Uma Ferramenta para Verificação de Conformidade Visando Diferentes Percepções de Arquiteturas de Software

Izabela Melo[1], **Dalton Serey**[1], **Marco Tulio Valente**[2]

[1]Laboratório de Práticas de Software – Departamento de Sistemas e Computação – Universidade Federal de Campina Grande (UFCG) – Campina Grande – PB – Brasil

[2]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

izabela@copin.ufcg.edu.br, dalton@dsc.ufcg.edu.br, mtov@dcc.ufmg.br

Abstract. *Current architecture conformance checking tools do not take into account the different levels of abstraction for defining architectural rules. For example, architects often use a descriptive language, whereas developers prefer automatic and testable technologies. In this paper we present ARTT, an architecture conformance checking tool which allows the automatic transformation between different architectural representations. ARRT extracts rules defined in a document, written in a declarative language, and generates design tests using the DesignWizard API. The tests generated by ARTT agreed with 89,12% of the tests written by a specialist.*

Resumo. *As atuais ferramentas de verificação de conformidade arquitetural não levam em consideração os diferentes níveis de abstração para definir regras arquiteturais. Por exemplo, arquitetos de software normalmente usam linguagem descritiva, enquanto os desenvolvedores preferem tecnologias automáticas e testáveis. Este trabalho apresenta ARTT, uma ferramenta de verificação arquitetural que considera a existência de diferentes níveis de abstração arquitetural, permitindo a transformação automática entre eles. ARTT extrai regras definidas em um documento, descritas em uma linguagem declarativa, e as transforma em testes de design, utilizando DesignWizard. Os testes gerados automaticamente por ARTT concordam com 89,12% dos testes escritos por um especialista.*

URL do vídeo: www.youtube.com/watch?v=PWleNX0mqDQ

1. Introdução

Arquitetura de *software* envolve um conjunto de decisões e regras arquiteturais que estabelecem relações entre os componentes de uma aplicação [Jansen and Bosch 2005], sendo um dos artefatos mais importantes no ciclo de vida de um sistema [Knodel and Popescu 2007]. Ela interfere nos objetivos de negócios, objetivos funcionais e na qualidade do sistema. A arquitetura, uma vez criada, é raramente atualizada. Aliado a esse fato, restrições técnicas de desenvolvimento e requisitos conflitantes de qualidade que podem surgir durante a implementação são fatores que geram violações arquiteturais. Com a evolução do sistema, o número de violações tende a crescer e não serem removidas [Brunet et al. 2012]. Para evitar o surgimento de problemas causados pelo acúmulo de violações arquiteturais e garantir a adequação da arquitetura, diversas técnicas de verificação de conformidade arquitetural já foram propostas [Passos et al. 2010] [Knodel and Popescu 2007]. Tais técnicas verificam se a

implementação do sistema está de acordo com a arquitetura planejada pelos desenvolvedores e arquitetos [Clements et al. 2003].

Garantir a conformidade arquitetural de um sistema é importante para permitir reuso, compreensão do sistema, consistência da documentação com a implementação, controle da evolução do sistema e permitir a discussão entre os membros da equipe sobre a estrutura do sistema [Knodel and Popescu 2007]. Porém, as atuais técnicas para verificação arquitetural não levam em consideração os diferentes níveis de abstração da arquitetura. Enquanto a equipe de arquitetura tende a preferir linguagens que expressam propriedades de forma declarativa e/ou descritiva, equipes de desenvolvedores tendem a preferir linguagens de natureza comportamental e/ou executáveis para expressar restrições e/ou regras arquiteturais. Com isso, nem sempre a comunicação entre os diferentes níveis de abstração é consistente. Além disso, transformar uma abstração em outra pode ser uma atividade dispendiosa e sujeita a erros.

A solução proposta neste artigo teve seu início em uma cooperação com a equipe responsável pelas atividades de verificação arquitetural de software da Dataprev. No contexto dessa empresa, equipes distintas executam as atividades de verificação arquitetural e de desenvolvimento de software. Um dos arquitetos dessa empresa afirmou: *"De início, pensamos em utilizar UML para escrever a arquitetura do software. Porém, na prática, UML não é utilizado. É pesado, estrito e sofre mudanças constantes. Nós optamos por escrever as restrições arquiteturais em português e utilizar DesignWizard para realizar a checagem de conformidade. Mas escrever, manualmente, os testes de design em Java, apesar de bem aceito na equipe de desenvolvimento, tomaria tempo da equipe de arquitetura e/ou da equipe de desenvolvimento. A equipe de arquitetura precisa de uma linguagem que se aproxime do português ou do inglês, que seja mais descritiva, e de uma ferramenta que transforme nossas regras em testes de design"*.

Com isso, este artigo apresenta ARTT *(Architectural Representation Transformation Tool)*, uma ferramenta de transformação automática entre os níveis de abstração das equipes de desenvolvimento e arquitetura. O objetivo central é permitir uma comunicação mais rápida e consistente entre essas equipes. De um lado, a equipe de arquitetura define as regras arquiteturais em uma linguagem declarativa, inspirada em DCL *(Dependency Constraint Language)* [Terra and Valente 2009], uma linguagem de domínio específico criada para representar arquiteturas de *software*. Do outro lado, a equipe de desenvolvimento terá as definições das regras arquiteturais escritas em testes de *design*, os quais poderão, então, ser incorporados ao conjunto de testes funcionais do sistema. Os testes de *design* são escritos em Java (linguagem mais próxima da equipe de desenvolvimento) com o auxílio da API DesignWizard [Brunet et al. 2009], que dá suporte a análise estática de programas Java. A transformação automática proposta economiza tempo no processo de verificação arquitetural e garante que cada equipe utilize seu próprio nível de abstração.

Como estudo de caso da ferramenta, foi utilizado um sistema real (e-Pol - Sistema de Gestão das Informações de Polícia Judiciária). Nosso estudo mostrou que os testes gerados automaticamente por ARTT concordam com 89,12% dos testes gerados manualmente por um especialista. A discordância entre os dois conjuntos de testes foi causada por defeitos na API DesignWizard. Estes, já foram comunicados e discutidos com equipe de evolução da API.

O restante deste artigo está organizado da seguinte forma. A seção 2 apresenta a ferramenta ARTT. A seção 3 apresenta um estudo de caso. A seção 4 apresenta os trabalhos relacionados e a seção 5 apresenta as conclusões.

2. A Ferramenta ARTT

ARTT é uma ferramenta de verificação que leva em consideração dois níveis de abstração arquitetural: o da equipe de arquitetura, utilizando uma linguagem simples e declarativa para descrever regras arquiteturais; e o da equipe de desenvolvimento, utilizando uma linguagem de programação para escrever regras arquiteturais no formato de testes automáticos.

A Figura 1 apresenta a ideia geral de ARTT. Documentos arquiteturais, escritos numa extensão *.pdf*, são recebidos como parâmetro de entrada. Nesses documentos, as regras arquiteturais são precedidas de *"#archtest"*. Com isso, é possível detalhar, em linguagem natural, informações sobre a arquitetura que sejam relevantes para o leitor do documento. Em seguida, as regras são extraídas e armazenadas em um arquivo com extensão *.arch*. Caso o usuário da ferramenta não queira escrever um arquivo em *.pdf*, ele pode também passar como parâmetro de entrada o arquivo *.arch*. Essa especificação é, então, transformada em testes de *design* escritos em Java, utilizando DesignWizard e JUnit. Os testes, neste momento, estão prontos para serem executados, de forma a capturar violações arquiteturais.

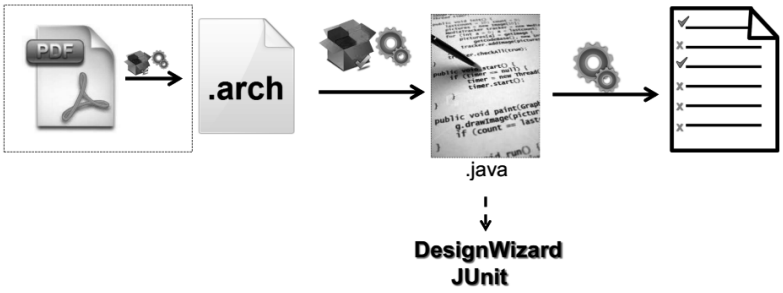

Figura 1. Funcionamento da ferramenta.

Acredita-se que utilizar uma linguagem declarativa para definir regras arquiteturais seja mais simples e rápido para o arquiteto (usuário da ferramenta em questão). Essa linguagem foi inspirada em DCL e possui a sintaxe apresentada na Figura 2. A diferença entre a linguagem utilizada nessa ferramenta e DCL é pequena. As modificações adicionadas por ARTT tiveram como objetivo apenas aproximá-la da língua inglesa. Por exemplo, enquanto em DCL define-se um módulo com *"module NOME: DIRETORIO"*, na linguagem de ARTT utiliza-se *"module NOME is DIRETORIO"* ou *"#archtest module NOME is DIRETORIO"* (se a definição for realizada num arquivo *.pdf*).

Por outro lado, testes de *design* são adequados para representar a arquitetura para a equipe de desenvolvimento, pois são escritos em uma linguagem mais próxima dessa equipe, são verificáveis durante a execução dos testes funcionais e podem garantir integridade e consistência arquitetural. Porém, escrever e especificar testes de *design* normalmente é uma tarefa dispendiosa para arquitetos de *software*. Portanto, foi criado um tradutor que transforma automaticamente uma abstração na outra para que a comunicação

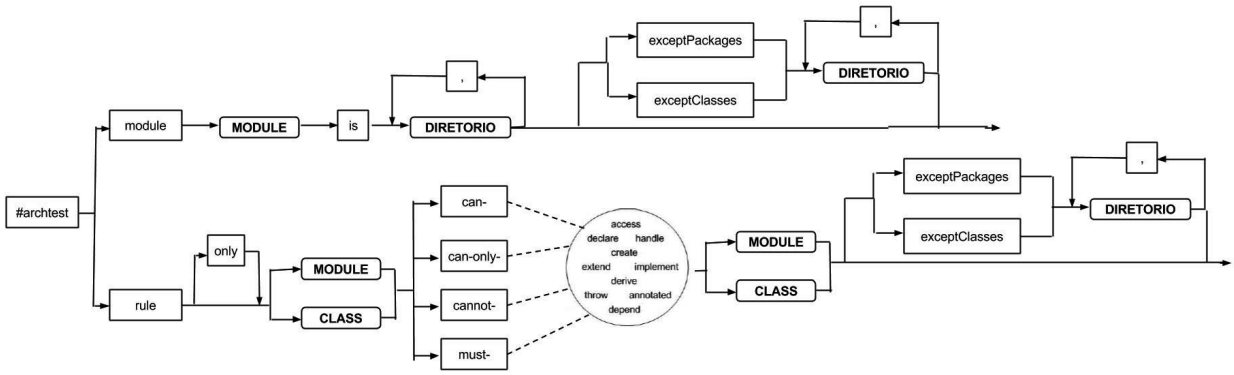

Figura 2. Sintaxe da versão de DCL utilizada por ARTT.

Tabela 1. Possíveis entradas da ferramenta.

Documento Arquitetural em .pdf
O módulo A é formado pelas classes Exemplo1 e Exemplo2. #archtest module A is Exemplo1, Exemplo2 O módulo B é formado pelas classes com endereço br.gov.classesB.* #archtest module B is br.gov.classesB Uma das regras arquiteturais definidas pelos arquitetos determina que A não pode acessar B #archtest rule A cannot-access B
Documento Arquitetural em .arch
module A is Exemplo1, Exemplo2 module B is br.gov.classesB rule A cannot-access B

seja mais rápida e consistente. As transformações são realizadas seguindo as regras apresentadas na Tabela 2. Como exemplo, a regra definida pelo arquiteto na Tabela 1 foi transformada, por ARTT, no teste de *design* apresentado no Algoritmo 1.

3. Estudo de Caso

O objeto do estudo de caso é o projeto e-Pol - Sistema de Gestão das Informações de Polícia Judiciária, desenvolvido em parceria pela Polícia Federal do Brasil e a Universidade Federal de Campina Grande. Os arquitetos do projeto e-Pol definiram três módulos e seis regras arquiteturais básicas. As regras foram escritas em um *.pdf* na linguagem proposta pela ferramenta. ARTT foi executado com sucesso, transformando as regras definidas em testes de *design*. Com a execução dos testes de *design*, foram detectadas 1489 violações arquiteturais distribuídas em 4 regras.

Em seguida, um *survey* foi aplicado aos arquitetos do projeto e-Pol com o objetivo de capturar a sua percepção com relação à ferramenta proposta. Todos os arquitetos envolvidos no experimento consideraram a linguagem expressiva, simples e fácil de ser utilizada. Além disso, eles consideraram que demandará um tempo para que os

Tabela 2. Exemplos de transformações.

Regra	Pseudocódigo

module M is S exceptClasses C	for element in S:
	  for class in allClassesDesignWizard:
	  if S contains class and C not contains class:
	  M.add(c)

rule A cannot-access B exceptClasses C	for a in A:
	  for element in a.getCalleeClasses():
	  assert B not contains element or
	  C contains element

rule only A can-implement B	for b in B:
	  for element in b.getEntitiesThatImplements():
	  assert A contains element

rule A cannot-extends B	for a in A:
	  for b in B:
	  assert not a.extendsClass(b)

membros da equipe se acostumem com a linguagem e a abordagem de testes contínuos e automáticos. Porém, esse tempo seria maior se os testes de *design* tivessem que ser escritos manualmente. O plano de verificação arquitetural definido por eles não mudará, visto que apenas será inserida uma ferramenta de transformação. Um dos arquitetos citou que *"Apesar de existir um custo, é importante implementar essa abordagem para que aumente a expectativa de vida útil do sistema"*.

Com o objetivo de verificar se os testes gerados automaticamente pela ferramenta realmente capturam violações arquiteturais, foram inseridas algumas violações mutantes no código. Como mostrado na Figura 3, as violações inseridas foram realmente capturadas pelos testes (regras 2, 4, 5 e 6).

Por fim, para verificar se os testes gerados automaticamente estavam corretos, foi solicitado que um especialista em testes de *design* (usando DesignWizard) escrevesse os testes para as seis regras manualmente. Os resultados das execuções dos testes foram comparados (Figura 4) e 162 das violações encontradas (10,87%) pelos testes gerados automaticamente não foram encontradas pelos testes do especialista. Enquanto isso, duas das violações encontradas pelos testes do especialista (0,13%) não foram encontradas pelos testes gerados automaticamente.

Nas regras que apontaram inconsistência no número de violações entre os dois conjuntos de testes, foi observado que a única diferença entre as implementações era que o especialista utilizou o método *getCallerClasses()* do DesignWizard, enquanto os testes

Algoritmo 1. Teste de *design* gerado automaticamente pela ferramenta.

```
1   // A cannot-access B
2   public void testRule1() {
3       System.out.println("Regra:  A cannot-access B");
4       Set<ClassNode> allClassesExcept = new HashSet<ClassNode>();
5       for (ClassNode caller : A) {
6           Set<ClassNode> calleeA = new HashSet<ClassNode>();
7           calleeA.addAll(caller.getCalleeClasses());
8           if (!calleeA.isEmpty()) {
9               for (ClassNode callee : calleeA) {
10                  try {
11                      Assert.assertTrue(allClassesExcept.contains(callee)
12                          || (!B.contains(callee)));
13                  } catch (AssertionFailedError e) {
14                      System.out.println(caller.getName() + " dependsOn "
15                          + callee.getName());
16                  }
17              }
18          }
19      }
20  }
```

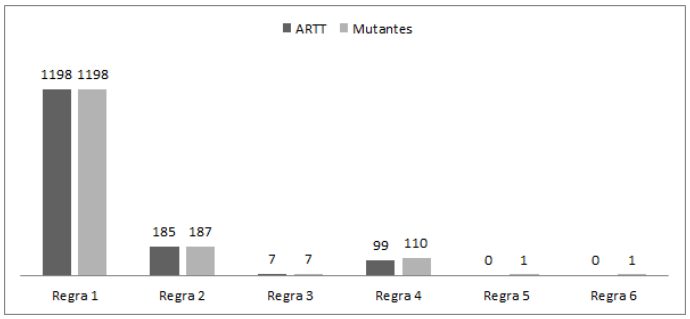

Figura 3. Comparação da execução dos testes gerados automaticamente em código com e sem mutantes.

gerados automaticamente utilizam o método *getCalleeClasses()*. Após estudo e inspeção do código do DesignWizard, concluiu-se, então, que a diferença obtida entre os resultados dos dois conjuntos de testes refere-se à API utilizada. Este fato foi comunicado à equipe de evolução da API e já está em processo de correção.

Assim, acredita-se que seja confiável utilizar ARTT para transformar a representação arquitetural descritiva em testes de *design*. Há economia de tempo, evitando que os arquitetos precisem escrever manualmente os testes de *design*. Por outro lado, os desenvolvedores podem verificar continuamente se há violações arquiteturais sendo inseridas pois os testes podem ser incorporados ao conjunto de testes do projeto e, além disso, os testes de *design* são escritos numa linguagem próxima da equipe de desenvolvimento.

Sabemos que não é possível generalizar os resultados deste estudo de caso. Além disso, os defeitos encontrados na API DesignWizard afetam nossos resultados. Somente após a correção da API será possível realizar um estudo mais aprofundado.

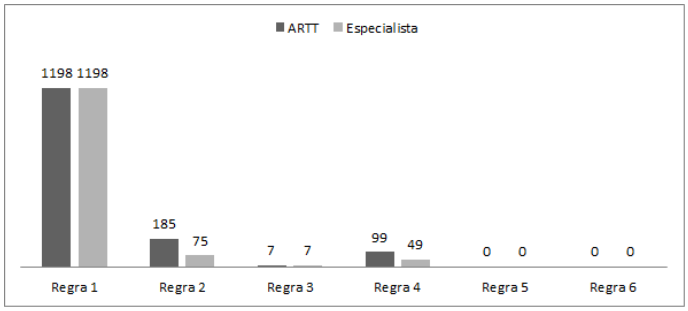

Figura 4. Comparação da execução dos testes gerados automaticamente com testes escritos pelo especialista.

4. Trabalhos Relacionados

Na área de evolução de *software*, há várias pesquisas com o objetivo de encontrar formas simples e práticas de verificação arquitetural. Em sua maioria, a automatização de alguma etapa no processo de verificação é uma questão de grande importância, visto que realizar uma verificação arquitetural de forma manual, muitas vezes, se torna uma atividade complexa, principalmente se o sistema é de larga escala [Postma 2003].

Existem várias abordagens para realizar a verificação arquitetural, mas nem sempre são usadas. Muitas vezes isso ocorre porque a linguagem utilizada na verificação é diferente da linguagem usada na aplicação [Brunet et al. 2009]. Nesse contexto, Brunet et al. desenvolveram uma API, chamada DesignWizard, que permite escrever testes de *design* para implementações em Java, usando JUnit. Com DesignWizard, os desenvolvedores podem ter um melhor entendimento do sistema, já que utiliza a mesma linguagem de desenvolvimento. Além disso, a documentação da arquitetura passa a ser executável, facilitando a tarefa de conformidade arquitetural. Como podem ser agregados ao conjunto de testes funcionais, testes de *design* são úteis para garantir que as decisões arquiteturais sejam seguidas (sem demandar uma análise manual). Porém, os arquitetos de *software* necessitam de mais tempo para entender a API e escrever as regras arquiteturais antes de serem repassadas para os desenvolvedores.

Além disso, nenhuma dessas técnicas de verificação arquitetural leva em consideração os diferentes níveis de abstração entre as equipes, nem tão pouco, uma transformação automática entre eles. Enquanto para a equipe de arquitetura um nível declarativo é mais recomendável, para a equipe de desenvolvimento um nível executável e testável tende a ser mais adequado.

Há dois trabalhos que tratam de transformações entre diferentes níveis de abstração. Pires et al. propuseram uma técnica para transformar automaticamente diagramas de classe em UML para testes de *design* [Pires et al. 2008]. Rabelo et al. propuseram uma técnica para transformar automaticamente diagramas de sequência em UML para testes de *design* [Rabelo and Pinto 2012]. Apesar de ambos serem tradutores de níveis de abstrações diferentes, nenhum deles se refere à verificação arquitetural.

5. Conclusões

DesignWizard é capaz de agilizar o processo de verificação de conformidade arquitetural [Brunet et al. 2011]. Portanto, acredita-se que a ferramenta ARTT pode introduzir econo-

mia de tempo no processo de verificação arquitetural, pois permite transformar automaticamente os dois níveis de abstração (da equipe de arquitetura e da equipe de desenvolvimento). Os desenvolvedores terão sua representação arquitetural escrita numa linguagem próxima a de desenvolvimento e, além disso, poderão incorporar os testes de *design* ao conjunto de testes funcionais do projeto. A comunicação entre as duas equipes, portanto, pode ser mais rápida e consistente. Pelos estudos realizados, a ferramenta transforma a linguagem definida neste trabalho em testes de *design* de forma satisfatória, possuindo uma concordância de 89,12% com os testes escritos manualmente por um especialista.

Como trabalho futuro, esta ferramenta pode ser estendida para outras linguagens. Para tanto, é preciso implementar a API DesignWizard para as linguagens de saída desejadas e adaptar ARTT. Ainda como trabalho futuro, pretende-se realizar uma pesquisa qualitativa para avaliar como os arquitetos de *software* realizam atividades de verificação arquitetural. Espera-se entender melhor porque a indústria não utiliza ferramentas automáticas para tal atividade, apesar de existirem diversas abordagens na academia.

Referências

Brunet, J., Bittencourt, R., Guerrero, D., and Figueredo, J. (October 2012). On the Evolutionary Nature of Architectural Violations. *Proceedings of the 19th International Conference on Reverse Engineering (WCRE 2012)*.

Brunet, J., Guerrero, D., and Figueredo, J. (2011). Structural Conformance Checking with Design Tests: An Evaluation of Usability and Scalability. *ICSM*.

Brunet, J., Guerrero, D., and Figueredo, J. (May 2009). Design Tests: An Approach to Programmatically Check you Code Against Design Rules. *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*.

Clements, P., Garlan, D., Little, R., Nord, R., and Stafford, J. (2003). Documenting software architectures: views and beyond. *Proceedings of the 25th International Conference on Software Engineering*, pages 740–741.

Jansen, A. and Bosch, J. (2005). Software architecture as a set of architectural design decisions. *Proceedings of the 5th Working Conference on Software Architecture*, pages 109–120.

Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. *In IEEE/IFIP Working Conference on Software Architecture*, pages 44–53.

Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonca, N. C. (2010). Static architecture conformance checking – an illustrative overview. *IEEE Software*, 27(5):82–89.

Pires, W., Brunet, J., Ramalho, F., and Guerrero, D. (2008). UML-based design test generation. *23nd ACM Symposium on Applied Computing (SAC 2008)*, pages 735–740.

Postma, A. (2003). A method for module architecture verification and its application on a large component-based system. *Information & Software Technology 45(4)*, pages 171–194.

Rabelo, J. and Pinto, S. E. (2012). Verificação de conformidade entre diagramas de sequência UML e código Java. *Dissertação de Mestrado. Campina Grande, Brasil*.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094.