Visão introdutória sobre os conceitos de código limpo

Introductory view of clean code concepts

Carlos Roberto Mayer
Universidade Estadual de Ponta Grossa – UEPG – Ponta Grossa – PR
crmayer@uol.com.br
Mse. Ademir Mazer Jr.
Universidade Tecnológica Federal do Paraná – UTFPR – Ponta Grossa – PR
ademir.mazer.jr@gmail.com

Resumo

A evolução das linguagens de programação, principalmente devido ao progresso e melhoria dos computadores, trouxe consigo a necessidade de que os códigos fossem escritos de forma com que facilitem as manutenções. Durante muitos anos e em contínuo desenvolvimento, são criados padrões que permitam a facilidade de entendimento do código, chamados de código limpo, que agregam ao software muita qualidade. O código limpo, que deixa seu programa mais claro e belo, pois programar é um “trabalho de arte”, permite aos desenvolvedores uma visão melhor do seu funcionamento, maior facilidade de encontrar as rotinas que possam estar com algum erro ou necessitem de alteração, fazendo com que o tempo da manutenção possa ser menor e de maior confiabilidade. Neste trabalho apresentam-se algumas técnicas utilizadas e como alguns desenvolvedores fazem em seus programas para apresentar um código de qualidade, com menor probabilidade de erros, falhas e confusões. É de grande importância a participação de todos os envolvidos no projeto, pois somente assim poderemos produzir um código que tenha qualidade, que necessite de muito poucas manutenções, pois normalmente são elas que geram os problemas encontrados em muitos programas. Esta visão introdutória deverá abrir caminho para outras pesquisas sobre desenvolvimento de programas utilizando as técnicas de codificação para qualidade de software, código limpo, que estão em constante evolução.

Palavras-chave: Código Limpo. Qualidade de Software. Técnicas de Desenvolvimento de Software.

Abstract

The evolution of programming languages, mainly due to the progress and improvement of computers, brought with it the need for codes that were written so it could facilitate the maintenance. For many years and in continuous development, standards are created in order to allow and facilitate the code understanding, called clean code that add great quality to the software. The clean code let the program code lighter and beautiful because to develop is a "work of art", allowing developers to have a better view of its operation, easier way to find the

routines that may be with a mistake or require some update, making the maintenance time smaller and more reliable. This paper presents some techniques used and how some developers make their programs to present a code with quality, with less failures and confusion. It is of great importance the participation of all involved in the project, to so produce a code that has quality, requiring very little maintenance because they usually are the ones who generate the problems found in many programs. This introductory view should open for further research on development using techniques for software quality, clean code, which are constantly evolving.
Keywords: Clean Code. Software Quality. Software Development techniques.

Sobre a história do desenvolvimento de software é possível relatar que antes de 1940, período anterior às linguagens de programação, os algoritmos conhecidos mais antigos reportam a Mesopotâmia e tratavam de uma série de conjuntos de dados utilizados para cálculos. A partir dos séculos XIII e XIV as constatações matemáticas começam a evoluir para funções relacionais. Babbage e Lady Lovelace criaram alguns programas, no que na década de 40 chamaríamos de código de máquina, um deles para calcular uma sequência de números racionais chamada de “Números de Bernoulli” (FONSECA FILHO, 2007, p. 111).

Leonardo Torres y Quevedo produziu, em 1914, um código na sua língua natural para o primeiro jogo computadorizado conhecido “O Xadrezista”, um autômato que jogava xadrez (DALAKOV, 2015). Em 1939, Helmut Schreyer, criou um código análogo para a primeira máquina de computação, o Z3 criado por Konrad Zuse (LEE, 2015). O pai da ciência da computação Alan Turing, desenvolveu uma linguagem ou uma série de regras simples de manipulação de símbolos para seu equipamento (BBC, 2015). Já Alonzo Church, criador do “Cálculo Lambda”, baseado no trabalho de Turing, influenciou o desenvolvimento das linguagens Lisp de John McCarthy, lendário no campo da ciência da computação e inteligência artificial (LISP..., 2015).

A partir do fim da 2ª Guerra Mundial, os computadores ainda eram limitados a usos acadêmicos e principalmente militares, os programadores inseriam de forma direta, através de cabeamento, nos computadores os códigos de máquina. Criou-se então uma forma de programação chamada “*assembling*” (montador), porém, para inserir o código nos computadores ainda

utilizava-se o código de máquina. Linguagens como Fortran e Lisp foram criadas nesse período ainda utilizadas nos computadores de 1ª geração.

A 2ª geração de computadores, que utilizavam transistores, fazendo com que reduzissem muito de tamanho e melhorando seu desempenho, fazendo-se necessários melhores sistemas operacionais e principalmente novas linguagens de programação, com isso e com a invenção dos circuitos integrados surgem os computadores de 3ª geração. Toda essa evolução tecnológica ficou conhecida como a “crise de software” que nos afeta até a atualidade, pois como as máquinas evoluíram muito tornando necessário que os programas evoluíssem muito também, trazendo complexidade e grandes problemas. Com a 4ª geração de computadores iniciada em 1971, que utilizavam os recém-criados microprocessadores, surgem na mesma época muitas linguagens de programação, mas destaca-se o Pascal pela sua simplicidade, por poder ser utilizado em máquinas com configurações diferentes e também pela facilidade didática (FONSECA FILHO, 2007, p. 112).

As linguagens de programação evoluíram desde a década de 70, principalmente linguagens estruturadas, porém não existiam ferramentas, softwares que auxiliassem aos programadores como temos atualmente. A partir dos anos 90 a programação orientada a objetos teve um crescimento maior, devido à internet estar mais presente na vida das pessoas e ao surgimento de novas arquiteturas como cliente-server fazendo com que as linguagens de programação também evoluíssem nesse sentido, linguagens como Object Pascal que utiliza a IDE Delphi, C++, Visual Basic entre outras, culminando nas linguagens atuais como Java, PHP, Python entre outras (FONSECA FILHO, 2007, p. 126). Ainda Koscianski; Soares (2007, p.22) citam que com tudo isso surgiu problemas, que nos acompanham até os dias atuais.

> • cronogramas não observados;
> • projetos com tantas dificuldades que são abandonados;
> • módulos que não operam corretamente quando combinados;
> • programas que não fazem exatamente o que era esperado;
> • programas tão difíceis de usar que são descartados;
> • programas que simplesmente param de funcionar (KOSCIANSKI; SOARES, 2007, p. 22).

A qualidade dos softwares é discutível? Com os erros que aparecem de todos os lados mesmo após muitos anos de trabalho, um grande exemplo foi o chamado “Bug do Milênio”, que poderia ter causado danos irreparáveis em

áreas vitais como: médica, aérea, militar, bancária, dentre outros. E este acontecimento não poderia ter sido previsto antes do desenvolvimento dos softwares? É difícil a resposta para essas questões, pode-se considerar que a atividade de desenvolver um software é complexa e às vezes até imprevisível. Isso acontece porque a dificuldade para se desenvolver um software começa bem antes do programador estar envolvido, inicia-se pela tarefa árdua para determinar o escopo do projeto, conseguir com que o usuário exponha toda a sua necessidade no momento da análise dos requisitos é de uma complexidade altíssima (KOSCIANSKI; SOARES, 2007, p. 22).

Além de tudo isto as mudanças de legislação e novas necessidades do usuário fazem com que os softwares sofram muitas alterações durante sua vida útil. Em todo processo, temos pessoas envolvidas, com suas limitações, emoções e sentimentos, o usuário com sua necessidade, o analista e o programador que desenvolvem seu trabalho intelectual que podemos chamar de “trabalho de arte” poderão entrar em conflitos, portanto é de extrema importância a disciplina estar aliada a todos os passos desse desenvolvimento. Para tanto, ao longo do tempo, metodologias, tecnologias e ferramentas estão em constante evolução procurando reduzir os problemas durante o desenvolvimento (KOSCIANSKI; SOARES, 2007, p. 22).

Para Machado, et al. (2006, p.82) todo desenvolvedor já deve ter se deparado com um código ruim que tenha criado, que após sofrer várias alterações no decorrer do tempo tornou-se um código caótico, ou tenha sido criado por outro programador, isso causará no mínimo uma demora muito maior para finalizar as alterações necessárias, se for possível é claro, pois cada alteração realizada pode causar problemas em outros pontos do software, com isso a produtividade irá diminuir até chegar à zero sendo necessária a reescrita total do código aumentando ainda mais o atraso na entrega.

> O planejamento e o gerenciamento da qualidade têm representado um papel cada dia mais forte no contexto do desenvolvimento de software. Desde o início de um projeto, a qualidade deve ser vista como um fator crítico para o sucesso do software e deve ser considerada no planejamento e gerenciamento do mesmo (MACHADO, et al., 2006, p.82).

Segundo Garvin (1987 apud PRESSMAN, 2011, p. 360) um software deve agregar valor para o usuário, fornecendo todas as funcionalidades observadas na análise de requisitos. Devem possuir inovações que façam com

que o usuário se encante na primeira utilização. Tem que fazer com que o usuário confie que estará sempre funcionando, disponível e livre de erros. O software deve estar em conformidade com os demais disponíveis no mercado, ter uma codificação padronizada e limpa de acordo com o que foi projetado. Deve inspirar a confiança do usuário mesmo que tenha sofrido mudanças ou correções no decorrer do tempo sem causar danos ao funcionamento e a vida útil do software. Também terá de ser de fácil interpretação e manutenção pela equipe técnica que efetuará as mudanças ou correções. Ser esteticamente bem definido, fazendo com que o usuário tenha prazer em lhe utilizar. E finalmente fazer com que o usuário perceba a qualidade do software sem que tenha necessidade de conhecimentos técnicos na área de informática.

Para McCall; Richards e Walters (1977 apud PRESSMAN, 2011, p. 361), que em muito concordam com Garvin, para eles um software tem que estar em conformidade com as definições realizadas na análise de requisitos junto ao usuário. Deve realizar o que se propõe de forma precisa fazendo com que seja confiável para o cliente. Que utilize o máximo os recursos disponíveis para funcionar com eficiência. Ser seguro e íntegro em relação a acesso aos dados e ao próprio programa. Deve ser de fácil manuseio, fazendo com que o usuário o utilize de forma responsável e conclusiva. Fazer com que a equipe técnica tenha facilidade no momento da correção ou manutenção do software. Que tenha sido devidamente testado, garantindo seus funcionamentos, antes de ser colocado em produção. Ser desenvolvido de maneira que possibilite a utilização de suas funcionalidades em outros projetos. E finalmente, e se possível, necessidade mínima de mudanças se for preciso à troca de equipamentos e até mesmo outros programas que sejam substituídos. “Uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o utilizam.” PRESSMAN, 2011, p.360.

Já nos princípios do site “Manifesto para o desenvolvimento ágil de software”, pode-se dizer que sem dúvida o usuário tem que ficar satisfeito tanto com o prazo de entrega quanto com o programa. Que alterações que sejam solicitadas pelo usuário, que tragam benefícios para o mesmo, sejam realizadas independentemente da etapa do desenvolvimento. Encantar o usuário com entregas periódicas, preferencialmente adiantadas e com

qualidade. Todo o projeto deve ser desenvolvido em conjunto por analistas e programadores, motivados e em ambiente e condições de trabalho de excelência, lembrando que o que fazemos é um “trabalho de arte”, tudo isso fará com que o trabalho realizado seja confiável e de qualidade. A equipe como um todo, deve passar informações em reuniões, pois somente assim todos os envolvidos estarão “falando a mesma língua”, isso tornará a equipe auto-organizável e que sempre buscará a excelência, refletindo de tempos em tempos como aperfeiçoar e ajustar sua conduta (BECK et al., 2015).

Para medirmos a evolução é necessário que o programa esteja prático e útil. Reuniões entre cliente, programadores e *stackholders* devem ser realizadas em uma frequência que não atrapalhe o projeto, de forma indefinida e constante buscando sempre métodos eficientes para manter a sustentabilidade do ambiente de trabalho. Nunca se esquecendo da qualidade do software, aplicando metodologias superiores, elegância no código e beleza na apresentação visual. Utilizar programação simples e reutilizar muito, maximizando a produtividade (KOSCIANSKI; SOARES, 2007, p. 27).

Beck et al. (2015) em seu manifesto nos dizem que:

> Estamos descobrindo maneiras melhores de desenvolver software fazendo-o nós mesmos e ajudando outros a fazê-lo. Através deste trabalho, passamos a valorizar:
> **Indivíduos e interação entre eles** mais que processos e ferramentas
> **Software em funcionamento** mais que documentação abrangente
> **Colaboração com o cliente** mais que negociação de contratos
> **Responder a mudanças** mais que seguir um plano
> Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

Martin (2009, p. 7) explica o que alguns programadores famosos dizem sobre qualidade do código ou código limpo, como Bjarne Stroustrup, o pai da linguagem de programação C++, diz que não se deve utilizar de rodeios, pois estes escondem problemas e dificultam alterações, os erros devem ser tratados estrategicamente, evitando que sejam criadas rotinas que prejudiquem o desempenho do programa, crê na beleza e na efetividade do código.

Concordando com ele, de sua forma Grady Booch, criador das raízes do projeto orientado a objetos, compara um código a uma prosa, pois a objetividade simples e direta criada pelo programador torna sua leitura facilmente compreensível (MARTIN, 2009, p. 8).

Já Dave Thomas, fundador da Object Technology International Inc. (OTI) onde foi responsável pelo início do desenvolvimento da IDE Eclipse, o programador além de desenvolver seu código pode sempre melhorá-lo através da utilização de testes de qualidade, deve usar nomes significativos, não permitir que uma rotina possa ser utilizada de mais de uma forma, deve demonstrar, caso existam dependências e também quando alguma informação não estiver explícita, de uma forma que as torne facilmente compreendidas (MARTIN, 2009, p. 9).

Michael Feathers, já tem uma definição mais pessoal, pois das várias particularidades que tornam um código limpo, o programador estava tão envolvido e interessado no desenvolvimento, que criou um código que não deixa nada evidente que possa melhorá-lo (MARTIN, 2009, p. 9).

Ainda Ron Jeffries, co-fundador da Metodologia de desenvolvimento de software: Extreme Programming (XP), concentra-se nas regras de Beck, um dos autores do Manifesto Ágil, onde os testes devem ser realizados na sua totalidade, reutilizar código evitando assim repetições, checar se tudo que foi projetado está contemplado no software e maximizar a abstração no código, a revisão de seu código e essencial para evitar duplicações, pois isso pode acontecer quando um conceito não estiver bem estabelecido pelo desenvolvedor (MARTIN, 2009, p. 10).

Por fim Ward Cunningham, desenvolvedor do primeiro Wiki, como Stroustrup e Booch também prima pela beleza do código, pois quando o desenvolvedor lê seu código e é exatamente como havia sido projetado faz parecer que até a linguagem foi criada para isso (MARTIN, 2009, p. 11).

Percebe-se que utilizar nomes significativos é de suma importância para tornar nosso código de fácil entendimento e belo, então é possível seguir algumas regras, Martin (2009) cita que tudo em um programa recebe um nome, portanto deve ser claro e deve porque está ali, qual sua utilidade e como será utilizado. Deve-se tomar cuidado também ao utilizar nomes muito parecidos ou que possam ser utilizados equivocadamente. Para distinguir nomes dentro do código, é interessante utilizar palavras cujo significado demonstre com clareza para o que será utilizada. Um nome deve poder ser lido na sua língua nativa, ou seja, nada de abreviaturas que criarão palavras inexistentes e sem nexo, não é necessário estabelecer abreviaturas de tipagem junto ao nome definido,

pois o código deve ser simplificado, que não permitirão que seja encontrada com facilidade dentro do código, causando problemas nos momentos da manutenção dos programas.

Para Martin (2009) deve-se buscar não utilizar repetições de código, para evitá-las crie funções, procurando não utilizar estruturas para tratar erros em funções específicas que chamem a função que tem relação com o funcionamento do software. As funções devem ter o mínimo de linhas possível, mas sem perder seu objetivo, usar o mínimo de parâmetros possível de preferência não ultrapassar três, procurar não criar aqueles enormes aninhamentos e deve ter uma endentação que o torne visualmente bonito.

> “As funções devem fazer uma coisa. Devem fazê-la bem. Devem fazer apenas ela.” (MARTIN, 2009, p. 35).

Ainda no conceito de Martin (2009) comentários são simplesmente formas de expressar nossa falta de condições de conseguir que nosso código possa passar tal informação, nem sempre por erros de programação, pois certas linguagens não nos permitem expressar exatamente o que queremos expressar no código. Outro motivo de que comentários não são úteis é que dificilmente são atualizados nos momentos que o programa sofre manutenção.

Portanto, é necessário fazer com que o código seja auto-explicável, o máximo use comentários para explicar alguma consequência que a rotina irá causar ou para descrever algum funcionamento de alguma rotina padrão. Como já visto anteriormente que endentação ou formatação do código é algo que fará com que seu programa fique mais legível, de fácil entendimento e criando uma beleza muito importante principalmente para os futuros profissionais que terão de realizar manutenções. Assim, também se faz necessário procurar usar o mesmo padrão em todo seu código, discutindo com a equipe qual a melhor formatação para o mesmo, dando preferência para pular uma linha a cada novo conceito.

Considerações finais

Sendo este trabalho uma introdução ao código limpo, ele poderá ser complementado, aprofundando-se nas técnicas atualmente utilizadas e na sempre busca de aperfeiçoamento, bem como realizar uma ampla pesquisa

junto aos desenvolvedores de software procurando descobrir onde estão ou não sendo aplicados tais conhecimentos, para entender os ganhos e perdas da utilização ou não de tais conceitos. Empiricamente é possível constatar que existe um grande campo para pesquisas na área, pois esse assunto não é muito divulgado e nem difundido, sendo possível ver que muitos acadêmicos recém-formados ou até em formação não sabem do que trata o código limpo, normalmente pensam que tão somente é se utilizar de tabulação e quebra de linhas.

Referências

BBC Homepage. History: Alan Turing (1912-1954). Disponível em: http://www.bbc.co.uk/history/people/alan_turing. Acesso em: 10/02/2015 22:45.

BECK, Kent et al. Manifesto para o desenvolvimento ágil de software Disponível em: http://www.manifestoagil.com.br/principios.html. Acesso em: 09/02/2015 20:00.

DALAKOV, Georgi. Biography of Leonardo Torres y Quevedo (1852-1936). Disponível em: http: //history-computer.com/Dreamers/Torres_chess.html, http://history-computer.com/Babbage/LeonardoTorres.html e http://history-computer.com/People/TorresBio.html. Acesso em: 10/02/2015 22:00.

FONSECA FILHO, Cléuzio. História da computação: O Caminho do Pensamento e da Tecnologia. Porto Alegre-RS: EDIPUCRS, 2007. 205 p.

KOSCIANSKI, André e SOARES, Michel dos Santos. Qualidade de Software: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. Segunda Edição. São Paulo-SP: NOVATEC, 2007. 395 p.

LEE, J. A. N. Computer Pioneers : Konrad Zuse (1910-1995). Disponível em: http://history.computer.org/pioneers/zuse.html. Acesso em: 10/02/2015 22:30.

LISP of John McCarthy. (1927-2011) Disponível em: http://history-computer.com/ModernComputer/Software/LISP.html. Acesso em: 10/02/2015 23:00.

MACHADO, Cristina Ângela Filipak et al. Introdução à Engenharia de Software e à Qualidade de Software. Lavras-MG: UFLA/FAEPE, 2006. 156 p.

MARTIN, Robert C. Código Limpo: Habilidades Práticas do Ágile Software. Rio de Janeiro-RJ: Alta Books, 2009. 440 p.

PRESSMAN, Roger S. Engenharia de software: uma abordagem profissional. Sétima Edição. Porto Alegre-RS: AMGH, 2011. 764 p.