ALINHANDO PERSPECTIVAS DE QUALIDADE EM CÓDIGO FONTE A PARTIR DE ESTUDOS EXPERIMENTAIS – UM CASO NA INDÚSTRIA

Talita Vieira Ribeiro

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Guilherme Horta Travassos

Rio de Janeiro
Setembro de 2014

ALINHANDO PERSPECTIVAS DE QUALIDADE EM CÓDIGO FONTE A PARTIR DE ESTUDOS EXPERIMENTAIS – UM CASO NA INDÚSTRIA

Talita Vieira Ribeiro

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____

Prof. Guilherme Horta Travassos, D.Sc.

_____

Profª. Cláudia Maria Lima Werner, D.Sc.

_____

Prof. Alessandro Fabricio Garcia, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2014

Ribeiro, Talita Vieira

Alinhando Perspectivas de Qualidade em Código Fonte a partir de Estudos Experimentais – Um Caso na Indústria / Talita Vieira Ribeiro – Rio de Janeiro: UFRJ/COPPE, 2014.

IX, 161 p.: il.; 29,7 cm.

Orientador: Guilherme Horta Travassos.

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 108-114.

1. Engenharia de Software. 2. Reconstrução de Código Fonte. 3. Qualidade de Código Fonte. 4. Diretrizes de Codificação. 5. Engenharia de Software Experimental. 6. Pesquisa-Ação. I. Travassos, Guilherme Horta II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Ao meu orientador Guilherme pelas conversas relacionadas e não relacionadas com o trabalho, pelo acompanhamento e direcionamento, pelas ideias, pelas cobranças, mas principalmente pela paciência que teve comigo. O senhor me fez conhecer um trabalho de um excelente professor, pesquisador e orientador no qual eu desejo me espelhar em muitos aspectos.

À empresa Alfa na pessoa dos seus dirigentes, gerentes e desenvolvedores. Poder trocar conhecimento e experiências com vocês foi de grande valia para o meu crescimento profissional e permitiu a viabilização de um trabalho onde academia e indústria pudessem de fato colaborar.

Aos professores Cláudia e Alessandro por participarem da minha banca de defesa de mestrado.

À amiga Luciana, pelas conversas sobre ciência, ensino/aprendizagem e pesquisa, às vezes filosóficas, mas que sempre me ajudaram no meu crescimento pessoal e acadêmico. Pra mim, você é uma das evidências que tenho de que uma pesquisa não é feita somente com orientando, orientador e artigos, mas também feita com pessoas críticas que estejam dispostas a ouvir e questionar aquilo que estamos fazendo.

Aos colegas do Grupo de Engenharia de Software Experimental, principalmente ao Breno, Jobson, Paulo, Rafael, Thiago, Verônica e Victor, que contribuíram com discussões coerentes para o andamento do meu trabalho.

Por fim, à CAPES pelo financiamento.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ALINHANDO PERSPECTIVAS DE QUALIDADE EM CÓDIGO FONTE A PARTIR DE ESTUDOS EXPERIMENTAIS – UM CASO NA INDÚSTRIA

Talita Vieira Ribeiro

Setembro/2014

Orientador: Guilherme Horta Travassos
Programa: Engenharia de Sistemas e Computação

Esta dissertação apresenta um conjunto de diretrizes de codificação para legibilidade e compreensibilidade de código fonte baseadas em evidência e configuradas para um contexto organizacional de desenvolvimento de *firmware* e software embarcado. A concepção das diretrizes surgiu como forma de alinhar perspectivas de diferentes programadores sobre qualidade em código fonte, após a descoberta de constantes modificações nos códigos dos projetos da organização para adequá-los a uma perspectiva pessoal de qualidade – atividade que denominamos de reconstrução de código fonte. Após a solicitação por parte da organização de auxílio à realização de atividades vistas como sendo de refatoração de código, um *survey* exploratório foi conduzido junto aos programadores, revelando a existência de reconstrução de código fonte na organização, não de refatoração de código. Tendo como base esse diagnóstico, atributos para legibilidade e compreensibilidade de código fonte foram identificados a partir da condução de um estudo baseado em revisão sistemática da literatura e de análises de códigos fonte da própria organização. O resultado foi um conjunto mínimo e necessário de diretrizes para legibilidade e compreensibilidade de código fonte que foi avaliado na organização através de um *focus group*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

TOWARDS A SOURCE CODE QUALITY PERSPECTIVES ALIGNMENT THROUGH EXPERIMENTAL STUDIES – AN INDUSTRIAL CASE

Talita Vieira Ribeiro

September/2014

Advisor: Guilherme Horta Travassos
Department: Computer Science and Systems Engineering

This dissertation presents a set of evidence-based coding guidelines for readability and comprehensibility of source code. These guidelines were formulated through the conduction of a series of experimental studies under the action-research methodology that has been applied in an industrial case. The results of an exploratory survey in a firmware and embedded software development company have shown that many of the re-work that had been happening in the company were caused by a misalignment of source code quality perspectives. Another important information observed with the survey results was about the main source code quality characteristics seen as priority for the developers, pointed out as readability and comprehensibility. In this way, the coding guidelines were formulated to contribute to the source code quality in these characteristics, to reduce the existing misalignment, and, by doing this, to reduce the re-work. A literature review based on a systematic review was conducted as a way to identify source code attributes that have impact on readability and comprehensibility of source code. These source code attributes and the analysis of the company's source code were used as input for the formulation of the coding guidelines, evaluated through a focus group with developers of the company.

Sumário

1 Introdução

Neste capítulo, são expostos a motivação e o contexto de realização do trabalho, além do problema, objetivos e metodologia de pesquisa adotada. A organização do trabalho é apresentada ao final do capítulo.

1.1 Motivação e Contexto

Não é difícil encontrar na literatura técnica relatos ou mesmo indícios da falta de consenso entre pesquisadores sobre os conceitos da engenharia de software. Diferentes termos são utilizados para descrever fenômenos similares e também um mesmo termo é utilizado para identificar fenômenos diversos. Esse desalinhamento de conceitos acaba prejudicando o compartilhamento e a disseminação de conhecimento científico, uma vez que cada grupo de pesquisa acaba se utilizando somente de conceitos e definições próprias para capturar e disseminar descobertas científicas (LOPES e TRAVASSOS, 2009)(ŠMITE *et al.*, 2014).

Na indústria, observamos que esse comportamento não é tão diferente. Embora muito do que é relatado na literatura técnica esteja atrelado a profissionais da prática em início de carreira que possuem divergências de concepções com a indústria (SUDOL e JASPAN, 2010), acreditamos que existe desalinhamento de conceitos e definições entre os membros mais experientes de uma mesma equipe de desenvolvimento e, também, entre a indústria e a academia. Esse fato pode prejudicar a comunicação entre os desenvolvedores e a procura por soluções para os problemas de desenvolvimento existentes em uma organização, uma vez que a falta de definição inequívoca do problema pode acarretar na identificação de soluções inadequadas para sua resolução. Um exemplo de conceito que acarreta divergências em sua definição é refatoração de código fonte, algumas vezes usada para indicar a reestruturação do código fonte visando atingir alguma característica de qualidade, outras vezes usada para indicar a pura reconstrução do código fonte sem foco de melhoria aparente.

Neste trabalho, refatoração de código fonte representa a atividade sistemática de reestruturar o código fonte para atender a algum objetivo de qualidade previamente definido e conhecido pelos diferentes participantes de uma mesma equipe de desenvolvimento. Fowler *et al.* (1999), enfatizaram que os aspectos de melhoria arquitetural e preservação do comportamento externo do software devem ser garantidos pelas atividades de refatoração de código fonte. Entretanto, a preservação do comportamento externo tem sido questionada nos resultados de estudos

experimentais recentemente realizados na indústria (KIM, ZIMMERMANN e NAGAPPAN, 2012).

Reconstrução de código fonte, por outro lado, é caracterizada neste trabalho como a atividade não sistemática de reescrita do código fonte para atender a um objetivo de qualidade pessoal, ou seja, não necessariamente conhecido ou pré-definido e sem consenso entre os diferentes participantes de uma mesma equipe de desenvolvimento. A tentativa de atender objetivos pessoais de qualidade pode acarretar em sucessivas reconstruções de um mesmo código fonte por diferentes programadores. A reconstrução, então, contrapõe-se à refatoração de código, não só por não ter relação explícita com a melhoria arquitetural do software, mas também por não ter bem definido os objetivos de qualidade desejados para o código quando este é modificado, representando alto risco ao projeto de construção e manutenção do software.

No âmbito do desalinhamento e sobrecarga conceitual dos termos refatoração e reconstrução de código fonte é que esta dissertação está inserida. A partir de uma solicitação de auxílio em atividades ditas como sendo de refatoração de código fonte, que estavam ocorrendo em demasia em uma empresa de desenvolvimento de *firmware* e software embarcado[1], um conjunto de estudos experimentais foi conduzido como forma de melhor entender o contexto organizacional e auxiliar a organização nos problemas levantados. A aplicação de um *survey* exploratório com os programadores da empresa evidenciou que apesar de existir de fato refatoração nos códigos da organização, a reconstrução de código fonte era a atividade mais recorrente. Diante desse diagnóstico, viu-se a oportunidade de realização de uma pesquisa guiada por pesquisa-ação para apoiar a organização no alinhamento da perspectiva de qualidade de seus programadores, já que a falta desse alinhamento estava acarretando as constantes reconstruções de código.

Com base em informações também coletadas durante o diagnóstico sobre as características de qualidade desejadas para compor os códigos fonte da empresa, uma busca baseada em revisão sistemática foi conduzida para identificar atributos de

[1] Para este trabalho utilizaremos as seguintes definições: i) firmware é o conjunto de instruções operacionais programadas no hardware de um equipamento eletrônico; essas instruções estão relacionadas principalmente com o gerenciamento dos componentes de hardware do equipamento; ii) software embarcado é um software para um propósito restrito que tem o hardware dedicado às tarefas que executa; essa dedicação do hardware é definida pelo firmware do equipamento.

qualidade em código fonte baseados em evidência[2] que pudessem auxiliar a organização a: i) alinhar as perspectivas de qualidade dos programadores; ii) alcançar a qualidade esperada para os códigos fonte da empresa; e iii) reduzir a incidência de reconstrução e, por conseguinte, de retrabalho nos projetos de manutenção. Além desses estudos, uma análise de códigos fonte da empresa e um *focus group*[3] foi conduzido para ajustar os resultados obtidos na literatura técnica ao contexto da organização.

1.1.1 Cooperação Academia-Indústria e Breve Contexto Organizacional

Esta dissertação foi resultado da pesquisa realizada pela pesquisadora durante o período em que participou de um projeto de cooperação entre o Grupo de Engenharia de Software Experimental da COPPE/UFRJ (Grupo ESE) e a empresa Alfa – como será chamada ao longo deste trabalho. O projeto de cooperação em questão teve como objetivo principal servir como instrumento de cooperação academia-indústria para de um lado subsidiar oportunidades de pesquisa na área de engenharia de software com foco em experimentação e de outro fornecer conhecimento e tecnologias de apoio ao desenvolvimento de software. Dentre os temas tratados ao longo do projeto estão: captura e modelagem de processos, gerência de projetos, requisitos, testes e qualidade de código fonte. Este último sendo o tema em que esta dissertação está inserida.

A empresa Alfa é uma empresa de desenvolvimento de soluções de hardware, firmware, software embarcado, software *desktop*, *web* e *mobile* para os domínios de rastreamento e inteligência ambiental. Possui subdivisões localizadas em diferentes cidades do Brasil e do Exterior responsáveis por setores específicos da produção de um novo produto que passa desde o projeto e montagem do hardware até o desenvolvimento e manutenção do firmware e software embarcado. Software *desktop*, *web* e *mobile*, apesar de também serem desenvolvidos na empresa, são de menor expressão quando comparados com o desenvolvimento de firmware e software embarcado e geralmente tem como objetivo apoiar a manipulação e visualização de informações providas pelo software embarcado dos produtos.

---

[2] O termo “baseado em evidências” se refere ao uso de evidências científicas obtidas através de estudos experimentais para fundamentar conceitos ou mesmo auxiliar na formulação de alguma tecnologia de software.

[3] *Focus Group* é uma estratégia de dinâmica em grupo que possibilita a discussão de um tópico de modo não estruturado e natural. Detalhes sobre *Focus Group* são apresentados no Capítulo 5 desta dissertação.

A empresa tem como foco desenvolver produtos inovadores e essa característica acaba modificando o processo de concepção de um novo produto, uma vez que os clientes e idealizadores dos produtos fazem parte da própria organização. O processo de desenvolvimento acaba também sendo impactado por essa característica, já que a proximidade com os clientes e idealizadores favorece a criação de um ambiente onde o conhecimento tácito supera o explícito.

Estando há 15 anos no mercado, a empresa desenvolveu diferentes tipos de rastreadores – para diferentes propósitos, mas com foco para veículos e carga – que se distinguem tanto em relação aos componentes de hardware, quanto funcionalidades de firmware e software embarcado, porém que mantém um conjunto mínimo de tipos de componentes de hardware e funcionalidades de firmware e software embarcado em comum. As similaridades entre os diferentes produtos possibilitaram o reaproveitamento de código fonte de um produto para outro ao longo desses 15 anos, na medida em que houve evolução dos componentes de hardware ou mesmo surgimento de novas oportunidades de atuação no mercado de rastreamento. Apesar da similaridade com o conceito de linha de produto de software, é importante destacar que o processo de reaproveitamento de código fonte dentro da empresa não tem nenhum tipo de gerenciamento ou controle, e muito código legado ainda existe nos produtos.

1.2 Problema e Questão de Pesquisa

Por estar inserido em um contexto de pesquisa para apoiar a resolução de um problema real dentro de uma organização de desenvolvimento de software, o problema de pesquisa deste trabalho está intimamente relacionado ao identificado na empresa Alfa. O principal problema levantado pela organização em relação ao tema de qualidade de código fonte foi o relacionado com a grande quantidade de esforço que as atividades de refatoração de código estavam demandando durante tarefas de manutenção (foco para atividades de reaproveitamento de código). Assim, nossas primeiras questões de pesquisa foram elaboradas como forma de entender: "*o que representa qualidade em código fonte para os programadores da empresa Alfa?*"; "*o que é entendido ser refatoração para os programadores da empresa Alfa?*"; "*que situações-problema têm levado os programadores da empresa Alfa a realizarem atividades de refatoração?*".

Detalhes dos resultados desse primeiro estudo de observação podem ser encontrados no Capítulo 2 desta dissertação, contudo é importante antecipar que a percepção de qualidade em código fonte e o entendimento sobre refatoração não era consenso entre os programadores da organização. Outro resultado de destaque desse

estudo foi a percepção equivocada sobre o que significava refatoração de código, vista por alguns como sendo correção de defeitos, inserção de novas funcionalidades e ainda reconstrução de código fonte. Os resultados desse estudo acrescidos da confirmação de informações junto aos desenvolvedores permitiram identificar que o que de fato ocorria na organização era reconstrução de código fonte e não refatoração e que isso ocorria devido à falta de consenso entre os programadores sobre o que era qualidade no código fonte.

Diante desse cenário, um novo conjunto de questões de pesquisa foi elaborado como forma de apoiar a condução dos demais estudos (guiados pela metodologia de pesquisa-ação) realizados na organização: "*como igualar as perspectivas de qualidade de código fonte entre os diferentes programadores da empresa Alfa?*"; "*é possível identificar preditores para reconstrução de código e assim diminuir o índice de retrabalho dentro dos projetos de manutenção da empresa Alfa?*".

1.3 Objetivos

O objetivo principal deste trabalho consiste em auxiliar a empresa Alfa a alinhar as perspectivas de qualidade em código fonte de seus desenvolvedores, provendo instrumentos que possibilitem esse alinhamento, como diretrizes de codificação para um conjunto específico de características de qualidade em código fonte. Duas características de qualidade foram selecionadas para compor esse conjunto inicial de diretrizes de codificação: legibilidade e compreensibilidade de código fonte.

A escolha por essas características é justificada através do diagnóstico inicial realizado junto aos programadores da empresa Alfa, que identificaram essas características de qualidade como sendo importantes (ver Capítulo 2). Além disso, acreditamos que para atender a qualidade de código fonte em outras características é preciso anteriormente conseguir ler e compreender o código fonte.

Assim, de forma a atingir o objetivo principal, os seguintes objetivos específicos foram traçados:

- Levantar atributos de código para legibilidade e compreensibilidade de código fonte baseados em evidência;
- Identificar atributos de qualidade em código fonte na indústria (em particular na empresa Alfa) avaliados como de alta e de baixa legibilidade e compreensibilidade de código;
- Formular diretrizes de codificação para legibilidade e compreensibilidade de código fonte, levando em consideração tanto o que a literatura técnica diz ser qualidade quanto o que os programadores consensuam ser qualidade no contexto de desenvolvimento da empresa Alfa;

- Avaliar a adequação das diretrizes formuladas para o contexto de desenvolvimento da empresa Alfa;
- Elaborar uma infraestrutura de apoio à disseminação das diretrizes entre os programadores da empresa Alfa.

1.4 Metodologia de Trabalho

A metodologia deste trabalho de dissertação foi inspirada na de pesquisa-ação. A pesquisa-ação é “um tipo de pesquisa social com base experimental que é concebida e realizada em estreita associação com uma ação ou com a resolução de um problema coletivo e no qual os pesquisadores e os participantes representativos da situação ou do problema estão envolvidos de modo cooperativo” (THIOLLENT, 2011). O processo de pesquisa-ação pode ser visto como um processo cíclico de cinco fases (SUSMAN e EVERED, 1978), apresentadas na Figura 1.1 a seguir.

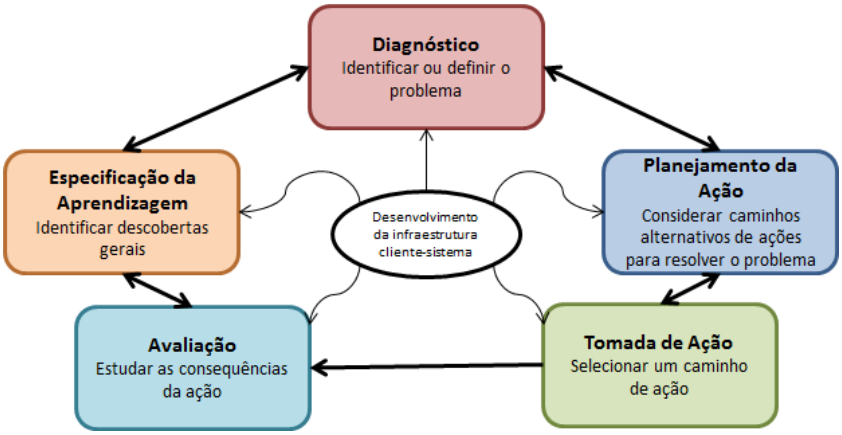


Figura 1.1 – Processo cíclico de 5 fases da pesquisa-ação – adaptado de (SUSMAN e EVERED, 1978)

A fase de **diagnóstico** compreende a identificação do problema de pesquisa em uma determinada área de conhecimento que se deseja resolver através de uma ação. É nessa fase que os interessados são identificados e o cliente expressa uma necessidade de melhoria que exige apoio científico para ser alcançada. Olhando para o aspecto experimental, diferentes estratégias podem ser utilizadas para realizar esse diagnóstico, como entrevistas, surveys e etnografias, dentre outras técnicas de descoberta de problemas.

O **planejamento da ação** dá início quando o quadro de diagnóstico estiver sido estabelecido. Hipóteses são então formuladas para traçar diferentes caminhos de ação que pode levar a resolução do problema diagnosticado. Esse planejamento pode requerer busca na literatura por conhecimento científico (teorias) que possam dar suporte às hipóteses levantadas ou ainda que forneçam subsídios para a formulação

das ações a serem tomadas para resolver o problema. Nesse sentido, revisões sistemáticas da literatura ou mesmo buscas estruturadas podem ser utilizadas para o levantamento desse conhecimento científico.

A fase de **tomada de ação** compreende justamente a seleção das ações que vão ser implantadas e a própria implantação das ações em si. Ao longo dessa fase, focus group, estudos de observação, inspeções e medições, podem ser utilizados para auxiliar na internalização e coleta de informações das ações implantadas.

A fase de **avaliação** compreende a análise dos impactos que a implantação das ações ocasionou ao ambiente e aos envolvidos. É importante que os resultados obtidos com medições, inspeções e observações sejam contrapostos com as hipóteses inicialmente levantadas, como forma de verificar se o problema inicial foi sanado.

A **especificação da aprendizagem** visa registrar e comunicar os resultados obtidos com a avaliação para os envolvidos no processo de pesquisa-ação. Nessa fase, também é avaliado o retorno ou não ao ciclo de pesquisa-ação, tomando como base a satisfação das duas partes (cliente-pesquisador) com o que até então foi alcançado. Diferentes estratégias de dinâmica de grupo podem ser utilizadas para comunicar os resultados e avaliar os próximos passos a serem dados, dentre os quais se encontra o brainstorming.

É importante destacar que as setas de transição de uma fase para a outra são bidirecionais. Isso significa que ao longo da execução do processo de pesquisa-ação pode ser que ocorra iteração e/ou adaptação entre as fases, possivelmente desencadeadas por novos problemas encontrados ou novas possibilidades de ação que possam ter surgido (SANTOS, 2009).

Como é possível inferir, a pesquisa-ação serve como um arcabouço de pesquisa que guia demais estudos experimentais. As subseções a seguir explicitam os estudos e atividades realizadas, bem como os produtos de trabalho gerados ao longo do desenvolvimento desta pesquisa de dissertação guiada por pesquisa-ação.

1.4.1 Estudos, Atividades e Produtos da Fase de Diagnóstico

A pesquisa teve início com a empresa Alfa solicitando auxílio com a execução das atividades de refatoração de código que estavam demandando muito esforço dos programadores – detalhes da sequência de atividades e resultados na Figura 1.2. Como forma de melhor diagnosticar o problema levantado pela empresa, um *survey* exploratório foi planejado para capturar as expectativas de qualidade em código fonte e as perspectivas de refatoração de código existentes na organização. O plano do *survey* foi gerado e avaliado por pesquisadores internos do grupo ESE e sua

condução ocorreu em três momentos, dois presenciais e outro via correio eletrônico. Os resultados do *survey* foram apresentados e confirmados junto aos próprios programadores que participaram do estudo, o que subsidiou a formulação de um diagnóstico sobre qualidade e refatoração de código fonte na empresa Alfa (detalhes no Capítulo 2).

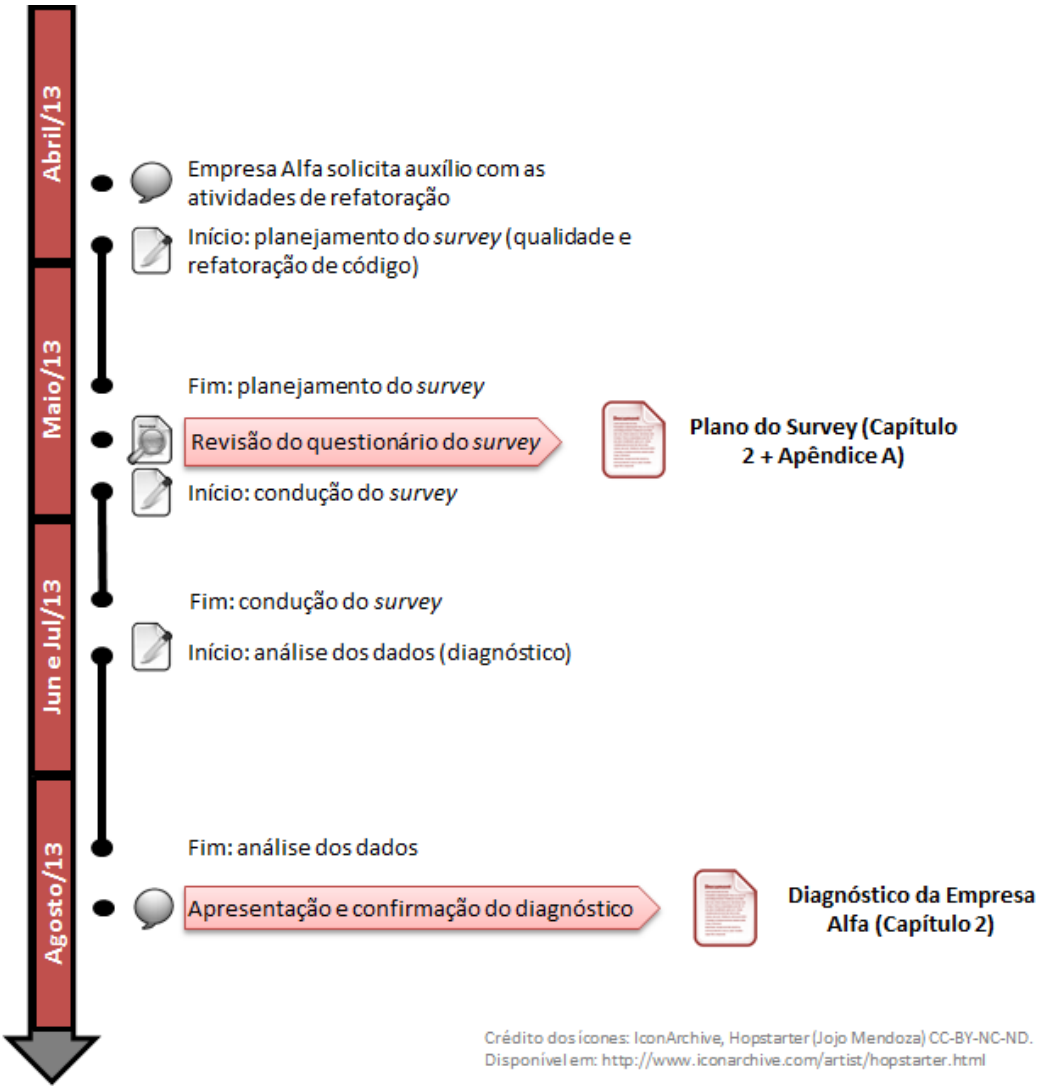


Figura 1.2 – Atividades e produtos de trabalho gerados na fase de diagnóstico

1.4.2 Estudos, Atividades e Produtos da Fase de Planejamento da Ação

O diagnóstico da empresa Alfa, que revelou a existência de reconstrução de código devido a um desalinhamento de perspectivas de qualidade, viabilizou a condução do estudo de pesquisa-ação deste trabalho. **Objetivando** contornar o problema de desalinhamento de perspectivas de qualidade em código fonte e, por conseguinte, diminuir a incidência de reconstrução de código fonte nos projetos da

empresa Alfa, propusemos como ação a formulação e aplicação de um conjunto de diretrizes de codificação baseadas em evidência e específicas para a organização. Partimos do pressuposto que as diretrizes serviriam então como fonte comum de conhecimento sobre atributos de qualidade em código fonte ao mesmo tempo em que estipulariam a forma como esses atributos de qualidade deveriam ser agregados ao código fonte dos projetos da empresa.

A proposta dessa ação foi apresentada e avaliada junto com os programadores da empresa Alfa. O interesse por parte da empresa desencadeou a execução do conjunto de atividades – elencadas na Figura 1.3 – relacionadas com a busca por atributos de qualidade em código fonte baseados em evidência e com a identificação de atributos de qualidade em código fonte nos códigos da empresa Alfa.

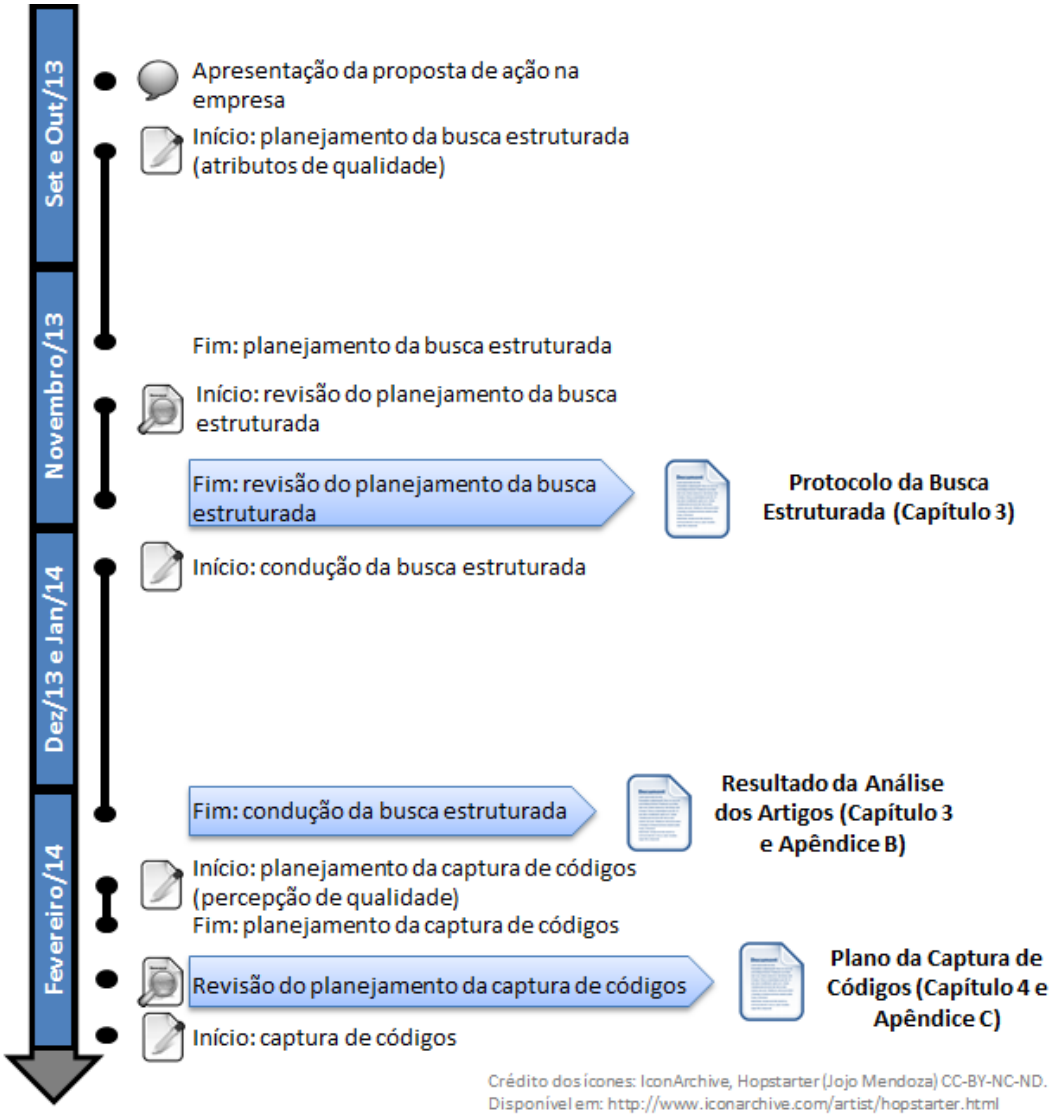


Figura 1.3 – Atividades e produtos de trabalho gerados na fase de planejamento (levantamento e identificação de atributos de qualidade em código fonte)

O propósito da busca estruturada e da captura de códigos na empresa foi, em ambos os casos, identificar atributos de legibilidade e compreensibilidade de código fonte; porém, no primeiro caso planejamos coletar atributos de código que tivessem evidências científicas sobre seus impactos na legibilidade e compreensibilidade de código; já no segundo caso tínhamos o objetivo de identificar atributos de código que proporcionavam legibilidade e compreensibilidade ao código e que eram relevantes para a organização. Nosso objetivo foi então unir evidência científica ao contexto de qualidade da empresa, fazendo a empresa participar mais ativamente do planejamento da ação proposta pela pesquisa-ação.

A Figura 1.3 e a Figura 1.4 evidenciam os produtos de trabalhos gerados com essas atividades. Além dos planejamentos da busca estruturada e da captura dos códigos e da análise dos resultados desses estudos, um conjunto de diretrizes para legibilidade e compreensibilidade de código fonte foi formulado.

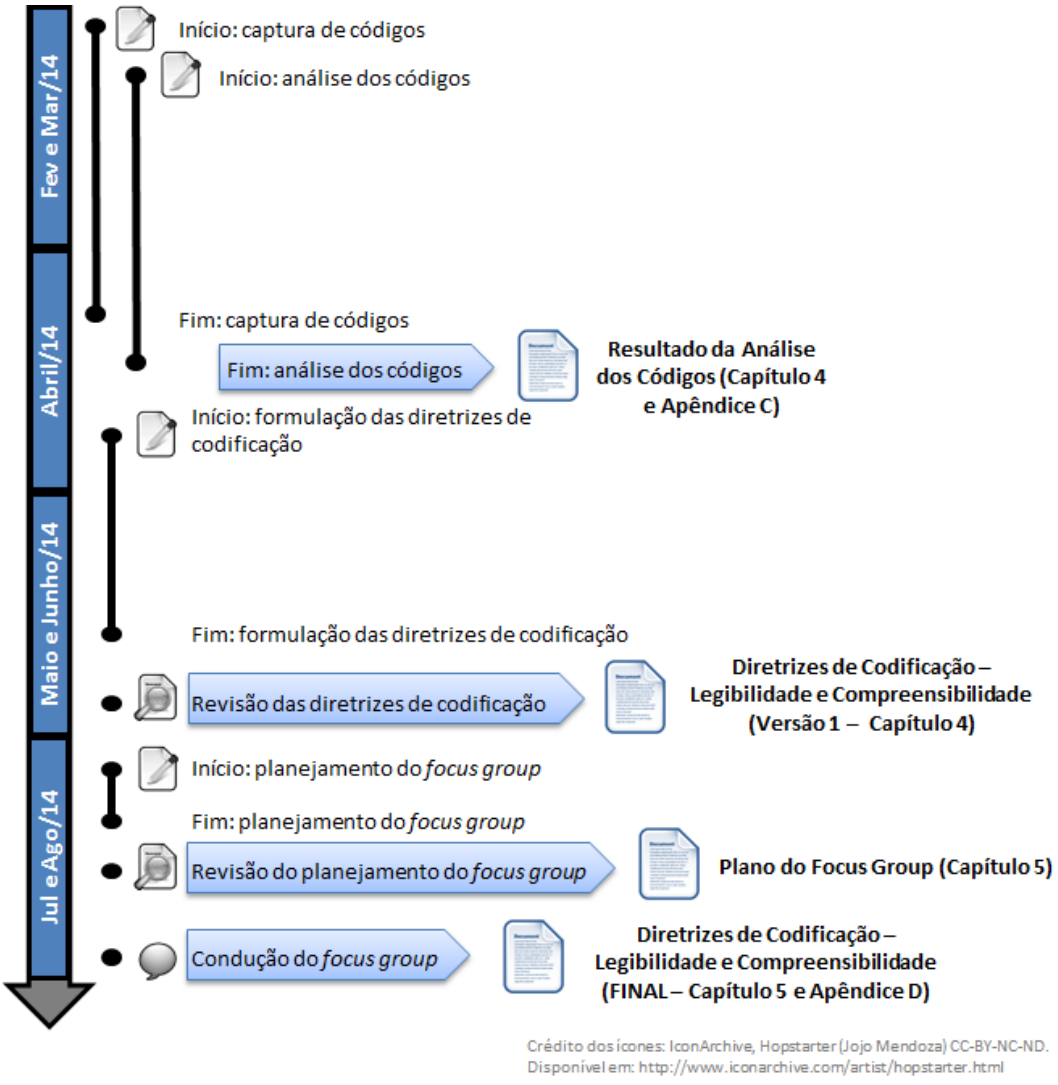


Figura 1.4 – Atividades e produtos de trabalho gerados na fase de planejamento (formulação e avaliação das diretrizes)

As diretrizes tiveram duas versões principais: uma formulada a partir dos resultados da busca estruturada e da análise dos códigos fonte da empresa Alfa; e outra pós-*focus group* no qual foram avaliadas e particularizadas ainda mais para o contexto organizacional. Além do documento, outras duas formas de apresentação das diretrizes foram fornecidas para a organização: um *folder* contendo o enunciado e a descrição das diretrizes, objetivando servir como instrumento de bancada de fácil acesso às diretrizes pelos programadores; e um hiperdocumento possível de ser acessado internamente por qualquer programador da organização. O hiperdocumento, além de fornecer o enunciado e a descrição das diretrizes, possibilita que o programador visualize os detalhes das diretrizes, incluindo: justificativas baseadas em evidência e no contexto organizacional para aplicação das diretrizes; códigos de exemplo e contraexemplo; descrição de comandos de uma ferramenta de formatação de código fonte que possibilita a automatização das diretrizes (quando aplicável); artigos utilizados para a construção das diretrizes; dentre outro conjunto de informações detalhados no Capítulo 4 desta dissertação.

Apesar de termos iniciado o trabalho adotando a metodologia de pesquisa-ação, evitamos classificar a metodologia seguida como sendo de pesquisa-ação pura por faltar atividades que compreendessem a implantação da ação e a avaliação dos impactos dessa implantação na empresa Alfa. Assim, atividades e produtos de trabalho referentes às demais fases da metodologia não puderam ser realizados no contexto desta dissertação, porém são esperados em trabalhos futuros. É importante destacar alguns fatores que inviabilizaram a realização dessas atividades no contexto deste trabalho, um deles foi o fator tempo para a realização da dissertação, que foi limitado tendo em vista o contexto de desenvolvimento da pesquisa: i) pesquisadores e empresa em estados distintos; ii) necessidade de disponibilidade dos programadores para a realização de algumas tarefas do trabalho; e iii) necessidade de coincidência de agendas entre participantes da academia e da indústria. Outro fator considerável foi a ausência de dados históricos que nos permitisse comparar a incidência e o esforço gasto em atividades de reconstrução pré e pós-implantação das diretrizes, fundamentais para a avaliação das ações implantadas no contexto de pesquisa-ação.

1.5 Organização do Texto

Neste capítulo de **Introdução,** foram apresentados a motivação para a realização do trabalho na área de qualidade de código fonte, bem como o contexto geral do trabalho. O problema, os objetivos do trabalho e a metodologia aplicada para o atendimento destes também foram identificados. Complementando o texto, os demais capítulos estão organizados da seguinte maneira:

O Capítulo 2 apresenta uma melhor contextualização da empresa Alfa bem como expõe os dados obtidos e as análises de diagnóstico realizadas com a condução do ***Survey* Exploratório sobre Qualidade e Refatoração de Código na Empresa Alfa**.

O Capítulo 3 faz uma breve introdução a conceitos relacionados com a área de qualidade de código fonte, com foco para os resultados de atributos de qualidade de código fonte baseados em evidência obtidos com a condução da **Busca Estruturada: Legibilidade e Compreensibilidade em Código Fonte**.

O Capítulo 4 apresenta as **Diretrizes de Codificação para Legibilidade e Compreensibilidade de Código Fonte**, bem como o planejamento e a captura de informações de código fonte na empresa Alfa que serviram como base para a formulação de diretrizes específicas para o contexto da organização.

O Capítulo 5 descreve o processo e apresenta os resultados do ***Focus Group* de Avaliação de Pertinência das Diretrizes de Codificação** que foi realizado na organização.

Por fim, o Capítulo 6 apresenta as **Conclusões e Trabalhos Futuros** desta pesquisa, identificando também as contribuições e limitações existentes.

2 *Survey* Exploratório sobre Qualidade e Refatoração de Código na Empresa Alfa

Neste capítulo são apresentados o plano e os resultados do survey exploratório sobre qualidade e refatoração de código na empresa Alfa. Os resultados obtidos com esse survey foram de fundamental importância para o desencadeamento das demais atividades do trabalho.

2.1 Introdução

No Capítulo 1, fornecemos uma visão geral sobre a empresa Alfa, porém alguns detalhes adicionais são necessários para entender a forma de organização do trabalho e evolução dos produtos dentro da empresa que possam influenciar atividades de construção e manutenção de software.

A empresa Alfa possui cinco diferentes setores que envolvem desenvolvimento de software. De forma simplificada e aproximada, podemos apresenta-los como (situação em julho de 2014):

i) desenvolvimento de firmware e software embarcado para o domínio de rastreamento → 7 programadores no Brasil e 2 no Exterior;

ii) desenvolvimento de software *desktop* e *web* para o domínio de rastreamento → 2 programadores no Brasil e 1 no Exterior;

iii) desenvolvimento de firmware e software embarcado para o domínio de inteligência ambiental → 3 programadores no Brasil e 1 no Exterior;

iv) desenvolvimento de software *web* e *mobile* para o domínio de inteligência ambiental → 7 programadores no Brasil;

v) desenvolvimento de sistemas de informação para controle e auxílio a execução dos processos internos da empresa (e.g. processos administrativos, financeiros, de vendas entre outros) → 3 programadores no Brasil.

Em cada um desses setores é possível ainda encontrar diferentes equipes de desenvolvimento que trabalham na criação, desenvolvimento e manutenção de softwares específicos da organização. A Figura 2.1 ilustra o arranjo dos setores e das equipes de desenvolvimento dentro da empresa Alfa.

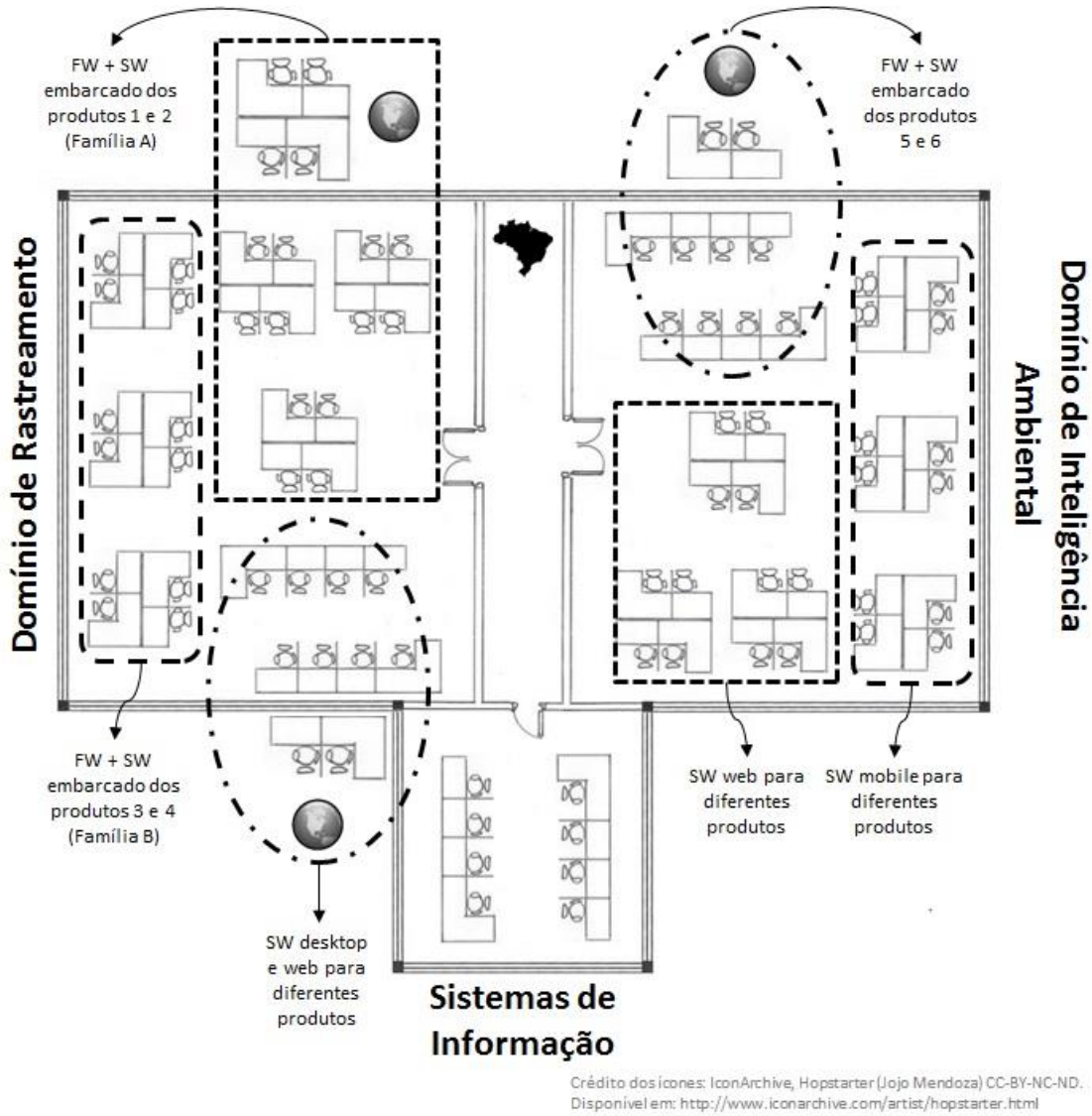


Figura 2.1 – Ilustração simplificada das diferentes equipes e setores de desenvolvimento da empresa Alfa

Algumas equipes específicas do Brasil trabalham em conjunto com desenvolvedores do Exterior, o que traz a característica de desenvolvimento distribuído de software para alguns projetos de desenvolvimento da organização – detalhes na Figura 2.1. Essa diferenciação de tipos de desenvolvimento (distribuído e colocalizado) entre as diferentes equipes da empresa acarretou na criação de diferentes "famílias de produtos"[4] para um mesmo domínio de desenvolvimento. No contexto de **desenvolvimento de firmware e software embarcado para o domínio**

[4] Um produto é visto como um composto de hardware, firmware e software embarcado. A empresa Alfa dá o nome de "família de produto" para o conjunto de produtos que evoluíram de um produto em comum e/ou que utilizam o mesmo protocolo de comunicação (recebimento e envio de comandos e informações).

de rastreamento, por exemplo, a empresa desenvolve produtos de duas famílias distintas: uma (A) resultante do desenvolvimento da equipe distribuída – Brasil e Exterior; e outra (B) resultante do desenvolvimento realizado pela equipe colocalizada – Brasil.

Os produtos de uma mesma família possuem, além do protocolo de comunicação de dados, componentes de hardware e funcionalidades de firmware e software embarcado muito parecidos; e, por geralmente terem evoluído de um mesmo produto, possuem também muito código fonte em comum. É importante enfatizar que o reaproveitamento de código de um produto para outro não é feito de forma sistemática e é possível encontrar trechos de código que sofreram modificação em um produto e que não sofreram modificação em outro produto.

Um produto específico pode ainda ter variações em relação a funcionalidades de firmware e software embarcado que são disponibilizadas para um determinado comprador. Essas variações de funcionalidades, apesar de estarem vinculadas a módulos do software, não causam diferença muito grande no código fonte de dois produtos de compradores distintos. Isso acontece porque é comum a prática de utilizar o mesmo código fonte, alterando apenas o acesso dos compradores às funcionalidades de seus respectivos produtos.

O setor de **desenvolvimento de software *desktop* e *web* para o domínio de rastreamento** é pequeno se comparado ao desenvolvimento de firmware e software embarcado. Envolve o desenvolvimento: i) dos softwares de configuração dos produtos de rastreamento; ii) das interfaces de comunicação dos produtos com softwares de terceiros (de compradores dos produtos); e iii) de softwares de visualização de informações e para controle remoto dos produtos. Esse setor já foi mais expressivo em relação ao desenvolvimento de soluções para visualização e controle remoto dos produtos, contudo, por questões estratégicas, a organização resolveu investir maior esforço no desenvolvimento de firmware e software embarcado para rastreamento e acabou remanejando muitos desenvolvedores desse setor para o setor de desenvolvimento de software *web* e *mobile* do domínio de inteligência ambiental. Assim como nos outros, a prática de reaproveitamento de código é recorrente neste setor, já que produtos de uma mesma família, por terem funcionalidades parecidas de software embarcado e por seguirem um mesmo protocolo de comunicação, também têm informações parecidas a serem configuradas e fornecidas, levando a similaridades de funcionalidades requeridas para configuração e disponibilização de dados.

O **desenvolvimento para o domínio de inteligência ambiental** é mais recente que o de rastreamento (aproximadamente dois contra 15 anos do de

rastreamento); e muitas ideias de produtos ainda estão em fase de prototipação. É sabido, no entanto, que o desenvolvimento de software *web* e *mobile* é mais presente nesse domínio do que no de rastreamento, já que o produto para a empresa Alfa, nesse caso, agrega ao composto de hardware, firmware e software embarcado, os softwares *web* e *mobile*. O desenvolvimento distribuído ocorre somente para o firmware e software embarcado. Em relação ao reaproveitamento, à refatoração e à reconstrução de código fonte, é precipitado tirar conclusões nesse momento, dado que os produtos ainda não saíram de sua fase de ideação e prototipação. Nesse estágio de desenvolvimento, o retrabalho é algo muito típico, não necessariamente para se atingir a um objetivo de qualidade, mas sim para se entender melhor sobre a própria solução a ser desenvolvida.

O setor de **desenvolvimento de sistemas de informação** é responsável por desenvolver sistemas de apoio a diferentes processos da organização, desde administrativos e financeiros até o de comunicação com compradores. Esse setor é responsável também por manter sistemas terceirizados. Por ser um setor de suporte à empresa em geral, é um setor que existe há muito tempo na organização e que lida com um conjunto grande de sistemas legados construídos com diferentes linguagens de programação.

2.1.1.1 Rotatividade das Equipes

Os últimos 18 meses foram de mudanças na organização. A própria criação de uma nova frente de desenvolvimento – inteligência ambiental – desencadeou uma série de transformações positivas na empresa, uma delas, o remanejamento de funcionários em diferentes setores. Os pesquisadores acompanharam essas mudanças, mesmo que remotamente, já que tínhamos consciência que isso poderia influenciar a condução dos estudos realizados na organização.

Após a criação do setor de desenvolvimento para o domínio de inteligência ambiental, muitos desenvolvedores da empresa do domínio de rastreamento migraram (aos poucos) para o desenvolvimento de soluções para esse novo domínio; desenvolvedores que trabalhavam com desenvolvimento de firmware e software embarcado continuaram nessa linha de desenvolvimento só que agora em outro domínio, o mesmo ocorrendo com os desenvolvedores de software *web*. Nesse período, também ocorreram muitas contratações, porém também houve saída de membros da equipe.

As seções seguintes fornecem detalhes do planejamento do survey exploratório conduzido na empresa Alfa, bem como dos resultados obtidos que serviram tanto como informações de contexto adicionais para diagnosticar a

organização como para evidenciar a causa do real problema de retrabalho na empresa Alfa: reconstrução de código fonte e não refatoração, como anteriormente imaginado.

2.2 Planejamento e Execução do *Survey* na Empresa Alfa

Como a solicitação de auxílio às atividades de refatoração de código ocorreu quatro meses depois do início do projeto de cooperação, como participantes do projeto, já tínhamos noção do panorama geral de desenvolvimento da empresa, contudo entendemos como necessário capturar informações mais específicas a respeito do tema de qualidade e refatoração de código fonte como forma de melhor entender a demanda da organização. Para isso, um *survey* exploratório foi considerado um instrumento útil para auxiliar no diagnóstico mais específico da empresa Alfa. As subseções a seguir detalham o plano e a execução do *survey* sobre qualidade e refatoração de código realizado na empresa Alfa.

2.2.1 Plano do *Survey* sobre Qualidade e Refatoração de Código na Empresa Alfa

O problema inicial levantado por um dos dirigentes da empresa foi que muitas atividades de refatoração de código que ocorriam na organização estavam demandando muito esforço dos programadores sem trazer os resultados esperados. Foi informado que era constante o reporte de realização de atividades de refatoração de código fonte na organização pelas equipes de desenvolvimento e que não era explícito o que estava sendo agregado ao código fonte em relação a sua qualidade. Outro fato que também ocorria na empresa era o de modificações realizadas no código fonte para agregar qualidade em uma característica específica de qualidade (e.g. modularidade) que acabavam influenciando negativamente outra característica de qualidade importante (e.g. desempenho), levando a um investimento de esforço em atividades que, ao final, não atendiam as expectativas de negócio.

Assim, como forma de melhor entender o contexto organizacional relacionado com as informações levantadas, iniciamos um planejamento de um *survey* sobre qualidade e refatoração de código na empresa Alfa com o propósito de responder às seguintes questões: "*o que representa qualidade em código fonte para os programadores da empresa Alfa?*"; "*o que é entendido ser refatoração para os programadores da empresa Alfa?*"; "*que situações-problema têm levado os programadores da empresa Alfa a realizarem atividades de refatoração?*".

2.2.1.1 Visão Geral sobre Survey

O *survey* é um tipo de estudo de investigação de informações em retrospecto, ou seja, investiga fatos, acontecimentos ou dados passados (TRAVASSOS, GUROV e AMARAL, 2002), traçando descrições, em sua maioria quantitativa, sobre diferentes aspectos de uma população. Geralmente, o *survey* é associado a questionários com pessoas como participantes, porém os participantes em um *survey* são mais bem definidos como sendo as fontes de informações desejadas para o estudo. Essa definição permite representar tanto pessoas, como projetos e artigos como participantes de um *survey*.

O *survey* distingue-se de simples questionários justamente por causa de sua metodologia científica (KASUNIC, 2005). O questionário faz parte do *survey* como um instrumento de coleta de informações, porém além do questionário é necessário que um conjunto de passos metodológicos (ver Figura 2.2) seja executado para que o estudo seja considerado um *survey*. Outra característica diferenciadora do *survey* de questionários é a capacidade de generalização dos resultados obtidos. A ideia é que os dados amostrais coletados em um *survey* sejam significativos, permitindo concluir sobre o comportamento da população alvo – geralmente uma grande quantidade de fontes de informação.

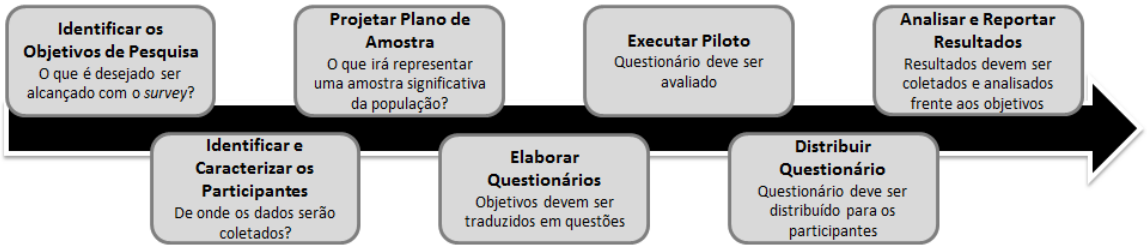


Figura 2.2 – Processo de condução de um *survey* – adaptado de (KASUNIC, 2005)

Dependendo do objetivo geral traçado para o *survey*, é possível classificá-lo em um dos três tipos (PINSONNEAULT e KRAEMER, 1993): i) descritivo – objetiva determinar a distribuição de atributos, características, eventos, atitudes ou opiniões que ocorrem em uma dada população; ii) explanatório – objetiva avaliar hipóteses e relacionamentos de causa e efeito entre variáveis, tentando identificar não somente a existência entre os relacionamentos, mas também a explicação para a existência deles; iii) exploratório – objetiva investigar um tópico, tentando traçar conceitos preliminares sobre ele para uma dada população. O objetivo geral do estudo irá influenciar diretamente o projeto do *survey*, a escolha da amostra da população, bem como os instrumentos de coleta de informações.

2.2.1.2 Objetivos do *Survey*

Escolhemos o tipo de *survey* exploratório e traçamos como objetivos do estudo, definidos utilizando a abordagem *Goal-Question-Metric* (GQM) proposta em (BASILI, CALDIERA e ROMBACH, 1994), os seguintes:

- ***Analisar** a percepção de qualidade de código fonte dos programadores da empresa Alfa **com o propósito de** caracterizar **com respeito ao** entendimento comum sobre qualidade e priorização de características de qualidade em código fonte **do ponto de vista dos** programadores da empresa Alfa **no contexto de** desenvolvimento de firmware, software embarcado e software.*
- ***Analisar** a percepção de refatoração de código fonte dos programadores da empresa Alfa **com o propósito de** caracterizar **com respeito ao** entendimento comum sobre o que representa e objetiva a refatoração de código fonte e para que ela é realizada no contexto da organização **do ponto de vista dos** programadores da empresa Alfa **no contexto de** desenvolvimento de firmware, software embarcado e software.*

2.2.1.3 Participantes e Plano de Amostra

Apesar de saber que o foco da questão colocada por um dos dirigentes da empresa estava mais relacionada a software embarcado do domínio de rastreamento, entendemos ser interessante capturar informações dos programadores da empresa nos diferentes setores, já que tínhamos identificado a existência de rotatividade de programadores na empresa Alfa e precisávamos conhecer melhor informações de contexto adicionais que pudessem influenciar qualquer ação a ser tomada para tratar a questão na organização. Dessa forma, planejamos executar o *survey* em uma única sessão presencial, utilizando formulários físicos (APÊNDICE A – Instrumentos do Estudo de Observação sobre Refatoração versus Reconstrução) e com todos os programadores da empresa Alfa localizados no Brasil.

2.2.1.4 Questionários

Três questionários contendo questões abertas, fechadas, de classificação, de ordenação e híbridas foram elaborados (ver APÊNDICE A – Instrumentos do Estudo de Observação sobre Refatoração versus Reconstrução):

1. Formulário de Caracterização: contém perguntas para conhecimento sobre o perfil do programador, como: tempo de indústria e de empresa; setor da empresa em que trabalha; papéis desempenhados na empresa; quantidade de

projetos[5] que participa; características dos projetos que participa; percepção de ocorrência de substituição dos membros da equipe; e experiências em codificação e refatoração. Além das perguntas de caracterização, o formulário contempla também uma solicitação de definição de refatoração de código fonte.

2. Formulário sobre Qualidade de Código Fonte: contém um conjunto de características de código fonte em formas de assertivas para o respondente elencar na importância para a qualidade de um código fonte; um conjunto de situações possíveis de serem encontradas em um código fonte para o respondente elencar sua frequência de ocorrência nos projetos em que participa; e um conjunto de razões para a existência dessas situações em código fonte para o respondente priorizar.
3. Formulário sobre Percepção de Refatoração de Código Fonte: contém questões sobre as atividades de refatoração realizadas pelo programador; um conjunto de situações (as mesmas identificadas no formulário de qualidade de código) para o respondente identificar a frequência com que elas o levam a refatorar o código; questões para capturar a motivação do programador para realizar uma refatoração e a opinião do programador sobre algumas assertivas a respeito da refatoração de código fonte. Algumas das questões sobre refatoração foram formuladas com base no *survey* executado por Kim, Zimmermann e Nagappan na Microsoft Corporation (KIM, ZIMMERMANN e NAGAPPAN, 2012) (KIM, ZIMMERMANN e NAGAPPAN, 2012).

Os formulários devem ser respondidos na seguinte ordem: 1º) formulário de caracterização; 2º) formulário de percepção de qualidade; e 3º) formulário de percepção de refatoração, sendo que não é permitido que o respondente tenha acesso às informações do formulário 2 sem antes finalizar o formulário 1, o mesmo se aplicando para o formulário 3 em relação ao formulário 2. Essa configuração é necessária para que as respostas dos primeiros formulários não sejam influenciadas pelos formulários seguintes.

O primeiro formulário tem como objetivo levantar características gerais sobre a organização que pudessem fornecer indícios sobre o motivo para algumas respostas aos questionamentos dos demais formulários. As respostas para o primeiro formulário,

[5] Para a empresa Alfa, um projeto existe para o ciclo de vida completo de um produto: hardware + firmware + software embarcado; software desktop; software web; ou software mobile. Isso quer dizer que para cada produto existe um único projeto que engloba construção e manutenção.

então, devem servir como instrumento de auxílio à análise das respostas dos outros dois formulários, mais relacionados com os objetivos de pesquisa do *survey*. De modo estratégico, a pergunta sobre refatoração de código fonte foi inserida no primeiro formulário sendo uma das primeiras perguntas realizadas, já que queremos entender a definição inicial dos programadores sobre essa atividade, sem que eles fossem influenciados pelos formulários seguintes.

O segundo formulário visa capturar a percepção de importância das características de qualidade de código fonte para os programadores da empresa Alfa. Além disso, tem o objetivo de identificar que situações, que contrapõem às características de qualidade, mais ocorrem na organização e quais são as possíveis e mais comuns causas para sua ocorrência. O objetivo é capturar não só a percepção de qualidade dos programadores, mas também informações de contexto que ajudem a entender as respostas para os questionamentos sobre refatoração de código fonte. É importante enfatizar que não temos o objetivo de fornecer uma lista completa de características de qualidade em código fonte nem de situações e problemas para serem analisadas pelos participantes. Entretanto, um campo adicional para explicitação de eventuais novos itens pelos programadores está disponível. Para o questionamento sobre características de qualidade em um código fonte, em especial, colocamos 3 falsos positivos (características não relacionadas com a qualidade em código fonte) como forma de avaliar a confiança nas respostas dos programadores. São eles: “não é extenso”; “apresenta baixa complexidade”; “é rápido de ser compilado”.

O terceiro formulário serve para capturar informações sobre a realização de atividades de refatoração de código. Inserimos algumas questões para capturar informações sobre o processo das atividades de refatoração, como frequência, planejamento e motivação para a realização das atividades de refatoração. Adicionalmente, queremos identificar que situações estão levando os programadores a realizarem tarefas de refatoração de código. Nesse caso, queremos verificar se existe alguma relação entre as situações mais recorrentes e as situações que os levam a refatorar o código fonte na empresa. Como forma mais uma vez de capturar o entendimento dos programadores sobre refatoração, inserimos uma questão para avaliar alguma situação de refatoração que ocorreu na organização e teve que ser interrompida; e outra questão com assertivas para avaliar a concordância dos programadores ao que é estipulado para a refatoração de código.

2.2.1.5 Agregação e Análise dos Dados

Como forma de facilitar a captura e agregação dos dados, para algumas questões foram dadas opções de respostas para classificação do respondente utilizando escala *likert*. Para os casos em que existe a possibilidade de surgir outras opções de respostas além das fornecidas no questionário, foram utilizadas questões híbridas, ou seja, questões que permitem o respondente adicionar novas opções ao conjunto de respostas fornecidas.

A agregação das respostas deve ser conforme o seguinte conjunto de critérios:

- Questões abertas: realização de codificação das respostas e contagem de ocorrência de cada código criado;
- Questões fechadas: contagem de ocorrência de cada opção de resposta dada;
- Questões de classificação (escala *likert*): seja w o peso da opção de classificação (ver Tabela 2.1); x a contagem de ocorrência da opção para uma dada resposta; e valorMaximo o máximo valor que a resposta pode assumir, a agregação a, em porcentagem, das classificações para uma resposta será como segue,

$$a = \left(\frac{\mathrm{x1w1} + \mathrm{x2w2} + \mathrm{x3w3} + \cdots + \mathrm{xnwn}}{\mathrm{valorMaximo}}\right)100$$

Tabela 2.1 – Pesos das opções de classificação das respostas

Pesos	Opções de Classificação das Respostas		
-3	Não se Aplica	Não Ocorre/Não Influencia	-
-2	-	-	Discordo Fortemente
-1	-	-	Discordo
0	-	-	Ainda Não Percebi
1	Desejável	Raramente	Concordo
2	Muito Importante	Eventualmente	Concordo Fortemente
3	Imprescindível	Sempre	-

- Questões de ordenação: agregação semelhante com a de questões de classificação, modificando o peso das opções, no caso, das colocações. O peso será definido de acordo com a quantidade de colocações existentes. A última colocação irá receber o peso 1, a penúltima, 2 e assim sucessivamente até que a primeira colocação receba o maior peso, correspondendo a quantidade de colocações existentes. Assim, seja w o peso da colocação; x a contagem de ocorrência da colocação para uma dada resposta; e valorMaximo o máximo valor que a resposta pode assumir – no caso que todos os

respondentes coloquem a resposta em primeiro lugar –, a agregação a, em porcentagem, das colocações para uma resposta seguirá a mesma fórmula de cálculo de questões de classificação.

Para a análise dos dados, as respostas das questões de classificação e de ordenação devem retornar para a sua escala inicial (*likert*). Para tanto, a análise de distribuição dos valores obtidos nas respostas deve ser feita, classificando as respostas nas opções da escala inicial de acordo com o quartil ou percentil correspondente de cada opção/colocação.

Por se tratar de um *survey* exploratório, é importante que os dados coletados sejam agrupados para os diferentes setores da empresa, bem como para a empresa como um todo, como forma de avaliar tendências de respostas. Para a análise, a questão de definição da refatoração deverá ser contraposta com as demais informações de refatoração. É possível que uma definição equivocada sobre a refatoração comprometa as demais respostas sobre o tema.

2.2.2 Avaliação dos Questionários

Não tivemos oportunidade de executar nenhum piloto dos questionários com uma amostra dos participantes, até mesmo porque não queríamos perder fontes de informação importantes para o diagnóstico da empresa. Contudo, e como forma de avaliar a aplicabilidade dos questionários, solicitamos que dois membros do grupo ESE (um aluno de iniciação científica e um de doutorando) e um aluno de mestrado da Linha de Pesquisa de Engenharia de Software do PESC/COPPE realizassem uma inspeção *ad-hoc* dos questionários.

Problemas reportados com a inspeção foram relacionados com ambiguidade e falta de clareza de algumas questões; organização de questões que tinham interdependência entre si (como as questões 1, 2 e 3 do formulário de percepção individual sobre refatoração que antes estavam na ordem 2 3 1); e organização das opções de resposta das questões. Além disso, os revisores contribuíram com ideias de opções para algumas questões dos formulários, além de indicar possíveis questões de ordenação que poderiam ser transformadas em questões de classificação, como forma de facilitar a resposta ao questionário pelos respondentes (como as questões 1 e 2 do formulário de percepção da qualidade do código fonte que antes eram de ordenação).

2.2.3 Execução do *Survey*

Embora tenha sido planejado para ser executado em uma única sessão presencial, a execução do *survey* ocorreu em três momentos distintos. Na época do planejamento todos os setores de desenvolvimento localizados no Brasil se

encontravam em uma única cidade brasileira, contudo, da data de planejamento até a data prevista para a execução, os setores de desenvolvimento para o domínio de inteligência ambiental migraram para outra cidade brasileira que, embora estivesse localizada no mesmo estado, inviabilizou a captura de informações desse grupo de programadores.

A primeira execução ocorreu no dia 24/05/2013 pessoalmente junto às diferentes equipes de desenvolvimento do domínio de rastreamento e de sistemas de informação (12 programadores), na qual foram distribuídos os três formulários na ordem previamente planejada. A segunda execução ocorreu no dia 10/06/2013 pessoalmente junto aos programadores do domínio de rastreamento e sistemas de informação (4 programadores) que não puderam participar do primeiro momento, na ocasião os mesmos formulários já descritos foram entregues para cada programador seguindo a ordem estabelecida. A terceira e última execução ocorreu remotamente, com o envio (dia 11/06/2013) dos formulários via correio eletrônico ao então responsável pelo desenvolvimento do domínio de inteligência ambiental, de modo que este pudesse repassar os formulários para os então programadores desse domínio (5 programadores). As instruções de como passar os questionários aos programadores foram também repassadas via correio eletrônico.

2.3 Análise e Reporte dos Resultados do *Survey* na Empresa Alfa

Os resultados do *survey* são apresentados nas subseções a seguir, sendo divididos da seguinte forma: informações gerais; informações sobre qualidade em código fonte; e informações sobre refatoração de código na empresa Alfa.

2.3.1 Informações Gerais

Como informação de contexto geral, é importante enfatizar que a migração de desenvolvedores do domínio de rastreamento para o domínio de inteligência ambiental ainda estava em estágio inicial durante a execução desse *survey*. Assim, 21 programadores respondentes do *survey* estavam distribuídos pelos setores da empresa como o apresentado na Tabela 2.2.

Tabela 2.2 – Distribuição de programadores nos setores da empresa Alfa (situação em junho de 2013)

Setor de Desenvolvimento	Programadores
FW e SW embarcado – rastreamento	10

SW - rastreamento	4
FW e SW embarcado – inteligência ambiental	1
SW – inteligência ambiental	4
Sistemas de Informação	2

Em relação à percepção de substituição de membros da equipe, muitos programadores identificaram que ocorria, porém de forma eventual ou rara (ver Figura 2.3). Somente dois programadores identificaram que não ocorria. É importante destacar que a entrada e saída de desenvolvedores não é tão comum na organização quanto o remanejamento de desenvolvedores de um setor de desenvolvimento para outro. Talvez esse fato tenha influenciado as respostas obtidas.

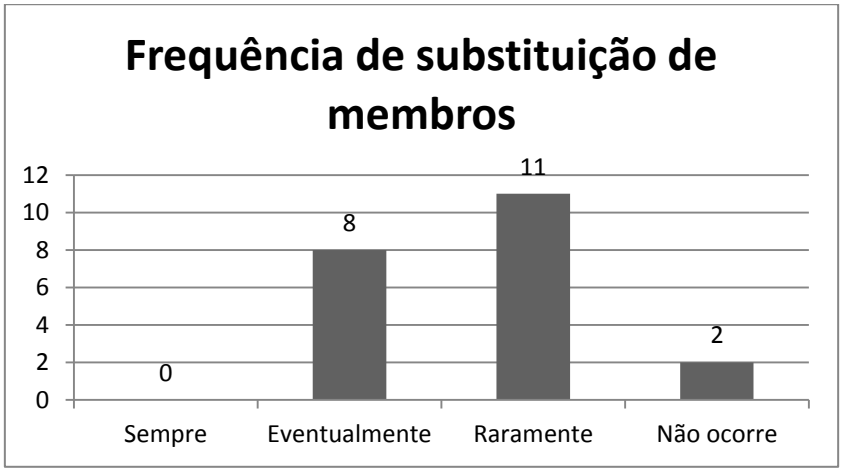


Figura 2.3 – Percepção de substituição de membros na equipe

Em média, cada programador participa de 2 projetos concomitantes, ou seja, trabalha na construção e manutenção de 2 produtos. Os diferentes produtos da organização de firmware, software embarcado, software *desktop*, *web* e *mobile* estão escritos em pelo menos uma das 13 linguagens de programação e de *script* seguintes: ActionScript, Assembly, C, C++, C#, JAVA, JavaScript, Lua, Objective C, Pascal (Delphi), Perl, PHP e Python. Linguagens como Assembly e C são utilizadas para o desenvolvimento do firmware dos produtos, enquanto que Lua vem sendo utilizada para o desenvolvimento dos softwares embarcados. Linguagens de *script* e Java são utilizadas para desenvolvimento *web* em geral; Java e Objective C para programação *mobile*; e Java, Pascal (Delphi) e PHP para construção de novos sistemas de informação e manutenção de alguns sistemas de informação legados.

A experiência dos programadores é bem diversa. A maioria somente estudou ou utilizou em 1 projeto requisitos para modelagem de arquitetura e codificação. Em relação ao uso de modelagem de software, a maioria dos programadores relatou ter utilizado modelos de estruturas ou diagramas de classes em alguns projetos na

indústria. Sobre refatoração, os programadores relataram experiência na refatoração de códigos próprios e de terceiros, embora não tenham feito planejamento nem análise de impacto das atividades de refatoração realizadas. Uma informação adicional sobre os programadores da organização e que foi observada durante o período do projeto de cooperação com a empresa Alfa é que há uma diversidade grande na formação dos programadores da empresa. Os cursos mais presentes são: engenharia elétrica, eletrônica, microeletrônica e de computação, além de sistemas de informação.

2.3.2 Percepção de Qualidade de Código Fonte

A percepção de qualidade de código fonte foi avaliada de duas formas. Primeiramente, analisamos a importância que algumas características de qualidade de código fonte tinham para os programadores da empresa Alfa. Depois avaliamos que situações-problema – algumas contrárias às características de qualidade – eram relatadas como sendo as mais frequentes nos códigos dos produtos da empresa.

Na questão para captura da percepção de importância das características de qualidade de código fonte, ao invés de fornecermos as características de qualidade diretamente para serem avaliadas, fornecemos uma frase que representasse a característica de qualidade, por entender ser essa uma forma mais intuitiva de avaliação pelos respondentes. O programador tinha ainda a opção de inserir novas frases que pudessem expressar alguma qualidade de código fonte adicional. A lista de frases utilizadas e o mapeamento para as características de qualidade são apresentadas na Tabela 2.3 a seguir.

Tabela 2.3 – Características de qualidade e frases equivalentes utilizadas no formulário 2

Característica de Qualidade	**Frase Equivalente**
Compreensibilidade	É de fácil compreensão.
Confiabilidade	Se mantém previsível em relação a seu funcionamento e desempenho nas condições estabelecidas para uso.
Conformidade com Padrões	Está de acordo com padrões previamente estabelecidos pela equipe de desenvolvimento.
Corretude/Efetividade	Provê as funcionalidades previstas.
Desempenho	Apresenta bom tempo de processamento.
Extensibilidade	É fácil de incluir novas funcionalidades ou comportamentos.
Falso Positivo 1 (não é extenso)	Não é muito extenso.

Falso Positivo 2 (é rápido de ser compilado)	É rápido de ser compilado.
Falso Positivo 3 (apresenta baixa complexidade)	Apresenta baixa complexidade.
Flexibilidade/Portabilidade	É fácil de ser adaptado para ambientes diferentes daquele para o qual tenha sido construído.
Legibilidade/Usabilidade	É fácil de usar e/ou fazer referência.
Manutenibilidade	É fácil de manter.
Redundância	Não possui trechos redundantes.
Reusabilidade	É fácil reaproveitá-lo.
Robustez	Utiliza de forma apropriada os recursos de memória e energia disponíveis.
Testabilidade	É possível ser testado.

Importante destacar a existência de três falsos positivos na listagem. Queríamos avaliar se a nossa percepção de qualidade em código fonte estava de alguma forma alinhada com a dos programadores da empresa. Os resultados da avaliação já agregados e retornados para a escala inicial são apresentados na Tabela 2.4. Os falsos positivos foram corretamente capturados pelos programadores que os identificaram como “não se aplica”.

Tabela 2.4 – Características de qualidade em ordem de importância para os desenvolvedores

% (valor agregado/máximo valor possível)	Característica de qualidade	Percepção de qualidade
98,33	Corretude/Efetividade	IMPRESCINDÍVEL
80,00	Confiabilidade	
73,33	Manutenibilidade	
71,67	Testabilidade	
66,67	Conformidade com Padrões	MUITO IMPORTANTE
65,00	Compreensibilidade	
63,33	Redundância	
63,33	Extensibilidade	
58,33	Legibilidade/Usabilidade	DESEJÁVEL
58,33	Robustez	
51,67	Reusabilidade	
51,67	Desempenho	
23,33	Flexibilidade/Portabilidade	NÃO SE APLICA
6,67	Falso Positivo 1 (não é extenso)	
0,00	Falso Positivo 3 (apresenta baixa complexidade)	
-18,33	Falso Positivo 2 (é rápido de ser compilado)	

Como esperado, as principais características de qualidade imprescindíveis para o código fonte foram as relacionadas com a corretude e confiabilidade do código, ou seja, para os programadores da empresa Alfa, um código tem qualidade quando provê as funcionalidades previstas e quando se mantém previsível em relação a seu funcionamento e desempenho nas condições estabelecidas para uso. Entendemos que a corretude está relacionada com requisitos, e a confiabilidade e testabilidade – quarta característica mais importante para a empresa – com testes de software. Assim, essas características de qualidade já estavam sendo trabalhadas junto com outros desenvolvedores da empresa através de atividades de requisitos e testes do projeto de cooperação ESE-Empresa Alfa.

A terceira característica mais importante para a organização, a manutenibilidade, por sua vez, é uma característica de qualidade que está intimamente relacionada com outras características de qualidade, como a legibilidade, compreensibilidade e extensibilidade. A manutenibilidade é então vista no código fonte, quando este consegue satisfazer a outras características de qualidade com as quais a manutenibilidade tem dependência. Possíveis novas ações na empresa Alfa para se atingir a qualidade de código desejada poderiam estar relacionadas com a conformidade com padrões e com a compreensibilidade de código fonte.

É importante destacar que essa questão para priorização das características de qualidade de código fonte era híbrida e três programadores indicaram uma nova frase para completar a fornecida: “Um código fonte possui qualidade quando...” “...está comentado ou documentado”. Vemos que essa nova frase identificada pelos programadores está relacionada com a compreensibilidade do código fonte – característica já identificada na listagem do questionário. Contudo, é interessante perceber a importância (e podemos inferir também a falta) dessa característica de qualidade para os códigos fonte da organização, já que ela foi identificada por três programadores diferentes.

As respostas para a questão relacionada com as situações que ocorrem mais frequentemente na organização foram menos consensual do que as das características de qualidade (ver percentual de cada situação na Tabela 2.5). Situações-problema relacionadas com legibilidade, manutenibilidade e coesão foram as identificadas como as que mais ocorrem dentre as apresentadas na listagem oferecida.

Tabela 2.5 – Situações mais frequentes na empresa Alfa

% (valor agregado/máximo valor possível)	Situação que ocorre na empresa	Ocorrência na empresa
56,67	Legibilidade deficiente, ou seja, variáveis, constantes, métodos, funções e procedimentos com nomes não intuitivos ou localizados em lugares inapropriados.	SEMPRE OCORRE
53,33	Código de difícil manutenção.	
48,33	Métodos, funções e procedimentos com muitas responsabilidades.	
46,67	Código de difícil reutilização.	EVENTUALMENTE OCORRE
46,67	Código muito fragmentado.	
45,00	Código de difícil extensão de funcionalidades.	
38,33	Existência de código legado com o qual é necessário trabalhar.	RARAMENTE OCORRE
36,67	Redundância de código.	
31,67	Baixo desempenho da aplicação.	
30,00	Forte dependência com código de outros times de desenvolvimento.	NÃO OCORRE
28,33	Variáveis, constantes, métodos, funções e procedimentos desnecessários.	
13,33	Dificuldade de testar o código sem antes aplicar refatoração.	
-1,67	Métodos, funções e procedimentos com muitos parâmetros desnecessários.	

Como possíveis motivos para a ocorrência dessas situações, os programadores ordenaram a lista de motivos fornecida e incluíram outros 2 motivos, como pode ser observado na Tabela 2.6 a seguir.

Tabela 2.6 – Principais motivos para a ocorrência das situações-problema na empresa Alfa

% (valor agregado/máximo valor possível)	Motivos	Colocação
66,67	Pressão para entrega do produto.	1º
61,11	Inexistência de requisitos explícitos.	
56,67	Inexistência de padrões previamente estabelecidos.	2º
45,56	Inexistência de modelos de projeto (arquitetura de software).	3º
21,11	Pouca experiência dos responsáveis pelo código.	4º
5,56	Novo (“Aplicativos de setores separados.”)	5º
4,44	Novo (“Falta de foco (muitos produtos).”)	

No topo da lista está a pressão para a entrega do produto que é algo inevitável. A pressão é inerente ao ambiente de trabalho de qualquer empresa que preza pela

competitividade e inovação. Fato interessante, no entanto, são os dois outros motivos elencados como os que mais influenciam as situações-problema: inexistência de requisitos e inexistência de padrões. O primeiro reforça a necessidade das práticas de requisitos que estavam sendo realizadas na organização e o segundo fornece indícios de ações que poderiam ser tomadas nessa área para auxiliar de alguma forma a organização nos problemas encontrados.

2.3.3 Percepção Individual sobre Refatoração de Código Fonte

Os dados a serem apresentados nesta subseção correspondem aos dados coletados de 20 programadores. Um programador indicou explicitamente que não tinha como responder sobre refatoração de código, uma vez que desconhecia o termo e a área.

A primeira questão sobre refatoração requeria justamente que o programador definisse o termo refatoração. Queríamos saber se todos entendiam uniformemente o conceito refatoração e se a prática que estava ocorrendo na organização era de fato refatoração de código. As respostas para essa questão, que era livre, foram codificadas e tiveram seus códigos separados em três grupos diferentes, um para cada dimensão das respostas obtidas. As dimensões são: i) “refatoração é...”; ii) “refatoração não pode...”; e iii) “refatoração auxilia no(a)...”. A Figura 2.4 a seguir apresenta os códigos do grupo “refatoração é...”.

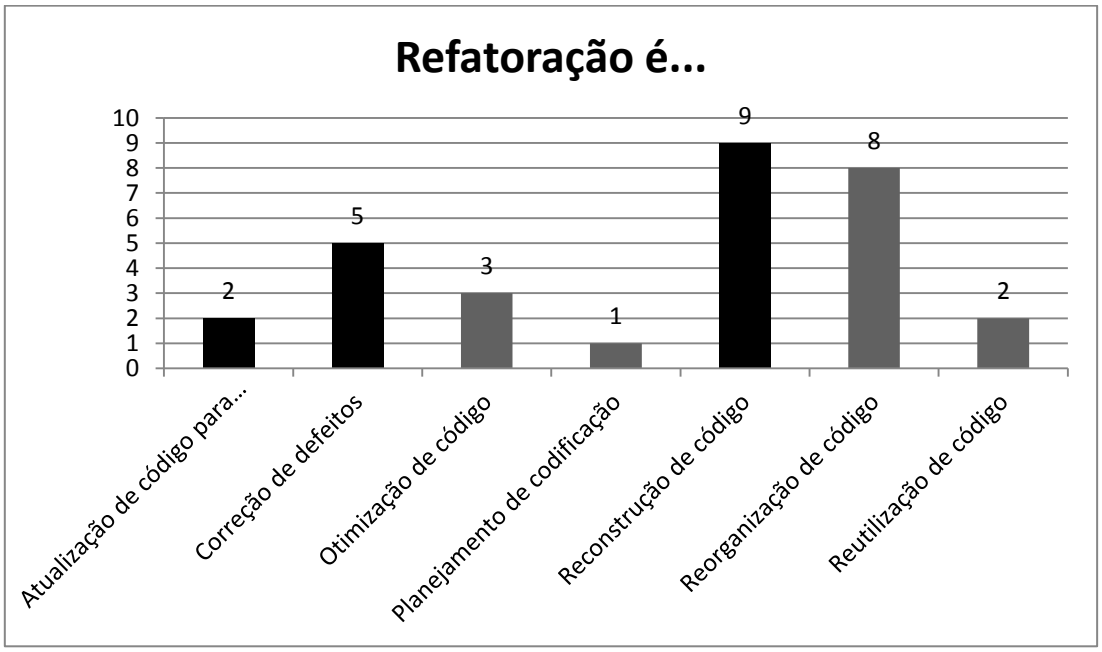


Figura 2.4 – Códigos para o grupo de respostas “refatoração é...”

As respostas de definição de refatoração de código contemplaram não somente a definição em si, mas também algumas informações adicionais que vimos

como interessante de serem capturadas e analisadas e que estão relacionadas com restrições para as atividades de refatoração e com os prováveis impactos que a refatoração pode ocasionar no código fonte. Para a definição em si, fato importante de ser observado é a quantidade de respostas definindo refatoração como sendo reconstrução de código fonte, ou seja, retrabalho puro. Outras definições de destaque são as que relacionam refatoração de código com correção de defeitos e ainda atualização de código para inserção de requisitos. Essas respostas foram os primeiros indícios que identificamos de que os programadores não estavam falando da mesma prática e de que o que provavelmente estava ocorrendo na organização não era refatoração e sim reconstrução de código fonte. Vale destacar que de uma mesma resposta surgiram mais de um código, por isso a soma total dos códigos não é exatamente o número de respostas (20).

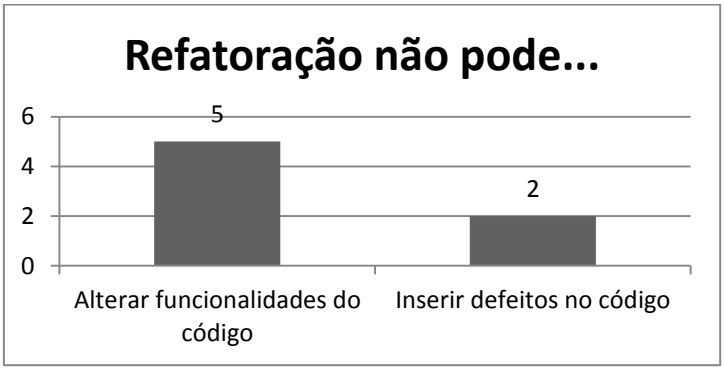


Figura 2.5 – Códigos para o grupo de respostas “refatoração não pode...”

Alguns programadores mencionaram ainda que a refatoração não pode acarretar em alterações de funcionalidades do código ou ainda inserir defeitos (ver Figura 2.5). Restrições que vão ao encontro da definição de refatoração apresentada em (FOWLER *et al.*, 1999) que preconiza a não alteração do comportamento externo, porém que estão distantes do que profissionais da prática de outras organizações de desenvolvimento de software relatam (KIM, ZIMMERMANN e NAGAPPAN, 2012).

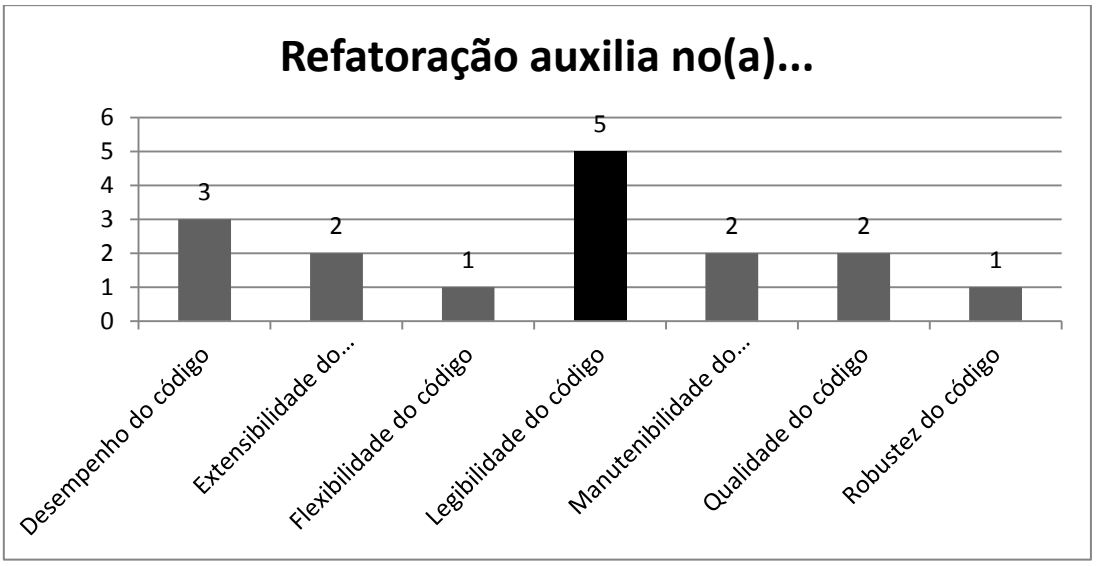


Figura 2.6 – Códigos para o grupo de respostas “refatoração auxilia no(a)...”

Outro conjunto de respostas dos programadores a respeito da refatoração foi relacionado com os benefícios que ela poderia trazer para o código fonte. Para os programadores, a refatoração de código impacta diretamente características de qualidade como legibilidade, desempenho, extensibilidade e manutenibilidade (ver Figura 2.6). Fato interessante nessas respostas é a quantidade expressiva de programadores que indicaram legibilidade como sendo uma característica impactada pela refatoração de código, pode ser mais um indício de que eles veem a refatoração como sendo uma atividade de melhoria pontual no código fonte.

Através dos dados obtidos com o *survey*, conseguimos perceber que embora existisse de fato refatoração de código na organização, a atividade mais frequente era a de reconstrução de código fonte. Além disso, percebemos que havia uma dissonância entre as percepções sobre refatoração de código na organização e muitos programadores entendiam ainda que refatorar o código era simplesmente modificar o código, seja para corrigir problemas ou inserir funcionalidades. Pudemos confirmar esses resultados encontrados através da análise de outras respostas ao formulário de refatoração, como as respostas obtidas com a questão sobre a frequência com que os programadores refatoravam o código dos projetos que participavam. A maioria dos programadores disse que raramente eles realizavam tarefas de refatoração – como possível de ser visualizado na Figura 2.7 –, porém houve aqueles que responderam diariamente, semanalmente e mensalmente, frequências impensáveis para atividades que, de certa forma, trazem riscos e custos aos projetos.

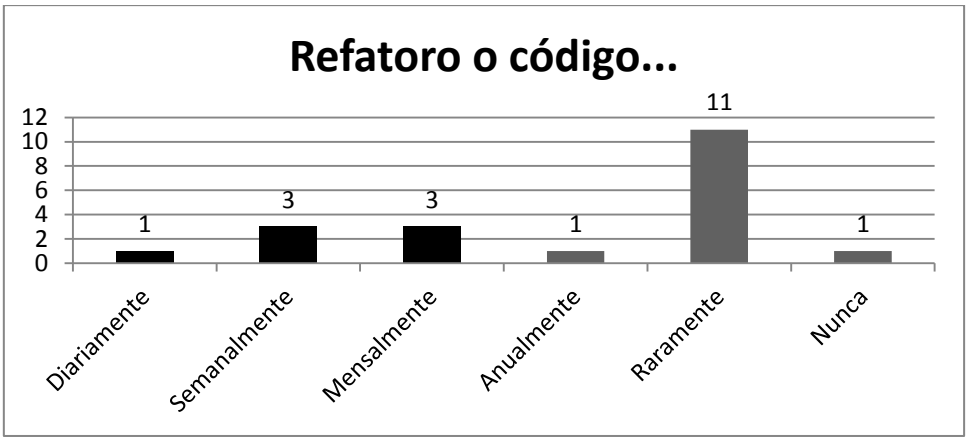


Figura 2.7 – Frequência de realização de atividades de refatoração de código fonte

Outro conjunto de respostas que trouxe ainda mais respaldo às conclusões alcançadas até então foi o de 10 respondentes que indicaram que já tinham interrompido atividades de refatoração de código iniciadas. Dentre os motivos para a refatoração, alguns programadores indicaram que queriam atualizar funcionalidades e bibliotecas de desenvolvimento. Foco para um respondente que indicou ter realizado a refatoração para melhorar a compreensibilidade do código fonte, melhoria que entendemos poder ser pontual.

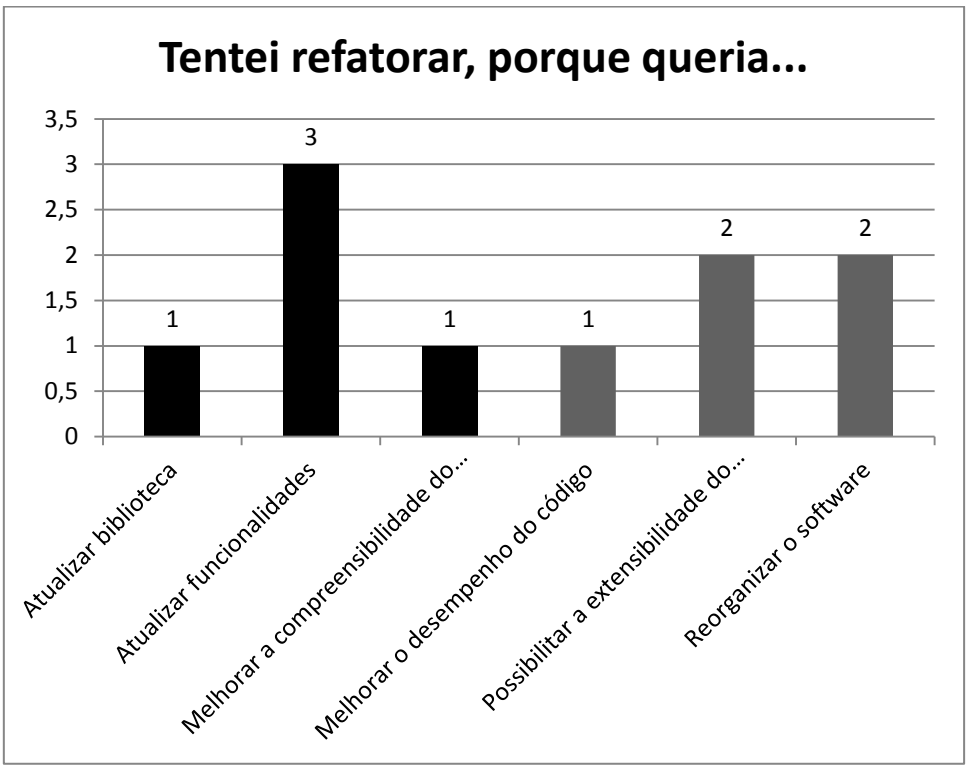


Figura 2.8 – Códigos extraídos das respostas sobre experiências com atividades de refatoração que foram interrompidas

Dentre as situações que mais influenciavam os programadores a realizarem tarefas de “refatoração” de código estavam as relacionadas com mudanças pontuais

no código fonte, como a existência de variáveis, constantes, métodos, funções, procedimentos e/ou parâmetros desnecessários, além de problemas na legibilidade de seus identificadores. Quando analisamos essa listagem (ver Tabela 2.7) em conjunto com os problemas que mais ocorrem na organização, vemos que os problemas mais recorrentes não são necessariamente os mais tratados, possivelmente porque a prioridade dos problemas recorrentes é menor do que a dos problemas mais tratados; ou ainda porque os problemas mais tratados são vistos como mais rápidos e fáceis de serem resolvidos e causam menos efeitos colaterais ao código fonte.

Tabela 2.7 – Situações que mais influenciam a refatoração de código na empresa Alfa

% (valor agregado/máximo valor possível)	Situação que influencia a refatoração na empresa	Influencia na Refatoração...
68,42	Redundância de código.	SEMPRE
66,67	Variáveis, constantes, métodos, funções e/ou procedimentos desnecessários.	
64,91	Legibilidade deficiente, ou seja, variáveis, constantes, métodos, funções e/ou procedimentos com nomes não intuitivos ou localizados em lugares inapropriados.	
63,16	Métodos, funções e/ou procedimentos com muitos parâmetros desnecessários.	
59,65	Baixo desempenho da aplicação.	EVENTUALMENTE
57,89	Código de difícil manutenção.	
49,12	Código de difícil extensão de funcionalidades.	
45,61	Código de difícil reutilização.	RARAMENTE
45,61	Métodos, funções e/ou procedimentos com muitas responsabilidades.	
42,11	Dificuldade de testar o código sem antes aplicar refatoração.	
40,35	Código muito fragmentado.	
31,58	Falso Positivo (Código com alta complexidade.)	
22,81	Falso Positivo (Código muito extenso.)	NÃO INFLUENCIA
19,30	Existência de código legado com o qual é necessário trabalhar.	
-5,26	Falta de modelos de projeto/arquitetura que permitam entender o software.	
-19,30	Forte dependência com código de outros times de desenvolvimento.	
-64,91	Falso Positivo (Alto tempo de compilação do código.)	

2.4 Ameaças à Validade do Estudo

Em relação ao *survey* de um modo geral, algumas situações que ocorreram ao longo do planejamento e condução, e também posterior a essas fases, acabaram se mostrando como ameaças à validade do estudo. Uma delas foi em relação a cobertura das alternativas de respostas fornecidas nos questionários. Tentamos fornecer, sempre que possível, alternativas de respostas aos respondentes, até mesmo para auxiliar na agregação dos dados coletados, no entanto temos conhecimento que não listamos todas as respostas possíveis para as questões, até mesmo para não trazer exaustão aos respondentes. Contudo, esse fato pode ter influenciado os resultados que obtivemos visto que poucos programadores incluíram novos itens às listagens fornecidas.

Outra ameaça relaciona-se com a execução do estudo que ocorreu em datas e de formas distintas – duas presenciais e uma via correio eletrônico. Não sabemos quanto das informações do estudo foram repassadas aos respondentes que não puderam participar da primeira execução. Também não tivemos controle sobre a execução do estudo via correio eletrônico. Embora tenhamos fornecido os detalhes sobre os procedimentos para entrega dos formulários ao responsável por executar o *survey* com os demais respondentes, não sabemos se eles foram seguidos de forma adequada.

Em relação à análise, optamos por realizar uma análise de toda organização em detrimento a análise por setor de desenvolvimento. Esse fato pode ter influenciado um ou outro resultado obtido, fazendo com que os dados de um setor sobressaíssem em relação aos de outro setor, contudo percebemos que, de modo geral, a análise do todo era mais coerente devido às mudanças que estavam ocorrendo na organização relacionadas com a rotatividade de desenvolvedores. Em todo caso, pudemos perceber que as respostas para as questões principais, como as de definição de refatoração; as de classificação das situações que levam os programadores a refatorarem o código; e as de priorização das características de qualidade em código fonte não foram influenciadas por nenhum setor de desenvolvimento específico da organização.

2.5 Considerações sobre o Capítulo

Ao longo deste capítulo apresentamos uma visão mais detalhada sobre a empresa Alfa e suas características mais propensas a influenciar atividades de refatoração e reconstrução de código fonte; como o reaproveitamento de código em diferentes produtos e a rotatividade de programadores nos diferentes setores da

organização. A partir da análise dos dados coletados com a condução do *survey* na empresa, vimos que o conhecimento a respeito de refatoração de código fonte na organização não era homogêneo e que muitas atividades de reconstrução de código estavam ocorrendo, uma vez que as situações que levavam os programadores a "refatorar" os códigos dos projetos eram pontuais e representavam modificações para atender perspectivas pessoais de qualidade em código, como a legibilidade.

Com o *survey*, identificamos ainda as características de qualidade destacadas como as mais importantes para a organização. Vimos que as principais características de qualidade, de alguma forma, já estavam sendo trabalhadas com os desenvolvedores, através de atividades de melhoria em requisitos e testes de software do projeto de cooperação ESE-Empresa Alfa. Outras características de qualidade vistas também como importantes pela organização eram a adequação do código fonte a padrões previamente estabelecidos e a compreensibilidade do código fonte.

Outra observação realizada com a análise dos dados foi que, apesar da legibilidade em código fonte não ter sido identificada como característica de qualidade prioritária, muitos programadores estavam realizando reconstrução de código fonte para adequar a legibilidade do código ao seu padrão pessoal de qualidade. Durante a exposição dos resultados do *survey* para os respondentes do estudo, pudemos confirmar esse fato e conseguimos ainda capturar relatos de situações que indicavam que um mesmo código já tinha sido reconstruído diversas vezes por programadores diferentes com o propósito de adequá-lo ao padrão de legibilidade do programador.

Diante dessas análises, concluímos que as ações de qualidade em código fonte a serem tomadas inicialmente na empresa Alfa para diminuir os riscos de reconstrução de código fonte deveriam estar relacionadas com a homogeneização das perspectivas de qualidade em código fonte, principalmente relacionadas com a legibilidade de código. Além disso, como forma de agregar aos códigos da empresa, características de qualidades importantes para a organização, eram necessárias também ações que pudessem beneficiar a compreensibilidade e a conformidade com padrões dos códigos. De alguma forma, essas características se mostram como pré-requisitos para qualquer outra ação de melhoria em código fonte, uma vez que outras características de qualidade só podem ser alcançadas no código quando este é legível e compreensível para os programadores que o utilizam.

3 Busca Estruturada: Legibilidade e Compreensibilidade em Código Fonte

Neste capítulo o conceito de qualidade adotado ao longo do trabalho, bem como alguns trabalhos da literatura que visam auxiliar na leitura e compreensão de código fonte são apresentados; com destaque para os trabalhos resultantes da busca estrutura realizada para identificar atributos de código que pudessem agregar legibilidade e compreensibilidade ao código fonte.

3.1 Introdução

Como mencionado no capítulo anterior, identificamos que ações de melhoria em qualidade de código fonte na empresa Alfa deveriam estar relacionadas com práticas para homogeneização do entendimento sobre legibilidade e melhoria na compreensibilidade de código; e com padrões para conformidade do código fonte. Assim, identificamos como uma oportunidade de ação a formulação e implantação de diretrizes de codificação, com foco em legibilidade e compreensibilidade de código, nos códigos da organização, estabelecendo as seguintes hipóteses:

- *H1: O uso de diretrizes de codificação auxilia no alinhamento de diferentes perspectivas de qualidade de código fonte.*
- *H2: O uso de diretrizes de codificação melhora a qualidade do código fonte.*
- *H3: O uso de diretrizes de codificação diminui a incidência de reconstrução de código fonte nos projetos de manutenção da empresa Alfa.*

Para conseguir analisar essas hipóteses foi preciso primeiramente construir um conjunto de diretrizes que estivessem de fato relacionadas com legibilidade e compreensibilidade de código fonte. Optamos, então, pela realização de uma revisão bibliográfica inicial (*ad-hoc*) como forma de identificar os principais conceitos envolvidos na área de qualidade de código fonte, bem como os tipos de trabalhos existentes sobre qualidade de código, com foco em legibilidade e compreensibilidade. Posteriormente, evoluímos a pesquisa para realizar uma busca estruturada (inspirada em revisão sistemática da literatura) por atributos de código fonte que pudessem contribuir para a legibilidade e compreensibilidade de código e, assim, fornecer subsídios para a formulação de diretrizes de codificação para legibilidade e compreensibilidade baseadas em evidências científicas.

É importante enfatizar que embora seja possível encontrar disponível, em meio digital e em livros, diferentes padrões/estilos/guias de codificação para várias das linguagens de programação existentes na empresa Alfa, como o GCC *Coding Conventions*[6], o Google C++ *Style Guide*[7], o MISRA C++ *Guidelines*[8], o Java *Code Conventions*[9], Pear PHP *Coding Standards*[10], dentre outros, observa-se que os padrões identificados se mostram para propósitos genéricos, não possuem relação explícita com características de qualidade de código fonte e/ou não apresentam evidências científicas de sua importância para a melhoria da qualidade do código fonte. Assim, torna-se necessário garantir um conjunto mínimo de diretrizes de codificação que possa satisfazer as necessidades da organização ao mesmo tempo em que garanta a melhoria da qualidade do código fonte nas características de qualidade de código previamente identificadas. Desta forma, é imprescindível a análise da literatura técnica-científica como subsídio para a identificação desse conjunto de diretrizes, cujos detalhes da revisão bibliográfica inicial bem como do planejamento e resultados de uma busca estruturada são apresentados nas seções seguintes.

3.2 Revisão Bibliográfica Inicial – Qualidade de Código Fonte

Qualidade é um conceito naturalmente complexo, porque seu significado depende tanto da expectativa de diferentes pessoas como do contexto em que está sendo requerida (KITCHENHAM e PFLEEGER, 1996). Garvin explora cinco diferentes visões de qualidade descritas com base na percepção de qualidade de grupos distintos de pessoas (detalhes na Figura 3.1): i) visão transcendental, qualidade é algo que pode ser reconhecido, porém não definido; ii) visão de usuário, qualidade é adequação ao propósito; iii) visão de produção, qualidade é conformidade com especificação; iv) visão de produto, qualidade está relacionada com as características

---

[6] https://gcc.gnu.org/codingconventions.html

[7] http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

[8] http://www.misra-cpp.com/activities/misrac/tabid/171/default.aspx

[9] http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[10] http://pear.php.net/manual/en/standards.php

inerentes do produto; v) visão baseada em valor, qualidade depende do quanto o cliente está disposto a pagar por ela (GARVIN, 1984).

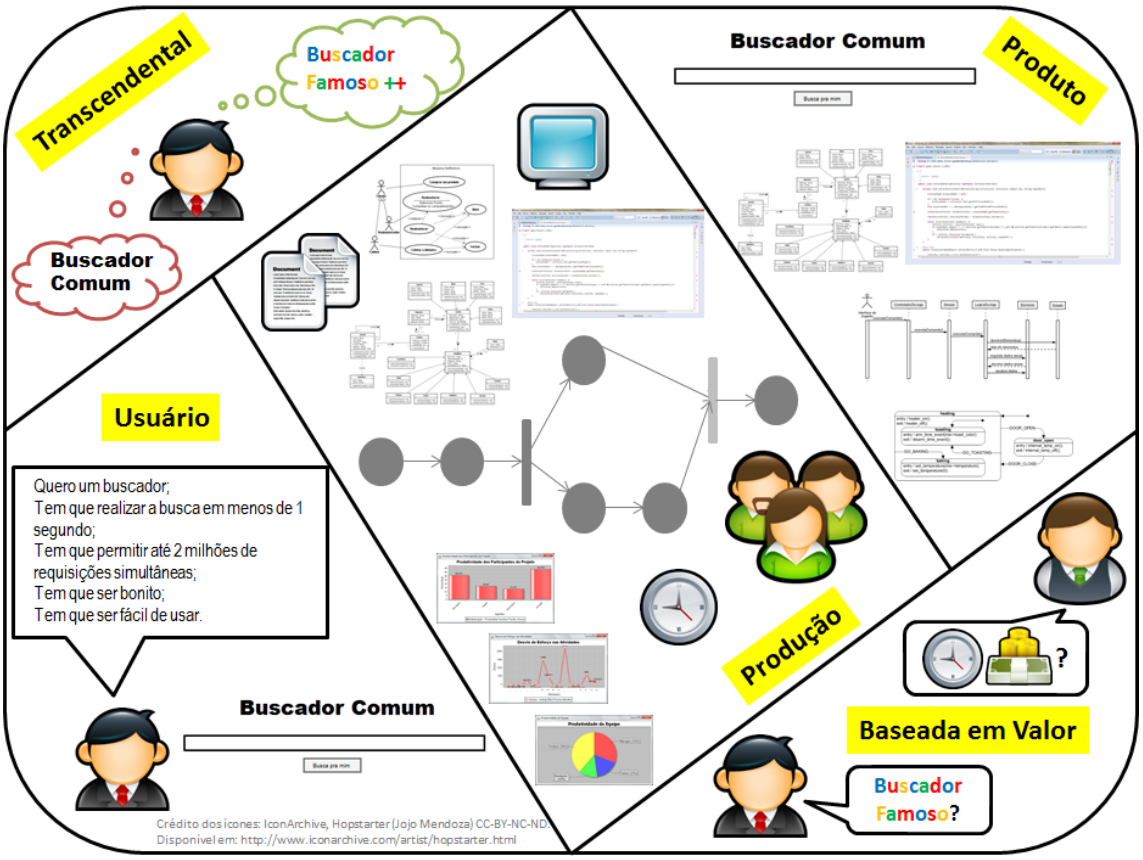


Figura 3.1 – As cinco visões de qualidade de (GARVIN, 1984)

A visão transcendental é uma visão idealizada daquilo que se deseja para o software, ela é então percebida somente quando vista (fato improvável de ocorrer por ser uma idealização), não sendo possível descrevê-la. A visão de usuário está relacionada com o que foi solicitado e o que foi entregue, isso envolve tanto funcionalidades quanto não funcionalidades. Comparada com as demais visões, na visão de usuário, é possível avaliar concretamente a qualidade do software, embora algumas não funcionalidades, como confiabilidade e segurança, necessitem de um maior tempo de avaliação para se ter alguma conclusão de qualidade.

A visão de produção nos remete ao processo de desenvolvimento como um todo e às diferentes estratégias de desenvolvimento adotadas de forma a sistematizar; controlar custo, prazo e defeitos; e, em alguns casos, até prever o desenvolvimento. Nesse sentido, modelos de capacidade e maturidade como CMMI-Dev (*Capability Maturity Model Integrated for Development*) (SEI, 2010) e o MR-MPS (Modelo de Referência de Melhoria do Processo de Software) (SOFTEX, 2012) estabelecem um conjunto de práticas a serem adotadas ao longo do desenvolvimento de forma a

garantir qualidade ao processo e apresentam evidência da relação entre a qualidade do processo e do produto final (TRAVASSOS e KALINOWSKI, 2014).

A visão de produto está relacionada com as características inerentes do produto. Enquanto as visões de usuário e de produção focam em aspectos externos ao software, a visão de produto lida com aspectos internos, como o cumprimento ao paradigma de programação adotado, a estruturação e modularidade do código e a conformidade com padrões de codificação previamente estabelecidos.

A visão baseada em valor é de alguma forma dependente das demais visões de usuário, produção e produto e dos prováveis desentendimentos que possam ter ocorrido ao longo do processo de desenvolvimento, já que pessoas diferentes valorizam características diferentes. Nesse caso, a negociação é um ponto chave para se alcançar ou mesmo aceitar a qualidade do produto.

3.2.1 Qualidade de Produto: Legibilidade e Compreensibilidade de Código

Neste trabalho, focamos na visão de qualidade de produto. É importante enfatizar que mesmo quando focamos em uma única visão de qualidade ainda é possível ter uma extensa lista de características de qualidade a serem observadas. A série de padrões SQuaRE (*Systems and software product Quality Requirements and Evaluation*) (ISO/IEC, 2011), evolução da ISO/IEC 9126 (ISO/IEC, 2001), propõe um conjunto de elementos de software possíveis de serem mensurados como forma de aferir a qualidade do software em diferentes características e sub-características de qualidade (ver Figura 3.2). A qualidade do software é dada então pela avaliação da capacidade do software em satisfazer necessidades estabelecidas e implícitas relacionadas com uma ou mais dessas características, podendo ser ainda avaliada internamente – quando os elementos mensuráveis estão relacionados com propriedades estáticas do software – e externamente – quando os elementos mensuráveis estão relacionados com comportamentos do sistema como um todo.

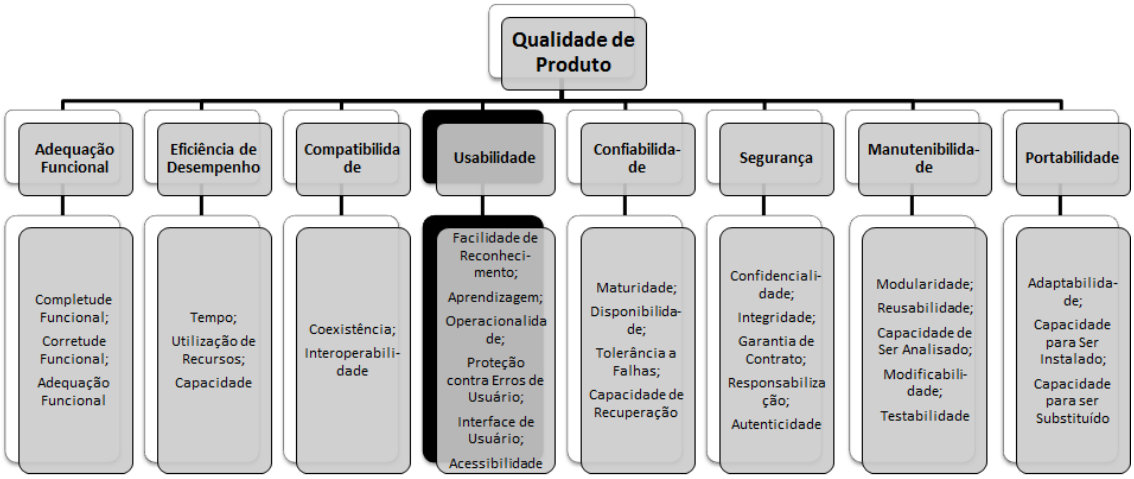


Figura 3.2 – Características e subcaracterísticas de qualidade conforme SQuaRE

A Figura 3.2 fornece uma visão geral das características (segundo nível) e subcaracterísticas (terceiro nível) de qualidade de produto. Para cada uma das subcaracterísticas de qualidade, a norma provê definição e estabelece ao menos uma medida – definida para avaliar a qualidade interna e/ou externa – possível de ser observada a partir de elementos mensuráveis de qualidade (QME – *Quality Measurement Element*). Este trabalho encontra-se no escopo das subcaracterísticas de usabilidade, focando para a qualidade interna do produto.

3.2.1.1 Legibilidade e Compreensibilidade de Código Fonte

A legibilidade e compreensibilidade é parte da característica *usability* da SQuaRE – destacada na Figura 3.2. Para a SQuaRE, a usabilidade é mensurada pelo grau com que usuários específicos (aqui também estão inseridos usuários do código fonte) conseguem usar o software para atender aos objetivos desejados. Medidas de usabilidade interna são utilizadas para avaliar a extensão com que um software consegue ser entendido, aprendido, operado e atrativo para o usuário do código fonte. Proporção de funções contendo descrições de uso; documentadas e compreendidas corretamente são alguns exemplos de medidas de usabilidade interna existentes na norma.

Tendo como base esse contexto de usabilidade e sabendo que algo que é legível é algo que pode ser lido; que é escrito em caracteres nítidos (PRIBERAM INFORMÁTICA, 2013); e que algo que é compreensível é algo que pode ser compreendido; inteligível (PRIBERAM INFORMÁTICA, 2013), entendemos que a legibilidade em código fonte está relacionada à apresentação e disposição dos elementos do programa para o programador, enquanto que a compreensibilidade está relacionada com o entendimento dos elementos do programa pelo programador.

Assim, é possível que um código seja legível e não compreensível e vice-versa, contudo a alta legibilidade pode facilitar a compreensão do código. A própria organização dos arquivos do código pode afetar sua legibilidade, já que a dificuldade de acesso a diferentes elementos do programa acarreta problemas na leitura do código fonte.

Destacamos duas diferentes dimensões que a compreensibilidade de código fonte pode ter: i) compreensibilidade de baixo nível de abstração; e ii) compreensibilidade de alto nível de abstração. Trabalhos que lidam com a compreensibilidade de baixo nível tratam de aspectos autocontidos do código fonte, como nomenclatura dos identificadores e comentários. Em contrapartida, trabalhos que lidam com compreensibilidade de alto nível tratam de aspectos mais estruturais do código.

3.2.2 Trabalhos na Área de Leitura e Compreensão de Código

Apesar de a série de normas SQuaRE apresentar um conjunto de medidas para aferir a qualidade do produto para diferentes características de qualidade, é precipitado acreditar que elas são únicas e suficientes para aferir a qualidade em software. É possível encontrar na literatura técnica alguns trabalhos que usam diferentes estratégias para avaliar ou obter qualidade em software. Especificamente para as características de legibilidade e compreensibilidade de código fonte, alguns trabalhos propõem diferentes estratégias para ou melhorar a legibilidade e compreensibilidade do código ou auxiliar a ler e compreender o código.

As subseções a seguir detalham alguns trabalhos e suas estratégias para melhorar ou auxiliar na leitura e compreensão do código fonte. A fim de melhor entender os diferentes tipos de trabalho na área, subdividimo-los em quatro categorias distintas, porém não únicas, que foram observadas ao longo da pesquisa bibliográfica.

3.2.2.1 Categoria A: Análise e Mudança Estrutural Objetivando Compreensibilidade do Código

Trabalhos na área de análise e mudança estrutural visam avaliar a aplicação de técnicas de refatoração de código no atendimento à compreensibilidade de código fonte. Ainda estão nesse grupo, trabalhos que utilizam técnicas de mineração de dados e inteligência artificial como forma de verificar padrões em código fonte que permitam identificar oportunidades de melhorias na estrutura do código, acreditando que isso irá provocar melhorias em compreensibilidade. É importante enfatizar que a compreensibilidade do código nesses trabalhos é vista em um alto nível de abstração, em outras palavras, não é especificamente o trecho de código que é avaliado e sim a

composição de vários trechos de código e, algumas vezes, a própria implementação da arquitetura do software.

O trabalho descrito em (ALSHAYEB, 2009) avalia quantitativamente o uso da aplicação de diferentes técnicas de refatoração de código – propostas anteriormente em (FOWLER *et al.*, 1999) – para atender a algumas características de qualidade como adaptabilidade, manutenibilidade, compreensibilidade, reusabilidade e testabilidade. Para tanto, o autor utiliza medidas de adaptabilidade, manutenibilidade, compreensibilidade e reusabilidade avaliadas em (DANDASHI, 2002) e de testabilidade avaliadas em (BRUNTINK e DEURSEN, 2006) como forma de verificar se a aplicação da refatoração produz algum efeito positivo sobre a qualidade do código. Embora o autor tenha observado alguns efeitos positivos na qualidade dos software avaliados em relação às características em questão, ele também observou efeitos negativos na qualidade, o que não o permitiu concluir sobre os impactos reais das técnicas de refatoração.

Outro trabalho nessa linha é o apresentado em (BOIS *et al.*, 2006) Os autores conduziram um estudo com alunos de ciência da computação para avaliar o impacto das técnicas de refatoração de código para decomposição de "*god* class" na compreensibilidade do código. A compreensibilidade do código nesse caso foi aferida a partir da análise da habilidade dos participantes do estudo em mapear informações do domínio do problema para a solução de código fornecida. Os autores observaram ainda que algumas técnicas de decomposição de código levaram os participantes a terem maior acurácia do que outras na identificação de informações do domínio.

Ainda com foco em compreensibilidade estrutural do código está o trabalho de Kreimer (KREIMER, 2005) que utiliza técnicas de mineração de dados e inteligência artificial para identificar padrões de problemas em código fonte, como "*design flaws*" e "*bad smells*" além de fornecer apoio adicional a atividades de inspeção de código. No caso específico de Kreimer, é argumentado que muitos problemas existentes em código impactam a compreensibilidade do código; e que é possível identificar esses problemas a partir da combinação de medidas – como tamanho, complexidade, acoplamento e coesão – com técnicas de aprendizado de máquina. A ideia base do trabalho é modelar um conjunto de situações-problema em código, associando-as a diferentes medidas aplicáveis a código fonte. Desvios nessas medidas são então indicativos de que uma situação-problema está ocorrendo. Esse trabalho não tem como objetivo melhorar a compreensibilidade do código, mas sim fornecer instrumentos para identificar quando a compreensibilidade do código não está adequada.

3.2.2.2 Categoria B: Extração de Conhecimento do Código como Apoio à Compreensão do Software

Enquanto os trabalhos expostos na categoria A lidam com estratégias para agregar ou tentar avaliar a compreensibilidade do código fonte, existe outro grupo de trabalhos que visa utilizar o código como forma de extrair informações e auxiliar na compreensão do software como um todo. Nesse caso, o objetivo final não é somente avaliar a qualidade do software, mas sim obter conhecimento como forma de derivar e/ou mapear conceitos sobre o domínio do problema; e ainda extrair ou evoluir estilos de programação já existentes.

Os trabalhos propostos em (ANTONIOL *et al.*, 2002), (GUERROUJ, 2010) e (GUERROUJ, 2013) são exemplos de abordagens que objetivam mapear informações de código fonte para documentos de projeto ou conceitos do domínio do problema. A ideia base é fornecer uma fonte de informação a mais para auxiliar o desenvolvedor a compreender o software, visto que muitas vezes o código – com pouca documentação e insuficiência de comentários – não é suficiente para permitir uma compreensão adequada. Além disso, Guerrouj em (GUERROUJ, 2013) acredita que esse mapeamento pode ainda auxiliar na evolução do vocabulário utilizado no código fonte e também da compreensibilidade do código, uma vez que viabiliza a transferência de informação da documentação do domínio para o código.

Outro tipo de proposta ainda nessa categoria de trabalho é apresentado por (CORBO, GROSSO e PENTA, 2007). Os autores propõem uma ferramenta para descoberta de estilos de programação de código fonte. A ideia é auxiliar os programadores iniciantes no projeto a conhecer o estilo de programação utilizado no código e alertá-los quando esse estilo estiver sendo quebrado.

3.2.2.3 Categoria C: Visualização de Software como Apoio à Compreensão do Código

Trabalhos dessa categoria visam exclusivamente auxiliar programadores na compreensão do código por meio visual. Desta forma, não pretendem fornecer artifícios para melhorar a compreensibilidade do código em si. Incluem-se nessa categoria trabalhos que avaliam a utilização de diagramas conhecidos – *control structure diagram* (CSD), por exemplo – e também de visualizações inovadoras combinadas com o código fonte para prover informação adicional ao programador e assim auxiliá-lo a compreender o código.

Em (HENDRIX *et al.*, 2000), os autores avaliaram através de um experimento o benefício do uso de diagramas de estrutura de controle para a compreensão do código pelo programador, avaliada com base nas respostas que os participantes forneceram

para perguntas relacionadas com o código fonte. O resultado do estudo mostrou que o grupo de participantes que avaliou o código em conjunto com o diagrama teve menor tempo de resposta e melhor acurácia que o grupo de participantes que avaliou somente o código. Umphress e Endrix (UMPHRESS *et al.*, 2006), além de avaliarem o uso de CSD, também avaliaram o uso do *complexity profile graph* (CPG) para auxiliar na compreensão e mensurar a compreensibilidade do código. O uso do grafo de análise de complexidade mostrou ter correlação com o tempo para entender corretamente o código fonte, já que ele permitiu que o programador identificasse partes do código vistas como mais complexas e que iriam requerer maior atenção na análise.

Na linha de trabalhos que utilizam visualizações inovadoras para auxiliar a compreensão do código estão trabalhos de computação gráfica como o (RILLING e MUDUR, 2005) que tem a engenharia reversa como sua principal motivação. O trabalho utiliza técnicas de renderização 3D para prover visualização da estrutura do código fonte, enfatizando características de complexidade e acoplamento entre componentes, o que os autores acreditam que melhora a percepção de compreensão sobre o código.

Outro trabalho que também utiliza a visualização para apoiar a compreensão de código, mais especificamente a evolução de funcionalidades (*features*), é o apresentado em (NOVAIS *et al.*, 2012). Os autores apresentam uma estratégia dita como proativa que utiliza um conjunto de heurísticas para analisar a evolução da aplicação e sugerir elementos de implementação que estejam trabalhando com determinada funcionalidade. Visualizações como *treemap*, *polymetric view* e *interactive directed graphs* são utilizadas como instrumentos de apoio à utilização da estratégia à compreensão da evolução das funcionalidades.

3.2.2.4 Categoria D: Utilização de Cores como Apoio à Leitura e Compreensão do Código

Por fim, existe outro grupo de trabalhos que utiliza recursos também visuais – geralmente associados a IDEs de programação – para destacar características importantes no código fonte. A compreensão do código nesses trabalhos – diferente do que o que ocorre nas demais categorias – é vista em um baixo nível de abstração. O trabalho de Rambally em (RAMBALLY, 1986) destaca que existem muitos aspectos que influenciam a leitura e compreensão do código fonte, como nomeação de identificadores, documentação interna, modularidade e formatação do código. Focando na formatação, o autor utiliza combinação de cores para destacar estruturas de controle; de decisão; e tipos de instruções específicas, como declarações e entrada

e saída de dados. Um experimento foi conduzido como forma de avaliar o impacto de uso de cores na leitura e compreensão do código e como resultado foi observado que existe uma relação entre eles, indicando maior acurácia nas respostas sobre o código por participantes que avaliaram o código fonte coloridos do que por aqueles que avaliaram o código em suas cores usuais.

Outro trabalho nessa linha é o apresentado em (FEIGENSPAN *et al.*, 2011). A diferença nesse caso é que o que é colorido não é o código em si e sim o plano de fundo de trechos de código. O trabalho é da área de linha de produto de software e, no caso, os autores utilizam as cores para diferenciar trechos de código referentes a diferentes características da linha de produto. Para a realização da coloração os autores propuseram uma ferramenta que também fornece visualização de outras informações especificas de linhas de produto de software. Como avaliação da proposta, foram passadas tarefas de compreensão – identificar arquivos de determinada característica, por exemplo – e manutenção – localizar/corrigir um defeito no código – a programadores. O resultado foi uma melhoria na compreensão do código, avaliada com base na corretude e tempo de execução das tarefas, por programadores que utilizaram códigos coloridos.

3.3 Busca Estruturada por Atributos de Código para Legibilidade e Compreensibilidade de Código Fonte

Diante do cenário de trabalhos na área de legibilidade e compreensibilidade identificado na revisão bibliográfica inicial, vimos que a categoria de trabalhos mais relacionada com o que esperávamos para apoiar a construção de diretrizes de codificação para legibilidade e compreensibilidade de código fonte era a **Categoria E**. A categoria em questão contém artigos que avaliam a legibilidade e compreensibilidade de códigos fonte com base em atributos específicos do código (atributos de mais baixo nível de compreensibilidade), como a existência de comentários; indentações e linhas em branco para separação de trechos de códigos; o tamanho da linha e dos identificadores; dentre outros atributos.

Nessa categoria, estão trabalhos como os de (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (POSNETT, HINDLE e DEVANBU, 2011) que utilizam a mineração de dados para identificar atributos de código fonte que influenciam a legibilidade e compreensibilidade de códigos avaliados subjetivamente como legíveis e compreensíveis por estudantes e programadores. Outros tipos de trabalhos dessa categoria são os de (LAWRIE *et al.*, 2006) e (LAWRIE *et al.*, 2007) que avaliam que

tipo de identificadores (contendo palavras completas, abreviações ou caractere simples) acarreta em melhor entendimento do código por programadores.

Para identificar outros trabalhos nessa categoria e, assim, agregar maior evidência sobre atributos de código fonte que possam influenciar a legibilidade e compreensibilidade de código, uma busca estruturada baseada em revisão sistemática da literatura foi conduzida e tem seu planejamento e resultados apresentados nas subseções a seguir.

3.3.1 Revisão Sistemática da Literatura

A revisão sistemática da literatura é uma revisão que tem como base uma estratégia de busca previamente definida, planejada e documentada, visando abranger o máximo de artigos relevantes, ou seja, que atendem ao objetivo de pesquisa definido. A sistematização e documentação do processo possibilita também a reaplicação da busca para audição ou mesmo para continuação da pesquisa para outros períodos não considerados anteriormente.

O processo para realização de uma revisão sistemática pode ser dividido em quatro fases, são elas (BIOLCHINI *et al.*, 2007): i) planejamento da revisão; ii) execução da revisão; iii) análise dos resultados; e iv) empacotamento (ver Figura 3.3).

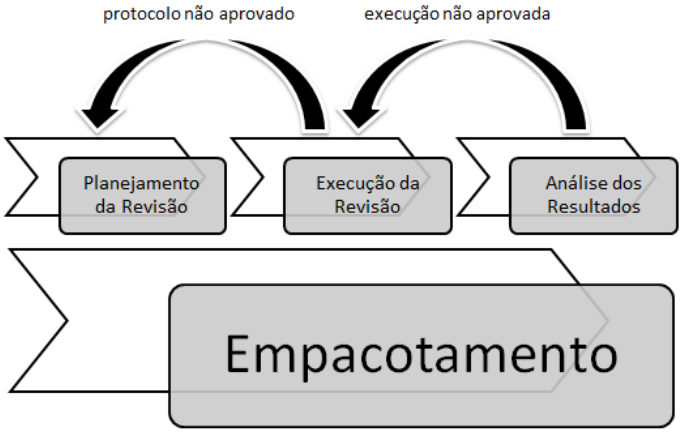


Figura 3.3 – Processo para realização de revisão sistemática da literatura contendo quatro fases. Adaptado de (BIOLCHINI *et al.*, 2007)

A fase de **planejamento** contempla atividades responsáveis por identificar as questões de pesquisa a serem respondidas a partir do estudo e também por formular e avaliar o protocolo da revisão sistemática que guiará todas as demais atividades do processo (MAFRA e TRAVASSOS, 2006). É nessa fase que a *string* de busca é formulada seguindo a abordagem PICO de (PAI *et al.*, 2004) – ***P**opulation of interest*, *evaluated **I**ntervention*, *Intervention **C**omparison* e *expected **O**utcome*. Nessa fase são importantes ainda que sejam detalhados os métodos de filtro dos artigos retornados

das bases de busca, bem como os critérios de inclusão e exclusão dos artigos como forma de possibilitar a auditoria da revisão.

A fase de **execução** engloba atividades de execução de fato do protocolo nas quais há: i) a aplicação da *string* de busca nas bases de indexação dos artigos; e ii) o filtro dos artigos retornados com base, geralmente, em uma leitura inicial de título, resumo e palavras-chave e uma avaliação dessas informações frente aos critérios de inclusão e exclusão de artigos. A fase de **análise** compreende a extração e a sintetização das informações dos artigos que passaram pelos critérios de inclusão como forma de atender aos objetivos identificados no protocolo da revisão. Por fim, a fase de **empacotamento** agrega atividades para empacotar todas as informações geradas com a realização da revisão sistemática que vai desde a formulação do protocolo até o relato e apresentação dos resultados encontrados nos artigos.

Uma variação da revisão sistemática da literatura é a *quasi*-revisão sistemática da literatura proposta em (TRAVASSOS *et al.*, 2008), utilizada quando não se tem dados de comparação (*intervention comparison*) para realização das buscas. Nesse trabalho, utilizamos uma terceira estratégia de revisão da literatura, que denominamos busca estruturada. Todo o procedimento da revisão sistemática da literatura foi utilizado, contudo restringimos o uso de bases de busca e simplificamos também o uso da abordagem PICO para a construção da *string* de busca, como é possível de ser observado na seção a seguir.

3.3.2 Planejamento da Busca Estruturada

3.3.2.1 Objetivo da Busca Estruturada

Seguindo a abordagem GQM, o objetivo da busca estruturada é como segue:

- ***Analisar*** *atributos de código fonte* ***com o propósito de*** *caracterizar* ***com respeito ao*** *impacto (negativo ou positivo) na legibilidade e/ou compreensibilidade de código fonte* ***do ponto de vista de*** *programadores de software* ***no contexto de*** *desenvolvimento de software.*

É importante destacar que legibilidade e compreensibilidade são termos que se confundem na literatura técnica. Embora tenhamos definido a nossa perspectiva para essas características de qualidade, vimos como mais adequado não realizar essa separação no protocolo da busca estruturada uma vez que isso dificultaria a seleção e análise dos artigos. Em relação à compreensibilidade de código fonte, optamos por, nesse primeiro momento, tratar atributos de mais baixo nível de abstração até mesmo para não misturar trabalhos de legibilidade com trabalhos de compreensibilidade mais estrutural (abstrações diferentes).

3.3.2.2 Questões de Pesquisa

Como forma de identificar os atributos de código fonte que tem influência sobre a legibilidade e/ou compreensibilidade, bem como sua forma de mediação (quando aplicável), estruturamos as seguintes questões a serem respondidas com a busca estruturada:

Questão Principal:

- *Quais atributos têm sido utilizados para avaliar a legibilidade e/ou a compreensibilidade de baixo nível de abstração do código fonte?*

Questões Secundárias:

- *Como os atributos identificados podem ser mensurados?*
- *Qual a relação de impacto existente entre os atributos identificados e a característica de qualidade avaliada (impacto negativo ou positivo)?*

3.3.2.3 Estratégia de Busca e Artigos de Controle

Identificamos a revisão realizada como busca estruturada, pois simplificamos a estrutura PICO para elaboração da *string* de busca, focando na intervenção e no resultado. Como essa busca estruturada está inserida num contexto de um projeto com a indústria, algumas restrições de tempo para execução do estudo limitam o rigor no planejamento da revisão. Além dessa simplificação da *string* de busca, limitamos a base de busca dos artigos, sendo selecionada somente a Scopus.

Base de Busca: Scopus

***String* de Busca:** TITLE-ABS-KEY((metric OR measure OR attribute OR predictor OR evaluation OR assessment OR improvement OR style OR standard OR pattern) AND (readability OR comprehensibility OR understandability OR identifier OR naming OR comment) AND ("software quality" OR "software readability" OR "software comprehension" OR "software understanding" OR "program quality" OR "program readability" OR "program comprehension" OR "program understanding" OR "code quality" OR "code readability" OR "code comprehension" OR "code understanding"))

Critérios de Inclusão dos Artigos:

- (I1) Relacionado com qualidade de código fonte em relação à legibilidade e/ou compreensibilidade de baixo nível do código; **AND**
- (I2) Apresentando atributos que avaliem a qualidade ou falta de qualidade do código fonte em relação à legibilidade e/ou compreensibilidade de baixo nível do código; **AND**
- (I3) Apresentando avaliação dos atributos de qualidade identificados frente aos impactos que estes causam na legibilidade e/ou compreensibilidade de baixo nível do código.

Critério de Exclusão de Artigos:

- (E1) Fora da área de computação; **OR**
- (E2) Tendo o seu foco de pesquisa em outras áreas da computação que não seja a de Engenharia de Software (ex.: Banco de Dados, Computação Gráfica, Inteligência Artificial); **OR**
- (E3) Tendo o seu foco de pesquisa em outras áreas da Engenharia de Software que não seja a relacionada com a qualidade do código fonte (ex.: Gerência, Requisitos, Educação); **OR**
- (E4) Não apresentando atributos de qualidade que avaliem a qualidade do código fonte em relação à legibilidade ou compreensibilidade de baixo nível do código; **OR**
- (E5) Indisponível; **OR**
- (E6) Não escrito em um dos seguintes idiomas: inglês, português e espanhol; **OR**
- (E7) Não apresentando nenhuma avaliação dos atributos de qualidade identificados frente aos impactos que estes causam na legibilidade ou compreensibilidade de baixo nível do código.

Objetivamos, com os critérios de exclusão, limitar os artigos selecionados para aqueles que compreendessem a **Categoria E**. Artigos das demais categorias, apesar de importantes para compor o conhecimento na área de legibilidade e compreensibilidade de código, não irão responder às questões de pesquisa levantadas.

Em relação ao processo de seleção de artigos: após a execução da *string* de busca na Scopus, cada artigo retornado deve ser avaliado pela pesquisadora de acordo com os critérios de inclusão e exclusão de artigos. Artigos que possam levar à duvida em relação a sua pertinência para o estudo devem ser incluídos para leitura completa.

Artigos de Controle:

- BUSE, R. P. L.; WEIMER, W. R. **A Metric for Software Readability**. Proceedings of the 2008 International Symposium on Software Testing and Analysis. New York: ACM Press. 2008. p. 121-130. DOI 10.1145/1390630.1390647.
- BUSE, R. P. L.; WEIMER, W. R. **Learning a Metric for Code Readability**. IEEE Transactions on Software Engineering, 36, n. 4, 2010. 546-558. DOI 10.1109/TSE.2009.70.

- LAWRIE, D. et al. **What's in a Name? A Study of Identifiers**. Proceedings of the 14th IEEE International Conference on Program Comprehension. Athens: IEEE. 2006. p. 3-12. DOI 10.1109/ICPC.2006.51.
- LAWRIE, D. et al. **Effective Identifier Names for Comprehension and Memory. Innovations in Systems and Software Engineering**, London, 3, n. 4, 2007. 303-318. DOI 10.1007/s11334-007-0031-2.
- POSNETT, D.; HINDLE, A.; DEVANBU, P. A **Simpler Model of Software Readability**. Proceedings of the 8th Working Conference on Mining Software Repositories. New York: ACM Press. 2011. p. 73-82. DOI 10.1145/1985441.1985454.

3.3.2.4 Campos de Extração e Avaliação da Qualidade dos Artigos

Os campos de extração foram selecionados tão somente para capturar os dados de resposta das questões de pesquisa (atributos de código e procedimentos de mensuração) e de avaliação da qualidade dos artigos selecionados. Imaginamos que existe a possibilidade de conflito entre artigos a respeito do impacto na legibilidade e/ou compreensibilidade que um atributo de código possa ocasionar, assim, a avaliação da qualidade do artigo irá auxiliar na definição sobre o impacto final do atributo em questão.

Campos de Extração:

- Descrição da característica de qualidade em questão dada pelo(s) autor(es);
- Atributos de código, suas respectivas descrições, procedimentos de mensuração e escala;
- Tipo de avaliação empírica da qualidade de código fonte avaliada com base nos atributos;
- Descrição geral da avaliação;
- Características dos códigos avaliados (linguagem de programação, aplicação real ou acadêmica dentre outras);
- Características dos participantes da avaliação (profissionais ou acadêmicos), se aplicável;
- Resultados da avaliação – impacto dos atributos na característica de qualidade avaliada.

Critérios para Avaliação da Qualidade dos Artigos:

- A - Existe alguma definição para os atributos de qualidade apresentados ou eles são intuitivos? (número total de atributos que possuem definição ou são intuitivos / número total de atributos)

- B - Existe alguma descrição sobre como os atributos de qualidade podem ser mensurados? (número total de atributos com procedimentos de mensuração/ número total de atributos)
- C - Os procedimentos de mensuração são descritos de forma a serem possíveis de reproduzir? (0 – não, nenhum; 0,25 – a maior parte não; 0,5 – a metade sim 0,75 – a maior parte sim; ou 1 – sim, todos)
- D - Existe algum resultado empírico relacionado com a avaliação da qualidade do código fonte frente aos atributos identificados? (2 – prova de conceito; 3 – *survey* ou estudo de observação; 4 – estudo de caso ou experimental; 5 – experimento controlado)
- E - Os códigos fonte avaliados são reais ou acadêmicos? (0 – acadêmicos ou *toy* ; 1 – reais)
- F – Os participantes do estudo são profissionais ou acadêmicos? (0 – acadêmicos ou não relacionados com a computação; 1 – profissionais da área da computação)

3.3.3 Execução da Busca Estruturada

A execução final da busca estruturada ocorreu em 08 de Janeiro de 2014 na qual foram retornados 263 artigos. O resultado da leitura do título e *abstract* dos artigos e da avaliação dos artigos frente aos critérios de inclusão e exclusão é apresentado na Tabela 3.1 a seguir.

Tabela 3.1 – Estados dos artigos após leitura de título e *abstract*

Excluídos	221
Duplicados	1
Dúvida	18
Incluídos	18
Controle	5
Total	**263**

Apesar da *string* de busca ter sido executada em apenas uma base, inesperadamente foi identificada uma duplicata. A grande quantidade de artigos marcados inicialmente como dúvida deveu-se ao fato de não ser possível inferir, somente pelo título e *abstract*, se os atributos de código mencionados foram avaliados em relação a suas influências na legibilidade e/ou compreensibilidade de código, especificamente. Posteriormente, com a leitura dos artigos de fato – todos os artigos em dúvida tinham de ser lidos por completo –, identificamos que boa parte dos artigos

até mencionavam atributos de código que poderiam impactar na legibilidade e/ou compreensibilidade de código, contudo a avaliação realizada era sob o impacto deles em outras características de qualidade que não as do objetivo do estudo, sendo eliminados pelo critério E4. O resultado final, após a leitura dos artigos é como segue.

Tabela 3.2 – Estados dos artigos após leitura completa

Excluídos	241
Duplicados	1
Dúvida	1
Incluídos	15
Controle	5
Total	**263**

O artigo que ainda ficou no estado de dúvida não foi possível de ser encontrado. 2 artigos incluídos também não estavam disponíveis na web, sobrando no final 13 artigos para extração de dados, além dos 5 artigos de controle identificados. Os dados extraídos dos artigos podem ser visualizados no APÊNDICE B – Informações Extraídas dos Artigos Incluídos na Busca Estruturada – desta dissertação.

3.3.4 Agregação dos Resultados Obtidos – Atributos para Legibilidade e Compreensibilidade de Código Fonte

Ao todo foram identificados 85 atributos de código fonte – incluindo os repetidos – dentre os quais se encontram atributos possíveis de serem coletados quantitativamente e outros cuja análise é qualitativa. É importante destacar que nem todos os atributos identificados foram levados em consideração na agregação final dos resultados, haja vista que o objetivo inicial da revisão era analisar somente atributos que de fato influenciassem (ou negativamente ou positivamente) a legibilidade e/ou compreensibilidade de código fonte. Dessa forma, somente atributos com evidência de impacto nessas características foram considerados. As subseções a seguir fornecem maiores detalhes dos 23 atributos qualitativos e dos 36 atributos quantitativos – total de 59 –, já removendo os repetidos, considerados para a formulação das diretrizes de codificação para a empresa Alfa.

3.3.4.1 Respostas às Questões de Pesquisa – Atributos Qualitativos

Os 23 atributos qualitativos representam características do código fonte que podem apenas serem observadas de forma subjetiva. Estão relacionados com

identificadores de variáveis, comentários de código, indentação e organização do código.

3.3.4.1.1 Atributos Qualitativos para Identificadores

Os atributos de identificadores dizem respeito tanto a sua forma de escrita – na qual pode ser utilizado algum estilo específico para separação de palavras, como o *camel case* e o *underscore*; ou ainda podem ser utilizadas palavras completas, abreviações e caractere simples para a sua formação – quanto ao seu conteúdo – identificadores mnemônicos ou não relacionados com o código. As tabelas Tabela 3.3,

Tabela 3.4 e Tabela 3.5 sumarizam os resultados obtidos com a análise dos artigos selecionados para esse grupo.

Tabela 3.3 – Atributos qualitativos para identificadores – *camel case versus underscore*

	(BINKLEY *et al.*, 2009) **– nota 9**	(BINKLEY *et al.*, 2013) **– nota 9**	(SHARAFI *et al.*, 2012) **– nota 8**	(SHARIF e MALETIC, 2010) **– nota 8**
Identificador com camelCase	O estilo *camel case* proporciona maior corretude, porém maior tempo de resposta em questões de compreensão de código, quando comparado com o *undercore.* Para programadores iniciantes, o *underscore* apresenta melhor tempo de resposta em questões de compreensão.		O estudo em questão tem o objetivo de avaliar o impacto do gênero na compreensão de identificadores escritos nos dois estilos em questão. Quando o gênero não foi levado em consideração, não houve diferença na corretude de respostas para questões de compreensão de código entre os participantes e nos dois estilos avaliados. Quando o gênero foi levado em consideração, o estudo mostrou que programadores homens acertam mais respostas quando avaliam código com estilo *camel case* do que quando avaliam código com *underscore*. Apesar disso, não houve diferença na acurácia das respostas entre homens e mulheres para esse o estilo *camel case*. Para o estilo underscore, homens acertaram menos questões de compreensão do que mulheres, contudo estas levaram mais tempo na avaliação das questões.	Não existe diferença significativa dos resultados de corretude entre identificadores com *camel case* e identificadores com *underscore*. Embora tenha sido identificado que o *camel case* pode levar a uma falha na corretude quando há erros de escrita no final do identificador. *Camel case* produz um menor gap de tempo entre iniciantes e experientes. Identificadores com *underscore* levam a uma maior agilidade na identificação de identificadores corretos. Iniciantes tem mais proveito de tempo que experientes quando estão frente a identificadores com *underscore*.
Identificador com under_score				

Tabela 3.4 – Atributos qualitativos para identificadores – palavras completas *versus* abreviações e caractere simples

	(LAWRIE *et al.*, 2006) **e** (LAWRIE *et al.*, 2007) **– nota 9**
Identificador com palavras completas	Acarreta na melhor compreensão do código fonte quando comparado com abreviação e caractere simples.
Identificador com abreviações	Compreensão maior que a de caractere simples. Em relação a identificadores com palavras completas não foi constatada grande diferença.
Identificador com apenas 1 caractere	Baixa compreensão quando comparada com palavras completas e abreviações.

Tabela 3.5 – Atributos qualitativos para identificadores – identificadores mnemônicos *versus* inapropriados

	(TEASLEY, 1994) **– nota 6**
Identificadores mnemônicos	No caso do experimento com inexperientes, os identificadores mnemônicos foram significativos para a compreensão apenas de funções, indicando que identificadores problemáticos prejudicam mais esse nível de entendimento do código. Não houve nenhuma diferença significativa na compreensibilidade de códigos diferentes por estudantes mais avançados.
Identificadores inapropriados	

3.3.4.1.2 Atributos Qualitativos para Comentários

Os atributos qualitativos para comentários dizem respeito à importância dos comentários em diferentes configurações do código fonte, principalmente relacionadas com a modularidade do código. As Tabela 3.8 e Tabela 3.7 sumarizam os resultados obtidos com a análise dos artigos selecionados para esse grupo.

Tabela 3.6 – Atributos qualitativos para comentários – procedimentos e comentários

	(TENNY, 1988) **– nota 6**
Código em uma linha e sem comentários	Os resultados do estudo evidenciaram que os comentários só fizeram diferença na compreensibilidade dos códigos, quando estes foram inseridos no contexto de códigos sem procedimentos. Isso pode ser explicado pelo fato de que os códigos sem procedimentos não tinham divisão de responsabilidades e os comentários de alguma forma passaram essa noção de responsabilidade para os participantes do estudo.
Código em uma linha e com comentários	
Código com procedimento e sem comentário	
Código com procedimento e com comentário.	
Código sem procedimento e sem comentário	
Código sem procedimento e com comentário	

Tabela 3.7 – Atributos qualitativos para comentários – modularidade e comentários

	(WOODFIELD, DUNSMORE e SHEN, 1981) **– nota 8**
Presença de comentários em códigos com modularidade monolítica	Os participantes que receberam códigos fonte com comentários pontuaram mais nas perguntas do que aqueles que receberam códigos sem comentários, independente do tipo de modularidade do código analisado. O uso de comentários em códigos de modularidade funcional auxiliou na compreensão geral do que uma determinada função faz e no seu objetivo para o programa como um todo. Em contrapartida, comentários para identificar módulos ao longo do código não se mostraram úteis.
Presença de comentários em códigos com modularidade funcional	
Presença de comentários em códigos com super modularidade	
Presença de comentários em códigos com modularidade de tipo abstrato de dados	

3.3.4.1.3 Atributos Qualitativos para Indentação

Os atributos qualitativos para indentação dizem respeito à quantidade de espaços utilizados para indentar o código. A Tabela 3.8 sumariza os resultados obtidos com a análise do artigo selecionado para esse grupo.

Tabela 3.8 – Atributos qualitativos para indentação

	(MIARA *et al.*, 1983) **– nota 9**
Inexistência de indentação	Os melhores resultados de compreensibilidade do estudo obtidos foram com a análise de códigos com indentação de 2 e 4 espaços. Indentação com 6 espaços começam a prejudicar a compreensão conforme a profundidade (aninhamento) do programa vai crescendo, já que a linha do programa fica cada vez maior e mais difícil de ler por completo na tela.
Indentação com 2 espaços	
Indentação com 4 espaços	
Indentação com 6 espaços	

3.3.4.1.4 Atributos Qualitativos para Organização do Código

Os atributos qualitativos para organização do código dizem respeito à separação existente entre declarações de variáveis e a referência de uso dessas variáveis. A Tabela 3.9 sumariza os resultados obtidos com a análise do artigo selecionado para esse grupo.

Tabela 3.9 – Atributos qualitativos para organização do código

	(SASAKI, HIGO e KUSUMOTO, 2013) **– nota 8**
Declarações espaçadas de suas referências	16 dos 20 métodos avaliados tiveram melhor legibilidade/compreensibilidade quando declarações de variáveis estavam próximas de suas referências de uso.
Declarações próximas de suas referências	

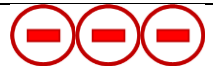
3.3.4.2 Respostas às Questões de Pesquisa – Atributos Quantitativos

Os 36 atributos quantitativos representam características do código fonte que podem apenas ser mensuradas a partir de números. Estão relacionados com diferentes aspectos do código, principalmente sintáticos. A Tabela 3.10 sumariza os resultados obtidos com a análise dos artigos selecionados para esse grupo.

Tabela 3.10 – Atributos quantitativos e suas relações de impacto para a legibilidade e compreensibilidade de código fonte

CATEGORIA	**Atributo**	(BUSE e WEIMER, 2008) **e** (BUSE e WEIMER, 2010) **– nota 8**	(DEYOUNG e KAMPEN, 1979) **– nota 5,95**	(FEIGENSPAN *et al.*, 2011) **– nota 7,5**	(JøRGENSEN, 1980) **– nota 9**	(POSNETT, HINDLE e DEVANBU, 2011) **– nota 7,5**	(STEIDL, HUMMEL e JUERGENS, 2013) **–nota 8**
LAYOUT	**Extensão da Indentação**	-	-	-		-	-
LAYOUT	**Extensão das linhas em branco**	-	-	-		-	-
LAYOUT	**Indentação (espaço em branco precedente) máxima**		-	-	-	-	-
LAYOUT	**Indentação (espaço em branco precedente) média**		-	-	-	-	-
LAYOUT	**Quantidade média de linhas em branco**		-	-	-	-	-
LAYOUT	**Quantidade média de parênteses**		-	-	-	-	-
LAYOUT	**Tamanho máximo de linha**		-	-	-	-	-

LAYOUT	Tamanho médio de linha	⊖⊖⊖	-	-	-	-	-
COMENTÁRIO	Coeficiente de correlação entre comentário e código fonte	-	-	-	-	-	⊖
COMENTÁRIO	Extensão dos comentários	-	-	-	⊕	-	-
COMENTÁRIO	Quantidade média de comentários	⊕	-	-	-	-	-
COMENTÁRIO	Quantidade média de períodos	⊖⊖⊖	-	-	-	-	-
COMENTÁRIO	Tamanho do comentário (quantidade de palavras em um comentário)	-	-	-	-	-	⊕
COMENTÁRIO	Volume dos comentários (em caracteres)	-	⊕⊕	-	-	-	-
NOMENCLATURA	Tamanho máximo de identificador	⊖⊖	-	-	⊕	-	-
NOMENCLATURA	Tamanho médio de identificador	✖	⊕⊕	-	⊕	-	-

PROGRAMAÇÃO	**Nível de implementação estimado**	-		-	-	-	-
PROGRAMAÇÃO	**Número de linhas contendo *statements***	-		-	-	-	-
PROGRAMAÇÃO	**Número de operandos**	-		-	-	-	-
PROGRAMAÇÃO	**Quantidade de ocorrências do caractere que mais apareceu no bloco**		-	-	-	-	-
PROGRAMAÇÃO	**Quantidade de ocorrências do identificador que mais apareceu no bloco**		-	-	-	-	-
PROGRAMAÇÃO	**Quantidade máxima de identificadores**		-	-	-	-	-
PROGRAMAÇÃO	**Quantidade média de comandos goto por *label***	-	-	-		-	-
PROGRAMAÇÃO	**Quantidade média de identificadores**		-	-	-	-	-
PROGRAMAÇÃO	**Quantidade média de *loops***		-	-	-	-	-

PROGRAMAÇÃO	Quantidade média de operadores aritméticos		-	-	-	-	-
PROGRAMAÇÃO	Quantidade média de operadores aritméticos por 100 linhas	-	-	-		-	-
PROGRAMAÇÃO	Quantidade média de operadores de comparação		-	-	-	-	-
PROGRAMAÇÃO	Quantidade média de palavras reservadas		-	-	-	-	-
PROGRAMAÇÃO	Quantidade média de vírgulas		-	-	-	-	-
PROGRAMAÇÃO	Tamanho (número de caracteres)	-	-	-	-		-
PROGRAMAÇÃO	Tamanho (número de linhas)	-	-		-		-
PROGRAMAÇÃO	Tamanho (número de palavras)	-	-	-	-		-
PROGRAMAÇÃO	Total de operandos e operadores ou tamanho do programa (Halstead Metrics)	-		-	-		-

PROGRAMAÇÃO	**Total de operandos únicos e operadores únicos ou vocabulário do programa (Halstead Metrics)**	-	-	-	-	⊖	-
PROGRAMAÇÃO	**Volume (Halstead Metrics)**	-	-	-	-	⊖	-

⊕⊕⊕	Impacto positivo. Relação alta com a característica de qualidade.
⊕⊕	Impacto positivo. Relação moderada com a característica de qualidade.
⊕	Impacto positivo. Relação baixa ou não especificada com a característica de qualidade.
⊗	Atributo relatado no artigo, porém sem relação com a característica de qualidade.
⊖⊖⊖	Impacto negativo. Relação alta com a característica de qualidade.
⊖⊖	Impacto negativo. Relação moderada com a característica de qualidade.
⊖	Impacto negativo. Relação baixa ou não especificada com a característica de qualidade.
-	Atributo não mencionado no artigo.

A Tabela 3.10 mostra todos os atributos quantitativos diferentes encontrados nos artigos e que impactam (positiva ou negativamente) na legibilidade e/ou compreensibilidade de código fonte. Não consideramos, nessa agregação, atributos de baixa relação nem atributos sem relação de impacto com as características de qualidade em questão, à exceção daqueles constantes ou semelhantes a atributos de qualidade que aparecem em outros artigos como atributos de impacto na legibilidade e/ou compreensibilidade de código fonte (e. g. “Quantidade média de operadores aritméticos” – atributo semelhante a outros atributos que lidam com operadores – e “Tamanho (número de linhas)” – que não apresenta relação de impacto em um estudo, porém que apresenta impacto positivo em outro estudo).

É importante destacar que somente foram agregados resultados de estudos que utilizaram o mesmo atributo para avaliação. É possível perceber que alguns atributos possuem forte relação entre si, como é o caso do atributo “Quantidade média de operadores aritméticos por 100 linhas” e do “Quantidade média de operadores aritméticos”, contudo a fórmula de cálculo deles não é a mesma, não configurando, então, um mesmo atributo. Em todo caso, é pretendido uma análise agrupada desses atributos similares posteriormente, durante a formulação das diretrizes de codificação baseadas em evidência.

Ainda em relação a essa fase de agregação, identificamos quatro grupos diferentes de atributos quantitativos. São eles: i) atributos de layout – que lidam com aspectos mais visuais do código; ii) atributos de comentários – que lidam com características específicas dos comentários do código; iii) atributos de nomenclatura – que lidam com identificadores; e iv) atributos de programação – que lidam com aspectos mais sintáticos e de lógica do programa.

3.4 Considerações sobre o Capítulo

Ao longo deste capítulo levantamos a questão da subjetividade do conceito qualidade e delimitamos o escopo de qualidade tratado pelo trabalho: qualidade de produto. Mostramos que podemos avaliar a qualidade de produto sob diferentes perspectivas (características de qualidade), sendo que para esta dissertação foram escolhidas a legibilidade e compreensibilidade de código fonte devido ao diagnóstico obtido com o *survey* na empresa Alfa.

Diferentes tipos de trabalhos em cinco categorias distintas foram apresentados como modo de fornecer uma visão geral do que tem sido proposto para avaliar ou melhorar a legibilidade e a compreensibilidade do código fonte ou ainda para auxiliar tarefas de compreensão do código e do software como um todo. Nesse contexto, enfatizamos que é possível ver a compreensibilidade sobre duas dimensões: baixo

nível de abstração e alto nível de abstração. Conduzimos, então, uma busca estruturada para identificação de atributos de código fonte que impactassem a legibilidade e a compreensibilidade de baixo nível de abstração. Queríamos com esses atributos ter subsídios para a formulação de diretrizes de codificação para legibilidade e compreensibilidade baseadas em evidências.

De 18 artigos disponíveis que passaram pelos critérios de inclusão foram extraídos um total de 59 atributos de código fonte diferentes e com evidências dos seus impactos para legibilidade e/ou compreensibilidade, sendo 23 atributos qualitativos e 36 atributos quantitativos. O capitulo a seguir mostra como esses atributos foram utilizados, juntamente com características de códigos avaliados como sendo de baixa, alta e intermediária legibilidade/compreensibilidade pelos programadores da empresa Alfa, para a formulação de diretrizes de codificação específicas para a organização.

4 Diretrizes de Codificação para Legibilidade e Compreensibilidade de Código Fonte

Neste capítulo o procedimento para criação das diretrizes e a primeira versão das diretrizes de codificação para legibilidade e compreensibilidade de código fonte são detalhados. Para a construção das diretrizes foram utilizados tantos atributos de código vindos da literatura técnica como da empresa Alfa. Detalhes da captura de código fonte e da análise dos atributos de código para legibilidade e compreensibilidade identificados na organização também constam neste capítulo.

4.1 Introdução

As diretrizes de codificação devem servir para auxiliar as empresas (em particular a empresa Alfa) a alinharem suas perspectivas de qualidade em código fonte. Assim, é importante que sejam observadas as perspectivas de qualidade da própria empresa em relação à legibilidade e compreensibilidade de código para então formular diretrizes de codificação específicas para a organização.

Para tanto, foi elaborada uma estratégia para extração de atributos de código que, na percepção da empresa Alfa, impactassem na legibilidade e compreensibilidade de código fonte. A seção a seguir detalha o planejamento para a captura e a análise dos códigos da organização que subsidiaram a extração de atributos de código para legibilidade e compreensibilidade na organização e que, em conjunto com os atributos de código identificados na literatura técnica, serviram de insumo para a formulação das diretrizes de codificação baseadas em evidência.

4.2 Captura e Análise de Códigos Fonte da Empresa Alfa

As subseções a seguir detalham o planejamento de captura dos códigos fonte da empresa Alfa, bem como o processo de análise e agrupamento dos dados obtidos com o estudo.

4.2.1 Planejamento da Captura de Códigos

O objetivo desse estudo de observação é:

- ***Analisar*** *atributos de código fonte* ***com o propósito de*** *caracterizar* ***com respeito ao*** *impacto (negativo ou positivo) na legibilidade e/ou compreensibilidade de código fonte* ***do ponto de vista de*** *programadores de firmware, software embarcado e software da empresa Alfa* ***no contexto de*** *desenvolvimento de firmware, software embarcado e software da empresa Alfa.*

4.2.1.1 Participantes do Estudo

Assim como ocorreu no *survey*, planejamos capturar a percepção de legibilidade e compreensibilidade de todos os programadores da empresa Alfa localizados no Brasil. Sabendo agora que os setores de desenvolvimento do domínio de rastreamento e de inteligência ambiental estavam fisicamente separados, decidimos então realizar duas seções para explicar a tarefa a ser realizada, uma para cada domínio de desenvolvimento, sendo que os programadores do setor de sistemas de informação devem participar da sessão junto com os programadores do domínio de rastreamento. De acordo com a quantidade de participantes do *survey*, temos uma expectativa de 21 participantes nesse novo estudo.

4.2.1.2 Tarefa a Ser Realizada e Formulários de Captura de Código

Para capturar a perspectiva de legibilidade e compreensibilidade de baixo nível de abstração (tipo 1), planejamos a realização da seguinte tarefa a ser repassada aos programadores da empresa Alfa:

> *"Identificar e extrair 3 trechos de códigos, um de alta, outro de baixa e outro de intermediária legibilidade e compreensibilidade, não produzidos por você, mas com os quais você mantém contato direto em suas tarefas de desenvolvimento na empresa. Os trechos de código devem ser autocontidos, ou seja, devem representar um algoritmo independente de outros trechos de código dentro do software. É importante que a lógica de programação do trecho de código extraído não seja quebrada. O trecho de código deve conter no mínimo 3 comandos consecutivos (sendo pelo menos 1 composto), devendo respeitar as marcas de mínimo e máximo no espaço delimitado no template de captura de código. Comentários, espaços, tabulações e linhas em branco existentes não devem ser retirados dos trechos de código, ou seja, o código deve ser extraído exatamente como foi encontrado."*

Como instrumentos de auxílio a essa tarefa, dois formulários distintos foram elaborados. O primeiro formulário contém instruções para a captura de trechos de código de alta, baixa e intermediária legibilidade e compreensibilidade de baixo nível

de abstração (compreensibilidade tipo 1). Como queremos capturar a perspectiva pessoal de cada participante, fornecemos uma descrição literal sobre legibilidade e compreensibilidade, não entrando no mérito de como esses conceitos poderiam ser mapeados para o código fonte. O segundo formulário, cujo acesso pelo programador deve ser dado somente após a captura dos códigos, contém um conjunto de atributos de código fonte (extraídos da literatura técnica) mapeados em frases que devem ser avaliados pelo programador em relação a seu impacto na legibilidade e compreensibilidade de código. Esse formulário contém ainda questões abertas para que os participantes indiquem aspectos/atributos de código existentes em códigos de alta e de baixa legibilidade e compreensibilidade. Detalhes dos formulários no APÊNDICE C – Instrumentos de Captura de Código Fonte na Empresa Alfa e Agregação de Informações dos Códigos.

É importante informar que foram elaborados ainda uma segunda tarefa e um terceiro formulário para capturar códigos relacionados com compreensibilidade de alto nível de abstração (compreensibilidade tipo 2), contudo pretendemos tratar nesse primeiro momento somente a legibilidade e compreensibilidade tipo 1, no entanto aproveitamos o momento com os programadores para adiantar ações futuras na organização.

4.2.1.3 Agregação e Análise dos Dados

Os códigos coletados devem ser analisados qualitativamente pela pesquisadora à luz dos atributos de código identificados com a busca estruturada, e levando em consideração os quatro grupos de atributos encontrados, a saber: *layout*, comentário, nomenclatura e programação. As tabelas a seguir elencam e fornecem uma descrição sobre os aspectos e atributos de código para cada grupo.

Tabela 4.1 – Aspectos e atributos de código para o grupo *layout*

Aspectos e Atributos de Código para *Layout*	**Descrição**
Declaração e Utilização das Variáveis	Refere-se ao posicionamento das declarações de variáveis e de suas referências nos trechos de código. Referência ao atributo encontrado em (SASAKI, HIGO e KUSUMOTO, 2013).
Tamanho dos Comandos	Refere-se ao tamanho dos comandos presentes nos trechos de código. Referência aos atributos encontrados em (BUSE e WEIMER, 2008) e (BUSE e WEIMER, 2010).
Separação dos Comandos	Refere-se ao uso de linhas em branco para

	separação de comandos nos trechos de código. Referência aos atributos encontrados em (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (JøRGENSEN, 1980).
Indentação	Refere-se ao uso da indentação nos trechos de código. Referência aos atributos encontrados em (MIARA *et al.*, 1983), (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (JøRGENSEN, 1980).
Uso de Parênteses	Refere-se ao uso de parênteses em expressões aritméticas, de comparação e lógicas. Referência aos atributos encontrados em (BUSE e WEIMER, 2008) e (BUSE e WEIMER, 2010).
Outras Observações de *Layout*	Refere-se a demais atributos relacionados com o visual do código que possam ser observados nos trechos de código e que não foram encontrados nos artigos selecionados.

Tabela 4.2 – Aspectos e atributos de código para o grupo comentário

Aspectos e Atributos de Código para Comentário	**Descrição**
Tamanho dos Comentários	Refere-se ao tamanho dos comentários presentes nos trechos de código. Referência aos atributos encontrados em (DEYOUNG e KAMPEN, 1979), (DEYOUNG e KAMPEN, 1979), (JøRGENSEN, 1980) e (STEIDL, HUMMEL e JUERGENS, 2013).
Objetivo e Características dos Comentários	Refere-se ao propósito de uso dos comentários, a correlação deles com os trechos de códigos, bem como demais características possíveis de serem observadas. Referência aos atributos encontrados em (TENNY, 1988), (WOODFIELD, DUNSMORE e SHEN, 1981), (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (STEIDL, HUMMEL e JUERGENS, 2013).

Tabela 4.3 – Aspectos e atributos de código para o grupo nomenclatura

Aspectos e Atributos de Código para Nomenclatura	**Descrição**

Tamanho dos Identificadores	Refere-se ao tamanho dos identificadores utilizados nos trechos de código. Referência aos atributos encontrados em (LAWRIE *et al.*, 2006), (LAWRIE *et al.*, 2007), (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010), (DEYOUNG e KAMPEN, 1979) e (JøRGENSEN, 1980).
Estilo de Escrita dos Identificadores	Refere-se ao estilo de escrita dos identificadores presentes nos trechos de código, bem como o uso de caractere simples, abreviações e palavras completas, além da característica de entendimento dos identificadores. Referência aos atributos encontrados em (BINKLEY *et al.*, 2009), (BINKLEY *et al.*, 2013), (SHARAFI *et al.*, 2012), (SHARIF e MALETIC, 2010), (LAWRIE *et al.*, 2006), (LAWRIE *et al.*, 2007) e (TEASLEY, 1994).

Tabela 4.4 – Aspectos e atributos de código para o grupo programação

Aspectos e Atributos de Código para Programação	**Descrição**
Loops	Refere-se ao uso de comandos de loops em geral, incluindo goto, nos trechos de código. Referência aos atributos encontrados em (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (JøRGENSEN, 1980).
Operações	Refere-se aos tipos de operações presentes nos trechos de código. Referência aos atributos encontrados em (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010) e (JøRGENSEN, 1980).
Operadores e Operandos	Refere-se às características dos operandos e operadores presentes nos trechos de código. Referência aos atributos encontrados em (BUSE e WEIMER, 2008), (BUSE e WEIMER, 2010), (DEYOUNG e KAMPEN, 1979), (JøRGENSEN, 1980) e (POSNETT, HINDLE e DEVANBU, 2011).

Cada trecho de código coletado deve lido e, para cada aspecto/atributo de código identificado, uma codificação, registrada; por exemplo, para o atributo “tamanho dos comandos”, um código indicando “curtos”; “médios”; “longos” ou “muito longos” deve ser registrado como forma de qualificar o código fonte em relação a esse

atributo. Ao final da análise dos códigos fonte coletados, uma percepção qualitativa sobre cada grupo de atributos/aspectos e para cada tipo de código fonte (de alta, baixa e intermediária legibilidade e compreensibilidade) deve ser realizada. Essa percepção servirá como base para justificar a existência de determinada diretriz e mesmo para aprimorar a explicação das diretrizes em geral.

É importante salientar que apesar de alguns atributos encontrados nos artigos da busca estruturada referirem-se à simples contagem de ocorrência, como a análise dos códigos será feita de forma qualitativa, objetivamos também avaliar qualitativamente os atributos quantitativos mencionados nos artigos. Além disso, os atributos identificados nos artigos devem servir como guia de análise dos códigos capturados, não devendo, no entanto, limitar a análise dos códigos. Assim, são esperados que novos aspectos/atributos surjam com a análise qualitativa dos códigos.

Em relação às questões objetivas e abertas do formulário de percepção sobre legibilidade e compreensibilidade, as respostas objetivas devem ser contabilizadas e agrupadas de acordo com a escala *likert*. Seja w o peso da opção (1: sim; -1: não; 0: indiferente); x a contagem de ocorrência da opção para uma dada frase; a agregação a das opções para a frase será como segue, $a = \mathrm{x1w1} + \mathrm{x2w2} + \mathrm{x3w3}$. Somente os atributos com saldo positivo final após a agregação devem ser levados em consideração para a formulação das diretrizes. Atributos relacionados com compreensibilidade de alto nível de abstração devem ser desconsiderados nesse momento de formulação de diretrizes para compreensibilidade de baixo nível de abstração. Para as questões abertas, as respostas devem ser codificadas, tendo sua ocorrência contabilizada.

4.2.1.4 Planejamento para Formulação das Diretrizes

As diretrizes devem ser formuladas com base primeiramente na percepção dos programadores da empresa sobre a legibilidade e compreensibilidade de código fonte. Entendemos que nem todos os atributos identificados na literatura técnica podem ser aplicados ao contexto da empresa Alfa, e objetivamos formular um conjunto de diretrizes mínimas, porém que consigam suprir as necessidades iniciais da organização.

Assim, os aspectos/atributos de código identificados pelos programadores como importantes para a legibilidade e compreensibilidade devem servir como ponto inicial para a criação das diretrizes de codificação, sendo estas refinadas com a análise qualitativa dos códigos capturados e suportadas pelas evidências encontradas na literatura em relação ao seu impacto para legibilidade e compreensibilidade de código. As diretrizes devem indicar instruções a serem seguidas nos códigos da

organização como forma a atender a expectativa de apresentação/comportamento de algum atributo de código fonte. Detalhes das tarefas a serem realizadas e dos produtos intermediários de trabalho a serem produzidos até a elaboração das diretrizes de codificação são apresentados na Figura 4.1 a seguir.

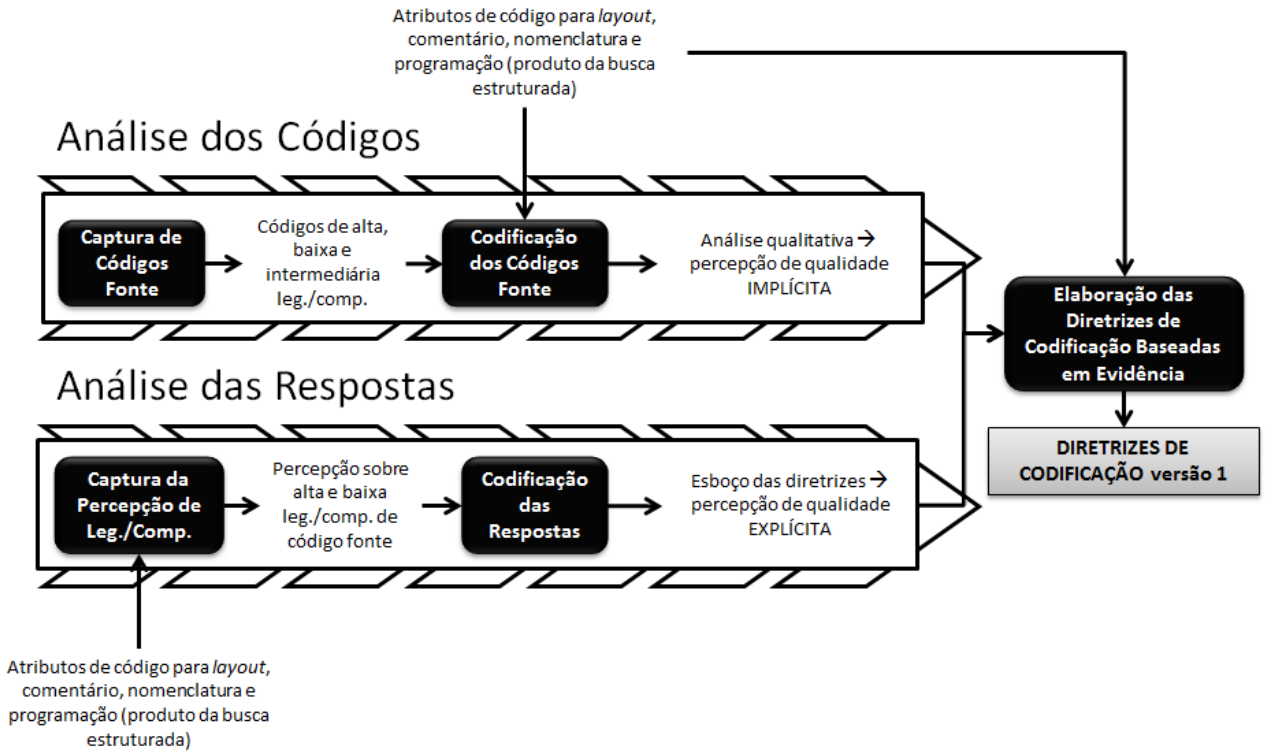


Figura 4.1 – Tarefas e produtos intermediários a serem produzidos até a formulação das diretrizes de codificação

4.2.2 Captura e Análise de Códigos

A dependência da disponibilidade dos respondentes para participar das seções de explanação da tarefa a ser realizada acabou modificando o planejamento do estudo. No dia 20/02/2014 realizamos 4 seções de explanação diferentes com os programadores da empresa Alfa. A primeira com 5 programadores; a segunda com 3; a terceira com 3 e a última com 10; totalizando 21 programadores (alguns diferentes dos que participaram do *survey*). Todas tiveram duração média de 30 minutos e a última foi realizada somente com os programadores do domínio de inteligência ambiental.

Ao longo dos dias 20 e 21/02/2014 foram recebidos pessoalmente os códigos de 9 programadores dos setores de rastreamento e sistemas de informação. Ao receber os códigos, o formulário com as questões sobre legibilidade e compreensibilidade de código fonte foi entregue aos programadores que tiveram um prazo de uma semana para entregá-lo preenchido via correio eletrônico. Até o dia

17/04/2014 tínhamos recebido 11 formulários contendo os trechos de código solicitados (nenhum do setor de inteligência ambiental) e 8 formulários de capturada da percepção sobre legibilidade e compreensibilidade de código fonte.

4.2.2.1 Respostas sobre Legibilidade e Compreensibilidade

Após ter realizado o agrupamento das respostas para a questão fechada, os dados apresentados na Tabela 4.5 foram obtidos. Somente dados de agrupamento relacionados com legibilidade e compreensibilidade de baixo nível de abstração são exibidos.

Tabela 4.5 – Agrupamento das respostas para a questão fechada sobre legibilidade e compreensibilidade de código fonte – 8 respondentes

Um código é legível/compreensível quando...	SIM	NÃO	INDIFE-RENTE	AGRUPA-MENTO	ID do Programador (SIM)
Os identificadores das variáveis, constantes, funções, procedimento e métodos são consistentes quanto à forma de escrita (mesma língua ou mesmo esquema de separação entre palavras).	8	0	0	8	3; 4; 7; 22; 23; 24; 25; 27
É indentado.	8	0	0	8	3; 4; 7; 22; 23; 24; 25; 27
Possui identificadores de funções/procedimentos/métodos que exprimem o real comportamento da respectiva função/procedimento/método.	8	0	0	8	3; 4; 7; 22; 23; 24; 25; 27
Separa os comandos em linhas diferentes.	7	0	1	7	3; 4; 22; 23; 24; 25; 27
Apresenta comentários que sumarizam o funcionamento de determinado trecho de código.	7	0	1	7	3; 4; 7; 22; 23; 24; 25
Possui identificadores de variáveis e constantes que exprimem a real finalidade da respectiva variável/constante.	7	0	0	7	3; 4; 7; 22; 23; 24; 25; 27
Separa as palavras dos identificadores utilizando o estilo CamelCase. (ex.: outraPalavra)	6	0	2	6	3; 4; 7; 22; 23; 24; 25; 27
Utiliza palavras completas para compor identificadores. (ex.: contador)	6	0	2	6	3; 7; 22; 24; 25; 27
Apresenta comentários que explicam como deve ocorrer o uso correto do trecho de código.	5	1	2	4	4; 22; 23; 24; 25
Separa as palavras dos identificadores com underscore. (ex.: uma_palavra)	4	3	1	1	4; 23; 24; 25
Insere linhas em branco entre diferentes comandos.	3	2	3	1	4; 24; 25

Apresenta documentação do trecho de código separada da implementação.	2	2	4	0	25; 27
Não envolve a utilização de muitas comparações.	1	2	5	-1	23
Utiliza abreviações de palavras para compor identificadores. (ex.: cont)	1	3	4	-2	27
Não envolve a utilização de muitas expressões aritméticas.	0	3	5	-3	-
Apresenta comentários que sumarizam o funcionamento de cada linha do código.	0	4	4	-4	-
É dependente de muitos outros trechos de código.	0	4	3	-4	-
Não diferencia diferentes palavras existentes em um mesmo identificador. (ex.: maisoutrapalavra)	0	7	0	-7	-

Interessante observar com as respostas obtidas que, diferente do que foi apresentado pela literatura técnica, as operações de modo geral parecem não ser tão cruciais para a legibilidade e compreensibilidade do código fonte para os programadores da empresa Alfa; talvez isso seja justificado pelo estilo de problema que eles estão acostumados a lidar na empresa, que envolve muita manipulação de dados em baixo nível de código. Outro fato a ser observado é que, embora não exista um consenso em relação a que estilo de escrita é melhor de ser utilizado nos identificadores, é consenso que a não diferenciação de palavras é prejudicial para a legibilidade do código.

Para as questões abertas, as tabelas seguintes sumarizam os códigos identificados nas respostas. Os itens sublinhados indicam códigos que apareceram na característica (legibilidade ou compreensibilidade) “errada”, fato que enfatiza a confusão que esses dois termos pode causar.

Tabela 4.6 – Agrupamento das respostas para a questão aberta sobre “um código é mais legível quando...” – 8 respondentes

MAIS LEGÍVEL quando há...	Ocorrências	ID do Programador
Padronização	3	4; 7; 23
Indentação	3	23; 25; 27;
Espaçamento	1	27
Tamanho de comandos limitados	1	7
Organização do código dentro do arquivo	1	22
Declaração em conjunto de variáveis	3	3; 23; 24
Diferenciação de variáveis globais de locais	2	22; 24
Identificadores autoexplicativos	2	4; 7

Uso de defines e enumeradores para substituir números isolados	1	23
Comentários para explicitação dos parâmetros e dos retornos de funções;	1	23

Tabela 4.7 – Agrupamento das respostas para a questão aberta sobre “um código é mais compreensível quando...” – 8 respondentes

MAIS COMPREENSÍVEL quando há...	Ocorrências	ID do Programador
Programação objetiva	1	3
Cabeçalho	2	4; 25
Comentários	2	7; 23
Comentários para explicação da finalidade e do modo de funcionamento de módulos	3	22; 23; 25
Comentários para explicitação dos parâmetros e dos retornos de funções	3	4; 23; 25
Referência em caso de utilização de códigos de terceiros	1	4
Uso de defines e enumeradores para substituir números isolados	1	24
Identificadores autoexplicativos	2	25;27
Comentário de linha quando necessário	1	23

Tabela 4.8 – Agrupamento das respostas para a questão aberta sobre “um código é menos legível quando...” – 8 respondentes

MENOS LEGÍVEL quando há...	Ocorrências	ID do Programador
Inconsistência de língua utilizada no código	1	4
Falta de indentação	1	7
Ausência de espaçamento	1	23
Dificuldade na identificação de demarcadores de início e fim de bloco	1	24
Ausência de parênteses em expressões aritméticas e de comparação	1	23
Existência de múltiplos comandos por linha	2	4; 27
Comandos extensos e sem quebra de linha	2	4; 23
Ausência de linha em branco para separar comandos	1	25
Existência de diretivas de compilação ao longo do código	1	22
Existência de códigos mortos	1	4
Identificadores com pouco significado	1	7

Tabela 4.9 – Agrupamento das respostas para a questão aberta sobre “um código é menos compreensível quando...” – 8 respondentes

MENOS COMPREENSÍVEL quando há...	Ocorrências	ID do Programador
Ausência de comentários	2	4; 25
Identificadores com pouco significado	2	25; 27
Uso de números sem explicação em comparações/atribuições/etc.	1	23
Falta de referência a variáveis externas	1	7
Ausência de espaçamento	1	23
Ausência de linha em branco para separar comandos	1	23
Funções muito grandes	1	24

Ao analisar as respostas obtidas nos dois tipos de questões (fechada e aberta), é possível identificar um mapeamento, às vezes direto, entre elas, como é o caso das respostas relacionadas com a indentação de código. Vista unanimemente como atributo agregador de legibilidade nas respostas para a questão fechada, a indentação foi ainda enfatizada por 4 programadores nas respostas às questões abertas. Alguns aspectos/atributos foram observados somente nas respostas à questão fechada, como é o caso da utilização de estilos de escrita para diferenciação de palavras nos identificadores. Outros aspectos/atributos foram identificados apenas nas respostas às questões abertas, como a utilização de parênteses para enfatizar operandos em expressões aritméticas e de comparação. A agregação dessas informações é dada pela Tabela 4.10 a seguir. Todos os itens elencados nas respostas às questões abertas foram considerados nessa agregação, contudo para os itens da questão fechada, somente aqueles que obtiveram agregação positiva foram considerados.

Tabela 4.10 – Agrupamento das respostas obtidas com o formulário de percepção de legibilidade e compreensibilidade de código – esboço das diretrizes

ESBOÇO DAS DIRETRIZES	ID do Programador (Questões Fechadas)	ID do Programador (Questões Abertas)	Percepção de Importância na Equipe (explícita) - Total 8
Arquivos e Pastas			
Organizar e tipar arquivos por conteúdo*	-	-	1 (mentor)
Organizar código dentro dos arquivos*	-	3; 22; 23; 24	1 (mentor) + 4
Layout			
Indentar o código de forma consistente	3; 4; 7; 22; 23; 24; 25; 27	7; 23; 25; 27	8

Utilizar demarcadores de início e fim de bloco de forma consistente	-	24	1
Espaçar o código de forma consistente	-	23; 27	2
Utilizar parênteses em expressões aritméticas, lógicas e de comparação	-	23	1
Utilizar linhas em branco para separar comandos	4; 24; 25	23; 25	4
Escrever um comando por linha	3; 4; 22; 23; 24; 25; 27	4; 27	7
O tamanho do comando deve ser limitado	-	4; 7; 23	3
Variáveis devem ser declaradas em conjunto	-	3; 22; 23; 24	4
Constantes devem ser declaradas em conjunto	-	3; 22; 23; 24	4
Comentário			
Inserir cabeçalho de arquivo	-	4; 7; 25	3
Inserir comentários para explicação da finalidade, funcionamento, parâmetros, retornos, uso de código de terceiros em funções/procedimentos	3; 4; 7; 22; 23; 24; 25	4; 7; 22; 23; 25	7
Utilizar comentários de linha quando necessário	-	23	1
Nomenclatura			
Utilizar idioma único	3; 4; 7; 22; 23; 24; 25; 27	4	8
Identificadores devem ser mnemônicos e significativos	3; 4; 7; 22; 23; 24; 25; 27	4; 7; 25; 27	8
Diferenciar variáveis globais de locais	-	22; 24	2
Identificadores devem ser consistentes quanto à forma de escrita	3; 4; 7; 22; 23; 24; 25; 27	-	8
Constantes devem ser simbólicas	-	23; 24	2
Separar as palavras dos identificadores utilizando algum estilo específico	3; 4; 7; 22; 23; 24; 25; 27	-	8
Utilizar palavras completas para compor identificadores	3; 7; 22; 24; 25; 27	-	6
Programação			
Evitar comentado não utilizado	-	4	1
Programar de forma objetiva	-	3	1
Manter tamanho do código pequeno	-	4; 24	2
*Informação obtida em conversa com o mentor do desenvolvimento de software embarcado e um dos dirigentes da organização.			

As respostas foram unificadas em um código que pudesse indicar um esboço de diretriz a ser seguida. Ou seja, casos em que foi mencionado que o uso de idiomas

diferentes na codificação prejudicava a compreensibilidade do código foram mapeados para uma diretriz inicial que pudesse enfatizar o uso de um idioma único para a escrita de identificadores. O mesmo ocorrendo para as demais respostas obtidas.

Uma informação interessante que emergiu dessa análise inicial foi a criação de um novo grupo de atributos, agora, grupo de diretrizes, a saber: arquivos e pastas. Em uma conversa particular com o mentor do desenvolvimento de firmware e software embarcado, identificamos uma necessidade de contribuir também para a organização dos arquivos e das pastas dos projetos, não entrando no mérito de arquitetura, mas somente da simples organização. Foi identificado, por exemplo, que não se tinha nenhuma definição sobre como os elementos do programa deveriam ser apresentados no código ou mesmo sobre como deveriam ser organizados os arquivos dos projetos de um modo geral. Entendemos que essa solicitação estava também dentro do escopo das diretrizes propostas, uma vez que tinham relação com o acesso a informações do código e com a própria legibilidade e compreensibilidade dos arquivos e de seus conteúdos. Ainda nesse contexto, identificamos que 4 programadores indicaram a organização do código dentro do arquivo como um aspecto que auxilia a legibilidade do código fonte, confirmando a necessidade da solicitação realizada.

4.2.2.2 Análise dos Códigos de Alta, Baixa e Intermediária Legibilidade e Compreensibilidade

À luz dos atributos qualitativos e quantitativos para código fonte identificados na busca estruturada, foi realizada uma avaliação qualitativa dos códigos fonte capturados. Ao todo foram 33 trechos de código diferentes capturados, sendo 11 qualificados como sendo de alta, 11 como de baixa e 11 como de intermediária legibilidade e compreensibilidade. Conforme o planejamento, uma codificação foi realizada para sumarizar os aspectos e atributos de código existentes em cada tipo de código capturado (alta, baixa e intermediária leg./comp.) e em cada um dos quatro grupos então identificados: layout, comentário, nomenclatura e programação. A codificação realizada para os 33 trechos de código analisados pode ser vista no APÊNDICE C – Instrumentos de Captura de Código Fonte na Empresa Alfa e Agregação de Informações dos Códigos.

4.2.2.2.1 Códigos de Alta Legibilidade e Compreensibilidade

Layout: De modo geral, os códigos de alta legibilidade apresentaram declarações de variáveis separadas de suas referências; apresentaram tamanho de comandos curtos ou médios; foram escritos com apenas um comando por linha e utilizando linhas em branco para separar alguns comandos específicos; apresentaram indentação consistente; utilizaram parênteses somente quando necessário em

expressões aritméticas, de comparação e lógicas e tiveram espaçamento normal entre operandos e operadores.

Comentário: A maioria dos trechos apresentaram comentários médios a longos, explicando modo de operação, parâmetros e retorno das funções; sendo que alguns indicaram, ainda, o passa a passo dos comandos através de comentários de linha.

Nomenclatura: Em relação aos identificadores utilizados, a maioria eram curtos ou médios, porém quase todos significativos e mnemônicos e utilizavam palavras completas ou abreviações para a sua construção.

Programação: De modo geral poucos comandos de decisão e *loops* foram utilizados, sendo que a maioria nem continham *loops*. Operações diversas foram utilizadas, como aritméticas, de comparação, lógicas, com ponteiros e deslocamento de bits. Somente em um caso foi identificada a utilização de números como operandos (constantes literais) e em dois casos, a utilização de chamadas de funções como operandos.

4.2.2.2.2 Códigos de Baixa Legibilidade e Compreensibilidade

Layout: De modo geral, os códigos de baixa legibilidade apresentaram declarações de variáveis separadas de suas referências, apenas em dois dos seis casos com declarações de variáveis esse fato não ocorreu. Os trechos de código apresentaram tamanho de comandos médios a muito longos; foram escritos com apenas um comando por linha, porém não utilizando linhas em branco para separar comandos e sendo que em dois casos foi observado o uso de mais de um comando por linha. Os códigos ainda apresentaram indentação não consistente em quatro dos onze casos. Nos seis casos em que houve uso de parênteses, três foram utilizados para enfatizar operandos em expressões aritméticas, de comparação e lógicas. Em relação ao espaçamento, três apresentaram pouco espaçamento.

Comentário: Somente em três dos onze casos foram identificados comentários, porém em um deles o comentário era referente a um código não mais utilizado e em outro caso o comentário foi utilizado para demarcar fim de bloco.

Nomenclatura: Os identificadores ou eram curtos ou médios, porém quase todos não significativos nem mnemônicos, sendo que a maior parte utilizava-se de abreviações para a sua construção.

Programação: Nesse caso, uma quantidade maior de códigos contendo comandos de decisão e *loops* foi observada, alguns contendo quantidade razoável de comandos. Operações diversas foram utilizadas, como aritméticas, de comparação, lógicas, com ponteiros e deslocamento de bits, porém diferente do que ocorreu nos códigos de alta legibilidade e compreensibilidade, dos onze trechos de código, nove

apresentaram números como operandos (constantes literais) e em quatro foram identificadas chamadas de funções como operandos.

4.2.2.2.3 Códigos de Intermediária Legibilidade e Compreensibilidade

Em geral, os códigos de intermediária legibilidade e compreensibilidade apresentaram características boas e ruins dos outros dois grupos, sendo que quando apresentaram características ruins, apresentaram outras características boas, como alternativa para contornar possíveis problemas de legibilidade e compreensibilidade. Exemplo desses fatos é a presença de muitos comandos com tamanhos de médios (3) a longos (4) e muito longos (1), em contrapartida, quebras de linha foram utilizadas em 2 casos para separar comandos longos, além do uso de linhas em branco (6) para separação de comandos. A existência de comentários e identificadores significativos e mnemônicos foi outra característica marcante dos códigos de intermediária legibilidade e compreensibilidade. É importante destacar que uma grande quantidade de código apresentou números como operandos (constantes literais), porém a existência de comentários para explicação dessas constantes literais provavelmente influenciou a indicação desses códigos como sendo de intermediária e não de baixa legibilidade e compreensibilidade.

4.3 Formulação das Diretrizes

As respostas obtidas com o formulário de percepção da legibilidade e compreensibilidade de código fonte serviram como ponto de partida para a formulação das diretrizes. A função da análise dos códigos e dos dados obtidos com os artigos, nessa nova etapa de formulação das diretrizes, seria a de justificar a existência das diretrizes e mesmo a de agregar evidência de que a aplicação delas pode trazer benefícios à legibilidade e compreensibilidade de código fonte. A Tabela 4.11 apresenta novamente o esboço das diretrizes agora com foco para os resultados das análises dos códigos e para a percepção (implícita) da importância das diretrizes pela equipe.

Tabela 4.11 – Agrupamento das análises dos códigos com base no esboço das diretrizes

ESBOÇO DAS DIRETRIZES	Análise dos Códigos	Percepção de Importância na Equipe (implícita) - Total 11
Arquivos e Pastas		
Organizar e tipar arquivos por conteúdo	Sem dados.	-

Organizar código dentro dos arquivos	Sem dados conclusivos.	-
Layout		
Indentar o código de forma consistente	Todos os códigos de alta leg./comp. apresentaram indentação consistente, fato que não ocorreu em códigos de baixa leg./comp..	11
Utilizar demarcadores de início e fim de bloco de forma consistente	O uso de demarcadores consistentemente (incluindo para instruções simples) foi identificado como regra somente em códigos de alta leg./comp..	10
Espaçar o código de forma consistente	Sem dados conclusivos.	-
Utilizar parênteses em expressões aritméticas, lógicas e de comparação	Sem dados conclusivos.	-
Utilizar linhas em branco para separar comandos	Maioria dos códigos de alta leg./comp. separaram comandos com linhas em branco, oposto do que ocorreu em códigos de baixa leg./comp..	7
Escrever um comando por linha	Unanimidade entre os códigos de alta leg./comp., fato que não ocorreu em outros códigos.	11
O tamanho do comando deve ser limitado	O tamanho do comando foi identificado como fator prejudicial para leg./comp. principalmente quando o código não utiliza mecanismos para quebrar e separar os comandos ou ainda quando utiliza mais de um comando por linha.	5
Variáveis devem ser declaradas em conjunto	Sem dados conclusivos.	-
Constantes devem ser declaradas em conjunto	Sem dados conclusivos.	-
Comentário		
Inserir cabeçalho de arquivo	Somente em um código e de alta leg./comp. foi identificado o uso de cabeçalho.	1
Inserir comentários para explicação da finalidade, funcionamento, parâmetros, retornos, uso de código de terceiros em funções/procedimentos	Esses comentários só apareceram em códigos de alta leg./comp. e em um código de intermediária leg./comp..	4
Utilizar comentários de linha quando necessário	Sem dados.	-
Nomenclatura		
Utilizar idioma único	Somente em códigos de baixa e intermediária legibilidade houve inconsistência de idiomas	2

Identificadores devem ser mnemônicos e significativos	Quase todos os identificadores dos códigos de alta leg./comp. eram significativos, diferente do que ocorreu nos códigos de baixa leg./com..	10
Diferenciar variáveis globais de locais	Sem dados.	-
Identificadores devem ser consistentes quanto à forma de escrita	Sem dados conclusivos.	-
Constantes devem ser simbólicas	Maioria dos códigos de baixa leg./comp. apresentaram constantes literais e nenhum comentário para explicação. Códigos de intermediária leg/comp. apresentam constantes literais, porém com comentários para explicação.	9
Separar as palavras dos identificadores utilizando algum estilo específico	Muitos identificadores de códigos de baixa leg./comp. não apresentaram nenhuma divisão de palavras.	5
Utilizar palavras completas para compor identificadores	Maioria dos códigos de alta leg./comp. apresentaram identificadores com palavras completas.	7
Programação		
Evitar código comentado não utilizado	Sem dados conclusivos.	-
Programar de forma objetiva	Sem dados conclusivos.	-
Manter tamanho do código pequeno	Sem dados conclusivos.	-

A Tabela 4.12, a seguir, exibe, para cada esboço de diretriz, os prováveis impactos que ela pode causar na legibilidade e compreensibilidade de código fonte. Para a identificação desse impacto, uma análise dos resultados dos estudos selecionados com a busca estruturada foi feita. Cada diretriz representa pelo menos um atributo de código fonte, alguns possíveis de serem identificados nos estudos. O impacto do atributo na legibilidade e compreensibilidade de código obtido no estudo realizado é o que definiu o impacto da diretriz nessas características de qualidade. É interessante observar que nem todas as diretrizes apresentam impacto positivo nos atributos de qualidade em questão, contudo, nesse momento essas informações foram utilizadas apenas como alerta ao uso das diretrizes.

Tabela 4.12 – Relação de impacto das diretrizes para a legibilidade e compressibilidade de código fonte de acordo com a literatura técnica.

ESBOÇO DAS DIRETRIZES	Evidência na Literatura – Impacto na Legibilidade e Compreensibilidade

Arquivos e Pastas	
Organizar e tipar arquivos por conteúdo	-
Organizar código dentro dos arquivos	-
Layout	
Indentar o código de forma consistente	⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8) ⊕ (JØRGENSEN, 1980) (nota 9) ⊕ (MIARA, MUSSELMAN, *et al.*, 1983) (nota 9)
Utilizar demarcadores de início e fim de bloco de forma consistente	-
Espaçar o código de forma consistente	-
Utilizar parênteses em expressões aritméticas, lógicas e de comparação	⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8)
Utilizar linhas em branco para separar comandos	⊕ (BUSE e WEIMER, 2008) (nota 8) ⊕ (BUSE e WEIMER, 2010) (nota 8) ⊕ (JØRGENSEN, 1980) (nota 9)
Escrever um comando por linha	-
O tamanho do comando deve ser limitado	⊕ (BUSE e WEIMER, 2008) (nota 8) ⊕ (BUSE e WEIMER, 2010) (nota 8)
Variáveis devem ser declaradas em conjunto	⊖ (SASAKI, HIGO e KUSUMOTO, 2013) (nota 8)
Constantes devem ser declaradas em conjunto	⊖ (SASAKI, HIGO e KUSUMOTO, 2013) (nota 8)
Comentário	
Inserir cabeçalho de arquivo	-
Inserir comentários para explicação da finalidade, funcionamento, parâmetros, retornos, uso de código de terceiros em funções/procedimentos	⊕ (BUSE e WEIMER, 2008) (nota 8) ⊕ (BUSE e WEIMER, 2010) (nota 8) ⊕ (DEYOUNG e KAMPEN, 1979) (nota 5,95) ⊕ (JØRGENSEN, 1980) (nota 9) ⊕ (STEIDL, HUMMEL e JUERGENS, 2013) (nota 8) ⊕ (TENNY, 1988) (nota 6) ⊕ (WOODFIELD, DUNSMORE e SHEN, 1981) (nota 8)
Utilizar comentários de linha quando necessário	⊕ (STEIDL, HUMMEL e JUERGENS, 2013) (nota 8)
Nomenclatura	
Utilizar idioma único	-
Identificadores devem ser mnemônicos e significativos	⊕ (LAWRIE, MORRELL, *et al.*, 2006) (nota 9) ⊕ (LAWRIE, MORRELL, *et al.*, 2007) (nota 9) ⊕ (TEASLEY, 1994) (nota 6)
Diferenciar variáveis globais de locais	-
Identificadores devem ser consistentes quanto à forma de escrita	-
Constantes devem ser simbólicas	-

Separar as palavras dos identificadores utilizando algum estilo específico	(+) (BINKLEY, DAVIS, *et al.*, 2009) (nota 9) (+) (BINKLEY, DAVIS, *et al.*, 2013) (nota 9) (+) (SHARAFI, SOH, *et al.*, 2012) (nota 8) (+) (SHARIF e MALETIC, 2010) (nota 8)
Utilizar palavras completas para compor identificadores	(−) (BUSE e WEIMER, 2008) (nota 8) (−) (BUSE e WEIMER, 2010) (nota 8) (+) (DEYOUNG e KAMPEN, 1979) (nota 5,95) (+) (JØRGENSEN, 1980) (nota 9) (+) (LAWRIE, MORRELL, *et al.*, 2006) (nota 9) (+) (LAWRIE, MORRELL, *et al.*, 2007) (nota 9)
Programação	
Evitar código comentado não utilizado	-
Programar de forma objetiva	-
Manter tamanho do código pequeno	(+) (DEYOUNG e KAMPEN, 1979) (nota 5,95) (×) (FEIGENSPAN, APEL, *et al.*, 2011) (nota 7,5) (−) (POSNETT, HINDLE e DEVANBU, 2011) (nota 7,5)

A junção das informações obtidas na empresa e na literatura técnica culminou na elaboração da versão 1 das diretrizes de codificação apresentadas na Tabela 4.13 a seguir. Algumas diretrizes foram agrupadas, outras divididas e outras reescritas, contudo a ideia inicial continua a mesma.

Tabela 4.13 – Diretrizes de codificação para legibilidade e compreensibilidade

ID	DIRETRIZES (Versão 1)	Percepção de Importância na Equipe (explícita) - Total 8	Percepção de Importância na Equipe (implícita) - Total 11	Evidência na Literatura – Impacto na Legibilidade e Compreensibilidade
Arquivos e Pastas				
1	Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo	1 (mentor)	-	-
2	Cada módulo deve ser composto por pelo menos um arquivo de definição e outro de implementação	1 (mentor)	-	-
3	Utilize a seguinte ordem de organização para arquivos de definição	1 (mentor) + 4	-	-
4	Utilize a seguinte ordem de organização para arquivos de implementação	1 (mentor) + 4	-	-
Layout				
5	A indentação deve ser feita de forma consistente	8	11	⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8) ⊕ (JøRGENSEN, 1980) (nota 9) ⊕ (MIARA, MUSSELMAN, *et al.*, 1983) (nota 9)
6	O uso de chaves para identificação de blocos deve ser consistente	1	10	-
7	O espaçamento entre operadores e operandos deve ser feito de forma consistente (1 espaço)	2	-	-
8	Utilize parênteses para delimitar operandos em expressões	1	-	⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8)
9	Utilize linhas em branco para separar instruções extensas das	4	7	⊕ (BUSE e WEIMER, 2008) (nota 8)

	demais instruções e para separar blocos de instruções de diferentes propósitos			⊕ (BUSE e WEIMER, 2010) (nota 8) ⊕ (JØRGENSEN, 1980) (nota 9)
10	Faça apenas uma declaração por linha	7	11	-
11	Escreva apenas uma instrução simples por linha	7	11	-
12	Tamanho de linha deve ser menor que 80 caracteres	3	5	⊕ (BUSE e WEIMER, 2008) (nota 8) ⊕ (BUSE e WEIMER, 2010) (nota 8)
13	Variáveis devem ser declaradas em conjunto e no início do seu escopo válido	4	-	⊖ (SASAKI, HIGO e KUSUMOTO, 2013) (nota 8)
14	Constantes devem ser declaradas em conjunto e no início do seu escopo válido	4	-	⊖ (SASAKI, HIGO e KUSUMOTO, 2013) (nota 8)
Comentários				
15	Selecione um idioma (Inglês ou Português) no qual será redigido os comentários	8	2	-
16	Insira um comentário de identificação (cabeçalho) para cada arquivo	3	1	-
17	Insira um comentário de sumarização do propósito de cada função	7	4	⊕ (BUSE e WEIMER, 2008) (nota 8) ⊕ (BUSE e WEIMER, 2010) (nota 8) ⊕ (DEYOUNG e KAMPEN, 1979) (nota 5,95) ⊕ (JØRGENSEN, 1980) (nota 9) ⊕ (STEIDL, HUMMEL e JUERGENS, 2013) (nota 8) ⊕ (TENNY, 1988) (nota 6) ⊕ (WOODFIELD, DUNSMORE e SHEN, 1981) (nota 8)
18	Comentários de linha devem ser utilizados somente em casos específicos	1	-	⊕ (STEIDL, HUMMEL e JUERGENS, 2013) (nota 8)
Nomenclatura				
19	Selecione um idioma (Inglês ou Português) no qual será redigido o código	8	2	-

20	Os identificadores devem ser de fácil memorização	8	10	⊕ (LAWRIE, MORRELL, *et al.*, 2006) (nota 9) ⊕ (LAWRIE, MORRELL, *et al.*, 2007) (nota 9) ⊕ (TEASLEY, 1994) (nota 6)
21	Constantes e variáveis globais devem ser identificadas como tais	2	-	-
22	Os identificadores de variáveis devem ser consistentes quanto a sua forma de representação	8	9	⊕ (BINKLEY, DAVIS, *et al.*, 2009) (nota 9) ⊕ (BINKLEY, DAVIS, *et al.*, 2013) (nota 9) ⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8) ⊕ (DEYOUNG e KAMPEN, 1979) (nota 5,95) ⊕ (JØRGENSEN, 1980) (nota 9) ⊕ (LAWRIE, MORRELL, *et al.*, 2006) (nota 9) ⊕ (LAWRIE, MORRELL, *et al.*, 2007) (nota 9) ⊕ (SHARAFI, SOH, *et al.*, 2012) (nota 8) ⊕ (SHARIF e MALETIC, 2010) (nota 8)
23	Os identificadores de constantes devem ser consistentes quanto a sua forma de representação	8	9	⊕ (BINKLEY, DAVIS, *et al.*, 2009) (nota 9) ⊕ (BINKLEY, DAVIS, *et al.*, 2013) (nota 9) ⊖ (BUSE e WEIMER, 2008) (nota 8) ⊖ (BUSE e WEIMER, 2010) (nota 8) ⊕ (DEYOUNG e KAMPEN, 1979) (nota 5,95) ⊕ (JØRGENSEN, 1980) (nota 9) ⊕ (LAWRIE, MORRELL, *et al.*, 2006)

				(nota 9) (LAWRIE, MORRELL, *et al.*, 2007) (nota 9) (SHARAFI, SOH, *et al.*, 2012) (nota 8) (SHARIF e MALETIC, 2010) (nota 8)
24	As constantes utilizadas devem ser simbólicas	2	9	-
Orientações de Programação				
-	Evitar código comentado que não é mais utilizado	1	-	-
-	Atentar para o tamanho da solução (em código) dado ao problema, isso pode ser um indicativo de que ou o problema não está bem estruturado ou a complexidade estrutural da solução é alta. Em blocos, funções e arquivos	3	-	(DEYOUNG e KAMPEN, 1979) (nota 5,95) (FEIGENSPAN, APEL, *et al.*, 2011) (nota 7,5) (POSNETT, HINDLE e DEVANBU, 2011) (nota 7,5)

A Tabela 4.13 apresenta apenas o título das diretrizes elaboradas, cada uma delas, com exceção das de programação, é composta ainda por: i) uma explicação adicional, caso o título não seja suficiente para o seu entendimento; ii) instrução referente à diretriz no AStyle[11] ; características de qualidade que (iii) são e que (iv) podem ser influenciadas pela diretriz, dentre as seguintes: legibilidade, compreensibilidade, manutenibilidade, reutilização e integridade; v) referência para evidências experimentais na literatura técnica que indicam, mesmo que indiretamente, o uso da diretriz para promover a característica de qualidade em questão, bem como vi) o número de profissionais da organização que identificaram a diretriz como importante; vii) uma justificativa para a existência da diretriz com base no contexto organizacional e levando em consideração às características de qualidade de código fonte identificadas; viii) um exemplo, contra exemplo e/ou *template* para a diretriz.

É importante destacar que embora tenhamos focado nas características de legibilidade e compreensibilidade de código fonte, percebemos que algumas das diretrizes poderiam também impactar outras características de qualidade. Como não tínhamos dados que pudessem comprovar essa percepção (que também foi compartilhada por alguns programadores da empresa), utilizamos o item (iv) para elencar as características de qualidade que poderiam ser afetadas pelas diretrizes. Esse registro é importante para caso se queira fazer uma nova avaliação empírica dos atributos de código fonte ou mesmo das diretrizes de codificação, permitindo avaliar o impacto desses em outras características de qualidade além da legibilidade e compreensibilidade de código fonte.

Em relação às diretrizes de programação, decidimos não caracterizá-las como sendo de fato diretrizes e sim orientações, já que entendemos que certos atributos do código, como tamanho e até mesmo código comentado, não é algo que se pode padronizar ou definir limites; certos aspectos de programação são muito dependentes da natureza do problema tratado. Todos os detalhes referentes às diretrizes e orientações finais podem ser vistos no APÊNDICE D – Diretrizes de Codificação para Legibilidade e Compreensibilidade de Código Fonte.

[11] O AStyle é um formatador de código fonte que utiliza comandos configuráveis para formatar o código de acordo com as necessidades do usuário. Em várias diretrizes (de *layout*) foram identificados comandos do *AStyle* possíveis de serem utilizados em códigos fonte para fazer com que estes fiquem de acordo com o que é descrito pela diretriz.

4.4 Ameaças à Validade do Estudo

Algumas considerações precisam ser feitas em relação ao estudo de observação realizado. Primeiro, não conseguimos capturar informações de todos os participantes da organização, como planejado. Muitos participantes, embora tenham participado do anúncio da tarefa, não tiveram disponibilidade para a realização da tarefa em si. Segundo, nem todos os participantes que entregaram os códigos, entregaram o formulário de percepção da legibilidade e compreensibilidade de código fonte, mesmo com o prazo estendido. Esses fatos acabam prejudicando a análise dos dados e limitando as ações a serem tomadas na organização, passando a direcionar o trabalho para setores de desenvolvimento mais específicos dentro da empresa. Fato que ocorreu com as diretrizes elaboradas, que foram parametrizadas para softwares escritos em C/C++.

Outra ameaça à validade do estudo relaciona-se com a própria agregação dos dados. Era inviável tratar os dados coletados isoladamente, assim tivemos que lançar mão de estratégias para agrupar as informações e, assim, obter dados com os quais pudéssemos trabalhar. Invariavelmente dados importantes acabaram sendo perdidos nesse processo de agregação. Como exemplo mais evidente desse fato, temos a própria análise e agregação dos resultados de impacto dos atributos de código fonte identificados com a busca estruturada. Como forma de evidenciar o impacto das diretrizes na legibilidade e compreensibilidade de código, unimos informações de diferentes artigos que, apesar de tratarem de atributos parecidos com os das diretrizes, não eram iguais. Além disso, entendemos que os resultados dos estudos foram de certo modo específicos para o contexto em que foram realizados, porém a utilização e agregação somente de dados de mesmo contexto iria inviabilizar todo o processo de agregação de evidências e impossibilitaria a inferência do impacto que determinadas diretrizes poderiam ter na legibilidade e compreensibilidade de código fonte.

4.5 Considerações sobre o Capítulo

Este capítulo apresentou o plano de captura de código e percepção de legibilidade e compreensibilidade dos programadores da empresa Alfa, além da análise dos dados obtidos com o referido estudo de observação. Foi mostrado, ainda, como os atributos da literatura foram utilizados em conjunto com os dados coletados na organização para a formulação da versão 1 das diretrizes de codificação baseadas em evidência e específicas para a empresa Alfa.

A lista de diretrizes apresentada na Tabela 4.13 resume os dados coletados com a busca estruturada e o estudo de observação. Perguntas objetivas e subjetivas sobre legibilidade e compreensibilidade de código fonte possibilitaram a extração da percepção explícita dos programadores a respeito da importância das diretrizes no contexto da empresa Alfa. A análise dos códigos subsidiou a extração da percepção implícita da importância das diretrizes para a organização. Já os dados da literatura serviram como base para a avaliação de impacto das diretrizes na legibilidade e compreensibilidade de código fonte.

Como é possível perceber, de acordo com os estudos selecionados, nem todas as diretrizes impactam positivamente às características de qualidade avaliadas. Além disso, nem todas as diretrizes são consenso entre os diferentes programadores da organização, algumas sendo elencadas por apenas um programador da empresa. Esses dados enfatizam a necessidade de primeiramente avaliar a pertinência dessas diretrizes na organização para posteriormente realizar a tomada de ação na empresa, que seria a própria implantação das diretrizes. O capítulo seguinte explica e exibe os dados da avaliação das diretrizes de codificação na empresa Alfa.

5 *Focus Group* de Avaliação de Pertinência das Diretrizes de Codificação

Neste capítulo apresentamos a avaliação realizada através de um focus group das diretrizes de codificação formuladas. Das 24 diretrizes de codificação, foram selecionadas 10 diretrizes para terem sua pertinência para o contexto organizacional avaliada. A avaliação realizada culminou no ajuste de algumas diretrizes e na expansão de outras, resultando na versão final de diretrizes de codificação para legibilidade e compreensibilidade de código fonte para a empresa Alfa.

5.1 Introdução

Como vimos no capítulo anterior, nem todas as diretrizes de codificação foram mencionadas pelos participantes do estudo (ver coluna de percepção explícita de importância da Tabela 4.13). Enfatizamos que essa percepção explícita capturada foi obtida com a análise de respostas a questões tanto fechadas quanto abertas em relação a aspectos/atributos de código fonte que contribuíam ou prejudicavam a legibilidade e compreensibilidade do código. Assim, não necessariamente uma diretriz foi identificada como não importante, ela pode apenas não ter sido mencionada nas respostas. Em todo caso, era importante avaliar se essas diretrizes, não consensuais, eram de fato pertinentes no contexto de desenvolvimento da empresa Alfa.

As seções a seguir detalham o planejamento, a execução e a análise dos resultados obtidos com a avaliação, realizada através de um *focus group*, de um conjunto específico de diretrizes. Destacamos o uso do *focus group* para realizar essa avaliação, pois quisemos uma estratégia de avaliação que:

1. pudesse também servir como instrumento de divulgação das diretrizes;
2. permitisse a troca de experiências de codificação entre os participantes e, assim, facilitasse a captura de informações de contexto que indicassem a aplicabilidade das diretrizes em cenários específicos;
3. possibilitasse a captura de informações para subsidiar a criação de novas diretrizes de codificação aplicados para o contexto da organização.

5.2 Visão Geral sobre *Focus Group*

O *focus group* é um tipo de estratégia de dinâmica em grupo que possibilita a discussão objetiva, conduzida ou moderada para introdução de um tópico a um grupo

de respondentes, direcionando a discussão sobre o tema de uma maneira não estruturada e natural (PARASURAMAN, GREWAL e KRISHNAN, 2006). Para Kitzinger e Barbour qualquer discussão pode ser chamada de *focus group* desde que o pesquisador encoraje e se preocupe com a interação do grupo (KITZINGER e BARBOUR, 1999) – por interação do grupo entende-se interação entre os participantes e não interação exclusivamente com o pesquisador/moderador da dinâmica, como ocorre em entrevistas, por exemplo.

Um dos usos mais comuns do *focus group* é para capturar informações durante a fase exploratória de uma pesquisa (BARBOUR, 2008). Identificação de conceitos e ideias; coleta e priorização de problemas e descoberta de motivações em relação a um tópico de pesquisa são algumas dessas informações possíveis de serem capturadas através do *focus group*. Apesar desse uso mais exploratório, é possível utilizar essa estratégia para obter informações e testar hipóteses sobre objetos de estudo de pesquisa (KONTIO, BRAGGE e LEHTOLA, 2008), podendo servir também como uma estratégia de avaliação inicial de uma tecnologia.

O processo de condução do *focus group* perpassa muitos passos já previstos no processo geral de experimentação para estudos primários, como a definição do problema e a análise e reporte de resultados (ver Figura 5.1); porém algumas particularidades são específicas dessa metodologia, como a seleção dos participantes e moderadores e a segmentação dos participantes em diferentes grupos de *focus group*. Esse último passo é particularmente importante para garantir a existência de grupos coesos, consistentes e que não ultrapasse o tamanho máximo recomendado (8-12 participantes). Como a ideia base do *focus group* é a interação entre os participantes, grupos com características muito distintas ou muito grandes podem prejudicar a interação e a própria captura de informações. Também como forma de não prejudicar a interação e a captura de informações é importante que as seções de *focus group* não sejam muito longas, sendo no máximo de 2-3 horas.

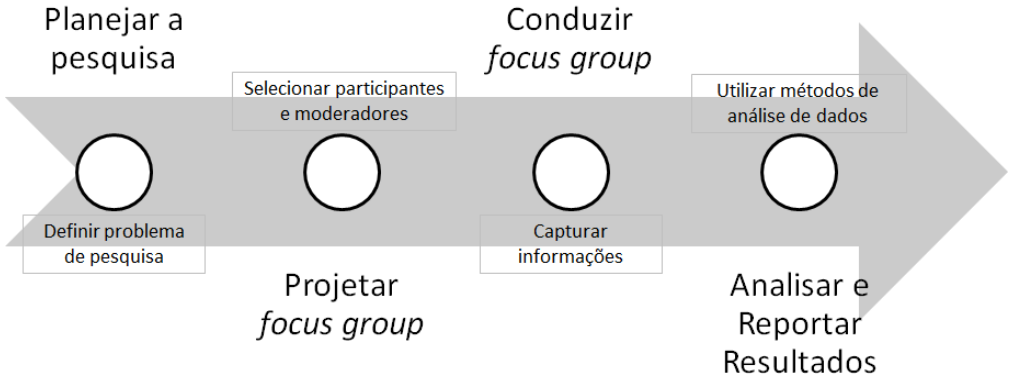


Figura 5.1 – Passos de um *focus group* por (KONTIO, BRAGGE e LEHTOLA, 2008)

5.3 Planejamento do *Focus Group* de Avaliação de Pertinência das Diretrizes

O objetivo do *focus group* é essencialmente o de avaliar algumas diretrizes de codificação formuladas que não temos confiança suficiente sobre sua pertinência para a organização. Utilizando a estrutura GQM, o objetivo é como segue:

- ***Analisar*** *diretrizes de codificação para legibilidade e compreensibilidade* ***com o propósito de*** *avaliar* ***com respeito à*** *pertinência e parametrização de uso na empresa Alfa* ***do ponto de vista de*** *programadores de firmware, software embarcado e software da empresa Alfa* ***no contexto de*** *desenvolvimento de firmware, software embarcado e software da empresa Alfa.*

Cada diretriz formulada pode ser parametrizável de acordo com o ambiente de desenvolvimento da organização, dependente da linguagem de programação utilizada ou ainda das necessidades de informação dos programadores. Por exemplo, a diretriz 1 (utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo), apesar de possuir um título genérico, a depender da linguagem de programação utilizada, requer uma explicação diferenciada. As extensões de arquivos de códigos escritos em C/C++ são distintas das de códigos escritos em PHP, por exemplo. Assim, essas parametrizações também devem ser avaliadas junto com as diretrizes propriamente ditas.

5.3.1 Seleção das Diretrizes a Serem Avaliadas

Para a sessão de *focus group*, precisávamos escolher um conjunto restrito de diretrizes a serem avaliadas, até mesmo para não tornar a sessão muito longa e pouco produtiva. Das 24 diretrizes mais 2 orientações, queríamos selecionar no máximo 10 diretrizes a serem discutidas. Inicialmente, planejamos avaliar diretrizes/orientações que obtiveram percepção de importância explícita inferior ou igual a 50% dos participantes do estudo (igual ou inferior a 4 programadores). Essa restrição acarretou na seleção das seguintes diretrizes listadas na Tabela 5.1:

Tabela 5.1 – Diretrizes de codificação com indicações explícitas de importância iguais ou inferiores a 4

ID	DIRETRIZES (Versão 1)
1	Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo
2	Cada módulo deve ser composto por pelo menos um arquivo de definição e outro de implementação
6	O uso de chaves para identificação de blocos deve ser consistente
7	O espaçamento entre operadores e operandos deve ser feito de forma consistente (1

	espaço)
8	Utilize parênteses para delimitar operandos em expressões
9	Utilize linhas em branco para separar instruções extensas das demais instruções e para separar blocos de instruções de diferentes propósitos
12	Tamanho de linha deve ser menor que 80 caracteres
13	Variáveis devem ser declaradas em conjunto e no início do seu escopo válido
14	Constantes devem ser declaradas em conjunto e no início do seu escopo válido
16	Insira um comentário de identificação (cabeçalho) para cada arquivo
18	Comentários de linha devem ser utilizados somente em casos específicos
21	Constantes e variáveis globais devem ser identificadas como tais
24	As constantes utilizadas devem ser simbólicas
-	Evitar código comentado que não é mais utilizado
-	Atentar para o tamanho da solução (em código) dado ao problema, isso pode ser um indicativo de que ou o problema não está bem estruturado ou a complexidade estrutural da solução é alta. Em blocos, funções e arquivos

Das 15 diretrizes/orientações selecionadas, três (9, 12 e 18) possuíam estudos que indicavam seu impacto positivo na legibilidade e/ou compreensibilidade de código fonte, sendo que duas (9 e 12) tinham mais de 35% de programadores indicando sua importância de forma explícita e mais de 45% de programadores indicando sua importância de forma implícita. Retiramos essas 2 diretrizes (9 e 12) do conjunto de diretrizes a serem avaliadas, juntamente com a diretriz 2 (vinculada a diretriz 1 que já seria avaliada) e com as 2 orientações de programação, que entendemos não serem críticas por se tratarem de orientações e não de diretrizes propriamente ditas. Assim, o grupo final de diretrizes a serem avaliadas é formado pelas diretrizes: 1, 6, 7, 8, 13, 14, 16, 18, 21 e 24, totalizando 10 diretrizes.

5.3.2 Seleção dos Participantes e Moderadores e Projeto do *Focus Group*

A avaliação das diretrizes deve ser realizada pelos programadores dos setores de desenvolvimento que tiveram participantes no estudo de captura de código fonte e que pensamos ter disponibilidade para participar da avaliação, a saber: setor de desenvolvimento de firmware e software embarcado para o domínio de rastreamento e setor de desenvolvimento de sistemas de informação, totalizando 10 programadores. Os moderadores do *focus group* devem ser: a mestranda Talita Ribeiro e o doutor e professor Guilherme Travassos.

O *focus group* deve ser realizado em uma única sessão com todos os 10 programadores, divididos em dois grupos, um para cada setor de desenvolvimento. Cada diretriz deverá ser avaliada individualmente de acordo com a sua aplicação no

contexto de desenvolvimento dos programadores em questão. A Tabela 5.2 a seguir ilustra o quadro a ser montado na sala de realização do *focus group*.

Tabela 5.2 – Protótipo do quadro a ser utilizado no *focus group*

	*Não se aplica a **software embarcado/sistemas de informação**, porque...*	*Se aplica a **software embarcado/sistemas de informação** e...*		
		Não traz benefícios nos casos em que...	*Auxilia nos casos em que...*	*Pode ajudar se...*
(1) Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo				
(6) O uso de chaves para identificação de blocos deve ser consistente				
(7) O espaçamento entre operadores e operandos deve ser feito de forma consistente (1 espaço)				
(8) Utilize parênteses para delimitar operandos em expressões				
(13) Variáveis devem ser declaradas em conjunto e no início do seu escopo válido				
(14) Constantes devem ser declaradas em conjunto e no início do seu escopo válido				
(16) Insira um comentário de identificação (cabeçalho) para cada arquivo				
(18) Comentários de linha devem ser utilizados somente em casos específicos				
(21) Constantes e variáveis globais devem ser identificadas como tais				
(24) As constantes utilizadas devem ser simbólicas				

Dois quadros diferentes (escritos à mão em cartolina) montados na sala de realização do *focus group* irão apoiar a coleta das informações, um específico para o contexto de desenvolvimento de firmware e software embarcado (simplificado para somente software embarcado) e outro para o de sistemas de informação. Os quadros

devem ser montados sem as diretrizes inicialmente, estas devendo ser inseridas somente no momento de sua avaliação. Para cada diretriz a ser avaliada, seu título deve ser inserido no quadro e a sua especificação completa exibida na projeção para que todos possam ter acesso aos detalhes.

A avaliação de cada diretriz deve ocorrer como segue:

1. Os moderadores inserem nos quadros a diretriz a ser avaliada e apresentam a sua especificação completa no projetor;
2. Os participantes tem um momento para discutir entre si sobre a diretriz;
3. Cada participante escreve sua opinião sobre a diretriz, informando em um *post-it*:
    a. Rosa: o por quê de ela não se aplicar ao seu contexto de desenvolvimento;
    b. Amarelo: em que casos excepcionais ela não se aplica ao seu contexto de desenvolvimento;
    c. Verde: em que casos ela se aplica ao seu contexto de desenvolvimento sem restrições;
    d. Azul: como ela poderia ser melhorada/alterada para se adequar ainda mais ao seu contexto de desenvolvimento.
4. Após escritas as opiniões, os *post-it* são colados em suas respectivas posições (conforme o planejamento do quadro).
5. Ao final de uma rodada de avaliação de 3 a 5 diretrizes (dependendo das discussões a serem realizadas) os participantes tem a oportunidade de levantar para analisar as demais opiniões escritas, indicando concordância ou não com as opiniões.

Informações adicionais sobre as discussões das diretrizes devem, ainda, ser registradas pelos moderados do *focus group*.

5.4 Execução e Análise dos Resultados

A sessão de *focus group* ocorreu no dia 16/07/2014 e teve 2 horas e 40 minutos de duração. Participaram 4 programadores do setor de desenvolvimento de firmware e software embarcado e 2 do domínio de sistemas de informação. Demais membros da equipe estavam sem disponibilidade para participar da avaliação no dia em questão.

A condução do *focus group* ocorreu como o planejado. A Figura 5.2 apresenta o registro da organização do local onde foi conduzido o *focus group*.

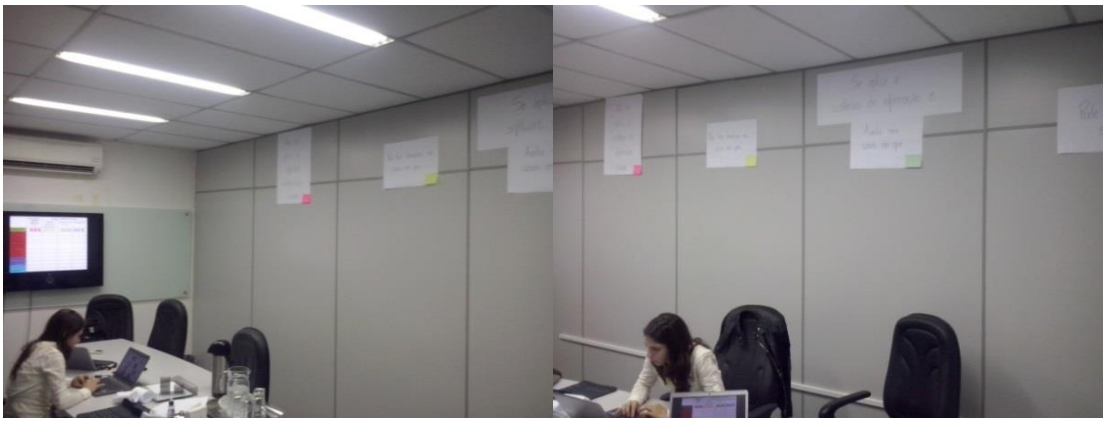


Figura 5.2 – Organização do local de execução do *focus group.* Foto da direita apresentando o lado de software embarcado e a da esquerda, de sistemas de informação

A cada diretriz exposta para avaliação, os participantes comentavam entre si sobre sua aplicação no ambiente de desenvolvimento de cada um. Houve interação não somente entre participantes do mesmo grupo, mas também entre participantes de grupos diferentes, geralmente para esclarecer dúvidas de aplicação da diretriz ou mesmo para explicar casos em que a falta da diretriz causou problemas de legibilidade e compreensibilidade de código fonte.

De modo geral, todas as diretrizes receberam *post-it* informando que elas eram aplicáveis ao contexto de desenvolvimento dos participantes, somente uma diretriz (1) foi indicada por um participante como não aplicável ao contexto desenvolvimento de sistemas de informação. Um dos participantes do grupo de sistemas de informação indicou que, para códigos escritos em PHP, a diretriz 1 não se aplicava, já que a linguagem PHP é escrita juntamente com HTML. É importante destacar, no entanto, que a diretriz foi formulada justamente para separar esses tipos de informações, já que acreditasse que isso prejudica a reutilização e mesmo a manutenibilidade do código fonte, além, é claro, da legibilidade, uma vez que dificulta a leitura e o acesso a informações específicas do código.

Além do único *post-it* rosa, o *focus group* resultou também em 11 amarelos, 40 verdes e 22 azuis no total; sendo 7 amarelos, 30 verdes e 16 azuis das avaliações dos participantes do grupo de software embarcado. Os *post-it* verde foram utilizados mais para enfatizar a importância das diretrizes e de seu impacto para o código fonte. Os *post-it* que trouxeram mais benefícios sob o ponto de vista de melhoria e parametrização das diretrizes foram os amarelos, mas, especialmente, os azuis. A Figura 5.3 registra o momento de discussão entre os envolvidos no estudo, bem como o estado de preenchimento dos quadros para as primeiras 6 diretrizes avaliadas.

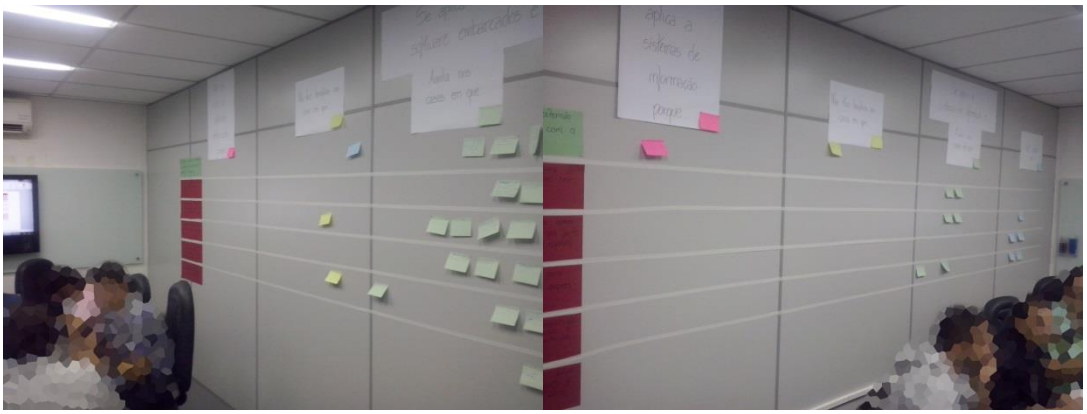


Figura 5.3 – Registro da participação dos programadores no *focus group*

A maioria dos *post-it* azuis indicaram melhorias nas diretrizes sob o ponto de vista de sua expansão. Como exemplo disso tem-se a diretriz 7 (O espaçamento entre operadores e operandos deve ser feito de forma consistente (1 espaço)), para a qual muitas oportunidades de melhoria foram elencadas como forma de expandir a diretriz de espaçamento para outros elementos do programa. Outro exemplo foi a diretriz 21 (Constantes e variáveis globais devem ser identificadas como tais), muitos programadores levantaram problemas relacionados com a não distinção de funções recursivas de outras funções, além também dos problemas que já ocorreram em diferentes projetos da empresa por não se ter um demarcador da origem (arquivo) das funções públicas, principalmente em código C.

Post-it amarelos indicaram basicamente restrições a algumas diretrizes. Como exemplo tem-se a diretriz 1 (Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo), para qual os programadores levantaram restrições em relação a usar extensões .inc de C para tabelas, já que isso prejudicava a referência e busca de informações através da IDE institucionalizada na organização. Outro exemplo de restrição foi a apresentada para a diretriz 24 (As constantes utilizadas devem ser simbólicas), apesar dos programadores terem notado esse fato como algo prejudicial à compreensão do código, muitos indicaram que constantes simbólicas devem ser utilizadas somente em casos em que o número realmente tivesse um significado fora daquele contexto de uso, como é o caso de uma constante para TIME_OUT, por exemplo, ou ainda PI e RAIO_TERRESTRE. Nos demais casos em que as constantes forem simples e pontuais, um comentário de comando deve ser inserido para explicar a lógica da expressão. A listagem com as diretrizes finais pós-*focus group* é dada pela Tabela 5.3 a seguir (atentar para a nova numeração das diretrizes, diferente da utilizada na versão 1).

Tabela 5.3 – Diretrizes finais pós-*focus group*

ID	DIRETRIZES FINAIS
	ARQUIVOS E PASTAS
1	Organize as pastas e arquivos do projeto de forma coesa
2	Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo
3	Cada módulo deve ser composto por pelo menos um arquivo de definição e outro de implementação
4	Utilize a seguinte ordem de organização para arquivos de definição
5	Utilize a seguinte ordem de organização para arquivos de implementação
	LAYOUT
6	A indentação deve ser feita de forma consistente
7	Quebre e utilize indentação em expressões lógicas com 2 ou mais operadores não iguais
8	O uso de chaves para identificação de blocos deve ser consistente, respeitando o estilo 2TBS
9	Atente para o espaçamento consistente (1 espaço) entre diferentes estruturas sintáticas e elementos do programa
10	Utilize parênteses para delimitar operandos em expressões
11	Utilize linhas em branco para separar instruções extensas das demais instruções e para separar blocos de instruções de diferentes propósitos
12	Faça apenas uma declaração por linha
13	Escreva apenas uma instrução simples por linha
14	Tamanho de linha deve ser menor que 80 caracteres
15	Variáveis devem ser declaradas em conjunto e no início do seu escopo válido
16	Constantes devem ser declaradas em conjunto e no início do seu escopo válido
	COMENTÁRIO
17	Selecione um idioma (Inglês ou Português) no qual será redigido os comentários
18	Insira um comentário de identificação (cabeçalho) para cada arquivo
19	Insira um comentário de sumarização do propósito de cada função
20	Comentários de linha devem ser utilizados somente em casos específicos
	NOMENCLATURA
21	Selecione um idioma (Inglês ou Português) no qual será redigido o código
22	Os identificadores devem ser de fácil memorização
23	Atentar para o uso adequado de prefixos e sufixos em elementos do programa
24	Os identificadores dos elementos do programa devem ser consistentes quanto a sua forma de representação
25	As constantes utilizadas devem ser simbólicas
	ORIENTAÇÕES DE PROGRAMAÇÃO
-	Evitar o uso de goto
-	Evitar o uso de operadores ternários (“?”) quando a expressão ultrapassar o limite de tamanho de linha
-	Evitar código comentado que não é mais utilizado
-	Atentar para o tamanho da solução (em código) dado ao problema, isso pode ser um indicativo de que ou o problema não está bem estruturado ou a complexidade estrutural da solução é alta. Em blocos, funções e arquivos

Em destaque estão as diretrizes que foram inseridas ou alteradas. Algumas diretrizes foram agrupadas, porém sendo transformadas em diretrizes mais abrangentes, como o caso das antigas relacionadas com identificadores de variáveis e constantes que foram agrupadas em uma única genérica para identificadores. A diretriz 1 e duas orientações adicionais de programação também surgiram durante as discussões do *focus group*, isso porque no tempo final da sessão, discussões relacionadas com outras diretrizes não expostas para avaliação acabaram surgindo também. Detalhes das diretrizes finais no APÊNDICE D – Diretrizes de Codificação para Legibilidade e Compreensibilidade de Código Fonte.

5.5 Ameaças à Validade do Estudo

Algumas situações que ocorreram ao longo do planejamento e execução do *focus group* tornaram-se ameaças à validade do estudo. O grupo de desenvolvedores participantes desse estudo foi consideravelmente menor que o grupo de participantes do *survey*. Isso foi devido aos novos direcionamentos do projeto ao longo do tempo – priorização do setor de desenvolvimento de rastreamento – e também devido à indisponibilidade de alguns programadores durante o período de realização das atividades dos diferentes estudos conduzidos. Esse fato pode ter influenciado o resultado final de avaliação das diretrizes, possivelmente não representando a visão de toda a organização.

Outro ocorrido foi em relação à distribuição do documento de diretrizes para os programadores antes da seção de avaliação. Planejamos exibir as diretrizes de codificação aos programadores somente no momento da avaliação como forma de capturar a primeira impressão dos programadores sobre a aplicabilidade das diretrizes, contudo tivemos o conhecimento de que parte dos programadores que participaram da avaliação acabou tendo acesso ao documento antes da realização do *focus group*. Nenhum programador do setor de sistemas de informação teve acesso ao documento de diretrizes. Esse fato pode ter influenciado a participação diferenciada entre o grupo de programadores que leram o documento – programadores do setor de rastreamento – e o grupo de programadores que não leram o documento – programadores do setor de sistemas de informação. Os que leram o documento previamente pareceram concordar mais entre si sobre as opiniões levantadas para as diretrizes, como se já houvessem conversado antes sobre o assunto.

5.6 Considerações sobre o Capítulo

Apresentamos nesse capítulo o planejamento e resultados da avaliação das diretrizes de codificação para legibilidade e compreensibilidade de código fonte na empresa Alfa. Vimos que todas as diretrizes avaliadas foram identificadas como aplicáveis ao contexto de desenvolvimento da empresa Alfa, algumas sofrendo expansão para atender mais casos na organização e outras sofrendo limitações.

É importante enfatizar que a sessão de *focus group* tinha o objetivo de avaliar a pertinência das diretrizes, até então não consensuais, para o contexto de desenvolvimento de software da empresa Alfa. Algumas diretrizes precisavam ainda de evidências em relação ao seu impacto positivo na legibilidade e/ou compreensibilidade de código fonte. Além disso, outras diretrizes que apresentaram dados de impactos controversos na literatura técnica também precisavam ser avaliadas novamente. Esses dados revelam a necessidade de novos estudos de avaliação, agora experimentais, das diretrizes em questão.

Destacamos o fato de termos configurado as diretrizes finais para o contexto de desenvolvimento de firmware e software embarcado (C/C++), devido à priorização dada a esse setor de desenvolvimento pela alta gerência da empresa e também devido à discordância em alguns aspectos das diretrizes entre programadores do setor de desenvolvimento de sistemas de informação. Esse último fato foi provavelmente ocasionado pela baixa experiência dos programadores que participaram do *focus group*, já que o programador de sistemas de informação com mais tempo de indústria e de empresa Alfa estava indisponível para participar da sessão de avaliação realizada. Em todo caso, é ainda esperado que as diretrizes formuladas sejam configuradas para o setor de desenvolvimento de sistemas de informação após uma avaliação junto ao programador sênior desse setor.

6 Conclusão e Trabalhos Futuros

Neste capítulo, as conclusões desta dissertação são apresentadas, resumindo sua motivação e proposta, destacando suas contribuições e explicitando as limitações. Ao final listamos um conjunto de trabalhos futuros possíveis de serem realizados para dar continuidade à pesquisa aqui apresentada.

6.1 Considerações Finais

Esta dissertação apresentou os resultados da pesquisa realizada no contexto de desenvolvimento de firmware, software embarcado e software da empresa Alfa. A partir de um diagnóstico no qual se identificou um desalinhamento de conceitos a respeito de refatoração de código fonte e de expectativas de qualidade em código fonte, um conjunto de atividades foi desenvolvido, com base científica, para auxiliar a organização a: i) alinhar as perspectivas de qualidade de código existentes e, assim, ii) agregar qualidade aos códigos da organização e iii) diminuir o esforço até então dispendido em atividades de retrabalho (reconstrução).

Além dos estudos conduzidos na organização, que por si só contribuíram com o diagnóstico interno do contexto de desenvolvimento de software da empresa Alfa e auxiliaram na internalização de conceitos sobre refatoração, reconstrução e qualidade de código fonte, diretrizes de codificação focadas para mitigar a real causa do problema de retrabalho existente na organização foram formuladas. As seções seguintes detalham as contribuições, limitações e trabalhos futuros resultantes deste trabalho de pesquisa.

6.2 Contribuições

Do ponto de vista científico, vemos que a pesquisa conduzida serviu como indicador de que existem de fato desalinhamentos de conceitos e definições não só na academia, mas também entre academia-indústria e, ainda, dentro de um mesmo contexto organizacional; e que esses desalinhamentos podem tanto acarretar problemas de comunicação entre desenvolvedores de uma organização como prejudicar a busca por soluções tecnológicas para mitigar as reais causas dos problemas existentes. Assim, o conjunto de estudos experimentais utilizados, bem como a explicitação dos critérios para a tomada de decisão sobre sua utilização são contribuições deste trabalho, podendo servir como mecanismos de apoio à formulação

de estudos adicionais com propósitos semelhantes, ou seja, relacionados ao alinhamento de conceitos ou de perspectivas de qualidade.

Do ponto de vista de aplicação na indústria, acreditamos que a organização de diretrizes de codificação com base em evidência pode contribuir para dois propósitos gerais: i) fornecer aos programadores guias para agregar qualidade ao código fonte; e ii) auxiliar os inspetores no processo de controle da qualidade do código fonte (STAA, 2000). Em particular, as diretrizes de codificação com foco em legibilidade e compreensibilidade contribuem para aquelas organizações que possuem expectativas de qualidade de código fonte semelhante e pretendem guiar seu processo de codificação com base em evidência. Em um contexto mais específico – empresa Alfa – identificam-se duas contribuições adicionais: i) a diminuição esperada do esforço de reconstrução do código fonte devido ao realinhamento dos conceitos de refatoração e reconstrução e das perspectivas de qualidade de legibilidade e compressibilidade; e ii) o fornecimento de um critério explícito para verificar e controlar a qualidade do código fonte produzido ou solicitado pela empresa.

Em suma, podemos dividir ainda as contribuições deste trabalho em algumas perspectivas:

Contribuições Acadêmicas:

A revisão da literatura e a busca estruturada sobre o tema de legibilidade e compreensibilidade de código fonte, além de terem auxiliado a elaboração das diretrizes de codificação, fornecem um panorama geral sobre o que vem sendo produzido nessa área de pesquisa.

A utilização de diferentes estratégias de estudos experimentais em um contexto real de desenvolvimento de software contribuiu não só com a disponibilização dos protocolos de execução dos estudos – possíveis de serem aplicados ou ajustados em outros contextos de desenvolvimento –, mas também com a própria utilização e encadeamento dessas metodologias através da pesquisa-ação em engenharia de software; que auxiliou na: descoberta da causa de problemas (*survey* exploratório); elaboração de tecnologia com base científica (busca estruturada) e avaliação de tecnologia (*focus group*).

Contribuições para a Indústria:

As próprias diretrizes de codificação (conjunto inicial) se mostram como contribuição não somente para a empresa Alfa, mas também para organizações de desenvolvimento de software de modo geral.

Os benefícios advindos da formulação e aplicação das diretrizes de codificação na empresa Alfa também são contribuições deste trabalho, a saber: alinhamento das

perspectivas de qualidade em código fonte para legibilidade e compreensibilidade de código e diminuição dos riscos de reconstrução de código fonte.

A elaboração de instrumentos de disseminação das diretrizes dentro da organização, como *folder* e hiperdocumento, também é resultante deste trabalho.

6.3 Limitações

Por ser um trabalho que contempla diferentes estudos experimentais, todas as limitações dos estudos também se mostram como limitações do trabalho:

Os resultados do *survey* exploratório são limitados ao contexto de desenvolvimento da empresa Alfa, assim não podemos generalizar a priorização das diferentes características de qualidade de código fonte nem a própria existência de desalinhamento de perspectivas de qualidade de código fonte entre programadores. Contudo, olhar para esses aspectos em um diagnóstico organizacional, pode ser um promissor passo inicial para a descoberta de possíveis situações de reconstrução de código fonte que estejam ocorrendo.

A busca por atributos de código fonte foi limitada às características de legibilidade e compreensibilidade de código fonte, pois eram prioritárias para serem tratadas no contexto específico da empresa Alfa, fato que também limitou a elaboração das diretrizes para essas características de qualidade. Contudo, o protocolo elaborado poderá ser ajustado para a descoberta de novos atributos de código para outras características de qualidade de código.

A análise dos códigos fonte da organização foi limitada aos dados obtidos em dois setores específicos de desenvolvimento da empresa Alfa, fato que também limitou a elaboração das diretrizes de codificação. Apesar disso, vemos que um conjunto inicial de diretrizes já se mostra útil para servir como ponto de partida para discussão com demais programadores da empresa.

A não conclusão do ciclo completo de pesquisa-ação também é uma limitação do trabalho. Embora isso seja justificado pelo tempo do próprio mestrado e do projeto de cooperação e também pela complexidade do contexto de atuação da pesquisa: pesquisador e empresa separados geograficamente; diferentes equipes de desenvolvimento existentes na organização e também geograficamente dispersas; dificuldade de sincronização de agendas (disponibilidade de todos os envolvidos) para realização de tarefas.

6.4 Trabalhos Futuros

Considerando o contexto da empresa Alfa, e entendendo que o ciclo de pesquisa-ação deve ser continuado, vislumbramos:

1. avaliar experimentalmente diretrizes cujo impacto na legibilidade e compreensibilidade de código é desconhecido ou dúbio conforme a literatura técnica;
2. acompanhar a implantação e utilização das diretrizes de codificação; e
3. avaliar o impacto da implantação das diretrizes de codificação para legibilidade e compreensibilidade de código no esforço de reconstrução de código.

A evolução das diretrizes de codificação para outras características de qualidade de código fonte (i.e. desempenho, compatibilidade, confiabilidade, segurança, portabilidade dentre outros) seguindo o ciclo de pesquisa-ação e adotando as estratégias experimentais descritas neste trabalho representa um desafio de pesquisa que merece atenção.

Adicionalmente, a formulação de *frameworks* de auxílio à identificação de (des)alinhamento conceituais e de perspectivas de qualidade em código fonte, incluindo a organização de taxonomias para diretrizes e tipos de desalinhamento, pode contribuir para reduzir os riscos e o esforço necessário para o tratamento de questões semelhantes em larga escala, considerando as dificuldades normalmente envolvidas na análise qualitativa dos dados resultantes dos diferentes estudos experimentais que precisam ser, a princípio, realizados.

6.5 Considerações Adicionais

Como considerações finais deste trabalho, destacamos a cooperação academia-indústria que foi alcançada com a condução desta dissertação. Algumas informações importantes referentes a essa cooperação precisam ser destacadas:

- Muitas informações capturadas da organização, sejam de contexto ou ainda advindas do diagnóstico da empresa (*survey* e captura de código fonte), não puderam ser apresentadas no contexto desta dissertação devido a um acordo de confidencialidade assinado entre pesquisadores e empresa. Esse fato evidencia a necessidade de um acordo explícito entre os participantes de um estudo de pesquisa-ação como forma a garantir benefícios para ambas as partes envolvidas.
- Vimos no contexto do projeto de cooperação Grupo ESE-Empresa Alfa – que envolvia ações de melhoria em outras áreas da engenharia de software como gerência, requisitos e teste – que enquanto não tratávamos de assuntos com os quais os desenvolvedores tinham mais intimidade, a motivação e participação dos envolvidos nas ações de melhoria de qualidade não era tão intensa. Apesar da indisponibilidade de alguns programadores, as ações na

área de qualidade de código fonte foram muito proveitosas para ambas as partes, com participação e troca de experiências entre os envolvidos.

- Por fim, um item de destaque em relação à pesquisa realizada em parceria com a indústria foi o nível de aparente trivialidade dos problemas identificados; enfatizando que a indústria ainda enfrenta problemas que para muitos na academia já foram resolvidos. Isso nos leva a questionar a respeito da aplicabilidade das tecnologias e abordagens disponíveis para a engenharia de software e mesmo a respeito da disseminação da importância que determinadas práticas de engenharia de software tem para o processo de desenvolvimento de software como um todo.

Referências Bibliográficas

ALSHAYEB, M. Empirical Investigation of Refactoring Effect on Software Quality. **Information and Software Technology**, 51, n. 9, 2009. 1319-1326. DOI 10.1016/j.infsof.2009.04.002.

ANTONIOL, G. et al. Recovering Traceability Links between Code and Documentation. **IEEE Transactions on Software Engineering**, 28, n. 10, 2002. DOI 10.1109/TSE.2002.1041053.

BARBOUR, R. **Doing Focus Group**. London: SAGE Publications Ltd, 2008. 168 p. ISBN 978-0761949787.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The Goal Question Metric Approach. In: _____ **Encyclopedia of software engineering**. Hoboken: John Wiley & Sons, v. 2, 1994. p. 528-532.

BINKLEY, D. A. et al. The Impact of Identifier Style on Effort and Comprehension. **Empirical Software Engineering**, Hingham, 18, n. 2, 2013. 219-276. DOI 10.1007/s10664-012-9201-4.

BINKLEY, D. et al. **To Camelcase or Under_score**. Proceedings of the 17th IEEE International Conference on Program Comprehension. Vancouver: IEEE. 2009. p. 158-167.

BIOLCHINI, J. C. D. A. et al. Scientific Research Ontology to Support Systematic Review in Software Engineering. **Advanced Engineering Informatics**, Amsterdam, 21, n. 2, 2007. 133-151. DOI 10.1016/j.aei.2006.11.006.

BOIS, B. D. et al. **Does God Class Decomposition Affect Comprehensibility?** Proceedings of the IASTED International Conference on Software Engineering. Innsbruck: [s.n.]. 2006. p. 346-355.

BRUNTINK, M.; DEURSEN, A. V. An Empirical Study into Class Testability. **Journal of Systems and Software**, 79, n. 9, 2006. 1219-1232. DOI 10.1016/j.jss.2006.02.036.

BUSE, R. P. L.; WEIMER, W. R. **A Metric for Software Readability**. Proceedings of the 2008 International Symposium on Software Testing and Analysis. New York: ACM Press. 2008. p. 121-130. DOI 10.1145/1390630.1390647.

BUSE, R. P. L.; WEIMER, W. R. Learning a Metric for Code Readability. **IEEE Transactions on Software Engineering**, 36, n. 4, 2010. 546-558. DOI 10.1109/TSE.2009.70.

CORBO, F.; GROSSO, C. D.; PENTA, M. D. **Smart Formatter:** Learning Coding Style from Existing Source Code. Proceedings of the IEEE International Conference on Software Maintenance. Paris: IEEE. 2007. p. 525-526.

DANDASHI, F. **A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements**. Proceedings of the 2002 ACM Symposium on Applied Computing. New York: ACM Press. 2002. p. 997-1003. DOI 10.1145/508791.508985.

DEYOUNG, G. E.; KAMPEN, G. R. **Program Factors as Predictors of Program Readability**. Proceedings of the Third IEEE Computer Society's International Computer Software and Applications Conference. [S.l.]: IEEE. 1979. p. 668-673. DOI 10.1109/CMPSAC.1979.762579.

FEIGENSPAN, J. et al. **Exploring Software Measures to Assess Program Comprehension**. Proceedings of the International Symposium on Empirical Software Engineering and Measurement. Banff: IEEE. 2011. p. 127-136. DOI 10.1109/ESEM.2011.21.

FEIGENSPAN, J. et al. **Using Background Colors to Support Program Comprehension in Software Product Lines**. Proceedings of the 15th Annual Conference on Evaluation & Assessment in Software Engineering. Durham: IET. 2011. p. 66-75. DOI 10.1049/ic.2011.0008.

FOWLER, M. et al. **Refactoring:** Improving the Design of Existing Code. 1ª. ed. Boston: Addison-Wesley Longman Publishing Co., 1999. ISBN 0-201-48567-2.

GARVIN, D. What does “Product Quality” Really Mean? **Sloan Management Review**, 26, n. 1, 1984. 25-43.

GUERROUJ, L. **Automatic Derivation of Concepts Based on the Analysis of Source Code Identifiers**. Proceedings of the 2010 17th Working Conference on Reverse Engineering. Washington: IEEE Computer Society. 2010. p. 301-304. DOI 10.1109/WCRE.2010.45.

GUERROUJ, L. **Normalizing Source Code Vocabulary to Support Program Comprehension and Software Quality**. Proceedings of the 2013 35th International Conference on Software Engineering. San Francisco: IEEE. 2013. p. 1385-1388. DOI 10.1109/ICSE.2013.6606723.

HENDRIX, T. D. et al. **Do Visualizations Improve Program Comprehensibility? Experiments with Control Structure Diagrams for Java**. Proceedings of the

Thirty-First SIGCSE Technical Symposium on Computer Science Education. New York: ACM Press. 2000. p. 382-386. DOI 10.1145/331795.331890.

ISO/IEC. **ISO/IEC 9126-1 - Information Technology - Software Product Quality - Quality Model**. Geneva, p. 34. 2001.

ISO/IEC. **ISO/IEC 25010 - Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models**. Geneva, p. 43. 2011.

JøRGENSEN, A. H. A Methodology for Measuring the Readability and Modifiability of Computer Programs. **BIT Numerical Mathematics**, 20, 1980. 393-405. DOI 10.1007/BF01933633.

KASUNIC, M. **Designing an Effective Survey**. Carnegie Mellon University. Pittsburgh, p. 133. 2005. CMU/SEI-2005-HB-004.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. **A Field Study of Refactoring Challenges and Benefits**. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York: ACM Press. 2012. Article 50. DOI 10.1145/2393596.2393655.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. **Appendix to A Field Study of Refactoring Rationale, Benefits, and Challenges at Microsoft**. Microsoft Corporation. Redmond, p. 6. 2012. MSR-TR-2012-4.

KITCHENHAM, B.; PFLEEGER, S. L. Software Quality: The Elusive Target. **IEEE Software**, v. 1, n. 13, p. 12-21, January 1996.

KITZINGER, J.; BARBOUR, R. S. Introduction: The Challenge and Promise of Focus Groups. In: BARBOUR, R. S.; KITZINGER, J. **Developing Focus Group Research:** Politics, Theory and Practice. London: SAGE Publications Ltd, 1999. p. 1-20. ISBN 9781849208857. DOI 10.4135/9781849208857.

KONTIO, J.; BRAGGE, J.; LEHTOLA, L. The Focus Group Method as an Empirical Tool in Software Engineering. In: SHULL, F.; SINGER, J.; SJøBERG, D. I. K. **Guide to Advanced Empirical Software Engineering**. London: Springer-Verlag, 2008. Cap. 4, p. 93-116.

KREIMER, J. Adaptive Detection of Design Flaws. **Electronic Notes in Theoretical Computer Science**, Amsterdam, 141, n. 4, 2005. 117–136. DOI 10.1016/j.entcs.2005.02.059.

LAWRIE, D. et al. **What's in a Name? A Study of Identifiers**. Proceedings of the 14th IEEE International Conference on Program Comprehension. Athens: IEEE. 2006. p. 3-12. DOI 10.1109/ICPC.2006.51.

LAWRIE, D. et al. Effective Identifier Names for Comprehension and Memory. **Innovations in Systems and Software Engineering**, London, 3, n. 4, 2007. 303-318. DOI 10.1007/s11334-007-0031-2.

LOPES, V. P.; TRAVASSOS, G. H. **Experimentação em Engenharia de Software:** Glossário de Termos. Proceedings of the 6th Experimental Software Engineering Latin American Workshop. São Carlos: [s.n.]. 2009. p. 33-44.

MAFRA, S. N.; TRAVASSOS, G. H. **Estudos Primários e Secundários Apoiando a Busca por Evidências em Engenharia de Software**. Universidade Federal do Rio de Janeiro. Rio de Janeiro, p. 32. 2006. RT-ES 687/06.

MIARA, R. J. et al. Program Indentation and Comprehensibility. **Communications of the ACM**, New York, 26, n. 11, 1983. 861-867. DOI 10.1145/182.358437.

NOVAIS, R. et al. **On the Proactive and Interactive Visualization for Feature Evolution Comprehension:** An Industrial Investigation. Proceedings of the 34th International Conference on Software Engineering. Zurich: IEEE. 2012. p. 1044-1053. DOI 10.1109/ICSE.2012.6227115.

PAI, M. et al. Systematic Reviews and Meta-Analyses: An Shown, Step-by-Step Guide. **The National Medical Journal Of India**, 17, n. 2, 2004. 86-95.

PARASURAMAN, A.; GREWAL, D.; KRISHNAN, R. **Marketing Research**. Boston: Houghton Mifflin Company, 2006. 638 p. ISBN 9780618660643.

PINSONNEAULT, A.; KRAEMER, K. L. Survey Research Methodology in Management Information Systems: An Assessment. **Journal of Management Information Systems - Special Section: Strategic and Competitive Information Systems**, 10, n. 2, 1993. 75-105.

POSNETT, D.; HINDLE, A.; DEVANBU, P. **A Simpler Model of Software Readability**. Proceedings of the 8th Working Conference on Mining Software Repositories. New York: ACM Press. 2011. p. 73-82. DOI 10.1145/1985441.1985454.

POSNETT, D.; HINDLE, A.; DEVANBU, P. **A Simpler Model of Software Readability**. Proceedings of the 8th Working Conference on Mining Software Repositories. New York: ACM Press. 2011. p. 73-82. DOI 10.1145/1985441.1985454.

PRIBERAM INFORMÁTICA. compreensível, 2013. Disponivel em: <http://www.priberam.pt/dlpo/compreensível>. Acesso em: 11 Agosto 2014.

PRIBERAM INFORMÁTICA. legível. **Dicionário Priberam da Língua Portuguesa**, 2013. Disponivel em: <http://www.priberam.pt/dlpo/legível>. Acesso em: 11 Agosto 2014.

RAMBALLY, G. K. **The Influence of Color on Program Readability and Comprehensibility**. Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education. New York: ACM Press. 1986. p. 173-181.

RILLING, J.; MUDUR, S. P. 3D Visualization Techniques to Support Slicing-Based Program Comprehension. **Computers and Graphics**, New York, 29, n. 3, 2005. 311-329. DOI 10.1016/j.cag.2005.03.007.

SANTOS, P. S. M. D. **Uma Análise da Utilização de Pesquisa-Ação em Engenharia de Software**. Universidade Federal do Rio de Janeiro. Rio de Janeiro, p. 165. 2009.

SASAKI, Y.; HIGO, Y.; KUSUMOTO, S. **Reordering Program Statements for Improving Readability**. Proceedings of the 17th European Conference on Software Maintenance and Reengineering. Genova: IEEE. 2013. p. 361-364. DOI 10.1109/CSMR.2013.50.

SEI. **CMMI for Development (CMMI-DEV)**. Carnegie Mellon University. Pittsburgh, p. 482. 2010. (CMU/SEI-2010-TR-033). Versão 1.3.

SHARAFI, Z. et al. **Women and Men - Different but Equal:** on the Impact of Identifier Style on Source Code Reading. Proceedings of the IEEE 20th International Conference on Program Comprehension. Passau: IEEE. 2012. p. 27-36. DOI 10.1109/ICPC.2012.6240505.

SHARIF, B.; MALETIC, J. **An Eye Tracking Study on camelCase and under_score Identifier Styles**. Proceedings of the IEEE 18th International Conference on Program Comprehension. Braga: IEEE. 2010. p. 196-205. DOI 10.1109/ICPC.2010.41.

ŠMITE, D. et al. An Empirically Based Terminology and Taxonomy for Global Software Engineering. **Empirical Software Engineering**, New York, 19, n. 1, 2014. 105-153. DOI 10.1007/s10664-012-9217-9.

SOFTEX. **Melhoria de Processo do Software Brasileiro – Guia Geral**. Associação para Promoção da Excelência do Software Brasileiro. [S.l.], p. 58. 2012. (978-85-99334-48-5).

STAA, A. V. **Programação Modular:** Desenvolvendo Programas Complexos de Forma Organizada e Segura. Rio de Janeiro: Editora Campus, 2000. 767 p. ISBN 85-352-0608-6.

STEIDL, D.; HUMMEL, B.; JUERGENS, E. **Quality Analysis of Source Code Comments**. Proceedings of the IEEE 21st International Conference on Program Comprehension. San Francisco: IEEE. 2013. p. 83-92. DOI 10.1109/ICPC.2013.6613836.

SUDOL, L. A.; JASPAN, C. **Analyzing the Strength of Undergraduate Misconceptions about Software Engineering**. Proceedings of the Sixth International Workshop on Computing Education Research. New York: ACM. 2010. p. 31-40. DOI 10.1145/1839594.1839601.

SUSMAN, G. L.; EVERED, R. D. An Assessment of the Scientific Merits of Action Research. **Administrative Sciences Quartely**, 23, 1978. 582-603. DOI 10.2307/2392581.

TEASLEY, B. The Effects of Naming Style and Expertise on Program Comprehension. **International Journal of Human - Computer Studies**, Duluth, 40, n. 5, 1994. 757-770. DOI 10.1006/ijhc.1994.1036.

TENNY, T. Program Readability: Procedures versus Comments. **IEEE Transactions on Software Engineering**, Piscataway, 14, n. 9, 1988. 1271-1279. DOI 10.1109/32.6171.

THIOLLENT, M. **Metodologia de Pesquisa-Ação**. 18ª. ed. São Paulo: Cortez Editora, 2011.

TRAVASSOS, G. H. et al. **An Environment to Support Large Scale Experimentation in Software Engineering**. Proceeedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems. Belfast: IEEE. 2008. p. 193-202. DOI 10.1109/ICECCS.2008.30.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. D. **Introdução à Engenharia de Software Experimental**. Universidade Federal do Rio de Janeiro. Rio de Janeiro, p. 52. 2002. RT-ES-590/02.

TRAVASSOS, G. H.; KALINOWSKI, M. **iMPS 2013: Evidências sobre o Desempenho das Empresas**. SOFTEX. Campinas, p. 102. 2014. (978-85-99334-75-1).

UMPHRESS, D. A. et al. Software Visualizations for Improving and Measuring the Comprehensibility of Source Code. **Science of Computer Programming**, 60, n. 2, 2006. 121–133. DOI 10.1016/j.scico.2005.10.001.

WOODFIELD, S.; DUNSMORE, H.; SHEN, V. **Effect of Modularization and Comments on Program Comprehension**. Proceedings of the 5th International Conference on Software Engineering. Piscataway: IEEE Press. 1981. p. 215-223.

APÊNDICE A – Instrumentos do Estudo de Observação sobre Refatoração versus Reconstrução

Este apêndice apresenta os instrumentos utilizados para capturar a percepção de qualidade em código fonte e de refatoração dos desenvolvedores da empresa Alfa.

A. 1. Questionário 1 – Formulário de Caracterização

Nome (opcional): _____

1 Há quantos anos você tem trabalhado na indústria de software/firmware? _____

2 Como você define "refatoração" de código?

_____
_____...

3 Setor da empresa onde trabalha: _____

4 Há quanto tempo você trabalha na empresa? _____

5 Quais os papéis que você desempenhou na empresa (documentador, testador, desenvolvedor, coordenador de produto, gerente, outro), em ordem de ocorrência:

_____
_____...

6 Há quanto tempo você trabalha como desenvolvedor de software na empresa: _____

7 Em quantos projetos você participa atualmente? _____

8 Qual o seu papel em cada projeto que participa?

_____
_____...

9 Com que frequência você percebe ocorrer a substituição de membros nas equipes de desenvolvimento dos projetos que participa?

( ) Sempre ( ) Eventualmente ( ) Raramente ( ) Não ocorre

10 Quais das características a seguir descrevem os projetos que você participa (use um mesmo identificador para cada projeto, por exemplo, P1; P2; P3,... para as marcações):

PARADIGMA	Projetos
Funcional	
Orientado a Eventos	
Orientado a Objetos	
Procedural	

LINGUAGEM	Projetos
Assembly	
C	
C++	
C#	
JAVA	
Lua	
Object C	
Pascal (Delphi)	
PHP	
Scheme	

TIPO	Projetos
Equipe de Desenvolvimento Localizada na Empresa (Interior ou Capital)	
Equipe de Desenvolvimento Distribuída no País (Interior – Capital)	
Equipe de Desenvolvimento Distribuída (Brasil – Exterior)	

Sua Experiência em Desenvolvimento de Software

Por favor, indique o seu grau de experiência considerando a escala abaixo:
1 → nenhum
2 → estudei em aula, em livro ou pratiquei em sala de aula
3 → usei em 1 projeto na indústria
4 → usei em alguns projetos na indústria (2 a 4 projetos)
5 → usei em vários projetos na indústria (mais do que 4 projetos)

Experiência em Arquitetura e Codificação de Software	Nível de Experiência
Experiência utilizando requisitos para construir os modelos de projeto/arquitetura do software.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência utilizando requisitos para construir o código.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em modelagem de software.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em modelos de estrutura (Procedural).	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em diagramas de classe (Orientada a Objetos).	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência alterando modelos (arquitetura) do software para manutenção.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência alterando modelos (arquitetura) do software para reutilização.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )

Experiência em Refatoração (mudança ou melhoria do código)	Nível de Experiência
Experiência em refatorar seu próprio código.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em refatorar código de terceiros.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em identificar código que precisa ser refatorado.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em planejar a refatoração.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )
Experiência em analisar o impacto da refatoração.	1( ) 2 ( ) 3 ( ) 4 ( ) 5 ( )

A. 2. Questionário 2 – Percepção da Qualidade do Código Fonte

1 Indique (de acordo com a escala abaixo) para cada uma das situações a seguir sua percepção de qualidade em relação a um código fonte.

IM – Imprescindível **MI** – Muito Importante **DE** – Desejável **NA** – Não se Aplica

Um código fonte possui qualidade quando...	IM	MI	DE	NA
É possível ser testado.				
É fácil reaproveitá-lo.				
Está de acordo com padrões previamente estabelecidos pela equipe de desenvolvimento.				
Não possui trechos redundantes.				
Não é muito extenso.				
Apresenta bom tempo de processamento.				
Provê as funcionalidades previstas.				
Se mantém previsível em relação a seu funcionamento e desempenho nas condições estabelecidas para uso.				
É rápido de ser compilado.				
É de fácil compreensão.				
É fácil de incluir novas funcionalidades ou comportamentos.				
É fácil de manter.				
É fácil de ser adaptado para ambientes diferentes daquele para o qual tenha sido construído.				
É fácil de usar e/ou fazer referência.				
Apresenta baixa complexidade.				
Utiliza de forma apropriada os recursos de memória e energia disponíveis.				
Outros:				

2 Indique a frequência de ocorrência (de acordo com a escala abaixo) de cada uma das situações a seguir nos códigos fonte dos projetos que você participa.

SE – Sempre **EV** – Eventualmente **RA** – Raramente **NO** - Não Ocorre

	SE	EV	RA	NO
Baixo desempenho da aplicação.				
Código de difícil extensão de funcionalidades.				
Código de difícil manutenção.				
Código de difícil reutilização.				
Código muito fragmentado.				
Dificuldade de testar o código sem antes aplicar refatoração.				
Existência de código legado com o qual é necessário trabalhar.				
Forte dependência com código de outros times de desenvolvimento.				
Legibilidade deficiente, ou seja, variáveis, constantes, e/ou métodos/funções/procedimentos com nomes não intuitivos ou localizados em lugares inapropriados.				
Métodos/funções/procedimentos com muitas responsabilidades.				
Métodos/funções/procedimentos com muitos parâmetros desnecessários.				

Redundância de código.				
Variáveis, constantes, métodos/funções e/ou procedimentos desnecessários.				
Outros:				

3 Considerando suas indicações na questão anterior (2), indique, em sua opinião, os principais motivos para a existência das situações marcadas como SE (sempre) ou EV (eventualmente) no código fonte. (Enumere em ordem de relevância: do mais relevante (1) para o menos relevante. Só enumere os motivos que você imagina serem relevantes para explicar as situações da questão anterior.)

( ) Inexistência de padrões previamente estabelecidos.
( ) Pouca experiência dos responsáveis pelo código.
( ) Inexistência de requisitos explícitos.
( ) Pressão para entrega do produto.
( ) Inexistência de modelos de projeto (arquitetura de software).
( ) Outros: _____

A. 3. Questionário 3 – Percepção Individual sobre Refatoração

1 Com que frequência você refatora código?

( ) Diariamente ( ) Semanalmente ( ) Mensalmente ( ) Anualmente
( ) Raramente ( ) Nunca

2 Como você costuma realizar a tarefa de refatoração?

( ) Em grupo ( ) Em par ( ) Sozinho

3 A refatoração costuma ser planejada, tendo seus riscos e impactos analisados?

( ) Sim, sempre. ( ) Sim, algumas vezes. ( ) Sim, porém raramente. ( ) Não.

4 Indique (de acordo com a escala abaixo) com que frequência cada uma das situações a seguir influencia sua decisão sobre a refatoração (modificação ou melhoraria) do código fonte.

SE – Sempre **EV** – Eventualmente **RA** – Raramente **NI** - Não Influencia

	SE	EV	RA	NI
Baixo desempenho da aplicação.				
Código de difícil extensão de funcionalidades.				
Código de difícil manutenção.				
Código de difícil reutilização.				
Código muito fragmentado.				
Dificuldade de testar o código sem antes aplicar refatoração.				
Existência de código legado com o qual é necessário trabalhar.				
Alto tempo de compilação do código.				
Forte dependência com código de outros times de desenvolvimento.				
Falta de modelos de projeto/arquitetura que permitam entender o software.				
Código com alta complexidade.				
Legibilidade deficiente, ou seja, variáveis, constantes, e/ou métodos/funções/procedimentos com nomes não intuitivos ou localizados em lugares inapropriados.				
Métodos/funções/procedimentos com muitas responsabilidades.				
Métodos/funções/procedimentos com muitos parâmetros desnecessários.				
Código muito extenso.				
Redundância de código.				
Variáveis, constantes, métodos/funções e/ou procedimentos desnecessários.				
Outros:				

5 Quais das opções a seguir melhor definem sua motivação para refatorar um código? (Enumere em ordem de frequência: mais frequente (1) para menos frequente. Deixe em branco os itens que não ocorrem.)

( ) Quando o gerente aloca horas para refatoração.
( ) Quando identifico uma não conformidade no código.
( ) Quando não entendo o que o código faz.
( ) Quando tenho tempo disponível.
( ) Quando não encontro modelos de projeto e/ou arquitetura para ajudar no entendimento.
( ) Quando estou corrigindo defeitos que se relacionam com um código não conforme.
( ) Quando preciso adicionar novas funcionalidades ao código (evolução).

( ) Quando preciso reutilizar o código.
( ) Outros:_____

6 Você utiliza alguma ferramenta de apoio a refatoração de código?

( ) Sim, sempre. ( ) Sim, algumas vezes. ( ) Sim, porém raramente. ( ) Não.

Em caso afirmativo, qual(is) ferramenta(s) é(são) utilizada(s)?

_____
_____...

7 Alguma vez você teve que cancelar a refatoração de um código? Se sim, qual era o objetivo da mudança e por que teve que cancelar a refatoração?

_____
_____...

8 Com base em sua experiência em refatoração, indique o quanto concorda com as afirmativas a seguir:

A refatoração de código...	Concordo Fortemente	Concordo	Discordo	Discordo Fortemente	Ainda não percebi
Introduz mais defeitos no código.					
Melhora a compreensão do código.					
Melhora a legibilidade do código.					
Melhora a modularidade do código.					
Melhora o desempenho do código.					
Pode sempre ser evitada.					
Ajuda a quebrar (deixar de funcionar) código de terceiros.					
Pode ser reduzida se os requisitos estiverem explícitos.					
É influenciada pela Linguagem de Programação utilizada no projeto.					
Reduz a dependência com código de outros grupos de desenvolvimento.					
Reduz a redundância de código.					
Pode ser reduzida se existirem modelos de projeto/arquitetura.					
Torna a junção (merge) com outras mudanças de código mais complicada.					
Torna mais fácil a adição de novas funcionalidades.					
Torna mais fácil a reutilização de código.					
Torna o código mais fácil de corrigir possíveis defeitos.					
Traz alto custo ao desenvolvimento.					

APÊNDICE B – Informações Extraídas dos Artigos Incluídos na Busca Estruturada

Este apêndice apresenta as informações extraídas dos artigos selecionados pela busca estruturada.

Campos Extraídos Diretamente	
Título	To Camelcase or Under_score
Autores	Binkley, D.; Davis, M.; Lawrie, D. & Morrell, C.
Ano de Publicação	2009
Fonte de Publicação	IEEE International Conference on Program Comprehension
Link para Referência	(BINKLEY, DAVIS, *et al.*, 2009)
Abstract	
Naming conventions are generally adopted in an effort to improve program comprehension. Two of the most popular conventions are alternatives for composing multi-word identifiers: the use of underscores and the use of camel casing. While most programmers have a personal opinion as to which style is better, empirical study forms a more appropriate basis for choosing between them. The central hypothesis considered herein is that identifier style affects the speed and accuracy of manipulating programs. An empirical study of 135 programmers and non-programmers was conducted to better understand the impact of identifier style on code readability. The experiment builds on past work of others who study how readers of natural language perform such tasks. Results indicate that camel casing leads to higher accuracy among all subjects regardless of training, and those trained in camel casing are able to recognize identifiers in the camel case style faster than identifiers in the underscore style.	
Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Sem definição explícita. A compreensibilidade é vista em relação a rapidez e acurácia na assimilação de um termo.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Identificador com camelCase	-	-	-
Identificador com under_score	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	O experimento foi realizado com programadores e não programadores da área de computação e tinha como objetivo investigar o impacto do estilo de descrição de identificadores na compreensibilidade do identificador tanto por pessoas treinadas como pessoas não treinadas nos estilos. Os participantes deviam ler uma dada frase e memorizá-la para posteriormente identificar o identificador correspondente à frase em questão em um conjunto de identificadores. O que é analisado no estudo controlado é a acurácia e rapidez da resposta quando o participante desconhece um determinado estilo e quando ele recebe treinamento sobre o estilo.
Características do Código Utilizado na Avaliação	Não é utilizado código fonte, mas sim frases de identificadores de aplicações *open source*.
Características dos Participantes da Avaliação	Participaram 135 pessoas, das quais 35% estavam em algum curso da área de ciência da computação. 16% dos participantes tinham entre alguns meses e um ano de estudo na área de computação; 7% entre 1 e 2 anos de estudo; e 8% mais de 2 anos de estudo na área de computação. Do total 54% eram do sexo masculino.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Identificador com camelCase	No estudo em questão, identificadores com camelCase apresentaram maior probabilidade de indicação correta por participantes do que os identificadores com under_score, embora os participantes tenham levado mais tempo para identificar as frases corretamente. Foi também indicado que quanto mais treinados, mais rápidos os participantes conseguem corretude na identificação dos identificadores com camelCase.
Identificador com under_score	Apesar de perder para o camelCase na corretude, os identificadores com under_score foram identificados com mais agilidade por participantes sem treinamento do que por participantes com treinamento.

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	5	**E**	1	**F**	0	**Total**	9

Campos Extraídos Diretamente	
Título	The Impact of Identifier Style on Effort and Comprehension
Autores	Binkley, D.; Davis, M.; Lawrie, D.; Maletic, J.; Morrell, C. & Sharif, B.
Ano de Publicação	2013
Fonte de Publicação	Empirical Software Engineering
Link para Referência	(BINKLEY, DAVIS, *et al.*, 2013)

Abstract

A family of studies investigating the impact of program identifier style on human comprehension is presented. Two popular identifier styles are examined, namely camel case and underscore. The underlying hypothesis is that identifier style affects the speed and accuracy of comprehending source code. To investigate this hypothesis, five studies were designed and conducted. The first study, which investigates how well humans read identifiers in the two different styles, focuses on low-level readability issues. The remaining four studies build on the first to focus on the semantic implications of identifier style. The studies involve 150 participants with varied demographics from two different universities. A range of experimental methods is used in the studies including timed testing, read aloud, and eye tracking. These methods produce a broad set of measurements and appropriate statistical methods, such as regression models and Generalized Linear Mixed Models (GLMMs), are applied to analyze the results. While unexpected, the results demonstrate that the tasks of reading and comprehending source code is fundamentally different from those of reading and comprehending natural language. Furthermore, as the task becomes similar to reading prose, the results become similar to work on reading natural language text. For more "source focused" tasks, experienced software developers appear to be less affected by identifier style; however, beginners benefit from the use of camel casing with respect to accuracy and effort.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Sem definição explícita. A compreensibilidade é vista em relação a rapidez, acurácia e esforço visual na assimilação de um termo.

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Identificador com camelCase	-	-	-
Identificador com under_score	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	O projeto do primeiro experimento segue praticamente o mesmo estilo do trabalho anterior realizado pelos autores. Os participantes deviam ler uma dada frase e memorizá-la para posteriormente identificar o identificador correspondente à frase em questão em um conjunto de identificadores. O que é analisado no estudo controlado é a acurácia e rapidez da resposta quando o participante desconhece um determinado estilo e quando ele recebe treinamento sobre o estilo. O experimento foi conduzindo utilizando um *Applet* para análise de frases e escolha de identificadores. Nesse estudo foi utilizado ainda um *eye-tracker* para identificar quais dos estilos trouxe mais facilidade de trabalho para os participantes. Além do tempo e da acurácia na assimilação de termos por participantes com e sem treinamento nos estilos, esse estudo ainda analisou o esforço visual que os estilos provocam nos participantes. O segundo experimento consistiu na análise de um trecho de código fonte e na solicitação da identificação de todas as ocorrências de um dado identificador no referido trecho pelos participantes do estudo. O terceiro experimento analisa a compreensibilidade a partir da análise das respostas a duas questões de compreensão sobre sentenças fornecidas aos participantes para leitura. As sentenças tinham algumas palavras modificadas para seguirem um dos dois estilos. O quarto estudo também avaliou a compreensibilidade em relação às respostas dadas a perguntas de compreensão. Contudo, neste caso, ao invés de textos, foram utilizados códigos fonte. O quinto estudo solicitou que participantes lessem um método e resumissem o seu funcionamento. Os métodos foram escritos utilizando um dos dois estilos de identificadores. Os resultados não foram conclusivos.
Características do Código Utilizado na Avaliação	No primeiro estudo foram utilizados conjunto de frases de identificadores de aplicações de projetos *open source*. No segundo estudo foram utilizados quatro trechos de código: dois de aplicações *open source*, sendo um em C e outro em Java; e os outros dois de livros de computação, sendo um em C e outro em Java. No terceiro estudo foram utilizadas textos de tópicos diversos. No quarto estudo foram utilizados códigos escritos em C++. No quinto estudo foram utilizados códigos escritos em Java.
Características dos Participantes da Avaliação	Conjunto de 5 estudos envolvendo 3 grupos de participantes: i) 135 estudantes (programadores e não programadores) da Universidade de Loyola. Maioria dos programadores treinados no estilo camelCase. Idade variando de 17 a 22 anos. 54% do total de participantes homens, enquanto que 67% dos programadores eram homens. ii) 19 estudantes. Subconjunto de programadores do primeiro grupo que estavam pelo menos há 2 anos na universidade. 79% homens. iii) 15 programadores da Universidade de Kent State com experiência nos dois estilos de identificação.
Resultados da Avaliação	

Atributos	Relação de Impacto na Característica de Qualidade
Identificador com camelCase	Primeiro estudo: Os resultados obtidos identificaram que camelCase aumenta a corretude na assimilação dos identificadores. Contudo, o estilo camelCase leva a um tempo maior na identificação dos termos. O estudo mostrou ainda que participantes com treinamento em camelCase eram mais rápidos na identificação de identificadores com esse estilo. Em relação ao esforço visual, o estilo camelCase leva a um maior esforço e maior erro por não programadores. Segundo estudo: Identificadores com camelCase levaram mais uma vez vantagem na corretude e desvantagem no tempo. Terceiro estudo: O estilo camelCase reduziu o desempenho dos participantes em todos os aspectos. Quarto estudo: iniciantes com baixa experiência se obtiveram melhores resultados com camelCase
Identificador com under_score	Primeiro estudo: O resultado do estudo mostrou ainda que o treinamento em um estilo impacta negativamente na performance de identificação de frases no outro estilo, porque participantes com mais treinamento em camelCasel tiveram desempenho pior na identificação de frases no estilo under_score do que participantes sem treinamento em nenhum dos estilos. Quarto estudo: Experientes tiveram melhores resultados com under_score do que iniciantes.

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	5	**E**	1	**F**	0	**Total**	9

Campos Extraídos Diretamente	
Título	A Metric for Software Readability (artigo da conferência) / Learning a Metric for Code Readability (artigo estendido para o periódico)
Autores	Buse, R. & Weimer, W.
Ano de Publicação	2008 (artigo da conferência) 2013 (artigo estendido para o periódico)
Fonte de Publicação	International Symposium on Software Testing and Analysis / IEEE Transactions on Software Engineering
Link para Referência	(BUSE e WEIMER, 2008) e (BUSE e WEIMER, 2010)

Abstract

In this paper, we explore the concept of code readability and investigate its relation to software quality. With data collected from 120 human annotators, we derive associations between a simple set of local code features and human notions of readability. Using those features, we construct an automated readability measure and show that it can be 80 percent effective and better than a human, on average, at predicting readability judgments. Furthermore, we show that this metric correlates strongly with three measures of software quality: code changes, automated defect reports, and defect log messages. We measure these correlations on over 2.2 million lines of code, as well as longitudinally, over many releases of selected projects. Finally, we discuss the implications of this study on programming language design and engineering practice. For example, our data suggest that comments, in and of themselves, are less important than simple blank lines to local judgments of readability.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	É definida como sendo um julgamento humano a respeito da facilidade de compreensão de um texto.

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Tamanho médio de linha	-	O tamanho da linha é medido a partir da quantidade de caracteres que a linha apresenta. O tamanho médio de linha é medido a partir da média dos tamanhos das linhas do bloco avaliado.	Racional
Tamanho máximo de linha	-	O tamanho máximo de linha é caracterizado pelo maior tamanho de linha existente no bloco avaliado.	Racional
Quantidade média de identificadores	-	A quantidade de identificadores é medida a partir do número de identificadores existentes em uma linha. A quantidade média de identificadores é medida a partir da média das quantidades de identificadores em cada linha do bloco avaliado.	Racional
Quantidade máxima de identificadores	-	A quantidade máxima de identificadores é caracterizada pela maior quantidade de identificadores existente em uma linha no bloco avaliado.	Racional
Tamanho médio de identificador	-	O tamanho do identificador é medido a partir da quantidade de caracteres que o identificador apresenta. O tamanho médio de identificador é medido a partir da média dos tamanhos dos identificadores do bloco avaliado.	Racional
Tamanho máximo de identificador	-	O tamanho máximo de identificador é caracterizado pelo maior tamanho de identificador existente no bloco avaliado.	Racional
Indentação (espaço em branco precedente) média	-	A indentação é medida a partir da quantidade de espaços em branco que precedem o início de uma linha. A indentação média é medida a partir da média das indentações das linhas do bloco avaliado.	Racional
Indentação (espaço em branco precedente) máxima	-	A indentação máxima é caracterizada pela maior indentação existente em uma linha no bloco avaliado.	Racional
Quantidade média de palavras reservadas	-	A quantidade de palavras reservadas é medida a partir do número de palavras reservadas existentes em uma linha. A quantidade média de palavras reservadas é medida a partir da média das quantidades de palavras reservadas das linhas do bloco avaliado.	Racional
Quantidade máxima de palavras	-	A quantidade máxima de palavras reservadas é caracterizada pela maior quantidade de palavras	Racional

reservadas		reservadas existente em uma linha no bloco avaliado.	
Quantidade média de números	-	A quantidade de números é medida a partir do número de números existentes em uma linha. A quantidade média de números é medida a partir da média das quantidades de números das linhas do bloco avaliado.	Racional
Quantidade máxima de números	-	A quantidade máxima de números é caracterizada pela maior quantidade de números existente em uma linha no bloco avaliado.	Racional
Quantidade média de comentários	-	A quantidade de comentários é medida a partir do número de comentários existentes em uma linha. A quantidade média de comentários é medida a partir da média das quantidades de comentários das linhas do bloco avaliado.	Racional
Quantidade média de períodos	-	A quantidade de períodos é medida a partir do número de períodos existentes em uma linha. A quantidade média de períodos é medida a partir da média das quantidades de períodos das linhas do bloco avaliado.	Racional
Quantidade média de vírgulas	-	A quantidade de vírgulas é medida a partir do número de vírgulas existentes em uma linha. A quantidade média de vírgulas é medida a partir da média das quantidades de vírgulas das linhas do bloco avaliado.	Racional
Quantidade média de espaços	-	A quantidade de espaços é medida a partir do número de espaços existentes em uma linha. A quantidade média de espaços é medida a partir da média das quantidades de espaços das linhas do bloco avaliado.	Racional
Quantidade média de parênteses	-	A quantidade de parênteses é medida a partir do número de parênteses existentes em uma linha. A quantidade média de parênteses é medida a partir da média das quantidades de parênteses das linhas do bloco avaliado.	Racional
Quantidade média de operadores aritméticos	-	A quantidade de operadores aritméticos é medida a partir do número de operadores aritméticos existentes em uma linha. A quantidade média de operadores aritméticos é medida a partir da média das quantidades de operadores aritméticos das linhas do bloco avaliado.	Racional
Quantidade média de operadores de comparação	-	A quantidade de operadores de comparação é medida a partir do número de operadores de comparação existentes em uma linha. A quantidade média de operadores de comparação é medida a partir da média das quantidades de operadores de comparação das linhas do bloco avaliado.	Racional
Quantidade média de atribuições	-	A quantidade de atribuições é medida a partir do número de atribuições existentes em uma linha. A quantidade média de atribuições é medida a partir da média das quantidades de atribuições das linhas do bloco avaliado.	Racional
Quantidade média de blocos de decisão	-	A quantidade de blocos de decisão é medida a partir do número de blocos de decisão existentes em uma linha. A quantidade média de blocos de decisão é medida a partir da média das quantidades de blocos de decisões das linhas do bloco avaliado.	Racional
Quantidade média de loops	-	A quantidade de loops é medida a partir do número de loops existentes em uma linha. A quantidade média de loops é medida a partir da média das quantidades de loops das linhas do bloco avaliado.	Racional
Quantidade média de linhas em branco	-	A quantidade de linhas em branco é medida a partir do número de linhas em branco existentes no bloco analisado. A quantidade média de linhas em branco é medida a partir da média das quantidades de linhas em branco dentre o total de linhas do bloco avaliado.	Racional
Quantidade de ocorrências do caractere que mais apareceu no bloco	-	Quantidade de ocorrência do caractere que mais apareceu no bloco.	Racional
Quantidade de ocorrências do identificador que mais apareceu no bloco	-	Quantidade de ocorrência do identificador que mais apareceu no bloco.	Racional
Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos			
Tipo de Avaliação	Estudo experimental (*survey* + mineração de dados).		
Descrição Geral da Avaliação	Conduzido de 23 de outubro a 2 de novembro de 2008, com bonificação de 5 dólares aos participantes, sendo que os 50 primeiros que começassem ganhariam 10 dólares. Para saber se trechos de códigos eram compreensíveis, os autores primeiramente passaram uma tarefa de avaliação de trechos de código para um grupo de participantes avaliarem sua legibilidade/ compreensibilidade. Os participantes foram aconselhados a avaliarem os trechos de código sob a perspectiva particular de compreensibilidade (quão fácil um código é de ser entendido), indicando os trechos mais compreensíveis (valor atribuído perto de 5), os neutros (3) e os menos compreensíveis (valor atribuído perto de 1). Os códigos então tinham seus atributos mensurados e posteriormente minerados para identificação dos atributos que mais se relacionavam com códigos mais legíveis/compreensíveis. Assim, os autores conseguiram chegar em um conjunto de atributos que mais influenciam a legibilidade/compreensibilidade de código fonte.		
Características do Código Utilizado na Avaliação	Os trechos de código analisados foram retirados de 5 projetos de código aberto (*open source*) escritos em linguagem Java para diferentes propósitos, são eles: Azureus: Vuze 4.0.0.4 (*internet file sharing*); JasperReports 2.04 (*dynamic content*); Hibernate 2.1.8 (*database*); jFreeChart 1.0.9 (data representation); FreeCol		

	0.7.3 (game); TV Browser 2.7.2 (*tv guide*); jEdit 4.2 (*text editor*); Gantt Project 3.0 (*scheduling*); SoapUI 2.0.1 (*web services*); Data Crow 3.4.5 (data management); Xholon 0.7 (*simulation*); Risk 1.0.9.2 (*game*); JSh 0.1.37 (*security*); jUnit 4.4 (*software development*); jMencode 0.64 (*video encoding*).
Características dos Participantes da Avaliação	Participaram 110 estudantes de graduação (17 do primeiro ano; 63 do segundo ano; 30 no terceiro ou quarto ano) e 10 graduados em cursos relacionados com ciência da computação na Universidade de Virginia.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Tamanho médio de linha	**Alta relação** (> 0.9). Preditor negativo de legibilidade/compreensibilidade.
Tamanho máximo de linha	**Alta relação** (> 0.7). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de identificadores	**Alta relação** (> 0.9). Preditor negativo de legibilidade/compreensibilidade.
Quantidade máxima de identificadores	Moderada relação (> 0.6). Preditor negativo de legibilidade/compreensibilidade.
Tamanho médio de identificador	Sem relação (prox 0).
Tamanho máximo de identificador	Moderada relação (< 0,5). Preditor negativo de legibilidade/compreensibilidade.
Indentação (espaço em branco precedente) média	Moderada relação (> 0.5). Preditor negativo de legibilidade/compreensibilidade.
Indentação (espaço em branco precedente) máxima	Moderada relação (prox. de 0.5). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de palavras reservadas	Moderada relação (> 0.5). Preditor negativo de legibilidade/compreensibilidade.
Quantidade máxima de palavras reservadas	Baixa relação (< 0.2). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de números	Baixa relação (< 0.3). Preditor negativo de legibilidade/compreensibilidade.
Quantidade máxima de números	Baixa relação (< 0.2). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de comentários	Moderada relação (< 0.4). Preditor positivo de legibilidade/compreensibilidade.
Quantidade média de períodos	**Alta relação** (> 0.7). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de vírgulas	Moderada relação (< 0.5). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de espaços	Baixa relação (< 0.3). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de parênteses	**Alta relação** (> 0.9). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de operadores aritméticos	Baixa relação (< 0.1). Preditor positivo de legibilidade/compreensibilidade.
Quantidade média de operadores de comparação	Baixa relação (< 0.3). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de atribuições	Baixa relação (< 0.3). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de blocos de decisão	Baixa relação (prox. 0.2). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de loops	Baixa relação (< 0.2). Preditor negativo de legibilidade/compreensibilidade.
Quantidade média de linhas em branco	Moderada relação (> 0.5). Preditor positivo de legibilidade/compreensibilidade.
Quantidade de ocorrências do caractere que mais apareceu no bloco	Moderada relação (< 0.4). Preditor negativo de legibilidade/compreensibilidade.
Quantidade de ocorrências do identificador que mais apareceu no bloco	Moderada relação (< 0.5). Preditor negativo de legibilidade/compreensibilidade.

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	4	**E**	1	**F**	0	**Total**	8

Campos Extraídos Diretamente	
Título	Program Factors as Predictors of Program Readability
Autores	DeYoung, G. E. & Kampen, G. R.
Ano de Publicação	1979
Fonte de Publicação	IEEE Computer Society's International Computer Software and Applications Conference
Link para Referência	(DEYOUNG e KAMPEN, 1979)

Abstract

In an attempt to find a measure of program readability, a set of 30 Algol 68 programs written by students in an upper-level course was graded by hand for readability and the results compared with a set of program parameters extracted by a SNOBOL program. Correlation and multiple regression techniques were used to test hypotheses suggested in the literature on program quality, and to find a combination of parameters with maximum predictive value. A set of three variables was found to be effective in predicting readability.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	-

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**

Número de linhas *statements*	-	-	Racional
Número de operandos	-	-	Racional
Volume dos comentários (em caracteres)	-	-	Racional
A média normalizada do tamanho das variáveis	-	-	Racional
Nível de implementação estimado	-	2* (NU2/NU1) * N2, sendo N2 o número de operandos, NU1 o número operadores únicos, e NU2 o número de operandos únicos.	Racional
O nível da linguagem utilizada	-	L^2 * V, sendo L o nível de implementação estimado, e V o volume do programa dado por N*log_2 (NU1+NU2), com N sendo o número total de operadores e operandos.	Racional
O esforço mental requerido para entender o programa	-	V/L	Racional
Total de operandos e operadores	-	-	Racional
Número ciclomático	-	Soma dos *branches* do programa acrescido de 1.	Racional
Erro na predição do tamanho do programa	-	|(N-NP)/NP|, sendo NP dado por NU1* log_2 (NU1+Nu2)* log_2 (NU2).	Racional

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Estudo experimental (*survey* + regressão múltipla).
Descrição Geral da Avaliação	Participantes foram convocados para construir códigos e posteriormente avaliar a compreensibilidade dos códigos dos demais participantes, pontuando de 1 a 5 os códigos quanto a sua compreensibilidade. Posteriormente os códigos foram avaliados em relação a diferentes atributos que se tinha a suposição que pudessem estar relacionados com compreensibilidade. A partir de um conjunto de combinações de atributos e análises de regressões múltiplas, os autores então chegaram em um conjunto de atributos que melhor poderiam prever a compreensibilidade do código fonte.
Características do Código Utilizado na Avaliação	Estudantes foram convocados para construir um programa em Algol 68 que variaram de 120 a 424 linhas de código.
Características dos Participantes da Avaliação	Participaram 30 graduados e estudantes de graduação avançados da área de ciência da computação. A maioria dos estudantes já tinha tido aulas de programação básica.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Número de linhas contendo *statements*	**Alta relação** (-.50). Preditor negativo de legibilidade/compreensibilidade.
Número de operandos	**Alta relação** (-.44). Preditor negativo de legibilidade/compreensibilidade.
Volume dos comentários (em caracteres)	Moderada relação (+.35). Preditor positivo de legibilidade/compreensibilidade.
A média normalizada do tamanho das variáveis	Moderada relação (+.34). Preditor positivo de legibilidade/compreensibilidade.
Nível de implementação estimado	Moderada relação (+.34). Preditor positivo de legibilidade/compreensibilidade.
O nível da linguagem utilizada	Baixa relação (+.28). Preditor positivo de legibilidade/compreensibilidade.
O esforço mental requerido para entender o programa	Baixa relação (-.28). Preditor negativo de legibilidade/compreensibilidade.
Total de operandos e operadores	Baixa relação (-.24). Preditor negativo de legibilidade/compreensibilidade.
Número ciclomático	Baixa relação (-.23). Preditor negativo de legibilidade/compreensibilidade.
Erro na predição do tamanho do programa	Baixa relação (-.19). Preditor negativo de legibilidade/compreensibilidade.

Critérios de Avaliação													
A	0,4	**B**	0,8	**C**	0,75	**D**	4	**E**	0	**F**	0	**Total**	5,95

Campos Extraídos Diretamente	
Título	Exploring Software Measures to Assess Program Comprehension
Autores	Feigenspan, J.; Apel, S.; Liebig, J. & Kastner, C.
Ano de Publicação	2011
Fonte de Publicação	International Symposium on Empirical Software Engineering and Measurement
Link para Referência	(FEIGENSPAN, APEL, *et al.*, 2011)

Abstract

Software measures are often used to assess program comprehension, although their applicability is discussed controversially. Often, their application is based on plausibility arguments, which, however, is not sufficient to decide whether software measures are good predictors for program comprehension. Our goal is to evaluate whether and how software measures and program comprehension correlate. To this end, we carefully designed an experiment. We used four different measures that are often used to judge the quality of source code: complexity, lines of code, concern attributes, and concern operations. We measured how subjects understood two comparable software systems that differ in their implementation, such that one implementation promised considerable benefits in terms of better software measures. We did not observe a difference in program comprehension of our subjects as the software measures suggested it. To explore how software measures and program comprehension could correlate, we used several variants of computing the software measures. This brought them closer to

our observed result, however, not as close as to confirm a relationship between software measures and program comprehension. Having failed to establish a relationship, we present our findings as an open issue to the community and initiate a discussion on the role of software measures as comprehensibility predictors.

Campos Extraídos a partir do Entendimento da Pesquisadora

Descrição de Legibilidade e/ou de Compreensibilidade	A compreensibilidade de código é um processo cognitivo que não é possível de ser observado diretamente. A compreensibilidade é vista, então, em relação a corretude e o tempo de execução de tarefas de manutenção de software.

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Número ciclomático	-	Soma dos *branches* do programa acrescido de 1.	Racional
Quantidade de linhas de código	-	-	Racional
Número de atributos de uma unidade conceitual	-	-	Racional
Número de operações de uma unidade conceitual	-	-	Racional

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos

Tipo de Avaliação	Experimento
Descrição Geral da Avaliação	Os participantes tinham que realizar tarefas de manutenção de software em duas versões de código distintas de um mesmo software, uma escrita em AspectJ e outra escrita em Java. A corretude e o tempo para a realização das atividades eram mensurados. As versões tinham diferenças grandes em relação aos atributos de código fonte avaliados, assim os autores acreditavam que caso houvesse relação entre as métricas e a compreensão de código fonte, isso iria ser refletido no resultado do estudo de alguma forma, fato que não ocorreu. Apesar das diferenças em relação às métricas, nenhuma diferença significativa em relação à compreensibilidade foi observada.
Características do Código Utilizado na Avaliação	Dois códigos diferentes, um em Java e outro em AspectJ do mesmo produto. O produto consistia em um software de médio porte para manipulação de dados multimídia em dispositivos móveis.
Características dos Participantes da Avaliação	Alunos de graduação da Universidade de Passau matriculados em uma disciplina de Paradigmas Contemporâneos de Programação, na qual eram trabalhados assuntos avançados da computação como AspectJ e implementação de linhas de produto de software.

Resultados da Avaliação

Atributos	Relação de Impacto na Característica de Qualidade
Número ciclomático	Sem relação.
Quantidade de linhas de código	Sem relação.
Número de atributos de uma unidade conceitual	Sem relação.
Número de operações de uma unidade conceitual	Sem relação.

Critérios de Avaliação

A	1	**B**	0,25	**C**	0,25	**D**	5	**E**	1	**F**	0	**Total**	7,5

Campos Extraídos Diretamente

Título	A Methodology for Measuring the Readability and Modifiability of Computer Programs
Autores	Jørgensen, A.
Ano de Publicação	1980
Fonte de Publicação	BIT Numerical Mathematics
Link para Referência	(JøRGENSEN, 1980)

Abstract

This paper describes an experimentally based methodology for investigating how the readability and modifiability of computer programs can be expressed in terms of programming style. As one part of the methodology three different measures of the readability and modifiability of a program are established. The measures are obtained by human gradings or scores. As another part these measures are related to computer-evaluable programming style characteristics. The result is a formula yielding the readability and modifiability as a function of measurable program characteristics. In testing the methodology one experiment provided a formula of readability containing six computer-evaluable programming style characteristics. A further experiment indicated that the general validity of this formula of program readability needs further investigation.

Campos Extraídos a partir do Entendimento da Pesquisadora

Descrição de Legibilidade e/ou de Compreensibilidade	A compreensibilidade de um texto é a soma dos fatores estilísticos que fazem com que o texto seja mais ou menos compreensível para o leitor.

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Extensão dos comentários	-	O número total de caracteres em comentários, dividido pelo número total de caracteres (brancos e não brancos) em um programa, multiplicado por 100.	Racional

Extensão dos espaços em branco na margem esquerda	-	O número total de caracteres em branco na margem esquerda dividida pelo total de número de caracteres (brancos e não brancos) em um programa, multiplicado por 100.	Racional
Extensão das linhas em branco	-	O número total de linhas em branco dividido pelo total de número de linhas (em branco e não em branco) em um programa, multiplicado por 100.	Racional
Média de tamanho dos nomes das variáveis	-	O número total de caracteres em todas as ocorrências de nomes de variáveis dividido pelo número de ocorrência dos nomes de variáveis.	Racional
Média de número de operadores aritméticos por 100 linhas	-	O número total de operadores aritméticos (+, -, *, /, and //) dividido pelo número total de linhas em um programa, multiplicado por 100.	Racional
Média de número de comandos goto por *label*	-	O número total de comandos goto dividido pelo número total de *labels* em um programa.	Racional

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Estudo experimental (survey + regressão linear).
Descrição Geral da Avaliação	Cada expert deveria avaliar os códigos conforme a sua percepção de compreensibilidade, julgando com 1 como excelente; 4 como justo; e 7 como muito pobre. O estudo ocorreu em 1972-74. Segundo estudo: Cada avaliador, avaliou 1 programa dos 10 disponíveis, sendo cada programa avaliado por 10 estudantes diferentes. Instruções e questionários foram também entregues aos participantes. A mesma escala de avaliação da compreensibilidade foi utilizada pelos estudantes (1 muito alta compreensibilidade e 7 muito baixa). Os estudantes tinham ainda que avaliar um conjunto de características dos códigos utilizando a escala de 1-7.
Características do Código Utilizado na Avaliação	48 conjunto de códigos escritos em Algol 60 com diferentes modificações, como falta de indentação e comentários. O tamanho do programa variou de 2 (mínimo) 6 (média) a 14 (máximo) páginas. Segundo estudo: 10 programas escritos em Pascal representando algoritmos similares em relação a tarefas e algoritmos de fato e escritos por estudantes da Universidade de Arhus. O tamanho dos programas, variaram de 2, 3 e 7 páginas.
Características dos Participantes da Avaliação	Participaram 10 cientistas da computação experientes.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Extensão dos comentários	Preditor positivo de legibilidade/compreensibilidade.
Extensão dos espaços em branco na margem esquerda	Preditor positivo de legibilidade/compreensibilidade.
Extensão das linhas em branco	Preditor positivo de legibilidade/compreensibilidade.
Média de tamanho dos nomes das variáveis	Preditor positivo de legibilidade/compreensibilidade.
Média de número de operadores aritméticos por 100 linhas	Preditor negativo de legibilidade/compreensibilidade.
Média de número de comandos goto por *label*	Preditor negativo de legibilidade/compreensibilidade.

Critérios de Avaliação													
A	1	**B**	1	**C**	1	**D**	4	**E**	1	**F**	1	**Total**	9

Campos Extraídos Diretamente	
Título	What's in a Name? A Study of Identifiers / Effective Identifier Names for Comprehension and Memory
Autores	Lawrie, D.; Morrell, C.; Feild, H. & Binkley, D.
Ano de Publicação	2006 / 2007
Fonte de Publicação	IEEE International Conference on Program Comprehension / Innovations in Systems and Software Engineering
Link para Referência	(LAWRIE, MORRELL, *et al.*, 2006) / (LAWRIE, MORRELL, *et al.*, 2007)

Abstract

Readers of programs have two main sources of domain information: identifier names and comments. When functions are uncommented, as many are, comprehension is almost exclusively dependent on the identifier names. Assuming that writers of programs want to create quality identifiers (e.g., include relevant domain knowledge) how should they go about it? For example, do the initials of a concept name provide enough information to represent the concept? If not, and a longer identifier is needed, is an abbreviation satisfactory or does the concept need to be captured in an identifier that includes full words? Results from a study designed to investigate these questions are reported. The study involved over 100 programmers who were asked to describe twelve different functions. The functions used three different "levels" of identifiers: single letters, abbreviations, and full words. Responses allow the level of comprehension associated with the different levels to be studied. The functions include standard algorithms studied in computer science courses as well as functions extracted from production code. The results show that full word identifiers lead to the best comprehension; however, in many cases, there is no statistical difference between full words and abbreviations.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Sem definição explícita. A compreensibilidade é vista em relação à escolha de identificadores mnemônicos.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Identificador com palavras completas	-	-	-

Identificador com abreviações	-	-	-
Identificador com apenas 1 caractere	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	Os autores solicitam uma análise externa via survey de 3 formas de construção de identificadores: i) palavras completas; abreviações; caracteres isolados. Participantes foram requisitados a analisarem e posteriormente descreverem uma de três variantes de 12 funções e *snippets* (um por vez), sendo que as variantes só se diferenciavam na construção de seus identificadores. Os participantes eram solicitados a descrever livremente as funções apresentadas, além de prover uma medida de conforto em relação aà linguagem de programação do código (de 1 - baixo a 5 - alto).
Características do Código Utilizado na Avaliação	Foram utilizados algoritmos em C, C++ e Java "conhecidos" como de busca binária e *quick sort*, além de trechos de código variados encontrados em livros, notas de aulas e na internet. As funções variaram de 8 a 36 linhas de código. Comentários foram removidos como forma de chamar a atenção para os identificadores. As 3 variantes do código foram então criadas, focando apenas na modificação dos identificadores. No caso das modificações do código para abreviar os termos que não tinham abreviações conhecidas, um programador externo ao trabalho foi convocado para abreviar os identificadores dos códigos selecionados.
Características dos Participantes da Avaliação	Participaram 128 programadores, sendo 96 profissionais da prática e 32 estudantes. A média de idade dos participantes era de 30 anos. A média de anos de trabalho na área era de 7,5. 10% dos participantes eram mulheres.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Identificador com palavras completas	Acarreta na melhor compreensão do código fonte quando comparado com abreviação e caractere simples.
Identificador com abreviações	Compreensão maior que a de caractere simples. Em relação a identificadores com palavras completas não foi constatada grande diferença.
Identificador com apenas 1 caractere	Baixa compreensão quando comparada com palavras completas e abreviações.

Critérios de Avaliação													
A	1	**B**	1	**C**	1	**D**	5	**E**	0	**F**	1	**Total**	9

Campos Extraídos Diretamente	
Título	Program Indentation and Comprehensibility
Autores	Miara, R. J.; Musselman, J. A.; Navarro, J. A. & Shneiderman, B.
Ano de Publicação	1983
Fonte de Publicação	Communications of the ACM
Link para Referência	(MIARA, MUSSELMAN, *et al.*, 1983)

Abstract

The consensus in the programming community is that indentation aids program comprehension, although many studies do not back this up. The authors tested program comprehension on a Pascal program. Two styles of indentation were used - blocked and nonblocked - in addition to four possible levels of indentation (0, 2, 4, 6 spaces). Both experienced and novice subjects were used. Although the blocking style made no difference, the level of indentation had a significant effect on program comprehension. (2 - 4 spaces had the highest mean score for program comprehension). It is recommended that a moderate level of indentation be used to increase program comprehension and user satisfaction.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	A legibilidade é vista como sendo uma característica do programa de ser fácil de ler e entender.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Inexistência de indentação	-	-	-
Indentação com 2 espaços	-	-	-
Indentação com 4 espaços	-	-	-
Indentação com 6 espaços	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	Foi pedido para que programadores avaliassem trechos de código em diferentes formas de indentação. A compreensibilidade do código fonte foi mensurada com base na análise da corretude dos participantes ao responderem a um *quiz* contendo 13 questões sobre o código avaliado. O código foi ainda avaliado subjetivamente pelos participantes em relação a sua dificuldade.
Características do Código Utilizado na Avaliação	Um programa escrito em Pascal retirado de um livro acadêmico foi alterado para satisfazer as 4 diferentes configurações de indentação avaliadas.
Características dos Participantes da Avaliação	86 participantes inexperientes e experientes em programação. Inexperientes: Menos de três anos de experiência com programação na universidade ou menos de dois anos de experiência com programação na indústria. Experientes: Três ou mais anos de experiência com programação na universidade ou

	dois anos ou mais de experiência com programação na indústria.
Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Inexistência de indentação	-
Indentação com 2 espaços	Os melhores resultados de compreensibilidade obtidos foram com indentação de 2 e 4 espaços.
Indentação com 4 espaços	Os melhores resultados de compreensibilidade obtidos foram com indentação de 2 e 4 espaços.
Indentação com 6 espaços	Indentação com 6 espaços começam a complicar conforme a profundidade (aninhamento) do programa vai crescendo, já que a linha do programa fica cada vez maior e mais difícil de ler na tela.

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	5	**E**	0	**F**	1	**Total**	9

Campos Extraídos Diretamente	
Título	A Simpler Model of Software Readability
Autores	Posnett, D.; Hindle, A. & Devanbu, P.
Ano de Publicação	2011
Fonte de Publicação	Working Conference on Mining Software Repositories
Link para Referência	(POSNETT, HINDLE e DEVANBU, 2011)

Abstract

Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into development tool-chains, and thus continually inform developers about the readability level of the code. Unfortunately, readability is a subjective code property, and not amenable to direct automated measurement. In a recently published study, Buse et al. asked 100 participants to rate code snippets by readability, yielding arguably reliable mean readability scores of each snippet; they then built a fairly complex predictive model for these mean scores using a large, diverse set of directly measurable source code properties. We build on this work: we present a simple, intuitive theory of readability, based on size and code entropy, and show how this theory leads to a much sparser, yet statistically significant, model of the mean readability scores produced in Buse's studies. Our model uses well-known size metrics and Halstead metrics, which are easily extracted using a variety of tools. We argue that this approach provides a more theoretically well-founded, practically usable, approach to readability measurement.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	É uma propriedade que influencia o quão fácil é um trecho de código de ser lido e compreendido. É uma impressão subjetiva que programadores têm de dificuldade de um determinado código enquanto tentam entendê-lo.

Atributos de Qualidade de Código Fonte

Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Tamanho (número de linhas)	-	Medido a partir da contagem do número de linhas existentes no bloco. (não especifica se contam linhas em branco)	Racional
Tamanho (número de palavras)	-	Medido a partir da contagem do número de palavras existentes no bloco. (não especifica se conta palavras reservadas)	Racional
Tamanho (número de caracteres)	-	Medido a partir da contagem do número de caracteres existentes no bloco. (não especifica se conta caracteres de palavras reservadas)	Racional
Tamanho do Programa (N') - Halstead Metrics	-	N' = N1 + N2 (É a soma do total de número de operadores (N1) e operandos (N2).)	Racional
Vocabulário do Programa (n) - Halstead Metrics	-	n = n1 + n2 (É a soma do número de operadores únicos (n1) e operandos únicos (n2).)	Racional
Volume (V) - Halstead Metrics	Essa métrica é similar à entropia, mas representa o menor número de bits necessários para representar o programa.	V = N*log_2n (É o tamanho do programa (N) vezes o log_2 do vocabulário do programa (n).)	Racional
Dificuldade (D) - Halstead Metrics	-	D = (n1/2)*(N2/n2) (É metade do número de operadores únicos multiplicado pelo total de número de operandos, dividido pelo número de operadores distintos.)	Racional
Esforço (E) - Halstead Metrics	É uma sugestão de quanto tempo uma revisão de código pode levar.	E = D*V (É a dificuldade multiplicada pelo volume.)	Racional
Entropia	É vista como complexidade, o grau de desordem ou a quantidade de informação em um conjunto de dados.	É calculada a partir do número de termos (tokens e bytes) bem como o número de termos únicos e bytes. Ver seção 2.4.	Racional

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Estudo experimental (survey + mineração de dados).

Descrição Geral da Avaliação	Mesmos dados do estudo de (BUSE e WEIMER, 2008)/ (BUSE e WEIMER, 2010). A diferença principal do estudo é o conjunto de métricas utilizadas na mineração dos dados.
Características do Código Utilizado na Avaliação	Mesmos dados do estudo de (BUSE e WEIMER, 2008)/ (BUSE e WEIMER, 2010).
Características dos Participantes da Avaliação	Mesmos dados do estudo de (BUSE e WEIMER, 2008)/ (BUSE e WEIMER, 2010).
Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Tamanho (número de linhas)	Preditor positivo de legibilidade/compreensibilidade. Foi notado que sozinho o tamanho não tem influência grande com a compreensibilidade, mas também foi notado que existe uma relação entre o tamanho e demais características de código analisadas por (BUSE e WEIMER, 2008)/ (BUSE e WEIMER, 2010), assim sendo, ele não poderia ficar fora do modelo de predição.
Tamanho (número de palavras)	Preditor negativo de legibilidade/compreensibilidade.
Tamanho (número de caracteres)	Preditor negativo de legibilidade/compreensibilidade.
Tamanho do Programa (N') - Halstead Metrics	Preditor negativo de legibilidade/compreensibilidade.
Vocabulário do Programa (n) - Halstead Metrics	Preditor negativo de legibilidade/compreensibilidade.
Volume (V) - Halstead Metrics	Preditor negativo de legibilidade/compreensibilidade.
Dificuldade (D) - Halstead Metrics	Sem relação.
Esforço (E) - Halstead Metrics	Sem relação.
Entropia	Baixa relação. Preditor negativo de legibilidade/compreensibilidade.

Critérios de Avaliação												
A	0,55	**B**	1	**C**	0,75	**D**	4	**E**	1	**F**	0	**Total**	7,3

Campos Extraídos Diretamente	
Título	Reordering Program Statements for Improving Readability
Autores	Sasaki, Y.; Higo, Y. & Kusumoto, S.
Ano de Publicação	2013
Fonte de Publicação	European Conference on Software Maintenance and Reengineering
Link para Referência	(SASAKI, HIGO e KUSUMOTO, 2013)

Abstract

In order to understand source code, humans sometimes execute the program in their mind. When they illustrate the program execution in their mind, it is necessary to memorize what values all the variables are along with the execution. If there are many variables in the program, it is hard to their memorization. However, it is possible to ease to memorize them by shortening the distance between the definition of a variable and its reference if they are separated in the source code. This paper proposes a technique reordering statements in a module by considering how far the definition of a variable is from its references. We applied the proposed technique to a Java OSS and collected human evaluations for the reordered methods. As a result, we could confirm that the reordered methods had better readability than their originals. Moreover, we obtained some knowledge of human consideration about the order of statements.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Declarações espaçadas de suas referências	-	-	-
Declarações próximas de suas referências	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	A avaliação foi feita através de perguntas para os programadores participantes a respeito da facilidade de leitura de métodos quando estes tinham as declarações das variáveis afastadas das referências e quando eles tinham as declarações próximas das referências de uso. Os participantes tinham que avaliar qual dos métodos fornecido era mais legível ou se não existia diferença.
Características do Código Utilizado na Avaliação	20 métodos extraídos de um código escrito em Java referente a uma aplicação de TV Browser foram avaliados por 44 participantes em relação a sua facilidade de leitura. Os métodos foram avaliados em sua versão original e na versão modificada, com alterações de ordenação de declarações, seguindo as regras de ordenação estabelecidas pelos autores.
Características dos Participantes da Avaliação	44 participantes convocados em uma rede social. 8 tinham experiência com menos de 1000 linhas de código em Java; 23 tinham experiência com 1000 a 10000 linhas; e 13 com mais de 10000. Pelo menos 28 dos participantes já tinham utilizado Java em projetos acadêmicos e 12 em ambientes de trabalho.
Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**

Declarações espaçadas de suas referências Declarações próximas de suas referências	16 dos 20 métodos avaliados tiveram melhor legibilidade quando declarações de variáveis estavam próximas de suas referências.

Critérios de Avaliação												
A	1	**B**	0	**C**	0	**D**	5	**E**	1	**F**	1	**Total**	8

Campos Extraídos Diretamente	
Título	Women and men - Different but Equal: On the Impact of Identifier Style on Source Code Reading
Autores	Sharafi, Z.; Soh, Z. b.; Gueheneuc, Y.-G. & Antoniol, G.
Ano de Publicação	2012
Fonte de Publicação	IEEE International Conference on Program Comprehension
Link para Referência	(SHARAFI, SOH, *et al.*, 2012)

Abstract

Program comprehension is preliminary to any program evolution task. Researchers agree that identifiers play an important role in code reading and program understanding activities. Yet, to the best of our knowledge, only one work investigated the impact of gender on the memorability of identifiers and thus, ultimately, on program comprehension. This paper reports the results of an experiment involving 15 male subjects and nine female subjects to study the impact of gender on the subjects' visual effort, required time, as well as accuracy to recall Camel Case versus Underscore identifiers in source code reading. We observe no statistically-significant difference in term of accuracy, required time, and effort. However, our data supports the conjecture that male and female subjects follow different comprehension strategies: female subjects seem to carefully weight all options and spend more time to rule out wrong answers while male subjects seem to quickly set their minds on some answers, possibly the wrong ones. Indeed, we found that the effort spent on wrong answers is significantly higher for female subjects and that there is an interaction between the effort that female subjects invested on wrong answers and their higher percentages of correct answers when compared to male subjects.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Sem definição explícita. A legibilidade e compreensibilidade é vista em relação à retenção de informação do código fonte, no caso, relacionadas com identificadores.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Identificador com camelCase	-	-	-
Identificador com under_score	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento.
Descrição Geral da Avaliação	Os autores avaliaram a memorização de identificadores e o esforço de compreensão dos participantes. O objetivo principal do estudo era avaliar o impacto do gênero, do estilo camelCase e do estilo under_score na leitura de códigos. Para isso os autores mediram a acurácia das respostas dadas pelos participantes a questões de múltipla escolha sobre três trechos de código avaliados (selecionados aleatoriamente com um estilo especifico); o tempo gasto na atividade de análise dos códigos e das perguntas; e o esforço visual gasto para responder o questionário (menor atenção e menor tempo, menor esforço).
Características do Código Utilizado na Avaliação	Duas variações (usando camelcase e under_score) de fragmentos de três programas em Java, um 2D Graphical Frame; um Tester de Base de Dados; e um calculador de números primos.
Características dos Participantes da Avaliação	Participaram 25 estudantes de graduação, metrado e doutorado do Departamento de Computação e Software Engineering da École Polytechnique de Montreál. 9 eram mulheres e 15 homens. 29% (3 mulheres e 4 homens) não tinham preferências em relação a estilos de identificadores, o restante preferiam camelCase.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Identificador com camelCase	Em relação à corretude não houve diferença entre os diferentes estilos de identificados quando os gêneros não foram avaliados independentemente. Quando o gênero foi levado em consideração na análise, o estudo mostrou que programadores homens acertam mais respostas quando avaliam código com estilo camelCase do que quando avaliam código com under_score. Apesar disso, não houve diferença na acurácia das respostas entre homens e mulheres para esse estilo. De modo geral não houve diferença significativa entre os dois estilos.
Identificador com under_score	Homens acertaram 75% das respostas para códigos que usavam estilo under_score, contudo não utilizaram muito tempo avaliando as questões. Mulheres acertaram 83% porém gastando 18% mais tempo nas respostas.

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	5	**E**	0	**F**	0	**Total**	8

Campos Extraídos Diretamente	
Título	An Eye Tracking Study on camelCase and under_score Identifier Styles
Autores	Sharif, B. & Maletic, J.

Ano de Publicação	2010
Fonte de Publicação	IEEE International Conference on Program Comprehension
Link para Referência	(SHARIF e MALETIC, 2010)

Abstract

An empirical study to determine if identifier-naming conventions (i.e., camelCase and under_score) affect code comprehension is presented. An eye tracker is used to capture quantitative data from human subjects during an experiment. The intent of this study is to replicate a previous study published at ICPC 2009 (Binkley et al.) that used a timed response test method to acquire data. The use of eye-tracking equipment gives additional insight and overcomes some limitations of traditional data gathering techniques. Similarities and differences between the two studies are discussed. One main difference is that subjects were trained mainly in the underscore style and were all programmers. While results indicate no difference in accuracy between the two styles, subjects recognize identifiers in the underscore style more quickly.

Campos Extraídos a partir do Entendimento da Pesquisadora

Descrição de Legibilidade e/ou de Compreensibilidade	-

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Identificador com camelCase	-	-	-
Identificador com under_score	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos

Tipo de Avaliação	Experimento
Descrição Geral da Avaliação	É um estudo de replicação do experimento realizado por (BINKLEY, DAVIS, *et al.*, 2009). Como ponto adicional, o estudo em questão utiliza-se de *eyetracking* como forma de auxiliar na análise dos resultados de corretude e rapidez na análise dos identificadores.
Características do Código Utilizado na Avaliação	Não é utilizado código fonte, mas sim frases de identificadores de aplicações *open source*.
Características dos Participantes da Avaliação	Os participantes do experimento, diferentemente do que ocorre no de Blinkley, são todos programadores e com experiências nos dois estilos (camelCase e under_score), sendo as preferências deles divididas quase que meio a meio. São 15 estudantes do departamento de ciência da computação da Universidade de Kent State, dos quais 7 eram alunos de graduação e 8 eram graduados.

Resultados da Avaliação

Atributos	Relação de Impacto na Característica de Qualidade
Identificador com camelCase	Não existe diferença significativa dos resultados de corretude entre identificadores com camelCase e identificadores com under_score. Embora tenha sido identificado que identificadores com camelCase podem levar a uma falha na corretude quando há erros de escrita no final da frase. CamelCase produz um menor gap de tempo entre iniciantes e experientes.
Identificador com under_score	Identificadores com under_score levam a uma maior agilidade na identificação de identificadores corretos. Iniciantes tem mais proveito de tempo que experientes quando estão frente a identificadores com under_score.

Critérios de Avaliação

A	1	B	1	C	1	D	5	E	0	F	0	Total	8

Campos Extraídos Diretamente

Título	Quality Analysis of Source Code Comments
Autores	Steidl, D.; Hummel, B. & Juergens, E.
Ano de Publicação	2013
Fonte de Publicação	IEEE International Conference on Program Comprehension
Link para Referência	(STEIDL, HUMMEL e JUERGENS, 2013)

Abstract

A significant amount of source code in software systems consists of comments, i. e., parts of the code which are ignored by the compiler. Comments in code represent a main source for system documentation and are hence key for source code understanding with respect to development and maintenance. Although many software developers consider comments to be crucial for program understanding, existing approaches for software quality analysis ignore system commenting or make only quantitative claims. Hence, current quality analyzes do not take a significant part of the software into account. In this work, we present a first detailed approach for quality analysis and assessment of code comments. The approach provides a model for comment quality which is based on different comment categories. To categorize comments, we use machine learning on Java and C/C++ programs. The model comprises different quality aspects: by providing metrics tailored to suit specific categories, we show how quality aspects of the model can be assessed. The validity of the metrics is evaluated with a survey among 16 experienced software developers, a case study demonstrates the relevance of the metrics in practice.

Campos Extraídos a partir do Entendimento da Pesquisadora

Descrição de Legibilidade e/ou de Compreensibilidade	-

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Coeficiente de correlação entre comentário e	Métrica utilizada para avaliar a coerência	Denota o número de palavras correspondentes pelo total de palavras do comentário.	Racional

código fonte	entre código e comentários do tipo *member*.		
Tamanho do comentário	Métrica para avaliar comentários do tipo *inline*.	Denota a quantidade de palavras em um comentário.	Racional

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Survey
Descrição Geral da Avaliação	Os autores propõem um modelo para avaliação de comentários, no qual os comentários são extraídos e classificados de acordo com a seguinte classificação: copyright comments; header comments; member comments; inline comments; section comments; code comments; task comments. Os autores se propõem a avaliar os comentários de acordo com relevância deles para o código. Para isso eles identificam duas métricas: c_coeff para avaliar a coerência entre o comentário e o código ao redor dele; e tamanho do comentário para avaliar o limiar de tamanho que torna o comentário relevante ou não para o código. Para avaliar o modelo eles então realizam um survey com programadores experientes que responderem em relação a relevância de diferentes comentários para os códigos em que eles estão presentes.
Características do Código Utilizado na Avaliação	Código Java de diferentes projetos *open source*, como CSLessons, EMF, Jung, ConQAT, jBoss, voTUM, mylyn, pdfsam, jMol, jEdit, Eclipse, jabref.
Características dos Participantes da Avaliação	16 desenvolvedores membros do Software and System Engineering Chair of Technische Universit at Munchen; do Software Engineering Group da University of Bremen; estudantes do TUM; empregados da empresa de consultoria em qualidade de software CQSE GmbH e alguns de seus clientes. Os participantes tinham pelo menos 5 anos, 62% tinham mais de 10 anos de experiência em programação. 94% tinham experiência de pelo menos 5 anos com programação Java.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Coeficiente de correlação entre comentário e código fonte	Quanto menor o valor, menor a correlação do código com o comentário. O estudo mostrou que quando o coeficiente chegou perto de zero, os participantes indicaram que os comentários estavam precisando de informações adicionais e quando o coeficiente passou de 0,5, os participantes indicaram que os comentários eram desnecessários.
Tamanho do comentário	Em relação ao tamanho dos comentários, os participantes indicaram comentários com menos de 2 palavras como sendo dispensáveis para o código e comentários com mais de 30 palavras como sendo importantes para o código.

Critérios de Avaliação													
A	1	**B**	1	**C**	1	**D**	3	**E**	1	**F**	1	**Total**	8

Campos Extraídos Diretamente	
Título	The effects of naming style and expertise on program comprehension
Autores	Teasley, B.
Ano de Publicação	1994
Fonte de Publicação	International Journal of Human - Computer Studies
Link para Referência	(TEASLEY, 1994)

Abstract
The question of whether the use of good naming style in programs improves program comprehension has important implications for both programming practice and theories of program comprehension. Two experiments were done based on Pennington's (Stimulus structures and mental representations in expert comprehension of computer programs, Cognitive Psychology,19, 295-341, 1987) model of programmer comprehension. According to her model, different levels of knowledge, ranging from operational to functional, are extracted during comprehension in a bottom-up fashion. It was hypothesized that poor naming style would affect comprehension of function, but would not affect the other sorts of knowledge. An expertise effect was found, as well as evidence that knowledge of program function is independent of other sorts of knowledge. However, neither novices nor experts exhibited strong evidence of bottom-up comprehension. The results are discussed in terms of emerging theories of program comprehension which include knowledge representation, comprehension strategies, and the effects of ecological factors such as task demands and the role-expressiveness of the language.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Compreensão é construída com base no reconhecimento de operações, através de fluxos de controle e entendimento do propósito local.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Identificadores mnemônicos	-	-	-
Identificadores inapropriados	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento
Descrição Geral da Avaliação	Primeiro estudo: Para a avaliação dos identificadores mnemônicos e não apropriados, os autores formularam um conjunto de 30 questões de verdadeiro e falso para avaliação da compreensão de cada um dos programas analisados. Segundo estudo: O mesmo estudo anterior foi realizado, só que agora restringindo o tempo para apenas 1 minuto de análise do código e 2 minutos para

	responder as questões de compreensão.
Características do Código Utilizado na Avaliação	Primeiro estudo e segundo estudo: Dois programas em pascal de 21 linhas de código. Duas versões dos programas foram criadas. Uma contendo identificadores mnemônicos e outra com identificadores de apenas um caractere ou ainda não relacionados com o contexto do código. Todos os códigos estavam indentados e sem comentários.
Características dos Participantes da Avaliação	Primeiro estudo: 43 estudantes iniciantes de ciência da computação inscritos em um curso de programação avançada que requer a utilização de programação em Pascal. Segundo estudo: 18 estudantes seniores de ciência da computação inscritos em um curso de banco de dados.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Identificadores mnemônicos	No caso do experimento com novatos, os identificadores mnemônicos foram significativos para a compreensão apenas de funções, indicando que identificadores problemáticos prejudicam mais esse nível de entendimento por novatos. Não houve nenhuma diferença significativa na compreensibilidade de códigos diferentes por estudantes mais avançados.
Identificadores inapropriados	

Critérios de Avaliação												
A	1	**B**	0	**C**	0	**D**	5	**E**	0	**F**	0	**Total**	6

Campos Extraídos Diretamente	
Título	Program Readability: Procedures Versus Comments
Autores	Tenny, T.
Ano de Publicação	1988
Fonte de Publicação	IEEE Transactions on Software Engineering
Link para Referência	(TENNY, 1988)

Abstract

A 3*2 factorial experiment was performed to compare the effects of procedure format (none, internal, or external) with those of comments (absent or present) on the readability of a PL/1 program. The readability of six editions of the program, each having a different combination of these factors, was inferred from the accuracy with which students could answer questions about the program after reading it. Both extremes in readability occurred in the program editions having no procedures: without comments the procedureless program was the least readable and with comments it was the most readable.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	Legibilidade é definida no contexto de manutenção. Um programa é legível se as informações necessárias para mantê-lo são fáceis de encontrar com a leitura do código. Legibilidade é expressa pela média de respostas corretas a uma série de questões sobre um programa em um espaço de tempo.

Atributos de Qualidade de Código Fonte			
Atributos	**Descrição**	**Procedimento de Mensuração**	**Escala**
Código em uma linha e sem comentários	-	-	-
Código em uma linha e com comentários	-	-	-
Código com procedimento e sem comentário	-	-	-
Código com procedimento e com comentário.	-	-	-
Código sem procedimento e sem comentário	-	-	-
Código sem procedimento e com comentário	-	-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos	
Tipo de Avaliação	Experimento
Descrição Geral da Avaliação	A avaliação sobre a compreensibilidade do código com base nos comentários foi feita com estudantes avaliando trechos de código e respondendo a questões relacionadas com os códigos avaliados.
Características do Código Utilizado na Avaliação	Programa de "*athletic record-keeping*" codificado em PL/I. Os códigos selecionados foram complexos o suficiente para que os participantes não avaliassem ele muito rápido, mas curtos o suficiente para que os respondentes pudessem realizar a tarefa de compreensão do código em menos de 1 hora.
Características dos Participantes da Avaliação	Estudantes de engenharia de software da Universidade de Oklahoma na primavera de 1985 e outono de 1985. A maioria dos estudantes eram do último período da graduação. Participaram de cursos envolvendo Pascal, Fortran, Cobol e Assembly. Alguns com experiência de programação na indústria. Maioria dos estudantes não tinham experiência com PL/I. Ao todo foram 148 estudantes que participaram do estudo e que completaram as tarefas solicitadas.

Resultados da Avaliação	
Atributos	**Relação de Impacto na Característica de Qualidade**
Código em uma linha e sem comentários	Os comentários fizeram diferença na análise de compreensibilidade quando foram inseridos no contexto de códigos sem procedimentos.
Código em uma linha e com comentários	
Código com procedimento e sem comentário	

Código com procedimento e com comentário.	
Código sem procedimento e sem comentário	
Código sem procedimento e com comentário	

Critérios de Avaliação												
A	1	**B**	0	**C**	0	**D**	5	**E**	0	**F**	0	**Total**	6

Campos Extraídos Diretamente	
Título	The Effect of Modularization and Comments on Program Comprehension
Autores	Woodfield, S.; Dunsmore, H. & Shen, V.
Ano de Publicação	1981
Fonte de Publicação	International Conference on Software Engineering
Link para Referência	(WOODFIELD, DUNSMORE e SHEN, 1981)

Abstract

An experiment was conducted to investigate how different types of modularization and comments are related to programmers' ability to understand programs. Forty-eight experienced programmers were given eight different versions of the same program and asked to answer a twenty question quiz used to measure comprehension. These eight different versions were the result of the program being constructed with four types of modularization (monolithic, functional, super, and abstract data type), each with and without comments. Subjects whose programs contained comments were able to answer more questions than those without comments. Subjects who were given abstract data type version of the program were able to do significantly better than those with any other type of modularization.

Campos Extraídos a partir do Entendimento da Pesquisadora	
Descrição de Legibilidade e/ou de Compreensibilidade	

Atributos de Qualidade de Código Fonte

Atributos	Descrição	Procedimento de Mensuração	Escala
Presença de comentários em códigos com modularidade monolítica	-	-	-
Presença de comentários em códigos com modularidade funcional	-	-	-
Presença de comentários em códigos com super modularidade		-	-
Presença de comentários em códigos com modularidade de tipo abstrato de dados		-	-

Avaliação Empírica da Característica de Qualidade de Código Fonte com Base nos Atributos

Tipo de Avaliação	Experimento
Descrição Geral da Avaliação	Os participantes, ao receber um código, deviam responder a um conjunto de perguntas relacionadas ao código e em caso de algum tipo de dificuldade eles deveriam sinalizar na resposta que ficaram com dificuldade "stuck".
Características do Código Utilizado na Avaliação	Um programa em Fortran implementado por um dos autores foi escrito de 8 forma diferentes, conforme a combinação dos 4 tipos de modularidade identificados com a ausência e a presença de comentários. Os comentários foram introduzidos como forma de evidenciar e prover uma explicação em alto nível dos módulos implementados no programa. Fatores adicionais que pudessem impactar na compreensibilidade do código foram removidos, como indentação.
Características dos Participantes da Avaliação	Participaram 48 alunos de graduação avançados e graduados com proficiência em Fortran. Os participantes foram divididos em 8 grupos aleatoriamente (combinação entre os tipos de modularidade elencados com a ausência e presença de comentários explicativos).

Resultados da Avaliação

Atributos	Relação de Impacto na Característica de Qualidade
Presença de comentários em códigos com modularidade monolítica	Os participantes que receberam códigos com comentários pontuaram mais nas perguntas sobre os códigos do que aqueles que receberam o código sem comentários. Os comentários puderam auxiliar na compreensão geral do que uma determinada função faz e na sua função para o programa como um todo. Em contrapartida, comentários não se mostraram úteis para identificar módulos ao longo do código.
Presença de comentários em códigos com modularidade funcional	
Presença de comentários em códigos com super modularidade	
Presença de comentários em códigos com modularidade de tipo abstrato de dados	

Critérios de Avaliação												
A	1	**B**	1	**C**	1	**D**	5	**E**	0	**F**	0	**Total**	8

APÊNDICE C – Instrumentos de Captura de Código Fonte na Empresa Alfa e Agregação de Informações dos Códigos

Este apêndice apresenta os instrumentos utilizados para capturar os códigos da empresa Alfa, bem como a percepção de legibilidade e compreensibilidade de código fonte para os programadores da empresa.

C. 1. Formulário – Captura de Trechos de Código para Legibilidade/Compreensibilidade em Código Fonte Tipo 1

Nome do desenvolvedor: _____.
ID do desenvolvedor (não preencher): _____.

TIPO 1:

- O trecho de código deve ser autocontido, ou seja, deve representar um algoritmo independente de outros trechos de código dentro do software. Qualquer trecho de código é válido desde que a lógica de programação dele não seja quebrada **E** não tenha sido escrito por você mesmo!
- O trecho de código deve conter no mínimo 3 comandos consecutivos (sendo pelo menos 1 composto), devendo respeitar as marcas de mínimo e máximo no espaço delimitado no *template* de captura de código.
    - Comando simples: são exemplos de comandos simples os comandos de decisão, repetição, switch, de atribuição e expressão, de tratamento de exceção dentre outros.
    - Comando composto: também é chamado de bloco e geralmente aparece como corpo de outro comando simples, como no caso do comando de decisão, sendo delimitado por marcadores de início e fim e contendo mais de um comando em seu interior.
- Comentários, espaços, tabulações e linhas em branco existentes não devem ser retirados dos trechos de código, ou seja, o código deve ser extraído exatamente como foi encontrado.

Definições literais:
LEGÍVEL:
"que se pode ler"; "escrito em caracteres nítidos".
COMPREENSÍVEL:
"que pode ser compreendido"; "inteligível".
"acessível"; "fácil".

1 LEGIBILIDADE/COMPREENSIBILIDADE (TIPO 1) **ALTA**

Projeto e/ou Produto:

Inserir o Código no Espaço a Seguir com Fonte Courier New e Tamanho 10

Mínimo

Máximo

2 LEGIBILIDADE/COMPREENSIBILIDADE (TIPO 1) **BAIXA**

Projeto e/ou Produto:

Inserir o Código no Espaço a Seguir com Fonte Courier New e Tamanho 10

Mínimo

Máximo

3 LEGIBILIDADE/COMPREENSIBILIDADE (TIPO 1) **INTERMEDIÁRIA**

Projeto e/ou Produto:

Inserir o Código no Espaço a Seguir com Fonte Courier New e Tamanho 10

Mínimo

Máximo

C. 2. Formulário – Percepção sobre Legibilidade e Compreensibilidade em Código Fonte

Nome do desenvolvedor: _____.

ID do desenvolvedor (não preencher): _____.

1 De acordo com a sua perspectiva, indique quais itens abaixo estão relacionados com o seu conceito de LEGIBILIDADE e/ou COMPREENSIBILIDADE.

Um trecho de código é legível/compreensível quando...	SIM	NÃO	INDIFERENTE
Os identificadores das variáveis, constantes, funções, procedimento e métodos são consistentes quanto à forma de escrita (mesma língua ou mesmo esquema de separação entre palavras).			
É indentado.			
Não é preciso acessar muitos trechos de código diferentes para compreender a sua real contribuição para o software como um todo.			
Não envolve a utilização de muitas comparações.			
É independente de outros trechos de código.			
Separa os comandos em linhas diferentes.			
Apresenta comentários que sumarizam o funcionamento de cada linha do código.			
Utiliza abreviações de palavras para compor identificadores. (ex.: cont)			
Não diferencia diferentes palavras existentes em um mesmo identificador. (ex.: maisoutrapalavra)			
Apresenta comentários que explicam como deve ocorrer o uso correto do trecho de código.			
É fácil de entender a sua função para o software como um todo ou para uma parte do software.			
Apresenta comentários que sumarizam o funcionamento de determinado trecho de código.			
Separa as palavras dos identificadores com underscore. (ex.: uma_palavra)			
É dependente de muitos outros trechos de código.			
Possui identificadores de funções/procedimentos/métodos que exprimem o real comportamento da respectiva função/procedimento/método.			
Utiliza frameworks e APIs.			
Apresenta documentação do trecho de código separada da implementação.			
Não envolve a utilização de muitas expressões aritméticas.			
Possui identificadores de variáveis e constantes que exprimem a real finalidade da respectiva variável/constante.			
Implementa padrões de projeto.			
Não depende de muitos trechos de código.			
Separa as palavras dos identificadores utilizando o estilo CamelCase. (ex.: outraPalavra)			
Insere linhas em branco entre diferentes comandos.			
Utiliza palavras completas para compor identificadores. (ex.: contador)			

2 Que outros fatores, na sua opinião, tornam um código:

MAIS Legível:

_____

_____...

MAIS Compreensível:

_____

_____...

MENOS Legível:

_____

_____...

MENOS Compreensível:

_____

_____...

C. 3. Agregação de Informações dos Códigos Tipo 1

Alta Legibilidade e Compreensibilidade

Layout

ID	Declaração e Utilização das Variáveis	Tamanho dos Comandos	Separação dos Comandos	Indentação	Uso de Parênteses	Outras Observações de Layout
1	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses somente quando necessário;	; pouco espaçamento; uso de 2TBS;
3	-	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal; uso de 1TBS;
4	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal;
7	; declaração de variáveis em conjunto e no início;	; longos;	; um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 2TBS;
12	-	; médios;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 2TBS;
22	-	; longos;	; um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS;
23	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS;
24	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 1TBS;
25	; declaração de variáveis em conjunto e no início;	; médios;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal; uso de 2TBS;
26	-	; muito longos;	; um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses somente quando necessário;	; pouco espaçamento; uso de 1TBS;
27	; declaração de variáveis ao longo do código;	; longos;	; um comando por linha; uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 1TBS;

Comentário

ID	Tamanho dos Comentários	Objetivo e Características dos Comentários
1	; médios;	; funcionamento de função; explicação de comando;
3	; comentário inexistente;	-

4	; médios;	; cabeçalho; separação de trechos de código com propósitos distintos;
7	; comentário inexistente;	-
12	; comentário inexistente;	-
22	; médios;	; funcionamento de função; explicação de parâmetros e retorno; explicação de comando; passo a passo do algoritmo;
23	; longos;	; funcionamento de função; explicação de parâmetros e retorno; explicação de comando; passo a passo do algoritmo;
24	; curtos;	; explicação de comando;
25	; longos;	; funcionamento de função; explicação de parâmetros e retorno; explicação de comando; passo a passo do algoritmo;
26	; longos;	; código comentado;
27	; comentário inexistente;	-

Nomenclatura

ID	Tamanho dos Identificadores	Estilo de Escrita dos Identificadores
1	; curtos;	; não mnemônicos; caractere simples; abreviações; camel case; underscore; nomenclatura não consistente;
3	; curtos;	; mnemônicos; palavras completas; camel case;
4	; curtos;	; mnemônicos; abreviações; underscore; nomenclatura não consistente;
7	; curtos;	; mnemônicos; palavras completas; underscore;
12	; médios;	; mnemônicos; abreviações; palavras completas; camel case;
22	; longos;	; mnemônicos; abreviações; palavras completas; underscore; camel case;
23	; médios;	; mnemônicos; abreviações; camel case; underscore; identificador sem divisor de palavras; nomenclatura não consistente;
24	; curtos;	; mnemônicos; palavras completas; underscore;
25	; longos;	; mnemônicos; palavras completas; underscore;
26	; médios;	; mnemônicos; abreviações; palavras completas; camel case; nomenclatura não consistente;
27	; curtos;	; mnemônicos; caractere simples; abreviações; camel case;

Programação

ID	Comandos de Decisão	Loops	Operações	Operadores e Operandos

1	-	; poucos loops;	; operações aritméticas; operações com ponteiros; deslocamento de bits;	_
3	; poucos comandos de decisão;	-	; deslocamento de bits;	_
4	-	-	-	_
7	; poucos comandos de decisão;	-	; operações com ponteiros;	; operador ternário; operador unário;
12	-	; poucos loops;	-	_
22	; alguns comandos de decisão;	-	; operações com ponteiros; deslocamento de bits;	; números como operandos; operador unário;
23	; poucos comandos de decisão;	; poucos loops;	; operações aritméticas; operações de comparação; operações lógicas; operações com ponteiros;	; operador unário;
24	; poucos comandos de decisão;	; poucos loops;	; operações aritméticas; operações de comparação; operações lógicas; operações com ponteiros;	; operador unário;
25	; muitos comandos de decisão;	-	; operações de comparação;	; chamadas de funções como operandos;
26	; poucos comandos de decisão;	-	; operações de comparação; operações lógicas;	; chamadas de funções como operandos;
27	; alguns comandos de decisão;	-	; operações de comparação;	; operador ternário;

Baixa Legibilidade e Compreensibilidade

Layout

ID	Declaração e Utilização das Variáveis	Tamanho dos Comandos	Separação dos Comandos	Indentação	Uso de Parênteses	Outras Observações de Layout
1	; declaração de variáveis em conjunto e no início;	; médios;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação não consistente;	; uso de parênteses para enfatizar operandos em expressões;	; pouco espaçamento; uso de 2TBS somente para instruções não simples;
3	-	; longos;	; mais de um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação não consistente;	-	; pouco espaçamento;
4	-	; longos;	; um comando por linha; uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal;
7	; declaração de variáveis em conjunto e no início;	; muito longos;	; um comando por linha; uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; muitos comandos aninhados;
12	; declaração de variáveis ao longo do código;	; curtos;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal; uso de 2TBS;
22	-	; muito longos;	; um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação não consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 1TBS; muitos comandos aninhados;

23	; declaração de variáveis ao longo do código;	; curtos;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS somente para instruções não simples; muitos comandos aninhados;
24	-	; curtos;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal; uso de 1TBS;
25	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal;
26	; declaração de variáveis em conjunto e no início;	; longos;	; mais de um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação não consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal;
27	-	; médios;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente; muitos comandos aninhados;	-	; pouco espaçamento; uso de 2TBS;

Comentário

ID	Tamanho dos Comentários	Objetivo e Características dos Comentários
1	; comentário inexistente;	-
3	; comentário inexistente;	-
4	; médio;	; explicação de comando; código comentado;
7	; comentário inexistente;	-
12	; comentário inexistente;	-
22	; comentário inexistente;	-
23	; comentário inexistente;	-
24	; longos;	; explicação de comando;
25	; comentário inexistente;	-
26	; comentário inexistente;	-
27	; curtos;	; demarcar fim de bloco;

Nomenclatura

ID	Tamanho dos Identificadores	Estilo de Escrita dos Identificadores
1	; curtos;	; mnemônicos; abreviações; palavras completas; underscore;

3	; médios;	; mnemônicos; abreviações; camel case; idiomas diferentes;
4	; curtos;	; não mnemônicos; abreviações; camel case; identificador sem divisor de palavras; nomenclatura não consistente;
7	; curtos;	; não mnemônicos; abreviações; identificador sem divisor de palavras;
12	; curtos;	; não mnemônicos; abreviações; palavras completas; camel case;
22	; curtos;	; não mnemônicos; abreviações; underscore; identificador sem divisor de palavras; nomenclatura não consistente;
23	; curtos;	; não mnemônicos; abreviações; identificador sem divisor de palavras;
24	; médios;	; não mnemônicos; palavras completas; camel case;
25	; curtos;	; não mnemônicos; abreviações; underscore;
26	; médios;	; não mnemônicos; abreviações; palavras completas; identificador sem divisor de palavras;
27	; médios;	; mnemônicos; abreviações; palavras completas; camel case;

Programação

ID	Comandos de Decisão	Loops	Operações	Operadores e Operandos
1	-	; muitos loops;	; operações aritméticas; operações de comparação; operações lógicas; operações com ponteiros; deslocamento de bits;	; números como operandos; operador unário;
3	; poucos comandos de decisão;	; poucos loops;	-	_
4	; poucos comandos de decisão;	; alguns loops;	; operações aritméticas; deslocamento de bits;	; números como operandos; chamadas de funções como operandos;
7	-	-	-	; números como operandos; chamadas de funções como operandos;
12	; poucos comandos de decisão;	; alguns loops;	; operações aritméticas; operações de comparação;	; números como operandos; operador unário;
22	; alguns comandos de decisão;	-	; operações aritméticas; operações de comparação;	; números como operandos; operador unário;
23	; muitos comandos de decisão;	; muitos loops;	; operações aritméticas; operações de comparação;	; números como operandos; operador unário;
24	; poucos comandos de decisão;	-	; operações lógicas;	; números como operandos;
25	-	; poucos loops;	; operações aritméticas;	; números como operandos; chamadas de funções como operandos;
26	; alguns comandos de decisão;	; poucos loops;	; operações de comparação;	; números como operandos; chamadas de funções como operandos; operador unário;
27	; poucos comandos de decisão;	; poucos loops;	; operações de comparação;	-

Intermediária Legibilidade e Compreensibilidade

Layout

ID	Declaração e Utilização das Variáveis	Tamanho dos Comandos	Separação dos Comandos	Indentação	Uso de Parênteses	Outras Observações de Layout
1	-	; médios;	; um comando por linha; uso de linha em branco para separar comandos; com quebra de linha;	; indentação não consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS;
3	-	; curtos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação não consistente;	-	; espaçamento normal; uso de 2TBS;
4	-	; médios;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	-	; pouco espaçamento;
7	; declaração de variáveis em conjunto e no início;	; longos;	; um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 2TBS somente para instruções não simples;
12	; declaração de variáveis em conjunto e no início;	; médios;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS;
22	-	; longos;	; um comando por linha; não uso de linha em branco para separar comandos; com quebra de linha;	; indentação consistente;	; uso de parênteses para enfatizar operandos em expressões;	; espaçamento normal; uso de 2TBS somente para instruções não simples;
23	; declaração de variáveis em conjunto e no início;	; longos;	; um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 1TBS;
24	; declaração de variáveis em conjunto e no início;	; curtos;	; um comando por linha; não uso de linha em branco para separar comandos;	; indentação consistente;	; uso de parênteses somente quando necessário;	; espaçamento normal; uso de 2TBS somente para instruções não simples;
25	; declaração de variáveis ao longo do código;	; longos;	; um comando por linha; uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	-	; espaçamento normal;
26	; declaração de variáveis ao longo do código;	; muito longos;	; mais de um comando por linha; não uso de linha em branco para separar comandos; sem quebra de linha;	; indentação consistente;	-	; pouco espaçamento; uso de 1TBS;
27	; declaração de variáveis em conjunto e no início;	; curtos;	; mais de um comando por linha; uso de linha em branco para separar comandos;	; indentação consistente;	-	; espaçamento normal; uso de 1TBS somente para instruções não simples;

Comentário

ID	Tamanho dos Comentários	Objetivo e Características dos Comentários
1	; médios;	; explicação de comando; explicação de constantes literais existentes; idiomas diferentes;
3	; comentário inexistente;	-

4	; comentário inexistente;	-
7	; curtos;	; explicação de comandos; código comentado;
12	; longos;	; explicação de comando; passo a passo do algoritmo; explicação de constantes literais existentes;
22	; curtos;	; explicação de comando; passo a passo do algoritmo;
23	; médios;	; explicação de comando; passo a passo do algoritmo;
24	; comentário inexistente;	-
25	; médios;	; explicação de comando; código comentado; idiomas diferentes;
26	; comentário inexistente;	-
27	; médios;	; explicação de parâmetros e retorno;

Nomenclatura

ID	Tamanho dos Identificadores	Estilo de Escrita dos Identificadores
1	; curtos;	; mnemônicos; palavras completas; underscore; idiomas diferentes;
3	; curtos;	; mnemônicos; caractere simples; abreviações; identificador sem divisor de palavras;
4	; curtos;	; mnemônicos; abreviações; camel case; identificador sem divisor de palavras;
7	; longos;	; mnemônicos; palavras completas; underscore;
12	; longos;	; mnemônicos; palavras completas; camel case;
22	; longos;	; mnemônicos; abreviações; camel case; underscore; identificador sem divisor de palavras;
23	; curtos;	; mnemônicos; camel case; underscore; nomenclatura não consistente;
24	; curtos;	; não mnemônicos; abreviações; underscore;
25	; médios;	; mnemônicos; abreviações; underscore;
26	; curtos;	; mnemônicos; abreviações;
27	; curtos;	; não mnemônicos; abreviações; palavras completas; underscore; camel case; nomenclatura não consistente;

Programação

ID	Comandos de Decisão	Loops	Operações	Operadores e Operandos

1	; poucos comandos de decisão; operador ternário;	-	; operações aritméticas; operações de comparação; operações lógicas; deslocamento de bits;	; números como operandos; operador ternário; operador unário;
3	; muitos comandos de decisão;	-	-	-
4	; poucos comandos de decisão;	-	-	; números como operandos; operador unário;
7	; muitos comandos de decisão;	; poucos loops;	; operações de comparação; operações lógicas;	; números como operandos;
12	; poucos comandos de decisão;	; muitos loops;	; operações aritméticas; operações de comparação; operações lógicas; operador ternário; deslocamento de bits;	; números como operandos; operador ternário;
22	; muitos comandos de decisão;	-	; operações lógicas; deslocamento de bits;	; números como operandos;
23	; poucos comandos de decisão;	; poucos loops;	; operações aritméticas; operações de comparação; operações lógicas; operações com ponteiros; deslocamento de bits;	; números como operandos;
24	; poucos comandos de decisão;	; poucos loops;	; operações aritméticas; operações de comparação; operações lógicas; operações com ponteiros; deslocamento de bits;	; números como operandos; operador unário;
25	-	; poucos loops;	; operações aritméticas;	-
26	; poucos comandos de decisão;	; poucos loops;	; operações aritméticas; operações de comparação;	; números como operandos;
27	; muitos comandos de decisão;	-	; operações aritméticas; operações de comparação;	; números como operandos;

APÊNDICE D – Diretrizes de Codificação para Legibilidade e Compreensibilidade de Código Fonte

Este apêndice apresenta o conjunto final das diretrizes de codificação baseadas em evidências e próprias para o contexto organizacional da empresa Alfa. As diretrizes apresentadas nesse apêndice foram configuradas para o contexto de desenvolvimento de software escrito em C/C++ - contexto mais comum da organização para o desenvolvimento de firmware e software embarcado. No entanto, as diretrizes se mostram genéricas o suficiente para serem parametrizadas para códigos escritos em outras linguagens de programação existentes na organização;

D. 1. DIRETRIZES DE ARQUIVOS E PASTAS

1. Organize as pastas e arquivos do projeto de forma coesa
A coesão refere-se à relação existente entre os arquivos e as pastas. É importante que arquivos de uma mesma pasta tenham relação entre si ou tenham um propósito em comum. O nível de hierarquia dos arquivos e pastas deve estar de acordo com o destino de uso deles. **Sempre:** Arquivos que são utilizados em diferentes projetos devem estar no mesmo nível dos projetos que o utilizam.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Legibilidade; Manutenibilidade; Reutilização.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: *Focus Group*
Justificativa: A adequada organização dos arquivos e pasta pode facilitar a busca por elementos de programa. Além de favorecer a legibilidade, pode favorecer a manutenibilidade do código, uma vez que possibilita identificar com maior agilidade elementos de programa candidatos a serem modificados; e pode favorecer a reutilização de código, uma vez que evidencia arquivos de propósitos mais gerais.
Exemplo de Utilização/Contra Exemplo/*Template*: Não Aplicável.

2. Utilize os nomes de extensão de arquivos de acordo com o conteúdo do arquivo
i. .c: para módulos de implementação escritos em C. ii. .h: para módulos de definição escritos em C e para arquivos contendo tabelas. iii. .cpp: para módulos de implementação escritos em C++. iv. .hpp: para módulos de definição escritos em C++. v. .hh: para arquivos contendo módulos de definição de componentes. Estes arquivos de definição contêm, tipicamente, as definições de APIs tornadas disponíveis. **Atenção:** Para casos de tabelas de uso para contextos específicos e tabelas com menos de 50 linhas o programador terá a opção de descrevê-las em arquivos .h ou no próprio arquivo .c de implementação.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Legibilidade; Manutenibilidade.
Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 1 (mentor)
Justificativa: Identificar corretamente os arquivos auxilia na assimilação do tipo de conteúdo que cada arquivo possui. Além de favorecer a legibilidade, pode favorecer a manutenibilidade do código, uma vez que possibilita identificar com maior agilidade elementos de programa candidatos a serem modificados.
Exemplo de Utilização/Contra Exemplo/*Template*: Não Aplicável.

3. Cada módulo deve ser composto por pelo menos um arquivo de definição e outro de implementação
Informações de definição (isso inclui diretivas de compilação) e de implementação não podem ser descritas em um mesmo arquivo, por isso cada módulo deverá conter ao menos dois arquivos.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Legibilidade; Manutenibilidade; Reutilização.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: 1 (mentor)
Justificativa: Separar os diferentes interesses auxilia na diferenciação dos códigos de definição versus de implementação. Além de favorecer a legibilidade, pode favorecer a manutenibilidade do código, uma vez que possibilita identificar com maior agilidade elementos de programa candidatos a serem modificados; e pode favorecer a reutilização de código, uma vez que evidencia arquivos de definição que podem ser utilizados em outros contextos de implementação.
Exemplo de Utilização/Contra Exemplo/*Template*: Não Aplicável.

4. Utilize a seguinte ordem de organização de código para arquivos de definição
i. Header; ii. Includes; iii. Exported constants; iv. Exported macro; v. Exported types; vi. Exported functions prototype.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Legibilidade; Manutenibilidade.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: 1 (mentor) + 4/8 (explícitos)
Justificativa: Desenvolvedores da organização veem a prática de separação de elementos de programa diferentes como algo que facilita a legibilidade do código, uma vez que possibilita a maior agilidade ao acesso dos dados para verificação ou mesmo para manutenção.
***Template*:** Verificar o arquivo “CodeTemplates.ENU”.

5. Utilize a seguinte ordem de organização de código para arquivos de implementação
i. Header; ii. Includes; iii. Private define; iv. Private typedef; v. Private macro; vi. Private variables; vii. Private functions prototypes; viii. Private functions.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Legibilidade; Manutenibilidade.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: 1 (mentor) + 4/8 (explícitos)
Justificativa: Desenvolvedores da organização veem a prática de separação de elementos de programa diferentes como algo que facilita a legibilidade do código, uma vez que possibilita a maior agilidade ao acesso dos dados para verificação ou mesmo para manutenção.
***Template*:** Verificar o arquivo “CodeTemplates.ENU”.

D. 2. DIRETRIZES DE LAYOUT

6. A indentação deve ser feita de forma consistente

Por indentação consistente entenda a constância na indentação de blocos de níveis diferentes, ou seja, blocos que estejam no mesmo nível na árvore do programa devem estar indentados igualmente; blocos em níveis distintos devem estar indentados de forma diferente, identificando o aninhamento na estrutura do código.

Instrução do *AStyle*:
3 espaços para a indentação → (--indent=spaces=3);
Indentar o case dentro do switch → (--indent-switches);
Indentar os comentários juntamente com o código → (--indent-col1-comments).

Características de Qualidade Influenciadas: Legibilidade.

Pode Contribuir para: -

Evidência na Literatura Técnica:
- ⊖ **(BUSE e WEIMER, 2008)**;
- ⊖ **(BUSE e WEIMER, 2010)**;
- ⊕ **(JøRGENSEN, 1980)**;
- ⊕ **(MIARA, MUSSELMAN, *et al.*, 1983)**.

Percepção de Importância na Equipe: 8/8 (explícitos) + 11/11 (implícitos)

Justificativa: A indentação foi observada tanto nos resultados dos estudos experimentais quanto por desenvolvedores como uma prática que auxilia a legibilidade do código, principalmente no que diz respeito à identificação de comandos aninhados. 2 artigos (sendo um extensão do outro) avaliaram a indentação como um preditor negativo de legibilidade, contudo, esses artigos tiveram menor nota de avaliação do que os artigos que avaliaram positivamente a indentação. Além disso, podemos inferir que a indentação excessiva é que pode ocasionar problemas de legibilidade, como identificado por **(MIARA, MUSSELMAN, *et al.*, 1983)**, que aconselha de 2 a 4 espaços de indentação.

Exemplo de Utilização:

```
#include <iostream>
#include <cstdlib>
#include <ctype.h>

using namespace std;

int restart ( void );

int main( void )
{
   restart();
}

int restart( void )
{
   char option;

   cout << "Do you want to restart the program? [Y/N]\n";
   cin >> option;
   option = toupper( option );

   // it evaluates option provided
   switch( option )
   {
      case 'Y':
         system ( "cls" );
         main ();
      case 'N':
         return EXIT_SUCCESS;
      default:
         cout << "\aInvalid option!\n\n";
         restart();
    }
}
```

7. Quebre e utilize indentação em expressões lógicas com 2 ou mais operadores não iguais

Em expressões lógicas muito grandes e não contendo mesmo tipo de operador (com 2 ou mais operadores não iguais), quebrá-la e indentá-la de acordo com a lógica da expressão, respeitando a

ordem de precedência da operação.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: -

Pode Contribuir para: Legibilidade; Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: *Focus Group*

Justificativa: A quebra e indentação de expressões lógicas foram observadas no *focus group* como uma prática que pode beneficiar a legibilidade de expressões lógicas muito extensas, já que auxilia na visualização das diferentes comparações e condições da expressão. Olhando para a literatura técnica podemos ainda fazer uma relação dessa diretriz com o tamanho da linha, visto como prejudicial nos estudos para a retenção de informação (legibilidade e compreensibilidade do código).

Exemplo de Utilização:

```
if(
      ( x == 0 )
      ||
      (
         ( x >= 2 )
         &&
         ( x <= 5 )
       )
   )
---------------------------------
if(
   ( x == 0 )
   ||
   (
      ( x >= 2 )
      &&
      (
         ( x <= 5 )
         ||
         ( y == 3 )
      )
   )
)
```

Contra Exemplo:

```
if( ( x == 0 ) || ( ( x >= 2 ) && ( x
<= 5 ) ) )
---------------------------------
if( ( x == 0 ) || ( ( x >= 2 ) && ( (
x <= 5 ) || ( y == 3 ) ) ) )
```

8. O uso de chaves para identificação de blocos deve ser consistente, respeitando o estilo 2TBS

Utilize o estilo 2TBS (*two true braces style* – a chave inicial e final aparecem em linhas separadas e alinhadas com o elemento que inicia o bloco).

Atenção: Isso também deve ser aplicado mesmo para instruções simples.

Instrução do *AStyle*:
Estilo 2TBS→(--style=allman);
Aplicação da diretriz para instruções simples → (--add-brackets).

Características de Qualidade Influenciadas: -

Pode Contribuir para: Legibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 1/8 (explícito) + 10/11 (implícitos)

Justificativa: Foi identificada como importante a escolha de um estilo único e consistente para o uso de chaves como forma de melhorar a visibilidade do início e do fim de um dado bloco.

Exemplo de Utilização:

```
//C hello world example
#include <stdio.h>

int main()
{
   printf( "Hello world\n" );
   return 0;
}
```

Contra Exemplo:

```
//C hello world example
#include <stdio.h>

int main(){
   printf( "Hello world\n" );
   return 0;
}
---------------------------------
//C hello world example
#include <stdio.h>

int main(){
   printf( "Hello world\n" );
   return 0;}
```

9. Atente para o espaçamento consistente (1 espaço) entre diferentes estruturas sintáticas e elementos do programa

Utilize apenas 1 espaço para separar:

i. Parênteses dos parâmetros de entrada em definição e chamada de funções e procedimentos;
ii. Parênteses da expressão constante em estruturas de controle completas;
iii. Operandos de operadores.

Atenção: Para operadores unários e para aqueles usados como referência a elementos não deve ser aplicada essa diretriz, exemplo: .; ->; ().

Instrução do *AStyle*:
Parênteses dos parâmetros e das expressões → (--pad-paren);
Espaçamento entre operandos e operadores → (--pad-oper);
Alinhamento dos operadores de ponteiro e de referência com o tipo → (--align-pointer=type).

Características de Qualidade Influenciadas: -

Pode Contribuir para: Legibilidade; Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 2/8 (explícitos)

Justificativa: O espaçamento é uma prática mencionada por desenvolvedores como importante para facilitar a visualização dos elementos do programa. É possível que um número muito grande de espaços separando estruturas sintáticas e elementos do programa dificulte a legibilidade e compreensão de toda a instrução, por isso é importante que exista o espaçamento, mas ele não deve ser superior a 1 espaço. Essa regra de 1 espaço não é diretamente mencionada em artigos, contudo pode ser inferida de alguns atributos de qualidade que se relacionam com a legibilidade do código, como é o caso do tamanho total da linha de código.

Exemplo de Utilização:

```
#include <stdio.h>

int main()
{
   int first;
   int second;
   int add;
   int subtract;

   printf( "Enter two integers\n" );
   scanf( "%d%d", &first, &second );

   add = first + second;
   subtract = first - second;

   printf( "Sum = %d\n", add );
   printf(    "Difference    =    %d\n",
subtract );

   return 0;
}
```

Contra Exemplo:

```
#include <stdio.h>

int main()
{
   int first;
   int second;
   int add;
   int subtract;

   printf("Enter two integers\n");
   scanf(  "%d%d",&first,&second);

   add=first+second;
   subtract=first-second;

   printf("Sum = %d\n",add);
   printf("Difference=%d\n"          ,
subtract);

   return 0;
}
```

10. Utilize parênteses para delimitar operandos em expressões

Os parênteses em expressões servem para indicar a ordem de precedência da operação, contudo pode ainda ser utilizado para melhorar a legibilidade de expressões com muitos operandos envolvidos. Sempre utilizar parênteses para separar os operandos em uma expressão como forma de enfatizar a precedência de operação da expressão.

Atenção: Para operadores unários não deve ser aplicada essa diretriz.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Legibilidade; Compreensibilidade.

Pode Contribuir para: Manutenibilidade.

Evidência na Literatura Técnica:
- **(BUSE e WEIMER, 2008)**;
- **(BUSE e WEIMER, 2010)**.

Percepção de Importância na Equipe: 1/8 (explícito)

Justificativa: A utilização de parênteses, mesmo em casos opcionais, foi vista por um desenvolvedor como uma boa prática para auxiliar a visualização de expressões e evitar erros. Contudo é importante destacar que artigos identificaram que grande quantidade de parênteses pode ter um efeito negativo em relação à legibilidade. Nesse caso é importante avaliar subjetivamente se a inserção de parênteses pode auxiliar na melhor visualização do escopo das expressões. Expressões muito grandes talvez obtenham

benefícios com esse recurso.

Exemplo de Utilização:

```
int checkVowel( char a )
{
   if( ( a >= 'A' ) && ( a <= 'Z' ) )
   {
      a = a + ( 'a' - 'A' );
   }

   if( ( a == 'a' ) || ( a == 'e' ) || ( a == 'i' ) || ( a == 'o' ) || ( a == 'u' ) )
   {
      return TRUE;
   }

   return FALSE;
}
```

Contra Exemplo:

```
int checkVowel( char a )
{
   if( a >= 'A' && a <= 'Z' )
   {
      a = a + 'a' - 'A';
   }

   if( a == 'a' || a == 'e' || a == 'i' || a == 'o' || a == 'u' )
   {
      return TRUE;
   }

   return FALSE;
}
```

11. Utilize linhas em branco para separar instruções extensas das demais instruções e para separar blocos de instruções de diferentes propósitos

O propósito dessa diretriz é utilizar linhas (1 linha) em branco como forma a separar instruções que exijam um maior nível de atenção do programador ou ainda que se distinguem das demais instruções no que tange a sua responsabilidade dentro do programa.

Exemplos de códigos a serem separados:

i. Includes das demais instruções;
ii. Declarações das demais instruções;
iii. Atribuições das demais instruções;
iv. Trechos de código que trabalham com variáveis diferentes;
v. Estrutura de controle completa das demais instruções.

Instrução do *AStyle*: Separa estruturas de controle completas de demais instruções → (--break-blocks)

Características de Qualidade Influenciadas: Legibilidade.

Pode Contribuir para: -

Evidência na Literatura Técnica:

- **(BUSE e WEIMER, 2008)**;
- **(BUSE e WEIMER, 2010)**;
- **(JøRGENSEN, 1980)**.

Percepção de Importância na Equipe: 4/8 (explícitos) + 7/11 (implícitos)

Justificativa: A linha em branco nesse caso serve explicitamente para melhorar a legibilidade de código com muitas linhas de instruções juntas, que muitas vezes dificulta a fácil visualização do código. Esse caso é agravado quando as instruções são muito extensas. Instruções muito extensas (uma seguida da outra) podem confundir o desenvolvedor na identificação da linha que estava sendo visualizada, podendo provocar desvios na leitura do código e, por conseguinte, problemas na legibilidade.

Exemplo de Utilização:

```
#include<stdio.h>

main()
{
   int n;

   printf( "Enter an integer\n" );
   scanf( "%d", &n );

   if( ( n%2 ) == 0 )
   {
      printf( "Even\n" );
   }
   else
   {
      printf( "Odd\n" );
   }

   return 0;
}
```

Contra Exemplo:

```
#include<stdio.h>
main()
{
   int n;
   printf( "Enter an integer\n" );
   scanf( "%d", &n );
   if( ( n%2 ) == 0 )
   {
      printf( "Even\n" );
   }
   else
   {
      printf( "Odd\n" );
   }
   return 0;
}
```

12. Faça apenas uma declaração por linha

Cada declaração deverá ser realizada independentemente de outras declarações, mesmo que elas sejam relacionadas ao mesmo tipo de dados. Em nenhuma hipótese devem-se declarar diferentes estruturas de dados em uma mesma linha ou ainda misturar declarações de constantes com variáveis e funções.

Instrução do *AStyle*: *Default*

Características de Qualidade Influenciadas: Legibilidade.

Pode Contribuir para: Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 7/8 (explícitos) + 11/11 (implícitos)

Justificativa: Apesar de não explícito na literatura técnica, a declaração em uma única linha de diferentes estruturas de dados, variáveis, constantes e/ou funções acaba interferindo no tamanho total da linha de comando, essa sim vista como prejudicial para a compreensão da ideia geral do comando e ainda para a legibilidade do programa. Para os desenvolvedores, realizar apenas uma declaração por linha foi visto como importante para garantir a legibilidade e compreensibilidade do código fonte. Além disso, essa prática pode auxiliar a realização de comentários de linha quando necessários.

Exemplo de Utilização:

```
char string[ 100 ];
char* temp;
char* pointer;
char* ch;
char* start;
```

Contra Exemplo:

```
char string[ 100 ], *temp, *pointer,
ch, *start;
```

13. Escreva apenas uma instrução simples por linha

Cada instrução deverá ser descrita em uma linha particular. Observar que blocos e estruturas de controle completas não são afetados por essa diretriz. Por outro lado, as instruções simples que os compõem devem observar esta diretriz.

Instrução do *AStyle*: *Default*

Características de Qualidade Influenciadas: -

Pode Contribuir para: Legibilidade; Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 7/8 (explícitos) + 11/11 (implícitos)

Justificativa: Não é uma recomendação explícita na literatura técnica, porém está relacionada com o tamanho da linha que foi visto como prejudicial para a legibilidade e compreensibilidade do código fonte. Para os desenvolvedores, descrever apenas um comando por linha foi visto como importante para garantir a legibilidade e compreensibilidade ao código fonte.

Exemplo de Utilização:

```
#include <stdio.h>

int main()
{
   int n;
   int reverse;

   reverse = 0;

   printf(   "Enter   a   number   to
reverse\n" );
   scanf( "%d", &n );

   while( n != 0 )
   {
      reverse = reverse * 10;
      reverse = reverse + n%10;
      n = n/10;
   }

   printf( "Reverse of entered number
is = %d\n", reverse);

   return 0;
}
```

Contra Exemplo:

```
#include <stdio.h>

int main()
{
   int n, reverse = 0;

   printf(   "Enter   a   number   to
reverse\n" ); scanf( "%d", &n );

   while( n != 0 )
   {
      reverse = reverse * 10; reverse
= reverse + n%10; n = n/10;
   }

   printf( "Reverse of entered number
is = %d\n", reverse); return 0;
}
```

14. Tamanho de linha deve ser menor que 80 caracteres

As linhas de comandos do programa devem ter no máximo 80 caracteres, independente da resolução

utilizada no computador ou da IDE de programação utilizada. O valor de 80 caracteres é sugerido por ser um valor intermediário que possibilita a visualização completa do comando descrito na linha.

Para casos em que a linha exceda o limite, a linha deve ser quebrada seguindo as seguintes premissas:

1. Quebre depois de uma vírgula;
2. Quebre antes de um operador;
3. Alinhe a nova linha com o início da expressão da linha anterior.

Instrução do *AStyle*: Define tamanho máximo de linha → (--max-code-length=80)

Características de Qualidade Influenciadas: Legibilidade; Compreensibilidade.

Pode Contribuir para: Manutenibilidade.

Evidência na Literatura Técnica:

- **(BUSE e WEIMER, 2008)**;
- **(BUSE e WEIMER, 2010)**.

Percepção de Importância na Equipe: 3/8 (explícitos) + 5/11 (implícitos)

Justificativa: Tamanho de linha muito grande foi visto como preditor negativo de legibilidade uma vez que prejudica a atenção do leitor para o comando como um todo, principalmente quando ele não é visto por completo na tela do computador. Além disso, dificulta a leitura e memorização do comando como um todo, prejudicando a sua compreensão.

Exemplo de Utilização:

```
int soma(int a, int b, int c,
         int d, int e, int f,
         int g)
-------------------------------------------------------
if( ( a == 'a' ) || ( a == 'e' )
  || ( a == 'i' ) || ( a == 'o' )
  || ( a == 'u' ) )
```

Contra Exemplo:

```
int soma(int a, int b, int c, int d,
int e, int f, int g)

-------------------------------------------------------
if( ( a == 'a' ) || ( a == 'e' ) || (
a == 'i' ) || ( a == 'o' ) || ( a ==
'u' ) )
```

15. Variáveis devem ser declaradas em conjunto e no início do seu escopo válido

Todas as variáveis devem ser declaradas e estar localizadas no início do escopo válido para elas. Escopo válido pode ser tanto no início de um arquivo; no início de uma função/procedimento; no início de uma estrutura de controle completa; ou ainda no início de um bloco. Utilize blocos aninhados para declarar as variáveis locais de modo a garantir que tenham o menor escopo possível.

Lembre: Comandos de *switch* sem delimitadores de início e fim de bloco não configuram um escopo por si só.

Atenção: Essa diretriz também é válida para a variável declarada dentro do *for*, para evitar problemas com compiladores.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Legibilidade.

Pode Contribuir para: Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica:

- **(SASAKI, HIGO e KUSUMOTO, 2013)**.

Percepção de Importância na Equipe: 4/8 (explícitos)

Justificativa: Desenvolvedores veem essa prática como positiva na obtenção de boa legibilidade, uma vez que a localização de informações das variáveis fica facilitada, além de evitar a presença de declarações entre comandos, o que pode prejudicar o entendimento de trechos de códigos. Contudo, é importante destacar que na literatura técnica foi encontrado um trabalho, indicando que quanto mais próxima a declaração da variável de sua referência, melhor a legibilidade do código.

Exemplo de Utilização:

```
#include <stdio.h>

int main()
{
   int array[ 100 ];
   int minimum;
   int size;
   int c;
   int location;

   printf( "Enter the number of
elements in array\n" );
   scanf( "%d", &size );

   printf( "Enter %d integers\n",
```

Contra Exemplo:

```
#include <stdio.h>

int main()
{

   printf( "Enter the number of
elements in array\n" );
   int size;
   scanf( "%d", &size );
   printf( "Enter %d integers\n", size
);

   int array[ 100 ];

   for( int c = 0; c < size; c++ )
```

```
size );

   for( c = 0; c < size; c++ )
   {
      scanf( "%d" , &array[ c ] );
   }

   minimum = array[ 0 ];
   location = 1;

   for( c = 1; c < size; c++ )
   {
      if ( array[c] < minimum )
      {
         minimum = array[ c ];
         location = c + 1;
      }
   }

   printf(  "Minimum  element  is
present at location %d and it's value
is %d.\n", location, minimum );
   return 0;
}
```

```
{
   scanf( "%d" , &array[ c ] );
}

int minimum;
minimum = array[ 0 ];
int location;
location = 1;

for( int c = 1; c < size; c++ )
{
   if ( array[c] < minimum )
   {
      minimum = array[ c ];
      location = c + 1;
   }
}

   printf( "Minimum element is present
at  location  %d  and  it's  value  is
%d.\n", location, minimum );
   return 0;
}
```

16. Constantes devem ser declaradas em conjunto e no início do seu escopo válido

Todas as constantes devem ser declaradas e estar localizadas no início do escopo válido para elas. Escopo válido pode ser tanto no início de um arquivo; no início de uma função/procedimento; no início de uma estrutura de controle completa; ou ainda no início de um bloco. Utilize blocos aninhados para declarar as variáveis locais de modo a garantir que tenham o menor escopo possível.

Lembre: Comandos de *switch* sem delimitadores de início e fim de bloco não configuram um escopo por si só.

Atenção: Atentar para o uso de arquivos de definição em C para constantes de propósitos gerais.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Legibilidade.

Pode Contribuir para: Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica:
(SASAKI, HIGO e KUSUMOTO, 2013).

Percepção de Importância na Equipe: 4/8 (explícitos)

Justificativa: Desenvolvedores veem essa prática como positiva na obtenção de boa legibilidade, uma vez que a localização de informações das constantes fica facilitada, além de evitar a presença de declarações entre comandos, o que pode prejudicar o entendimento de trechos de códigos. Especificamente para constantes, é possível que suas informações necessitem ser atualizadas em caso de alguma mudança nas regras que as definiam. Assim, a localização em conjunto auxilia nessa modificação. Contudo, é importante destacar que na literatura técnica foi encontrado um trabalho, indicando que quanto mais próxima a declaração da variável de sua referência, melhor a legibilidade do código.

Exemplo de Utilização:

```
#include <stdio.h>

#define MAJORITY 18

int main()
{
   const int RETIREMENT = 65;
   const int DRIVER = 16;

   int age;

   printf( "How old are you? " );
   scanf( "%d", &age );

   if( age >= MAJORITY )
   {
      printf( "You're old enough to
```

Contra Exemplo:

```
#include <stdio.h>

#define MAJORITY 18

int main()
{

   int age;

   printf( "How old are you? " );
   scanf( "%d", &age );

   const int DRIVER = 16;

   if( age >= MAJORITY )
   {
      printf( "You're old enough to
```

```
join the army and drive cars.\n" );
   }
   else if( age >= DRIVER )
   {
        {
           printf( "You're old enough
to drive cars, but not to join the
army.\n" );
        }
        else
        {
           printf( " You're too young
to join the army or drive cars.\n" );
        }
   }

   if( age >= RETIREMENT )
   {
      printf(   "You   can   already
retire!\n" );
   }
}
```

```
join the army and drive cars.\n" );
   }
   else if( age >= DRIVER )
   {
        {
           printf( "You're old enough
to drive cars, but not to join the
army.\n" );
        }
        else
        {
           printf( " You're too young
to join the army or drive cars.\n" );
        }
   }

   const int RETIREMENT = 65;

   if( age >= RETIREMENT )
   {
      printf(   "You   can   already
retire!\n" );
   }
}
```

D. 3. DIRETRIZES DE COMENTÁRIO

17. Selecione um idioma (Inglês ou Português) no qual será redigido os comentários
Comentários utilizados no código devem todos ser escritos em um mesmo idioma. A escolha do idioma deve ser feita conforme o perfil dos desenvolvedores que irão atuar no código em questão.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: -
Pode Contribuir para: Compreensibilidade; Manutenibilidade.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: 8/8 (explícitos) + 2/11 (implícitos)
Justificativa: Tanto no contexto de desenvolvimento de software colocalizado como no desenvolvimento distribuído de software é imprescindível que exista comunicação via código entre os diferentes desenvolvedores, para isso é necessário o uso de um idioma comum. O uso de um idioma comum é imprescindível para a boa compreensão do código e também para a sua manutenção.
Exemplo de Utilização/Contra Exemplo/*Template*: Não Aplicável.

18. Insira um comentário de identificação (cabeçalho) para cada arquivo
O comentário de cabeçalho deverá conter: i. Nome: é o nome do arquivo, deve refletir o propósito do arquivo; ii. Autores: é a lista dos autores que participaram da criação ou evolução do módulo para que atingisse a versão corrente; iii. Versão: última versão do arquivo; iv. Data de Criação: data da criação do arquivo; v. Data de Modificação: data da última alteração do arquivo; vi. Resumo: breve descrição sobre o arquivo e seu conteúdo; vii. Uso: em caso de API, descrever o seu modo de uso; viii. Abreviação: abreviação do arquivo como forma de utilizá-lo como prefixo de elementos do arquivo que sejam públicos. É importante que esta abreviação seja identificada e definida.
Instrução do *AStyle*: -
Características de Qualidade Influenciadas: Compreensibilidade.
Pode Contribuir para: Manutenibilidade.
Evidência na Literatura Técnica: -
Percepção de Importância na Equipe: 3/8 (explícitos)
Justificativa: A informação de cabeçalho é vista pelos desenvolvedores como uma boa prática de comunicação com os responsáveis do arquivo em caso de necessidade de sua modificação, o que indiretamente pode auxiliar na compreensibilidade do código. A literatura técnica, no entanto, não é direta em relação aos comentários de cabeçalho, embora destaque a importância em se comentar o código.
***Template*:**

Verificar arquivo CodeTemplate.ENU.

19. Insira um comentário de sumarização do propósito de cada função

O comentário de função deve conter informações do propósito da função de maneira bem direta, não se faz necessário à explicação de cada detalhe da função. Contudo, parâmetros de entrada e de retorno devem ser explicados. Variáveis externas (principalmente globais) referenciadas dentro do escopo da função também devem ser explicadas para evitar que o desenvolvedor tenha que procurar em demais trechos de código por informações adicionais para o entendimento da função.

O comentário deverá conter:

i. Resumo: breve descrição sobre o arquivo e seu conteúdo, podendo conter exemplo;
ii. Restrição: alguma restrição de uso da função;
iii. Parâmetros de Entrada: descrição dos argumentos;
iv. Retorno: descrição do retorno.

Atenção: É importante estar atento para o tipo de comentário que é descrito, comentários muito curtos ou que simplesmente repetem o que o código já diz não contribuem para melhorar a compreensão do código e prejudica a legibilidade.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Compreensibilidade.

Pode Contribuir para: -

Evidência na Literatura Técnica:
- **(BUSE e WEIMER, 2008)**;
- **(BUSE e WEIMER, 2010)**;
- **(DEYOUNG e KAMPEN, 1979)**;
- **(JøRGENSEN, 1980)**;
- **(STEIDL, HUMMEL e JUERGENS, 2013)**;
- **(TENNY, 1988)**;
- **(WOODFIELD, DUNSMORE e SHEN, 1981)**.

Percepção de Importância na Equipe: 7/8 (explícitos) + 4/11 (implícitos)

Justificativa: Essa diretriz é baseada em indicações de problemas evidenciados por desenvolvedores que encontram dificuldade para entender determinadas funções, principalmente quando estas utilizam variáveis fora do escopo da função. Além disso, os resultados de diferentes estudos experimentais indicam que a existência de comentários auxilia na compreensibilidade do código.

***Template*:**
Verificar arquivo CodeTemplate.ENU.

20. Comentários de linha devem ser utilizados somente em casos específicos

O comentário de linha só deve existir para explicar algo que não puder ser inferido da linha em questão.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Legibilidade; Compreensibilidade.

Pode Contribuir para: -

Evidência na Literatura Técnica:
- **(STEIDL, HUMMEL e JUERGENS, 2013)**.

Percepção de Importância na Equipe: 1/8 (explícito)

Justificativa: Embora artigos técnicos indiquem que quanto mais linhas de comentário mais compreensível um código tende a ser, o excesso de comentários por linha prejudica a legibilidade do código fonte. A menos que o comentário agregue informações adicionais para compreensão da linha de código, ele deverá ser eliminado ou ainda sumarizado em comentários de trechos de código. Um estudo revelou que quanto maior a correlação entre o comentário e o código, menos ele agrega de informação ao código e, por conseguinte, menos auxilia em sua compreensão.

Contra Exemplo:

```
int firstValue; // first value
int secondValue; // second value
```

D. 4. DIRETRIZES DE NOMENCLATURA

21. Selecione um idioma (Inglês ou Português) no qual será redigido o código

Identificadores utilizados no código devem todos ser escritos em um mesmo idioma. A escolha do idioma deve ser feita conforme o perfil dos desenvolvedores que irão atuar no código em questão.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: -

Pode Contribuir para: Compreensibilidade; Manutenibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 8/8 (explícitos) + 2/11 (implícitos)

Justificativa: Tanto no contexto de desenvolvimento de software colocalizado como no desenvolvimento distribuído de software é imprescindível que exista comunicação via código entre os diferentes desenvolvedores, para isso é necessário o uso de um idioma comum. O uso de um idioma comum é imprescindível para a boa compreensão do código e também para a sua manutenção.

Exemplo de Utilização/Contra Exemplo/*Template*:
Não Aplicável.

22. Os identificadores devem ser de fácil memorização

Os identificadores devem ser de fácil memorização, sendo intuitivos e representando de fato a real finalidade do elemento de programa em questão. Evitar identificadores com apenas uma letra ou abreviações não conhecidas.

Sempre:

i. Nomes de variáveis e constantes devem ser expressões substantivadas.
ii. Variáveis booleanas devem ser nomeadas para representar o significado do valor verdadeiro.
iii. Procedimentos devem ser nomeados pelo que eles fazem, não por como eles fazem.
iv. Nomes de função devem refletir o valor retornado.
v. Nomes de funções booleanas devem refletir o significado do valor verdadeiro.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Compreensibilidade

Pode Contribuir para: Manutenibilidade.

Evidência na Literatura Técnica:

- **(LAWRIE, MORRELL, *et al.*, 2006)**;
- **(LAWRIE, MORRELL, *et al.*, 2007)**;
- **(TEASLEY, 1994)**.

Percepção de Importância na Equipe: 8/8 (explícitos) + 10/11 (implícitos)

Justificativa: A ideia principal nessa diretriz é fazer com que os identificadores dos elementos de programa consigam representar o propósito daquilo que identificam, facilitando a sua assimilação pelo desenvolvedor, evitando interpretações ambíguas que possam levar ao erro.

Exemplo de Utilização:

```
void sortString(char *s)
{
   char ch;
   int i;
   int length;
   char *pointer;
   char* result;
   int aux;

   length = strlen( s );
   result = ( char* ) malloc( length +
1 );
   pointer = s;
   aux = 0;

   for ( ch = 'a'; ch <= 'z'; ch++ )
   {
      for ( i = 0; i < length; i++ )
      {
         if ( *pointer == ch )
         {
            *(   result   +   aux   )   =
*pointer;
            aux++;
         }
         pointer++;
      }
      pointer = s;
   }
   *( result + aux ) = '\0';

   strcpy( s , result );
   free( result );
}
```

Contra Exemplo:

```
void sort (char *s)
{
   char c;
   int i;
   int l;
   char *p;
   char* r;
   int a;

   l = strlen( s );
   r = ( char* ) malloc( l + 1 );
   p = s;
   a = 0;

   for ( c = 'a'; c <= 'z'; c++ )
   {
      for ( i = 0; i < l; i++ )
      {
         if ( *p == c )
         {
            *( r + a ) = *p;
            a++;
         }
         p++;
      }
      p = s;
   }
   *( r + a ) = '\0';

   strcpy( s , r );
   free( r );
}
```

23. Atentar para o uso adequado de prefixos e sufixos em elementos do programa

Utilize os seguintes **prefixos**:

i. tbl_: para arquivos .h contendo tabelas;
ii. g_: para variáveis globais;
iii. e_: para enum;
iv. E_: para constantes dentro de enum;
v. u<tamanho do union>: para union, substituindo <tamanho do union> pelo tamanho do union;
vi. <abreviação do nome do arquivo>_: para elementos públicos do arquivo, substituindo o <abreviação do nome do arquivo> pela abreviação presente no cabeçalho do arquivo do elemento público em questão.

Utilize os seguintes **sufixos**:

i. _: para parâmetros de entrada (argumentos);
ii. _r: para funções recursivas;
iii. _t: para typedef, union e structs.

Atenção: Atentar para o uso do static em variáveis globais que forem ser usadas em outros arquivos.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: -

Pode Contribuir para: Legibilidade; Integridade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 2/8 (explícitos)

Justificativa: Desenvolvedores identificaram a importância da distinção entre os diferentes elementos do programa como forma de agilizar o processo de associação do tipo e do dado manipulado. Alguns prefixos auxiliam ainda a evitar problemas de uso indevido de variáveis e funções.

Exemplo de Utilização:
Verificar o arquivo “padroes.c”.

24. Os identificadores dos elementos do programa devem ser consistentes quanto a sua forma de representação

Todos os identificadores de elementos do programa devem ser descritos utilizando um mesmo formato de escrita, permitindo a diferenciação entre os diferentes elementos do programa (variáveis, constantes, tipos de dados entre outros). É importante ainda que eles sejam descritos em sua forma mais completa (palavras completas), podendo estas ser substituídas por abreviações, desde que as abreviações sejam conhecidas e definidas.

O formato deve ser como segue:

i. Typedef, Union, Struct, Enum, Variáveis, Funções e Procedimentos: escritas em caixa baixa, utilizando o estilo *camel case* para separar as palavras/abreviações;
ii. Constantes: escritas em CAIXA ALTA, utilizando o *underscore* para separar as palavras/abreviações.

Instrução do *AStyle*: -

Características de Qualidade Influenciadas: Legibilidade; Compreensibilidade.

Pode Contribuir para:-

Evidência na Literatura Técnica:

- **(BINKLEY, DAVIS, *et al.*, 2009)**;
- **(BINKLEY, DAVIS, *et al.*, 2013)**;
- **(BUSE e WEIMER, 2008)**;
- **(BUSE e WEIMER, 2010)**;
- **(DEYOUNG e KAMPEN, 1979)**;
- **(JøRGENSEN, 1980)**;
- **(LAWRIE, MORRELL, *et al.*, 2006)**;
- **(LAWRIE, MORRELL, *et al.*, 2007)**;
- **(SHARAFI, SOH, *et al.*, 2012)**;
- **(SHARIF e MALETIC, 2010)**.

Percepção de Importância na Equipe: 8/8 (explícitos) + 9/11 (implícitos)

Justificativa: O principal benefício é possibilitar a diferenciação dos diferentes elementos de programa declarados ao longo do código. Além disso, a delimitação do uso de palavras completas ou de abreviações conhecidas auxilia no entendimento do código sem necessidade de navegação ou mesmo inserção de comentários adicionais ao código. É importante que os identificadores não sejam muito extensos, pois isso pode impactar no tamanho da linha, dificultando a assimilação e retenção da informação de um dado comando. A maioria dos resultados de estudos experimentais e desenvolvedores indica que o uso do estilo *camel case* contribui para a compreensibilidade do código quando comparado

ao uso de outro tipo de marcação, como *underscore* ou sem separação de palavras, por isso é viável a sua utilização quando possível.

Exemplo de Utilização:

```
#define BUFFER_LENGTH 2048

int firstValue;
int secondValue;
```

Contra Exemplo:

```
#define bufferLength 2048

int first_value;
int SECONDvalue;
```

25. As constantes utilizadas devem ser simbólicas

Evitar o uso de literais ao longo do código. Para isso, criar constantes simbólicas, respeitando a delimitação do escopo de cada constante. As constantes podem ser:

i. Públicas: definidas no módulo de definição servidor que será utilizado por algum módulo cliente;
ii. Encapsuladas: definidas no módulo de implementação, sendo utilizada somente no escopo desse módulo;
iii. Tabulares: definidas em arquivos de definição contendo exclusivamente constantes, podendo ser utilizadas em um ou mais módulos cliente.

Atenção: Avaliar a necessidade de criação de constantes simbólicas para os casos de constantes literais simples.

Instrução do *AStyle*: -

Características de Qualidade: -

Pode Contribuir para: Legibilidade; Compreensibilidade.

Evidência na Literatura Técnica: -

Percepção de Importância na Equipe: 2/8 (explícitos) + 9/11 (implícitos)

Justificativa: O uso de literais ao longo do código foi relatado por diferentes desenvolvedores como uma prática que dificulta o entendimento do código, principalmente quando este também não está comentado.

Exemplo de Utilização:

```
//    defined    constants:    calculate
circumference
#include <stdio.h>

#define PI 3.14159
#define NEWLINE "\n"

int main (){
   double r = 5.0; // radius
   double circle;

   circle = 2 * PI * r; // PI instead of
3.14...

   printf( "%f", circle );
   printf( "%s", NEWLINE );

       return 0;
}
```

Contra Exemplo:

```
//  defined  constants:  calculate
circumference
#include <stdio.h>

#define NEWLINE "\n"

int main (){
   double r = 5.0;
   double circle;

   circle = 2 * 3.14159 * r;

   printf( "%f", circle );
   printf( "%s", NEWLINE );

   return 0;
}
```

D. 5. ORIENTAÇÕES DE PROGRAMAÇÃO

- Evitar o uso de *goto*;
- Evitar o uso de operadores ternários (“?”) quando a expressão ultrapassar o limite de tamanho de linha;
- Evitar código comentado que não é mais utilizado;
- Atentar para o tamanho da solução (em código) dado ao problema, isso pode ser um indicativo de que ou o problema não está bem estruturado ou a complexidade estrutural da solução é alta. Em blocos, funções e arquivos.
  - (DEYOUNG e KAMPEN, 1979);
  - (FEIGENSPAN, APEL, *et al.*, 2011);
  - (POSNETT, HINDLE e DEVANBU, 2011).