INTERFACE TECNOLÓGICA

ARQUITETURA LIMPA: Como minimizar o custo de um software

CLEAN ARCHITETURE: How to minimize the cost of software

Daniel Poletto Donato – daniel.donato231@gmail.com
Faculdade de Tecnologia de Taquaritinga – Taquaritinga – São Paulo – Brasil

Jederson Donizete Zuchi – jederson.zuchi@fatec.sp.gov.br
Faculdade de Tecnologia de Taquaritinga – Taquaritinga – São Paulo – Brasil

DOI: 10.31510/infa.v18i2.1288
Data de submissão: 14/09/2021
Data do aceite: 03/11/2021
Data da publicação: 30/12/2021

RESUMO

Esse artigo tem como objetivo demonstrar, por meio de referência literária, quais as vantagens de performance e custos ao utilizar o padrão de arquitetura e código limpo para o desenvolvimento de software. Tomando como base o suporte teórico, serão demonstrados exemplos de decisões arquiteturais e de desenvolvimento, bem como será feito o uso de uma ferramenta para analisar o padrão do código e verificar se o mesmo precisa ser reescrito por estar se dirigindo a um caminho não limpo ou por infligir as validações definidas pelos autores literários e pela ferramenta de análise. Com os resultados, foi possível observar a perda de produtividade e o aumento no custo do software a cada novo *deploy*, o que ocorre por conta da dificuldade de entender um software que não seguiu o padrão citado e infligiu restrições na manutenção de um código legível. Esse aumento gira em torno de 80%, pois o custo de um software é dedicado à manutenção do mesmo.

Palavras-chave: Arquitetura. Código. Limpo.

ABSTRACT

This study aims to demonstrate, through literary reference, what are the performance and cost advantages of using the clean architectural and code pattern for software development. Based on the theoretical support, examples of architectural and development decisions will be demonstrated, as well as the use of a tool to analyze the code pattern and verify if it needs to be rewritten because it is connecting to an unclean path or by inflicting as validations defined by literary authors and the analysis tool. With the results, it was possible to observe the loss of productivity and the increase in the cost of the software with each new deployment, which occurs due to the difficulty of understanding the maintenance of software that did not follow the aforementioned pattern and inflicted restrictions on the maintenance of a readable código. This increase is around 80%, as the cost of a software is dedicated to its maintenance.

Keywords: Architecture. Clean. Code.

1. INTRODUÇÃO

Para criar softwares não é preciso ter muito conhecimento, pois fazer algo funcionar uma vez é simples. É por esse motivo que alunos do ensino médio conseguem criá-los e que empresários conseguem iniciar seus negócios com apenas algumas linhas de código. (MARTIN, 2019). No entanto, criar um software da maneira correta é difícil, pois requer conhecimentos e habilidades que os jovens programadores ainda não adquiriram. (MARTIN, 2019)

Todo código contém um design, que como definido por Martin (2019) são os detalhes de implementação das funcionalidades, de como as mesmas estarão divididas dentro do software, conseguir escrever um código com um bom design, ou como dito por Fowler (2020) um código que pessoas possam entender, é o que um diferencia um bom programador.

Vale destacar que um código com design ruim geralmente exige mais códigos para fazer as mesmas tarefas, em geral, porque o código faz as mesmas coisas em pontos diferentes. (FOWLER, 2020).

Ao obter o design citado, a produtividade da equipe de desenvolvimento diminui tendendo a zero, a gerência acaba precisando tomar a decisão de contratar mais pessoas para tentar recuperar a produtividade, aumentando assim o custo de manter esse software. (Martin, 2019).

Segundo King (1999) 80% do custo de um software é dedicado à sua manutenção e como no design não limpo existe a dificuldade em realizar manutenções, esse fator de custo se junta com as possíveis novas contratações realizadas pela gerência.

Conforme vai evoluindo, o software será reorganizado e reescrito pelos desenvolvedores, pois ele não serve apenas para os usuários, mas aos programadores, para poderem criar as novas funcionalidades e realizar manutenções. (EVANS, 2017). Quando um software traz consigo um comportamento complexo e é desprovido de um bom design, torna-se difícil refatorá-lo e os desenvolvedores detestam ter que fazer alterações que possam agravar esse emaranhado ou romper alguma coisa através de uma dependência não prevista. (EVANS, 2017).

Um software precisa ser agradável para seu desenvolvedor, pois, como os autores consultados mencionam, se o mesmo tiver um comportamento complexo e dispuser de um

design ruim, cada vez menos os programadores irão querer realizar alterações ou manutenções no sistema, por receio do rompimento de alguma funcionalidade.

Quando é atingido um bom design, exige-se menos tempo de um programador para desenvolver as novas funcionalidades, mas, no início, esse processo pode ser um pouco mais demorado. Entretanto, quando o design é ruim, acontece o contrário: no começo o processo se torna mais rápido e, no final, mais lento.

1.2. OBJETIVO

O principal objetivo deste trabalho é demonstrar, por meio da literatura, quais as vantagens de criar software utilizando os conceitos de arquitetura e de código limpo.

1.3. JUSTIFICATIVA

A maioria dos programadores sabe distinguir um código limpo de um código confuso, mas nem todos conseguem criar um código limpo. Para ilustrar tal informação, podemos tomar como exemplo o processo de pintura de um quadro, visto que a maioria das pessoas sabe dizer se aquela obra foi bem pintada, mas o reconhecimento dessa informação não quer dizer que elas conseguiriam fazer uma boa pintura. (ROBERT, 2009)

Nesse sentido, é válido ressaltar que a não elaboração de um código limpo significa que não houve uma preocupação com um futuro desenvolvedor que realizará manutenções no código, e isso pode fazer com que uma alteração que duraria algumas horas, na verdade, demore dias para ser concluída. (FOWLER, 2020)

Além disso, a perda de produtividade ocorre, também, quando o programador toma decisões erradas a nível arquitetural, de maneira que a cada *release* o nível de produtividade recaia tendendo a zero e de modo que a quantidade de desenvolvedores do projeto aumente. (MARTIN, 2019)

Portanto, minimizar o custo dos recursos humanos para construir um software e manter o esforço para satisfazer a demanda do cliente baixa significa que o software tem um bom design. Porém, ao aumentar tanto o custo quanto o esforço do mesmo, significa que o design está ruim. (MARTIN, 2019)

Então, esse estudo proverá, por meio de referências teóricas, quais decisões podem ser tomadas para manter o projeto com o mínimo necessário de recursos humanos e com um custo baixo.

2. FUNDAMENTAÇÃO TEÓRICA

Nesta seção será abordado o contexto necessário para o entendimento correto das tecnologias e das análises apresentadas neste trabalho.

2.1. CÓDIGO LIMPO

Um aspecto importante no desenvolvimento de um código é o reconhecimento de que ele não estará limpo na primeira tentativa. Desse modo, Martin (2009) ressalta que as versões iniciais de um método, classe e outras estruturas nunca são uma boa solução. Assim, é necessário tempo e preocupação, desde o nome escolhido para uma variável até uma hierarquia de classes. (MACHINI et al., 2018)

Nesse sentido, em um ambiente de desenvolvimento de software de uma empresa é comum o desenvolvedor se deparar com códigos que exijam horas de análise para a compreensão de sua funcionalidade (MARTIN, 2009)

Por sua vez, para Fowler (2020), qualquer pessoa consegue escrever um código que um computador consiga entender, visto que bons programadores escrevem códigos que os humanos podem entender. Ou seja, a ideia central de um bom código é ele ser legível, assim, o mesmo deve ser escrito para minimizar o tempo que levaria para outra pessoa entendê-lo. Por isso o código mais legível é equivalente à ausência de código. (BOSWELL; FOUCHER, 2012)

Nesse sentido, saber quando não escrever um código é possivelmente a habilidade mais importante que um programador pode aprender, pois cada linha de código que é escrita é uma linha a mais que deve ser testada e mantida. (BOSWELL; FOUCHER, 2012)

Portanto, o código precisa ser legível e minimizar o tempo de outra pessoa na tentativa de entender o que foi feito, ou seja, como destacado, é necessário ter menos código, ou não haver código, pois essa característica colaborará para a agilidade desse processo. Por isso, um

bom programador também precisa saber quando não escrever um código, de modo a não adicionar mais testes para manter o aquele sistema, quando for possível evitar.

2.2. ARQUITETURA LIMPA

A arquitetura representa as decisões de um software realizado em seu desenvolvimento resultando em sua estrutura atual, como sua divisão de camadas e organização do código, com o propósito de facilitar seu desenvolvimento, implementação, operação e manutenção. (MARTIN, 2019)

Para a arquitetura ser considerada limpa, essas decisões tomadas de estrutura devem facilitar os testes no software e ser independentes de estruturas, IU, banco de dados, e de qualquer fator externo, mantendo assim toda as regras de negócio imparcial as tecnologias escolhidas. (MARTIN, 2019)

A arquitetura, não deve conter dependências diretas de ferramentas, ou seja, ao optar por utilizar um banco dados MySQL e trocar para um NoSQL, por exemplo, essa mudança deverá semente se restringir à camada de dados.

O objetivo da arquitetura é minimizar ao máximo o uso de qualquer recurso para construir e manter um sistema, sendo esses humanos ou performance, para realizar alguma alteração quando necessário (MARTIN, 2019). Assim, a arquitetura tem a mesma premissa do código limpo, que seria manter um sistema com poucas pessoas, o que deve ocorrer sem que seja prejudicada sua produtividade, evitando que o custo aumente a cada *deploy*.

2.2.1. ARQUITETURA HEXAGONAL

Um exemplo de arquitetura limpa é a *Hexagonal architecture* (Arquitetura hexagonal), também conhecida como *Ports & Adapters* (Portas e adaptadores), a qual permite que um aplicativo seja desenvolvido e testado de modo isolado em relação aos eventuais dispositivos em tempo de execução e bancos de dados. (COCKBURN, 2005)

Ocorre que, conforme os eventos chegam do mundo externo à uma porta do sistema, um adaptador específico da tecnologia converte-os em uma chamada de procedimento ou em uma mensagem utilizável e a passa para o aplicativo. Na sequência, o aplicativo ignora

felizmente a natureza do dispositivo de entrada e, quando o aplicativo tem algo para enviar, o faz por meio da transmissão de uma porta para um adaptador, que cria os sinais apropriados necessários para a tecnologia receptora (humana ou automatizada). Assim, o aplicativo tem uma interação semanticamente sólida com os adaptadores em todos os lados, sem realmente conhecer a natureza daquilo que está do outro lado dos adaptadores. (COCKBURN, 2005)

Portanto, essa arquitetura tem como objetivo o não reconhecimento do mundo exterior que está em desacordo com as regras de negócios. Como dito pelo seu criador, Cockburn, qualquer evento que chega para o aplicativo é convertido antes de ir para o mesmo.

3. ESTUDO DE CASO

Nesta seção serão abordadas ferramentas e decisões sobre como manter o código, a arquitetura e o design de um projeto, de forma que os custos não aumentem e a produtividade não diminua.

3.1. DEFININDO A ARQUITETURA

Como já mencionado, é preciso definir qual arquitetura será utilizada no projeto, visto que isso irá refletir em todo o seu desenvolvimento, incluindo a criação dos componentes e de classes.

Assim, uma arquitetura precisa permitir que um aplicativo seja conduzido, de modo igualitário, por usuários, programas, testes automatizados ou scripts em lote, além de ser desenvolvido e testado isoladamente de seus eventuais dispositivos de tempo de execução e de bancos de dados. (COCKBURN, 2005)

Dessa maneira, um exemplo de arquitetura que pode ser usado é a arquitetura hexagonal, pois, como já mencionado, a mesma isola as lógicas de negócio das influências externas em portas e adaptadores. Então, essa escolha se baseia nos conceitos (também já citados) de arquitetura limpa, pois todo o componente externo, o banco de dados, a interface de usuário ou quaisquer outras ferramentas não terão influência nas regras contidas nessa arquitetura.

Então, em um programa orientado a objetos, o banco de dados e outros códigos de suporte geralmente são escritos diretamente nos objetos do negócio. Além disso, as outras

lógicas do negócio são embutidas no comportamento *scripts* do banco de dados e nas interfaces do usuário. Isso acontece porque é a maneira mais fácil de fazer com que as coisas funcionem a curto prazo. (EVANS, 2017)

3.2. CÓDIGO

Como dito pelos autores consultados para a realização do presente artigo, um código limpo precisa ser legível para outros desenvolvedores, minimizando, assim, o tempo necessário para que uma outra pessoa (que não seja necessariamente o programador) o possa entender. Essa é a ideia principal a ser considerada na criação de um bom código e, por isso, segundo Boswell e Foucher (2012), "o código mais legível é não ter código".

No entanto, para atingir tal efeito, é preciso evitar os "Maus cheiros" no código (FOWLER, 2020), pois, quando eles aparecem, é um sinal de que o código precisa ser refatorado. Assim, um dos indicativos que apontam para a necessidade desse procedimento é quando uma classe está com uma quantidade de linhas elevada (FOWLER, 2020). Ou seja, um arquivo com uma quantidade anormal de linhas precisa ser refatorado para que possa alcançar a menor quantidade de linhas possível. Segundo Boswell e Foucher (2012), um arquivo de 2000 linhas é mais fácil de entender do que um arquivo de 5000 linhas.

No livro *Refatoração* (2020), de Fowler, são citados 24 indicativos de que é preciso ter cuidado ao escrever um código. Além disso, existem ferramentas que auxiliam a manter o código livre desses indicativos, são os chamados *Linters.*

3.2.1. LINTER

Existem diversos *linters* e alguns deles estão inseridos em uma lista no Github, chamada de *Collections.* Além disso, como definido por *checkstyle*, uma ferramenta de linter para código Java com mais de seis mil de estrelas no Github, *linters* são ferramentas de desenvolvimento para ajudar os programadores a escrever códigos que sigam um padrão de codificação.

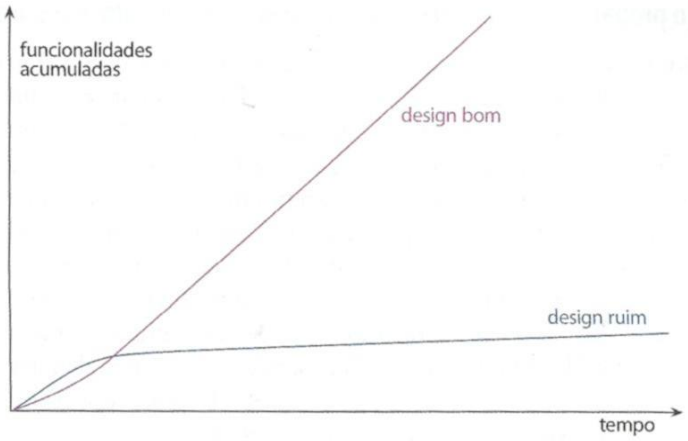
Essa ferramenta tem validações, entre as quais se pode citar os indicativos já mencionados por Fowler (2020), como, por exemplo, o *MethodLength.* Esse indicativo é descrito na documentação oficial como a verificação da existência de métodos e de construtores

longos. Ademais, segundo a documentação, o valor para a validação citada é de 150 linhas, mas o mesmo é personalizável, pois, como defendido por Robert (2008), “o conceito de código limpo é variável”.

Essas convenções de código são importantes, visto que 80% do custo de vida de um software destina-se à manutenção, sendo que quase nenhum software é mantido pelo autor de origem (ou seja, pelo criador). Assim, as convenções de código melhoram a legibilidade do software, permitindo que os engenheiros entendam o novo código de maneira mais rápida e completa. Então, se o código for enviado como um produto, é preciso ter certeza de que ele está tão bem embalado e limpo quanto qualquer outro produto que você criar. (KING et al., 1999)

4. RESULTADOS E DISCUSSÃO

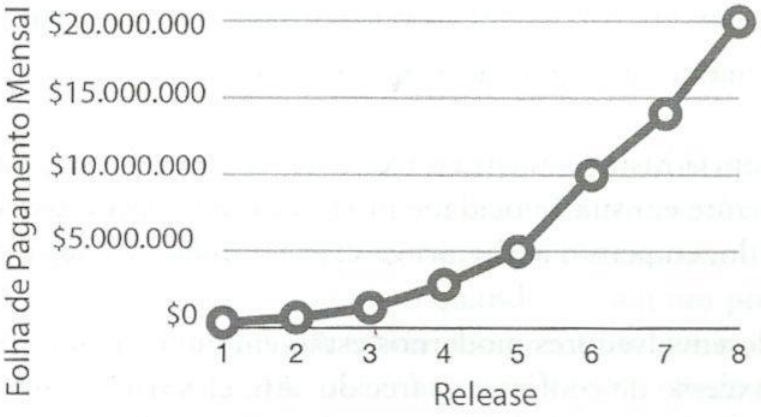
O presente estudo permitiu-nos verificar que, ao utilizar definições limpas e boas práticas de arquitetura e de design para desenvolver um software, a quantidade de desenvolvedores necessários para mantê-lo diminui de modo significativo. Além disso, outro benefício conquistado a partir da utilização de boas estratégias - as quais foram relacionadas nas etapas da pesquisa - seria o ganho na performance, pois é possível entregar mais funcionalidade em menos tempo, lembrando que isso não ocorre quando se trata de um design ruim, como se pode observar no gráfico a seguir:



Fonte: FOWLER, 2020.

Ademais, o custo do software por *deploy* não aumenta, já que não é preciso fazer contratações de desenvolvedores para entregar as funcionalidades dentro do prazo. Então, como mostram os gráficos a seguir, ao não utilizar esses padrões, amplia-se o custo do software, enquanto sua produtividade decai.

Fonte: MARTIN, 2019.



Fonte: MARTIN, 2019.

Como já mencionado pelo site da Oracle, 80% do custo de um software é destinado à manutenção do mesmo e, como já demonstrado no gráfico, quanto mais difícil de manter um software, mais seu custo tende a aumentar. Portanto, também é possível utilizar ferramentas específicas para ajudar a conservar um código limpo, de modo que os *linters* são uma delas, visto que ajudam a manter o código dentro de padrões que podem ser personalizados e utilizados da melhor maneira, independente do contexto.

5. CONCLUSÃO

Ao fazer uso de boas decisões e utilizar conceitos limpos enquanto se está desenvolvendo um software, é possível economizar horas de projetos e dinheiro para a manutenção do mesmo. Assim, optando-se por utilizar esses conceitos, o começo do projeto terá um ritmo de entrega menor, mas o mesmo aumentará e se manterá em crescimento.

Entretanto, ocorre o contrário quando não são utilizados os conceitos limpos no nascimento do software: o tempo de entrega é mais rápido, mas esse ritmo diminui e tende a cair cada vez mais, pois um software bagunçado, quando não se tem ações de refatoração, torna-se confuso e gera mais trabalho.

Dessa maneira, o código do software precisa ter como objetivo a minimização do tempo de entendimento, de modo que qualquer programador, ao manter aquele sistema posteriormente, tenha condições de reconhecê-lo com a mesma eficiência e rapidez obtida por aquele que o escreveu/elaborou antes.

Além disso, um software precisa manter-se limpo e organizado para não aumentar o seu custo de manutenção, visto que esse é o setor que mais gera despesas. Então, para que se alcance tal objetivo, um software deve servir, não apenas ao seu cliente final, como também aos seus desenvolvedores, sendo necessárias ações e ferramentas capazes de mantê-lo com um custo baixo e um ritmo de entregas alto.

Por isso, com boas decisões de arquitetura, design e com a manutenção de um código, bem como o seu objetivo, o mais claro possível, pode-se alcançar a manutenibilidade, o ritmo de entrega e o custo do software em proporções agradáveis.

6. REFERÊNCIAS BIBLIOGRÁFICAS

BOSWELL, Dustin; FOUCHER, Trevor. **The Art of Readable Code.** [S. l.] O'Reilly, 2012.

COCKBURN, Alistair. **Hexagonal architecture**. Disponível em: <https://alistair.cockburn.us/hexagonal-architecture >. Acesso em: 23 out. 2021.

Dietz, Linus W. et. al. **Teaching Clean Code**. Disponível em: <https://mediatum.ub.tum.de/doc/1428241/1428241.pdf >. Acesso em: 23 out. 2021.

EVANS, Eric. **Domain-Driven Design: Atacando as complexidades no coração do software.** [S. l.] Alta Books, 2017.

FOWLER, Martin. **Refatoração: Aperfeiçoando o Design de Códigos Existentes.** [S. l.] Novatec, 2020.

KING, Peter et. al. **Code Conventions for the Java Programming Language**. Disponível em: <https://www.oracle.com/java/technologies/javase/codeconventions-introduction.html>. Acesso em: 27 out. 2021.

MACHINI, Joao et. al. **Um Estudo de Caso do Mapeamento dos Conceitos de Codigo Limpo para Métricas de Código-fonte**. Disponível em:<https://www.ime.usp.br/~cef/mac499-10/monografias/lucianna-joao/arquivos/monografia.pdf>. Acesso em: 25 out. 2021.

MARTIN, Robert C. **Arquitetura limpa: O guia do artesão para estrutura e design de software.** [S. l.], Alta Books 2019.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software**. [S. l.]: Alta Books, 2009.