# Usando aprendizagem de máquina para identificar anomalias de design prejudiciais à manutenibilidade: um estudo preliminar

**Fabrício F. Cardim**[1], **Cláudio Sant'Anna**[1]

[1]Departamento de Ciência da Computação - Universidade Federal da Bahia (UFBA) – Salvador – BA – Brasil

fcardim@ufba.br, santanna@dcc.ufba.br

***Resumo.*** *As ferramentas que detectam anomalias de design ainda geram resultados muito imprecisos. Isso pode ocorrer por elas não levarem em consideração informações relacionadas ao contexto, ao domínio e a equipe de manutenção do sistema. Além disso, desenvolvedores podem ter opiniões divergentes sobre o impacto de anomalias pré-catalogadas, como por exemplo code smells, na manutenibilidade do software. Diante desse contexto, o objetivo deste trabalho é utilizar aprendizagem de máquina para identificar anomalias que, de acordo com a percepção dos desenvolvedores, possam afetar de forma negativa a manutenibilidade do software. Para isso, executamos alguns algoritmos de aprendizado de máquina e avaliamos a eficácia destes algoritmos para detecção de anomalias com base no conjunto de exemplos fornecido pelos próprios desenvolvedores.*

## 1. Introdução

Ferramentas que detectam anomalias de *design* de forma automática (e.g., NDepend[1] e JSpiRiT[2]), geralmente utilizam um conjunto de regras aplicadas sobre métricas extraídas do código fonte. Essas regras, conhecidas como estratégias de detecção [Marinescu 2004], são utilizadas para identificar anomalias previamente catalogadas na literatura, como por exemplo os *code smells* definidos por Fowler et al. (1999). Entretanto, as definições informais e subjetivas dessas anomalias acabam dificultando a criação de estratégias que reflitam bem as anomalias e que possam ser aplicadas a qualquer contexto [Amorim et al. 2015].

Um outro aspecto que tem influência significativa na detecção de anomalias, com o apoio de métricas, é a definição dos valores limiares [Marinescu 2004]. É difícil definir valores limiares adequados, pois não é trivial saber se um determinado valor de uma métrica indica que um trecho de código possui uma anomalia [Marinescu 2004]. Por causa disso, as ferramentas baseadas em estratégias de detecção geram muitos falsos positivos e falsos negativos [Fontana et al. 2016].

Por estas razões, algumas abordagens propuseram a utilização de aprendizado de máquina para apoiar o processo de detecção de anomalias de *design*, mais precisamente, *code smells* [Kreimer 2005] [Amorim et al. 2015] [Khomh et al. 2009]

[1]http://www.ndepend.com/

[2]https://sites.google.com/site/santiagoavidal/projects/jspirit

[Fontana et al. 2016]. No entanto, apesar da afirmação de que *code smells* podem afetar a manutenibilidade do software [Fowler et al. 1999], existem evidências conflitantes que indicam que a presença de *code smells* não necessariamente aumenta o esforço de manutenção [Sjøberg et al. 2013] [Yamashita and Moonen 2013] [Yamashita 2014]. Além disso, desenvolvedores podem ter opiniões divergentes sobre o impacto de *code smells* na manutenibilidade do software [Fontana et al. 2016]. Com isso, assumimos a hipótese de que ferramentas que detectam apenas *code smells* podem não ser eficientes para identificar trechos de código que, de acordo com os desenvolvedores, possuem algum problema que pode afetar a manutenibilidade do software.

Diante desse contexto, este artigo apresenta o resultado de um estudo preliminar que avaliou a eficácia da utilização de algoritmos de aprendizado de máquina para identificar anomalias no *design* que possam impactar negativamente a manutenibilidade do software de acordo com a opinião dos desenvolvedores. Para isso, criamos um oráculo com exemplos de código avaliados manualmente pelos desenvolvedores e executamos seis algoritmos sobre o conjunto de dados gerado. Avaliamos a eficácia dos algoritmos através das medidas: precisão, *recall* e medida-F.

O restante deste artigo está organizado como descrito a seguir. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 descreve a configuração do estudo preliminar. A Seção 4 apresenta os resultados encontrados. A Seção 5 discute as ameaças à validade e a Seção 6 apresenta as conclusões e sugestões de trabalhos futuros.

## 2. Trabalhos Relacionados

O uso de técnicas de aprendizado de máquina tem sido amplamente investigado para apoiar as ferramentas a detectarem anomalias de *design* com base em exemplos previamente classificados [Fontana et al. 2016]. Neste sentindo, Kreimer (2005) propôs uma abordagem, baseada em árvores de decisão, para criar modelos para detecção dos *code smells Big Class* e *Long Method*. O conjunto de dados foi composto de 40 exemplos avaliados manualmente por uma pessoa da equipe de testes. Os resultados apresentados indicaram que as árvores de decisão tiveram uma boa precisão para o conjunto de dados analisado.

Khomh et al. (2009) propuseram uma abordagem baseada em Bayesian Beliefs Networks (BBN) para detectar instâncias de *God Class*. Os autores utilizaram 2 estudantes de graduação e 2 estudantes de pós-graduação para avaliar, manualmente, um conjunto de classes, reportando se a classe era ou não uma *God Class*. Após esse procedimento, foi criado um oráculo com 19 instâncias de *God Class*. Os resultados da avaliação do modelo apontaram uma precisão de 68% para detecção de instâncias de *God Class*.

Fontana et al. (2016) realizaram um grande experimento com o objetivo de avaliar 6 algoritmos de aprendizado de máquina, com diferentes configurações, para detecção de quatro *code smells*. Para conduzir o experimento eles utilizaram um conjunto de dados composto por 1986 exemplos de *code smells* identificados manualmente por estudantes de pós-graduação. Os resultados mostraram que todos algoritmos avaliados alcançaram uma alta precisão. Além disso, os autores afirmaram que é possível alcançar uma precisão acima de 90% com apenas 100 exemplos de treinamento.

Mais recentemente, Hozano et al. (2017) realizaram um estudo para investigar a acurácia de algoritmos de aprendizado de máquina para detecção de *code smells* com

base na percepção de diferentes desenvolvedores. Eles avaliaram experimentalmente 6 algoritmos para detecção de 4 *code smells* identificados manualmente por 40 desenvolvedores recrutados da academia e da indústria. Os resultados mostraram que os algoritmos tiveram uma baixa acurácia pelo fato dos desenvolvedores terem diferentes percepções sobre os *code smells* avaliados.

Neste contexto, a principal diferença desses trabalhos para o nosso é que eles utilizaram aprendizado de máquina para detectar os elementos de código (métodos ou classes) que tinham determinados *code smells* definido pela literatura. A nossa proposta, por sua vez, busca identificar métodos com anomalias que, de acordo com os desenvolvedores, podem gerar futuras dificuldades de manutenção, independentemente de serem ou não *code smells*. Por fim, em nossas pesquisas, não identificamos trabalhos relacionados ao nosso, onde os próprios desenvolvedores do sistema foram utilizados como oráculo.

## 3. Configuração do Estudo

O objetivo principal deste estudo foi avaliar a capacidade dos algoritmos de aprendizado de máquina para detectar métodos com anomalias no *design* que, de acordo com a percepção dos desenvolvedores, podem impactar na manutenibilidade do sistema. Este estudo tenta responder as seguintes questões de pesquisa (QP):

- **QP1:** Quão precisos são os algoritmos de aprendizado de máquina para detectar métodos com anomalias de *design*, que possam prejudicar a manutenibilidade do software, de acordo com a percepção da equipe que mantém o sistema?
- **QP2:** Como os algoritmos de aprendizado de máquina lidam com desenvolvedores com diferentes níveis de conhecimento sobre orientação a objetos?

Para responder essas questões, definimos uma abordagem experimental e realizamos um estudo preliminar na Agência de Fomento do Estado da Bahia (DESENBAHIA). Esta abordagem foi composta de cinco etapas, como pode ser visto na Figura 1. A seguir descreveremos cada uma delas em detalhes.

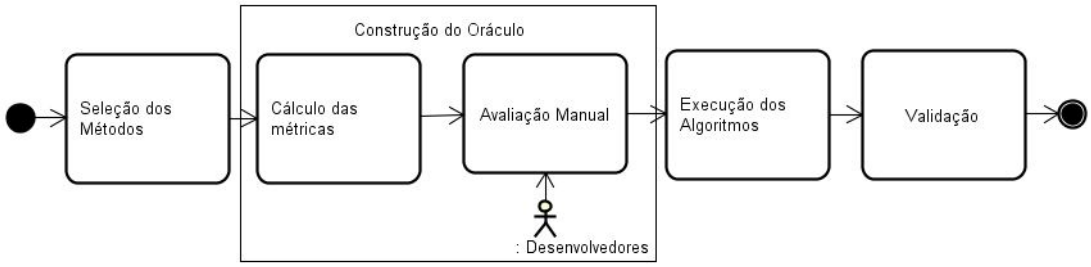


**Figura 1. Etapas do Estudo Preliminar**

**Seleção dos Métodos**. Para seleção dos métodos, realizamos uma amostragem estratificada por número de linhas de código (LOC). Os métodos foram organizados, de acordo com o valor de LOC, em três grupos: (i) métodos com LOC entre 1 e 10, (ii) métodos com LOC entre 10 e 20 e (iii) métodos com LOC maior que 20. Para cada grupo, foram selecionados 50 métodos de forma aleatória, totalizando 150 métodos. Optamos por estratificar por LOC para evitar selecionar, por acaso, apenas métodos extremamente pequenos e, portanto, menos prováveis de terem anomalias. Os métodos foram extraídos

de um sistema do tipo *workflow* que foi desenvolvido e é mantido pela equipe de desenvolvimento da DESENBAHIA. O sistema avaliado foi desenvolvido na linguagem VB.NET e possui 228 classes, 1621 métodos e 9185 linhas de código.

**Cálculo das Métricas**. Os valores das métricas são utilizados como atributos em nosso conjunto de treinamento. Para extrair os valores das métricas utilizamos a ferramenta NDepend, pois ela fornece um grande número de métricas de código fonte e tem suporte para linguagem VB.NET. As métricas selecionadas foram:

- **Número de Linhas de Código (LOC):** número de linhas de código presentes em cada método.
- **Número de Parâmetros (NP):** número de parâmetros declarados em um método.
- **Número de variáveis (NV):** número de variáveis declaradas no corpo de um método.
- **Complexidade Ciclomática (CC):** desenvolvida por McCabe (1976), ela mede a quantidade de caminhos de execução independentes a partir de um código fonte.
- **Acoplamento Eferente (AE):** refere-se à quantidade de métodos dos quais o método avaliado depende diretamente.
- **Profundidade de Aninhamento (PA):** número máximo de aninhamentos das estruturas de controle dentro de um método.

Para seleção das métricas utilizamos dois critérios: (i) as métricas tinham que ser aplicadas ao nível de métodos e (ii) as métricas tinham que refletir características que pudessem ser avaliadas pelos desenvolvedores, como por exemplo, o número de parâmetros. Métricas como acoplamento aferente, que refere-se à quantidade de métodos que acessam o método avaliado, não foram selecionadas, pois o desenvolvedor teria que avaliar outros trechos de código e poderia desviar o foco do método que estava sendo avaliado.

**Avaliação Manual**. Inicialmente, 15 desenvolvedores foram selecionados para participarem do experimento, e os 150 métodos foram divididos de forma aleatória entre eles. Por questões de disponibilidade, 2 desenvolvedores não puderam participar, com isso, dos 150 métodos selecionados, apenas 130 foram avaliados. Cada desenvolvedor avaliou manualmente 10 métodos, e para cada método, teve que responder a seguinte pergunta: *Este método apresenta potenciais problemas de design que podem vir a afetar de forma negativa a manutenibilidade do sistema?* Ao final da avaliação, disponibilizamos um questionário para sabermos o nível de experiência dos desenvolvedores com programação em geral e com orientação a objetos. As respostas de cada desenvolvedor, assim como o conjunto de dados construído ao final desta etapa, estão disponíveis no website[3] deste artigo. Ao final, o conjunto de dados foi composto por 85 instâncias (23 positivas e 62 negativas), pois os métodos que não foram avaliados, ou que tinham valores de métrica duplicados, foram removidos. É importante destacar aqui que os desenvolvedores não tiveram acesso aos valores das métricas calculadas para cada método. Além disso, os desenvolvedores tiveram acesso a todo o código fonte do software avaliado.

**Execução dos Algoritmos**. Antes de executar os algoritmos, foi necessário balancear o conjunto de dados, para que a diferença na quantidade de exemplos (positivos ou negativos) não influenciasse nos resultados. Para isso, utilizamos o filtro ClassBalancer[4],

[3]https://sites.google.com/view/vem2017/

[4]http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

disponível na ferramenta Weka [Hall et al. 2009], que atribui pesos para que cada classe tenha o mesmo peso total. Os algoritmos selecionados foram:

- **J48**. É uma implementação da árvore de decisão C4.5.
- **JRip**. É uma implementação de aprendizado baseado em regras proposicionais.
- **Random Forest**. É um classificador que cria muitas árvores de classificação como uma floresta de árvores de decisão aleatórias.
- **Naïve Bayes**. É o classificador probabilístico mais simples baseado na aplicação do teorema de Bayes.
- **SMO**. É uma implementação do algoritmo de otimização mínima sequencial de John Platt (1998) para treinar máquinas de vetores de suporte.
- **LibSVM** É um software integrado para a classificação com máquinas de vetores de suporte.

Optamos por utilizar os mesmos algoritmos utilizados no estudo realizado por Fontana et al. (2016), pois eles apresentaram um bom desempenho para a tarefa de detecção de *code smells*. Além disso, o conjunto de algoritmos selecionado abrange muitas abordagens diferentes de aprendizado de máquina, tais como: árvores de decisão, aprendizado baseado em regras, máquinas de vetor de suporte e redes bayesianas.

**Validação**. Cada um dos algoritmos foi avaliado através da técnica $k$-fold cross-validation [Refaeilzadeh et al. 2009] (com $k = 5$). Esta técnica de validação consiste em dividir o conjunto total de dados em $k$ subconjuntos mutuamente exclusivos do mesmo tamanho e, a partir disto, um subconjunto é utilizado para teste e os $k$-1 restantes são utilizados para treinamento do modelo de aprendizado. Avaliamos os resultados com base nas medidas de desempenho: precisão, *recall* e medida-F. Além disso, utilizamos o teste t pareado com 95% de confiança para comparar os resultados dos algoritmos.

## 4. Resultados

Para responder as questões de pesquisa **QP1** e **QP2** nós realizamos dois experimentos utilizando a plataforma de experimentação do Weka. Esta seção apresenta e discute os principais resultados obtidos em cada um deles.

### 4.1. Experimento 1

Para responder a questão de pesquisa **QP1**, executamos os algoritmos sobre o conjunto de dados completo, contendo os métodos avaliados por todos os desenvolvedores. Definimos o algoritmo NaiveBayes, que apresentou os melhores resultados individualmente, como base, e comparamos com os demais algoritmos utilizando o teste t pareado com 95% de confiança. A Tabela 1 apresenta o resultado deste primeiro experimento.

**Tabela 1. Resultado do Experimento com Conjunto Completo**

| Medidas de Desempenho | NaiveBayes | J48 | RandomForest | JRip | SMO | LibSVM |
|---|---|---|---|---|---|---|
| Precisão | 0.63 | 0.43 • | 0.54 • | 0.53 • | 0.59 • | 0.37 • |
| Recall | 0.62 | 0.50 • | 0.56 • | 0.54 • | 0.59 • | 0.52 • |
| Medida-F | 0.59 | 0.43 • | 0.52 • | 0.50 • | 0.57 | 0.38 • |

• Estatisticamente Pior

Como podemos observar, utilizando a opinião de todos os desenvolvedores, nenhum algoritmo foi capaz de alcançar uma precisão acima de 63%. Isso indica que, na

média, os algoritmos geraram um número alto de falsos positivos. Com relação ao *recall*, os algoritmos também não alcançaram resultados satisfatórios, o que indica que também geraram muitos falsos negativos. Por consequência, a medida-F, que é a média harmônica entre precisão e *recall* também foi baixa no geral. O NaiveBayes foi o algoritmo que apresentou o melhor resultado. A Tabela 1 mostra que o NaiveBayes foi estatisticamente superior aos demais algoritmos nas três medidas de avaliação utilizadas, com exceção da medida-F, que a diferença não foi significativa quando comparado ao SMO. O NaiveBayes foi capaz de apontar como anômalos 62% dos métodos também considerados anômalos pelos desenvolvedores. Além disso, ele apresentou 63% de precisão, ou seja, 37% dos métodos apontados pelo algoritmo eram falsos positivos.

Para tentar melhorar os resultados dos algoritmos, utilizamos um filtro da ferramenta Weka, chamado GPAttributeGeneration, que implementa um sistema de programação genética [Koza 1992] para gerar novos atributos para o conjunto de treinamento. A Tabela 2 apresenta os resultados obtidos após a aplicação do filtro.

**Tabela 2. Resultado do Experimento utilizando Programação Genética**

| Medidas de Desempenho | NaiveBayes | J48 | RandomForest | JRip | SMO | LibSVM |
|---|---|---|---|---|---|---|
| Precisão | 0.79 | 0.65 • | 0.65 • | 0.61 • | 0.76 • | 0.65 • |
| Recall | 0.77 | 0.72 • | 0.75 | 0.68 • | 0.77 | 0.72 • |
| Medida-F | 0.77 | 0.67 • | 0.67 • | 0.63 • | 0.75 | 0.66 • |

• Estatisticamente Pior

Como podemos observar, o NaiveBayes novamente apresentou o melhor resultado. Ele foi estatisticamente superior aos demais algoritmos com relação a precisão, foi superior aos algoritmos J48, JRip e LibSVM na medida *recall*, e foi superior a 4 dos 5 algoritmos avaliados, com relação a medida-F . Os resultados sugerem que é possível alcançar uma boa precisão quando os algoritmos de aprendizado são combinados com sistemas de programação genética para geração de atributos.

### 4.2. Experimento 2

Para responder a questão de pesquisa **QP2**, dividimos o conjunto de dados em dois subconjuntos: (i) composto por métodos avaliados apenas por desenvolvedores com conhecimento avançado em orientação a objetos (**Experientes OO**), e (ii) composto por métodos avaliados pelos desenvolvedores com conhecimento intermediário ou básico sobre orientação a objetos (**Inexperientes OO**). Os dois subconjuntos gerados possuem aproximadamente a mesma quantidade de métodos avaliados.

Para avaliar a eficácia das técnicas de aprendizado de máquina nos diferentes conjuntos de dados, optamos por utilizar o algoritmo NaiveBayes com o reforço da programação genética, por ter apresentado o melhor resultado no Experimento 1. A Tabela 3 apresenta os resultados deste experimento.

**Tabela 3. Eficácia do NaiveBayes nos Diferentes Conjuntos de Dados**

| Dataset | Completo | Experientes OO | Inexperientes OO |
|---|---|---|---|
| Precisão | 0.79 | 0.84 | 0.74 |
| Recall | 0.77 | 0.84 | 0.74 |
| Medida-F | 0.77 | 0.83 | 0.72 |

Como podemos observar, o algoritmo NaiveBayes foi mais eficaz no conjunto de dados "Experientes OO". O que indica que desenvolvedores mais experientes em orientação a objetos podem ter critérios semelhantes para avaliar a qualidade do *design* de métodos. Diante isso, utilizar a opinião desses desenvolvedores experientes para criar um oráculo de treinamento pode aumentar a precisão das técnicas de aprendizado de máquina. Esse resultado sugere que a precisão das técnicas de aprendizado de máquina seja sensível à diferentes perfis na equipe de desenvolvimento. Contudo, os valores de precisão e *recall* ainda são aceitáveis, até mesmo para o conjunto de dados "Inexperientes OO", formado pela opinião de desenvolvedores com pouca experiência em orientação a objetos. Isso sugere que os desenvolvedores do conjunto "Inexperientes OO" também utilizaram critérios semelhantes para avaliar a qualidade dos métodos. Apesar de não termos evidências empíricas suficientes, os resultados preliminares indicam que as técnicas de aprendizado de máquina se adaptam ao contexto e à equipe de desenvolvimento.

## 5. Ameaças à Validade

Nesta seção discutimos as ameaças à validade do nosso estudo preliminar e as ações que foram tomadas para minimizá-las.

**Validade de Construto.** O conjunto de dados (oráculo) que apoiou nosso estudo foi construído através de avaliações manuais realizadas pelos desenvolvedores. Neste caso, os desenvolvedores avaliaram cada método reportando as opções "SIM" e "NÃO", com relação a presença ou ausência de anomalias que impactassem na manutenibilidade do software. Avaliadores inexperientes podem ter tido dificuldade para entender e responder a pergunta da avaliação. Essa pergunta será melhor elaborada em estudos futuros. Além disso, fornecer somente duas opções pode ser uma ameaça, dado que os desenvolvedores não puderam informar o grau de confiança de suas respostas. Optamos por dar apenas duas opções para não tomarmos muito tempo dos desenvolvedores.

**Validade Interna.** A avaliação dos métodos foi realizada pelos desenvolvedores durante o seu expediente de trabalho. Isso pode ser uma ameaça, pois a pressão de trabalho pode afetar a avaliação. Para mitigar essa ameaça, os desenvolvedores tiveram que avaliar apenas 10 métodos e não precisaram justificar suas respostas. Com isso, não precisaram dedicar muito tempo para essa tarefa. Além disso, dois dos avaliadores não tinham experiência no software avaliado. Isso pode ser uma ameaça à validade.

**Validade Externa.** Os resultados obtidos neste experimento são válidos apenas no contexto da DESENBAHIA e para o sistema avaliado. Não podemos sugerir os mesmo resultados para outras organizações e outros sistemas. Outras equipes de desenvolvimento podem ter diferentes opiniões sobre anomalias que afetam a manutenibilidade do software. Para permitir a replicação deste experimento, disponibilizamos no website deste artigo as configurações de parametrização para cada algoritmo, os arquivos dos experimentos (.exp), o formulário de caracterização e os conjuntos de dados utilizados.

## 6. Conclusões e Trabalhos Futuros

Os resultados obtidos sugerem que algoritmos de aprendizado de máquina podem ser utilizados, no contexto da DESENBAHIA, para detectar trechos de código que, de acordo com os desenvolvedores, apresentam anomalias de *design* que podem comprometer a

manutenibilidade do sistema. Além disso, os resultados sugerem que a precisão dos algoritmos é sensível a diferentes perfis na equipe de desenvolvimento. Entretanto, até mesmo com diferentes perfis, observamos que é possível alcançar uma precisão de aproximadamente 80% e *recall* de 77%. Este estudo preliminar utilizou apenas uma pequena amostra como conjunto de dados e é necessário realizar novos experimentos para validar de forma empírica a nossa hipótese. Outros estudos, com a mesma configuração, podem ser realizados em outras organizações para comparação dos resultados. Com esses novos estudos, é possível avaliar se uma ferramenta de recomendação utilizando aprendizado de máquina seria viável e útil para identificar anomalias de *design*.

## Referências


Amorim, L., Costa, E., Antunes, N., Fonseca, B., and Ribeiro, M. (2015). Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Proceedings of the 26th International Symposium on Software Reliability Engineering*, pages 261–269. IEEE.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley, Reading, Massachusetts.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD*, 11:10–18.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314. IEEE.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE.

Refaeilzadeh, P., Tang, L., and Liu, H. (2009). Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer.

Sjøberg, D. I. K., Yamashita, A. F., Anda, B. C. D., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Yamashita, A. (2014). Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, 19(4):1111–1143.

Yamashita, A. F. and Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Information & Software Technology*, 55(12):2223–2242.