INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO TRIÂNGULO MINEIRO – Campus Ituiutaba

CURSO SUPERIOR DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LEONARDO VILARINHO CORREIA DE SOUZA

Inspecionando a qualidade de códigos PHP

ITUIUTABA, MG

2017

LEONARDO VILARINHO CORREIA DE SOUZA

Inspecionando a qualidade de códigos PHP

Projeto de pesquisa apresentado ao Instituto Federal de Educação, Ciência e Tecnologia do Triângulo Mineiro, Campus Ituiutaba, como requisito parcial para conclusão do Curso de Análise e Desenvolvimento de Sistemas.

Orientador: Prof. _____

Ailton Luiz Dias Siqueira Junior

Membro: Prof. _____

Giselle Corrêa de Souza

Membro: Prof. _____

Reane Franco Goulart

ITUIUTABA, MG

2017

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de uma aplicação em uma linguagem orientada a objetos não tem como principal objetivo a implementação de funcionalidades em menor tempo, e sim a montagem de um sistema que a qualquer momento pode ser ampliado e modificado (ANICHE, [s.d.])

Nesse sentido, para criar algo que seja expansível e modificável, é essencial conhecer o ciclo de vida de um projeto, onde é definido todo o processo de produção do *software*. Um dos seus pilares, é a garantia da qualidade do produto que está sendo criado, a qualidade define o nível de facilidade em manter e aumentar a vida útil do sistema (WALTERS; MCCALL, 1979).

Com o passar do tempo, notou-se que alguns aspectos e conceitos poderiam mostrar quando o código de um projeto tem ou não a qualidade desejada, dentre esses aspectos e conceitos, são destacados:

- **A abordagem *SOLID*** (*Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*), que agrupa princípios para produzir um código baseado em quebra-cabeça, ou seja, que pode incrementado ou decrementado sem grandes dificuldades (ANICHE, [s.d.]).

A seguir, é apresentada a Tabela 1 com os princípios que dão nome à abordagem:

Tabela 1. Apresenta e descreve alguns dos principais princípios do *SOLID*.

Princípio	**Descrição**
(SRP) Responsabilidade Única	Um objeto deve ter somente uma razão para mudar, deve tratar de apenas um objetivo (MARTIN; MICAH, 2006).
(OCP) Aberto-Fechado	As entidades de um sistema devem ser abertas para extensões e fechadas para

	modificações, qualquer nova implementação não pode modificar entidades existentes (MARTIN, 1996a; MEYER, 1989).
(LSP) Substituição de Liskov	Em herança, subclasses só podem expandir as entradas e restringir as saídas, além do fato que classes derivadas devem poder serem substituídas pela classe base (LISKOV; WING, 1994; MARTIN, 1996b).
(ISP) Interfaces magras	Objetos que implementam interface não devem implementar métodos não usados, interfaces com poucos métodos são mais reusadas (MARTIN, 1996c).
(DIP) Inversão de dependência	Sempre que há uma dependência, a sua origem deve ser mais estável que o objeto que depende, ou seja, uma abstração (MARTIN, 1996d; WANG; ZHOU, 2012).

- **O conceito de Código Limpo,** que cita que um código pode ser considerado limpo quando aproxima os três aspectos: simplicidade, flexibilidade e expressividade (THOMAZ ALMEIDA; MACHINI DE MIRANDA, 2010).

Conclui-se que, um código limpo é aquele que alia as boas práticas de programação (nomes descritivos e poucos comentários) e respeita a prática SOLID, pois juntas, entregam um código simples, flexível e expressivo.

Para validar e estudar se houve evolução de qualidade do sistema durante o seu desenvolvimento, foram criadas as métricas de código, essas métricas são métodos que analisam um código e retorna um número que expressa a sua qualidade decorrente de um aspecto (SCHNEIDEWIND, 1997).

Existem inúmeros aspectos dos quais pode-se extrair métricas, a partir disso é possível ter diversos tipos de medições, como: de tamanho, complexidade, manutenibilidade, classes, métodos, acoplamento, herança, sistema e outras (MEIRELLES, 2013). A Tabela 2 apresenta algumas das métricas existentes:

Métrica	**Descrição**
Linhas de código	Cálculo de quantas linhas de código uma classe ou método possui (MEIRELLES, 2013).
Complexidade ciclomática	Número de caminhos presentes em um código (MEIRELLES, 2013).
Coesão de método	Detecção de métodos que manipulam atributos distintos de uma classe (ANICHE, [s.d.]).
Acoplamento eferente	Quando uma classe depende de N outras, N será seu acoplamento eferente (ANICHE, [s.d.]).
Acoplamento aferente	Quando uma classe possui N dependentes, N será seu acoplamento aferente (ANICHE, [s.d.]).
Acoplamento lógico	Detecta quando arquivos de código distintos se dependem, ou seja, quando há alteração em um, outro também se altera (OLIVA; GEROSA, [s.d.]).

Tabela 2: Apresenta métricas de tipos diferentes com o seu conceito.

Apesar de ser fácil encontrar ferramentas para o cálculo de métricas, poucas, ou nenhuma, são usadas no dia a dia, a grande razão para isso está na complexidade dos resultados que são retornados para a análise do desenvolvedor. Também caem em desuso por não dar suporte a linguagens mais usadas, não apresentar uma heurística confiável e serem difíceis de usar.

Tais metodologias estão sendo alteradas para apresentar ferramentas que melhoram o quesito de visualização de resultados (BALOGH; BESZÉDES, 2013; WETTEL, 2009), por outro lado, novas ferramentas surgem fora da área acadêmica, no mundo open-source, trazendo suporte para linguagens populares, como o *PHPMetrics* e o *PHPLOC*, que realizam os cálculos de métricas primordiais para sistemas feitos na linguagem *PHP*, porém ainda pecando na visibilidade de resultados.

Com intuito de inovar, oferecendo uma maneira fácil e barata de avaliar a evolução da qualidade de um software, esse trabalho desenvolverá uma ferramenta chamada I*nsphptor*, que converte resultados complexos obtidos pelo calculo de métricas em dados simples, compreensíveis para qualquer homem (MURPHY, 2005), contando com o cálculo de métricas primordiais e extração de métricas através do *Git*, dando a possibilidade para que um gerente de equipe possa analisar o trabalho feito por cada um de seus membros.

2 METODOLOGIA

O processo de desenvolvimento de qualquer *software* deve ter um precedente, o tempo de planejamento é fundamental para se criar um sistema que realmente apresenta os requisitos que lhe foi prometido.

O período de construção e teste da aplicação *Insphptor* se divide em algumas partes, são elas: definição do escopo, seleção de métricas de código, seleção de métricas sociais, definição de módulos do sistema, execução de testes, e por fim, desenvolvimento de um sistema visualizador.

2.1 ESCOPO DA APLICAÇÃO

Antes de coletar os dados iniciais para o desenvolvimento, é preciso visualizar como o sistema funcionará depois de pronto, o *Insphptor* se trata de um programa em linha de comando disponibilizado em um arquivo compactado *phar* e executando em ambiente *php*.

A maneira de utilizar foi inspirada em ferramentas populares no desenvolvimento com *PHP*, o *Composer* e *PHPUnit*, onde há um arquivo de configuração presente na raiz do projeto a ser manipulado, ao executar o *software,* dentro da raiz do alvo, as configurações serão lidas e então o sistema de fato será executado. A seguir é apresentado o fluxo de funcionamento:

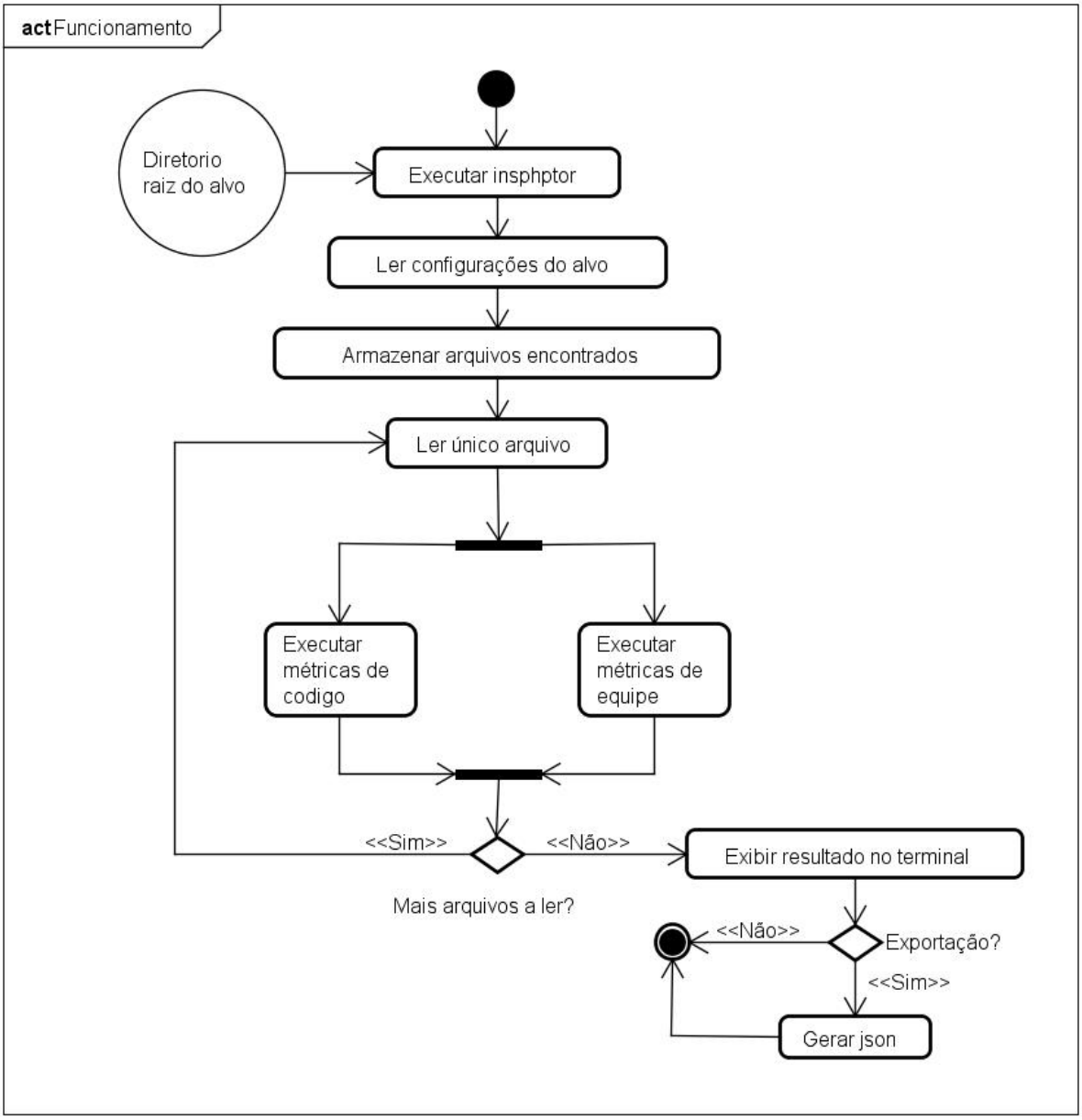

Figura 1. Diagrama de atividade, mostra o fluxo de funcionamento do sistema

2.2 MÉTRICAS DE CÓDIGO

Como mostrado na *Introdução*, é possível ter diversas métricas para calcular um mesmo aspecto de um código, porém, em um mesmo *software*, esse processo de revalidar um só aspecto leva tempo e processamento.

Visando uma maior produtividade e agilidade, foram selecionadas métricas de código que se referem a diferentes aspectos que fortificam o uso do *SOLID*, concluindo que mesmo com um número menor de métricas é possível qualificar um maior número de aspectos

2.2.1 Complexidade Ciclomática

Desenvolvida por Thomas J McCabe, em 1976, se define em calcular a quantidade de caminhos que um *software*, ou pedaço do mesmo, pode ter (MCCABE, 1976). Originalmente, a métrica analisa o sistema criando um grafo que expressa todo fluxo adotado pelo programa, no final, são retirados dados do grafo e calculados por uma fórmula matemática.

Visto que essa técnica é funcional apenas em sistemas com uma saída, em 1984, Warren A. Harrison definiu uma heurística adaptada para sistema com múltiplas saídas, onde são analisados os pontos de decisão do programa e os pontos de saída (HARRISON, 1984).

Atualmente, tal fórmula foi simplificada com intuito de ser funcional em qualquer tipo de sistema, usando a análise direta do código fonte, pode-se definir que a seguinte expressão retornará o valor do aspecto de complexidade:

Nr + Nc + Nd + 1

Onde:

- **Nr –** Quantidade de estruturas de repetições (*while*, *foreach*, *for* e do *while*).
- **Nc –** Quantidade de estruturas condicionais (*if*, *if else*, *else*, *case*, *try* e *catch*).
- **Nd –** Quantidade de estruturas de desvio bruto (*goto* e *break*).

2.2.2 Acoplamento Eferente

Citado pela primeira vez por Robert Martin, é o número de classes usadas dentro de um componente que foram declaradas fora do componente (MARTIN; MICAH, 2006), ou seja, é o número de dependências presentes em um arquivo ou classe.

A heurística simplista dessa métrica é a contagem de estruturas externas usadas na classe, como por exemplo: *use*, *require*, *require_once*, *include* e *include_once*.

2.2.3 Acoplamento Aferente

Também citada primeiramente por Robert Martin, é o número de classes fora do componente que dependem de classes desse componente (MARTIN; MICAH, 2006), ou seja, é o número de dependentes que uma classe possui.

O calculo é dado pela contagem de vezes em que a classe atual é importada em outras classes do sistema.

2.2.4 Linhas De Código

Talvez seja a métrica mais trivial existente, consiste em contar quantas linhas cada classe de código tem, é útil, pois considera que um arquivo exponencialmente maior que os demais oferece risco ao sistema como um todo, pois provavelmente terá mais peso para seu funcionamento.

2.2.5 Falta De Coesão

Essa métrica talvez seja a mais polêmica, seu calculo consiste em identificar métodos de uma mesma classe que manipulam atributos distintos (ANICHE, [s.d.]), por exemplo: em uma classe X, temos os métodos MA e MB, e os atributos A, B, C e D, o corpo do método MA faz uso dos atributos A e B, enquanto o corpo de MB faz uso apenas dos atributos C e D, permitindo assim a extração, pois a classe possui muitas responsabilidades.

A polêmica se deve ao fato de implementações simplistas não ignorarem *getters* e *setters*, deixando o número do resultado mais alto que o esperado.

2.2.6 Encapsulamento

Não se tem conhecimento de métricas relevantes para esse tipo de aspecto, porém, o princípio aberto-fechado da abordagem *SOLID* prevê que uma classe onde métodos públicos são maiores que métodos privados são pouco encapsuladas e menos legíveis.

Com isso, pode-se elaborar uma métrica para identificar classes onde os métodos públicos são maiores em número de linha que os métodos privados.

2.3 MÉTRICAS SOCIAIS

Em 1997, Thomas Ball et al, introduziram uma metodologia indicando que controladores de versões possuem uma grande gama de informações referentes ao projeto em si e que uma simples exploração nesses dados resultaria em análises relevantes (BALL; KIM; SIY, 1997).

Em todos *commits* são armazenados dados que informam aspectos alterados ou incluídos no sistema, com isso, uma básica mineração por palavras chaves em informações dos *commits* podem retornar dados preciosos para análise do produto, que ajudam a detecção e previsão de determinados problemas, como: impacto de alterações e previsão de falhas (SOKOL; ANICHE; GEROSA, 2013).

2.3.1 Acoplamento Lógico

Um ano após Ball e sua equipe relevarem o assunto, Harold Gall et al, observaram que arquivos de código podem possuir uma dependência lógica, definida como: um comportamento de mudanças iguais observadas entre diversos elementos durante a evolução de um sistema (GALL; HAJEK; JAZAYERI, 1998).

Em uma descrição informal, o acoplamento lógico se trata da detecção de arquivos que se alteram juntos, por exemplo: em todo *commit* incluso o arquivo A, o arquivo B também está sendo alterado.

2.3.2 *Bugs* No Código

Ao analisar projetos *open-source,* é possível encontrar a palavra *bug* (que se refere a falhas no sistema) em diversas mensagens de *commits*, com isso, é racional criar uma heurística que identifica arquivos de código que enfrentam mais problemas durante a evolução do sistema.

O calculo para essa métrica é considerado simples, deve-se encontrar primeiramente palavras chaves referentes a erros, como: *bug*, falha, *fix*, *fail* e *broken*. E então, pesquisá-las em mensagens de *commits* do projeto, ao detectá-las, será registrado que tais arquivos são propícios a falhas.

2.3.3 Produtividade

Ao enviar um *commit*, é registrado no controlador de versão o autor da modificação, com isso é possível contar quantas vezes um certo membro da equipe enviou uma nova alteração, tendo a conclusão de que a cada envio, sua contribuição para o término é maior, igual à sua produtividade.

2.3.4 Arquivos Instáveis

A partir dos *commits*, é possível captar quantas linhas de código foram incluídas ou deletadas de arquivos específicos, é notório que arquivos que sofrem mais alterações durante a evolução são aqueles mais instáveis do *software*, logo merecem mais atenção da equipe.

2.3.5 Construtores De Código

Essa métrica foi elaborada com um simples objetivo, relatar quais contribuidores do repositório mais escrevem novas funcionalidades. O calculo é feito através da soma de linhas adicionadas em todos *commits* dados pelo membro.

2.3.6 Destrutores De Código

Métrica criada para ser o inverso da anterior, informando os colaboradores que mais apagam linhas de código do sistema, efetuando o calculo em cada *commit* daquele membro.

2.4 MÓDULOS DO SISTEMA

Seguindo a abordagem *SOLID*, uma aplicação deve ser capaz de se expandir ou encolher a qualquer momento, sem afetar o funcionamento geral do sistema. Para isso, a aplicação deve se dividir em módulos e fazer uso ampliado dos princípios de Robert Martin.

Além disso, para transformar o *Insphptor* em um *software* estável, padrões de projetos foram explorados com intuito de solidificar classes e unificar módulos, pois definem a arquitetura de módulos e interconexões (MARTIN, 2000). A seguir, a Figura 2 exibe o diagrama dos módulos presentes na aplicação:

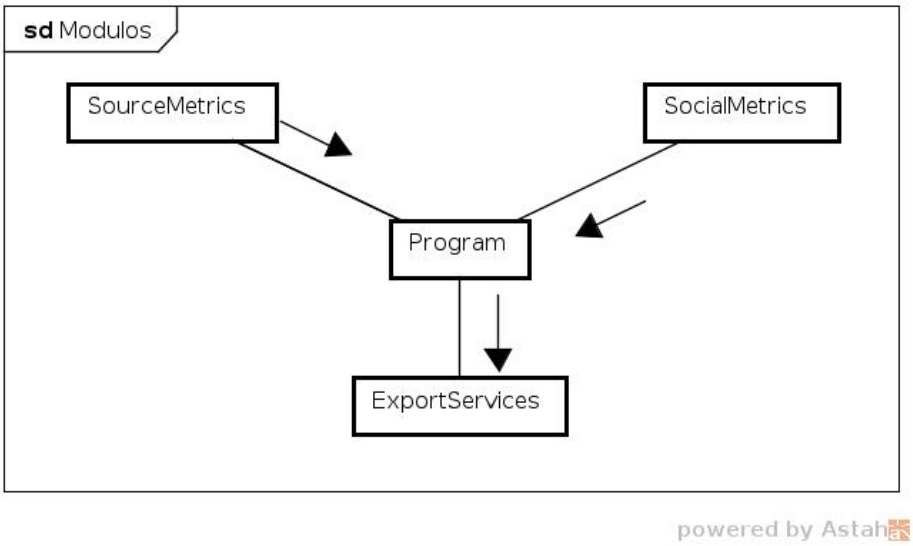

Figura 2. Diagrama de blocos, mostra a separação do código

Com essa separação de módulos, é possível incluir ou remover métricas do calculo final sem qualquer tipo de dificuldade, também deixando um módulo a parte para realizar todo o fluxo do sistema e controle de comandos recebidos do terminal, e, por fim, um módulo para exportar todo o resultado, sendo possível ampliar para novos formatos.

2.5 TESTES DO SISTEMA

Durante o desenvolvimento e a evolução de uma aplicação, é importante ter total controle do funcionamento, para que uma nova funcionalidade inserida não afete outra criando falhas, para verificar esse processo existem os testes automatizados, que são arquivos de código com objetivo de testar parte de códigos em produção, verificando se a saída dada ainda é válida.

Normalmente, testes são escritos depois que uma nova funcionalidade é codificada, porém, Kent Beck desenvolveu a técnica do

TDD (desenvolvimento guiado por testes), ao ver que escrevendo os testes automatizados antes mesmo do código, resultaria em um código mais limpo e refatorado (BECK, 2002). Levando isso em consideração, o sistema desenvolvido nesse trabalho terá a prática do *TDD* usando o *framework* de testes *PHPUnit*.

Além de testes unitários automatizados, serão desenvolvidos algoritmos que recolhem repositórios populares do *GitHub* para executar a análise da qualidade, possibilitando assim coletar dados para validar o funcionamento correto da aplicação *Insphptor* e gerar um resultado final satisfatório.

2.6 SISTEMA VISUALIZADOR

O escopo da aplicação determina a sua execução via terminal, onde é mostrado um resultado simplista no console. Porém, com a exportação de resultados, inicialmente via arquivos *json*, se torna possível o desenvolvimento de sistemas para visualizações personalizadas de resultados, tendo intuito expandir ao máximo o entendimento de métricas em diferentes tipos de ambientes.

Por padrão, o *Insphptor* disponibiliza um visualizador desenvolvido com o *framework VueJS*, que utiliza diagramas de Kiviat para exibição dos resultados, pois podem mostrar múltiplas variáveis em um plano 2D, tornando fácil o entendimento para qualquer pessoa.

3 CRONOGRAMA

Atividades	**Meses/2017**				
	Ago	**Set**	**Out**	**Nov**	**Dez**
Configurar o ambiente de desenvolvimento *php*	**X**				
Escolher as extensões a usar	**X**				
Configurar *autoload* e testes	**X**				
Criar comandos do sistema	**X**				
Modelar módulo *SourceMetrics*		**X**			
Implementar métricas de código		**X**	**X**		
Arquitetar módulo *SocialMetrics*		**X**			
Implementar métricas sociais			**X**	**X**	
Modelar módulo *ExportServices*		**X**			
Criar exportação de resultado em *json*			**X**	**X**	
Desenvolver sistema visualizador				**X**	
Escrever o artigo do projeto				**X**	**X**

REFERÊNCIAS

ANICHE, M. F. Orientação a Objetos e SOLID para Ninjas Projetando classes flexíveis. [s.d.].

ANICHE, M. F. **Orientação a Objetos e SOLID para Ninjas**. [s.l: s.n.].

BALL, T.; KIM, J.-M.; SIY, H. P. If your version control system could talk. **ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering**, n. April, 1997.

BALOGH, G.; BESZÉDES, Á. CodeMetropolis - A minecraft based collaboration tool for developers. **2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013**, 2013.

BECK, K. Test-Driven Development By Example. 2002.

GALL, H.; HAJEK, K.; JAZAYERI, M. Detection of logical coupling based on product release history. **Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)**, v. 1998, p. 190–198, 1998.

HARRISON, W. A. Applying Mccabe's complexity measure to multiple-exit programs. 1984.

LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. **ACM Transactions on Programming Languages and Systems**, v. 16, n. 6, p. 1811–1841, 1994.

MARTIN, R. C. The Open-Closed Principle. **C++ Report**, p. 1–14, 1996a.

MARTIN, R. C. The Liskov Substitution Principle. **More C++ gems**, p. 1–11, 1996b.

MARTIN, R. C. The Interface Segregation Principle. **C++ Report**, n. c, p. 1–13, 1996c.

MARTIN, R. C. The Dependency Inversion Principle. **C++ Report**, v. 8, p. 61–66, 1996d.

MARTIN, R. C. Design Principles and Design Patterns. n. c, p. 1–34, 2000.

MARTIN, R. C.; MICAH, M. **Agile Principles, Patterns, and Practices in C#**. [s.l: s.n.].

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, v. 2, n. 4, p. 308–320, 1976.

MEIRELLES, P. Monitoramento de métricas de código-fonte em projetos de software livre. 2013.

MEYER, B. Object-oriented software construction. **Science of Computer Programming**, v. 12, p. 88–90, 1989.

MURPHY, G. C. **Big data. Little Brain.**, 2005.

OLIVA, G.; GEROSA, M. **Dependências Lógicas e Suas Aplicações no Campo de Predição de Mudanças: Uma Revisão da BibliografiaValinhos.Ime.Usp.Br**, [s.d.]. Disponível em: <http://valinhos.ime.usp.br:55080/baile/sites/ime.usp.br.baile/files/Revisao_dependencias_logicas.pdf>

SCHNEIDEWIND, N. Software metrics model for quality control. **Software Metrics Symposium**, p. 127–136, 1997.

SOKOL, F. Z.; ANICHE, M. F.; GEROSA, M. A. MetricMiner: uma ferramenta web de apoio à mineração de repositórios de software. **IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013**, p. 142–146, 2013.

THOMAZ ALMEIDA, L.; MACHINI DE MIRANDA, J. Código Limpo e seu Mapeamento para Métricas de Código Fonte. 2010.

WALTERS, G. F.; MCCALL, J. A. Software Quality Metrics for Life-Cycle Cost-Reduction. **IEEE Transactions on Reliability**, v. R-28, n. 3, p. 212–220, 1979.

WANG, H.; ZHOU, H. Basic design principles in software engineering. **Proceedings - 4th International Conference on Computational and Information Sciences, ICCIS 2012**, p. 1251–1254, 2012.

WETTEL, R. Visual exploration of large-scale evolving software. **2009 31st International Conference on Software Engineering - Companion Volume, ICSE 2009**, p. 391–394, 2009.