Trabalho de Conclusão de Curso

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Como as mudanças na arquitetura de um software abraçadas durante o desenvolvimento com metodologia ágil impacta na manutenibilidade do sistema?

Autor: Beatriz Ferreira Gonçalves

Brasília, DF

2015

Beatriz Ferreira Gonçalves

Como as mudanças na arquitetura de um software abraçadas durante o desenvolvimento com metodologia ágil impacta na manutenibilidade do sistema?

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Brasília, DF

2015

Beatriz Ferreira Gonçalves

Como as mudanças na arquitetura de um software abraçadas durante o desenvolvimento com metodologia ágil impacta na manutenibilidade do sistema?/ Beatriz Ferreira Gonçalves. – Brasília, DF, 2015-

?? p. : il. (algumas color.) ; 30 cm.

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. Arquitetura de Software. 2. Metodologia Ágil. 3. Manutenibilidade. 4. Engenharia de Software. I. . II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Como as mudanças na arquitetura de um software abraçadas durante o desenvolvimento com metodologia ágil impacta na manutenibilidade do sistema?

CDU 02:141:005.6

Beatriz Ferreira Gonçalves

Como as mudanças na arquitetura de um software abraçadas durante o desenvolvimento com metodologia ágil impacta na manutenibilidade do sistema?

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 01 de junho de 2013:

Orientador

Titulação e Nome do Professor
Convidado 01
Convidado 1

Titulação e Nome do Professor
Convidado 02
Convidado 2

Brasília, DF
2015

Resumo

A arquitetura de software pode ser dita como uma atividade de fundamental importância na construção de um software: é durante esta atividade que, uma vez entendidos, os problemas e necessidades que o software atenderá serão modelados, ou seja, os elementos contidos no software serão identificados e relacionados para que a atividade posterior, de implementação, possa ser realizada com sucesso. Uma boa arquitetura permite a inserção, modificação ou retirada de funcionalidades de um sistema de software sem que seja necessário despender-se de enorme esforço, ou seja, facilita a manutenibilidade do sistema. A metodologia ágil tem como um de seus princípios a aceitação de requisições de mudanças e adição de novas funcionalidades e procura tratar a mutabilidade constante do software. Tais mudanças no software acarretam em mudança na arquitetura do mesmo e, quando não controladas, podem levar o desenvolvimento e manutenção do software ao caos. A presente pesquisa visa apontar possíveis maneiras de como estas modificações constantes durante o desenvolvimento de software pode afetar a fase de manutenção e as subcaracterísticas inerentes à característica de manutenibilidade do sistema.

Palavras-chaves: 1. Arquitetura de Software. 2. Metodologia Ágil. 3. Manutenibilidade 4. Engenharia de Software.

Abstract

Software architecture can be defined as a software development fundamental activity: it is during this activity that, once understood, problems and needs the software ought to satisfy are designed. In other words, it is over the software architecture activity that the software elements are identified and related to the accomplishment of a successful and well done code implementation. Well-designed software architecture allows insertions, changes and removal of functionalities from/to software systems without much effort, making the maintainability of the system easier. The agile methodology has as one of its principles accepting changes requests and new functionalities, dealing with constant software mutability. Such changes cause software architecture changes, and, when those changes are uncontrolled, the software development and maintenance might be lead to chaos. This research aims to indicate possible ways that constant changes during software development can become a matter of concerns related to the maintenance phase and attributes inherent to systems maintainability.

Key-words: 1. Software Architecture. 2. Agile Methodology. 3. Maintainability. 4. Software Engineering.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O processo de desenvolvimento de *software*, segundo Sommerville (2003), compreende, independentemente do modelo de processo adotado, as fases de especificação, projeto e implementação, validação e manutenção e evolução de software: após a definição de funcionalidades e restrições do sistema na fase de especificação, a modelagem e implementação do sistema é realizado de acordo com os dados sobre as funcionalidades e restrições definidos na fase de especificação do sistema; uma vez implementado, o sistema deverá ser validado (fase de validação) e posteriormente mantido e evoluído conforme a demanda (SOMMERVILLE, 2004).

Dentre as atividades de modelagem temos o desenvolvimento da arquitetura de *software*, geralmente realizada entre atividades de compreensão do problema ou necessidade para o qual o *software* será construído (especificação) e a sua implementação. Arquitetura de *software* é definida como "a estrutura ou estruturas de sistema que comprimem elementos de *software*, as propriedades visivelmente externas de tais elementos e o relacionamento entre eles" (BASS, CLEMENTS, KASMAN, 2003).

Além de modelos de processo, o desenvolvimento de *software* é executado adotando-se uma metodologia de desenvolvimento. No princípio, foi adotada a metodologia tradicional para desenvolvimento, que procura planejar e documentar todo o *software* para posterior implementação, processo geralmente aplicado à produção de produtos visíveis e tangíveis, como automóveis e aeroplanos (PRESSMAN, 2006). Sendo os produtos de *software* descritos por Brooks (1987) como invisíveis e por isto mutáveis e complexos, que devem estar em conformidade com requisitos pré-estabelecidos pelo usuário ou cliente, à necessidade de uma metodologia de desenvolvimento que melhor se adequasse a estas características foi proposta no ano de 2001 a partir do Manifesto Ágil, escrito por Kent Beck e outros 16 estudiosos, uma nova metodologia a ser aplicada ao desenvolvimento de *software*: a metodologia ágil.

Por ser um produto invisível e intangível, há uma dificuldade de idealizar tanto o processo quanto o produto de software como coisas reais e tangíveis e, portanto, a mutabilidade se destaca como uma característica onipresente em produtos de *software* (BROOKS, 1987; OSTERWEILL, 1987). A metodologia ágil visa "satisfazer o cliente desde o início por meio da entrega contínua de *software* valioso" e trata a mutabilidade do *software* como algo natural, uma vez que é uma propriedade essencial do produto invisível de *software*, e aceitando todas e quaisquer mudanças, mesmo em momentos tardios do desenvolvimento (PRESSMAN, 2006).

Mudanças no *software* significam mudanças na arquitetura do mesmo, que deve

estar preparada para receber tais mudanças e é um elemento que deve ser sempre considerado ao aprovar mudanças sem análise prévia de seus impactos. Tais mudanças na arquitetura do *software* afetam a manutenibilidade do *software*, seja tal modificação feita para adaptação, correção ou para atender requisições do usuário (ROMBACH, 1990). Entenda por manutenibilidade a definição feita pela ISO/IEC 25010/2011 como "o grau de eficácia e eficiência com o qual um produto ou sistema pode ser modificado".

A presente pesquisa visa responder à questão "*Como as mudanças na arquitetura de um software impactam na manutenibilidade do sistema quando utilizada a metodologia ágil para o desenvolvimento?*", visto que a metodologia ágil prega a adaptabilidade do sistema para quaisquer modificações requisitadas que significam mudanças na arquitetura do *software*. Quando a arquitetura do sistema é desenvolvida de maneira a ser suficientemente adaptativa a tais mudanças, pode-se presumir que a manutenção a feita no *software* após sua implantação pode ser realizada sem grandes dificuldades.

A resposta que deseja-se encontrar durante a execução deste trabalho tem como objetivo central apontar de que forma estas mudanças na arquitetura afetam a manutenibilidade do *software* quando adotada a metodologia ágil para o desenvolvimento do mesmo.

Ao indicar tais impactos, a sociedade envolvida no desenvolvimento e qualidade dos produtos de *software* poderá voltar-se aos problemas encontrados, pesquisando por soluções adequadas que os resolvam ou apontando soluções já existentes.

1.1 Organização

Este trabalho foi organizado de modo a proporcionar ao leitor as principais ideias a respeito dos objetos tratados por este estudo, que será dividido em quatro diferentes partes. A primeira parte da pesquisa realizada tratará do conceito de arquitetura de *software* bem como os princípios definidos pela literatura selecionada para a construção de uma arquitetura e design ótimo de um *software*.

Durante a segunda parte será abordada a metodologia ágil e os princípios ágeis de desenvolvimento, com enfoque maior no princípio de aceitação de mudanças a qualquer momento do desenvolvimento. Na penúltima parte da pesquisa, serão abordados os conceitos de manutenibilidade e manutenção de *software*.

Por fim, na última parte deste trabalho, serão tratadas as considerações finais sobre a conclusão da revisão sistemática realizada, visando responder parte da questão aqui proposta para investigação.

A metodologia adotada com todos os passos para a execução da pesquisa pode ser encontrada no capítulo final deste trabalho.

2 Arquitetura de Software

Podemos encontrar em citações feitas por vários autores a assertiva de que software geralmente é construído a partir de uma oportunidade de negócio ou da identificação de necessidades de usuários. Como anteriormente mencionado, a arquitetura de software é um composição de estruturas de sistema que exibem as características visíveis de elementos de software e o relacionamento e entre tais elementos. Além disso, a arquitetura de software nada mais seria do que uma "ponte" que conecta as necessidades ou oportunidades identificadas e o software ou sistema em si, devendo tal arquitetura ser desenhada, documentada e analisada para posterior implementação (BASS, CLEMENTS, KASMAN, 2003).

Sendo uma abstração do sistema a ser desenvolvido, a arquitetura de software exibe os detalhes julgados como necessários pelo arquiteto, bem como o comportamento do software. Isto permite a conclusão de que todo software possui uma arquitetura definida, sendo ela documentada ou não (BASS, CLEMENTS, KASMAN, 2003).

2.1 Tipos de estruturas arquiteturais de software

Primeiramente, é necessário a definição e diferenciação entre os termos visão e estrutura quando o assunto em questão é a representação da arquitetura de software. Kazman, Clements e Bass (2003) definem estrutura como o "conjunto de elementos de um sistema de software" e visão como a "representação do conjunto de elementos do sistema e o relacionamento entre eles, escrita e lida pelos stakeholders". Ainda defendem categorização das visões de software existentes: arquitetura baseada em estrutura de decomposição em módulos, baseada em componentes e conectores e em estruturas de alocação.

De acordo com Kazman, Clements e Bass (2003), a arquitetura baseada em decomposição do sistema em módulos procura representar o sistema de modo estático, de modo que os elementos do sistema são módulos definidos como classes, camadas ou, simplesmente, divisões de funcionalidades. Definindo o sistema em módulos, a análise de impactos de mudanças no sistema pode ser feita ao examinar a estrutura de módulos do sistema.

A arquitetura baseada em componentes e conectores procura demonstrar como os elementos do sistema irão interagir uns com os outros em tempo de execução, se caracterizando uma arquitetura dinâmica. É bastante útil quando características não funcionais, como performance e segurança, são analisadas. A última categoria definida, a estrutura

arquitetural de alocação, define como o sistema de software irá interagir com outros elementos que não são software, como harware e o time de desenvolvimento, permitindo a distribuição de atividades entre os membros da equipe de desenvolvimento e a definição de qual processador determinado elemento irá ser executado (KAZMAN, CLEMENTS, BASS, 2003).

É importante mencionar que a categoria para representação de estruturas de software escolhida como objeto de análise desta pesquisa é a estrutura de decomposição em módulos. Todos os princípios e definições a partir daqui feitas serão baseadas nesta categorização.

2.2 Princípios de construção de arquitetura de software baseado em orientação a objetos

Orientação a objetos é um dos paradigmas existentes para a construção de sistemas de software. Este paradigma procura retratar objetos do mundo real como objetos lógicos dentro da estrutura do sistema de software. Tais objetos são defnidos por classes que possuem atributos e métodos, representando as características e comportamentos dos objetos que estas classes representam (DALL'OGLIO, 2009). O relacionamento entre os elementos desta estrutura orientada a objetos é representada por uma arquitetura que deve ser feita baseada em princípios de orientação a objetos e princípios que guiam a elaboração de uma boa arquitetura de software. Serão descritos a seguir os princípios de orientação a objetos, problemas que podem surgir caso a arquitetura orientada a objetos não seja adequada e princípios que devem ser seguidos de modo a evitar estes problemas.

2.2.1 Princípios de Orientação a Objetos

PAGE-JONES (2000) define cinco principais princípios de orientação a objetos, sendo eles encapsulamento, coesão, acoplamento e herança, de onde surgem as subclasses e subtipos.

2.2.1.1 Encapsulamento

Segundo Page-Jones (2000), o encapsulamento é um princípio de orientação a objetos que permite uma maior abstração do sistema, exibindo o que deve ser feito e não como. A encapsulação pode ser feita em cinco diferentes níveis:

- Nível 0: não existe encapsulamento.
- Nível 1: encapsulamento dentro de módulos procedurais.
- Nível 2: encapsulamento dentro de classes.

- Nível 3: encapsulamento dentro de pacotes.
- Nível 4: encapsulamento por componentes do sistema.

2.2.1.2 Coesão

Coesão é definida como o grau de afinidade entre procedimentos (métodos e atributos) dentro de cada módulo do sistema (PAGE-JONES, 2000). Quando classes ou módulos do sistema são fortemente coesos as características destes estão relacionadas de modo a contribuir para a melhor compreensão da abstração implementada pela classe.

2.2.1.3 Acoplamento

Acoplamento é definido por Page-Jones (2000) como uma característica que relacionam dois ou mais elementos de modo que qualquer mudança realizada em um elemento do sistema obrigará a verificação ou até mesmo a modificação de outros elementos relacionados. O ideal seria que os elementos de um sistema estejam fortemente coesos e possuam acoplamento fraco (PAGE-JONES, 2000).

2.2.1.4 Herança - Subclasses e Subtipos

Herança é um princípio de orientação a objetos que permite o compartilhamento de características entre os elementos de um sistema a partir da derivação (ou generalização) de tais elementos: uma superclasse do sistema engloba todas as características comuns das subclasses derivadas, sendo a subclasse, na maioria dos casos, um subtipo do objeto definido da superclasse (PAGE-JONES, 2000). Page-Jones (2000) define que um objeto A somente é subtipo de um objeto B se um objeto A pode ser utilizado em qualquer contexto onde é esperado um objeto B sem que falhas ou comportamentos inesperados ocorram. Quando utilizado o paradigma de orientação a objetos para a construção de um sistema de software, espera-se que toda subclasse do sistema também seja um subtipo, embora esta premissa não ocorra em todos os casos.

2.2.2 Características de uma arquitetura de software pobre

Dada um arquitetura de software mal planejada e implementada sem qualquer técnica ou procedimentos sistemáticos, podem surgir problemas que dificultam modificações no software. Martin (2000) aponta quatro principais sintomas de que a arquitetura de um software possui problemas:

1. Rigidez: dificuldade para realização de mudanças, onde qualquer mudança em uma parte do software implica na necessidade de mudanças em vários outros módulos independentes do primeiro.

2. Fragilidade: relacionada à rigidez, qualquer mudança no software tende "quebrá-lo" em várias outras partes e de maneiras inesperadas.

3. Imobilidade: dificuldade de reuso de partes do software em outros softwares ou até mesmo no mesmo software por dependências entre os módulos/partes do sistema implementado.

4. Viscosidade: é identificado quando existem diferentes formas de realizar as mudanças na arquitetura de software e é sempre mais fácil realizar tais modificações de maneira errada.

2.2.3 Princípios para desenho de arquitetura orientada a objetos

Afim de propor soluções para que problemas de arquitetura de software pobres sejam resolvidos ou até mesmo que estes problemas sejam evitados durante a concepção de tais modelos de implementação, Martin (2000) nomeou cinco princípios para desenho e implementação de software orientado a objetos. Também conhecidos pelo acrônimo SOLID, estes princípios são:

- **S**ingle Responsability Principle (Princípio de Responsabilidade singular ou individual): define que uma classe, pacote ou módulo do sistema deve conter apenas uma única responsabilidade. Caso este aspecto não esteja implementado, existe o acoplamento de classes, pacotes ou módulos, tornando o sistema frágil quanto à realização de modificações.

- **O**pen/Closed Principle (Princípio "Aberto/Fechado"): define que qualquer módulo do sistema de software deve ser aberto para extensão do seu comportamento sem que mudanças sejam necessárias para tal extensão. Isso permite que novas funcionalidades sejam adicionadas ao software sem que haja a necessidade de modificação na arquitetura já existente.

- **L**iskov Substitution Principle (Princípio de Substituição de Liskov): define que "subclasses devem ser substituíveis por suas classes base", ou seja, se A é subclasse ou subtipo de B, então A pode substituir B quando o tipo B é esperado, preservando a corretude do sistema implementado.

- **I**nterface Segregation Principle (Princípio de Segregação de Interface): define que se existe um módulo utilizado por vários clientes de maneiras diferentes e ao mesmo tempo, interfaces específicas (e abstratas) devem ser criadas para cada cliente.

- **D**ependency Inversion Principle (Princípio de Inversão de Dependência): define que módulos, pacotes e classes do sistema de software devem ser desenhados de modo que

a dependência existente entre estes seja definida por meio de interfaces ou funções e classes abstratas ao invés de utilizar-se de funções e classes concretas.

3 Metodologia Ágil

Também chamada de metodologia "leve" pelos próprios criadores, a metodologia ágil não se declara "anti-metodológica" e procura estabelecer um balanceamento entre a confecção de documentação e o planejamento pesado da produção de *software* e a produção de *software* de fato. O "Manifesto Ágil para o desenvolvimento de *Software* Ágil", assinado em 2001 por Kent Beck e outros 16 produtores, desenvolvedores e consultores de *software*, declara as seguintes palavras:

> "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."

Sendo mais facilmente aplicável à pequenas e médias organizações, a metodologia ágil mais difundida é a *Extreme Programming*, que se baseia em requisitos simples que se modificam de forma rápida (BECK, 1999). Segundo BECK (1999), os quatro valores que norteiam esta metodologia são comunicação, simplicidade, *feedback* e coragem. Estes quatro valores baseiam-se nas ideias de que a comunicação entre membros da equipe e entre equipe e clientes deve ser efetiva e feita pessoalmente sempre que possível, sendo o principal fator para o sucesso do projeto. A simplicidade diz respeito à implementação da solução: tornar simples a solução atual, mas permitindo que modificações futuras possam acontecer ao invés de implementar algo complexo que jamais será utilizado. O *feedback* significa que o cliente terá sempre uma parte do *software* funcional para avaliar e que a equipe sempre terá uma resposta dos testes realizados e também do cliente para a melhoria da qualidade do produto. Para a prática dos três outros valores, o quarto se faz necessário: a coragem (SOARES, 2004).

3.1 Princípios da metodologia ágil

Além das quatro propostas descritas no "Manifesto Ágil para o desenvolvimento de *Software* Ágil", seus autores também propóem por este documento os 12 princípios fundamentais da metodologia ágil:

1. "Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de *software* de valor" (AGILE MANIFESTO, 2001). O Manifesto Ágil acredita que documentos de especificações e arquitetura de *software*, por exemplo, são importantes para o desenvolvimento de software, mas o que importa ao cliente, de fato, é a funcionalidade do *software* implementada, provando que a aplicação é um bom investimento para sua empresa.

2. "Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas" (AGILE MANIFESTO, 2001). Sendo o fututo imprevisível, a abordagem ágil visa aceitar requisições de mudanças no *software* mantendo a consciência das possíveis consequências.

3. "Entregar *software* funcionando com freqüencia, na escala de semanas até meses, com preferência aos períodos mais curtos" (AGILE MANIFESTO, 2001). Este princípio está atrelado à satisfação do cliente, pois a partir de um modelo de desenvolvimento iterativo e incremental é possível entregar *software* (partes dele) em períodos curtos, entregar valor ao cliente em períodos curtos.

4. "Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diáriamente, durante todo o curso do projeto" (AGILE MANIFESTO, 2001). A metodoglia ágil trabalha a partir de requisitos com alto nível de abstração que, obviamente, não são suficientes para a realização completa do desenho e arquitetura da funcionalidade e posterior codificação. Assim, o intenso relacionamento entre os *stakeholders* do projeto em questão.

5. "Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho" (AGILE MANIFESTO, 2001). Sendo o produto de *software* produzido por seres humanos, são estas pessoas que determinam o sucesso ou o fracasso de um projeto de *software*. Indivíduos motivados e confiantes podem colaborar para o sucesso do projeto.

6. "O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara" (AGILE MANIFESTO, 2001). Documentos escritos podem ser ambíguos, mesmo que escritos da melhor maneira julgada pelo autor: conversas cara a cara devem resolver tal problema, visto que todas as dúvidas e confirmações sobre o assunto tratado são conversadas e não escritas.

7. "*Software* funcional é a medida primária de progresso" (AGILE MANIFESTO, 2001). Sem a abstração das características do sistema em um *software* funcional não é possível constatar se o projeto terá sucesso.

8. "Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes" (AGILE MANIFESTO, 2001). Passos constantes, no contexto de metodologia ágil, dizem respeito à jornada de trabalho da equipe: ao passo que a jornada foi definida, ela deve ser cumprida, evitando ao máximo horas extras e desgaste da equipe.

9. "Contínua atenção à excelência técnica e bom design, aumenta a agilidade" (AGILE MANIFESTO, 2001). A abordagem ágil de desenvolvimento visa manter qualidade da arquitetura e do desenho do *software* reconhecendo que isto é vital para a manutenção da agilidade do desenvolvimento. O trabalho de desenhar a arquitetura do *software* ocorre em todas iterações do desenvolvimento.

10. "Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito" (AGILE MANIFESTO, 2001). Uma vez que a metodologia ágil aceita mudanças nos requisitos do *software* facilmente, manter a simplicidade do *software* colabora para que tais modificações sejam feitas de forma mais fácil e menos complexa.

11. "As melhores arquiteturas, requisitos e *designs* emergem de times auto-organizáveis" (AGILE MANIFESTO, 2001). Sendo arquiteturas, requisitos e desenhos advindos da criatividade humana, quando a interação entre membros da equipe é alto e as regras são poucas a criatividade é maior e os produtos são de maior qualidade.

12. "Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo" (AGILE MANIFESTO, 2001). A equipe deve sempre procurar monitorar-se e molhorar seus próprios métodos e processo de desenvolvimento.

4 Manutenção de Software

Segundo Sommerville (2004), manutenção de software "é o processo de modificação de um sistema depois que este foi colocado em uso". Os estudos realizados por Lehman e Belady (1985) sobre mudanças do sistema propõem um conjunto de leis sobre o assunto, conhecidas como Leis de Lehman. A lei de maior relevância para o tópico aqui discutido é a primeira lei, que define a manutenção de *software* como um processo inevitável, pois ao passo que determinado sistema é implantado no ambiente destinado e este ambiente modifica-se, a necessidade de mudança no sistema também ocorre.

Para que tal processo ocorra, existem diferentes estratégias cada qual com um objetivo distinto. Estas estratégias são manutenção de *software*, transformação da arquitetura e reengenharia de *software* (SOMMERVILLE, 2004).

Sommerville (2004) cita além das estratégias que podem ser seguidas para execução doprocesso de manutenção de software, este também possui três classificações distintas:

- **Manutenção corretiva:** visa identificar e corrigir erros encontrados no sistema.
- **Manutenção adaptativa:** procura adaptar o sistema a um novo ambiente operacional.
- **Manutenção perfectiva:** tem como objetivo atender à solicitação mudanças de funções existentes ou inclusão de mudanças existentes, bem como melhorar o *software* de uma maneira geral.

4.1 Manutenibilidade

Como anteriormente citado, a norma SQuARE (ISO/IEC 25010/2011) define manutenibilidade como "o grau de eficácia e eficiência com o qual um produto ou sistema pode ser modificado". Este modelo de qualidade de produto de software divide esta característica em outras cinco subcaracterísticas:

- **Modularidade:** grau com o qual um sistema é composto por diferentes módulos interrelacionados de modo que qualquer mudança em um destes módulos impacta na mudança de outro(s) módulo(s) (ISO/IEC 25010/2011).
- **Reusabilidade:** grau com o qual um recurso do sistema pode ser utilizado em outros sistemas (ISO/IEC 25010/2011).

- **Anasabilidade:** grau de efetividade e eficiência com o qual é possível identificar os impactos de uma mudança no sistema, suas deficiências ou causas de falhas e partes a serem modificadas (ISO/IEC 25010/2011).
- **Modificabilidade:** grau com o qual um sistema pode ser modificado sem que novos defeitos sejam introduzidos (ISO/IEC 25010/2011).
- **Testabilidade:** grau de efetividade e eficiência com o qual os critérios de teste podem ser estabelecidos e executados (ISO/IEC 25010/2011).

Dentre os diversos fatores existentes que podem afetar a manutenibilidade do *software* podemos destacar a arquitetura, a documentação, tecnologias empregadas no desenvolvimento e a compreensibilidade do programa (BRUSAMOLIN, 2004).

5 Considerações Finais

De acordo com Rombach (1990), o desenho da arquitetura do *software* tem mais influência na manutenibilidade do mesmo do que o desenho e implementação do algoritmo do *software*. Diversos experimentos foram realizados com o objetivo de averiguar a afirmação feita por Rombach. Em um de tais experimentos, foram elaboradas versões de um único sistema, cada versão com uma arquitetura diferente, mas utilizando-se as mesmas tecnologias. Dos resultados de qualidade obtidos, verificou-se o aumento da manutenibilidade do software quando é despendido um tempo considerável na arquitetura de *software* (BRUSAMOLIN, 2004).

Visto que dentre os princípios da metodologia ágil existe uma preocupação contínua em conservar a auto-organização do time de desenvolvimento de modo a produzir melhores requisitos, arquiteturas e *designs*, além de manter a atenção na construção de um bom *design*, reconhecendo que a qualidade da arquitetura e do desenho de *software* são importantes para a manutenção da agilidade do desenvolvimento e, consequentemente, da manutenibilidade do mesmo é possível afirmar que a arquitetura de *software* é impactada de forma sempre positiva, mesmo que o incremento da arquitetura seja realizado em toda iteração.

Desta forma, podemos citar como impacto visível na manutenibiliade de *software* quando este é construído no contexto ágil de desenvolvimento uma maior qualidade da arquitetura construída: a arquitetura é, geralmente, desenvolvida com base nos princípios SOLID aqui citados, uma vez que a aceitação de mudanças é facilitada e o modelo adotado é iterativo e incremental, ocorrendo entrega constante de *software* ao cliente. Entregando *software* (partes dele) em períodos curtos e incrementando a solução já existente, o desenvolvimento de software, após a primeira entrega, se assemelha à manutenção perfectiva e adaptativa, ao modificar ou incluir novas funcionalidades ao sistema de acordo com as requisições realizadas pelo cliente ou usuário e tais requisições podem ou não partir de uma mudança do ambiente operacional no qual parte do *software* entregue está inserido.

6 Metodologia

Segundo Kitchenham (2007), a revisão sistemática de literatura consiste em uma técnica utilizada para "avaliação e interpretação de todas as pesquisas existentes disponíveis para uma questão de pesquisa, área temática ou fenômeno de interesse e possui como objetivo apresentar uma avaliação justa sobre o tópico de pesquisa utilizando uma metodologia confiável, rigorosa e passível de reprodução".

Dado que uma das razões mais comuns para a execução de uma revisão sistemática é a necessidade de identificação de lacunas em uma presente pesquisa, que ajude a nomear novas áreas de estudo ou investigação e o objetivo da presente pesquisa é identificar tais gaps quando o assunto é impacto de mudanças de arquitetura de software na manutenibilidade do sistema, a metodologia adotada para a execução do presente trabalho consiste em uma revisão sistemática da literatura existente sobre os temas envolvidos.

6.1 Classificação da pesquisa

Uma pesquisa pode ser classificada de diversas formas e dentre estas formas, dentre tais formas existe a classificação quanto à abordagem da pesquisa, que pode ser qualitativa, quantitativa ou ambas (mista). Segundo Günther (2006), a pesquisa qualitativa é baseada na compreensão de conhecimentos e pesquisas já publicadas, tendo como objeto de estudo a "descoberta e construção de teorias" além de ser uma abordagem de pesquisa baseada em textos, ou seja, a coleta de dados para a pesquisa produz textos que são interpretados de acordo com o que foi escrito pelo autor da pesquisa. Partindo desta definição, pode-se afirmar que a presente pesquisa pode ser classificada como qualitativa, uma vez que os dados coletados não serão dados estatísticos ou tampouco será executada uma análise de dados de tal tipo.

6.2 Identificação das fontes

6.2.1 Critérios para seleção de fontes

Os critérios adotados para seleção de fontes foram:

- Bibliotecas digitais.
- Artigos publicados em periódicos nacionais e internacionais.
- Artigos publicados em congressos, conferências e seminários nacionais e internacionais.

O idioma em que o artigo foi publicado é também um critério de seleção da fonte. Foram utilizados os idiomas Português e Inglês. O inglês foi utilizado pela quantidade considerável de artigos publicados neste idioma em congressos, conferências, seminários e periódicos internacionais e o português para consulta de trabalhos publicados no Brasil.

6.2.2 Método de busca

O método de busca utilizado foi determinado pelo mecanismo de busca fornecido pela fonte de pesquisa utilizando-se filtros de data e tipos de fonte (livros, artigos ou periódicos, por exemplo).

6.2.3 String de Busca

As strings de busca foram utilizadas individualmente afim de encontrar pesquisas que tratam os assuntos relacionados. As strings de busca utilizadas foram:

- Processo de desenvolvimento de software = Software Development Process
- Metodologia ágil de desenvolvimento de software = Agile methodology of software development
- Manifesto Ágil = Agile Manifest
- Arquitetura de software = Software Architecture
- Mudanças na arquitetura de software = Modifications in software architecture
- Manutenção <or> manutenibilidade de software = Software Maintenance <or> maintenability
- Arquitetura <and> manutenibilidade de software = Software Maintenance <and> Architecture

6.2.4 Lista de Fontes de Pesquisa

Para busca de dados para esta pesquisa foi utilizado como ferramenta de busca o Google Acadêmico (https://scholar.google.com.br/) para que pudesse ser abrangido um número maior de fontes. O Google Acadêmico redireciona cada um dos trabalhos publicados para a sua respectiva “fonte raíz”. Deste modo, as bases de dados principais redirecionadas foram:

- Safari (https://www.safaribooksonline.com/)
- IEEE Explore Digital Library (http:// http://ieeexplore.ieee.org/)

- Object Mentor (http://www.objectmentor.com/)
- Scielo (http://www.scielo.br/)
- Portal de periódicos CAPES (http://www.periodicos.capes.gov.br/)

6.2.5 Seleção de Estudos

6.2.5.1 Critérios de inclusão de estudos

Os artigos foram incluídos na pesquisa em relação aos sequintes critérios:

- Artigos e livros de natureza qualitativa que tratam de definições a respeito das principais fases do processo de desenvolvimento de software.
- Artigos e livros de natureza qualitativa que tratam de definições a respeito dos valores e princípios da metodologia ágil.
- Artigos e livros de natureza qualitativa de tratam de definições e princípios de arquitetura de software.
- Artigos e livros de natureza qualitativa que tratam de conceitos relacionados a manutenção de software.
- Artigos e livros devem estar disponíveis na web.
- Artigos e livros devem apresentar textos completos de estudos em formato eletrônico.
- Artigos e livros na área de computação e desenvolvimento de software.

A seleção e/ou exclusão dos artigos e/ou livros (ou parte deste) encontrados será baseada na leitura do Resumo, Introdução e das Considerações Finais, que deverão conter aspectos relacionados às variáveis de estudo desta pesquisa dentro do que é tratado pela área de Engenharia de Software. Apenas os artigos selecionados e incluídos deverão ser armazenados.