

André de Paula Terceiro

***Utilização de ferramentas de análise estática para
diminuição de erros e aumento da manutenibilidade em
código fonte PHP***

São Paulo

2014

André de Paula Terceiro

***Utilização de ferramentas de análise estática para
diminuição de erros e aumento da manutenibilidade em
código fonte PHP***

Monografia apresentada à banca examinadora da
Universidade São Judas Tadeu, como exigência
parcial para obtenção do grau de especialização em
Engenharia de Software, sob orientação do Prof.
Ms. Aluizio Saiter Mota

Orientador:
Prof. Ms. Aluizio Saiter Mota

São Paulo

2014

André de Paula Terceiro

***Utilização de ferramentas de análise estática para
diminuição de erros e aumento da manutenibilidade em
código fonte PHP***

Monografia apresentada à banca examinadora da
Universidade São Judas Tadeu, como exigência
parcial para obtenção do grau de especialização em
Engenharia de Software, sob orientação do Prof.
Ms. Aluizio Saiter Mota

Aprovada em junho de 2014

Agradecimentos

Ao Prof. Ms. Aluizio Saiter Mota, orientador desta monografia, e aos demais professores da Universidade São Judas Tadeu pelos ensinamentos.

Ao meu amigo Thiago Pelizoni pelo apoio e por disponibilizar o modelo LaTeX que facilitou a formatação deste trabalho.

Resumo

Revisões podem ser realizadas durante diversas etapas do processo de desenvolvimento de software, localizando ajustes que podem evitar diversos tipos de problemas e um alto custo de correções de erros.

A utilização de ferramentas para auxílio ou complemento às revisões humanas tem um impacto positivo, pois pode tornar as revisões mais rápidas, detalhadas e precisas, além de liberar a mão de obra humana de tarefas repetitivas, pouco atrativas e triviais. Além disso, algumas ferramentas podem gerar relatórios com indicadores diversos, muitas vezes de extração trabalhosa para mão de obra humana. Com tais informações um especialista pode tomar uma decisão mais precisa.

Uma parte importante do foco das revisões é o código fonte do software. Neste caso, as revisões podem trazer diversos benefícios, como a localização de bugs, vulnerabilidades de segurança e a detecção de melhorias a serem realizadas em diversos aspectos, como manutenibilidade, abrangência de documentação do código fonte, entre outras.

Este trabalho tem como objetivo mostrar como algumas ferramentas de análise estática de código fonte PHP podem contribuir com a atividade de revisão para a diminuição de erros e aumento da manutenibilidade. Será apresentado o foco de cada ferramenta, como elas podem ser obtidas, parametrizadas, integradas a outras e usadas durante o processo de desenvolvimento de software, agindo diretamente em um problema ou gerando relatórios auxiliares para tomada de decisão.

Palavras-chave: PHP, manutenibilidade, bugs, ferramentas, análise estática

Abstract

Software revisions may be performed during various stages of the software development process, tracking adjustments that can prevent many types of problems and a high cost of bug fixes.

The use of tools to support or supplement human reviews has a positive impact, because revisions can become more accurate, fast and detailed, besides free the reviewer of repetitive, unattractive and trivial tasks. Moreover, some tools can generate reports with various indicators, often laborious extraction for human labor. With such information a specialist can make a more accurate decision.

The source code of the software is an important focus of the software reviews. In this case, software reviews can bring many benefits, such as detection of bugs, security vulnerabilities and potential improvements in various aspects, such as maintainability and coverage of the source code documentation.

This paper aims to show how some static analysis tools for PHP source code can contribute to the software review activity for error reduction and increase maintainability . The focus of each tool will be presented, how they can be obtained, parameterized, integrated to other tools and used during the software development process, acting directly on a problem or generating reports for human analysis.

Keywords: PHP, maintainability, bugs, tools, static analysis

Sumário

Lista de Figuras

1	Declaração do Problema	p. 9
2	Introdução	p. 11
2.1	O conceito de software	p. 11
2.2	Engenharia de software	p. 12
2.3	Processo de desenvolvimento e ciclo de vida do software	p. 13
3	Verificação e validação de software	p. 16
3.1	Técnicas estáticas e dinâmicas	p. 16
3.2	Análise estática automatizada	p. 19
4	Manutenção de software	p. 22
4.1	Manutenibilidade e suas subcaracterísticas	p. 22
4.2	Métricas e suas relações com a manutenibilidade	p. 23
5	PHP	p. 27
5.1	História	p. 28
5.2	PHP como linguagem para desenvolvimento web	p. 29
5.3	PHP como linguagem script de linha de comando	p. 31
5.4	PHP para aplicações desktop: PHP-GTK	p. 31
6	Linters	p. 33
6.1	História	p. 33
6.2	Linters para PHP e integrações com outras ferramentas	p. 33

7 Ferramentas para análise estática em relação a padrões de codificação	p. 37
7.1 Padrões de codificação	p. 37
7.2 Padrões de codificação para PHP	p. 38
7.2.1 Padrão de codificação do Wordpress	p. 40
7.2.2 Padrão de codificação Pear	p. 44
7.2.3 Padrões de codificação PSR	p. 47
7.2.4 Outros padrões de codificação	p. 49
7.3 PHP Code Sniffer	p. 52
7.3.1 Padrões de codificação disponíveis e criação de novos padrões	p. 53
7.3.2 Refinamento da lista de arquivos e verificações efetuadas	p. 55
7.3.3 Integração com outras ferramentas	p. 56
7.4 PHP Code Standards Fixer	p. 58
8 PHP Depend	p. 61
8.1 Instalação e execução	p. 61
8.2 Relatório de Abstração x Instabilidade	p. 61
8.3 Relatório de visão geral em forma de pirâmide	p. 63
8.4 Relatório geral de métricas em formato XML	p. 65
9 Outras ferramentas	p. 66
9.1 PHPMD	p. 66
9.2 PHPCPD	p. 66
9.3 PHPDCD	p. 67
10 Conclusão	p. 68
Referências Bibliográficas	p. 69

Lista de Figuras

2.1	Camadas da engenharia de software	p. 13
2.2	Elementos de um processo de software	p. 13
2.3	Modelo de processo cascata	p. 14
3.1	Comparação da formalidade usada em alguns tipos de revisões de software	p. 18
3.2	Aplicação de V&V estática e dinâmica em diferentes tipos de artefatos	p. 19
4.1	Exemplo de grafo de fluxo	p. 25
5.1	Uso de linguagens de programação server-side para sites web	p. 27
5.2	Diagrama simplificado de uma requisição web sendo interpretada pelo PHP	p. 29
5.3	Exemplo de um script PHP simples	p. 30
5.4	Conteúdo HTML gerado com base em um script PHP	p. 30
5.5	Exemplo de script PHP CLI simples	p. 32
5.6	Demonstração da execução do script da figura 5.5	p. 32
5.7	Janelas geradas via PHP-GTK	p. 32
6.1	Análise estática com o lint	p. 34
6.2	Arquivo exemplo com erros de sintaxe	p. 34
6.3	Análise do arquivo mostrado na figura 6.2 através do comando "php -l"	p. 34
6.4	Verificação de erros de sintaxe através do IDE Sublime Text	p. 35
6.5	Exemplo de execução do programa PHPLint via interface gráfica	p. 36
7.1	Exemplo de uso do mecanismo tradicional do PHP para tratar erros	p. 39
7.2	Exemplo de uso de exceções para o tratamento de erros	p. 40
7.3	Trecho de código do plugin Akismet	p. 41
7.4	Outro trecho de código do plugin Akismet	p. 41
7.5	Trecho de código do plugin FV-Antispam	p. 42

7.6	Trecho de código do plugin BB Press	p. 42
7.7	Trecho do padrão de codificação do Wordpress	p. 43
7.8	Trecho do padrão de codificação Pear	p. 47
7.9	Trecho do padrão de codificação PSR-2	p. 48
7.10	Trecho do padrão de codificação do Zend Framework 2	p. 49
7.11	Trecho do padrão de codificação do Framework CakePHP	p. 50
7.12	Trecho do padrão de codificação do Symfony Framework	p. 52
7.13	Instalação do PHP Code Sniffer	p. 52
7.14	Exemplo de execução do PHP Code Sniffer	p. 53
7.15	Arquivo ruleset.xml do padrão de codificação PEAR no PHP Code Sniffer	p. 54
7.16	Exemplo de classe de inspeção no PHP Code Sniffer	p. 55
7.17	Arquivo ruleset.xml do padrão PSR-2 no PHP Code Sniffer	p. 55
7.18	Solicitando ao PHP Code Sniffer que ignore o arquivo todo	p. 56
7.19	Solicitando ao PHP Code Sniffer que ignore parte do arquivo	p. 56
7.20	Acionando o PHP Code Sniffer através de uma hook pre-commit do Subversion	p. 57
7.21	Integração do PHP Code Sniffer com o Subversion pelo comando “ <i>svn-blame</i> ”	p. 57
7.22	Exemplo de arquivo antes e depois da execução do PHP Code Standards Fixer	p. 59
8.1	Instalação do PHP Depend via ferramenta Pear em um terminal Linux	p. 61
8.2	Geração do gráfico Abstração x Instabilidade através do PHP Depend	p. 62
8.3	Exemplo de gráfico instabilidade x abstração gerado pela ferramenta PHP Depend	p. 62
8.4	Agrupamento das métricas no relatório em forma de pirâmide	p. 63
8.5	Relatório em forma de pirâmide com ilustrações dos cálculos executados	p. 64
8.6	Exemplo de trecho do XML geral de métricas gerado pelo PHP Depend	p. 65

1 Declaração do Problema

Erros, apesar de indesejáveis, ocorrem no decorrer do desenvolvimento de um software. Pressman afirma que, pelo envolvimento humano, é natural que isto ocorra (PRESSMAN, 2011). Desta forma, além de elaborar estratégias para evitar que os erros ocorram, é necessário ter mecanismos para o enfrentamento destes erros.

Testes são uma das formas para detecção de tais erros, porém eles só podem ser realizados sobre versões executáveis do software, seja o programa final ou protótipos. Além disso, eles não verificam determinados atributos de qualidade de um software, como a manutenibilidade (SOMMERVILLE, 2007) . Uma estratégia complementar aos testes é a realização de revisões, que cobrem estas lacunas.

Um dos alvos a serem revisados é o código fonte do programa. Algumas softwares, chamados de ferramentas de análise estática de código fonte (por não executarem o programa analisado), podem auxiliar a realização destas revisões . O benefício do uso de tais ferramentas são diversos, como por exemplo:

- Redução de custos: As atividades de revisão, apesar de benéficas, podem ampliar o tempo necessário para o desenvolvimento do software e o custo até a liberação das primeiras versões. Uma ferramenta possivelmente não faça uma revisão tão completa quanto um ser humano especializado, porém, para os aspectos que cobrir, tenderá a ser muito mais veloz que um humano;
- Uniformidade e precisão no trabalho gerado: ferramentas conseguem realizar um trabalho com alta repetitibilidade e exatidão nos resultados;
- Liberação da mão de obra humana para tarefas não elementares: algumas tarefas de revisão podem não ser complexas, mas devido ao volume, trabalhosas. Um exemplo é o confronto de um programa em relação a um padrão de codificação. Tal tarefa pode ser executada por uma ferramenta, o que permite liberar a mão de obra humana deste trabalho e também facilitar a própria realização da revisão humana, atuando como um pré filtro.

PHP é um linguagem de programação popular (principalmente no segmento de desenvolvimento web) e poderosa. Sem o devido cuidado, pode-se gerar scripts de difícil entendimento, sem padrões de nomenclatura, com alta complexidade ciclomática, mistura de conteúdo de apresentação (como

HTML) com regras de negócio, programas com baixa testabilidade, entre outros possíveis problemas, que ficam mais evidentes quando os sistemas são complexos e envolvem grandes equipes.

Já existem diversas ferramentas que podem ser usadas para resolver estes possíveis problemas, como frameworks MVC, ferramentas para testes unitários e a análise estática de código fonte. Algumas das ferramentas para análise estática de código fonte para PHP serão abordadas neste trabalho, com o foco na diminuição de erros e aumento da manutenibilidade.

2 *Introdução*

2.1 O conceito de software

O termo software muitas vezes é associado somente a programas de computador. Porém, além de um programa de computador, o software é composto pelos seus dados de configuração e sua documentação (SOMMERVILLE, 2007).

O software evoluiu muito desde a criação do primeiro programa de computador. Há 50 anos era muito difícil prever que o software teria a atual importância, sendo indispensável em diversas áreas, possibilitando a criação de novas tecnologias (como a nanotecnologia e a engenharia genética), estendendo e gerando mudanças radicais em áreas como as telecomunicações e a indústria gráfica. Também era difícil prever a disseminação dos computadores pessoais, softwares sendo comprados em lojas de bairro, softwares evoluindo de produtos para serviços, entre outras coisas (PRESSMAN, 2011).

O desenvolvimento de um software como um produto para usuários finais continua existindo, porém há tempos diversas empresas desenvolvem um software, disponibilizam gratuitamente e vendem serviços com base neles, como por exemplo anúncios de publicidade. Exemplos são algumas empresas mantenedoras de motores de busca e de redes sociais.

Diversos dispositivos, dos mais variados tamanhos e com diferentes finalidades, podem executar um software. Softwares estão presentes em celulares, tablets, televisões, equipamentos de cozinha, elevadores, aviões, equipamentos ligados à medicina, sistemas de controles de usinas, entre outros.

Para a evolução dos produtos de software demandou-se um aprimoramento de antigas técnicas e a criação de novas. O papel de um programador solitário de antigamente foi substituído por equipes de especialistas que em conjunto abrangem as diversas atividades relacionadas ao desenvolvimento de um software. Porém a complexidade dos softwares aumentou e ainda persistem dúvidas em diversas áreas, como em relação a uma forma de entregar um software sem erros, de otimizar o custo e o tempo do desenvolvimento de um software e como medir o progresso durante desenvolvimento (PRESSMAN, 2011).

2.2 Engenharia de software

Engenharia de Software pode ser definida como "uma disciplina de engenharia relacionada com todos os aspectos da produção de software, desde os estágios iniciais de especificação dos sistemas até sua manutenção, depois que este entrar em operação". Seu foco é "o desenvolvimento dentro de custos adequados de sistemas de software de alta qualidade"(SOMMERVILLE, 2007).

Ao utilizar "disciplina de Engenharia", Sommerville diferencia a Engenharia de Software da Ciência da Computação, indicando que a primeira se dedica aos problemas práticos da produção de software e a segunda aos conceitos teóricos e métodos relacionados a sistemas de software e computadores, nem sempre aplicáveis aos problemas reais relacionados à produção de software (SOMMERVILLE, 2007). Além disso, a Engenharia de Software não é uma disciplina da Ciência da Computação, assim como a Engenharia Elétrica não é uma disciplina da Física da Eletricidade. E as vocações de um profissional da Engenharia de Software são diferentes de um cientista da computação, assim como as vocações de um engenheiro são diferentes das vocações de um físico (PAULA FILHO, 2000).

Ao utilizar "todos aspectos da produção de software", Sommerville indica a Engenharia de Software não foca apenas aspectos técnicos, mas também atividades de apoio à produção de software, como por exemplo atividades de gerenciamento de projetos (SOMMERVILLE, 2007).

O conceito de Engenharia de Software foi proposto inicialmente em 1968, em uma conferência em que se debatia a crise do software¹. Constatou-se nesta época que a forma de desenvolvimento informal predominante até então não era adequada para trabalhar-se com as demandas de softwares cada vez mais complexos. Eram procuradas soluções para problemas como grandes atrasos em projetos, custos acima dos previstos, baixa confiabilidade e manutenibilidade dos softwares gerados, entre outros problemas (SOMMERVILLE, 2007).

A Engenharia de Software é uma tecnologia dividida nas seguintes camadas: ferramentas, métodos, processos e foco na qualidade, sendo esta última a base em que se apoiam todas outras camadas, como mostra a figura 2.1. A camada de processos rege o desenvolvimento racional do software, estabelecendo quais métodos são utilizados em quais circunstâncias, os artefatos produzidos, o estabelecimento de marcos e o gerenciamento da qualidade e das mudanças. Os métodos indicam as técnicas para construir os softwares e as ferramentas fornecem os apoios automatizados (totalmente ou parcialmente) para apoiar os processos e métodos (PRESSMAN, 2011).

¹A expressão "crise do software" historicamente faz referência a um conjunto de problemas que, de forma recorrente, são enfrentados durante o desenvolvimento do software (REZENDE, 2005).



Figura 2.1: Camadas da engenharia de software
Fonte: PRESSMAN (2011)

2.3 Processo de desenvolvimento e ciclo de vida do software

Um processo de engenharia é composto por “uma série de atividades interligadas que transformam uma ou mais entradas em saídas, enquanto consomem recursos para realizar a transformação”. O gerenciamento de requisitos, o projeto do software, a construção, os testes e a gerência de configuração são exemplos destas atividades em relação à Engenharia de Software. Elas são realizadas por engenheiros de software para desenvolver, manter e operar o software. A figura 2.2 mostra que um processo de software deve incluir seus critérios de entrada e saída, e decompor as atividades em tarefas, que são pequenas unidades de trabalho gerenciáveis (BOURQUE; FAIRLEY, 2014).

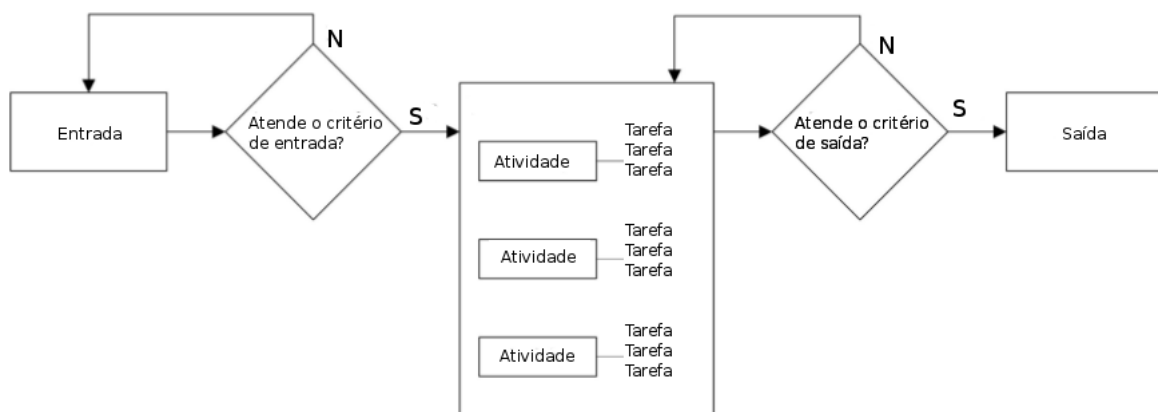


Figura 2.2: Elementos de um processo de software
Fonte: BOURQUE; FAIRLEY (2014)

No decorrer do desenvolvimento do software, não é gerado apenas o software executável, mas também diversos artefatos, como documentos de requisitos, documentos do projeto do software, entre outros, independente do formato em que estejam estes artefatos. Tais artefatos são importantes para se chegar ao software final. Um processo bem definido, com papéis, atividades e resultados desejados claros, tende a permitir um maior controle sobre os produtos resultantes do desenvolvimento do software.

Na engenharia de software, nem sempre a melhor estratégia é o planejamento e execução do pro-

jeto de forma sequencial, como normalmente é feito na engenharia civil (MARTINS, 2007). Existem diferentes modelos de processos, variando em diversas questões como uso ou não de um desenvolvimento iterativo (e até a forma como as iterações ocorrem), o quanto são prescritivos, os papéis exigidos, entre outros detalhes. A escolha do modelo pode ser feita usando critérios como a experiência da equipe com um modelo específico, a complexidade do software a ser desenvolvido, o tamanho da equipe, a experiência da equipe com o desenvolvimento de um software com características semelhantes, a variabilidade dos requisitos, entre outras coisas.

A existência de diferentes modelos de processos indica que não existe um modelo ideal para todas as situações. O desenvolvimento de um sistema crítico exige uma abordagem diferente em relação a um sistema de negócios, em que os requisitos podem mudar rapidamente. Desta forma, o processo de desenvolvimento mais eficaz tende a ser diferente em determinados casos (SOMMERVILLE, 2007).

O primeiro modelo de processo publicado é chamado de cascata (devido ao encadeamento de uma fase com a outra) e originou-se de processos mais gerais da Engenharia de Sistemas (SOMMERVILLE, 2007). Ele é representado pela figura 2.3. Pressman utiliza também a nomenclatura “ciclo de vida clássico” para o modelo cascata (PRESSMAN, 2011).

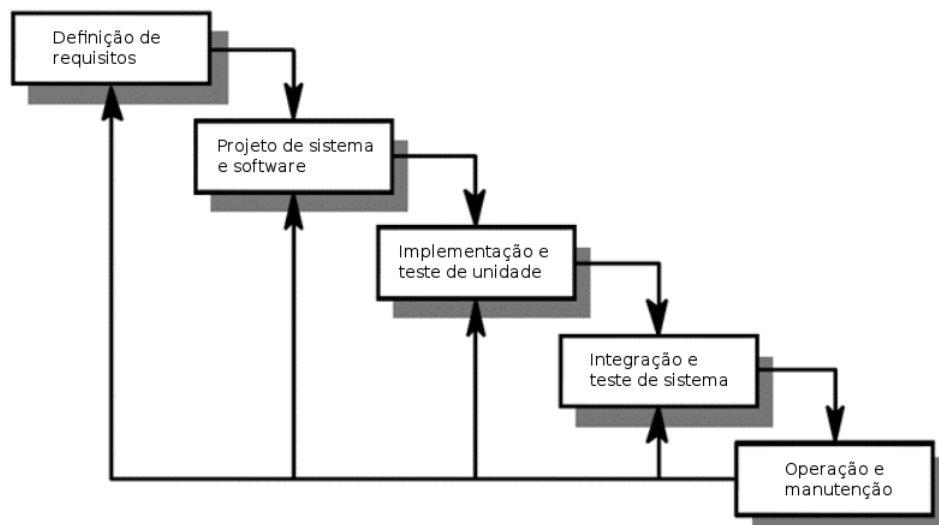


Figura 2.3: Modelo de processo cascata
Fonte: SOMMERVILLE (2007)

O modelo cascata sugere uma abordagem sequencial e sistemática para o desenvolvimento do software (PRESSMAN, 2011). Os estágios deste modelo representam as atividades fundamentais do desenvolvimento de software (SOMMERVILLE, 2007):

- Análise e definição de requisitos: neste estágio são extraídos os requisitos para o software a ser desenvolvido;
- Projeto de sistema e software: o software é projetado nesta etapa, sendo definidos detalhes como a arquitetura geral do software, aspectos relacionados a hardware, abstrações do sistema

de software e suas relações;

- Implementação e teste de unidade: neste estágio, unidades do software são desenvolvidas e seus testes são realizados;
- Integração e teste de sistema: as unidades individuais do programa são integradas e testadas como um sistema completo, sendo verificado se o sistema atende aos requisitos;
- Operação e manutenção: geralmente esta é a etapa mais longa do ciclo de vida, em que o sistema é instalado e colocado em operação. Passa a ser realizada a manutenção do software, abrangendo a correção de erros não detectados os estágios anteriores, atendimento de requisitos não identificados anteriormente ou adaptações para ambientes operacionais diferentes.

Há outros modelos de processos, como o modelo V (derivado do modelo cascata), modelo evolucionário, modelo espiral, modelos incrementais, entre outros (PRESSMAN, 2011). A utilização das ferramentas de análise estática no contexto apresentado neste texto não fica limitado ao desenvolvimento seguindo o modelo cascata. Este foi utilizado como exemplo apenas por ser um modelo clássico e de fácil entendimento.

3 *Verificação e validação de software*

A verificação e validação, também referenciada como V&V, engloba as atividades necessárias para certificar-se que o software em desenvolvimento atende a sua especificação e entrega a funcionalidade esperada. Os diversos estágios do processo de desenvolvimento de software podem contar com atividades de verificação e/ou validação (SOMMERVILLE, 2007).

A diferença entre verificação e validação é mostrada por Pressman: “Verificação refere-se ao conjunto de tarefas que garantem que o software implementa corretamente uma função específica. Validação refere-se a um conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente” (PRESSMAN, 2011). Resumidamente, Boehm expõe esta diferença da seguinte forma (BOEHM, 1984):

- Verificação: “estamos construindo o produto corretamente?”;
- Validação: “estamos construindo o produto correto?”.

A verificação e validação é um tópico derivado do estudo da qualidade de software. Esta, por sua vez, abrange questões gerais relacionadas à qualidade do software, tanto do processo quanto do produto de software (ABRAN et al., 2004).

Diversas atividades de SQA¹ são englobadas pela verificação e validação, como revisões técnicas², monitoramento de desempenho, simulação, testes diversos (de desenvolvimento, de usabilidade, de qualificação etc), análise de algoritmos, entre outras (PRESSMAN, 2011).

3.1 **Técnicas estáticas e dinâmicas**

As técnicas V&V de podem ou não envolver a execução do software, gerando a seguinte classificação:

¹Software Quality Assurance (Garantia de Qualidade de Software)

²Revisão técnica pode ser definida como “a verificação de um produto de trabalho de software por pessoas que não sejam o autor, com o objetivo de identificar defeitos (desvios de especificações ou padrões) e oportunidades de melhorias.” (WIEGERS, 2012)

- Técnicas dinâmicas: estão relacionadas à execução de uma implementação do software utilizando dados de teste;
- Técnicas estáticas: técnicas deste tipo não envolvem a execução do software.

As técnicas dinâmicas geralmente são realizadas através de testes de software, porém podem englobar outras técnicas, como simulação e execução simbólica (ABRAN et al., 2004). Os testes podem englobar desde pequenas unidades do software, a integração entre partes do software e o software como um todo. Segundo Pressman, são o último estágio em que a qualidade pode ser estimada e os erros podem ser descobertos antes da entrega, mas não podem ser vistos como uma rede de segurança, pois “você não pode testar a qualidade. Se a qualidade não está lá antes de um teste, ela não estará lá quando o teste terminar” (PRESSMAN, 2011).

Ao se utilizar a expressão “teste de software”, segundo Sommerville (SOMMERVILLE, 2007) e o SWEBOK (ABRAN et al., 2004), está implícito que se refere a uma técnica dinâmica, envolvendo a execução do software. Este é o significado de “teste” utilizado neste texto. Porém cabe ressaltar que o glossário do ISTQB³ contém a expressão “teste estático”, referenciando-se a técnicas que não envolvem a execução do software (ISTQB, 2014).

O comportamento do software em resposta aos testes pode ser analisado tanto em relação às expectativas dos usuários, quanto em relação a uma especificação. Segundo o SWEBOK, no primeiro caso está se fazendo uma validação e, no segundo caso, uma verificação (ABRAN et al., 2004). Sommerville utiliza a nomenclatura “teste de validação” para os testes em relação à expectativa do usuário e “teste de defeitos” para o conformtamento do resultado do teste em relação à especificação (SOMMERVILLE, 2007).

As técnicas estáticas envolvem a análise do software e dos documentos gerados durante o desenvolvimento do software, sem a necessidade da execução do software para isto (ABRAN et al., 2004). Diversos artefatos podem ser submetidos à análise estática, como documentos de requisitos, diagramas de projeto e o código fonte. A utilização de métodos formais, revisões técnicas e análise estática automatizada são formas de se realizar verificação e validação estática.

Os métodos formais tem sido usados principalmente no desenvolvimento de sistemas que contém partes críticas, por exemplo relacionadas a questões de segurança (ABRAN et al., 2004). Eles baseiam-se em representações matemáticas do software, geralmente como uma especificação formal. Exigem análises detalhadas e notações especializadas (compreendidas apenas por uma equipe especialmente treinada). Desta forma, tem a desvantagem de serem onerosos e consumirem muito tempo. Por outro lado, as especificações formais são menos propícias a conter anomalias. A abordagem Cleanroom para o desenvolvimento de software, cujo objetivo é o “software com defeito zero”, baseia-se na utilização métodos formais (SOMMERVILLE, 2007).

³International Software Testing Qualifications Board

Pode-se realizar tanto verificação quanto validação através de técnicas estáticas. Ao realizar revisões de requisitos, por exemplo, está se realizando uma validação (ABRAN et al., 2004). Já ao se revisar um código fonte diretamente ou via ferramenta de análise estática, está se realizando uma verificação (SOMMERVILLE, 2007).

As revisões são um “filtro” para a gestão de qualidade e podem ser também uma forma de se obter maior uniformidade nos produtos gerados (PRESSMAN, 2011). Há diferentes métodos de se realizar revisões. Uma das formas de comparar estes métodos é pelo nível de formalidade, como mostra a figura 3.1.

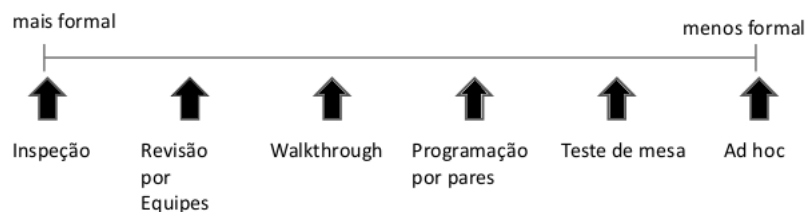


Figura 3.1: Comparação da formalidade usada em alguns tipos de revisões de software

Fonte: WIEGERS (2002)

Uma revisão Ad Hoc é realizada sem planejamento e preparação. Um exemplo deste tipo de revisão é um programador solicitar informalmente uma ajuda a outro por alguns minutos, com a finalidade de identificar um problema pontual no código em que está trabalhando. Neste caso, também não se utiliza um checklist para auxiliar o processo de revisão (WIEGERS, 2002).

Já uma inspeção (também referenciada como “inspeção formal”) tem estágios bem definidos (planejamento, visão geral, preparação, reunião, retrabalho, acompanhamento e análise causal) e seus membros, coordenados por um moderador, possuem papéis específicos, também bem definidos. Checklists específicos para cada tipo de artefato inspecionado são usados (WIEGERS, 2002). As inspeções geralmente enfocam o código-fonte, mas podem ser feitas também sobre artefatos diversos usados no desenvolvimento do software, como documentos de requisitos e modelos de projeto (SOMMERVILLE, 2007).

A figura 3.2 mostra como inspeções e testes podem ser utilizados em diferentes tipos de artefatos gerados durante o processo de desenvolvimento. Sommeville a usa como base de comparação da aplicação de técnicas estáticas e dinâmicas, afirmando que inspeções podem ser usadas em todos estágios do processo de software, mas somente pode ser testado um protótipo o uma versão executável do software ou de parte dele (SOMMERVILLE, 2007).

As técnicas estáticas e dinâmicas possuem um papel complementar para V&V. Isto pode ser exemplificado através da comparação entre a avaliação de desempenho e mautenibilidade em um software. O desempenho é avaliado através de técnicas dinâmicas. Já a manutenibilidade é avaliada através de técnicas estáticas (SOMMERVILLE, 2007).

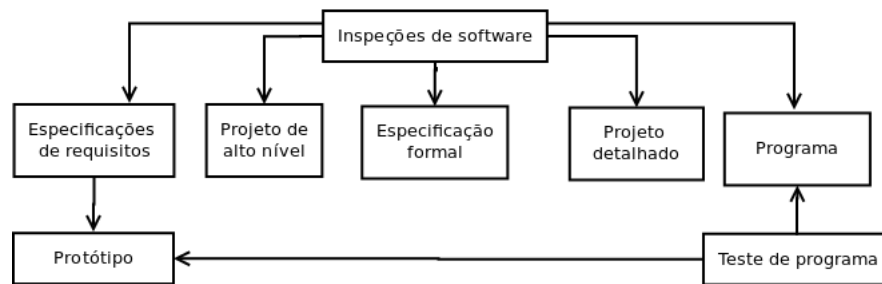


Figura 3.2: Aplicação de V&V estática e dinâmica em diferentes tipos de artefatos
Fonte: SOMMERVILLE (2007)

3.2 Análise estática automatizada

Analísadores estáticos são ferramentas de software que varrem o código fonte ou o código objeto⁴ de um programa sem executá-lo, podendo detectar defeitos ou anomalias (TERRA, 2008), calcular diversas métricas (MILLANI, 2013), avaliar regras de estilo de programação (TERRA; BIGONHA, 2008) e verificar o cumprimento de regras relacionadas ao uso de recursos disponíveis na linguagem de programação.

As análises feitas por este tipo de ferramenta dependem da linguagem de programação utilizada. A linguagem Java, por exemplo, tem características como variáveis fortemente tipadas, que fazem desnecessárias alguns tipos de checagem feitas por um analisador estático para linguagem C.

Algumas ferramentas tem objetivos mais amplos, por exemplo efetuando diversos tipos de análises, extraindo diversas métricas e permitindo a realização de vários tipos de configurações. Outras são mais simples, pois atuam fazendo uma ou poucas análises e são pouco configuráveis. Nos capítulos seguintes deste texto serão abordadas ferramentas com estes dois perfis.

De forma geral, sem levar em conta as particularidades de cada linguagem de programação, algumas aplicações de um analisador estático em relação a defeitos em código fonte são (SOMMERVILLE, 2007):

- Defeitos de dados, como variáveis utilizadas antes da inicialização, variáveis declaradas, mas não usadas, variáveis atribuídas duas vezes, mas nunca usadas entre as atribuições, possíveis violações de limites de vetores, variáveis não declaradas;
- Defeitos de controle, como código inacessível e ramificações incondicionais em loops;
- Defeitos de interface, como um tipo de parâmetro que não combina, número de parâmetros que não combina, resultados de funções não usados e funções ou métodos não usados;
- Defeitos de gerenciamento de ponteiros.

⁴Como por exemplo o bytecode, no caso da linguagem Java

Com base no nome da ferramenta Lint, criada por Stephen Johnson (JOHNSON, 1977), algumas ferramentas que fazem análise estática de código fonte voltada a verificação de defeitos usam o termo “Lint” em parte do nome. A classificação deste tipo de ferramenta como “Linter” também é usada.

A extração de métricas é um outro nicho para a análise estática automatizada de código. Ferramentas deste tipo são também chamadas de Extrator de Métricas. Há diversas opções disponíveis, cada uma tendo seus pontos fortes e fracos, sendo difícil definir claramente uma melhor que todas outras. (MILLANI, 2013)

Métricas relacionadas à complexidade ciclomática e à programação orientada a objetos são exemplos de métricas que podem ser extraídas pelo uso de uma ferramenta de análise estática de código fonte (MILLANI, 2013).

A comparação entre o código fonte de um programa e um conjunto de regras de estilo também pode ser feita através de análise estática de código. Existem de 2 categorias de ferramentas para isto (TERRA; BIGONHA, 2008):

- Verificador de regras de estilo (ou *style checker*, em inglês) somente verifica o código fonte em relação às regras de estilo desejadas, mas não altera o código fonte analisado;
- Um programa formatador de código fonte (ou *beautifier*, em inglês), além de efetuar a comparação do código fonte em relação às regras de estilo, gera um código fonte formatado equivalente ao analisado, sem alteração de semântica.

As regras para codificação podem não abranger somente questões estilísticas relacionadas à formatação do código fonte, podendo ser determinadas regras para uso dos recursos fornecido por uma linguagem de programação. Um exemplo é a proibição do uso de instruções ‘goto’.

Um conjunto de regras para codificação podem ser agrupadas em um padrão de codificação (também chamado de convenção de codificação). Um padrão de codificação, que será abordado com mais detalhes em um capítulo específico deste texto, pode ser estabelecido em diversos contextos, como por exemplo:

- Por uma empresa, para uso interno ou para exigir que seus fornecedores o usem;
- Por um fornecedor de uma ferramenta base para a construção de aplicações, como frameworks MVC;
- Por uma comunidade de desenvolvedores;
- Por mantenedores de ferramentas que permitam contribuições da comunidade através da adição

de trechos de código, como por exemplo alguns softwares CMS⁵.

Apesar da ampla gama de usos de ferramentas para análise estática de código fonte, este tipo de ferramenta não elimina a necessidade da realização de revisões humanas:

- Em relação à análise de defeitos, os que dependem do entendimento da lógica de um programa podem ser encontrados em revisões ou testes, mas não através de análise estática de código fonte. Por outro lado, as ferramentas de análise estática são mais são mais rápidas (ou conseguem ser mais minuciosas em um mesmo intervalo de tempo), resultando em menor custo para uma mesma tarefa (WAGNER et al., 2005);
- Em relação à avaliação de métricas, pode ser interessante a interpretação humana de um conjunto de métricas para tomada de decisão;
- As ferramentas de análise estática automatizada podem relatar também falsos positivos, ou seja, apontar defeitos que não existem. Em alguns casos, estes falsos positivos podem ocorrer em um alto número (WAGNER et al., 2005). Desta forma, pode ser necessária a calibração deste tipo de ferramenta ou mesmo uma contínua interpretação qualitativa dos relatórios gerado pelas ferramentas.

Uma combinação interessante é o uso de ferramentas de análise estática automatizadas antes da realização de revisões, ficando então a cargo das revisões apenas encontrar os problemas remanescentes (WAGNER et al., 2005).

⁵Sistemas de gerenciamento de conteúdo

4 *Manutenção de software*

A manutenção de software pode ser definida como “um processo geral de mudanças no software depois que ele é entregue” e dividida em 3 tipos (SOMMERVILLE, 2007):

- Manutenção para reparos de defeitos: podem ser relacionada a problemas diversos, como erros de codificação, erros de projeto e erros de especificação;
- Manutenção para adaptação a um ambiente operacional diferente: pode ser necessária quando algo relacionado ao ambiente em que está instalado muda, como uma alteração de hardware, de sistema operacional ou de algum software de apoio;
- Manutenção para adição ou alteração de funcionalidades: necessária quando os requisitos mudam, o que pode ocorrer por alterações organizacionais ou de negócios. Normalmente este tipo de manutenção está relacionada a mais mudanças no software do que os outros tipos.

Em uma de suas abordagens sobre manutenção de software, Pressman afirma que a evolução é algo natural no ciclo de vida de um software, independente do tamanho, complexidade ou domínio da aplicação. Parte dos fundamentos usados vêm das leis de Lehman e seus colegas, como a lei da mudança contínua e lei do crescimento contínuo, que pregam a necessidade de evolução do software para evitar que ele torne-se menos satisfatório (PRESSMAN, 2011).

4.1 **Manutenibilidade e suas subcaracterísticas**

A norma NBR ISO/IEC 9126, que aborda a qualidade do produto de software, define a manutenibilidade como a “capacidade do produto de software de ser modificado” e detalha que “as modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais.”. Esta mesma norma classifica a manutenibilidade como uma das 6 características de qualidade de um software (as demais são funcionalidade, confiabilidade, usabilidade, eficiência e portabilidade) e a divide em 4 subcaracterísticas (NBR ISO/IEC, 2003):

- Analisabilidade: “Capacidade do produto de software de permitir o diagnóstico de deficiências ou causas de falhas no software, ou a identificação de partes a serem modificadas”;
- Modificabilidade: “Capacidade do produto de software de permitir que uma modificação especificada seja implementada.”;
- Estabilidade: “Capacidade do produto de software de evitar efeitos inesperados decorrentes de modificações no software.”;
- Testabilidade: “Capacidade do produto de software de permitir que o software, quando modificado, seja validado.”;
- Conformidade relacionada à manutenibilidade: “Capacidade do produto de software de estar de acordo com normas ou convenções relacionadas à manutenibilidade.”.

Analisando as subcaracterísticas da manutenibilidade, é possível constatar que a forma como o software é codificado influi em diversos aspectos dela. A identificação das partes a serem modificadas (analisabilidade), por exemplo, pode ser muito mais alta em um software bem documentado (inclusive o próprio código fonte) e cujo código fonte foi gerado utilizando estratégias diversas para facilitar seu entendimento. Outro exemplo é a modificabilidade, pois a facilidade de executar uma modificação depende também de como está estruturado o código fonte atual do software.

4.2 Métricas e suas relações com a manutenibilidade

As medições são essenciais em qualquer processo de engenharia. Elas podem ser usadas para melhoria de entendimento e avaliação da qualidade dos produtos ou sistemas construídos (PRESSMAN, 2011). Uma medição pode ser definida como “o processo pelo qual números ou símbolos são atribuídos a atributos de entidades no mundo real, de maneira a descrevê-las seguindo regras claramente definidas” (FENTON; PFLEEGER, 1996).

Através da medição é determinada uma medida, que por sua vez proporciona uma indicação quantitativa de extensão, capacidade ou tamanho de algum atributo do processo ou do produto (PRESSMAN, 2011). Esta é relacionada à coleta de um único ponto de dado (como o número de erros descobertos em um componente de software). Um métrica de software relaciona as medidas de alguma maneira, como por exemplo o número médio de erros por teste de unidade.

Uma métrica ou uma combinação delas são denominadas “indicador” quando suas informações sobre o processo, projeto ou produto de software servem como base para análise e realização de melhorias (PRESSMAN, 2011). Podem ser obtidas informações tanto em perspectivas gerenciais quanto técnicas, por exemplo quanto aos custos envolvidos no processo, à produtividade da equipe,

à testabilidade dos requisitos, ao atingimento das metas do processo ou do produto, entre outras (FENTON; PFLEEGER, 1996).

Diversas métricas já foram propostas na área da engenharia de software, porém algumas tem difícil aplicação prática por serem complexas demais de serem medidas, de serem compreendidas ou até por violarem noções intuitivas básicas do que realmente é um software de alta qualidade. Desta forma, as métricas e as medições que levam a elas idealmente devem ser (PRESSMAN, 2011):

- Simples e computáveis, sendo relativamente fácil aprender a derivar a métrica e sua computação não deve demandar esforço ou tempo fora do normal;
- Empiricamente e intuitivamente persuasivas;
- Consistentes e objetivas, não gerando resultados ambíguos e sendo possível ser feita a mesma derivação por terceiros que utilizem as mesmas informações a respeito do software;
- Consistente no uso das unidades e dimensões, com a obtenção da métrica por uma combinação de unidades de medidas que realmente faça sentido;
- Independentes da linguagem de programação;
- Um mecanismo efetivo para feedback de alta qualidade, com a extração informações que sejam úteis para análise e melhoria da qualidade.

Isto não quer dizer que obrigatoriamente uma métrica deva ter todos estes atributos para que seja considerada. Um exemplo é o ponto de função, que pode não ser derivado da mesma forma por um terceiro e, independente disto, não deve ser rejeitado (PRESSMAN, 2011).

As métricas de produto podem ser fornecer informações importantes em diferentes aspectos, como para avaliação do modelo de requisitos e do projeto em relação à arquitetura, suas interfaces e seus componentes, do processo de testes, entre outros. Algumas métricas de produto dependem de como o sistema é projetado, como por exemplo as métricas para sistemas orientados a objetos (PRESSMAN, 2011).

Há métricas que podem ser usadas para avaliar a manutenibilidade de um software. Diversas delas podem ser extraídas diretamente do código fonte através de ferramentas, como por exemplo:

- Complexidade ciclomática - $V(G)$: esta métrica foi introduzida por McCabe com o intuito de identificar módulos de um software difíceis de testar ou de realizar manutenção. Nesta métrica, um grafo de fluxo com somente um nó de entrada e outro de saída, como o exemplificado pela figura 4.1, é traçado como representação de um programa. Cada nó do grafo (círculos na figura 4.1) corresponde a um bloco de código em que o fluxo é sequencial e as arestas (setas

na figura 4.1) correspondem a desvios no fluxo. Pode-se então calcular o número de caminhos independentes $v(G)$ do grafo analisado e utilizar esta medida como um parâmetro de análise (MCCABE, 1976);

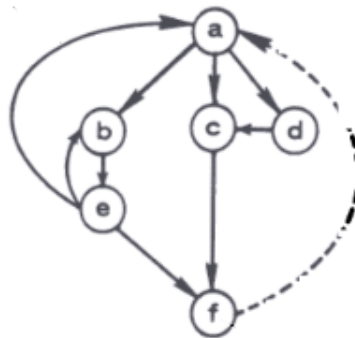


Figura 4.1: Exemplo de grafo de fluxo
Fonte: MCCABE (1976)

- Tamanho da classe - CS^1 ou CSZ^2 : determina o tamanho global de uma classe. Pode ser calculada de diferentes formas, sendo uma delas pela soma entre o número de atributos e o número de métodos encapsulados na classe. Valores baixos desta métrica indicam melhor testabilidade, maior manutenibilidade, maior capacidade de reutilização e que a responsabilidade da classe foi melhor definida (PICHLER, 2013d; PRESSMAN, 2011).
- Métricas de acoplamento: indicam o grau de conexão de um módulo com outros, dados globais e o ambiente externo (PRESSMAN, 2011). Exemplos deste tipo de métrica são o acoplamento aferente (CA), que representa o número de dependências de entrada únicas em um artefato de software (PICHLER, 2013b), o acoplamento eferente (EC), que representa o número de artefatos de software que uma entidade de software depende (PICHLER, 2013c). Um alto valor de acoplamento aferente significa que um artefato tem alta responsabilidade e que alterações nele podem afetar diversas partes do sistema (PICHLER, 2013b). Um artefato com um valor alto de acoplamento eferente pode ser afetado por diferentes fontes de mudança. Então é recomendável manter valores baixos de acoplamento eferente (PICHLER, 2013c). Porém alguns em alguns casos as dependências não são ruins, sendo o papel destas métricas fornecer um indicativo para avaliação (MARTIN, 1994; PICHLER, 2013c).
- Contagem do peso dos métodos - WMC: representa a soma das complexidades de todos os métodos de uma classe, sendo um bom indicador de quando esforço é necessário para desenvolver e manter uma classe. Valores baixos são melhores. A indicação de um limite superior varia na

¹Usado por exemplo por Pressman (PRESSMAN, 2011)

²Usado por exemplo na ferramenta PHP Depend (PICHLER, 2013d)

literatura e é dependente da forma como o cálculo é feito, pois não existe uma única forma de se computar o WMC (PICHLER, 2013h);

- Número de métodos públicos - NPM: é uma métrica simples de contagem, que, como sugere o nome, resulta do número de métodos públicos de uma classe. Um valor alto para esta métrica pode indicar que a classe é complexa e tem responsabilidades em excesso, o que poderia também gerar um alto acoplamento. O limite para um valor ser considerado "alto" pode variar conforme a finalidade da classe. Alta responsabilidade, complexidade e acoplamento podem afetar negativamente a manutenibilidade. A indicação desta métrica pode ser comparada aos valores dados pelas métricas de acoplamento e pela métrica WMC (PICHLER, 2013e).

Existem diversas outras métricas e parte delas podem auxiliar a avaliação da manutenibilidade. Cada ferramenta relacionada a extração de métricas tem suporte a um conjunto específico de métricas.

5 *PHP*

PHP é uma linguagem de scripting de uso geral, rápida e flexível, tendo como foco principal o desenvolvimento web. Apesar disto, permite também a criação de scripts para execução via linha de comando (ou em modo shell) e aplicações Desktop via PHP GTK. A sintaxe da linguagem lembra C, Java e Perl. (THE PHP GROUP, 2013b).

A linguagem de programação web mais utilizada é o PHP:

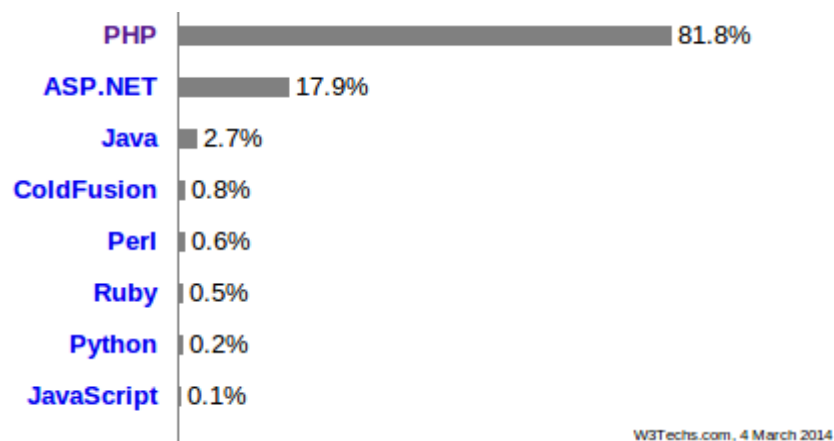


Figura 5.1: Uso de linguagens de programação server-side para sites web

Fonte: W3TECHS (2014)

São exemplos de sites populares que utilizam PHP:

- facebook.com;
- wikipedia.org;
- flickr.com;
- baidu.com;
- wordpress.com.

Diversas ferramentas opensource interessantes são baseadas em PHP, como por exemplo:

- CMS: Drupal, Wordpress, Joomla etc;

- Ferramentas para disparo de e-mail marketing: PHPList, poMMo;
- Plataformas de comércio eletrônico: Magento, OpenCart, osCommerce, Prestashop etc;
- Ferramentas para gerenciamento de fóruns: PHPBB, Vanilla etc;
- Ferramentas para gerenciamento de projetos: PHPCollab, dotProject, PHProjekt etc.

5.1 História

A primeira versão da linguagem foi criada em 1994 por Rasmus Lerdof como um conjunto de scripts CGI escritos em linguagem C, para acompanhamento de visitas em seu currículo on-line. Ele a chamou de "Personal Home Page Tools"(referenciada também como "PHP Tools)". O código fonte do PHP Tools foi liberado para o público em junho de 1995, permitindo que a comunidade aperfeiçoasse e corrigisse bugs no software.

No decorrer da evolução da linguagem até as versões atuais, o PHP chegou a trocar de nome para FI ("Forms Interpreter") e PHP/FI. A versão 3.0, lançada em 1998 já com o nome PHP, foi a primeira versão que realmente se assemelha com o PHP atual. Foi uma rescrita da linguagem com significativa participação de Andi Gutmans e Zeev Suraski. Trazia como pontos fortes a ampliação para sistemas operacionais Windows e Mac OS (a versão anterior era limitada a sistemas operacionais compatíveis ao POSIX), interface para múltiplos bancos de dados e protocolos, suporte (limitado) a orientação a objetos, melhorias de consistência na sintaxe da linguagem e a possibilidade de ampliação da linguagem através da criação de módulos. A versão 4.0 foi lançada oficialmente em maio de 2000 e continha como principal novidade um novo motor, chamado de "Zend Engine". Este motor tinha como principais objetivos o aumento de performance de aplicações complexas e a modularização do código fonte.

A versão 5 foi lançada em julho de 2004, após um longo período de desenvolvimento. Contava com a versão 2.0 da Zend Engine, continha diversas melhorias na orientação a objetos e diversos outros novos recursos. A versão atual estável do PHP é a 5.5.8. Apesar de parecer muito tempo quase 10 anos na versão 5 do PHP, desde a primeira liberação do PHP 5 até a atual, diversas melhorias foram lançadas na sintaxe da linguagem, como suporte a namespaces, traits, lambda functions, type hinting, generators, entre outros. Além disso, foi lançado suporte nativo a JSON, compactação de arquivos ZIP, foram melhorados aspectos de segurança, entre outras novidades. (THE PHP GROUP, 2013a). Outro destaque no período foram o lançamento de diversos projetos relacionados ao PHP, como por exemplo frameworks MVC (CakePHP, Zend Framework, Symfony, Code Igniter, entre outros), bibliotecas ORM e DBAL (como o Doctrine) e outros.

5.2 PHP como linguagem para desenvolvimento web

A forma como o PHP é usado no ambiente web pode ser ilustrado de forma simplificada pela figura abaixo:

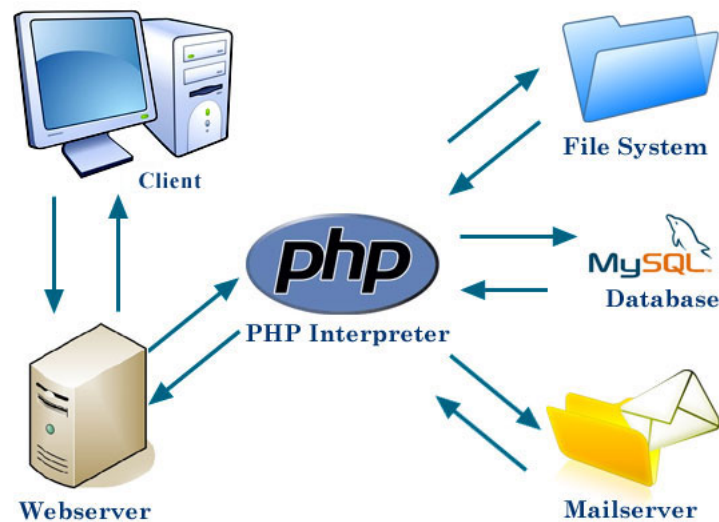


Figura 5.2: Diagrama simplificado de uma requisição web sendo interpretada pelo PHP

Fonte: HONG (2014)

Da figura anterior:

- **Client (cliente):** É quem faz a requisição ao servidor web e renderiza a resposta para o usuário. Normalmente é um navegador de Internet (ou browser), como por exemplo o Google Chrome, o Microsoft Internet Explorer e o Mozilla Firefox;
- **Webservice (servidor web):** Recebe a requisição e devolve a resposta para o cliente. Para recursos estáticos, como páginas HTML sem programação, o servidor web pode responder diretamente ao cliente. Porém, para um conteúdo gerado dinamicamente no servidor, o servidor web aciona uma ferramenta específica para geração da resposta, como por exemplo um interpretador de uma linguagem de programação, como o PHP. Apache, Ngnix, Lighthttpd e IIS são exemplos de servidores web;
- **Interpretador do PHP:** Recebe o encaminhamento da solicitação feita pelo servidor web, interpreta o script e retorna a resposta ao servidor web. Durante o processo de interpretação do script, o PHP pode acionar recursos externos ao PHP, como servidores de banco dados, servidores de e-mail, webservices, sistema de arquivos, comandos do sistema operacional etc;
- **File System (sistema de arquivos), database (servidor de banco de dados) e mailserver (servidor de e-mail):** São alguns exemplos dos diversos recursos externos que podem ser acionados pelo PHP. Tais recursos podem ser fornecidos pelo mesmo host em que é executado o PHP ou também por hosts remotos.

Os navegadores web interpretam linguagens específicas para renderização das páginas fornecidas pelo servidor web, sendo as principais: HTML, Javascript e CSS, não compreendendo as linguagens de programação server side, como o PHP. É função então do interpretador PHP trabalhar nos scripts PHP para gerar a saída nos formatos que o navegadores web compreendem.

Um arquivo PHP pode conter tanto as linguagens compreensíveis pelos navegadores web quanto a linguagem PHP. Para a distinção, os conteúdos PHP são envoltos por tags de abertura (como “<?php”) e fechamento (como “?>”) dos conteúdos PHP¹. A imagem seguinte mostra um exemplo de conteúdo PHP (parte da linha 17 na imagem) mesclado com conteúdos HTML, CSS e Javascript:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5 <title>Exemplo simples de script PHP</title>
6 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
7 <style>
8     body {
9         background-color: #0000FF;
10     }
11 </style>
12 <script type="text/javascript">
13     alert("Olá !");
14 </script>
15 </head>
16 <body>
17     Hoje é dia <?php echo date('d/m/Y');?>
18 </body>
19 </html>
```

Figura 5.3: Exemplo de um script PHP simples

Fonte: O autor da monografia

O script da figura 5.3, após a interpretação PHP, gera o seguinte conteúdo mostrado pela figura 5.4. Veja a alteração na linha 17, em que não há mais conteúdo PHP:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5 <title>Exemplo simples de script PHP</title>
6 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
7 <style>
8     body {
9         background-color: #0000FF;
10     }
11 </style>
12 <script type="text/javascript">
13     alert("Olá !");
14 </script>
15 </head>
16 <body>
17     Hoje é dia 04/03/2014
18 </body>
19 </html>
```

Figura 5.4: Conteúdo HTML gerado com base em um script PHP

Fonte: O autor da monografia

¹Existem outras tags de abertura e fechamento para conteúdo PHP

5.3 PHP como linguagem script de linha de comando

Para o PHP ser executado via linha de comando, ou em modo CLI (Command Line Interface), não é necessário um servidor web ou um navegador, mas deve estar disponível o executável do PHP.

Como neste caso o cliente que recebe a resposta não é um navegador web, não deve ser gerado um conteúdo nos formatos HTML/CSS/Javascript e sim um conteúdo em um formato adequado para a saída desejada, como por exemplo a tela mostrada pelo terminal em que o script PHP CLI foi executado.

O ambiente em que o script PHP CLI será executado pode diferir entre sistemas operacionais, então o PHP provê algumas formas de suprimir estas diferenças, como a constante `PHP_EOL`, que pode ser usada para gerar os caracteres de quebra de linha específicos para o sistema operacional em que o script será executado.

São exemplos de usos interessantes para scripts PHP CLI:

- Execução em horários específicos: os sistemas operacionais normalmente possuem formas de acionar scripts em horários específicos, como o Cron (Linux / Unix) ou o Agendador de Tarefas (Windows). Invocar um script PHP CLI através desta forma consome menos recursos do que acionar um navegador, que por sua vez faria uma requisição a um servidor web e só então o interpretador do PHP seria acionado;
- Portabilidade simples do script entre sistemas operacionais: sistemas operacionais tem shells específicos, como o interpretador de comandos do windows ou os shells Linux/Unix (bash, sh, tcsh, ksh etc). Apesar de existirem formas por exemplo de usar o bash em ambientes Windows, tais formas não são diretas e dependem de instalações adicionais. Tendo o interpretador do PHP instalado, o script PHP CLI pode ser executado em diferentes sistemas operacionais;
- Eliminação da necessidade do aprendizado de linguagens específicas para criação de scripts a serem executados em linha de comando.

Na figura 5.5 é mostrado um exemplo simples de script PHP CLI que gera uma saudação para o nome informado pelo usuário. Na figura 5.6 execução deste script é demonstrada.

5.4 PHP para aplicações desktop: PHP-GTK

Como informa o próprio manual do PHP “o PHP provavelmente não é a melhor linguagem para criação de aplicações desktop com interfaces gráficas, mas se você conhece bem o PHP, e gostaria de usar alguns dos seus recursos avançados nas suas aplicações do lado do cliente, você pode usar o

```
1 <?php
2 if ($_SERVER['argc'] === 2) {
3     echo 'Olá, ' . $_SERVER['argv'][1] . '!';
4 } else {
5     echo 'Você deve passar exatamente 1 argumento ao script.';
6 }
7
8 echo PHP_EOL;
9 ?>
```

Figura 5.5: Exemplo de script PHP CLI simples

Fonte: O autor da monografia

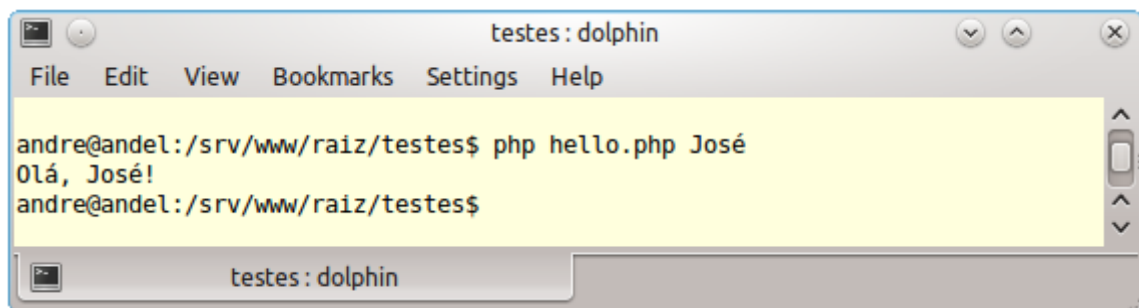


Figura 5.6: Demonstração da execução do script da figura 5.5

Fonte: O autor da monografia

PHP-GTK para escrever programas assim” (THE PHP GROUP, 2014e). As aplicações geradas via PHP-GTK, assim como as demais aplicações PHP, também são multiplataforma.

Da mesma forma que no caso de scripts PHP CLI, as aplicações PHP-GTK não necessitam de um navegador web nem de um servidor web para execução.

Na figura 5.7 são mostradas 2 janelas geradas via PHP-GTK:

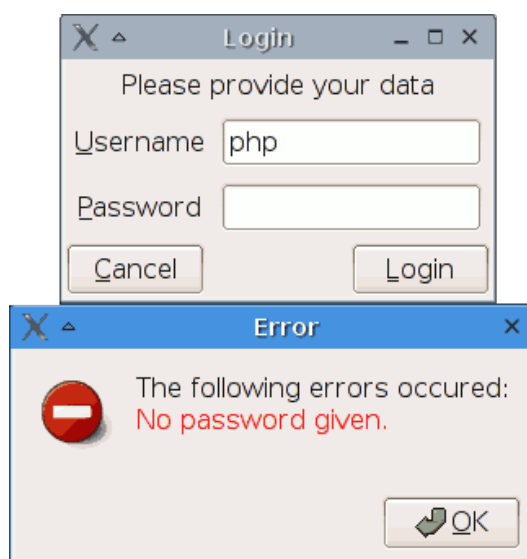


Figura 5.7: Janelas geradas via PHP-GTK

Fonte: THE PHP GROUP (2013c)

6 *Linters*

Neste capítulo serão abordados os linters, ferramentas de análise estática inspiradas no programa Lint, criado por Stephen Johnson para linguagem C.

6.1 História

Na década de 1970, Stephen Johnson adaptou seu compilador pcc para produzir a ferramenta Lint, que era capaz de analisar códigos fonte em linguagem C e indicar trechos propensos a erros, variáveis e funções não usadas, trechos de código que não podem ser alcançados mesmo com todas possíveis variações de comportamentos do programa, trechos de código que, apesar de aceitos pelos compiladores, não seguiam as regras oficiais da linguagem C, entre outras verificações (JOHNSON, 1977; RITCHIE, 1993).

Sommerville, ao abordar análise estática automatizada em seu livro de Engenharia de Software, utiliza o lint como exemplo de ferramenta, demonstrando a execução desta sobre um programa exemplo, como mostra a figura 6.1. E afirma: “Não há dúvidas que, para uma linguagem como C, a análise estática é uma técnica eficiente para descobrir erros de programa” (SOMMERVILLE, 2007).

Desde o surgimento do Lint, alguns programas com características semelhantes surgiram para diversas linguagens e passou-se a usar o termo “linter” ou usar “lint” em parte do nome da ferramenta. Há por exemplo o JSLint¹ para Javascript, criado por Douglas Crockford e o PyLint² para Python, mantido pela empresa Francesa Logilab. Ao fazer uma busca no repositório de pacotes do sistema operacional Ubuntu³, por exemplo, pode-se ver diversos outros exemplos.

6.2 Linters para PHP e integrações com outras ferramentas

Para PHP, o próprio interpretador possui uma funcionalidade classificada pelos desenvolvedores da linguagem como a realização de um processo de “lint” (THE PHP GROUP, 2014b). Porém tal

¹<http://www.jslint.com/lint.html>

²<http://www.pylint.org/>

³<http://packages.ubuntu.com/search?keywords=lint>

```

138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
int Anarray;
{
    printf("%d",Anarray);
}
main ()
{
    int Anarray[5]; int i; char c;
    printarray (Anarray, i, c);
    printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
lint_ex.c(11)
printf returns value which is always ignored

```

Figura 6.1: Análise estática com o lint
Fonte: SOMMERVILLE (2007)

funcionalidade realiza apenas uma verificação de sintaxe. Para acioná-la, é necessário apenas executar o interpretador do PHP com a opção “-l”. A figura 6.3 mostra a análise do arquivo da figura 6.2 pelo comando “php -l”.

```

<?php
Class Teste(){
    public function __construct( {
        echo 'A classe ' . __CLASS__ . ' foi instanciada';
    }
}
?>

```

Figura 6.2: Arquivo exemplo com erros de sintaxe
Fonte: O autor da monografia

```

andre@xub:/srv/www/raiz/testes/lint$ php -l index.php
PHP Parse error:  syntax error, unexpected '(', expecting '{' in index.php on line 2
Errors parsing index.php

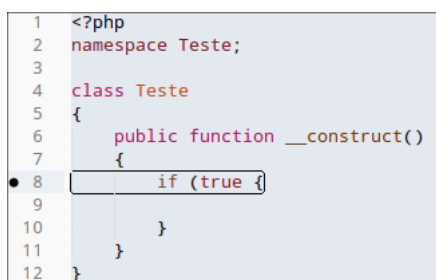
```

Figura 6.3: Análise do arquivo mostrado na figura 6.2 através do comando “php -l”
Fonte: O autor da monografia

Apesar da simplicidade comparada com o Lint para linguagem C, a verificação de sintaxe pode ser interessante em diferentes cenários. Utilizando *hooks pre-commit* de sistemas de versionamento como Git ou SVN, pode-se bloquear o envio de arquivos com erros de sintaxe ao repositório. Isto

porque o *hooks pre-commit* é executado após o servidor de versionamento receber os arquivos e antes dele adicionar os arquivos ao repositório. Um exemplo desta estratégia combinada com o uso de outras ferramentas é mostrada por Piyuesh Kumar (KUMAR, 2014).

Outro interessante uso é a integração com IDEs. O Sublime Text 2⁴, por exemplo possui um plugins que fazem uso do comando "php -l", como o SublimeLinter⁵ ou o sublime-phpcs⁶. A figura 6.4 mostra um exemplo do uso do plugin sublime-phpcs para permitir que o Sublime Text 2 mostre erros de sintaxe no código PHP através do comando "php -l".



```
1 <?php
2 namespace Teste;
3
4 class Teste
5 {
6     public function __construct()
7     {
8         if (true {
9     }
10 }
11 }
12 }
```

Figura 6.4: Verificação de erros de sintaxe através do IDE Sublime Text

Fonte: O autor da monografia

Umberto Salsi desenvolveu uma ferramenta chamada PHPLint, com verificações adicionais, como variáveis inicializadas mas não utilizadas, trechos de código não alcançáveis e outras (SALSI, 2014a). Esta ferramenta está disponível como um verificador on-line de código⁷ e também como um pacote para download, que possibilita aos usuários efetuarem as inspeções pela ferramenta através de um terminal via linha de comando ou através de uma interface gráfica.

A ferramenta é interessante, porém apresenta algumas limitações, como falta de suporte a funções anônimas e o não suporte a algumas versões minor do PHP. Para a não ocorrência de alguns falsos positivos (como mostra a figura 6.5), a ferramenta exige que sejam adicionados blocos de meta código através de comentários de código, que a instruem na interpretação dos arquivos (SALSI, 2014b). Cabe ressaltar que estratégias semelhantes são usadas por outras ferramentas, mesmo não relacionadas a análise estática. Um exemplo é o PHPUnit, uma ferramenta de testes unitários que tem suporte *annotations*, geradas também através de blocos de comentário de código.

Existem outras ferramentas de análise estática que, entre outras coisas, fazem verificações semelhantes às feitas pelo disponibilizando programa Lint para linguagem C, mas que não se auto classificam como "linters". Um exemplo delas é o PHPMD.

⁴O Sublime Text 2 é um sofisticado editor de textos de uso geral, além de programação e edição de linguagens de marcação. Possui excelente performance e interessantes recursos (SKINNER, 2014).

⁵<https://github.com/SublimeLinter/SublimeLinter-for-ST2>

⁶<https://github.com/benmatselby/sublime-phpcs>

⁷<http://www.icosaedro.it/phplint/phplint-on-line.html>

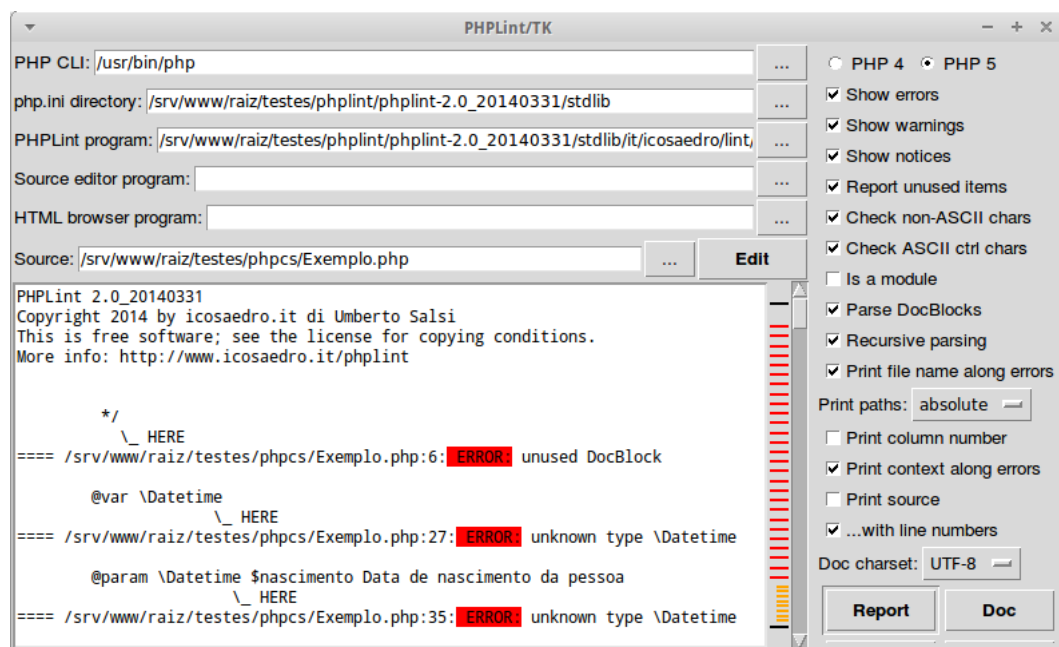


Figura 6.5: Exemplo de execução do programa PHPLint via interface gráfica

Fonte: O autor da monografia

7 *Ferramentas para análise estática em relação a padrões de codificação*

7.1 Padrões de codificação

Um padrão de codificação, também referenciado como convenção de codificação, é uma convenção de regras que governam o uso de uma linguagem de programação (FILHO, 2013). É composto por um ou mais documentos que contém instruções de como o código fonte de um programa deve ser escrito. Aplica-se a uma linguagem de programação específica, devido às diferenças existentes entre cada linguagem de programação.

Pode englobar características estilísticas para aumentar a legibilidade de um código fonte, como por exemplo:

- regras para posicionamento de chaves ("{"}) em situações específicas (declarações de classes ou funções, estruturas de controle, tratamento de exceções etc);
- regras para indentação;
- padrões de nomenclatura (variáveis, constantes, funções, classes etc).

Além disso, pode também conter regras que não estão diretamente relacionadas à legibilidade do código, como por exemplo:

- Regras que determinem boas práticas de programação;
- Especificação de como devem ser usados os recursos da linguagem de programação;
- Tipo de codificação de caracteres dos arquivos.

Alguns dos benefícios são:

- Permite que as revisões foquem em questões que realmente dependam de análise humana, deixando para a ferramenta as demais verificações;

- A manutenibilidade é aumentada. Pressman aponta que uma das características de um software manutenível é ser construído usando padrões e convenções de codificação bem definidos, levando a um código autodocumentado e inteligível (PRESSMAN, 2011);
- Padronização das soluções em situações específicas: muitos problemas de programação podem ser solucionados de formas diferentes. Apesar de nem sempre ser o foco de um padrão de codificação, ele pode orientar o uso de uma estratégia específica, deixando o código fonte mais uniforme, melhorando o entendimento deste por parte do leitor (inspetor, programador que realiza uma manutenção ou consulta etc);
- Redução de custos: Ao facilitar as revisões e as manutenções, estas podem ser realizadas em menos tempo, custando então menos.

7.2 Padrões de codificação para PHP

O PHP recebe críticas em diversos aspectos, mas parte destas críticas relacionam-se a projetos utilizando o PHP e não a linguagem em si. Exemplos são os problemas de segurança em projetos como o PHP Nuke (BACAS, 2011; MITRE CORPORATION, 2014) e código fonte mal projetado e estruturado, como no caso do Wordpress (JOHNSON, 2013).

Apesar disto, algumas críticas são focadas na própria linguagem (MUNROE, 2013; MARTIN, 2010), como por exemplo:

- Falta de padronização na estratégia de nomenclatura em relação ao uso de “underlines” (caracter “_”): algumas funções nativas usam underline na nomenclatura, outras não. Exemplos: `str_replace()`, `strpos()`, `str_pad()`, `strlen()`;
- Falta de padronização na estratégia de nomenclatura em relação ao uso de prefixos: tanto a função `usleep()` quanto a função `microtime()` estão relacionadas a microssegundos. Porém a primeira usa o prefixo “u” para indicar isto e a segunda usa o prefixo “micro” (WASTL, 2014) ;
- Falta de padronização na ordem dos argumentos recebidos por algumas funções. Os dois itens abaixo mostram um exemplo de funções nativas que recebem um número variável de arrays como argumentos e também uma função de callback. A ordem entre a função callback e os arrays a serem passados varia (WASTL, 2014):
 - Callback como último parâmetro: `array_intersect (array $array1 , array $array2 [, array $...], callback $data_compare_func);`
 - Callback como primeiro parâmetro: `array_map (callback $callback , array $arr1 [, array $...])`

- Algumas comparações feitas com o operador '==' levam a resultados estranhos até se entender a forma como o PHP faz a conversão de tipos para comparação (WASTL, 2014). Exemplos:
 - "foo"== true e "foo"== 0 são comparações verdadeiras (avaliadas como o booleano true), mas true == 0 é avaliado como false;
 - "6"== " 6" é avaliado como true;
 - "0x10"== "16" é avaliado como true;
- Há diferentes formas de tratar com erros lançados e, em um mesmo código fonte, pode ser necessário inegragir com mais de uma forma. O PHP suporta exceptions apenas a partir da versão 5. Como mostra o próprio manual do PHP “Funções internas do PHP usam principalmente Error reporting, somente extensões modernas usam exceções.” (THE PHP GROUP, 2007). Isto fez com que os desenvolvedores da Pear, por exemplo, criassem uma forma própria para lançamento e tratamento de erros através da classe PEAR_Error (THE PHP GROUP, 2004). Nas figuras 7.1 e 7.2 são mostrados exemplos de como tratar erros disparados usando a estratégia tradicional do PHP e erros lançados através de exceções.

```
1  <?php
2  // Exemplo de função responsável pelo tratamento de erros no modo tradicional
3  function tratamento_erros ($errno, $errstr, $errfile, $errline, $errcontext) {
4      // Parte do tratamento do erro, como por exemplo um log da ocorrência
5      // O programa poderia ser abortado, se fosse desejado
6  }
7
8  // Configurando a função 'tratamento_erros' como responsável pelo tratamento
9  // de erros
10 set_error_handler('tratamento_erros');
11
12 // Tendando abrir um arquivo inexistente
13 // A função 'tratamento_erros' será acionada e, se ela não abortar a execução,
14 // em seguida o fluxo normal do programa continuará
15 $resource_fopen = fopen('arquivo_inexistente.txt');
16
17 // Tratamento do erro no fluxo do programa, com base no retorno de fopen
18 if ($resource_fopen === false) {
19     // Código a ser executado caso tenha ocorrido um erro
20 } else {
21     // Código a ser executado caso não tenha ocorrido um erro
22 }
23 ?>
```

Figura 7.1: Exemplo de uso do mecanismo tradicional do PHP para tratar erros

Fonte: O autor da monografia

Apesar de um padrão de codificação não poder resolver problemas nativos da linguagem, pode fazer com que, em um projeto específico, tenha-se estratégias robustas para questões como padronização de nomenclaturas (arquivos, diretórios, classes, funções, variáveis etc), uso de recursos nativos da linguagem, melhor estruturação do código fonte da aplicação, entre outros. Estes são alguns dos focos dos padrões de codificação que serão abordados nos tópicos seguintes.

```
1  <?php
2  try {
3      // O caminho passado como parâmetro não existe
4      $client = new DirectoryIterator('caminho_inexistente');
5
6      // Código a ser executado se não for lançada uma exceção ao
7      // instanciar DirectoryIterator
8  } catch (UnexpectedValueException $e) {
9      // Tratamento da exception
10 }
11 ?>
12
```

Figura 7.2: Exemplo de uso de exceções para o tratamento de erros

Fonte: O autor da monografia

7.2.1 Padrão de codificação do Wordpress

O Wordpress é uma derivação do projeto b2 cafelog, que começou a ser desenvolvido em 2001. Em 2003 foi feita a derivação do b2 cafelog por Matt Mullenweg and Mike Little (WORDPRESS, 2013). Neste ano estava disponível apenas a versão 4 do PHP, que continha limitações como o suporte reduzido à orientação a objetos, a indisponibilidade de exceções e lazy loading, entre outras. Atualmente (16/03/2014), a versão 3.8.1 do Wordpress é a estável mais recente disponível, mas já está disponível para testes a versão 3.9.

Segundo o site oficial (WORDPRESS, 2014a), existem em 16/03/2014 76.860.744 sites com Wordpress no mundo. Nesta mesma data, estão cadastrados 2.372 temas e 29.907 plugins no site oficial. Com uma ampla comunidade contribuindo, é difícil que se chegue a um código fonte padronizado sem ter claro um padrão de codificação.

Nas figuras seguintes são mostrados alguns trechos de códigos de plugins do Wordpress. As figuras 7.3 e 7.4 estão relacionadas ao plugin Akismet. Na primeira, a estratégia de indentação usada dificulta a legibilidade, já que o conteúdo do método css não está indentado em relação à declaração do método. Já na segunda figura é mostrado um trecho em que são mescladas declarações de funções, chamadas de funções (algumas condicionais) e atribuições de variáveis. No arquivo original, akismet.php (da versão 2.5.9 do plugin Akismet), a função akismet_cron_recheck() está na linha 509. Então o programador que fará manutenção neste código fonte precisará navegar entre declarações de funções, no meio do arquivo, para achar as execuções de algumas funções e uma chamada condicional à função add_action(). Estes são exemplos de códigos de baixa manutenibilidade.

Já a figura 7.5 contém um trecho de código do plugin FV-Antispam. Pode-se ver uma função PHP que possui em seu conteúdo uma função Javascript. A legibilidade do código poderia ser melhorada tanto pela escolha dos nomes das funções quanto pela estratégia de indentação.

A figura 7.6 contém um trecho de código do plugin BBPress. A legibilidade do código deste

```

<?php
/**
 * @package Akismet
 */
class Akismet_Widget extends WP_Widget {

    function __construct() {
        parent::__construct(
            'akismet_widget',
            __( 'Akismet Widget' ),
            array( 'description' => __( 'Display the number of spam comments Akismet has caught' ) )
        );

        if ( is_active_widget( false, false, $this->id_base ) ) {
            add_action( 'wp_head', array( $this, 'css' ) );
        }
    }

    function css() {
?>

<style type="text/css">
.a-stats {
    width: auto;
}

```

Figura 7.3: Trecho de código do plugin Akismet
Fonte: O autor da monografia

```

function akismet_cron_recheck() {
    // Conteúdo omitido para facilitar a visualização do contexto desejado
}
add_action( 'akismet_schedule_cron_recheck', 'akismet_cron_recheck' );

function akismet_add_comment_nonce( $post_id ) {
    echo '<p style="display: none;">';
    wp_nonce_field( 'akismet_comment_nonce' . $post_id, 'akismet_comment_nonce', FALSE );
    echo '</p>';
}

$akismet_comment_nonce_option = apply_filters( 'akismet_comment_nonce', get_option( 'akismet_comment_nonce' ) );

if ( $akismet_comment_nonce_option == 'true' || $akismet_comment_nonce_option == '' )
    add_action( 'comment_form', 'akismet_add_comment_nonce' );

global $wp_version;
if ( '3.0.5' == $wp_version ) {
    remove_filter( 'comment_text', 'wpkses_data' );
    if ( is_admin() )
        add_filter( 'comment_text', 'wpkses_post' );
}

function akismet_fix_scheduled_recheck() {
    // Conteúdo omitido para facilitar a visualização do contexto desejado
}

```

Figura 7.4: Outro trecho de código do plugin Akismet
Fonte: O autor da monografia

plugin é muito maior:

- Os nomes dos métodos mostram claramente o intuito deles, sem a criação de siglas ou abreviações que não sejam claras. Esta questão é tão importante que Robert C. Martin dedicou 1 capítulo inteiro sobre ela em seu livro Clean Code (MARTIN, 2009);
- A estratégia de indentação permite maior legibilidade, estando no mesmo contexto dos outros exemplos (função ou método PHP que gera conteúdo HTML);

```

function disp_footer_scripts() { // some templates (Elegant Themes Chameleon) mess up the
?>
<script type="text/javascript">
function fvaq() {
    if( document.cookie.indexOf("fvaq_nojs") == -1 ) {
        if( !document.getElementsByClassName ) {
            document.getElementsByClassName=function(search){var d=document,elements,pattern,:
        }
        var fvaa = document.getElementsByClassName('fvaa');
        var fvaq = document.getElementsByClassName('fvaq');
        var fval = document.getElementsByClassName('fval');

        for (var i = 0; i < fvaq.length; ++i) {
            if( fvaa[i] !== undefined && fval[i] !== undefined ) {
                fvaq[i].value = fvaa[i].value;
                fvaq[i].style.display = 'none';
                fval[i].style.display = 'none';
            }
        }
    }
    if( document.getElementById('comment') != null ) {
        document.getElementById('comment').value = '';
    }
}

```

Figura 7.5: Trecho de código do plugin FV-Antispam

Fonte: O autor da monografia

- Existem comentários de código a respeito dos métodos.

```

/**
 * Inserts HTML at the top of the main content area to be compatible with
 * the Twenty Twelve theme.
 *
 * @since bbPress (r3732)
 */
public function before_main_content() {
?>

    <div id="bbp-container">
        <div id="bbp-content" role="main">

<?php
}

/**
 * Inserts HTML at the bottom of the main content area to be compatible with
 * the Twenty Twelve theme.
 *
 * @since bbPress (r3732)
 */
public function after_main_content() {
?>

        </div><!-- #bbp-content -->
    </div><!-- #bbp-container -->

<?php
}

```

Figura 7.6: Trecho de código do plugin BB Press

Fonte: O autor da monografia

Com diferentes estratégias de organização do código fonte, a equipe responsável pela manutenção do Wordpress decidiu adotar um padrão de codificação (WORDPRESS, 2014b). Ele está dividido em 4 partes, sendo mais extensas as partes diretamente relacionadas programação (PHP e Javascript):

- PHP: Contém dicas de boas práticas, instruções para aumentar a legibilidade do código, padrões

de nomenclatura, regras para o uso de recursos específicos da linguagem PHP e instruções de como comentar e documentar o código fonte;

- HTML: Instrui a validação do código HTML via W3C Validator e contém algumas regras simples para validação de tags e indentação;
- CSS: Contém instruções de como comentar o código e estruturar o arquivo CSS, dicas de boas práticas e padrões de nomenclatura;
- Javascript: Contém instruções de como comentar o código, dicas de boas práticas, instruções para aumentar a legibilidade do código, padrões de nomenclatura, indicações de ferramentas para validar o código gerado e regras para o uso de recursos específicos da linguagem Javascript.

Na figura 7.7 é mostrado um trecho do padrão de codificação para Wordpress. Como pode ser visto, há uma preocupação na legibilidade e clareza do documento, estando os trechos de código com numeração das linhas e realce de sintaxe¹.

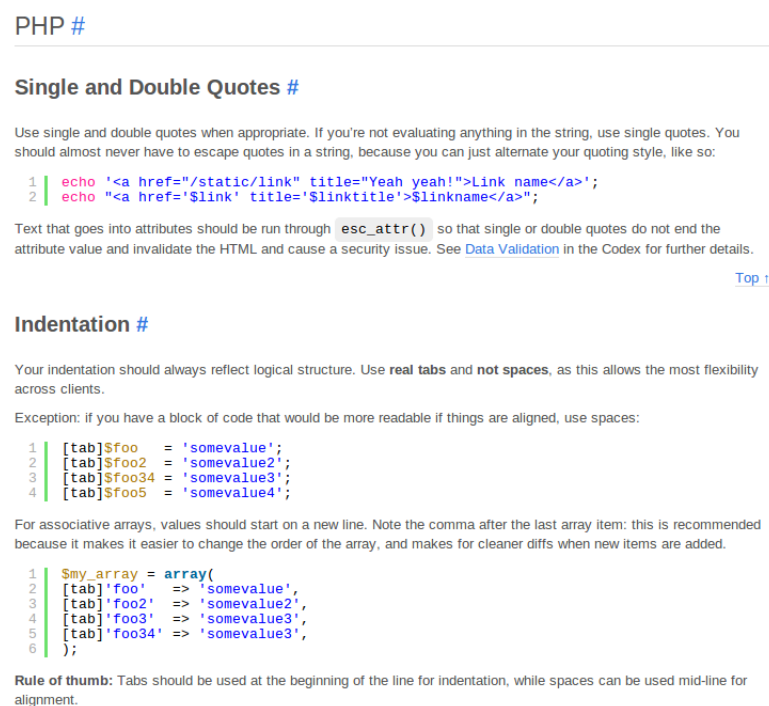


Figura 7.7: Trecho do padrão de codificação do Wordpress
Fonte: WORDPRESS (2014b)

Ao menos parte da comunidade usuária do Wordpress compreendeu a importância de se ter um código fonte atendendo a padrões de qualidade, sendo que o enquadramento no padrão de codificação é um dos pontos necessários. No próprio fórum do Wordpress há críticas a plugins com problemas no código fonte (WARD, 2013; DE GROO, 2012). Há ainda publicações adicionais que, além de

¹Realce de sintaxe, ou “syntax highlight” é a apresentação do código fonte com uma formatação específica para determinados grupos de termos de uma linguagem de programação, normalmente envolvendo coloração do texto.

abordarem o uso do padrão de codificação, mostram boas práticas no desenvolvimento relacionado a Wordpress, como as “Diretrizes para Qualidade com Wordpress”, publicadas por Bjørn Johansen (JOHANSEN, 2014).

7.2.2 Padrão de codificação Pear

A Pear, um acrônimo para "PHP Extension and Application Repository", define-se como "um framework e sistema de distribuição para componentes PHP reutilizáveis"(THE PHP GROUP, 2014g). A Pear fornece repositórios e instaladores, tornando possível instalar facilmente diversos pacotes PHP para áreas como encriptação, internacionalização, sistemas de pagamento, benchmarking etc.

Existem 2 versões principais disponíveis: a Pear 1.x (também chamada simplesmente de Pear) e a Pear2. São 2 repositórios independentes de pacotes. A Pear2, mais recente, trouxe algumas mudanças como obrigatórias para os pacotes, por exemplo o uso de namespaces, fazendo com que a execução dos pacotes só seja possível com o PHP a partir da versão 5.3 . Além disso, outra novidade foi o Pyrus, um instalador com melhorias em relação ao instalador Pear anterior (THE PHP GROUP, 2014j), adaptado ao novo repositório Pear e mantendo suporte ao repositório da versão anterior.

Em 28/01/2014 o número de pacotes disponíveis no repositório Pear é 595 (THE PHP GROUP, 2014h). Com o aumento do número de pacotes no decorrer do tempo, é muito difícil manter uma uniformidade no código fonte dos pacotes sem ter um padrão claramente estabelecido. E a Pear possui padrões, que estão indicados com detalhes em sua documentação (THE PHP GROUP, 2014a). A intenção em se estabelecer padrões é expressa na própria documentação: "Os Padrões de codificação Pear aplicam-se ao código que é parte da distribuição oficial Pear. Padrões de codificação, muitas vezes abreviados como CS entre os desenvolvedores, destinam-se a manter o código consistente para ter alta legibilidade e manutenibilidade para maioria da comunidade Pear".

Existem 2 padrões documentados no manual da Pear, um simplesmente denominado "Padrões de Codificação" ("Coding Standards") e outro denominado "Padrões de Codificação Pear2" ("Pear2 Coding Standards"). Cabe ressaltar que, apesar de ser usada no manual da Pear a terminologia no plural ("Padrões de Codificação" e "Padrões de Codificação Pear2"), o manual está se referindo em cada caso a um codificação que contém diversas instruções de como ser seguido.

Estes dois padrões se relacionam da seguinte forma: existe um padrão inicial, "Coding Standards", aplicável aos pacotes Pear 1.x. No "Pear2 Coding Standards" são listadas apenas as alterações em relação ao "Coding Standards" necessárias para um pacote Pear2.

Os tópicos abordados em "Coding Standards" são:

- **Indentação e comprimento de linha:** Instruem que as indentações devem ser feitas através de 4 espaços e não tabs e que as linhas devem conter no máximo de 75 a 85 caracteres;

- Estruturas de controle: contém regras e exemplos para manter estruturas de controle ("if", "switch", "for", operador ternário etc) legíveis;
- Chamadas de funções: regras para legibilidade em chamadas de funções, levando em conta chamadas sucessivas à mesma função, espaçamento e quebra de linha na passagem de parâmetros etc.
- Definições de classes: na declaração de uma classe, instrui a inserção de uma quebra de linha após o nome da classe e antes da chave de abertura;
- Definições de funções: instrui o uso do estilo K&R² (Kernighan and Ritchie). Além disso, aborda especificamente a declaração de argumentos em 1 ou mais linhas, a ordem entre argumentos obrigatórios e opcionais e instrui o retorno de um valor significativo pela função que estiver sendo declarada;
- Arrays: regras legibilidade na declaração de arrays, levando em conta o alinhamento e o uso de múltiplas linhas;
- Comentários e blocos de comentário de cabeçalho: São duas seções separadas na documentação. Instrui-se a declaração de comentários de código no estilo das linguagens C (`/* */`) e C++ (`//`), mas não no estilo Perl/Shell (`#`). Instrui o uso de dockblocks footnoteBlocos de documentação declarados no formato de comentário de código e mostra exemplos do uso destes em diversos casos (arquivo, classe, propriedade de classe, método de classe etc). São mostradas também as tags PHPDoc³ obrigatórias. Além disso, indica o uso de comentários descritivos quando forem necessários para compreensão do código;
- Inclusão de código: indica o uso das instruções "require_once" para inclusão não condicional e "include_once" para inclusão condicional;
- Tags PHP: A linguagem PHP possui diferentes tags de abertura e fechamento⁴. Destas, a Pear recomenda o uso das tags "`<?php`" e "`?>`" (ambas sem aspas), respectivamente para abertura e fechamento do bloco de código PHP;
- Uso do SVN: regras para uso do svn.php.net. Apesar deste repositório SVN ainda estar disponível, muito do código relacionado a Pear foi migrado para o GitHub (THE PHP GROUP, 2014f, 2014i);
- URLs de exemplo: indicam o uso de URLs de exemplo conforme as reservadas pela RFC 2606⁵;

²Estilo criado por Brian W. Kernighan e Dennis M. Ritchie no livro C Programming Language (ISBN 0131103628)

³PHPDoc é padrão para documentação de códigos PHP

⁴Mais detalhes sobre as possíveis tags de abertura e fechamento de código PHP podem ser obtidos no manual do PHP

⁵Mais detalhes da RFC 2606 podem ser obtidos no link <http://www.faqs.org/rfcs/rfc2606.html>

- *Naming Conventions*: contém regras para nomenclaturas de funções e variáveis globais, classes, propriedades, métodos e constantes. No caso de propriedades e métodos, são mostradas as variações conforme a visibilidade (público, protegido e privado). O uso de regras para nomenclatura de funções globais é interessante para evitar conflitos pelo uso dos mesmos nomes de funções em diferentes arquivos. Porém, a partir do PHP 5.3, o uso de namespaces é mais indicado para resolver este tipo de problema, evitando nomes demasiadamente longos (THE PHP GROUP, 2014d);
- *File Formats*: indica os padrões de codificação de caracteres aceitos (UTF-8 ou ISO-8859-1) para os scripts e como indicá-los nos arquivos. Além disso, instrui o uso da quebra de linha Unix (line feed, LF) e uso de apenas uma quebra de linha após a tag de fechamento PHP;
- *Código compatível com o modo E_STRICT*: o PHP possui diversas configurações de nível de saída de erros (THE PHP GROUP, 2014c). Isto significa que um desenvolvedor tem a possibilidade de simplesmente ocultar mensagens de erro do código, como por exemplo a não inicialização de variáveis. Para evitar isto, a Pear instrui a configuração no modo E_ALL | E_STRICT (a partir do PHP 5.4 isto é equivalente a apenas E_ALL), sendo responsabilidade do desenvolvedor evitar que sejam geradas saídas de erro nesta configuração;
- *Tratamento de erros*: o suporte a exceções foi lançado apenas no PHP 5. Antes desta versão, outras formas tinham que ser usadas, como o retorno de códigos de erros ou objetos de erro (como por exemplo instâncias da classe `PearError`⁶), ou até o uso do sistema de tratamento de erros tradicional do PHP, disparando um erro através da função `trigger_error`. O padrão de codificação Pear instrui o uso de exceções para o tratamento de erros;
- *Melhores práticas*: contém instruções gerais para melhorar a compreensão do código, como o uso da instrução "return" em funções o mais próximo possível do início da função e utilizar uma quebra de linha entre estruturas de controle;
- *Arquivo de exemplo*: contém um arquivo de exemplo do padrão de codificação, com uma classe, constantes, dockblocks etc.
- *Pear toolbox*: Contém uma documentação de ferramentas auxiliares, como o PHP Code Sniffer e o link para o resumo de QA dos pacotes Pear (aborda violações do padrão de codificação, existencia de testes unitários e existência de documentação para cada pacote).

Como pode ser visto nos itens anteriores, o padrão de codificação da Pear aborda questões diversas, como convenções para nomenclatura, regras para legibilidade de código, boas práticas de

⁶Mais detalhes da classe `Pear Error` podem ser obtidas no link http://pear.php.net/manual/pt_BR/core.pear.pear-error.php

programação, regras para documentação de código e regras para uso de recursos específicos do PHP. A figura 7.8 mostra um trecho deste padrão de codificação.

Ternary operators

The same rule as for if clauses also applies for the ternary operator. It may be split onto several lines, keeping the question mark and the colon at the front.

```
<?php

$a = $condition1 && $condition2
  ? $foo : $bar;

$b = $condition3 && $condition4
  ? $foo_man_this_is_too_long_what_should_i_do
  : $bar;

?>
```

Figura 7.8: Trecho do padrão de codificação Pear
Fonte: THE PHP GROUP (2014a)

7.2.3 Padrões de codificação PSR

Os padrões PSR são criados pelo grupo PHP-FIG. O intuito do grupo é encontrar características em comum nos projetos de seus membros e criar padrões relacionados a estas características. Por envolver projetos de grande visibilidade na comunidade PHP, os padrões são divulgados para uso pela comunidade PHP em geral (PHP-FIG, 2013).

O PHP-FIG é formado por representantes de diversos projetos da comunidade PHP, como por exemplo:

- Frameworks MVC/HMVC: CakePHP, Zend Framework 2, Yii, Symfony 2, Solar Framework etc;
- Gerenciadores de dependências: PEAR e Composer;
- CMS: Joomla, Contao Open Source CMS, Drupal, Ez Publish;
- Ferramentas e bibliotecas diversas feitas em PHP: Sculpin, Doctrine, Propel, Stash etc.

Como pode ser visto na lista anterior, há membros de projetos que possuem seu próprio padrão de codificação, como a PEAR e o Zend Framework 2.

Existem atualmente 5 padrões PSR:

- PSR-1 - padrão básico de codificação (“*Basic Coding Standard*”): contém padrões de nomenclatura, codificação de caracteres do arquivo fonte PHP, regras para o uso de tags de abertura de código PHP e uma regra que classifica os arquivos PHP em 2 tipos (com e sem “efeito colateral”);

- PSR-2 - guia de estilo de código: exige como premissa que o código siga o padrão PSR-1. Além disso, determina regras de estilo de código a serem seguidas, como por exemplo indentação, número máximo de caracteres indicado (mas não exigido) por linha, tipo de caracter de quebra de linha a ser usado, regras para escrita de estruturas de controle (“if”, “switch”, “while” etc) em relação à estética do código, regras para nomenclaturas de métodos etc.
- PSR-3 - Logger Interface: O objetivo é permitir que bibliotecas diversas escritas em PHP possam realizar logs de forma simples e padronizada. Para isto, este padrão determinou uma interface a ser implementada pelas classes responsáveis por efetuarem os logs. Tendo tal implementação, bibliotecas diversas podem exigir a injeção de um objeto que implemente esta interface para que a biblioteca possa efetuar os logs. Além desta interface, o padrão criou uma outra interface a ser usada na biblioteca que consome o objeto responsável por fazer o log e também uma classe com constantes que definem os níveis de erro relacionados aos logs. Tais níveis foram baseados na RFC 5424 ⁷;
- PSR-4 - Improved Autoloading: São regras, como no caso da PSR-0, envolvendo o auto carregamento de recursos, porém com diferentes premissas (SKVORC, 2013). Uma diferença, por exemplo, está na interpretação do significado de underscores no *fully-qualified name*⁸ da classe em relação à localização da classe no sistema de arquivos.

A figura 7.9 mostra um trecho do padrão de codificação PSR-2.



Figura 7.9: Trecho do padrão de codificação PSR-2
Fonte: PHP-FIG (2013)

⁷Mais detalhes da RFC 2606 podem ser obtidos no link <http://tools.ietf.org/html/rfc5424>

⁸O *fully-qualified name* é composto pelo *namespace* e pelo nome da classe.

7.2.4 Outros padrões de codificação

Há outros padrões de codificação para PHP, como os definidos por alguns frameworks MVC ou HMVC. O Zend Framework, por exemplo, possui em seu manual um padrão de codificação para a versão 1.x e outro para a versão 2.x⁹. A versão 2 aborda tópicos não abordados na versão 1, como *lambda functions*, *namespaces* e *exceptions*. É um padrão bem detalhado e organizado, contendo tópicos como (ZEND TECHNOLOGIES, 2014):

- Padrões para o arquivo PHP: número máximo de caracteres por linha, caracter de quebra de linha etc;
- Convenções de nomenclatura: estilo de nomenclatura para classes, funções, interfaces, entre outros, relação entre o *namespace* de uma classe e a estrutura de diretórios em que ela está, padrões para criação de *aliases* ao importar namespaces etc;
- Estilo de código: tags PHP permitidas, uso de aspas duplas ou aspas simples, regras para uso de *namespaces*, regras de estilo para criação de *arrays*, regras de estilo para criação de estruturas de controle diversas, regras para criação de documentação via *tags* PHPDoc etc.

A figura 7.10 mostra um trecho do padrão de codificação do Zend Framework 2.

Namespace Aliases

When aliasing within ZF library code, the aliases **SHOULD** typically follow these patterns:

- If aliasing a namespace, use the final segment of the namespace; this can be accomplished by simply omitting the "as" portion of the alias:

```
use Zend\Filter;           // Alias is then "Filter"
use Zend\Form\Element;    // Alias is "Element"
```

- If aliasing a class, either use the class name (no "as" clause), or suffix the class with the subcomponent namespace preceding it:

```
use Zend\Controller\Action\HelperBroker; // aliased as
"HelperBroker"
use Zend\Filter\Int as IntFilter;        // using suffix
```

See also the section on "Import Statements" for additional standards related to aliasing and imports.

Figura 7.10: Trecho do padrão de codificação do Zend Framework 2

Fonte: ZEND TECHNOLOGIES (2014)

O framework CakePHP, também possui seu padrão de codificação e, além disto, possui uma ferramenta oficial para ajuste automático do código fonte ao padrão de codificação (CAKE SOFTWARE FOUNDATION, INC., 2014). Este padrão contém os tópicos:

- Indicação da ferramenta para ajuste automático do código fonte ao padrão de codificação;

⁹Os padrões de codificação para as versões 1 e 2 do Zend Framework podem ser obtidas respectivamente nos links <http://framework.zend.com/manual/1.12/en/coding-standard.html> e <http://framework.zend.com/wiki/display/ZFDEV2/Coding+Standards>

- Regras para melhorar a legibilidade do código fonte, relacionadas a indentação, ao uso de estruturas de controle, ao uso do operador ternário, entre outras;
- Regras para uso de recursos da linguagem PHP, como a forma recomendada de efetuar comparações, o uso de type hinting, as tags de abertura e fechamento de código PHP permitidas etc;
- Regras de como comentar o código fonte, inclusive abordando PHPDoc;
- Convenções de nomenclatura, como nomes de arquivos, prefixos relacionados à visibilidade de métodos e propriedades, uso de *camel back* para métodos, *camel case* para classes, constantes usando letras maiúsculas e palavras separadas por *underscores* etc;
- Padrões para o arquivo PHP: número máximo de caracteres por linha, caracter de quebra de linha etc;
- Estilo de código: tags PHP permitidas, uso de aspas duplas ou aspas simples, regras para uso de namespaces, regras de estilo para criação de arrays, regras de estilo para criação de estruturas de controle diversas, regras para criação de documentação via tags PHPDoc etc.
- Indicação do uso de endereços de exemplo seguindo a RFC 2606.

Method Definition

Example of a method definition:

```
public function someFunction($arg1, $arg2 = '') {  
    if (expr) {  
        statement;  
    }  
    return $var;  
}
```

Parameters with a default value, should be placed last in function definition. Try to make your functions return something, at least `true` or `false`, so it can be determined whether the function call was successful:

```
public function connection($dns, $persistent = false) {  
    if (is_array($dns)) {  
        $dnsInfo = $dns;  
    } else {  
        $dnsInfo = BD::parseDNS($dns);  
    }  
  
    if (!$dnsInfo || !$dnsInfo['phpType']) {  
        return $this->addError();  
    }  
    return true;  
}
```

There are spaces on both side of the equals sign.

Figura 7.11: Trecho do padrão de codificação do Framework CakePHP
Fonte: CAKE SOFTWARE FOUNDATION, INC. (2014)

O Symfony Framework também possui um padrão de codificação. Ele tem base nos padrões PSR-0, PSR-1 e PSR-2 (SENSIO LABS, 2014) e, além disso:

- Tem um exemplo de código para uma visualização em linhas gerais da aplicação do padrão de codificação
- Aborda regras para melhorar a legibilidade do código fonte, como espaçamento entre operadores, uso obrigatório de chaves para estruturas de controle, casos em que se deve usar uma quebra de linha após uma instrução *return* etc;
- Aborda a estruturação do código fonte de classes, instruindo a declaração de propriedades antes de métodos e o agrupamento de métodos conforme a visibilidade, declarando primeiro o públicos, em seguida os protegidos e por último os privados.
- Determina regras para nomenclatura, como a forma a se nomear os *services*, sufixos a serem usados para *interfaces*, *traits* e classes de exceção, prefixos para classes abstratas etc;
- Determina regras para comentário de código fonte, abordando de forma simples algumas anotações PHPDoc.

Além disso, existe uma página abordando convenções usadas no framework que são aconselhadas também para o código de usuários, contendo os tópicos:

- Uso de métodos com nomes específicos, como “*get()*”, “*set()*”, “*register()*”, “*replace()*”, entre outros, e o que se espera da execução destes métodos;
- Representação de métodos *deprecated*¹⁰ através de anotação PHPDoc e disparo de erro com o nível `E_USER_DEPRECATED`, através da função “*trigger_error()*” do PHP.

Um trecho do padrão de codificação do Symfony Framework pode ser visto na figura 7.12.

Estes são apenas alguns exemplos dos diversos padrões de codificação disponíveis para PHP. A escolha de um padrão específico pode levar em conta fatores como um *framework* base utilizado em um projeto, o estilo de codificação já existente em um projeto, o estilo de codificação dos membros de uma equipe que trabalhará em um projeto, entre outros.

A página “*PHP The Right Way*”¹¹, por exemplo, indica que “Idealmente você deveria escrever código PHP que adere a um ou mais destes padrões. Pode ser qualquer combinação das PSR’s, ou um dos padrões de código feitos pela PEAR ou Zend. Isso significa que outros desenvolvedores podem facilmente ler e trabalhar no seu código, e aplicações que implementem os componentes possam ter consistência, mesmo trabalhando com bastante código de terceiros.”.

¹⁰São métodos ainda disponibilizados na API atual, mas que serão retirados em versões futuras do produto, por motivos que podem variar, como por exemplo baixo desempenho.

¹¹A página “*PHP The Right Way*” contém diversas boas práticas para desenvolvimento em PHP e conta com a contribuição de diversos membros da comunidade relacionada a esta linguagem de programação

Naming Conventions

- Use camelCase, not underscores, for variable, function and method names, arguments;
- Use underscores for option names and parameter names;
- Use namespaces for all classes;
- Prefix abstract classes with Abstract. Please note some early Symfony2 classes do not follow this convention and have not been renamed for backward compatibility reasons. However all new abstract classes must follow this naming convention;
- Suffix interfaces with Interface;
- Suffix traits with Trait;
- Suffix exceptions with Exception;
- Use alphanumeric characters and underscores for file names;
- Don't forget to look at the more verbose [Conventions](#) document for more subjective naming considerations.

Figura 7.12: Trecho do padrão de codificação do Symfony Framework
Fonte: SENSIO LABS (2014)

7.3 PHP Code Sniffer

O PHP Code Sniffer é uma ferramenta que detecta violações de um padrão de codificação definido (SHERWOOD, 2012). Foi criado pela SquizLabs e é mantido por esta mesma empresa, com algumas contribuições da comunidade PHP (como pode ser visto no repositório do projeto).

Esta ferramenta pode ser instalada de diversas formas, como:

- Através da ferramenta PEAR;
- Através da ferramenta Composer;
- Alguns sistemas operacionais fornecem o PHP Code Sniffer através de seus gerenciadores de pacotes. Para o Ubuntu 12.04, por exemplo, existe o pacote php-codesniffer.

Tendo por exemplo a ferramenta 'pear' instalada, o PHP Code Sniffer pode ser instalado conforme indicado pela figura 7.13:

```
$ pear install PHP_CodeSniffer|
```

Figura 7.13: Instalação do PHP Code Sniffer
Fonte: O autor da monografia

Para utilização da ferramenta com o padrão de codificação PEAR, é necessário apenas acionar o executável "phpcs" e indicar o arquivo ou diretório a ser verificado. Isto é exemplificado pela figura 7.14, em que é analisado o arquivo Exemplo.php. Como pode ser visto na figura, o PHP Code Sniffer indicou 5 erros em 4 diferentes linhas:

```
andre@xub:/srv/www/raiz/testes/phpcs$ phpcs Exemplo.php

FILE: /srv/www/raiz/testes/phpcs/Exemplo.php
-----
FOUND 5 ERROR(S) AFFECTING 4 LINE(S)
-----
 38 | ERROR | Whitespace found at end of line
 39 | ERROR | Opening brace should be on a new line
 40 | ERROR | Space after opening parenthesis of function call prohibited
 40 | ERROR | No space found after comma in function call
 60 | ERROR | Whitespace found at end of line
-----
```

Figura 7.14: Exemplo de execução do PHP Code Sniffer

Fonte: O autor da monografia

7.3.1 Padrões de codificação disponíveis e criação de novos padrões

O PHP Code Sniffer, em sua versão 1.5.2, vem por padrão com alguns padrões de codificação disponíveis:

- Squiz: Um padrão de codificação criado pela Squiz Labs, a mesma empresa que responsável pelo PHP Code Sniffer;
- Zend: Baseado no padrão de codificação do Zend Framework, porém está desatualizado, como indica o próprio arquivo de configurações de regras deste padrão no PHP Code Sniffer;
- Pear: Baseado no padrão de codificação da PEAR. O padrão de codificação da Pear 2 não vem por padrão com o PHP Code Sniffer;
- PSR-1 e PSR-2: Baseado nos padrões PSR de mesmo nome;
- My Source: Padrão de codificação usado no projeto My Source da Squizlabs;

Os padrões de codificação no PHP Code Sniffer podem utilizar como base regras genéricas definidas por um pacote “Generic”, que está na mesma estrutura de diretórios das regras para os padrões de codificação. Porém o pacote “Generic” não é destinado a ser selecionado como um padrão de codificação por usuários finais do PHP Code Sniffer.

A seleção de um padrão de codificação é feita ao se executar o PHP Code Sniffer com a opção “--standard=padrao” (sem aspas), como por exemplo “--standard=zend”.

Um padrão de codificação no PHP Code Sniffer é composto por um arquivo de configurações de regras, chamado de “ruleset.xml”, por arquivos XML de documentação em um diretório “Docs” e por classes em um diretório “Sniffs”, que fazem as verificações se o código a ser analisado adere a regras que compõem o padrão de codificação.

A figura 7.15 mostra como exemplo o arquivo ruleset.xml do padrão de codificação Pear. É possível notar como algumas regras genéricas são selecionadas e configuradas. Das linhas 21 a 25,

por exemplo, é configurada uma verificação de caracter de fim de linha, sendo o caracter de quebra de linha Unix selecionado como o padrão:

```
1 <?xml version="1.0"?>
2 <ruleset name="PEAR">
3   <description>The PEAR coding standard.</description>
4
5   <!-- Include some additional sniffs from the Generic standard -->
6   <rule ref="Generic.Functions.FunctionCallArgumentSpacing"/>
7   <rule ref="Generic.NamingConventions.UpperCaseConstantName"/>
8   <rule ref="Generic.PHP.LowerCaseConstant"/>
9   <rule ref="Generic.PHP.DisallowShortOpenTag"/>
10  <rule ref="Generic.WhiteSpace.DisallowTabIndent"/>
11
12  <!-- Lines can be 85 chars long, but never show errors -->
13  <rule ref="Generic.Files.LineLength">
14    <properties>
15      <property name="lineLimit" value="85"/>
16      <property name="absoluteLineLimit" value="0"/>
17    </properties>
18  </rule>
19
20  <!-- Use Unix newlines -->
21  <rule ref="Generic.Files.LineEndings">
22    <properties>
23      <property name="eolChar" value="\n"/>
24    </properties>
25  </rule>
26
27  <!-- This message is not required as spaces are allowed for alignment -->
28  <rule ref="Generic.Functions.FunctionCallArgumentSpacing.TooMuchSpaceAfterComma">
29    <severity>0</severity>
30  </rule>
31
32  <!-- Use warnings for inline control structures -->
33  <rule ref="Generic.ControlStructures.InlineControlStructure">
34    <properties>
35      <property name="error" value="false"/>
36    </properties>
37  </rule>
38
39 </ruleset>
```

Figura 7.15: Arquivo ruleset.xml do padrão de codificação PEAR no PHP Code Sniffer

Fonte: O autor da monografia

A figura 7.16 mostra a classe responsável por verificar se não estão sendo usadas tags de comentário de código “estilo linguagem Perl” (através do caracter “#”). Apesar de ser um tipo de comentário de código permitido pelo interpretador do PHP, no padrão de codificação Pear este tipo de comentário de código não é válido. Esta classe foi selecionada como exemplo pela simplicidade e facilidade de compreensão, mas algumas classes responsáveis por realizar outras inspeções são muito mais complexas que esta, chegando a ter mais que o quádruplo do número de linhas.

É possível ajustar o PHP Code Sniffer a um padrão de codificação personalizado. Para isto, pode-se criar os próprios arquivos de inspeção e também utilizar regras de outros padrões de codificação. O padrão PSR-2 no PHP Code Sniffer, por exemplo, utiliza as duas estratégias. Além de ter classes de inspeção próprias, utiliza como base todo padrão PSR-1, algumas regras genéricas e outras que já estavam disponíveis nos padrões Squiz, Zend e Pear. A figura 7.17 mostra o arquivo ruleset.xml do padrão PSR-2.


```
1 <?php
2 class PEAR_Sniffs_Commenting_InlineCommentSniff implements PHP_CodeSniffer_Sniff
3 {
4     /**
5      * Returns an array of tokens this test wants to listen for.
6      */
7     public function register()
8     {
9         return array(T_COMMENT);
10    }
11    /**
12     * Processes this test, when one of its tokens is encountered.
13     */
14    public function process(PHP_CodeSniffer_File $phpcsFile, $stackPtr)
15    {
16        $tokens = $phpcsFile->getTokens();
17
18        if ($tokens[$stackPtr]['content'] === '#') {
19            $error = 'Perl-style comments are not allowed. Use "// Comment."';
20            $error .= ' or "/* comment */" instead.';
21            $phpcsFile->addError($error, $stackPtr, 'WrongStyle');
22        }
23    }
24    }
25 }
```

Figura 7.16: Exemplo de classe de inspeção no PHP Code Sniffer
Fonte: O autor da monografia

```
1 <?xml version="1.0"?>
2 <ruleset name="PSR2">
3     <description>The PSR-2 coding standard.</description>
4     <rule ref="PSR1"/>
5     <rule ref="Generic.Files.LineEndings">
6         <properties>
7             <property name="eolChar" value="\n"/>
8         </properties>
9     </rule>
10    <rule ref="Zend.Files.ClosingTag"/>
11    <rule ref="Generic.Files.LineLength">
12        <properties>
13            <property name="lineLimit" value="120"/>
14            <property name="absoluteLineLimit" value="0"/>
15        </properties>
16    </rule>
17    <rule ref="Squiz.WhiteSpace.SuperfluousWhitespace">
18        <properties>
19            <property name="ignoreBlankLines" value="true"/>
20        </properties>
21    </rule>
22    <!-- algumas linhas foram omitidas para facilitar a visualização -->
23    <rule ref="PEAR.Functions.ValidDefaultValue"/>
24    <!-- algumas linhas foram omitidas para facilitar a visualização -->
25    <rule ref="Squiz.ControlStructures.LowercaseDeclaration"/>
26    <rule ref="Generic.ControlStructures.InlineControlStructure"/>
27 </ruleset>
```

Figura 7.17: Arquivo ruleset.xml do padrão PSR-2 no PHP Code Sniffer
Fonte: O autor da monografia

7.3.2 Refinamento da lista de arquivos e verificações efetuadas

Uma opção interessante do PHP Code Sniffer é escolher as extensões de arquivos a serem analisadas. Isto é feito acrescentando-se a opção “*–extensions*”. Supondo que o desejo de um usuário seja analisar os arquivos terminados em “.php” ou “.phtml”, pode-se completar o comando phpcs com “*–extensions=php,phtml*”. Alguns frameworks utilizam por exemplo extensões específicas para views, como o Zend Framework e o Cake PHP, que utilizam respectivamente as extensões “phtml” e “ctp”. Pode-se então incluir a verificação destes arquivos, já que por padrão estas extensões são ignoradas pelo PHP Code Sniffer, ou até criar uma verificação específica para os arquivos de view e outra para os demais arquivos PHP.

É possível também ignorar uma série de arquivos ou diretórios, o que é muito interessante

por exemplo ao se utilizar bibliotecas de terceiros, que não precisam aderir ao padrão de codificação do projeto. Para isto, pode-se acrescentar a opção *–ignore* ao executar o PHPCS. Para ignorar o diretório “vendor” logo abaixo do diretório atual, por exemplo, utiliza-se a opção “*–ignore=vendor*”.

Pode-se também usar anotações para indicar ao PHP Code Sniffer que este deve ignorar um trecho de um arquivo ou até mesmo o arquivo como um todo, como mostram as figuras 7.18 e 7.19.

```
<?php
// @codingStandardsIgnoreFile
$xmlPackage = new XMLPackage;
$xmlPackage['error_code'] = get_default_error_code_value();
$xmlPackage->send();
?>
```

Figura 7.18: Solicitando ao PHP Code Sniffer que ignore o arquivo todo
Fonte: SHERWOOD (2012)

```
$xmlPackage = new XMLPackage;
// @codingStandardsIgnoreStart
$xmlPackage['error_code'] = get_default_error_code_value();
// @codingStandardsIgnoreEnd
$xmlPackage->send();
```

Figura 7.19: Solicitando ao PHP Code Sniffer que ignore parte do arquivo
Fonte: SHERWOOD (2012)

7.3.3 Integração com outras ferramentas

É possível integrar o PHP Code Sniffer a outras ferramentas, como sistemas gerenciadores de versões. A própria documentação do PHP Code Sniffer (SHERWOOD, 2012) mostra como podem ser feitas 2 tipos de integração com o gerenciador de versão Subversion. A primeira delas é o negação do envio de código fora do padrão de codificação para o repositório do projeto. Isto pode ser feito por *hooks pre-commit*. Como mostra a figura 7.20, o “*commit*” foi negado pelo fato do arquivo estar fora do padrão de codificação, sem o trecho de comentário que indica qual a finalidade do arquivo.

Pode-se também gerar *hooks pre-commit* mais complexas utilizando as diversas opções do PHP Code Sniffer, como por exemplo envolvendo duas execuções do PHP Code Sniffer para um mesmo “*commit*”, verificando diferentes conjuntos de arquivos (baseando-se nas extensões) com regras específicas.

O segundo tipo de integração com sistemas de gerenciamento de versões mostrado na documentação do PHP Code Sniffer é também com o Subversion, mais especificamente com o comando “*svn blame*”. Neste caso, em vez de negar o “*commit*”, é feito um relatório do número de violações do

```
$ svn commit -m "Test" temp.php
Sending          temp.php
Transmitting file data .svn: Commit failed (details follow):
svn: 'pre-commit' hook failed with error output:

FILE: temp.php
-----
FOUND 1 ERROR(S) AND 0 WARNING(S) AFFECTING 1 LINE(S)
-----
 2 | ERROR | Missing file doc comment
-----
```

Figura 7.20: Acionando o PHP Code Sniffer através de uma hook pre-commit do Subversion
Fonte: SHERWOOD (2012)

padrão de codificação por usuário do Subversion, como mostra a figura 7.21. Nota-se também o uso da opção “-s”, que adicionalmente mostra os tipos de violação feitas por cada autor. A integração com o gerenciador de versões Git, que possui o comando “git-blame”, funciona de forma semelhante.

```
$ phpcs -s --report=svnblame /path/to/code

PHP CODE SNIFFER SVN BLAME SUMMARY
-----
AUTHOR  SOURCE                                                    COUNT (%)
-----
jsmith                                     51 (40.8)
      Squiz.Files.LineLength                             47
      PEAR.Functions.FunctionCallSignature                4
jblogs                                     44 (30)
      Squiz.Files.LineLength                             40
      Generic.CodeAnalysis.UnusedFunctionParameter        2
      Squiz.CodeAnalysis.EmptyStatement                   1
      Squiz.Formatting.MultipleStatementAlignment         1
-----
A TOTAL OF 95 SNIFF VIOLATION(S) WERE COMMITTED BY 2 AUTHOR(S)
-----
```

Figura 7.21: Integração do PHP Code Sniffer com o Subversion pelo comando “svn-blame”
Fonte: SHERWOOD (2012)

Cabe ressaltar que as integrações via hook pre-commit e via “svn blame” não são excludentes. Um uso interessante de ambas no mesmo repositório seria quando o padrão de codificação fosse formalizado após o início do projeto, podendo negar novos envios de arquivos fora do padrão de codificação e tendo um relatório das atuais violações do padrão de codificação. Porém, neste caso, a não ser que sejam usadas anotações para ignorar trechos de código antigos (como mostrado na

seção anterior), o responsável pela atualização de um arquivo no repositório terá que corrigir todas as violações do padrão de codificação no arquivo que está atualizando, mesmo as que não sejam relacionadas a atualização que esteja fazendo.

São possíveis também outros tipos de integrações, como com "IDEs" (utilizando inclusive o mesmo IDE e plugin abordados no exemplo de integração com o *linter* nativo do PHP), ferramentas de exibição de relatórios, servidores de integração contínua e ferramentas de correção automática de código (como o PHP Code Standards Fixer, que será abordado em seguida).

7.4 PHP Code Standards Fixer

O PHP Code Standards Fixer, desenvolvido por Fabien Potencier, é uma ferramenta que corrige a maioria das infrações de um código fonte PHP em relação aos padrões PSR-1 e PSR-2. O autor da ferramenta notou que a tarefa de corrigir os problemas apontados pelo PHP Code Sniffer poderia ser uma tarefa desagradável para alguns programadores e então desenvolveu esta ferramenta para realizar automaticamente este trabalho (POTENCIER, 2014).

O representante do Zend Framework 2 no PHP-Fig¹², Pádraic Brady, é um exemplo de desenvolvedor que incomoda-se com a análise minuciosa feita pelo PHP Code Sniffer: "O problema com os padrões de codificação não é a noção de seguir convenções para garantir que todo programador pode ler e entender o código (e outras coisas boas) rapidamente, mas que alguém criou uma ferramenta para verificar o cumprimento: PHP Code Sniffer. Isto não é uma reclamação sobre o funcionamento da ferramenta, mas sim sobre o simples fato de sua existência. Ferramentas automatizadas pegam tudo. Cada pequeno desvio, cada julgamento sobre estar elegante, limpo ou bonito, e cada caractere de espaço fundamental encontrado fora do lugar é registrado em uma parede de texto em seu console, exigindo que todos esses lapsos cruciais sejam consertados com urgência." (BRADY, 2014).

A utilização do PHP Code Standards Fixer é simples. A página do projeto¹³ disponibiliza um link para download direto da ferramenta. Após efetuar o download, para realizar uma correção do código fonte padrão da ferramenta, é necessário apenas executar o comando `"php-cs-fixer.phar fix <arquivo ou diretório>"`. Um exemplo de resultado gerado pela ferramenta é mostrado pela figura 7.22, que contém uma comparação entre o arquivo original (à esquerda) e o arquivo após a correção (à direita):

A figura 7.22 destaca em azul as linhas alteradas pelo PHP Code Standards Fixer. Foram realizadas as seguintes correções:

- A short tag de abertura do código PHP foi transformada em uma standard tag;
- Foram ajustados os posicionamentos de chaves de aberturas da classe, de métodos e do bloco

¹²Como pode ser visto na própria página inicial do grupo: <http://www.php-fig.org/>

¹³<http://cs.sensiolabs.org/>

Para que não seja necessário especificar sempre os parâmetros passados para a ferramenta, pode-se criar um arquivo com a extensão `”php_cs“`, que a ferramenta tentará interpretá-lo para extrair estes parâmetros.

O PHP Code Standards Fixer pode ser integrado a outras ferramentas, como gerenciadores de versão, IDEs (utilizando, como no caso do PHP Code Sniffer, o mesmo IDE e plugin abordados no exemplo de integração com o linter nativo do PHP), entre outras.

8 *PHP Depend*

O PHP Depend é uma ferramenta que realiza análise estática em código fonte PHP para extração de métricas. É executada via linha de comando e fornece relatórios em formato gráfico e XML.

8.1 Instalação e execução

A instalação da ferramenta pode ser feita das seguintes formas:

- Através da ferramenta PEAR;
- Através da ferramenta Composer;
- Fazendo do download do arquivo *Phar*.

A figura 8.1 mostra os comandos necessários em um terminal Linux para instalação do PHP Depend através da ferramenta PEAR.

```
pear channel-discover pear.pdepend.org  
pear install pdepend/PHP_Depend
```

Figura 8.1: Instalação do PHP Depend via ferramenta Pear em um terminal Linux
Fonte: O autor da monografia

Após a instalação, a execução é feita em um terminal invocando o comando “*pdepend*”. Deve-se atentar à ligeira diferença entre o nome da ferramenta e de seu comando executável.

8.2 Relatório de Abstração x Instabilidade

O gráfico de Abstração x Instabilidade é extraído do PHP Depend através do comando mostrado pela figura 8.2, sendo “<imagem-destino>” o nome do arquivo de imagem a ser salvo e “<alvo-analise>” o arquivo ou diretório a ser analisado.

Este interessante relatório é exemplificado pela figura 8.3. Ele se origina de um estudo de Robert Martin a respeito de dependências em um projeto orientado a objetos, em que foi abordado como

```
$ pdepend --jdepend-chart=<imagem-destino> <alvo-analise>
```

Figura 8.2: Geração do gráfico Abstração x Instabilidade através do PHP Depend

Fonte: O autor da monografia

evitar um projeto rígido, frágil e de difícil reutilização. Um projeto com estas características torna difícil a previsão do impacto de mudanças, resultando em uma baixa manutenibilidade (MARTIN, 1994).

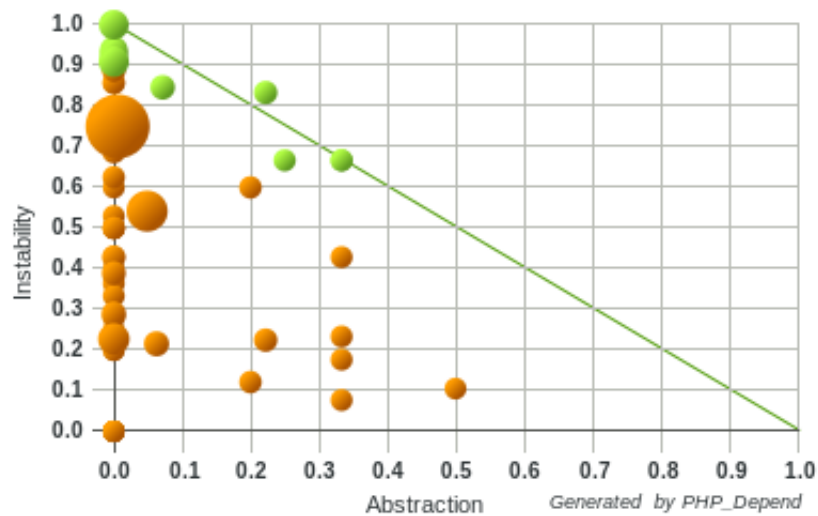


Figura 8.3: Exemplo de gráfico instabilidade x abstração gerado pela ferramenta PHP Depend

Fonte: O autor da monografia

Cada círculo no gráfico corresponde a um diferente pacote no código fonte analisado. No eixo Y da figura 8.3 está a instabilidade (I), que é calculada com base nos acoplamentos aferente (CA) e eferente (CE) através da fórmula (PICHLER, 2013a):

$$I = CE / (CA + CE)$$

No eixo X desta mesma figura está a abstração, que informa a taxa de classes abstratas¹ em relação ao total de classes. Sendo “ac” o total de classes abstratas e “cc” o total de classes não abstratas, calcula-se a abstração “A” pela fórmula (PICHLER, 2013a):

$$A = ac / (ac + cc)$$

O intervalo de valores para instabilidade e para abstração é de 0 a 1. Os extremos deste intervalo podem ser interpretados da seguinte forma (PICHLER, 2013a):

- Um valor 0 para instabilidade significa que o pacote é estável, não tendo dependências de outros pacotes;

¹Interfaces também são contadas como classes abstratas pelo programa

- Um valor 1 para instabilidade significa que o pacote é instável, tendo alta dependência de outros pacotes e não tendo outros pacotes dependentes de si;
- Um valor 0 para abstração significa que não há classes abstratas ou interfaces no pacote;
- Um valor 1 para abstração significa que o pacote contém apenas classes abstratas e interfaces

Os pontos ideais no gráfico são [1,0] e [0,1]. No primeiro caso, isto significaria total estabilidade e abstração. Já no segundo caso significaria total instabilidade e nenhuma abstração. Robert Martin explica que isto seria o ideal porque resultaria de alta dependência em relação a abstrações estáveis, sendo esta uma dependência boa porque adere ao princípio Aberto/Fechado (*Open/Closed* em inglês). Este princípio prega que “entidades de software devem ser abertas para extensão, mas fechadas para modificação” (MARTIN, 1997).

Porém como é difícil atingir estes valores, Robert Martin traçou uma reta entre estes pontos e a chamou de “Sequencia Principal”. Valores próximos a esta reta indicam um equilíbrio adequado entre instabilidade e abstração (MARTIN, 1994).

8.3 Relatório de visão geral em forma de pirâmide

O relatório de visão geral em forma de pirâmide é extraído do PHP Depend de forma semelhante ao relatório de Abstração x Instabilidade, apenas trocando a opção “*-jdepend-chart*” pela opção “*-overview-pyramid*”. Este relatório exibe um conjunto de métricas sobre herança, acoplamento, tamanho e complexidade. A localização de cada grupo de métricas no relatório é mostrada pela figura 8.4 (PICHLER, 2013f).

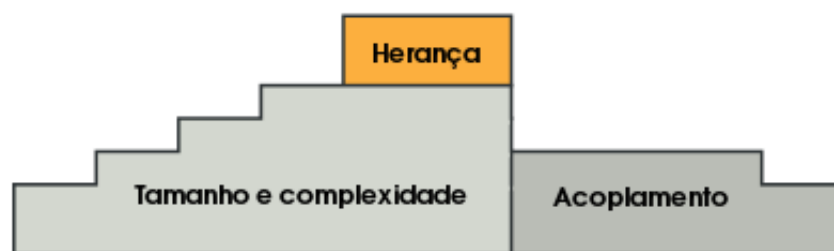


Figura 8.4: Agrupamento das métricas no relatório em forma de pirâmide
Fonte: PICHLER (2013f)

Na região do gráfico a respeito de herança, são exibidas as seguintes métricas (PICHLER, 2013f):

- ANDC: mostra a taxa de subclasses diretas de uma classe;
- AHH: exibe o tamanho médio das árvores de herança. O tamanho de uma única árvore de herança (DIT) é comprimento máximo desde o nó até a raiz da árvore desta árvore de (PRESSMAN, 2011).

Na região do gráfico a respeito de tamanho e complexidade, são mostradas as seguintes métricas (PICHLER, 2013f):

- NOP: número de pacotes;
- NOC: número de classes;
- NOM: soma do número de métodos das classes com o número de funções não encapsuladas em classes;
- LOC: número de linhas de código. A ferramenta não considera na contagem as linhas em branco e as linhas com somente comentários de código;
- CYCLO: Complexidade ciclomática de McCabe.

Na região do gráfico a respeito de acoplamento, são mostradas as seguintes métricas (PICHLER, 2013f):

- CALLS: número de chamadas distintas a funções ou métodos;
- FANOUT: informa sobre tipos referenciados por classes e interfaces, contando apenas os tipos que não são parte do mesmo ramo da árvore de herança.

A figura 8.5 mostra como as métricas são dispostas no relatório. O valor extratido para cada métrica está no centro da pirâmide. Ou seja, na figura 8.5 o valor da métrica ANDC é 0,524. Na laterais da pirâmide são mostradas métricas derivadas das métricas iniciais. O resultado da divisão do número de classes pelo número de pacotes nesta mesma figura é 16,406.

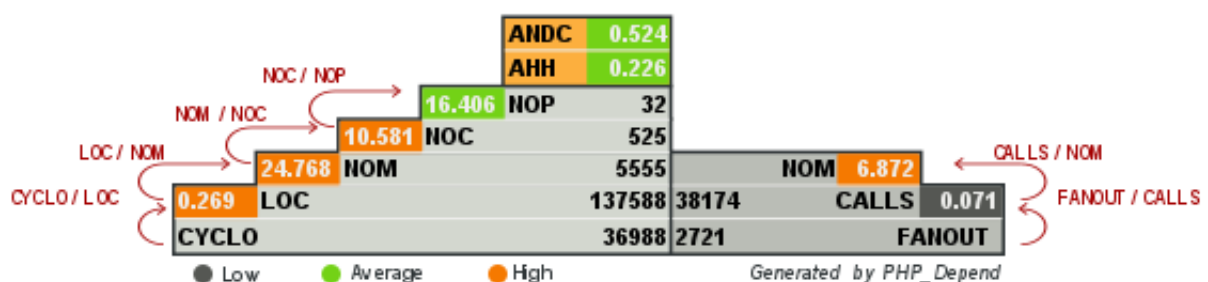


Figura 8.5: Relatório em forma de pirâmide com ilustrações dos cálculos executados

Fonte: PICHLER (2013f)

A ferramenta também apresenta através de cores uma análise dos valores encontrados, como mostra a legenda do gráfico. Na figura 8.5, por exemplo, é mostrado que a relação CYCLO/LOC está com um valor alto.

8.4 Relatório geral de métricas em formato XML

O relatório do PHP Depend que informa o maior número de métricas é exportado no formato XML, sendo geradas 40 métricas. Ele também é extraído do PHP Depend de forma semelhante ao relatório de Abstração x Instabilidade, mas neste caso deve-se trocar a opção “*-jdepend-chart*” pela opção “*-summary-xml*”. A figura 8.6 mostra um pequeno trecho deste relatório.

```
<?xml version="1.0" encoding="UTF-8"?>
<metrics generated="2014-06-02T19:21:15" pdepend="1.1.4" ahh="2.4" andc=
"0.2" calls="13" ccn="25" ccn2="25" cloc="328" clsa="0" clsc="10" eloc="235"
fanout="0" leafs="3" lloc="80" loc="625" maxDIT="3" ncloc="297" noc="10"
nof="1" noi="0" nom="15" nop="7" roots="5">
  <files>
    <file name="/projeto/public_html/Config/Schema/i18n.php" cloc="28" eloc=
"18" lloc="5" loc="53" ncloc="25"/>
    <file name="/projeto/public_html/Config/Schema/db_acl.php" cloc="26"
eloc="39" lloc="7" loc="74" ncloc="48"/>
    <file name="/projeto/public_html/Config/Schema/sessions.php" cloc="26"
eloc="15" lloc="5" loc="48" ncloc="22"/>
    <file name="/projeto/public_html/Console/Command/AppShell.php" cloc="25"
eloc="4" lloc="1" loc="33" ncloc="8"/>
    <file name="/projeto/public_html/Controller/AppController.php" cloc="35"
eloc="6" lloc="2" loc="44" ncloc="9"/>
    <file name="/projeto/public_html/Controller/PagesController.php" cloc=
"58" eloc="51" lloc="31" loc="123" ncloc="65"/>
  </files>
</metrics>
```

Figura 8.6: Exemplo de trecho do XML geral de métricas gerado pelo PHP Depend
Fonte: O autor da monografia

Na análise é feita uma contabilização geral de métricas referente ao local apontado pelo usuário e, além disso, são analisados separadamente pacotes, arquivos, classes, métodos e funções.

A leitura destes arquivos XML por humanos é considerada difícil pelo próprio autor da ferramenta, que classifica o formato como “Legível para máquinas” (PICHLER, 2013g). Por outro lado, este é um excelente formato para ser lido por um script ou uma ferramenta intermediária que extraia e trabalhe Anos dados do XML, podendo ser gerados gráficos, planilhas, alertas de diversos tipos, bloqueio do *commit* em um repositório pela extrapolação de limites pré estabelecidos, entre outros.

9 *Outras ferramentas*

Há diversos outros tipos de ferramentas que efetuam análise estática em código PHP, como as citadas nas seções seguintes.

9.1 PHPMD

O PHPMD, ou "*PHP Mess Detector*", é o equivalente para PHP à ferramenta PMD da linguagem Java. É uma ferramenta gratuita que analisa o código fonte PHP em busca de potenciais problemas, como (PICHLER, 2009):

- Possíveis bugs;
- Código passível de otimização:
- Nomenclatura indevida ou não indicada: constantes que não estão em letras maiúsculas, métodos ou variáveis com nomes muito curtos, métodos que retornam um valor booleano com prefixo indevido etc;
- Expressões complicadas em demazia: código com alta complexidade ciclomática, métodos com conteúdo excessivo, classes com conteúdo excessivo etc.
- Parâmetros, métodos e propriedades não usados.

9.2 PHPCPD

O phpcpd ou "*PHP Copy Paste Detector*" é um programa gratuito que analisa códigos fonte PHP em busca de trechos de códigos repetidos distribuídos em um programa. Foi criado por Sebastian Bergman, o mesmo autor do PHPUnit, uma famosa ferramenta de testes unitários para PHP (BERGMAN, 2014a; BORATE, 2009).

9.3 PHPDCD

O PHPDCD, ou *PHP Dead Code Detector*, é uma outra ferramenta gratuita de análise estática feita por Sebastian Bergman. Esta ferramenta analisa um código fonte PHP em busca de funções e métodos não usados. A ferramenta possui algumas limitações devido às características dinâmicas da linguagem PHP, como a *Reflection API*, chamadas automáticas a métodos mágicos (como `__toString()`), o uso de conteúdo de variáveis como referências a nomes de classes, entre outras (BERGMAN, 2014b).

10 Conclusão

As ferramentas para análise estática de código fonte PHP cobrem diversas atividades, como detecção de bugs, o não enquadramento a padrões, geração de métricas, detecção de práticas não aconselhadas de desenvolvimento, entre outras. Muitas destas ferramentas são gratuitas, inclusive para uso comercial.

O uso de tais ferramentas não substitui por completo as revisões realizadas por seres humanos, nem os testes do software. Mas estas ferramentas podem auxiliar as atividades de garantia de qualidade, executando de forma ágil e padronizada as tarefas a que se propõem.

O tempo necessário para instalação, entendimento, configuração, parametrização e interpretação dos resultados gerados varia conforme as opções dadas por cada ferramenta e as necessidades de seus usuários. Em geral, este tempo é pequeno, pois as ferramentas são bem documentadas, de fácil uso e os relatórios gerados são facilmente compreendidos.

Algumas das ferramentas estudadas permitem que os resultados gerados sejam interpretados por outras ferramentas e ações automáticas sejam disparadas, o que demanda uma configuração adicional, que também não é complexa.

Apesar do baixo tempo necessário para disponibilizar estas ferramentas (levando em conta apenas os aspectos técnicos), deve-se ter em mente a necessidade de se compreender corretamente o intuito de cada ferramenta e as informações geradas. Não há sentido por exemplo em se utilizar o PHP Depend sem compreender as métricas geradas por ele.

Tendo em vista os benefícios gerados, como o aumento da manutenibilidade e a redução de bugs, o uso das ferramentas de análise estática estudadas neste texto tendem a contribuir para o aumento na qualidade do software a ser desenvolvido.

Referências Bibliográficas

ABRAN, A. et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. [S.l.]: IEEE Computer Society, 2004.

BACAS, P. *Malicious Iframe infects PHP-Nuke site....again!* 2011. Disponível em: <http://nakedsecurity.sophos.com/2011/01/17/malicious-iframe-infects-php-nuke-site-again/>. Acesso em: 10 março 2014.

BERGMAN, S. *PHP Copy/Paste Detector (PHPCPD)*. 2014. Disponível em: <https://github.com/sebastianbergmann/phpcpd>. Acesso em: 22 março 2014.

BERGMAN, S. *PHP Dead Code Detector (PHPDCD)*. 2014. Disponível em: <https://github.com/sebastianbergmann/phpdcd>. Acesso em: 22 março 2014.

BOEHM, B. W. Verifying and validating software requirements and design specifications. *IEEE Software*, v. 1, p. 75–88, 1984.

BORATE, S. *Detecting duplicate code in PHP files*. 2009. Disponível em: <http://www.codediesel.com/tools/detecting-duplicate-code-in-php-files/>. Acesso em: 22 março 2014.

BOURQUE, P.; FAIRLEY, R. E. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. 3. ed. [S.l.]: IEEE Computer Society Press, 2014.

BRADY, P. *Coding Standards: Humans Are Not Computers*. 2014. Disponível em: <http://blog.astrumfutura.com/2014/02/coding-standards-humans-are-not-computers/>. Acesso em: 17 abril 2014.

CAKE SOFTWARE FOUNDATION, INC. *Coding Standards*. 2014. Disponível em: <http://book.cakephp.org/2.0/en/contributing/cakephp-coding-conventions.html>. Acesso em: 14 abril 2014.

DE GROO, A. *WP-Reservation booking system - The code quality is poor*. 2012. Disponível em: <http://wordpress.org/support/topic/plugin-wp-reservation-the-code-quality-is-poor>. Acesso em: 12 fevereiro 2014.

FENTON, N. E.; PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. 2. ed. Boston, MA, EUA: International Thomson Computer Press, 1996.

FILHO, F. dos S. *Padrões para Codificação (Coding Standards)*. 2013. Disponível em: http://www.angelfire.com/nt2/softwarequality/padrao_codificacao.pdf. Acesso em: 05 dezembro 2013.

ISTQB. *Standard glossary of terms used in Software Testing*. 2014. Disponível em: <http://www.istqb.org/downloads/finish/20/137.html>. Acesso em: 21 mai 2014.

JOHANSEN, B. *WordPress Quality Guidelines*. 2014. Disponível em: <https://bjornjohansen.no/wordpress-quality-guidelines>. Acesso em: 13 abril 2014.

- JOHNSON, M. *Wordpress: a fractal of bad design*. 2013. Disponível em: <<http://milesj.me/blog/read/wordpress-is-bad-mmmk>>. Acesso em: 10 março 2014.
- JOHNSON, S. C. Lint, a C program checker. *Computing Science TR*, Bell Labs, v. 65, 1977.
- KUMAR, P. *Gatekeeping your code using git hooks*. 2014. Disponível em: <<http://www.qed42.com/blog/gatekeeping-your-code-using-git-hooks>>. Acesso em: 25 abril 2014.
- MARTIN, E. *What I don't like about PHP*. 2010. Disponível em: <<http://www.bitstorm.org/edwin/en/php/>>. Acesso em: 13 março 2014.
- MARTIN, R. *OO Design Quality Metrics - An Analysis of Dependencies*. 1994. Disponível em: <<http://www.objectmentor.com/resources/articles/oodmetrc.pdf>>. Acesso em: 14 abr 2014.
- MARTIN, R. *The Open-Closed Principle*. 1997. Disponível em: <<http://www.objectmentor.com/resources/articles/ocp.pdf>>. Acesso em: 17 abr 2014.
- MARTIN, R. *Clean Code - A Handbook of Agile Software Craftsmanship*. [S.l.]: Pearson Education Inc., 2009.
- MARTINS, J. C. C. *Técnicas para gerenciamento de projetos de software*. [S.l.]: Brasport, 2007.
- MCCABE, T. J. A complexity measure. In: *Proceedings of the 2nd International Conference on Software Engineering*. Los Alamitos, CA, EUA: IEEE Computer Society Press, 1976. (ICSE '76), p. 407–.
- MILLANI, L. F. G. *Análise de Correlação Entre Métricas de Qualidade de Software e Métricas Físicas*. 2013. Disponível em: <<http://www.lume.ufrgs.br/bitstream/handle/10183/66095/000870909.pdf?sequence=1>>. Acesso em: 10 abr 2014.
- MITRE CORPORATION. *Php-nuke : Vulnerability Statistics*. 2014. Disponível em: <<http://www.cvedetails.com/vendor/961/Php-nuke.html>>. Acesso em: 10 março 2014.
- MUNROE, A. *PHP: a fractal of bad design*. 2013. Disponível em: <<http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/>>. Acesso em: 13 março 2014.
- NBR ISO/IEC. 9126-1:2003. Engenharia de software - Qualidade de produto - parte 1: Modelo de qualidade. *Rio de Janeiro: ABNT*, 2003.
- PAULA FILHO, W. de P. *Engenharia de Software: fundamentos, métodos e padrões*. [S.l.]: LTC, 2000.
- PHP-FIG. *PHP-Fig Faq*. 2013. Disponível em: <<http://www.php-fig.org/faq>>. Acesso em: 17 março 2014.
- PICHLER, M. *PHPMD - PHP Mess Detector*. 2009. Disponível em: <<http://http://phpmd.org/>>. Acesso em: 22 março 2014.
- PICHLER, M. *Manual do PHP Depend: Abstraction Instability chart*. 2013. Disponível em: <<http://pdepend.org/documentation/handbook/reports/abstraction-instability-chart.html>>. Acesso em: 16 abr 2014.

- PICHLER, M. *Manual do PHP Depend: CA - Afferent Coupling*. 2013. Disponível em: <<http://pdepend.org/documentation/software-metrics/afferent-coupling.html>>. Acesso em: 14 abr 2014.
- PICHLER, M. *Manual do PHP Depend: CE - Efferent Coupling*. 2013. Disponível em: <<http://pdepend.org/documentation/software-metrics/efferent-coupling.html>>. Acesso em: 14 abr 2014.
- PICHLER, M. *Manual do PHP Depend: CSZ - Class Size*. 2013. Disponível em: <<http://pdepend.org/documentation/software-metrics/class-size.html>>. Acesso em: 14 abr 2014.
- PICHLER, M. *Manual do PHP Depend: NPM - Number of Public Methods*. 2013. Disponível em: <<http://pdepend.org/documentation/software-metrics/number-of-public-methods.html>>. Acesso em: 15 abr 2014.
- PICHLER, M. *Manual do PHP Depend: Overview Pyramid*. 2013. Disponível em: <<http://pdepend.org/documentation/handbook/reports/overview-pyramid.html>>. Acesso em: 16 abr 2014.
- PICHLER, M. *Manual do PHP Depend: Reports*. 2013. Disponível em: <<http://pdepend.org/documentation/handbook/reports.html>>. Acesso em: 16 abr 2014.
- PICHLER, M. *Manual do PHP Depend: WMC - Weighted Method Count*. 2013. Disponível em: <<http://pdepend.org/documentation/software-metrics/weighted-method-count.html>>. Acesso em: 15 abr 2014.
- POTENCIER, F. *The PHP Coding Standards Fixer for PSR-1 and PSR-2*. 2014. Disponível em: <<http://cs.sensiolabs.org/>>. Acesso em: 27 abril 2014.
- PRESSMAN, R. S. *Engenharia de Software*. [S.l.]: Mcgraw-hill Interamericana, 2011.
- REZENDE, D. A. *Engenharia de Software e Sistemas de Informação*. [S.l.]: Brasport, 2005.
- RITCHIE, D. M. The development of the c language. In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, EUA: ACM, 1993. p. 201–208.
- SALSI, U. *PHPLint - Features*. 2014. Disponível em: <<http://www.icosaedro.it/phplint/features.html>>. Acesso em: 25 abril 2014.
- SALSI, U. *PHPLint - Tutorial*. 2014. Disponível em: <<http://www.icosaedro.it/phplint/phplint2/doc/tutorial.htm>>. Acesso em: 25 abril 2014.
- SENSIO LABS. *Coding Standards*. 2014. Disponível em: <<http://symfony.com/doc/current/contributing/code/standards.html>>. Acesso em: 15 abril 2014.
- SHERWOOD, G. *PHP_CodeSniffer Wiki*. 2012. Disponível em: <https://github.com/squizlabs/PHP_CodeSniffer/wiki>. Acesso em: 15 abril 2014.
- SKINNER, J. *Sublime Text - home*. 2014. Disponível em: <<http://www.sublimetext.com/>>. Acesso em: 23 abril 2014.
- SKVORC, B. *Battle of the Autoloaders: PSR-0 vs. PSR-4*. 2013. Disponível em: <<http://www.sitepoint.com/battle-autoloaders-psr-0-vs-psr-4/>>. Acesso em: 20 março 2014.

- SOMMERVILLE, I. *Engenharia de Software*. [S.l.]: Addison Wesley Editora, 2007.
- TERRA, R. *Ferramentas para análise estática de código Java*. 2008. Disponível em: <<http://www.ricardoterra.com.br/publications/>>. Acesso em: 10 abr 2014.
- TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. In: *III Encontro Brasileiro de Teste de Software (EBTS)*. [S.l.: s.n.], 2008. p. 1–5.
- THE PHP GROUP. *Documentação da classe PEAR_Error*. 2004. Disponível em: <http://pear.php.net/manual/pt_BR/core.pear.pear-error.php>. Acesso em: 13 março 2014.
- THE PHP GROUP. *Exceções*. 2007. Disponível em: <http://php.net/manual/pt_BR/language.exceptions.php>. Acesso em: 13 março 2014.
- THE PHP GROUP. *História do PHP*. 2013. Disponível em: <http://www.php.net/manual/pt_BR/history.php.php>. Acesso em: 05 dezembro 2013.
- THE PHP GROUP. *Home page do PHP*. 2013. Disponível em: <<http://www.php.net>>. Acesso em: 05 dezembro 2013.
- THE PHP GROUP. *Coding standards*. 2014. Disponível em: <<http://pear.php.net/manual/en/coding-standards.php>>. Acesso em: 28 janeiro 2014.
- THE PHP GROUP. *Command Line Options*. 2014. Disponível em: <<http://www.php.net/manual/en/features.commandline.options.php>>. Acesso em: 24 abril 2014.
- THE PHP GROUP. *Manual do PHP - Documentação da função error_reporting*. 2014. Disponível em: <<http://br2.php.net/manual/en/function.error-reporting.php>>. Acesso em: 02 fevereiro 2014.
- THE PHP GROUP. *Manual do PHP - Usando namespaces*. 2014. Disponível em: <<http://www.php.net/manual/en/language.namespaces.importing.php>>. Acesso em: 02 fevereiro 2014.
- THE PHP GROUP. *O que o PHP pode fazer?* 2014. Disponível em: <http://www.php.net/manual/pt_BR/intro-whatcando.php>. Acesso em: 12 fevereiro 2014.
- THE PHP GROUP. *O que você faria com 5 milhões de linhas de código? - Blog oficial da pear.php.net*. 2014. Disponível em: <<http://blog.pear.php.net/2012/01/24/what-would-you-do-with-5-million-lines-of-code/>>. Acesso em: 02 fevereiro 2014.
- THE PHP GROUP. *Pear - Home*. 2014. Disponível em: <<http://pear.php.net/>>. Acesso em: 28 janeiro 2014.
- THE PHP GROUP. *Pear - Packages*. 2014. Disponível em: <<http://pear.php.net/packages.php>>. Acesso em: 28 janeiro 2014.
- THE PHP GROUP. *PEAR - PHP Extension and Application Repository*. 2014. Disponível em: <<https://github.com/pear>>. Acesso em: 02 fevereiro 2014.
- THE PHP GROUP. *Pyrus: Improvements from the PEAR Installer*. 2014. Disponível em: <<http://pear.php.net/manual/en/pyrus.differences.frompear.php>>. Acesso em: 28 janeiro 2014.
- WAGNER, S. et al. Comparing bug finding tools with reviews and tests. In: *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*. [S.l.]: Springer-Verlag, 2005. p. 40–55.

WARD, T. *Google Drive for WordPress - It simply does not Work due to Poor Code*. 2013. Disponível em: <<http://wordpress.org/support/topic/it-simply-does-not-work-due-to-poor-code>>. Acesso em: 12 fevereiro 2014.

WASTL, E. *PHP Sadness*. 2014. Disponível em: <<http://phpsadness.com>>. Acesso em: 13 março 2014.

WIEGERS, K. E. *Peer Reviews in Software: A Practical Guide*. [S.l.]: Addison Wesley, 2002.

WIEGERS, K. E. *Software Peer Reviews: An Executive Overview*. 2012. Disponível em: <[http://www2.smartbear.com/rs/smartbear/images/\(Karl Wiegers\) Software-Peer-Reviews-An-Executive-Overview-KW_final.pdf](http://www2.smartbear.com/rs/smartbear/images/(Karl+Wiegers)+Software-Peer-Reviews-An-Executive-Overview-KW_final.pdf)>. Acesso em: 27 abril 2014.

WORDPRESS. *Wordpress - história*. 2013. Disponível em: <<https://codex.wordpress.org/History>>. Acesso em: 12 fevereiro 2014.

WORDPRESS. *Wordpress - stats*. 2014. Disponível em: <<http://en.wordpress.com/stats/>>. Acesso em: 05 fevereiro 2014.

WORDPRESS. *WordPress Coding Standards*. 2014. Disponível em: <http://codex.wordpress.org/WordPress_Coding_Standards>. Acesso em: 12 fevereiro 2014.

ZEND TECHNOLOGIES. *Zend Framework 2 Coding Standards*. 2014. Disponível em: <<http://framework.zend.com/wiki/display/ZFDEV2/Coding+Standards>>. Acesso em: 14 abr 2014.