elemento marcado com uma estrela é a semente selecionada, enquanto os demais são os subconceitos da semente.

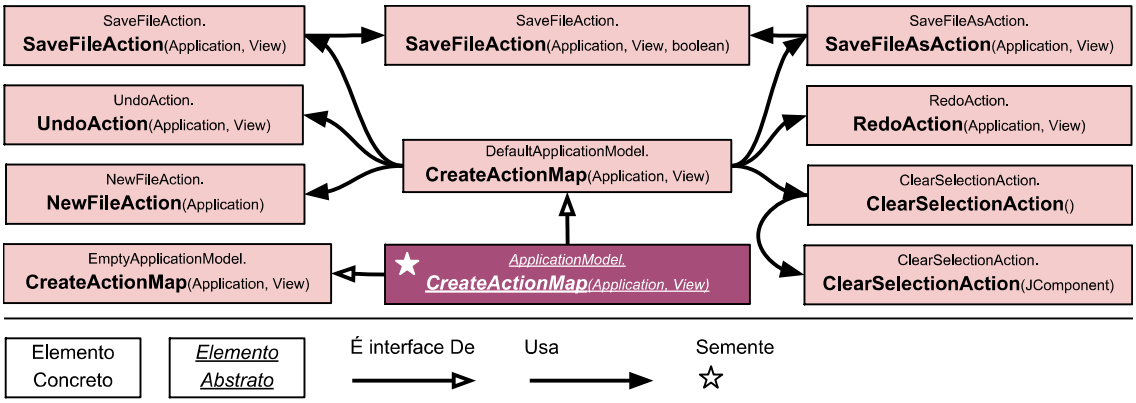

Figura 2. Todos os subconceitos da semente.

Por outro lado, ao navegar pelos métodos que referenciavam diretamente a semente, foram descobertos outros sete métodos, conforme ilustra a Figura 3. Com base na Regra de Composição, pôde-se concluir que a semente e o conjunto dos novos métodos descobertos delimitavam conceitos distintos. Em outras palavras, apesar de ter sido referenciada, a semente não era subconceito de nenhum destes sete elementos. Neste sentido, deu-se prosseguimento a exploração do código até que o método *AbstractApplication.Launch(String[])* foi descoberto sendo o superconceito da semente. Este método originava todos os fluxos em que a semente participava. O resultado desta inspeção pode ser conferido no grafo da Figura 3. O elemento com uma estrela representa a semente, enquanto que o elemento no canto inferior direito do grafo representa o superconceito da semente. Os demais elementos interligam a semente ao seu superconceito. Esses elementos, incluindo a semente, são alguns dos subconceitos do referido superconceito.

5. Conclusão

Embora diversas técnicas de localização de conceitos tenham sido propostas nas últimas décadas, o processo de localizar a extensão de conceitos no código-fonte ainda parece consumir mais da metade do esforço e tempo dos desenvolvedores em tarefas de modificação de código-fonte. Trabalhos sobre modelos cognitivos apontam que desenvolvedores compreendem software desconhecido através da identificação dos fluxos de controle e de dados no código-fonte. Baseado no ponto de vista da programação como um processo de tradução, este trabalho assume a premissa de que os elementos que iniciam um fluxo de

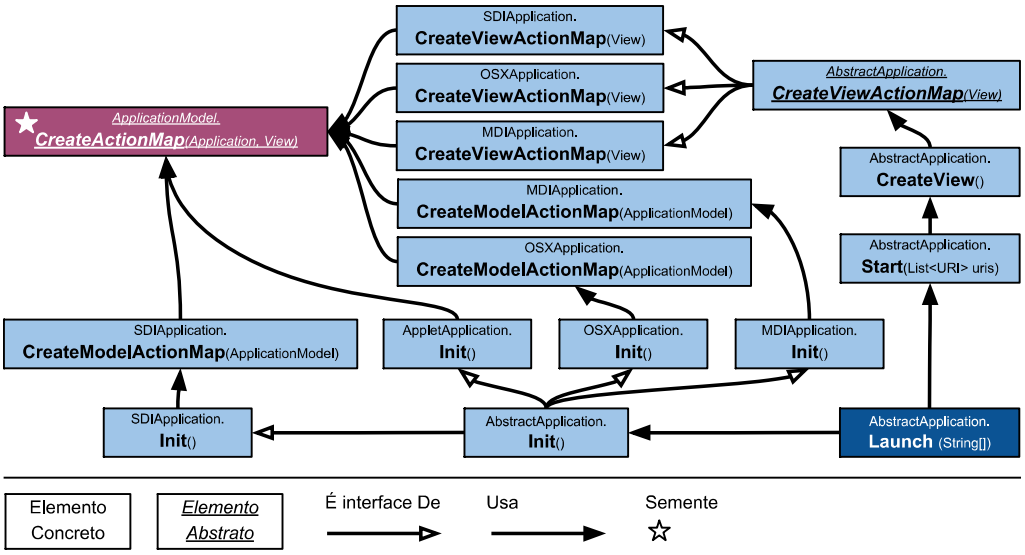

Figura 3. Os elementos que conectam a semente ao seu superconceito.

controle são mais significativos para o desenvolvedor do que os demais elementos deste fluxo. Desta forma, propõe-se uma nova técnica de localização de conceitos baseada no método CLDS. Nesta nova abordagem, o papel do desenvolvedor é selecionar uma semente. Enquanto que, diferente das demais técnicas, nesta proposta o papel da técnica é apresentar os subconceitos e superconceitos da semente. Nesta apresentação, a técnica destaca os limites entre a extensão do conceito representado pela semente e as extensões dos conceitos que compõe o contexto da semente. Para delimitar dos conceitos presentes no código-fonte, iniciamos o desenvolvimento de um modelo heurístico. Apesar de estar em fase inicial, o modelo demonstrou-se viável após sua aplicação em partes do JHotDraw.

Assim, as próximas atividades para o desenvolvimento dessa pesquisa consistem em: (i) concluir o modelo heurístico que identifica os fluxos de controle no código-fonte; (ii) concluir o desenvolvimento da ferramenta de apoio à técnica proposta; e (iii) avaliar a aplicabilidade da técnica em um experimento sobre localização de conceitos.

Agradecimentos. Esse trabalho foi apoiado pela FAPESB e pelo CNPq por meio do Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (processo 573964/2008–4) e Projeto Universal (processo 486662/2013-6).

Referências

Anquetil, N. (2000). A comparison of graphs of concept for reverse engineering. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 231–240.

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498, Los Alamitos, CA, USA. IEEE Computer Society Press.

Chen, K. and Rajich, V. (2001). Ripples: tool for change in legacy software. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 230–239.

Chen, K. and Rajlich, V. (2000). Case study of feature location using dependence graph. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 241–247.

Chen, K. and Rajlich, V. (2010). Case study of feature location using dependence graph, after 10 years. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 1–3.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

Eaddy, M., Aho, A., and Murphy, G. C. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM '07. First International Workshop on*, page 2.

Hansen, M. E., Lumsdaine, A., and Goldstone, R. L. (2012). Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 27–38, New York, NY, USA. ACM.

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA. ACM.

Mayrhauser, A. V. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.

Minelli, R., Mocci, A., and Lanza, M. (2015). I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35.

Rajlich, V. (2009). Intensions are a key to program comprehension. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 1–9.

Rajlich, V. (2011). *Software Engineering: The Current Practice*. Chapman & Hall/CRC, 1st edition.

Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278.

Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012). How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265.

Sillito, J., Voider, K. D., Fisher, B., and Murphy, G. (2005). Managing software change tasks: an exploratory study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–.

Soh, Z., Khomh, F., Guéhéneuc, Y. G., and Antoniol, G. (2013). Towards understanding how developers spend their effort during maintenance activities. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 152–161.

Wille, R. (1992). Concept lattices and conceptual knowledge systems. *Computers Mathematics with Applications*, 23(6):493 – 515.