INSTITUTO FEDERAL
Triângulo Mineiro
Campus
Uberlândia Centro

1º Workshop Controle de Qualidade em Análise e Desenvolvimento de Sistemas

14 de Dezembro de 2019

PADRÕES DE PROJETOS APLICADOS NA EVOLUÇÃO DE SOFTWARE PARA GARANTIR ALTA *PERFORMANCE* DE MANUTENÇÃO E EVOLUÇÃO.

Arthur A. Carvalho[1], Edson A. Júnior[2]
[1]Pós-Graduando, arthurangelo.ptc@gmail.com , Instituto Federal do Triângulo Mineiro (IFTM) – Campus Uberlândia Centro, Rua Blanche Galassi, 150 – Bairro Morada da Colina – Uberlândia/MG
[2]Mestre, angoti@iftm.edu.br, Instituto Federal do Triângulo Mineiro (IFTM) – Campus Uberlândia Centro, Rua Blanche Galassi, 150 – Bairro Morada da Colina – Uberlândia/MG

Resumo: Softwares construídos não são estáticos e após a sua conclusão necessitam de manutenção, sendo este ponto constituído por atividades corretivas ou evolutivas. No aspecto evolutivo são adicionadas novas funcionalidades e características que elevam o tamanho do software, bem como sua complexidade. Faz-se necessário o uso de boas práticas para que um código seja gerado de modo a permitir que o crescimento seja saudável e mais simples possível. Como ferramentas que auxiliam em qualquer atividade de desenvolvimento, foram elaborados padrões de projeto, sendo que, cada um é focado na resolução de um problema especifico. Esses padrões são baseados nos princípios Solid. Eles não estão acoplados a nenhuma linguagem de desenvolvimento, mas permitem que o software possa evoluir de maneira estável. Este trabalho tem por objetivo analisar e demonstrar por meio da refatoração de um sistema, como a aplicação correta destas ferramentas permite o desenvolvimento estável e colaborativo de um software.

Palavras-chave: Solid; Padrões de Projeto; *Template Method*; *Factory Method*.

Introdução

Softwares construídos por equipes de desenvolvedores possuem a característica de serem escaláveis e propensos a aumentos de funcionalidades após sua conclusão. Esse aumento promove uma demanda maior de manutenções e de melhorias no código que, acabam gerando um custo maior do que a etapa de desenvolvimento inicial do software (MEYER, 1988).

Um código mal escrito frequentemente gera duplicidade de funcionalidades, por não seguir um padrão de desenvolvimento que permita simplicidade e modularidade do sistema. Cada classe ou método deve possuir uma única responsabilidade, de modo que cada manutenção ou melhoria seja pontual e específica para que as partes alteradas não interfiram diretamente nas outras partes do sistema. Quando há duplicidade, a manutenção se torna um problema. A consequência disso são alterações em todas as partes do código que foram copiadas, resultando em perda de tempo

(MAGALHÃES; TIOSSO, 2019).

Atualmente, sabe-se que 70% do custo de um software está na manutenção do mesmo (MEYER, 1988). Por este motivo, torna-se necessária uma padronização no processo de desenvolvimento do código (STURM; SILVA, 2014). A padronização melhora a legibilidade do código, facilitando o entendimento pelos mantenedores do sistema e, consequentemente, facilitando a sua manutenção. Isso é extremamente necessário para que o sistema continue funcionando de forma eficiente (LINO, 2011).

Atualmente, existem diversos padrões e metodologias que devem ser adotadas para o que o software produzido seja de fácil manutenção e possibilite a inclusão de novas funcionalidades. Para isto, é extremamente importante adotar as boas práticas de programação (Solid). O Solid é composto por cinco princípios, que propiciam o desenvolvimento de um código limpo, bem escrito e legível (ARAGÃO, 2017).

A adoção das boas práticas de programação (Solid) melhora consideravelmente, dentre outras características, a sua manutenibilidade. Esta, por sua vez, permite uma evolução estável do software, por facilitar modificações futuras no sistema e prover a padronização do mesmo (LINO, 2011).

Assim baseado nestes princípios e também na análise e construção de soluções para problemas recorrentes, Gamma et al., (2000) descreve uma coleção de padrões arquiteturais de software (*Design Patterns*) elaborados para resolução especifica de problemas. Cada item desta coleção possui quatro elementos essenciais:

- Nome que o define;
- Problema específico ao qual se relaciona;
- Descrição abstrata de arranjo de elementos (classes e objetos) para resolução de problema;
- Consequências do uso do padrão.

O objetivo deste trabalho é demonstrar como a adoção de padrões de projetos de software permite a manutenibilidade e a evolução sustentável de um software, garantindo assim, a inclusão de novos atributos, sem que estes impactem negativamente nas funcionalidades já existentes, além de garantir baixa duplicidade de código.

Referencial Teórico

Princípios Solid

Solid é um acrônimo que representa cinco princípios da programação orientada a objetos e, podem ser aplicados a qualquer linguagem de programação. Esses princípios melhoram a manutenibilidade dos softwares e os tornam mais evolutivos (ARAGÃO, 2017).

Cada princípio possui um único e especifico significado:

- SRP – *Single responsibility principle* (Principio da responsabilidade única): Cada classe, método, módulo deve possuir uma única responsabilidade, se tornando assim especialista em uma única funcionalidade;
- OCP – *Open/Closed principle* (Principio do Aberto/Fechado): Cada classe deve estar estável para estender seus comportamentos, e fechada para alteração de seu comportamento padrão de maneira a evitar replicação de alteração em vários pontos do sistema, atingindo vários módulos e classes;
- LSP – *Liskov substituition principle* (Principio da Substituição de Liskov): Princípio definido por Barbara Liskov em 1988, onde é estabelecido que classes derivadas podem ser substituídas por suas classes bases e classes bases podem ser substituídas por qualquer classe derivada sem que o comportamento ao qual estão inseridas seja alterado;
- ISP - *Interface segregation principle* (Principio da Segregação de Interfaces): Interfaces como são análogas a contratos devem ser específicas, com uma única responsabilidade, onde seu comportamento deve ser reutilizado por outras classes, com um único propósito. Interfaces não devem possuir mais de um propósito, caso o possuam, deve ser quebrada em outras interfaces, para que os clientes que estendem seu comportamento, não utilizem comportamentos desnecessários;
- DIP – *Dependency inversion principle* (Princípio da inversão de dependência): Classes não devem estar acopladas aos conceitos concretos, mas às abstrações (interfaces ou classes genéricas), pois estas não mudam ou possuem pequenas chances de modificação, conceito este que provê maior estabilidade entre módulos.

Padrões de projetos e manutenibilidade

Os padrões de projeto são técnicas pensadas inicialmente para resolução de problemas recorrentes baseados em conceitos de orientação a objetos. São descrições de objetos de classes comunicantes que precisam ser alteradas para resolver um problema geral em uma situação particular (GAMMA et al., 2000).

Os padrões de projeto estão agrupados em três grupos específicos, como mostrado na tabela 1:

Tabela 1: Agrupamento padrões de projeto.

Padrões de criação	**Padrões estruturais**	**Padrões comportamentais**
Abstract Factory	*Adapter*	*Chain of Responsability*
Builder	*Bridge*	*Command*
Factory Method	*Composite*	*Interpreter*
Prototype	*Decorator*	*Iterator*
Singleton	*Façade*	*Mediator*
	Flyweight	*Memento*
	Proxy	*Observer*
		State
		Strategy
		Template Method
		Visitor

Os padrões de projetos melhoram consideravelmente, dentre outras características, a manutenibilidade do software. Esta, por sua vez, pode ser definida como a capacidade de um software de ser modificado. As modificações podem incluir correções, melhorias ou acréscimo de funcionalidades (LINO, 2011). A manutenibilidade é dividida em cinco menores características, segundo a norma ISO/IEC 9126-1:2001 (2001):

- Analisabilidade: Maneira que o sistema permite a análise de melhorias e correções;
- Modificabilidade: Capacidade de absorver mudanças específicas;
- Estabilidade: Evitar comportamentos inesperados após alterações realizadas;
- Testabilidade: Capacidade de ser testado após novas melhorias, permitindo novos testes e uso de testes já realizados;
- Conformidade: Software de acordo com convecções de manutenibilidade.

Materiais e Métodos

Como objeto de discussão deste trabalho, serão abordados alguns padrões de projeto na construção de um software de processamento de arquivos, utilizando a linguagem JAVA, demonstrando o sistema em dois momentos distintos, onde no primeiro momento é mostrada a construção da função básica do sistema sem considerar evoluções e novas funcionalidades e no segundo momento, apresentado a reconstrução do sistema de maneira a prover facilidades de evolução.

O sistema motivador é um pequeno exemplo de sistema de processamento de arquivos que lê o conteúdo de arquivos de extensão TXT e faz a inserção das informações deste arquivo em um banco de dados. Inicialmente o sistema deverá processar (ler e salvar) as informações de um único arquivo (Figura 1).

```
public static void main(String args[]) throws Exception {
    List<Cliente> listClientes = new ArrayList<>();
    List<String> lista = new ArrayList<>();

    File file = new File( pathname: System.getProperty("user.dir") + "/Cliente.txt");
    FileReader fr = new FileReader(file);
    BufferedReader br = new BufferedReader(fr);

    while (br.ready()) {
        lista.add(br.readLine());
    }

    for (String linha : lista) {
        Cliente cliente = new Cliente();
        cliente.setTipo(linha.charAt(1));
        cliente.setCpf(linha.substring(2,13));
        cliente.setNome(linha.substring(13,43).trim());
        cliente.setEndereco(linha.substring(43,73).trim());
        cliente.setBairro(linha.substring(73,103).trim());
        cliente.setCidade(linha.substring(103,133).trim());
        cliente.setEstado(linha.substring(133,135));
        String dataCadastro = linha.substring(135,143);
        String horaCadastro = linha.substring(143,149);
        SimpleDateFormat sdf = new SimpleDateFormat( pattern: "ddMMyyyy hhmmss");
        cliente.setDataHoraCadastro(sdf.parse( source: dataCadastro+" "+horaCadastro));

        listClientes.add(cliente);
    }
    DAO dao = new DAO();
    dao.saveAll(listClientes);
```

Figura 1. Código inicial

Porém, como dito anteriormente, o sistema passará por evoluções e manutenções. Por isso, deve-se analisar e prever alterações que possam surgir na evolução do sistema. Um exemplo de melhoria que pode surgir: O sistema nunca irá processar outro tipo de arquivo? E mais, estes arquivos nunca serão processados em momentos distintos?

Para adicionar a nova funcionalidade pode ser implementado um novo método de maneira simples, conforme demonstrado na Figura 2.

```
public class Importacao {
    public static void main(String args[]) throws Exception {
        List<String> arquivos = Arrays.asList("Cliente.txt", "Transacao.txt");
        for (String arquivo : arquivos) {
            if ("Cliente.txt".equals(arquivo)) {
                List<String> lista = new ArrayList<>();
                List<Cliente> listClientes = new ArrayList<>();
                File file = new File( pathname: System.getProperty("user.dir") + "/" + arquivo);
                FileReader fr = new FileReader(file);
                BufferedReader br = new BufferedReader(fr);

                while (br.ready()) {...}
                for (String linha : lista) {...}
                DAO dao = new DAO();
                dao.saveAll(listClientes);
            }
            if ("Transacao.txt".equals(arquivo)) {
                List<String> lista = new ArrayList<>();
                List<Transacao> listTransacao = new ArrayList<>();
                File file = new File( pathname: System.getProperty("user.dir") + "/" + arquivo);
                FileReader fr = new FileReader(file);
                BufferedReader br = new BufferedReader(fr);

                while (br.ready()) {
                    lista.add(br.readLine());
                }

                for (String linha : lista) {
                    Transacao transacao = new Transacao();
                    transacao.setCodigoEstabelecimento(Integer.parseInt(linha.substring(42, 48)));
                    String data = linha.substring(35, 40);
                    String hora = linha.substring(41, 46);
                    transacao.setDataTransacao(new SimpleDateFormat( pattern: "ddMMyyyy hhmmss").parse( source: data + " " + hora));
                    transacao.setDetalhe(linha.charAt(0));
                    transacao.setNumeroConta(new Integer(linha.substring(1, 7)));
                    transacao.setNumeroPlastico(new Integer(linha.substring(8, 15)));
                    transacao.setValorTransacao(new Double(linha.substring(16, 27)));
                }
                DAO dao = new DAO();
                dao.saveAllTransacoes(listTransacao);
            }
        }
    }
}
```

Figura 2. Adição de novo processamento de arquivo.

Este desenvolvimento, de fato, evoluiu as funcionalidades e processos do sistema de importação, contudo, esta solução não está de acordo com os princípios Solid. Neste caso, uma classe possui mais de uma responsabilidade, está acoplada com classes fracas e a classe não está fechada para modificações, impedindo assim, o seu reuso e sua manutenibilidade, por não garantir sua testabilidade e analisabilidade, por exemplo.

Neste caso, alguns aspectos foram feridos e devem ser levados em consideração: Deve-se garantir que o sistema possa evoluir com a adição de um novo arquivo sem que impacte o processamento de arquivos existentes; Permitir uma maneira simples de inclusão e manutenção no processamento de um arquivo específico; É necessário adicionar um novo processamento de arquivo sem o uso de várias estruturas de decisão *if/else*, caso seja processado arquivos em momentos distintos e, por último, deve-se garantir que cada processamento possua suas características especificas sem alterar as demais estruturas de processamento.

Para se obter uma estabilidade no software, deve-se antecipar os novos requisitos e as futuras mudanças nos requisitos já existentes. Os padrões de projeto evitam problemas nestes aspectos. Para isto deve-se utilizar alguns padrões de projeto (GAMMA et.al., 2000). Neste estudo, os padrões de projeto utilizados foram o *Template Method* e o *Factory Method*, pois são respectivamente padrão comportamental e padrão de criação.

Resultados e Discussão

O código de processamento de arquivos foi refatorado por meio de delegação de responsabilidades de funcionalidades para as classes, como por exemplo, somente uma classe é responsável pela leitura de arquivo, e cada processamento é orquestrado por classe especifica de modelo de arquivo.

Os dois padrões utilizados permitiram que o comportamento de processamento de arquivos seja estendido para qualquer nova classe. Caso esta classe possua um comportamento específico ou necessite de alterações no comportamento antes estabelecido, a modificação poderá ser feita de modo que não afete diretamente outras classes. Isso permitiu a eliminação de estruturas de decisão a respeito de qual arquivo processar, garantiu desacoplamento entre as estruturas do sistema, eliminou duplicidade de código e gerou padronização de código de desenvolvimento.

Template Method

Para Gamma et al., (2000) este padrão de projeto "Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses, permitindo que as subclasses redefinam certos passos sem mudar sua estrutura".

O modelo de processamento utilizado, contém três passos base para o processamento, que são respectivamente ler um arquivo, processar esta informação de modo que este processamento se torne a representação de alguma entidade específica e, por fim, salvar esta informação em um repositório.

Cada um destes passos poderá mudar, onde a leitura, por exemplo, poderá ser feita a partir de um arquivo de extensão TXT, XLS, a partir de um webservice ou até mesmo de um mainframe.

Como ambos fornecem a mesma funcionalidade, de buscar a informação em um local, foi criada uma interface para garantir o contrato de busca de informação. Em sua implementação default também foi adicionada uma interface de leitura e, assim a interface de leitura pode ser trocada a qualquer momento na implementação default ou, caso seja necessário adicionar nova forma de busca de dados, será somente adicionada uma nova interface de leitura e uma nova interface que busque os dados (Figuras 3 e 4).

```
public interface ObterDados {
    List<String> obterDados(String path) throws IOException;
}
```

Figura 3. Interface obter dados.

```
public class LeituraTxt implements Leitura{
    private List<String> linhas;
    @Override
    public  List<String> ler(String path) throws IOException {
        File file = new File(path);
        FileReader fr = new FileReader(file);
        BufferedReader br = new BufferedReader(fr);
        linhas = new ArrayList<>();
        while (br.ready()) {
            linhas.add(br.readLine());
        }
        fr.close();
        br.close();
        return linhas;
    }
}
```

Figura 4. Classe de leitura de arquivo TXT.

Com a interface “ObterDados”, especialista em obter dados, é possível que em qualquer ponto do sistema que seja necessário o uso de outra maneira de obtenção de dados, seja fornecida outra implementação e, assim os conceitos DIP e ISP do catálogo Solid foram assegurados.

A manipulação de cada arquivo é um passo que deve ser implementado por cada tipo de processamento, onde um método abstrato é utilizado para que este passo seja executado.

O último passo, salvar ou enviar informações, possui a mesma característica de leitura, porém, não é possível o uso de uma interface default, pois cada processamento necessita salvar seus valores em um local específico. Contudo, ainda é possível fazer uso de interface para garantir um contrato para enviar os dados (Figura 5) e gerar suas implementações (Figuras 6 e 7).

```
public interface EnviarDados {
    void enviarDados(List<Dados> dados);
}
```

Figura 5. Interface para envio de informações.

```
public class EnviarDadosCliente implements EnviarDados {
    @Override
    public void enviarDados(List<Dados> dados) {
        System.out.println("executando operacao de salvar cliente");
    }
}
```

Figura 6. Classe de envio de informações de arquivo cliente.

```
public class EnviarDadosTransacao implements EnviarDados {
    @Override
    public void enviarDados(List<Dados> dados) {
        System.out.println("executando operacao de salvar transacao");
    }
}
```

Figura 7. Classe de envio de informações de arquivo transação.

Com a abstração destes passos, de grande importância, realizada pela classe “Template” (Figura 8), é possível estender o comportamento padrão de processamento ou reutilizá-lo caso necessário, com melhor *performance* de agilidade e simplicidade de manutenção.

```java
public abstract class Template {

    protected ObterDados obterDados;
    protected EnviarDados enviarDados;

    protected List<String> obterDados(String fonte) throws IOException {
        return obterDados.obterDados(fonte);
    }

    protected void enviarDados(List<Dados> dados) {
        enviarDados.enviarDados(dados);
    }

    public void run(String fonte) throws IOException {
        List<String> dados = obterDados(fonte);
        List<Dados> processados = processar(dados);
        enviarDados(processados);
    }
    protected abstract List<Dados> processar(List<String> dados);

}
```

Figura 8. Classe Template.

Factory Method

De acordo com Gamma et.al., (2000) o *Factory Method* deve "Definir uma interface para criar um objeto, mas deixarem as subclasses decidirem qual classe instanciar".

Outro ponto crucial no sistema é a criação dos modelos de processamento que, poderiam ser feitos via estruturas de decisão. Contudo esta abordagem se demonstrou de baixa qualidade.

Este arquétipo visa centralizar a criação de objetos, de maneira a retirar das classes que fazem uso do processamento, todo conhecimento especifico sobre como fabricar uma instância e, assim fornecer o objeto devidamente criado. Deste modo, o uso de várias estruturas de decisão foi excluído, respeitando assim o principio OCP.

Cada instância será fornecida por uma classe do tipo Enum, devido ao fato de que cada modelo de processamento é único por arquivo e, devido à necessidade que pode futuramente existir de se utilizar uma nova processadora. A fábrica de templates é demonstrada na Figura 9.

```
public enum FactoryTemplates {
    CLIENTE( arquivo: "Cliente.txt"
            , fonte: System.getProperty("user.dir") + "/resources/Cliente.txt"
            ,new ClienteTemplate(new ObterDadosDefault(new LeituraTxt())
            ,new EnviarDadosCliente())),

    TRANSACAO( arquivo: "Transacao.txt"
            , fonte: System.getProperty("user.dir") + "/resources/Transacao.txt"
            , new TransacaoTemplate(new ObterDadosDefault(new LeituraTxt())
            , new EnviarDadosTransacao()));

    private String arquivo;
    private String fonte;
    private final  Template template;

    FactoryTemplates(String arquivo,String fonte, Template template) {
        this.arquivo = arquivo;
        this.fonte = fonte;
        this.template = template;
    }

    public static FactoryTemplates getTemplateByArquivo(String arquivo) {
        for (FactoryTemplates factory : FactoryTemplates.values()) {
            if (factory.arquivo.contains(arquivo)) {
                return factory;
            }
        }
        return null;
    }

    public Template getTemplate() { return this.template; }
    public String getFonte() { return this.fonte; }
}
```

Figura 9. Classe responsável pelo *Factory Method*.

Para realizar o processamento do arquivo, é necessário apenas o nome do mesmo, conforme mostra a Figura 10.

```
for(String arquivo : arquivosProcessar){
    FactoryTemplates factory = FactoryTemplates.getTemplateByArquivo(arquivo);

    Template template = factory.getTemplate();
    template.run(factory.getFonte());

}
```

Figura 10. Algoritmo utilizado para realizar processamentos.

Conclusão

O presente estudo mostrou como os requisitos de um sistema podem mudar e aumentar após sua conclusão, dificultando assim sua manutenção. Foi construído um sistema motivador que permitiu demonstrar que a manutenibilidade de um software deve ser pensada ainda no seu desenvolvimento, de modo que as mudanças e funcionalidades não sejam duplicadas e sim, estendidas. Isso ocorre através de conceitos de desenvolvimento de software que podem ser utilizados em qualquer linguagem de programação. Também foi demonstrado como estes princípios garantem um padrão de desenvolvimento, através de um modelo default para leitura/processamento, permitido pela herança.

Os padrões de projeto *Factory Method* e *Template Method* facilitaram a inclusão de novos processamentos e de alterações em processamentos existentes, sem impactá-los negativamente.

O código se tornou escalável e de fácil compreensão com o uso dos conceitos Solid como, por exemplo, o conceito OCP, em que a classe deve possuir comportamento expandido. O comportamento do processamento é facilmente expandido através do *Template Method*, que provê um modelo base para leitura, processamento e inclusão de dados processados. Deste modo, cada inclusão de um novo processamento terá um comportamento baseado no *Template Defult*. Assim, as classes já presentes desconhecerão a nova funcionalidade. O conceito ISP também foi garantido, pois foram criadas interfaces magras para cada tipo de processamento e, assim caso um modelo de processamento de arquivo futuramente necessite de uma funcionalidade específica, novamente o escopo de outros processamentos não irão conhecer a nova funcionalidade e, deste modo, não serão impactados.

Com o uso do *Factory Method* foi centralizada a criação de objetos de processamento, onde na necessidade de um novo processamento basta ser criada uma nova opção de processamento e, toda a lógica de processamento não será alterada.

Através do uso do conceito DIP, foram adicionadas melhorias que permitem a troca do modo que uma funcionalidade é realizada. Por exemplo, caso seja necessária a leitura de um arquivo de extensão XLS, o comportamento da interface “ObterDados” deverá ser estendido e, assim qualquer processamento poderá utilizar esta nova característica, visto que o sistema faz uso da interface e não da implementação.

As melhorias obtidas no uso das técnicas demonstram que apenas o uso de boas práticas faz com que qualquer sistema evolua de maneira saudável, independente da linguagem orientada a objetos utilizada.

Referências

ARAGÃO, Thiago. Solid: Princípios da Programação Orientada a Objetos. **Medium**, 13 de Maio de 2017. Disponivel em: < https://medium.com/thiago-aragao/solid-princ%C3%ADpios-da-programa%C3%A7%C3%A3o-orientada-a-objetos-ba7e31d8fb25>. Acesso em: 13 de Nov. de 2019.

GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos. 1. ed. Porto Alegre: Artmed Editora S.A., 2000. 364 p. v. 1.

ISO/IEC 9126-1:2001. Software engineering: Product quality. Part1: Quality model, 2001.

LINO, Carlos Eduardo. **Reestruturação de software com adoção de padrões de projeto para a melhoria da manutenibilidade**. Orientador: Antônio Maria Pereira de Resende. 2011. 66 p. Monografia (Graduação em Sistemas de Informação) - Universidade Federal de Lavras, Lavras, MG, 2011.

MAGALHÃES, Phelipe Alex; TIOSSO, Fernando. Código Limpo: padrões e técnicas no desenvolvimento de software. **Revista Interface Tecnológica**, São Paulo, v. 16, n. 1, p. 197-207, 2019.

MEYER, Bertrand. **Object-Oriented Software Construction**. 2. ed. Santa Barbara, CA: Interactive Software Engineering Inc. (ISE), 1988. 1254 p. v. 2.

STURM, Junior; SILVA, Madalena Pereira da. Aplicação de padrões de projeto no desenvolvimento de software para a melhoria de qualidade e manutenibilidade. **Revista de Exatas e Tecnológicas (RETEC)**, Rondonópolis, MT, v. 5, n. 1, ed. 5, p. 37-46, 2014.

TERRA, Ricardo; VALENTE Marco Tulio. Definição de Padrões Arquiteturais e seu Impacto em Atividades de Manutenção de Software, Belo Horizonte, p. 1-8, 2010. Disponivel em: < http://algol.dcc.ufla.br/~terra/publications_files/2010_wmswm.pdf >. Acesso em: 13 de Nov. de 2019.