MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

SOLUÇÃO REUTILIZÁVEL PARA CONSUMO DE ANOTAÇÕES DE CÓDIGO

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA

(PIBIC/CNPq/INPE)

Bolsista: Gabriel Amboss Pinto
E-mail: gabriel.amboss@gmail.com

Orientador: Eduardo Martins Guerra
E-mail: guerraem@gmail.com

Julho de 2016

São José dos Campos

1) INTRODUÇÃO

Um framework é um artefato de software que pode ser utilizado para a simplificar a construção de aplicações. Ele pode ser considerado como uma aplicação incompleta com pontos que precisam ser especializados com as necessidades da aplicação. Através de um framework é possível reutilizar não apenas código, mas também a modelagem criada para resolver os problemas mais comuns de um determinado domínio. Como consequência, um framework não apenas acelera a produtividade de uma equipe, como também melhora a estrutura e a qualidade do código.

Nos últimos anos, a utilização de frameworks baseados em metadados vem aumentando na indústria, principalmente para a construção de aplicações corporativas e em nuvem. Através dessa abordagem, é feita uma introspecção na estrutura das classes da aplicação, identificando pontos de extensão e regras que devem consideradas em seu processamento. Para utilizar esse tipo de solução, os desenvolvedores precisam configurar metadados específicos do domínio do framework para as classes da aplicação. Isso pode ser feito através de anotações de código, fontes externas, como arquivos XML e bancos de dados, ou utilizando convenções de codificação.

A experiência mostrou que ainda existe um grande trabalho repetitivo relativo a recuperação dos metadados e sua interpretação para utilização pelo framework. Apesar do código não ser duplicado, uma lógica muito similar acaba sendo criada em diferentes projetos.

Dessa forma, o objetivo dessa iniciação científica será o desenvolvimento do protótipo de um framework para o consumo de metadados em uma aplicação na forma de anotações. Essa solução irá compor o framework Esfinge Metadata. Para a avaliação do trabalho, essa solução será utilizada na refatoração de um framework baseado em metadados chamado Esfinge Gamification.

2) CONCEITOS

Nessa sessão serão explicados conceitos importantes para o entendimento do trabalho que foi feito.

2.1) FRAMEWORK

Já explicado brevemente na introdução, um framework é um software com funcionalidades genéricas que podem ser usados diretamente em aplicações mais específicas, ou extendidos para terem funcionalidades específicas além das originais e então serem implementados em um sistema maior (RIEHLE, D., 2000).

Frameworks não devem ser confundidos com bibliotecas. Frameworks, entre outras coisas, apresentam inversão de controle, o que permite maior modularidade e extensibilidade. Isso é importante pois frameworks devem ser extensíveis, porém não modificáveis, podendo fazê-los realizar tarefas mais específicas sem comprometer o funcionamento natural da ferramenta.

2.2) INTROSPECÇÃO

Introspecção é a capacidade do código de observar propriedades e tipos de objetos durante runtime. Apesar de útil (por exemplo, o operador instanceof em Java pode ser usado para fazer o cast apropriado em um objeto de classe genérica), introspecção é um subconjunto da reflexão e é, portanto, menos poderoso. Um modelo simplificado das diferenças pode ser visto na figura 1.

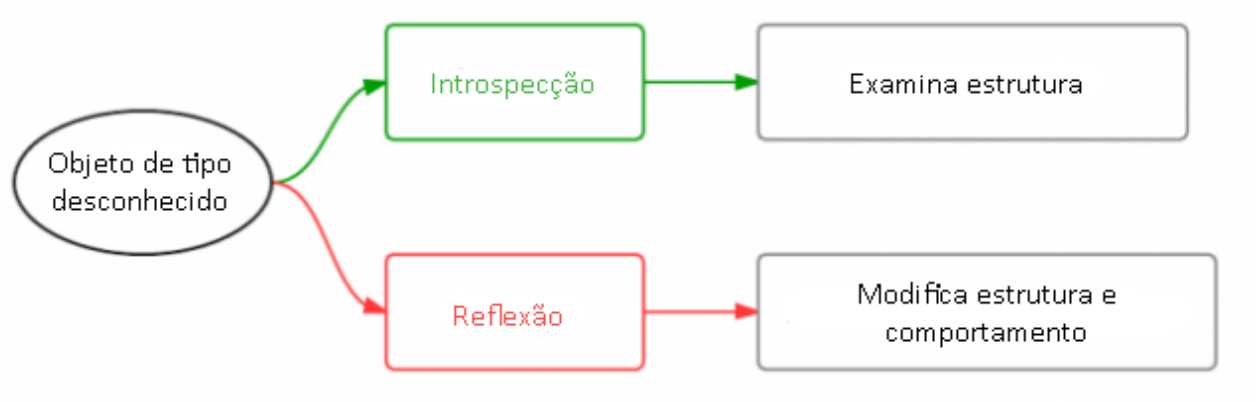

Figura 1: Diferença entre introspecção e reflexão

Introspecção por si só não consegue realizar o intuito do projeto, de ser uma solução reutilizável de metadados, porém é uma das muitas ferramentas utilizadas para se atingir o objetivo. Sem introspecção, seria impossível determinar o tipo dos objetos anotados e quais seus métodos e atributos, funcionalidade essencial para a criação do container de metadados.

2.3) REFLEXÃO

Reflexão é a capacidade de código de modificar sua estrutura e comportamento durante runtime (MALENFANT, JACQUES e DEMERS, 1995). Com reflexão é possível instanciar objetos conhecendo sua classe, e até mesmo invocar métodos desses objetos, abrindo então diversas oportunidades, como por exemplo:

- Uma classe da aplicação não precisa implementar uma interface para que o framework saiba suas propriedades, aumentando modularidade e criando código de maior qualidade.
- Um único método pode ser utilizado para tratar de diferentes classes recebidas, evitando assim a implementação de múltiplos métodos similares.
- É possível instanciar objetos de classes feitas por outras pessoas, e assim testar seus métodos em um ambiente seguro, sem o risco de exceções não-tratadas comprometerem o funcionamento do sistema como um todo.

Em java, a reflexividade está intrinsecamente ligada aos conceitos de programação orientada a objeto, no sentido em que um objeto do tipo classe gera uma instância de classe que é usada como molde para criação de um objeto qualquer. Esse esquema pode ser visualizado na figura 2.

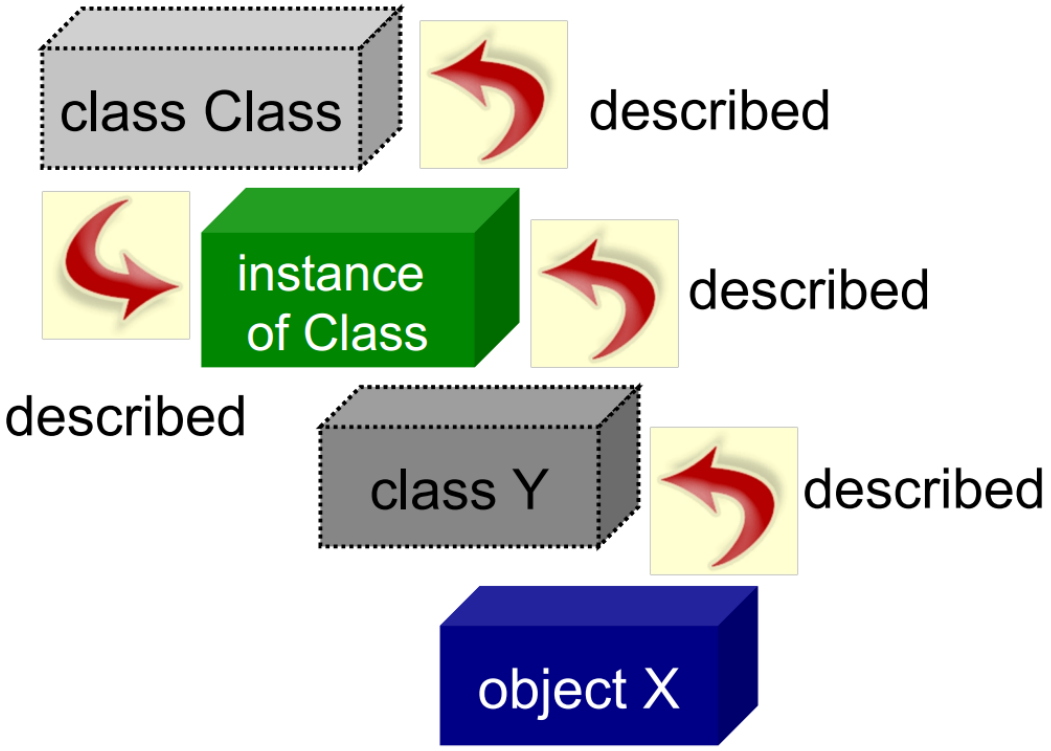

Figura 2: Estrutura de classes e objetos em Java

2.4) METADATA E ANOTAÇÕES

Metadata são, em termos simples, dados que armazenam informações sobre outros dados. Eles são principalmente usados em conjunto com reflexão para permitir uma manipulação mais específica da estrutura do código (GUENTHER e RADEBAUGH, 2004). É possível, por exemplo, marcar quais métodos modificam um parâmetro de um objeto, e então recuperar apenas esses métodos através de reflexão para fazer a validação desse campo. Em Java, metadata se encontra principalmente na forma de anotações, classes com formato específico que podem marcar outras classes, pacotes, métodos e até mesmo parâmetros (GUERRA, 2014).

Algumas anotações já existentes são bastante úteis, como por exemplo o @override, para override de um método de uma superclasse, ou @Test, para a declaração de um teste no framework JUnit. No entento, o verdadeiro poder de anotações está em poder criá-las para qualquer propósito desejado em seu programa. Isso, aliado com reflexividade, pode resolver elegantemente diversos problemas que seriam bastante complexos com apenas programação orientada a objeto, simplificando os design patterns e aumentando a confiabilidade e simplicidade do código.

No entanto, anotações não possuem nenhum tipo de inteligência, e portanto não podem processar dados nem executar funções. Elas podem ser usadas para marcar parâmetros, métodos ou classes e também podem armazenar alguns tipos simples de propriedades. Isso significa que o criador de anotações deve também criar um leitor de metadata e uma classe responsável por executar quaisquer ações utilizando a informação de

que certos objetos estão anotados. Isso pode ser bastante trabalhoso, e seria preferível o uso de uma framework já preparada para facilitar a criação e implementação de anotações, separando o uso de metadados da aplicação em si, evitando assim confusões entre as classes principais e aquelas que processam os metadados. Foi essa necessidade que criou esse projeto, pois o projeto Esfinge Framework (GUERRA, 2012) tinha como um de seus grupos uma iniciativa de gamefication para programação, então era essencial que o código referente aos elementos de jogos ficasse claramente separado do código da aplicação sendo desenvolvida.

3) ATIVIDADES REALIZADAS E RESULTADOS

3.1) ATIVIDADES REALIZADAS

Primeiramente, como o bolsista não tinha experiência prévia com metadados nem com programação reflexiva, foi necessário um período de adaptação e estudo. Dessa forma, as primeiras 4 semanas foram gastas aprendendo os básicos do framework e lendo sobre o assunto no livro Componentes reutilizáveis em Java com reflexão e anotações (GUERRA, 2014).

Em seguida, foi passada uma tarefa introdutória simples para acostumar o bolsista com o ambiente e com o funcionamento de anotações e metadata. Essa atividade envolveu a criação de anotações e a utilização delas para armazenamento e recuperação de informações. Como resultado da tarefa, teve-se uma *mock class* marcada por uma anotação com um inteiro e uma string determinados na própria declaração da classe. Esses parâmetros foram então recuperados por uma classe que recebia um objeto da *mock class*, extraia a classe e conferia a presença e os valores dentro da anotação.

Logo depois foi introduzido o principal objetivo: A criação de um container de metadados para a reutilização de metadata. Essa classe deveria automatizar o consumo e armazenamento de anotações arbitrárias aplicadas em classes, métodos e propriedades. O diagrama da figura 3 explica como esse container de matadata se encaixaria no framework Esfinge quando implementado.

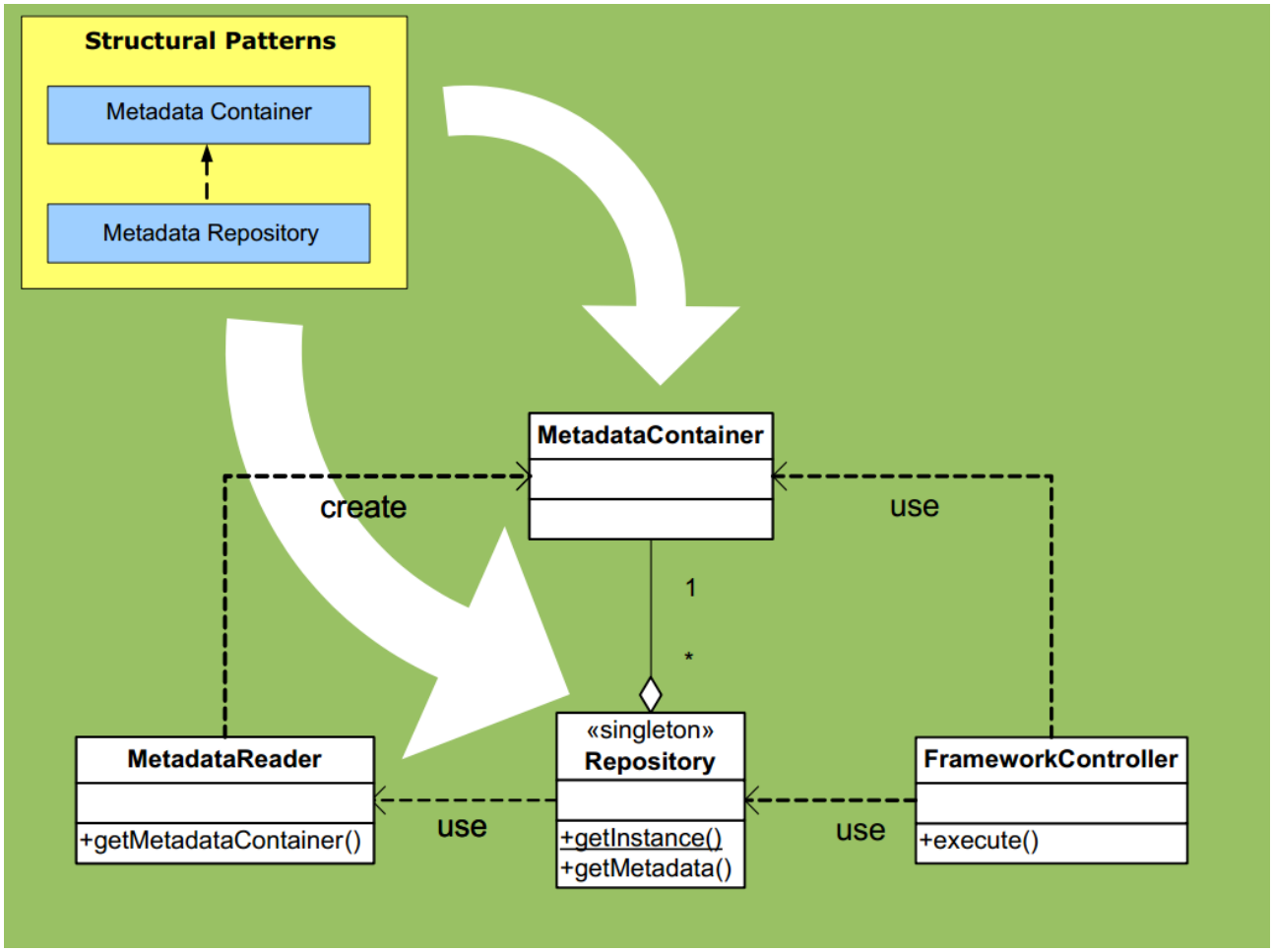

Figura 3: Diagrama explicando simplificadamente a estrutura de classes do Framework Esfinge

3.2) RESULTADOS

Para a configuração das propriedades consumidas pelo container, foi recomendado pelo orientador o uso da ferramenta BeanUtils, uma API disponibilizada pelo grupo Apache. Isso facilitou consideravelmente a tarefa de criar o container pois retirou o problema de se ter que criar propriedades e armazená-las em runtime, uma vez que não se sabe quantos, nem quais, propriedades vão precisar ser armazenadas.

Apesar de o container não ter sido desenvolvido por completo, foi possível progredir o suficiente no projeto para demonstrar uma *proof-of-concept* dentro do prazo estimado para a Iniciação Científica. As classes relevantes para tal prova se encontram nas tabelas a seguir:

Tabela 1: Classe LeitorMetadados.java

```
package org.esfinge.metadata.container;

import static org.apache.commons.beanutils.PropertyUtils.*;

import java.lang.reflect.Field;
import java.lang.reflect.Method;



public class LeitorMetadados {


	public <E> E lerMetadadosDePara(Class<?> classWithMetadata,
			Class<E> containerClass) throws Exception {

		E container = containerClass.newInstance();


		if(classWithMetadata.isAnnotationPresent(Persist.class))
			setProperty(container, "persist", true);
		else
			setProperty(container, "persist", false);

		if(classWithMetadata.isAnnotationPresent(Table.class))
			setProperty(container, "tableName",
classWithMetadata.getAnnotation(Table.class).name());

		for(Field field:classWithMetadata.getDeclaredFields()){
			if(field.isAnnotationPresent(AttAnnotation.class))
				setProperty(container, "attAnnotation", true);
		}

		for(Method m:classWithMetadata.getMethods()){
			if(m.isAnnotationPresent(MethodAnnotation.class))
				setProperty(container, "methodAnnotation", true);
		}


		return container;
	}
}
```

Tabela 2: Classe DummyClass.java

```
package org.esfinge.metadata.container;


@Table(name="teste")
@Persist
public class DummyClass {

	@AttAnnotation
	boolean attribute;

	@MethodAnnotation
	public void mockMethod(){}
}
```

Tabela 3: Classe DummyContainer.java

```
package org.esfinge.metadata.container;

public class DummyContainer {

	@ContainsAnnotation(Persist.class)
	private boolean persist;

	@AnnotationAttribute(annotation=Table.class,attribute="name")
	private String tableName;

	@ContainsAnnotation(AttAnnotation.class)
	private boolean attAnnotation;

	@ContainsAnnotation(MethodAnnotation.class)
	private boolean methodAnnotation;

	public boolean isPersist() {
		return persist;
	}

	public void setPersist(boolean persist) {
		this.persist = persist;
	}

	public String getTableName() {
		return tableName;
	}

	public void setTableName(String tableName) {
		this.tableName = tableName;
	}

	public void setAttAnnotation(boolean attAnnotation){
		this.attAnnotation = attAnnotation;
	}

	public boolean getAttAnnotation(){
		return attAnnotation;
	}

	public void setMethodAnnotation(boolean methodAnnotation){
		this.methodAnnotation = methodAnnotation;
	}

	public boolean getMethodAnnotation(){
		return methodAnnotation;
	}
}
```

Tabela 4: Classe TesteLeitorMetadados.java

```
package org.esfinge.metadata.container;

import static org.junit.Assert.*;
import org.junit.Test;

public class TestLeitorMetadados {

	@Test
	public void TestBoolean() throws Exception{
		LeitorMetadados lm = new LeitorMetadados();
		DummyContainer dc = lm.lerMetadadosDePara(DummyClass.class,
DummyContainer.class);

		assertTrue(dc.isPersist());
	}
```

```
	@Test
	public void TestString() throws Exception{
		LeitorMetadados lm = new LeitorMetadados();
		DummyContainer dc = lm.lerMetadadosDePara(DummyClass.class,
DummyContainer.class);

		assertEquals(dc.getTableName(), "teste");
	}

	@Test
	public void TestAttribute() throws Exception{
		LeitorMetadados lm = new LeitorMetadados();
		DummyContainer dc = lm.lerMetadadosDePara(DummyClass.class,
DummyContainer.class);

		assertTrue(dc.getAttAnnotation());
	}

	@Test
	public void TestMethod() throws Exception{
		LeitorMetadados lm = new LeitorMetadados();
		DummyContainer dc = lm.lerMetadadosDePara(DummyClass.class,
DummyContainer.class);

		assertTrue(dc.getMethodAnnotation());
	}
}
```

Todos os testes da tabela 4 passam, o que prova que as propriedades que o container procurou estão sendo recuperadas e armazenadas corretamente. Não apenas isso, como o container procurou as propriedades marcadas nele pelas anotações, o que permite facilmente a extensibilidade apenas adicionando o número de parâmetros que se deseja recuperar, ou, ainda mais genericamente, criando listas de propriedades de tamanho indeterminado.

A parte inacabada envolve o fato de a classe LeitorMetadados possuir os nomes das propriedades procuradas em seu código. É necessário descobrir esses nomes em runtime e adicionar seus nomes no método setProperty do BeanUtils. Feito isso, será necessário ainda refatorar o código da Esfinge Framework e adicionar o container e leitor ao framework em si, preferencialmente com alguns testes para facilitar a extensibilidade da ferramenta.

4) CONCLUSÃO

Independentemente de o projeto não ter atingido todos os objetivos definidos, ainda foi possível provar a viabilidade do sistema desejado e completá-lo parcialmente. O código fonte completo se encontra disponível para qualquer membro interessado em continuar a produção para eventualmente complementar o Esfinge Framework.

Os próximos passos foram claramente definidos, o que deve facilitar consideravelmente a finalização do projeto. É interessante também considerar que o trabalho feito agrega conhecimento sobre reflexão de código, o que pode ser útil para quaisquer membros interessados em aprender mais sobre essa ferramenta poderosa de programação.

5) REFERÊNCIAS

RIEHLE, Dirk. Framework Design: A Role Modeling Approach, 2000. Disponível em <http://dirkriehle.com/computer-science/research/dissertation/diss-a4.pdf>. Acesso em 28 jul. 2016

MALENFANT, J.; JACQUES, M.; DEMERS, F. N.; A Tutorial on Behavioral Re ection and its Implementation, 1995. Disponível em <http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf> . Acesso em 20 jul. 2016

GUENTHER, Rebecca; RADEBAUGH, Jaqueline. **Understanding Metadata**. 2004. Disponível em: <http://www.niso.org/publications/press/UnderstandingMetadata.pdf>. Acesso em: 13 fev. 2016.

GUERRA, E. M.; MATSUI, V. Componentes reutilizáveis em Java com reflexão e anotações. 1 Ed. Casa do Código, 2014. 368 p.

Guerra E. M. et al., **Projeto Esfinge**. Disponível em: <http://esfinge.sourceforge.net/>, Acesso em: 29 jul. 2016.