ESTUDO COMPARATIVO DE TECNOLOGIAS DA PLATAFORMA JAVA EE PARA DESENVOLVIMENTO DE SISTEMAS WEB[1]

Alvondi Rodrigues de Lima Junior[2]

Jorge Luis Boeira Bavaresco[3]

RESUMO

Este artigo tem como objetivo principal realizar a comparação de sistemas Java para *web*, aplicando métricas de software em dois estudos de caso desenvolvidos, buscando evidenciar por meio da análise dos resultados a vantagem ou desvantagem da utilização de tecnologias da plataforma Java EE como *Enterprise Java Beans* (EJB). Algumas métricas da engenharia de software foram utilizadas para analisar o nível de complexidade do código, como métodos ponderados por classe (WMC), resposta para uma classe (RFC) e medição do tamanho do código realizando a contagem de linhas (LOC). Com o resultado desta pesquisa foi possível identificar que a utilização do EJB torna o sistema menos complexo, facilita o trabalho do desenvolvedor e contém menor quantidade de linhas de código.

Palavras-chave: Métricas. Java. Engenharia de Software. EJB. WMC. RFC.

1 INTRODUÇÃO

O acesso à internet tem aumentado significativamente entre a população brasileira nos últimos anos (IBGE, 2015). O acesso difundido à rede solidifica o mercado online. Neste contexto, a criação de aplicações web pode gerar bons resultados e até grandes negócios e empresas. O cuidado e a preocupação em desenvolver aplicações com qualidade se tornaram obrigatórios, considerando a exigência dos clientes, e com a segurança das informações, para proteger os sistemas de ataques e roubo de dados.

Tendo isso em vista, se faz necessário escolher uma linguagem de programação difundida e confiável. O Java é a linguagem mais amplamente utilizada

---

[1] Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Câmpus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet, na cidade de Passo Fundo, em 2016.

[2] Graduando em Tecnologia de Sistemas para Internet no Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense de Passo Fundo (IFSUL). E-mail: alvondi.junior@gmail.com

[3] Orientador, professor do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense de Passo Fundo (IFSUL). E-mail: jorge.bavaresco@passofundo.ifsul.edu.br.

segundo Deitel (2010) e conforme pesquisa publicada em Janeiro de 2016 (REDMONK, 2016), Java é a segunda linguagem mais popular.

Este artigo visa realizar a comparação de tecnologias para o desenvolvimento de sistemas web em Java. Serão desenvolvidas duas aplicações Java, observando que uma delas será desenvolvida utilizando recursos da tecnologia EJB, e a outra sem a implementação do EJB, buscando através da aplicação de métricas de software, levantar possíveis vantagens e desvantagens do uso do EJB.

Complementando o objetivo principal do trabalho, que trata da comparação das aplicações via medição do software, adiciona-se também o desejo de difundir o conhecimento sobre métricas de software para sistemas orientados a objetos. As métricas de software são características de um sistema que podem ser mensuradas, com o objetivo de coletar informações que auxiliam na conclusão sobre a qualidade de sistemas.

2 REFERENCIAL TEÓRICO

Nesta seção serão apresentadas as principais tecnologias utilizadas para o desenvolvimento dos programas e as métricas de software que serão aplicadas.

2.1 JAVA

Conforme Deitel (2010), Java é uma linguagem de programação criada por James Gosling. Dentre outras aplicações, o Java é utilizado para desenvolver aplicativos para dispositivos móveis (JavaME) e aplicativos corporativos de grande porte tanto para computadores de mesa (JavaSE), quanto para web (JavaEE). Java implementa características como herança, encapsulamento e várias outras habilidades particulares que a orientação a objetos proporciona, como também as características já conhecidas no mundo da programação, como por exemplo instruções de controle e operadores lógicos (DEITEL, 2010).

2.2 JAVA EE

Segundo Gonçalves (2011) “O Java EE é um conjunto de especificações destinadas a aplicações empresariais”, onde possui inúmeras APIs para diversas aplicações, tornando o trabalho de desenvolvimento focado mais precisamente na regra de negócio do que desenvolver soluções de baixo nível, deixando para o Java fazer a gerência, permitindo tratar questões mais importantes como o negócio propriamente dito. Como exemplo, podemos citar a utilização de EJB e JTA, onde a gerencia do ciclo de vida das instancias das classes e aspectos transacionais ficam a cargo do servidor.

Fazem parte também do Java EE serviços como *Transaction API* (JTA), *Java Persistence API* (JPA), *Java Naming and Directory Interface* (JNDI), Serviços web entre outros (GONÇALVES, 2011).

2.3 ENTERPRISE JAVABEANS

No Java EE, os EBJs são componentes do lado servidor, utilizados para implementar a camada funcional de uma aplicação. Localizado acima da camada de persistência, a qual realiza o mapeamento (ORM) e o encapsulamento dos objetos em uma base relacional. E abaixo da camada de apresentação, a qual é a interface com o usuário, que é implementada por tecnologias do lado cliente, como o *JavaServer Faces*. Além de separar as camadas de apresentação e persistência, como *business logic*, a camada funcional implementa diversos serviços como por exemplo o gerenciamento de segurança e transações. O modelo de programação utilizado pelos EJBs é muito poderoso, tornando os EJBs menos complexos e ao mesmo tempo, altamente reutilizáveis e escaláveis, integráveis com outras tecnologias, sistemas e modelos de dados (GONÇALVES, 2011).

O componente responsável por realizar o encapsulamento da lógica de negócio é o *Session Bean*. Os *Session Beans* são divididos em três tipos: *Stateful*, *Stateless* e *Singleton* (ORACLE, 2015).

O estado de um objeto consiste nos valores de suas variáveis de instância. Em um *session bean stateful*, as variáveis são mantidas até o final da sessão, independentemente de quantas interações ocorreram na página. Um

exemplo é um carrinho de compras dentro de um *e-commerce*, que mesmo com conteúdo dentro dele, você pode continuar navegando nas páginas, sem perder os produtos contidos no carrinho.

O *session bean stateless* não guarda o estado de conversação com o cliente, ao contrário do *stateful*. Quando o método chamado é concluído, o estado do cliente não é salvo.

O *session bean singleton* é instanciado apenas uma vez, no início da aplicação. É projetado para os casos em que uma única instância seja compartilhada e acessada pelos clientes.

2.4 JAVA PERSISTENCE API

A *Java Persistence API* (JPA) é a especificação padrão para persistência do Java EE. Trata-se do mecanismo que salva os dados da aplicação em um banco de dados. A maioria dos bancos de dados utilizados em sistemas de informação são relacionais (BAUER; KING, 2007).

Com a ampla utilização de tecnologias orientada a objetos, surge um problema chamado de disparidade do paradigma objeto/relacional. Uma das mais importantes estratégias para resolver este problema é o mapeamento objeto/relacional (ORM), que realiza a persistência dos objetos em tabelas de banco de dados relacionais (BAUER; KING, 2007).

Gonçalves (2011) cita que “o princípio do ORM envolve a delegação de acesso a bases de dados relacionais a ferramentas ou estruturas externas, que, por sua vez, fornecem uma visão orientada por objetos de dados relacionais, e vice versa.”, construindo uma correspondência bidirecional entre a base de dados e os objetos.

Alguns dos principais componentes da JPA são (GONÇALVES, 2011): O ORM, que realiza o mapeamento dos objetos para dados relacionais; a API gerenciadora de entidades chamada *EntityManager*, que realiza as operações de persistência (CRUD) na base de dados; a linguagem de consulta de persistência em Java (JPQL), análoga ao SQL, que é a linguagem padrão de consultas a banco de dados e os mecanismos de transações, fornecidos pela *Java Transaction API* (JTA).

O ORM permite a manipulação de entidades, sendo que o que realmente está sendo acessado é a base de dados. Permite que as classes, objetos e atributos sejam mapeados em bases de dados relacionais compostas de tabelas com linhas e colunas. Isso é possível através de metadados, que descrevem o mapeamento.

2.5 JAVA TRANSACTION API

O gerenciamento de transações assegura que as aplicações tenham dados consistentes, como por exemplo, controlando os acessos concorrentes aos dados, tanto da própria aplicação quanto de outra, sempre garantindo um nível de confiabilidade e robustez ao sistema. Além de garantir a persistência de dados em uma ou mais bases de dados, também envolve outras operações como, de envio de mensagens utilizando *Java Message Service* e chamadas de serviços web (GONÇALVES, 2011).

Existem dois tipos de transações: transações locais e transações distribuídas. Transações locais tratam de operações que são realizadas em um único recurso como, por exemplo, uma base de dados. Enquanto as transações distribuídas podem envolver mais de um recurso, que podem ser mais de uma base de dados ou algum outro recurso disponível na rede (GONÇALVES, 2011).

Conforme Gonçalves (2011), componentes como o gerenciador de transações e gerenciador de recursos tratam das transações através da API de Transações do Java (JTA), que foi especificada pela JSR 907[4].

Com o desenvolvimento de aplicações utilizando EJB, não é preciso se preocupar em como lidar com os gerenciadores de recursos ou de transações, pois a JTA faz a abstração, implementando os protocolos de baixo nível da transação. Cada método é automaticamente empacotado em uma transação. O modelo EJB foi projetado desde sua criação para gerenciar transações, sendo de sua natureza ser transacional (GONÇALVES, 2011).

[4] https://jcp.org/en/jsr/detail?id=907

2.6 JAVA SERVER FACES

JavaServer Faces é um framework baseado em componentes para o desenvolvimento da interface de usuário de uma aplicação Java web, possibilitando a programação em um nível mais alto do que trabalhar com HTML puro. Permite a reutilização de componentes como também utilizar componentes de terceiros (GEARY; HORSTMANN, 2012).

2.7 MÉTRICAS

O objetivo da medição de software é coletar informações, possibilitando chegar a conclusões sobre a qualidade e eficácia do sistema. Uma métrica de software é uma característica do sistema que pode ser medida, como a quantidade de linhas de uma classe. Existem dois tipos de métricas: métricas de controle que são geralmente ligadas aos processos de software; e métricas de previsão ou produto, que auxiliam na análise das características do sistema (SOMMERVILLE, 2011). Foram coletados e analisados os valores resultantes da aplicação das métricas de produto nos dois sistemas propostos.

As métricas de produto se dividem em duas classes: métricas dinâmicas e métricas estáticas. Métricas dinâmicas coletam os dados com o programa em execução. Um exemplo pode ser a medição do tempo de execução de determinados processos. As métricas dinâmicas auxiliam a avaliar a eficiência e a confiabilidade do sistema. Já as métricas estáticas, são coletadas realizando a medição de representações do sistema, como o programa, medindo o tamanho do código por exemplo. Por sua vez, as métricas estáticas ajudam a avaliar a complexidade, compreensibilidade e a manutenibilidade do sistema (SOMMERVILLE, 2011).

Segundo Pressman (2011) “A classe é a unidade fundamental de um sistema orientado a objeto”, logo, são os principais pontos a serem analisados nas medições de software. Sua relação com outras classes e seus métodos, sua hierarquia, comportamentos e tamanho, por exemplo.

Existem métricas específicas para medição de softwares orientados a objetos. As mais amplamente utilizadas foram propostas em 1994 por Chidamber e Kemerer, em um artigo científico publicado na IEEE (CHIDAMBER; KEMERER, 1994). O

conjunto de métricas conhecido por suíte *ck* é composto de seis métricas (SOMMERVILLE, 2011) (PRESSMAN, 2011).

2.7.1 Métodos ponderados por classe (WMC):

De acordo com Sommerville (2011) “É o número de métodos de uma classe, ponderados pela complexidade de cada método”. Conforme Pressman (2011) “O número de métodos e sua complexidade são indicadores razoáveis do trabalho necessário para implementar e testar uma classe”. A classe que possui muitos métodos, e métodos muito complexos, torna-se específica, dificultando o seu reuso, e logicamente, sua compreensão. Além de afetar suas chances de ser uma superclasse e de dificultar os testes. Quanto menor o número recebido por essa métrica, entendemos que a classe está menos complexa e mais propensa ao reuso (SOMMERVILLE, 2011).

No decorrer do estudo referencial deste artigo e devido a dificuldades encontradas que serão relatadas na seção de resultados, foi encontrada outra forma de mensurar a WMC, por meio da complexidade ciclomática. A complexidade ciclomática (MCCABE, 1976) mede a quantidade de saídas lógicas possíveis na execução de controles de fluxo como *if else, while, switch case* entre outros. Com o resultado da aplicação da complexidade ciclomática, obtemos a quantidade de saídas diferentes possíveis do código testado. Essa informação torna-se importante, pois demonstra quantos casos de teste devem ser feitos para abranger o método analisado, facilitando a execução dos testes. Um número alto para complexidade ciclomática aponta maior complexidade no código, manutenibilidade dificultada devido à complexidade, dificuldade em realizar testes e chance diminuída de reuso, pois o código se torna específico.

Neste trabalho foram utilizadas as duas formas de medição da WMC, pela contagem dos métodos utilizando a nomenclatura WMC1 e pelo cálculo da complexidade ciclomática utilizando WMC2.

2.7.2 Árvore de profundidade de herança (DIT):

Quanto maior o número resultante de DIT, maior é a profundidade da hierarquia da superclasse, fazendo com que as classes de níveis inferiores herdem muitos métodos, dificultando prever seu comportamento além de aumentar a complexidade. Em contrapartida, um número alto para DIT implica na maior possibilidade de reuso dos métodos.

2.7.3 Número de filhos (NOC):

Diferente da DIT, a métrica NOC mede a largura da hierarquia da superclasse em relação a suas filhas imediatas, ou seja, quantas subclasses ela possui. Um número alto para NOC indica maior reuso, mas também pode aumentar o esforço para validação e testes (SOMMERVILLE, 2011).

2.7.4 Acoplamento entre classes de objeto (CBO):

O CBO refere-se ao nível de acoplamento de uma classe com outra, ou seja, quando métodos de uma classe, utilizam métodos ou atributos de outra classe. Um valor alto para CBO indica grande dependência de classes, e dificulta a manutenibilidade do código, pois as alterações são mais propensas a afetar várias partes do programa devido ao alto nível de acoplamento (SOMMERVILLE, 2011).

2.7.5 Resposta para uma classe (RFC):

Segundo Sommerville (2011) é a quantidade de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto desta classe. Quanto maior o número, a classe se torna mais complexa e propensa a erros.

2.7.6 Falta de coesão em métodos (LCOM):

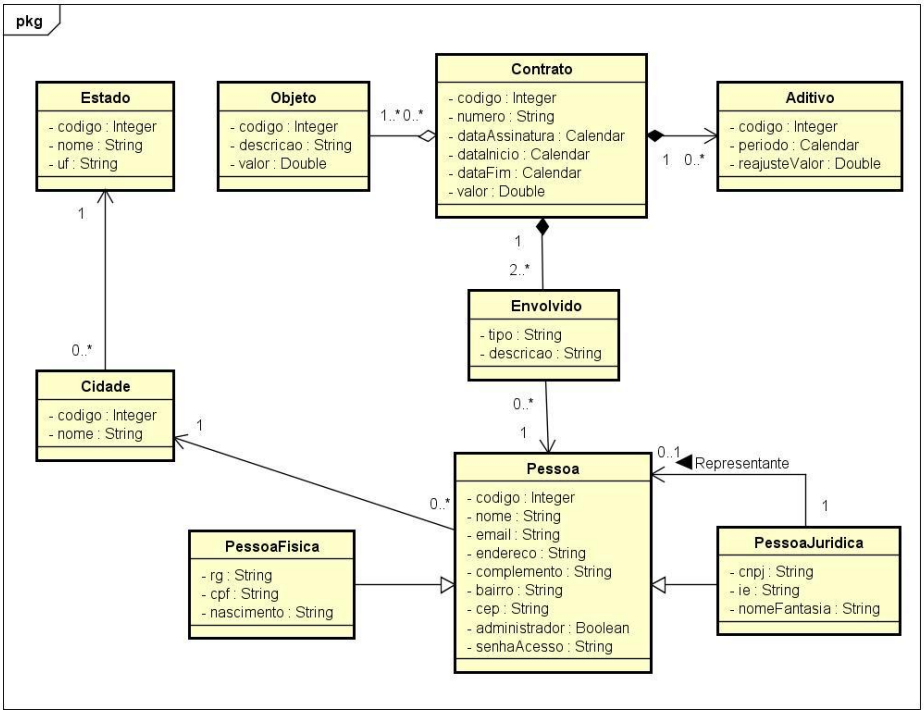
Esta métrica analisa a quantidade de métodos que acessam uma ou mais vezes o mesmo atributo, por exemplo, se uma classe possui três métodos, e dois

deles acessam o mesmo atributo, o valor de LCOM = 2 para esta classe. Se o número for alto, aumenta também a complexidade do código (PRESSMAN, 2011).

3 METODOLOGIA

Para realizar a aplicação das métricas foram desenvolvidas duas aplicações, baseadas no diagrama de classes demonstrado na Figura 1. Em uma delas será utilizado recursos da tecnologia EJB, e a outra não terá a utilização do EJB. Trata-se de um sistema para controle de contratos, registrando seus objetos como, por exemplo, a manutenção de computadores e links de internet, os envolvidos como contratado e contratante, além das demais classes de apoio, conforme o diagrama.

Figura 1 - Diagrama de classes



Fonte: Do Autor.

Entre as diferenças existentes entre as aplicações, uma das principais fica a cargo da utilização da JTA com *data-source*, por parte da aplicação com EJB. A gerência das transações sob responsabilidade do servidor torna o trabalho do

desenvolvedor facilitado, limpo e com menos linhas de código. De acordo com Gonçalves (2011), o desenvolvimento de EJB utilizando JTA abstrai a maior parte da complexidade, deixando que o contentor EJB implemente os protocolos de baixo nível da transação.

A aplicação sem EJB utiliza o tipo de transação *resource_local*, conforme arquivo *persistence.xml* demonstrado na Figura 2, tendo que declarar todas as informações de conexão com a base de dados.

Figura 2 - Arquivo persistence.xml com resource_local

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="NomeDaUnidadeDePersistencia" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>...</class>
    <class>...</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/nomedobanco"/>
      <property name="javax.persistence.jdbc.user" value="usuariodobanco"/>
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.password" value="senhadousuario"/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.connection.autocommit" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fonte: Do autor.

Em contrapartida, a aplicação que utiliza EJB, se beneficia da JTA, utilizando o tipo de conexão *data-source*. Pode-se observar na Figura 3 que o *data-source* faz referência para a conexão JNDI criada, que realiza a conexão com a base de dados.

Figura 3 - Arquivo persistence.xml com data-source

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="NomeDaUnidadeDePersistencia" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/nomedobanco</jta-data-source>
    <class>...</class>
    <class>...</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.transaction.jta.platform" value="org.hibernate.service.jta.platform.internal.SunOneJtaPlatform"/>
      <property name="hibernate.classloading.use_current_tccl_as_parent" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fonte: Do autor.

Notoriamente, identifica-se uma diferença nos arquivos DAO das aplicações. Na modelagem sem EJB conforme Figura 4, os métodos de persistência possuem linhas a mais do que com a utilização da JTA com EJB de acordo com a Figura 5. É necessária a programação da inicialização e finalização das transações, além de ter que declarar as *EntityManagers* em todos os métodos. Esse esforço não é necessário se for utilizado EJB, pois a gerência das transações é realizada automaticamente pelo servidor via JTA com a utilização da anotação *@PersistenceContext,* instanciando a *EntityManager* de uma forma mais simples em relação a outra aplicação, tornando a programação com EJB facilitada. Essa diferença pode ser observada nas Figuras 4 e 5, que demonstram o método *persist*.

Figura 4 - Método persist sem EJB

```
public void persist(T object) throws Exception {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("GenContracts_SemEJBPU");
    EntityManager em = emf.createEntityManager();
    try {
        if (em.getTransaction().isActive() == false) {
            em.getTransaction().begin();
        }
        em.persist(object);
        em.getTransaction().commit();
    } catch (Exception e) {
        if (em.getTransaction().isActive() == false) {
            em.getTransaction().begin();
        }
        em.getTransaction().rollback();
        throw new Exception("Erro na operação de persistência: " + e.getMessage());
    } finally {
        em.close();
        emf.close();
    }
}
```

Fonte: Do autor.

Figura 5 - Método persist com EJB

```
@PersistenceContext(unitName = "GenContractsPU")
private EntityManager em;

public void persist(T object) throws Exception {
    em.persist(object);
}
```

Fonte: Do autor.

Percebe-se também que na aplicação sem EJB, fica sob responsabilidade do desenvolvedor instanciar manualmente o DAO, conforme Figura 6:

Figura 6 – Como instanciar o DAO sem EJB

```
private CidadeDAO<Cidade> dao;
private Cidade objeto;
private EstadoDAO daoEstado;

public ControleCidade() {
    dao = new CidadeDAO();
    daoEstado = new EstadoDAO();
}
```

Fonte: Do autor.

Enquanto na outra aplicação, os DAOs são instanciados automaticamente pela anotação @EJB conforme Figura 7, liberando o desenvolvedor de tal tarefa.

Figura 7 - Como instanciar o DAO com EJB

```
@EJB
private CidadeDAO<Cidade> dao;
private Cidade objeto;
@EJB
private EstadoDAO daoEstado;

public ControleCidade() {
}
```

Fonte: Do autor.

Foram procuradas na engenharia de software, formas para a medição de sistemas orientados a objetos. Foram encontradas as métricas da suíte *ck* citadas na seção anterior deste artigo. O resultado a aplicação das métricas demonstra o nível de complexidade, de manutenibilidade do código, da dificuldade da realização de testes e também a propensão de reuso de métodos e classes.

Para realizar a análise das classes dos projetos desenvolvidos, foi utilizado o programa ckjm (SPINELLIS, 2015). Este programa realiza a verificação das classes, aplicando as métricas da suíte *ck*. A utilização do programa é demonstrada abaixo

na Figura 8. Após baixar e alocar a aplicação em uma pasta desejada, via linha de comando, é preciso executar o comando java -jar, em seguida apontar para o local do programa baixado, no caso o ckjm-1.5.jar. Em seguida, selecionar a pasta *build* do projeto, informando a ou as classes que irão ser analisadas.

Figura 8 - Execução do programa ckjm

```
java -jar /usr/local/lib/ckjm-1.5.jar build/classes/gr/spinellis/ckjm/*.class

gr.spinellis.ckjm.ClassMetricsContainer 3 1 0 3 18 0 2 2
gr.spinellis.ckjm.MethodVisitor 11 1 0 21 40 0 1 8
gr.spinellis.ckjm.CkjmOutputHandler 1 1 0 1 1 0 3 1
gr.spinellis.ckjm.ClassMetrics 24 1 0 0 33 196 6 23
gr.spinellis.ckjm.MetricsFilter 7 1 0 6 30 11 2 5
gr.spinellis.ckjm.ClassVisitor 13 1 0 14 71 34 2 9
gr.spinellis.ckjm.ClassMap 3 1 0 1 21 0 0 2
gr.spinellis.ckjm.PrintPlainResults 2 1 0 2 8 0 1 2
```

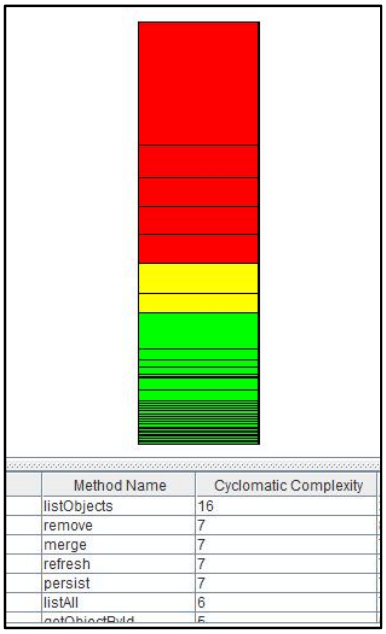
Fonte: Spinellis, 2015.

Após a execução, irão aparecer as classes selecionadas, seguidas com números resultantes das métricas aplicadas, conforme a seguir: WMC, DIT, NOC, CBO, RFC, LCOM, Ca, e NPM. As métricas Ca e NPM significam respectivamente, acoplamento aferente e número de métodos públicos para uma classe (SPINELLIS, 2015). Ca e NPM não são métricas da suíte *ck* e não serão abordadas e utilizadas neste artigo. Foram utilizadas as métricas DIT e LCOM para mensurar a complexidade do código, a métrica NOC para avaliar o nível de manutenibilidade do código, CBO para acoplamento e WMC e RFC para manutenibilidade e complexidade do código.

Além do ckjm, foram utilizados outros dois programas, o Cyvis e o CodeAnalyzer.

O Cyvis é um software livre baseado em Java para coleta, análise e visualização de métricas de software (CYVIS, 2016). Esta ferramenta foi utilizada para medir a complexidade ciclomática das aplicações. Após informar quais arquivos foram analisados, o Cyvis realiza a análise e expõe o resultado com visualizações tabular e em gráfico, do projeto, pacote e classe. Na Figura 9 é demonstrado o resultado da execução do Cyvis na classe GenericDAO da aplicação sem EJB.

Figura 9 - Demonstração do resultado gerado pelo Cyvis.

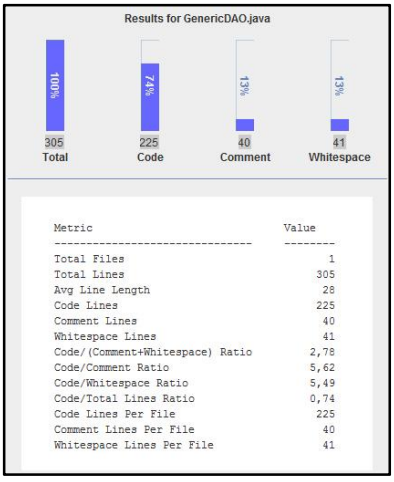


Method Name	Cyclomatic Complexity
listObjects	16
remove	7
merge	7
refresh	7
persist	7
listAll	6

Fonte: Do Autor.

O CodeAnalyzer também é software livre, escrito em Java, que realiza o cálculo de métricas relacionadas a quantidade de linhas de código (CODEANALYSER, 2016). Esta ferramenta será utilizada para medir a quantidade de linhas de código das aplicações. Após a verificação do projeto pelo software, o resultado é exibido conforme Figura 10.

Figura 10 - Demonstração do resultado gerado pelo CodeAnalyser.



Fonte: Do Autor.

Foi realizada a análise da classe GenericDAO para demonstrar a saída do programa CodeAnalyser.

4 RESULTADOS

Nesta seção serão apresentados os resultados obtidos no desenvolvimento deste artigo.

Nas primeiras aplicações das métricas, foi identificado que o valor para WMC estava igual para ambos os projetos. A métrica WMC como foi vista, mede a complexidade do código. A partir disso, foram realizadas buscas na documentação da ferramenta ckjm, com o objetivo de descobrir como a WMC é calculada. Foi identificado que esta métrica calcula apenas a contagem dos métodos contidos nas classes analisadas. Com esta informação, foi percebida a necessidade de buscar outra forma de medir a complexidade. E que por sua vez, foi encontrada a complexidade ciclomática (MCCABE, 1976) descrita na seção de MÉTRICAS. Neste artigo, é apresentado o resultado de WMC utilizando a contagem de métodos (WMC1) e também pela complexidade ciclomática (WMC2).

Para finalizar as métricas aplicadas, foram mensuradas e expostas como LOC, a quantidade de linhas de código utilizando o programa CodeAnalyzer.

A seguir serão expostos os resultados da aplicação das métricas.

A Tabela 1 mostra o resultado da análise no pacote DAO. Pode-se verificar que os valores para WMC2, RFC e LOC são superiores no projeto sem EJB, o tornando mais complexo do que a aplicação que se beneficia do EJB. Essa diferença se explica pela quantidade de linhas e processos que são necessários na aplicação sem EJB, onde é necessário realizar manualmente a instanciação da *EntityManagers*. Já na aplicação com EJB essa gerência é realizada através da anotação *@PersistenceContext*, demonstrada anteriormente na Figura 5.

Tabela 1 - Resultado da aplicação das métricas no pacote DAO.

	WMC1	WMC2	DIT	NOC	CBO	RFC	LCOM	LOC
Com EJB	63	92	3	8	29	193	407	453
Sem EJB	59	139	3	8	29	213	387	568

Fonte: Do autor.

A análise do pacote controle demonstrada na Tabela 2 não apresentou diferenças significativas. As classes do projeto com EJB possuem a anotação @EJB em alguns atributos, para o *Enterprise JavaBean* realizar a gerência do ciclo de vida do objeto. Já o projeto sem EJB precisa realizar a inicialização dos objetos dentro do construtor da classe.

Tabela 2 - Resultado da aplicação das métricas no pacote Controle.

	WMC1	WMC2	DIT	NOC	CBO	RFC	LCOM	LOC
Com EJB	106	134	7	0	31	213	455	600
Sem EJB	106	134	7	0	31	227	426	586

Fonte: Do autor.

Verificando os resultados da Tabela 3, identifica-se que no pacote converter as métricas WMC2 e RFC apresentaram as maiores diferenças. Ao contrário do projeto com EJB, os *EntityManagers* são instanciados em todos os métodos no projeto sem EJB. Isso o torna mais complexo e dificulta a manutenibilidade do código.

Tabela 3 - Resultado da aplicação das métricas no pacote Converter.

	WMC1	WMC2	DIT	NOC	CBO	RFC	LCOM	LOC
Com EJB	28	42	6	0	5	68	20	202
Sem EJB	18	54	6	0	5	80	15	212

Fonte: Do autor.

O resultado obtido na análise do pacote Modelo conforme Tabela 4, não apresentou diferenças significativas, pois o EJB não tem influência nas classes modelo dos projetos.

Tabela 4 - Resultado da aplicação das métricas no pacote Modelo.

	WMC1	WMC2	DIT	NOC	CBO	RFC	LCOM	LOC
Com EJB	124	146	7	2	10	163	805	851
Sem EJB	124	146	7	2	10	163	805	875

Fonte: Do autor.

Foram identificados nos pacotes DAO, controle e converter que o valor para LCOM foi superior no projeto com EJB. LCOM indica a quantidade de métodos que acessam uma ou mais vezes o mesmo atributo. Com o uso de EJB, é possível definir variáveis no escopo das classes, tornando possível sua utilização em diversos métodos. Um exemplo são as *EntityManagers*, que na aplicação sem EJB são instanciadas em cada método. Isso explica o maior valor para aplicação que usa EJB.

A fim de demonstrar uma visualização global das métricas aplicadas e também evidenciando o resultado positivo da aplicação com EJB na maioria das análises, foi construída a Tabela 5 fazendo um somatório dos resultados de cada pacote, permitindo a comparação como um todo das aplicações analisadas.

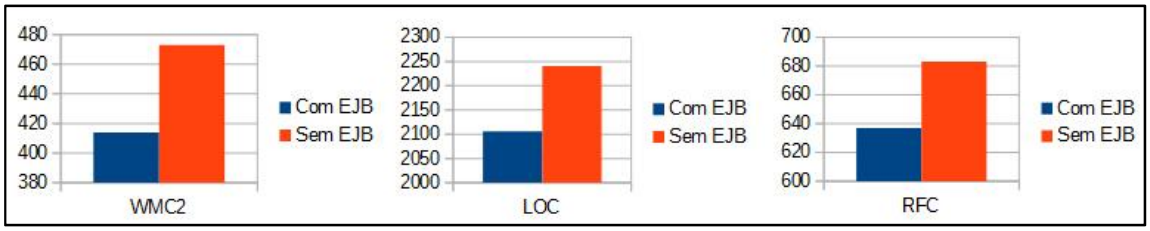
Tabela 5 – Resultados compilados

Métricas	Com EJB	Sem EJB	Diferença
WMC1	321	307	-14
WMC2	414	473	59
DIT	23	23	0
NOC	10	10	0
CBO	75	75	0
RFC	637	683	46
LCOM	1687	1633	-54
LOC	2106	2241	135

Fonte: Do autor.

Foram confeccionados os gráficos demonstrados na Figura 11, evidenciando a diferença da complexidade entre as aplicações para as métricas WMC2, LOC e RFC.

Figura 11 - Gráficos de WMC2, LOC e RFC



Fonte: Do Autor.

Analisando os gráficos prova-se que neste projeto, a aplicação que utiliza a tecnologia EJB é menos complexa.

5 CONSIDERAÇÕES FINAIS

O objetivo deste artigo era desenvolver dois estudos de caso similares. Um deles utilizando a tecnologia EJB e o outro sem a utilização do EJB, buscando demonstrar se existe superioridade do sistema em EJB em relação à outra aplicação. Foram utilizadas métricas de software para mensurar a diferença entre as aplicações. Acredita-se que o objetivo foi alcançado, pois os resultados das medições possibilitaram evidenciar a menor complexidade de código da aplicação que utiliza EJB. A tecnologia EJB disponibiliza várias ferramentas como a instanciação automática dos DAO, a gerencia do ciclo de vida dos objetos, controle transacional e pool de conexões. Isso torna o trabalho do desenvolvedor facilitado, onde poderá focar unicamente no código específico das regras de negócio.

O estudo de métricas de software contribuiu para a abstração do conhecimento, pois foi possível colocar em prática a teoria aprendida na qualidade de software. O estudo aprofundado da tecnologia EJB foi de grande valia, permitindo consolidar os conhecimentos adquiridos em sala de aula, e aplicar estes conhecimentos em um sistema que, no estudo de caso desenvolvido, segue os padrões de mercado.

Foram encontradas dificuldades na busca e estudo das métricas, pois o material não é facilmente encontrado e na maioria das vezes está em língua estrangeira. Os softwares encontrados para realização das medições são relativamente antigos e em alguns casos descontinuados, o que torna dificultada sua

utilização, e requer em determinadas situações a utilização de mais de uma ferramenta a fim de auditar os resultados.

Como sugestão para trabalhos futuros, podem ser realizadas medições quanto ao desempenho de memória e processador, simulando a utilização de recursos no servidor para aplicações com e sem EJB. Também pode ser útil a aplicação de métricas de software em sistemas existentes e maiores, auxiliando na definição de casos de testes e na manutenibilidade do código.

Por fim, aplicando as métricas durante o desenvolvimento do software e utilizando-se esta análise precoce, pode-se obter melhoras significativas no entendimento, manutenção do código e na qualidade do software em geral.

ABSTRACT

This article aims the comparison of Java web systems applying software metrics on both developed cases, seeking evidence to prove the advantage or disadvantage of using Java EE platform technologies like Enterprise Java Beans (EJB). Some software engineering metrics were used to analyze the code complexity level, as weighted methods per class (WMC), response for a class (RFC) and measurement of code size by performing the row count (LOC). With the result of this research it was identified that the use of EJB makes the system less complex, facilitates the work of the developer and contains fewer lines of code.

Keywords: Metrics. Java. Software Engineering. EJB. WMC. RFC.

REFERÊNCIAS

BAUER, Christian; KING, Gavin. *Java Persistence com Hibernate.* 1. Ed. Rio de Janeiro: Editora Ciência Moderna Ltda, 2007.

CHIDAMBER, Shyam. R; KEMERER, Chris, F. *A Metrics Suite for Object Oriented Design.* IEEE Transactions on Software Engineering, Vol. 20, Nº 6, 1994.

CODEANALYSER. *CodeAnalyzer - Multi-Platform Java Code Analyzer.* Disponível em <http://www.codeanalyzer.teel.ws/>, Acesso em: 2 jun. 2016.

CYVIS. *Software Complexity Visualiser.* Disponível em <http://cyvis.sourceforge.net/>, Acesso em: 24 mai. 2016.

DEITEL, Harvey. M; DEITEL, Paul J. *Java: Como programar.* 8. Ed. São Paulo: Pearson Prentice Hall, 2010.

GEARY, David; HORSTMANN, Cay. *Core JavaServer Faces.* 3. Ed. Rio de Janeiro: Alta Books, 2012.

GONÇALVES, Antônio. *Introdução à Plataforma Java EE 6 com GlassFish 3.* 2. Ed. Rio de Janeiro: Editora Ciência Moderna Ltda, 2011.

IBGE. *Acesso à Internet e à Televisão e Posse de Telefone Móvel Celular para Uso Pessoal*. Disponível em <www.mc.gov.br/publicacoes/doc_download/2555-pnad-tic-2013>, Acesso em: 09 nov. 2015.

MCCABE, Thomas. J. *A complexity measure*. IEEE Transactions on Software Engineering, 1976.

PRESSMAN, Roger S. *Engenharia de Software: Uma abordagem profissional.* 7. Ed. Porto Alegre: AMGH, 2011.

REDMONK. *The RedMonk Programming Language Rankings: January 2016.* Disponível em <https://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/>, Acesso em: 9 mai. 2016.

SOMMERVILLE, Ian. *Engenharia de Software.* 9. Ed. São Paulo: Pearson Prentice Hall, 2011.

SPINELLIS, Diomidis. Disponível em <http://www.spinellis.gr/sw/ckjm/>, acesso em: 23 out. 2015.