

# **Um estudo empírico do uso da comunicação para caracterizar a ocorrência de dependências de mudança no projeto Ruby on Rails**

**Igor S. Wiese<sup>1</sup>, Rodrigo T. Kuroda<sup>2</sup>, Reginaldo Ré<sup>1</sup>,  
Gustavo A. Oliva<sup>3</sup>, Marco A. Gerosa<sup>3</sup>**

<sup>1</sup>Programa de Pós Graduação em Informática – Universidade Tecnológica Federal do Paraná – Campus Cornélio Procópio

<sup>2</sup>Departamento de Ciência da Computação  
Universidade Tecnológica Federal do Paraná – Campus Campo Mourão – PR – Brasil

<sup>3</sup>IME/USP – Departamento de Ciência da Computação  
Universidade de São Paulo (USP) - São Paulo – SP – Brasil

{igor, reginaldo}@utfpr.edu.br, rodrigokuroda@gmail.com,  
{goliva, gerosa}@ime.usp.br

**Abstract.** This paper studies the communication to characterize the occurrence of change dependencies. First, we propose separate change dependencies into strong and weak ones, according to the distribution of their occurrence in a release. Then, using classification algorithms and metrics from the social network extracted from traces of developers' communication, we found that it is possible to characterize strong and weak change dependencies. The percentage of correctly classified pair of files stood in the range of 72-98%. We also used feature selection algorithms to identify the best metrics.

**Resumo.** Este trabalho estuda a comunicação para caracterizar a ocorrência de dependências de mudança. Primeiro, nós realizamos a separação das dependências de mudança em fortes e fracas, de acordo com a distribuição da sua ocorrência por release. Depois, utilizando algoritmos de classificação e métricas da rede social extraída a partir de rastros de comunicação de desenvolvedores, nós descobrimos que é possível caracterizar os tipos de dependências de mudanças fortes e fracas. Nós obtivemos um percentual de 72-98% de pares de arquivos corretamente classificados. Nós também utilizamos algoritmos de seleção de atributos para identificar as melhores métricas.

## **1. Introdução**

Sistemas de software são compostos de artefatos que dependem uns dos outros. Como resultado, alguns artefatos evoluem conjuntamente durante o desenvolvimento do software [Canfora et al. 2014]. Ball et al. (1997) foram os primeiros pesquisadores a observar esse fenômeno de forma empírica ao minerar *logs* do sistema de controle de versão de um compilador. No ano seguinte, Gall et al. (1998) chamaram essa relação de coevolução de dependência lógica (do inglês, *logical dependency*). Ao passar do tempo, outros nomes passaram a ser mais utilizados na literatura, incluindo dependências de mudança (*change dependencies*), dependências evolucionárias (*evolutionary dependencies*), e comudanças históricas (*historical co-changes*). Neste artigo, utilizaremos o termo *dependências de mudança*.

Há uma série de estudos referentes à *identificação* de dependências de mudança [Canfora et al. 2010; Ying et al. 2004; Zimmermann et al. 2005], dos quais destacam-se a proposta de Zimmermann et al. (2005). Tal estudo foi amplamente utilizado pelos trabalhos que aplicam dependências de mudança para melhorar a qualidade do software. Entretanto, poucos estudos têm direcionado esforços para *caracterizar* a ocorrência de dependências de mudança, isto é, as razões por detrás da formação de tais dependências [Oliva et al. 2011]. Zhou et al. (2008) predizem a existência ou ausência de dependência de mudança entre dois arquivos usando métricas como a existência de dependência estática de código, ocorrência de dependência de mudança entre dois arquivos no passado, a idade da mudança e quem realizou a mudança. Oliva e Gerosa, mostraram que a existência de um acoplamento estrutural nem sempre leva a formação de dependências de mudanças [Oliva and Gerosa 2011].

Embora trabalhos anteriores tenham desconsiderado aspectos sociais, o desenvolvimento de software é inherentemente uma atividade sociotécnica, especialmente dada a necessidade de colaboração e comunicação entre os envolvidos no projeto [Cataldo et al. 2009]. De fato, no começo de 1968, Conway formulou a hipótese de que organizações que projetam sistemas são restringidas a produzir projetos que são cópias de suas próprias estruturas de comunicação [Conway 1968]. Então, a “Lei de Conway” assume, de certa forma, uma associação entre a arquitetura do software e as relações sociais. Influenciados pela Lei de Conway, alguns pesquisadores defendem que a interação entre os desenvolvedores pode impactar diretamente na qualidade da arquitetura do software, e por meio desta, transitivamente, influenciar na formação das dependências de mudança [Sebastiano Panichella 2014].

Este trabalho contribui de duas formas. Primeiro, nós dividimos as dependências de mudança em dois grupos: fortes e fracas. Nós propomos esta divisão porque entendemos que algumas dependências podem precisar ser priorizadas em relação a outras. Neste sentido, entender as origens da formação de dependências fortes e fracas é mais importante do ponto de vista prático, que prever ausência ou ocorrência de dependências de mudanças. Segundo, nós investigamos se métricas obtidas a partir da comunicação dos desenvolvedores em torno das suas contribuições (*pull requests*) podem caracterizar a formação destes dois grupos de dependências de mudança.

Desta forma, inspirados na lei de Conway, nós utilizamos métricas de análise de redes sociais para caracterizar as origens das dependências fortes e fracas entre pares de arquivos em uma mesma release. Nós utilizamos o termo *caracterizar* ao invés de predizer, uma vez que não estamos coletando as métricas em uma release para prever a ocorrência em outra release. Nós estamos usando os algoritmos de classificação para determinar métricas relevantes para cada um dos grupos de dependência de mudança propostos. Duas questões de pesquisa foram investigadas neste trabalho:

**QP1) É possível caracterizar dependências de mudança fortes e fracas utilizando métricas obtidas a partir da comunicação dos desenvolvedores?**

**QP2) Quais métricas utilizadas foram mais relevantes para esta caracterização?**

Entender as origens da formação destas dependências é importante, pois pode ajudar na elaboração de métodos mais robustos para a identificação destas. Além disto, já se sabe que dependências de mudanças podem ser utilizadas, por exemplo, para

prever defeitos [D'Ambros et al. 2009] e realizar análise de impacto de mudança [Zimmermann et al. 2005]

## 2. Metodologia

Nas Subseções abaixo, nós descrevemos os passos para coletar e identificar as dependências fortes e fracas a partir das submissões de *pull-requests*, como as métricas de redes sociais são obtidas da comunicação dos desenvolvedores nos *pull-requests* e como usamos os algoritmos de classificação para caracterizar dependências de mudanças fortes e fracas.

### 2.1. Coleta de dados e identificação das dependências de mudanças fortes e fracas

O projeto *Ruby on Rails*<sup>1</sup> foi utilizado como objeto de estudo neste trabalho. Este projeto foi escolhido baseado no nível de interesse da comunidade de desenvolvedores que contribui no repositório Github. Ao todo, o projeto Rails possui mais de 22.200 “estrelas” e 8.300 *forks* do seu código.

Nós usamos a API do GitHub<sup>2</sup> para coletar os dados do histórico de quatro releases. Releases anteriores foram descartadas por não terem todo seu histórico de commits ou *pull requests* completos no Gitub. Desta forma, foram coletados dados sobre as modificações dos arquivos e as discussões realizadas pelos desenvolvedores em cada *pull request*. Para encontrar as dependências de mudança e separá-las em dependências de mudança fortes e fracas, nós seguimos os seguintes passos: 1. Agrupar todos os *pull requests* e *commits* por relase 2. Remover cada *pull request* sem *commit* ou com *commits* que não foram integrados (*merged*) no projeto; 3. Se em um *pull request* existir mais de um *commit*, juntá-los em uma única transação; 4. Combinar todos os arquivos modificados em um *pull request* em pares de arquivos, excluindo arquivos que tem extensão de imagens e textos; 5. Aplicar o algoritmo de regra de associação APRIORI, utilizado por Zimmerman (2005) para encontrar as dependências de mudança definindo um valor limiar para suporte e outro para confiança. Para este trabalho nós utilizamos suporte igual a 2 e confiança igual a 30%; 6. Analisar a distribuição das dependências de mudança encontradas após o algoritmo de regra de associação para atribuir as dependências de mudanças a um grupo (forte ou fraco). Assim, pares de arquivos que formam dependências acima ou iguais ao valor de limiar do terceiro quartil serão atribuídos ao grupo “forte”. As demais dependências de mudança estarão no grupo “fraca”. Desta forma, as dependências fortes representam pelo menos 25% das dependências mais recorrentes após a aplicação do algoritmo de regra de associação. Estas dependências, são as que devem receber mais atenção dos desenvolvedores, por exemplo, se houver a necessidade de priorização de revisão de código ou cobertura de testes.

---

<sup>1</sup> Detalhes do projeto podem ser encontrados na página oficial: <http://rubyonrails.org> e no Github: <https://github.com/rails/rails>.

<sup>2</sup> Para mais detalhes acessar: <https://developer.github.com/v3/>

## 2.2. Redes Sociais e Métricas de Análise de Redes Sociais (SNA)

Em nossa rede de comunicação, cada desenvolvedor que comentou em um *pull request* incluído no nosso estudo representa um nó. Arestas direcionadas representam a troca de mensagens entre os desenvolvedores. Nós adicionamos peso nas arestas contando a quantidade de vezes que um desenvolvedor interagiu com outro sobre a sequência de comentários. Depois da rede criada, nós utilizamos a API Jung<sup>3</sup> para calcular as métricas de análise de redes sociais. Nós calculamos 18 métricas que correspondem a métricas de centralidade da rede (*degree*, *betweenness*, *closeness*, e *eigenvector*), medidas de *ego network* (*ego Betweenness*, *ego size*, *ego ties* e *ego density*), medidas de *structural hole* (*efficiency*, *effective size*, *constraint* e *hierarchy*) e medidas globais (*size*, *ties*, *density*, *diameter*, *número de mensagens* e *número distintos de desenvolvedores que comentaram*). Mais detalhes sobre as métricas pode ser encontrados em [Wasserman and Faust 1994].

Além das métricas de rede social, nós também adicionamos três métricas obtidas do histórico de modificações. Nós adicionamos as medidas de *code churn* do arquivo um (codeChurn), *code churn* do arquivo dois (codeChurn2) e a média do *code churn* entre os dois arquivos que formavam o par para todo o histórico da release (codeChurnAVG). *Code churn* é uma métrica frequentemente utilizada como medida de controle, por ser um bom preditor em outros contextos [Hall et al. 2012]. Desta forma, nosso conjunto de dados final possui 21 métricas.

## 2.3. Classificação

Os algoritmos de classificação leem um conjunto de treinamento e constroem um modelo de aprendizado a partir das métricas que são mais úteis para diferenciar cada uma das classes. Portanto, neste estudo, nós usamos as métricas descritas na Subseção 2.2 como entrada para os algoritmos de classificação. As classes, por sua vez, são as de dependências de mudança fracas e fortes. Desta forma, nós executamos cinco classificadores frequentemente encontrados na literatura, a saber: *Naïve Bayes* (Bayes), *J48* (árvore de decisão), *JRip* (regras), *KNN - K-nearest neighbours* (preguiçoso ou IBk lazy) and *Bagging* (alvo ou target). Entre parênteses estão descritos sob quais critérios os algoritmos criam seus modelos de aprendizagem. Nós selecionamos estes algoritmos para retirar um possível viés nos resultados, por exemplo, se selecionássemos um único algoritmo de classificação os resultados poderiam refletir o sucesso ou fracasso daquele algoritmo em particular com o conjunto de dados utilizado no estudo.

Tipicamente, estudos de classificação são avaliados utilizando medidas de sensibilidade (*recall*), precisão (*precision*) e área abaixo da curva ROC (AUC) [Demšar 2006]. A sensibilidade identifica a proporção das instâncias de uma determinada classe que o modelo pode recuperar com sucesso. A precisão pode medir a porcentagem correta de identificação de uma classe. A AUC é um bom indicativo para comparar modelos de classificação. Todas as medidas variam de 0 até 1, com o valor 1 representando a perfeita classificação. Para avaliar estas medidas, nós utilizamos a validação cruzada estratificada, que divide o conjunto de dados em *n* subconjuntos de tamanho aproximadamente igual mantendo a proporção das classes em cada

---

<sup>3</sup> Para mais detalhes acessar: <http://jung.sourceforge.net/doc/api/>

subconjunto. Os objetos de  $n - 1$  partições são utilizados no treinamento do modelo. O subconjunto restante é usado como teste. A vantagem desta técnica é que todo o conjunto de dados é utilizado para treinamento e teste [Demšar 2006].

### 3. Resultados

#### 3.1. QP1) É possível caracterizar dependências de mudança fortes e fracas utilizando métricas obtidas a partir da comunicação dos desenvolvedores?

Para verificar se é possível caracterizar dependências de mudança fortes e fracas utilizando as métricas de comunicação, nós primeiro coletamos os dados e definimos quais pares de arquivos representavam uma dependência de mudança forte e fraca para o nosso estudo. Para isto, nós utilizamos os seis passos descritos na Subseção 2.1.

A Tabela 1 apresenta a sumarização da coleta de dados de cada uma das quatro releases estudadas. A coluna “máximo suporte” indica a quantidade de dependências de mudança encontradas entre os pares de arquivos. Além desta coluna, a mediana e o terceiro quartil do limiar de suporte são apresentados. A quantidade (#) de pares identificados como fortes e fracos, indica o balanceamento das classes formadas utilizando o algoritmo de regra de associação e a distribuição dos quartis. O percentual de cada classe é apresentado entre parênteses nas colunas #instâncias fortes e fracas. Por fim, são apresentadas a quantidade de *pull requests* e o total de instâncias (pares de arquivos) identificados em cada release. Desta forma, baseados na divisão do terceiro quartil, nós definimos o rótulo de forte ou fraca para cada dependência de mudança. Por exemplo, na release 3.1, para uma dependência de mudança ser classificada como forte, o valor do suporte dela deveria estar entre 4 e 14 mudanças. Isto significa que dois arquivos, A e B, deveriam ser modificados juntos em pelo menos 4 *pull requests*.

**Tabela 1. Sumarização das quatro releases estudadas.**

Release	Máximo Suporte	Mediana Suporte	Limiar de Suporte (3º Quartil)	Mediana Confiança	# inst. Fortes	# inst. Fracos	# Pull requests	Total de Instâncias
3.1	14	6	$\geq 4$	60,00%	873 (45,17%)	1060 (54,83%)	303	1933
3.2	7	2	$\geq 2$	100,00%	216 (46,87%)	215 (53,13%)	229	431
4.0	14	8	$\geq 3$	70,17%	513 (39,70%)	778 (60,30%)	1452	1291
4.1	7	2,27	$\geq 2$	66,66%	339 (16,12%)	1764 (83,88%)	769	339

Para executar os algoritmos de classificação nós utilizamos a ferramenta Weka<sup>4</sup>, que oferece implementações para os cinco algoritmos mencionados na Subseção 2.3. Utilizando o Weka, nós selecionamos os arquivos de entrada de dados armazenados em CSV, que continham nas linhas, cada uma das instâncias que representavam as dependências de mudança, e nas colunas cada uma das 21 métricas. A última coluna do CSV representava o valor da classe ao qual aquela dependência de mudança pertencia (forte ou fraca). Para cada um dos algoritmos e releases, nós executamos a tarefa de classificação. A Tabela 2 apresenta os resultados da caracterização das dependências de mudança fortes e fracas por meio do uso das métricas sociais e do *code churn*. Esses resultados foram obtidos com as melhores métricas já selecionadas (discussão na

---

<sup>4</sup> Para acessar mais informações sobre a ferramenta: <http://www.cs.waikato.ac.nz/ml/weka/>

Subseção 3.2). Observando os valores da AUC, que é a métrica para comparação de modelos, nós observamos que os algoritmos Bagging e KNN tiveram os melhores resultados nas releases 3.1 e 3.2. O algoritmo Bagging teve o melhor desempenho para as release 4.0 e 4.1. Apesar do Bagging ser o melhor algoritmo em todas as releases, somente na release 4.0 é possível perceber uma diferença maior entre os algoritmos. Nesta release, a diferença do Bagging (maior AUC) para o Naïve Bayes (menor AUC) foi de 0.234. A diferença do Bagging para o segundo melhor algoritmo foi somente 0.043. Na Release 4.1 a diferença entre Bagging (maior AUC) para Naïve Bayes (menor AUC) foi de 0.175. Todos os algoritmos, com exceção do Naïve Bayes, tiveram desempenho similar ao Bagging, o que significa dizer que a escolha de um algoritmo, não influencia em diferença de acertos na classificação de dependências de mudança fortes ou fracas.

**Tabela 2. Resultados da classificação de dependências de mudança forte e fracas**

Release	Algoritmos	% Instâncias corretamente Classificadas	Precisão (forte)	Precisão (fraca)	Sensibilidade (forte)	Sensibilidade (fraca)	AUC
3.1	J48	0,977	0,983	0,974	0,968	0,986	0,977
	<b>Bagging</b>	<b>0,981</b>	<b>0,975</b>	<b>0,973</b>	<b>0,967</b>	<b>0,979</b>	<b>0,995</b>
	<b>KNN</b>	<b>0,980</b>	<b>0,979</b>	<b>0,977</b>	<b>0,971</b>	<b>0,983</b>	<b>0,995</b>
	Naïve Bayes	0,904	0,921	0,892	0,863	0,905	0,943
	JRip	0,967	0,967	0,968	0,961	0,973	0,964
3.2	J48	0,951	0,953	0,949	0,949	0,953	0,951
	<b>Bagging</b>	<b>0,948</b>	<b>0,929</b>	<b>0,971</b>	<b>0,972</b>	<b>0,926</b>	<b>0,986</b>
	<b>KNN</b>	<b>0,974</b>	<b>0,977</b>	<b>0,972</b>	<b>0,972</b>	<b>0,977</b>	<b>0,986</b>
	Naïve Bayes	0,798	0,911	0,734	0,662	0,822	0,875
	JRip	0,951	0,930	0,975	0,977	0,926	0,963
4.0	J48	0,869	0,857	0,876	0,805	0,911	0,875
	<b>Bagging</b>	<b>0,888</b>	<b>0,898</b>	<b>0,883</b>	<b>0,811</b>	<b>0,940</b>	<b>0,947</b>
	KNN	0,874	0,856	0,886	0,823	0,909	0,904
	Naïve Bayes	0,721	0,719	0,723	0,493	0,873	0,713
	JRip	0,793	0,833	0,778	0,602	0,920	0,789
4.1	J48	0,981	0,969	0,984	0,917	0,994	0,976
	<b>Bagging</b>	<b>98,24</b>	<b>0,978</b>	<b>0,983</b>	<b>0,912</b>	<b>0,996</b>	<b>0,986</b>
	KNN	97,86	0,945	0,985	0,92	0,99	0,984
	Naïve Bayes	97,77	0,876	0,911	0,499	0,986	0,805
	JRip	98,11	0,943	0,975	0,920	0,935	0,970

Assim, podemos concluir que é possível caracterizar dependências de mudança fortes e fracas utilizando métricas obtidas a partir da comunicação dos desenvolvedores. Esta conclusão é obtida verificando os valores de AUC obtidos nas quatro releases do projeto Rails. Utilizando diferentes algoritmos, nós obtivemos mais de 0.9 de AUC para cada uma das releases. Este resultado mostra que os valores de sensibilidade e precisão foram maiores que 90% em 14 dos 20 testes realizados (5 algoritmos X 4 releases). Somente a release 4.0 não apresentou este índice nas métricas de avaliação medidas.

Obviamente, os resultados obtidos não podem ser generalizados para outros projetos, e novos testes devem ser realizados em maior escala para verificar se as métricas de comunicação podem ser úteis em outros projetos. No entanto, os resultados

indicam consistência, uma vez que as métricas foram úteis em quatro diferentes releases com mais de um algoritmo de classificação.

### **3.2. QP2) Quais métricas utilizadas foram mais relevantes para esta caracterização?**

Para responder a segunda questão de pesquisa, nós utilizamos um algoritmo de seleção de atributos. Para realizar a seleção de atributos (métricas) relevantes, nós usamos um método chamado “Wrapper”, disponível na ferramenta Weka. Este método, pesquisa no espaço de subconjuntos de métricas e calcula a precisão estimada de um algoritmo de aprendizagem para cada métrica quando ela é adicionada ou removida do subconjunto. Usamos o algoritmo “WrapperSubSetEval” como avaliador da relevância de cada métrica e o algoritmo BestFirst como o método de pesquisa. O algoritmo “Best First” foi configurado com o parâmetro “backward”, que começa construindo o modelo com todas as variáveis e vai removendo-as conforme a performance do modelo melhora ou piora. Na média, 8,3 métricas foram selecionados pelos 5 algoritmos através das 4 releases. Para obter este resultado, o algoritmo de seleção de atributos testou na média, 281 modelos diferentes. A release 3.1 teve média de 9,4 métricas e 342,6 modelos. A release 3.2 teve média de 9,6 métricas com 242,8 modelos. As últimas duas releases (4.0 e 4.1) tiveram 9 e 5,2 métricas com 250,4 e 186,6 modelos respectivamente.

As métricas mais relevantes foram a densidade da rede (*density* – medidas globais) e o número de comentários (propriedades globais) com 17 seleções em 20 possíveis. Logo após estas duas métricas, *ego Betweenness* (*ego network*) e número de desenvolvedores distintos que participaram da discussão (propriedades globais) foram selecionadas 11 vezes. *Constraint*, *hierarchy* (ambos *structural hole*) e *diameter* (propriedade global) foram as próximas métricas selecionadas com 10 vezes. As métricas *code churn* do primeiro arquivo e do segundo arquivo apareceram 8 vezes. Como estas métricas foram usadas como controle, todas as métricas abaixo delas foram consideradas menos relevantes para caracterizar as dependências de mudança fortes e fracas. É importante observar que as métricas referentes a medidas globais da rede (3 métricas) e as métricas de *structural holes* (2 métricas) foram mais relevantes que as métricas de *ego network* (uma métrica) e *centrality* (nenhuma seleção). Assim, encontramos evidências que indicam que *o papel e a importância de um nó (desenvolvedor) são menos importantes do que a estrutura (organização) da rede*. As métricas de *structural hole* indicam a inexistência de ligações entre dois nós na rede, o que pode indicar falhas e ausência de comunicação.

## **4. Conclusão e Trabalhos Futuros**

Dada a importância que o gerenciamento de dependências tem para a evolução do desenvolvimento do software, nós propusemos o estudo de dependências de mudança separadas em dois grupos: fortes e fracas. Durante este trabalho mostramos que é possível caracterizar a ocorrência destes dois grupos de dependência de mudança utilizando métricas extraídas da comunicação dos desenvolvedores e algoritmos de classificação. Utilizando a seleção de atributos encontramos evidências de que medidas globais da rede e métricas de *structural hole* são mais relevantes para a caracterização.

No entanto, novos estudos são necessários para avaliar a importância das métricas de comunicação para a caracterização de dependências de mudança fortes e fracas em novos projetos. Atualmente nós estamos investigando o uso de métricas de

código fonte e métricas do histórico de modificações de arquivos e realizando testes em novos projetos.

### Agradecimentos

Nós gostaríamos de agradecer a Fundação Araucária, NAWEB e NAPSOL pelo suporte financeiro a esta pesquisa. Marco G. recebe apoio individual do CNPq e FAPESP. Igor W. e Gustavo Oliva recebem apoio da CAPES (Processo BEX 2039-13-3 e 250071/2013-4).

### Referências

- Ball, T., Adam, J.-M. K., Harvey, A. P. and Siy, P. (mar 1997). If Your Version Control System Could Talk... In ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering.
- Canfora, G., Ceccarelli, M., Cerulo, L. and Di Penta, M. (2010). Using multivariate time series and association rules to detect logical change coupling: An empirical study. In Software Maintenance (ICSM), 2010 IEEE International Conference on.
- Canfora, G., Cerulo, L., Cimitile, M. and Penta, M. D. (2014). How changes affect software entropy: an empirical study. Empirical Software Engineering, v. 19, n. 1, p. 1–38.
- Cataldo, M., Mockus, A., Roberts, J. A. and Herbsleb, J. D. (2009). Software Dependencies, Work Dependencies, and Their Impact on Failures. IEEE Transactions on Software Engineering, v. 35, n. 6, p. 864–878.
- Conway, M. E. (1968). How do Committees Invent? Datamation,
- D'Ambros, M., Lanza, M. and Robbes, R. (oct 2009). On the Relationship Between Change Coupling and Software Defects. In Reverse Engineering, 2009. WCRE '09. 16th Working Conference on.
- Demšar, J. (2006). Statistical Comparisons of Classifiers over Multiple Data Sets. J. Mach. Learn. Res., v. 7, p. 1–30.
- Gall, H., Hajek, K. and Jazayeri, M. (1998). Detection of Logical Coupling Based on Product Release History. In Proceedings of the International Conference on Software Maintenance. , ICSM '98. IEEE Computer Society.
- Hall, T., Beecham, S., Bowes, D., Gray, D. and Counsell, S. (2012). A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering, v. 38, n. 6, p. 1276–1304.
- Oliva, G. A. and Gerosa, M. A. (2011). On the Interplay between Structural and Logical Dependencies in Open-Source Software. In SBES.
- Oliva, G. A., Santana, F. W., Gerosa, M. A. and Souza, C. R. B. De (2011). Towards a classification of logical dependencies origins: a case study. In EVOL/IWPSE.
- Sebastiano Panichella, R. O. Gerardo Canfora Massimiliano Di Penta (2014). How the Evolution of Emerging Collaborations Relates to Code Changes: An Empirical. In IEEE International Conference on Program Comprehension (ICPC 2014).
- Wasserman, S. and Faust, K. (1994). Social network analysis: Methods and applications. Cambridge university press. v. 8
- Ying, A. T. T., Murphy, G. C., Ng, R. and Chu-Carroll, M. C. (sep 2004). Predicting Source Code Changes by Mining Change History. IEEE Trans. Softw. Eng., v. 30, n. 9, p. 574–586.
- Zhou, Y., Wurch, M., Giger, E., Gall, H. and Lu, J. (2008). A Bayesian Network Based approach for change coupling prediction. In Reverse Engineering, 2008 WCRE.
- Zimmermann, T., Zeller, A., Weissgerber, P. and Diehl, S. (jun 2005). Mining version histories to guide software changes. Software Engineering, IEEE Transactions on, v. 31, n. 6, p. 429–445.