Polimorphic View: visualizando o uso de polimorfismo em projetos de software

Fabio Petrillo[1], **Guilherme Lacerda**[12], **Marcelo Pimenta**[1] **e Carla Freitas**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]Faculdade de Informática - Centro Universitário Ritter dos Reis (Uniritter)
Rua Orfanotrófio, 555 – 90840-440 – Porto Alegre – RS – Brazil

{fspetrillo,gslacerda,mpimenta,carla}@inf.ufrgs.br

Abstract. *Polymorphism is a powerful and flexible programming mechanism, being one of the key concepts for object-oriented software design and development. Nevertheless, polymorphism has a dark side: if it is used improperly, it can ruin application understandability. Despite its importance, very little attention has been paid to how it is used in software projects, especially because it is very difficult to find and visualize where - in source code or specifications - polymorphism is adopted. In this paper, we discuss how polymorphism is used and propose an approach to explicit the polymorphism usage by representing the software as a static call graph.*

1. Introdução

O processo de manutenção de software é uma atividade complexa [Lange and Nakamura 1997] e a maior parte do tempo gasto neste processo é dedicado à compreensão do sistema a ser modificado [Caserta and Zendra 2010]. Compreender as dependências entre os diferentes componentes de um software é uma das tarefas mais importantes e desafiadoras em engenharia de software [Hoogendorp 2010]. Consequentemente, a criação de um projeto claro e compreensível, a fim de apoiar a manutenção de sistemas orientados a objetos (OO) é uma preocupação vital para a indústria de software [Mihancea 2009].

Neste contexto, o polimorfismo é um conceito chave na programação OO[Mihancea 2009] [Ponder and Bush 1994], sendo, provavelmente, o seu mecanismo mais poderoso e flexível[Ambler 1995] [Booch et al. 2007]. Polimorfismo oferece benefícios como maior extensibilidade e reutilização [Benlarbi and Melo 1999]. Por consequência, a caracterização dos projetos de software com respeito à maneira como eles usam o polimorfismo é importante tanto para a compreensão de software quanto na avaliação da sua qualidade [Mihancea 2009]. Entretanto, apesar de importante, pouco se sabe sobre como polimorfismo é usado em projetos de software, especialmente porque é difícil de encontrar e visualizar sua adoção.

Este trabalho tem como objetivo apresentar uma abordagem para a investigação do uso de polimorfismo em projetos de software. Em particular, este artigo aborda os efeitos na invocação de métodos, destacando chamadas polimórficas através da análise estática de software, mostrando a maneira pelo qual os clientes de uma classe com métodos abstratos fazem uso de polimorfismo. Para alcançar este objetivo, foram analisados dois

projetos desenvolvidos em Java, utilizando uma visualização que torna explícito o uso de polimorfismo através de um grafo de chamadas estáticas, denominado **Polymorphic View**.

Este artigo está organizado da seguinte forma: inicialmente, a seção 2 faz-se uma breve contextualização do problema e a apresenta a visualização proposta. A seção 3 descreve a análise de dois projetos desenvolvidos em Java e a seção 4 apresenta os trabalhos relacionados. Finalmente, a seção5 sumariza as contribuições e descreve algumas possibilidades de trabalhos futuros.

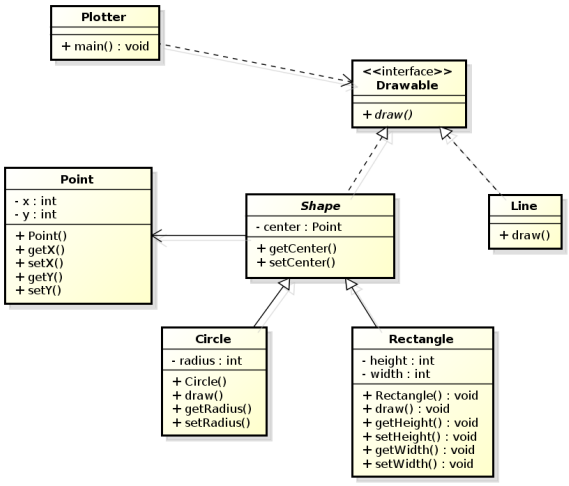

Figure 1. Um exemplo de polimorfismo

2. Visualizando o Polimorfismo

Esta seção apresenta uma abordagem para visualização do uso de polimorfismo. Inicialmente, contextualizamos o problema, com a descrição de um exemplo, com diagramas da UML. Em seguida, descrevemos a representação proposta.

2.1. Contextualizando o problema

Um exemplo típico de polimorfismo é ilustrado no diagrama de classes da Figura 1. Esse diagrama é uma adaptação do modelo apresentado por Booch et al. [Booch et al. 2007].

Nesse exemplo, a interface *Drawable* é implementada pela classe abstrata *Shape* e pela classe concreta *Line*. *Shape* tem duas subclasses: *Circle* e *Rectangle*. Consequentemente, em um programa Java, *Line*, *Circle* e *Rectangle* implementam um método chamado *draw()*.

A classe *Plotter* é um cliente das implementações de *Drawable*, instanciando objetos do tipo *Drawable* e fazendo uma **invocação polimórfica**, chamando o método *draw()*.

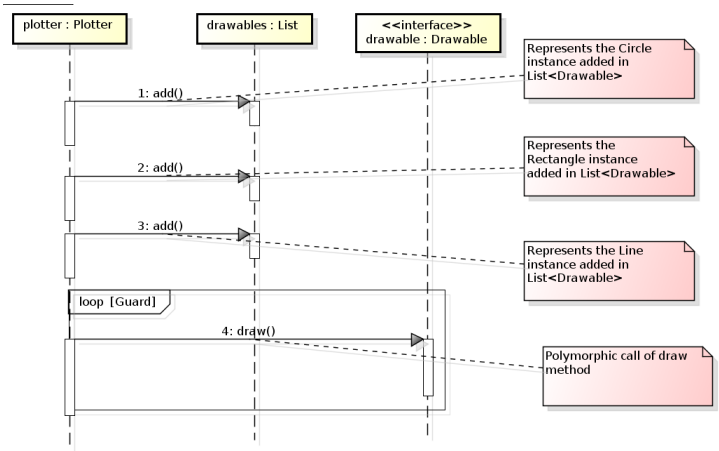

Figure 2. Diagrama de sequência para o exemplo de polimorfismo

Em UML, o diagrama de sequência é usado para descrever um conjunto de interações entre objetos [Dutoit and Bruegge 2010]. No diagrama de sequência, para explicitar o polimorfismo, é necessário usar **vários diagramas**, sendo um para apresentar a mensagem polimórfica para as classes abstratas e outros diagramas para

detalhar o uso do polimorfismo, iniciando com a mensagem polimórfica [Larman 2004]. Outra possibilidade é o uso comentários ou anotações no diagrama, como pode ser visto na Figura 2. Nesse exemplo, as instâncias de *Drawable* não são explicitamente representadas e a invocação polimórfica do método *draw()* pode não ser claramente observada, necessitando de comentários para esclarecer o uso do polimorfismo.

Portanto, mesmo em um exemplo simples, não é trivial visualizar invocações polimórficas utilizando diagramas da UML. Finalmente, três artefatos (código, diagramas de classe e de sequência) são necessários para visualizar e compreender o uso de polimorfismo. Com o objetivo de tratar essas questões, propomos uma abordagem para explicitar o uso de polimorfismo em um só diagrama através de uma representação em grafo de chamadas estáticas.

2.2. Polymorphic View

O Polymorphic View é uma representação para modelagem de software baseado em grafo orientado de chamadas [Grove et al. 1997] para explicitar o uso de polimorfismo. Ele consiste nos seguintes elementos básicos:

Tipos: representado por um retângulo com bordas arredondadas (Figura 3), usado para representar classes e tipos. Tipos concretos (ou instanciáveis) são representados como retângulos com linhas sólidas, enquanto tipos abstratos e interfaces (não instanciáveis) são representados com linhas tracejadas. Como na UML, os tipos abstratos tem seu rótulo em itálico. Somente tipos que tem código fonte disponível são representados, pertencentes ao domínio da aplicação.

Figure 3. Tipos Concreto, Abstrato e Interface

Invocações: são representadas por linhas sólidas com setas. Já as invocações polimórficas são representados por linhas tracejadas. Os dois tipos de invocações são ilustrados na Figura4.

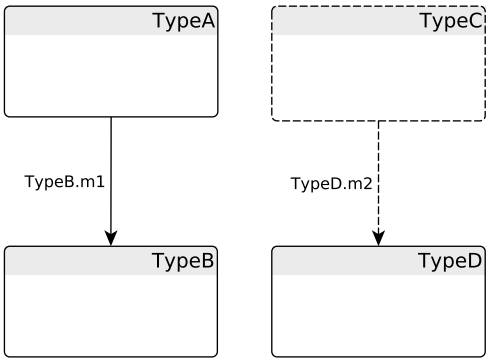

Figure 4. Invocações

Namespace: *namespace* é um conteîner para tipos organizados em diretórios. Um *namespace* é representado por um retângulo com bordas arredondadas e linhas pontilhadas. Em Java, *namespaces* correspondem aos *packages*.

Usando esta notação, o Polymorphic View é elaborado a partir da análise estática do código. Como pode ser observado pela Figura 5, a interface *Drawable* é explicitamente representada como um tipo abstrato (borda tracejada) e as invocações polimórficas (chamadas pelo método *draw*) são representadas por linhas tracejadas para cada implementação de *Drawable* (tipos *Rectangle*, *Circle* e *Line*). Através dessa visulização, é possível observar que a invocação do método *draw* pela classe *Plotter* é representada por uma linha sólida, uma vez que é, de fato, uma invocação do método implementado nos tipos concretos *Rectangle*, *Circle* e *Line*.

Neste sentido, o **Polymorphic View permite com apenas um diagrama observar o relacionamento entre tipos polimórficos e seus clientes** (*Plotter*, por exemplo), bem como o relacionamento entre os tipos polimórficos e outras classes que implementam seus métodos. Nossa abordagem permite também a análise do uso de polimorfismo a partir de uma **perspectiva arquitetural**, destacando relacionamentos que estão tipicamente escondidos.

Para gerar o Polimorphic View, usamos uma ferramenta chamada Software Pathfinder, desenvolvida pelo nosso grupo de pesquisa. Na próxima seção, é demonstrado o uso do Polymorphic View em dois projetos Java.

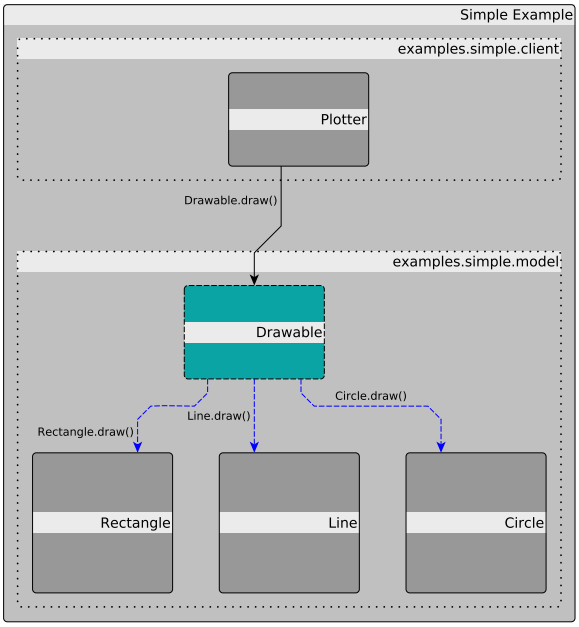

Figure 5. Polymorfic View para a Fig. 1

3. Entendendo o Polimorfismo

Nesta seção, apresentamos como o Polymorphic View pode auxiliar na compreensão do uso de polimorfismo. Para este propósito, foram analisados dois projetos de código aberto: JUnit e FindBugs. A análise foi desenvolvida através dos seguintes passos:

1. Extrair os dados dos projetos, através da análise estática feita pelo Software Pathfinder;
2. Pesquisar todas as classes abstratas e interfaces;
3. Filtrar os tipos usando seguintes as regras:
   - *Profundidade da hierarquia* $(PH) >= 1$;
   - *Número de filhos* $(NDF) >= 1$;
   - *Número de métodos abstratos* $(NMA) >= 1$;
   - ter pelo menos um cliente;
4. Construir o Polymorfic View para cada tipo filtrado;
5. Analisar o uso do polimorfismo através da visualização;
6. Complementar a análise com o código fonte.

Nas próximas seções iremos apresentar alguns exemplos do uso de polimorfismo encontrados nos projetos JUnit e FindBugs. Como critério para selecionar os exemplos, focamos essencialmente em padrões de projeto GOF [Gamma et al. 1995].

3.1. JUnit

JUnit[1] é um *framework* escrito em Java, seguindo a arquitetura *xUnit* para *frameworks* de teste unitário. Após construírmos os Polymorphic Views para o código do JUnit, encontrou-se um caso do padrão ***Composite***, apenas observando a invocação polimórfica de *TestSuite* para *Test*, realizada pelo método *run()*, ilustrado na Figura 6. *Test* é uma interface implementada por *DoubleTestCase*, *RepeatedTest*, *JUnit4TestCaseFacade* e *JUnit4TestAdapter*, mas especialmente por *TestSuite*, caracterizando uma composição. Finalmente, analisando o código, foi confirmado a presença do padrão *Composite*.

Um outro bom exemplo de uso do polimorfismo pode ser observado na Figura 8, no qual um tipo abstrato (*BaseTestRunner*) é usado para evitar uma **dependência cíclica**. O tipo *ResultPrinter* usa *TestListener*, que é uma classe abstrata e *TestRunner* estende *BaseTestRunner*, retornando valores para *ResultPrinter*.

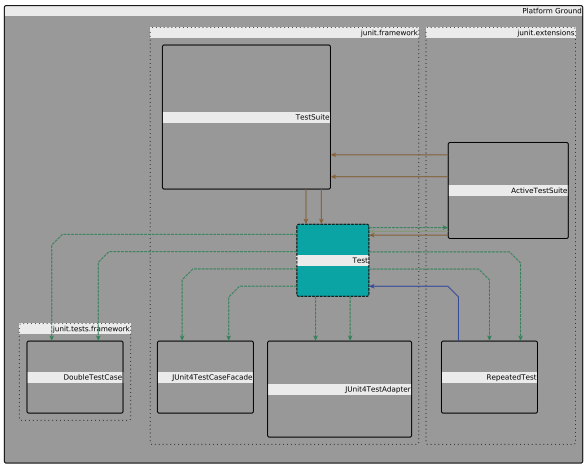

Figure 6. Polymorphic View para interface *Test*

3.2. FindBugs

O FindBugs[2] é um programa que usa análise estática para encontrar problemas conhecidos como *bug patterns*: instâncias de código que provavelmente são erros em projetos Java.

Analisando o Polymorphic Views para este projeto, foi encontrado um exemplo de **injeção de dependência** que pode ser observado na Figura 7. A classe *SourceFile* define uma estrutura de *cache* para arquivos que podem ser manipulados. Para isto, é usado invocações polimórficas para *ZipSourceFileDataSource* e *FileSourceDataSource* através da interface *SourceFileDataSource*.

3.3. Discussão

A análise desses dois projetos de código aberto mostrou que o Polymorphic View auxilia na localização de estruturas polimórficas, que geralmente são escondidas em outras visualizações. Usando esta abordagem, inferências complexas sobre os projetos podem ser feitas, ajudando a entender e encontrar padrões, que provavelmente exigiriam uma busca exaustiva no código. O Polymorphic View fornece uma perspectiva estrutural (estática - *namespaces* e tipos) e comportamental (dinâmica - invocações) em um único diagrama, com a exibição das invocações polimórficas.

[1]Mais informações em http://www.junit.org
[2]Mais informações em http://findbugs.sourceforge.net

Ao destacar os tipos polimórficos e suas chamadas em uma única representação, o Polymorphic View ajuda na compreensão de padrões arquiteturais. Para que o mesmo estudo fosse feito somente utilizando diagramas da UML, seria necessário, pelo menos, usar um diagrama de classe e vários diagramas de sequência, um para cada caso de polimorfismo, conforme discutido na seção 2.

4. Trabalhos Relacionados

Este trabalho encontra-se dentro do campo da análise de polimorfismo e reconstrução da arquitetura. Está relacionado também com os estudos empíricos que investigam a capacidade de uma visualização em apoiar a análise arquitetural polimórfica. Vários autores realizaram trabalhos sobre sobe polimorfismo e esta seção apresenta algumas das suas principais contribuições.

Benlarbi e Melo [Benlarbi and Melo 1999] investigaram o impacto do polimorfismo sobre a qualidade do design OO, propondo um conjunto de métricas para projetos. Inicialmente, eles concluíram que o polimorfismo pode aumentar a probabilidade de falhas em software OO e indicaram que ele deve ser usado com precaução pelos projetistas e desenvolvedores. Eles sugeriram que o polimorfismo tende a ocorrer mais em projetos com elevado número de métodos e árvores de herança profundas.

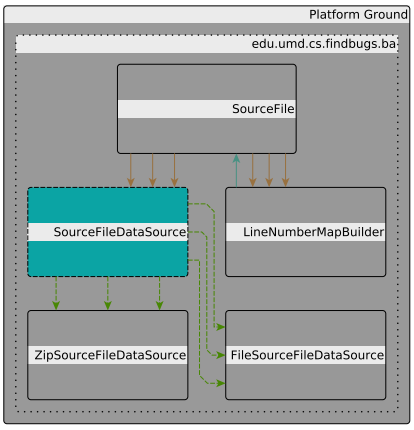

Figure 7. Polymorphic View para a classe *SourceFile* e relacionamentos

Denier e Gueheneuc [Denier and Gueheneuc 2008] propuseram um modelo de herança para ajudar a compreensão da hierarquia de classes com foco em métodos das classes. Eles também definiram métricas e regras para destacar classes e comportamentos interessantes no que diz respeito à herança. Além disso, o modelo fornece como herança é utilizada em um programa. Denier e Sahraoui [Denier and Sahraoui 2009] propuseram uma visão única e compacta de todas as hierarquias de classe usando um layout personalizado, chamado Sunburst.

Nos últimos anos, os estudos mais importantes sobre a herança e análise polimórficas foram desenvolvidos por Mihancea e Marinescu. Mihancea [Mihancea 2006] introduziu uma abordagem baseada em métricas relacionadas ao grau em que uma classe base foi destinada para reutilização da interface. Através da análise de como os clientes usam a interface da classe base, essas métricas quantificam o grau em que os clientes tratam uniformemente as instâncias dos descendentes da classe base, ao invocar métodos que pertencem a esta interface comum.

Em trabalhos subsequentes [Mihancea 2008] [Mihancea 2009], Mihancea introduziu uma abordagem visual baseada em métricas para capturar a medida em que os clientes de uma hierarquia polimórfica manipulam essa hierarquia. Um vocabulário de padrão visual também foi apresentado de modo a facilitar a comunicação entre analistas.

Todas as abordagens acima descritas, embora valiosas, não são adequadas quando o interesse é fornecer uma visão detalhada de polimorfismo. Este trabalho complementa os estudos com foco em uma visualização que destaca tipos abstratos e seus clientes, permitindo assim que o usuário possa capturar a maneira pela qual os clientes de uma hierarquia de classes fazem uso de polimorfismo.

5. Conclusões e Trabalhos Futuros

Neste trabalho, foi a apresentado uma abordagem de explicitar o uso do polimorfismo, utilizando uma visualização baseada em grafo de chamadas estáticas, denominada Polymorphic View. Polymorphic View é composta por tipos, invocações e *namespaces*, permitindo o entendimento dos relacionamentos tipicamente escondidos em outras visualizações. Destina-se a ajudar na análise do uso de polimorfismo sobre uma perspectiva arquitetônica.

Apesar das vantagens promissoras, algumas limitações surgiram durante a sua utilização nas análises apresentadas na seção 3. Em primeiro lugar, existe uma dificuldade em analisar todos os níveis da hierarquia de tipos, escondidas na representação de tipos. Além disso, em alguns casos, a fim de confirmar as hipóteses sobre os padrões de projeto, foi necessário analisar o código fonte.

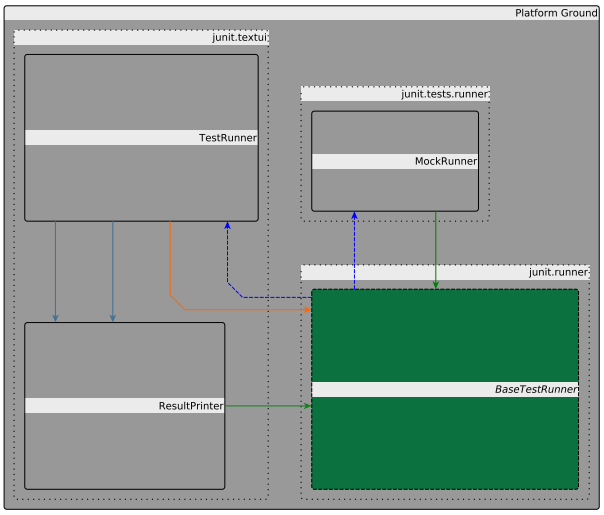

Figure 8. Polymorphic View para classe abstrata *ResultPrinter*

Os resultados obtidos até o momento abrem perspectivas para novas investigações sobre o polimorfismo. Em particular, temos algumas questões em aberto para serem respondidas: a) como os tipos polimórficos se relacionam com seus clientes? b) quais padrões de projeto que adotam polimorfismo são encontrados? c) quais anti-padrões são encontrados? d) há diferenças entre o uso de polimorfismo em Java e em outras linguagens? e) uso do polimorfismo é uma opção deste as primeiras versões de um tipo ou é o resultado de um processo de evolução do software, através de refatorações, por exemplo?

Como parte de trabalhos futuros, pretendemos desenvolver uma visualização interativa, além de acrescentar detalhes como a hierarquia de herança e estruturas internas dos tipos (métodos e atributos). Em outra direção, iremos avaliar o uso do Polymorphic View através de experimentos controlados. Finalmente, iremos investigar porque os arquitetos de software usam polimorfismo em seus projetos, pesquisando as intenções por trás da adoção de estruturas polimórficas.

References

Ambler, A. (1995). Invocation polymorphism. In *Proceedings of Symposium on Visual Languages*, pages 83–90. IEEE Comput. Soc. Press.

Benlarbi, S. and Melo, W. L. (1999). Polymorphism measures for early risk prediction. In *Proceedings of the 21st international conference on Software engineering - ICSE '99*, pages 334–344, New York, New York, USA. ACM Press.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., and Houston, K. A. (2007). *Object Oriented Analysis & Design with Application*. Pearson Education, Boston, MA, 3rd ed edition.

Caserta, P. and Zendra, O. (2010). Visualization of the Static Aspects of Software: A Survey. *IEEE transactions on visualization and computer graphics*, 17(7):913–933.

Denier, S. and Gueheneuc, Y.-G. (2008). Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies. In *2008 16th IEEE International Conference on Program Comprehension*, number 2, pages 143–152. IEEE.

Denier, S. and Sahraoui, H. (2009). Understanding the use of inheritance with visual patterns. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 79–88. IEEE.

Dutoit, A. H. and Bruegge, B. (2010). *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, New York, USA.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call graph construction in object-oriented languages. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, pages 108–124.

Hoogendorp, H. (2010). *Extraction and Visual Exploration of Call Graphs for Large Softeware Systems*. PhD thesis, Groningen University.

Lange, D. and Nakamura, Y. (1997). Object-oriented program tracing and visualization. *Computer*, 30(5):63–70.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e*. Prentice Hall, Upper Saddle River, NJ, USA, 3 edition edition.

Mihancea, P. (2006). Towards a Client Driven Characterization of Class Hierarchies. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 285–294. IEEE.

Mihancea, P. F. (2008). Type Highlighting: A Client-Driven Visual Approach for Class Hierarchies Reengineering. *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 207–216.

Mihancea, P.-f. (2009). *A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment*. PhD thesis, University of Timisoara.

Ponder, C. and Bush, B. (1994). Polymorphism considered harmful. *ACM SIGSOFT Software Engineering Notes*, 19(2):35–37.