ArchGraph: Modularização Automática de Sistemas Usando Clusterização de Grafos de Dependência

Guilherme A. Avelino[1], **Marco Tulio Valente**[1]

[1]Departamento de Ciência da Computação, UFMG, Brasil

{gaa,mtov}@dcc.ufmg.br

Abstract. *A proper modularization is very important to aid in the maintenance and evolution of a system. In order to help to modularize an existing software, this paper presents a strategy for automatic modularization based on data extracted from source code. In our approach, dependencies between software entities are modeled using graphs and, by relying on clustering algorithms, we construct logical groupings of such entities.*

Resumo. *Uma adequada modularização é de grande importância para auxilio na manutenção e evolução de um sistema. Buscando auxiliar a atividade de modularizar um sistema já existente, esse artigo apresenta uma estratégia para geração automática da modularização baseada em dados extraídos de seu código fonte. Em nossa abordagem, dependências entre entidades de um software são modeladas utilizando grafos e, aplicando algoritmos de clusterização, construímos agrupamentos lógicos de tais entidades.*

1. Introdução

Uma boa modularização é essencial para o adequado desenvolvimento e evolução de um software. Um bom projeto modular facilita o gerenciamento do software, permitindo a divisão de atividades, além de limitar o escopo de alterações e melhorar o entendimento do sistema [Parnas 1972]. Entretanto, existem diversos sistemas que não foram desenvolvidos de forma modular, ou ainda sua modularização pode ter sofrido uma degradação durante sua evolução [Lehman et al. 1997]. Para tornar a manutenção de tais sistemas novamente gerenciável é preciso remodularizá-los, agrupando as entidades de software baseados em suas semelhanças e dependências.

Nesse trabalho, apresentamos uma técnica e uma ferramenta de apoio, denominada *ArchGraph*, que extrai as dependências entre classes a partir do código fonte de um sistema Java e representa tais dependências utilizando grafos. Com base no grafo gerado, são usados dois diferentes algoritmos de clusterização, objetivando identificar possíveis agrupamentos lógicos desses artefatos baseados em suas dependências. Os algoritmos utilizados, representam duas diferentes estratégias de clusterização: algoritmos de otimização e algoritmos baseados na teoria dos grafos [Wiggerts 1997], permitindo assim avaliar duas diferentes estratégias, as quais são comparadas no final.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta a solução proposta no artigo, explicando como essa foi construída e detalhando os algoritmos utilizados. A Seção 3 apresenta um estudo de caso desenvolvido com objetivo de avaliar a qualidade da modularização gerada. Trabalhos relacionados são apresentados na Seção 4. Por fim, a Seção 5 conclui este trabalho e levanta alguns trabalhos futuros.

2. Solução Proposta

O *ArchGraph* foi desenvolvido como um *plug-in* para o ambiente de desenvolvimento Eclipse. Essa abordagem, além de facilitar a utilização da ferramenta ao integrá-la em um ambiente de desenvolvimento largamente utilizado, ajuda na extração das informações de dependências entre entidades de projetos em desenvolvimento. As subseções seguintes apresentam detalhes de como a solução foi construída.

2.1. Arquitetura

Conforme mostra a Figura 1, o *plug-in ArchGraph* é formado por três componentes principais: *Extrator de Dependências*, *Motor de Clusterização* e *Gerador da Visualização*, cada um responsável por uma etapa do processo de geração dos módulos.

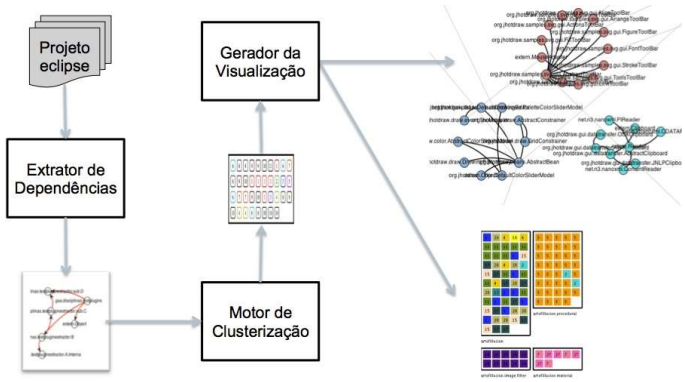


Figura 1. Arquitetura proposta

Na primeira etapa, o *Extrator de Dependências* faz o *parsing* de uma árvore sintática abstrata extraída de um projeto Java e constrói seu grafo de dependências. Nesse grafo, cada classe é representada através de um nó e a dependência entre classes é sinalizada através de uma aresta.

Na segunda etapa, o grafo é repassado ao *Motor de Clusterização*, o qual aplica um dos algoritmos detalhados na Seção 2.2, gerando um conjunto de conjuntos de classes, que representam a nova modularização proposta por nossa solução.

A última etapa, realizada pelo *Gerador da Visualização*, consiste na apresentação gráfica da modularização proposta. Para auxiliar não só o entendimento, como também na avaliação da modularização produzida, são geradas duas representações diferentes: uma utilizando grafos, onde os agrupamentos são sinalizados através de cores e outra utilizando um Mapa de Distribuição [Ducasse et al. 2006], que apresenta os módulos gerados mapeados nos pacotes reais do sistema. A Seção 3 fornece mais detalhes de como os resultados são apresentados.

2.2. Algoritmos de Clusterização

Após a geração do grafo de dependências das classes do sistema alvo, é preciso identificar quais classes estão relacionados e assim inferir como essas podem ser agrupadas. Para realizar tais atividades, o *ArchGraph* implementa os dois algoritmos detalhados a seguir.

2.2.1. Hill Climbing

Para avaliar a qualidade da modularização de um sistema dois critérios largamente utilizados são coesão e acoplamento [Anquetil et al. 1999]. Coesão avalia o grau de similaridade entre classes agrupadas (*intra-cluster*) e acoplamento define a similaridade entre classes de diferentes agrupamentos (*inter-clusters*). Utilizando tais características como critério de qualidade, uma boa modularização deve maximizar a coesão e diminuir o acoplamento.

Adotando tais critérios, podemos definir uma função, que dado um grafo e um conjunto de agrupamentos de classes, retorne um valor baseado no numero de arestas *intra-cluster* e *inter-clusters*, sendo retornado valores maiores para agrupamentos com mais arestas *intra-cluster* do que *inter-clusters*. Com tal função, nosso problema de modularizar um software poderia ser resolvido com uma busca pelo agrupamento ideal, ou seja aquele com maior valor para tal função. O problema dessa abordagem é que mesmo para sistemas pequenos a quantidade de agrupamentos possíveis é muito grande. Como exemplo, para um sistema com 15 classes, teríamos 1.382.958.545 agrupamentos possíveis. Como a maioria dos sistemas possui bem mais classes, normalmente centenas ou milhares delas, tal abordagem é inviável computacionalmente.

Quando a solução ótima não é viável devemos buscar soluções aproximadas, que embora não garantam o resultado ótimo, retornam resultados suficientemente adequados. O algoritmo *Hill Climbing* [Russell and Norvig 2009] se encaixa nesse cenário. Ele inicia com uma solução arbitrária do problema e tenta achar um resultado melhor, incrementalmente, mudando um único elemento da solução obtida até aquele momento. Se a mudança produzir um melhor resultado, uma nova mudança é feita nessa nova solução, sendo esse passo repetido até que não seja possível encontrar mais melhorias no resultado.

Nossa implementação para o algoritmo *Hill Climbing* se baseia no trabalho de Mancoridis e Mitchell [Mancoridis et al. 1998], o qual utiliza a função de qualidade da modularização (MQ) apresentada abaixo. A fórmula define a qualidade em termos da soma do grau de dependências entre classes do mesmo *cluster* (A_i) e de dependências entre classes de *clusters* diferentes (E_{ij}), sendo *k* o número total de *clusters*.

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j} & \text{, se } k > 1 \\ A_1 & \text{, se k=1} \end{cases} \quad (1)$$

2.2.2. Betweenness

O segundo algoritmo tem como base o trabalho de Girvan e Newman [Girvan and Newman 2002], que propõe uma variação do algoritmo tradicional de *Betwenness*, realizando o cálculo da centralidade das arestas ao invés da centralidade dos vértices. A centralidade das arestas de um grafo é calculada tendo como base o número de caminhos mais curtos entre todos vértices do grafo. Se o grafo possui conjuntos de vértices fortemente conectados e esses são fracamente conectados a outro grupos, por meio de uma ou poucas arestas, então todos os caminhos mais curtos entre grupos passarão por essas poucas arestas, fazendo com que elas possuam alto valor de centralidade. Assim, o algoritmo calcula a centralidade das arestas e, ao remover as de

maior valor, expõe as comunidades existentes no grafo.

Em nossa solução utilizamos uma implementação do algoritmo *Betwenness* disponibilizada pela biblioteca de modelagem de grafos JUNG[1]. Esse algoritmo requer que seja informado o número de arestas a serem removidas.

3. Estudo de Caso

Para avaliar a qualidade da modularização gerada, foram realizados experimentos com dois diferentes sistemas de código aberto: *ArtOfIllusion*[2] e *JHotDraw*[3]. Os experimentos realizados têm como objetivo dar respostas aos seguintes questionamentos:

- Q1: *Qual dos algoritmos de clusterização implementado produz melhores resultados?* Mais do que eleger um vencedor, esse questionamento busca identificar direcionamentos para futuras pesquisas e indícios de oportunidades de aprimoramento da solução.

- Q2: *A modularização gerada representa uma adequada divisão lógica do sistema?* A resposta a esse questionamento é de fundamental importância para o uso prático da ferramenta. O fato de ser gerada uma modularização com ótimos valores de acoplamento e coesão pode não necessariamente representar o agrupamento adequado em termos da estrutura lógica do sistema.

- Q3: *A modularização gerada é capaz de apontar falhas e/ou oportunidades de melhoria da modularização do sistema?* Com essa pergunta buscamos identificar se mesmo o sistema tendo sido desenvolvido de forma modular nossa solução gera informações que auxiliem a aprimorar essa modularização.

3.1. Análise dos Algoritmos de Clusterização (Q1)

Os dois algoritmos foram aplicados a cada um dos sistemas selecionados. Os resultados apresentados focam em dependências do tipo herança, ou seja, as arestas ligam classes que possuem uma herança direta do tipo pai e filha.

A Tabela 1 apresenta um resumo dos experimentos. O algoritmo *Betweenness* foi configurado para remover 10% das arestas existentes (32 para *ArtOfIllusion* e 48 para o *JHotDraw*). Foram realizadas 10 execuções do *Hill Climbing*, sendo os valores apresentados uma média dos resultados obtidos.

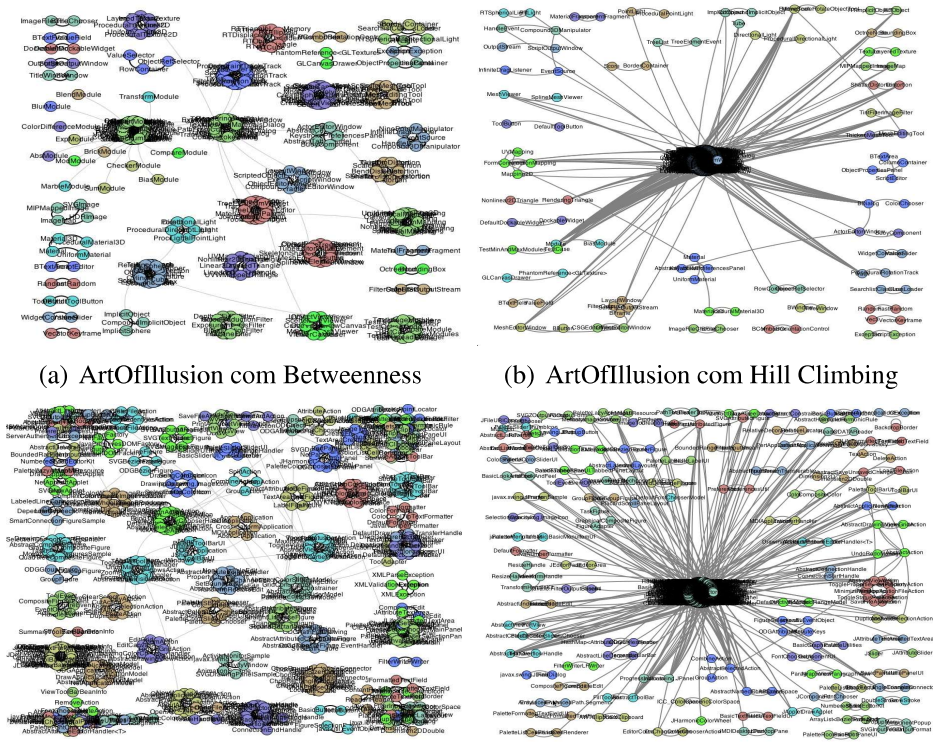
Tabela 1. Resultados dos testes. *média, com desvio padrão inferior a 0,01

Sistema	Hill Climbing*		Betweenness	
	MQ	Nº de Clusters	MQ	Nº de Clusters
ArtOfIllusion	0,5280	49	0,2728	57
JHotDraw	0,5928	104	0,4236	120

Analisando os *clusters* gerados pelos dois algoritmos, mostrados na Figura 2, é possível observar que o algoritmo de *Hill Climbing* tende a agrupar uma grande quantidade de classes em um único *cluster* e o restante das classes são divididas em pequenos

[1]Java Universal Network/Graph. http://jung.sourceforge.net
[2]http://www.artofillusion.org
[3]http://www.jhotdraw.org

clusters. Essa forma de agrupamentos é descrita como uma divisão não adequada de um sistema em [Anquetil et al. 1999], sendo denominada *black hole*, pois um *cluster* atua como um buraco negro absorvendo praticamente todas as classes. Já a modularização sugerida pelo algoritmo *Betweenness*, mesmo sem uma análise individual dos *clusters*, é mais próxima de uma modularização esperada, onde temos diversos módulos de tamanhos similares e poucas classes não agrupadas.



(a) ArtOfIllusion com Betweenness

(b) ArtOfIllusion com Hill Climbing

(c) JHotDraw com Betweenness

(d) JHotDraw com Hill Climbing

Figura 2. Resultado da clusterização com os dois diferentes algoritmos

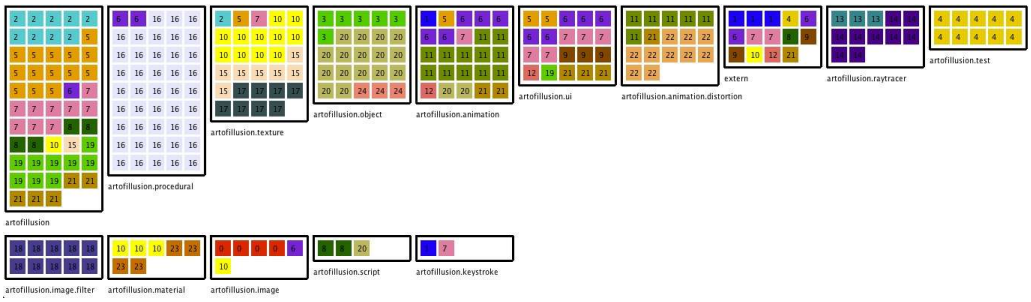
Tendo em vista que o algoritmo *Betweenness* apresentou sugestões de modularização mais próximas de uma divisão lógica do sistema, os questionamentos Q2 e Q3 serão analisados com foco nos resultados obtidos com esse algoritmo.

3.2. Análise da Qualidade da Modularização Gerada (Q2 e Q3)

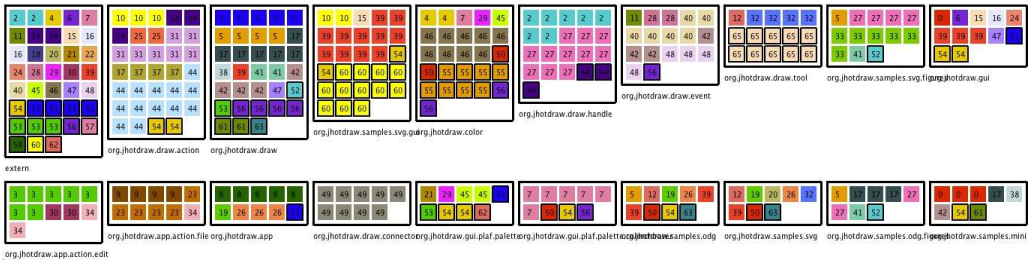
Para avaliar a qualidade da modularização utilizaremos a visualização do resultado como um Mapa de Distribuição. Um Mapa de Distribuição é uma visualização na qual os pacotes são mostrados como caixas, sendo essas preenchidas com quadrados coloridos que representam as classes do pacote. Em nossa visualização a cor/número do quadrado representa o módulo sugerido pela ferramenta para aquela dada classe. Dessa forma, utilizando o Mapa de Distribuição é possível comparar a modularização sugerida pela ferramenta (cor das classes) com a modularização real (caixas representando os pacotes do sistema).

Os Mapas de Distribuição, obtidos com o algoritmo *Betweenness* na configuração descrita anteriormente, são apresentado na Figura 3. Para evitar uma poluição excessiva da figura, removemos *clusters* com menos de três classes e apresentamos apenas os 20 pacotes com maior número de classes.

Para análise da qualidade da modularização destacamos algumas observações:



(a) ArtOfIllusion



(b) JHotDraw

Figura 3. Mapa de Distribuição para os dois sistemas analisados. Cada número/cor corresponde a um dos agrupamentos gerados pela ferramenta

- A sugestão de modularização para o pacote *artofillusion.image.filter* foi exatamente implementada. Esse pacote contem todas as classes do *cluster* 18. O mesmo acontece com os pacotes *org.jhotdraw.draw.connector* e *artofillusion.test*.

- Dentre as 45 classes do pacote *artofillusion.procedural*, 43 classes pertencem ao módulo 16, sugerido pelo *ArchGraph*, e as duas outras foram colocadas no módulo 6. Ao analisar as classes detalhadamente é possível observar que todas as 43 classes do pacote estão relacionadas com a classe denominada *Module* e que o padrão de nome dessas 43 classes tem como subnome "*Module*". A mesma análise leva a crer que as outras duas classes estariam melhor agrupadas no pacote *artofillusion.ui*, pois elas estão relacionadas a criação de interface gráfica e menus.

- O pacote *artofillusion.raytracer* possui 12 classes, sendo que nesse caso foi sugerida a distribuição dessas classes em dois módulos diferentes, 13 e 14. Uma análise, utilizando apenas o padrão de nomenclatura das classes, valida a sugestão da ferramenta, pois as classes do módulo 13 possuem em sua nomenclatura o subnome "*Light*"(*RTLight*, *RTSphericalLight* e *RTDirectionalLight*), enquanto que as demais possuem outro padrão de nomenclatura. Análise semelhante pode ser estendida a pacotes como *animation.distorcion* e *org.jhotdraw.app.action.edit*.

- O pacote *artofillusion* é um dos pacotes que apresentou maior variedade de módulos. Foi sugerido que muitas das classes desse pacote fossem agrupadas com classes de outros pacotes. Um fato a ser observado nesse caso é que o pacote *artofillusion* é a raiz do projeto. A raiz do projeto é normalmente o local onde as classes são criadas quando não se sabe ou se está em dúvida sobre qual pacote uma classe deve pertencer. Dessa forma, o fato de a ferramenta ter sugerido uma

modularização bem diferente da implementada nesse pacote pode ser entendido como indício de que não há uma adequada modularização de suas classes.

As quatro observações destacadas contribuem para mostrar que a modularização gerada pela ferramenta faz sentido do ponto de vista lógico do sistema (Q2), pois representa em muitos casos a implementação observada na prática para os sistemas analisados. Além disso as três últimas observações sugerem que a ferramenta é capaz de gerar indícios de erros de modularização, bem como oportunidades para aprimoramento dessa (Q3).

4. Trabalhos Relacionados

Dois trabalhos importantes na área são [Anquetil et al. 1999] e [Wiggerts 1997], pois realizam uma explanação ampla sobre o tema, sendo bastante úteis para conhecimento das principais técnicas de clusterização utilizadas para engenharia reversa de sistemas.

A ferramenta BUNCH [Mancoridis et al. 1998, Mitchell and Mancoridis 2006] representa um dos principais trabalhos relacionadas a clusterização de sistemas baseado em informações extraídas do código fonte. BUNCH realiza a clusterização com base na busca por agrupamentos de entidades que apresentem maior valor para uma função de qualidade da modularização, utilizando para isso um algoritmo baseado na técnica de *Hill Climbing* e heurísticas para geração do agrupamento inicial e dos agrupamentos vizinhos. Outros trabalhos que seguem abordagem semelhantes são [Praditwong et al. 2011] e [Annervaz et al. 2013].

Nosso trabalho difere desses trabalhos mencionados, pois não se restringe ao uso de algoritmos baseado em busca heurística. Como abordagem alternativa, adaptamos um algoritmo de identificação de comunidades em grafo, baseado na centralidade de arestas, para o problema de modularização de sistemas. No melhor do nosso conhecimento, não existem trabalhos que empreguem tal abordagem.

5. Conclusões e Trabalhos Futuros

Esse trabalho apresentou uma solução para modularização automática de sistemas baseada no uso de técnicas de clusterização. Os resultados obtidos mostram que embora a modularização proposta pela ferramenta não seja necessariamente a ideal é uma boa indicação do caminho a ser seguido, sendo útil como um modelo inicial, a ser refinado por um engenheiro de software.

Foi possível observar que mesmo com valores inferiores de MQ os resultados apresentados pelo algoritmo *Betweenness* são mais próximos de uma modularização adequada. Tais resultados indicam que essa é uma abordagem promissora, sendo uma alternativa ao tradicional uso do algoritmo *Hill Climbing* (usado, por exemplo, pela conhecida ferramenta Bunch).

Em trabalhos futuros deseja-se realizar testes com outros algoritmos de clusterização, bem como a combinação de diferentes tipos de dependências entre classes. Pretende-se também investigar modificações na fórmula de cálculo da qualidade da modularização, pois acreditamos que é possível aprimorá-la de forma que ela gere melhores resultados.

Por fim, pretende-se investigar o uso dos clusters propostos para detectar violações arquiteturais [Passos et al. 2010, Terra and Valente 2009] e também para comparar e analisar remodularizações de sistemas [Santos et al. 2014, Silva et al. 2014].

Agradecimentos

Gostaríamos de agradecer o auxílio da FAPEMIG e CNPQ.

Referências

Annervaz, K. M., Kaulgud, V. S., Misra, J., Sengupta, S., Titus, G., and Munshi, A. (2013). Code clustering workbench. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 31–36.

Anquetil, N., Fourrier, C., and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255.

Ducasse, S., Girba, T., and Kuhn, A. (2006). Distribution Map. *22nd IEEE International Conference on Software Maintenance (ICSM)*, 0:203–212.

Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *National Academy of Sciences*, 99(12):7821–7826.

Lehman, M., Ramil, J., Wernick, P., Perry, D., and Turski, W. (1997). Metrics and laws of software evolution-the nineties view. In *4th International Software Metrics Symposium*, pages 20 –32.

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y.-F., and Gansner, E. R. (1998). Using automatic clustering to produce high-level system organizations of source code. In *6th International Workshop on Program Comprehension (IWPC)*, pages 45–52.

Mitchell, B. and Mancoridis, S. (2006). On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonca, N. C. (2010). Static architecture conformance checking – an illustrative overview. *IEEE Software*, 27(5):82–89.

Praditwong, K., Harman, M., and Yao, X. (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37:264–282.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach, 3rd edition*. Pearson Higher Education.

Santos, G., Valente, M. T., and Anquetil, N. (2014). Remodularization analysis using semantic clustering. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 224–233.

Silva, L., Valente, M. T., and Maia, M. (2014). Assessing modularity using co-change clusters. In *13th International Conference on Modularity*, pages 49–60.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094.

Wiggerts, T. (1997). Using clustering algorithms in legacy systems remodularization. *4th Working Conference on Reverse Engineering (WCRE)*, pages 33–43.