8-10-2020

# Desenvolvimento Guiado por Interpretação de Metadados

Júlio Gustavo S-F-da-Costa
*Departamento de Computação e Automação/UFRN*, juliogustavocosta@gmail.com

Samuel Xavier-de-Souza
*Departamento de Computação e Automação/UFRN*, samuel@dca.ufrn.br

Reinaldo Antônio Petta
*Departamento de Geologia/UFRN*, petta@geologia.ufrn.br

# Desenvolvimento Guiado por Interpretação de Metadados

*Artigo em Desenvolvimento*

**Júlio Gustavo S-F-da-Costa**
Aluno de Mestrado
Departamento de Computação e Automação/UFRN
juliogustavocosta@gmail.com

**Samuel Xavier-de-Souza**
Professor Ph.D
Departamento de Computação e Automação/UFRN
samuel@dca.ufrn.br

**Reinaldo Antônio Petta**
Departamento de Geologia/UFRN
petta@geologia.ufrn.br

## Abstract

The software construction research field, from the perspective of reuse, is considered a relatively mature field in academia and industry - whose foundation is the abstraction of components. However, relevant changes in the Software Engineering scenario have been taking place in the last decade. The emergence of new technologies and changes in demands regarding the way to consume software services, mainly due to the emergence of computational clouds, reinforce the need to re-think the actual methodologies of construction and consider new possibilities. What is proposed here is a software development methodology as an alternative in face to current methodologies, with a view to building low-cost software, through maximizing the reuse of its artifacts.

**Key-words**

Separation of Concerns, Reuse, Metadata Interpretaion, Components

## Resumo

O campo de pesquisa de construção de software, pela perspectiva do reuso, é considerado um campo relativamente maduro na academia e na indústria – cujo fundamento é a abstração de componentes. Contudo, mudanças relevantes no cenário da Engenharia de Software vem ocorrendo na última década. O surgimento de novas tecnologias e as mudanças nas demandas relativas à forma de consumir serviços de software, principalmente devido à emergência das nuvens computacionais, reforçam a necessidade de re-pensar as formas de construção e considerar novas possibilidades. O que está proposto aqui é uma metodologia de desenvolvimento de software como alternativa às metodologias atuais, com vistas à construção de softwares com baixo custo, via maximização do reuso de seus artefatos.

**Palavras-chave**

Separação de Interesses, Reuso, Interpretação de Metadados, Componentes

## Introdução

A composição de *custos* relativos ao processo de construção e evolução de sistemas de software são um bom ponto de partida para compreender quais os gargalos relevantes nesse processo. A esse respeito, dois estudos de dois autores diferentes foram apropriados aqui para fundamentar a discussão que segue: (Rezende 2005) e (de Vasconcelos et al. 2017). Tais estudos mostram que o custo do ciclo de vida de um

software está mais concentradamente associado ao momento posterior a seu desenvolvimento. Para o primeiro, tal momento é denominado fase de *manutenção*; para o segundo, fase de *evolução*. Eles mostram que os custos associados a este momento do ciclo de vida podem alcançar até 90% do custo total de todo o ciclo.

Estes mesmos autores apontam algumas razões para isso. Entre elas, a razão central é relativa à demanda por mudanças, ou evolução do domínio de negócios em que situam-se os sistemas. Tais demandas ocorrem de tal forma que implicam necessidades de *atualizações* nos sistemas de software a ponto de, por vezes, constituír grave desafio o gerenciamento do processo quanto à realização do menor custo econômico. A esse respeito (de Vasconcelos et al. 2017) ainda ressalta o caráter dinâmico dessas mudanças, no sentido de haver uma tendência a aumentar ainda mais a frequência com que ocorrem.

Para fazer frente ao desafio de frear o incremento de custos devido a este cenário, muitas pesquisas vem sendo realizadas e ferramentas vem sendo propostas. Entre elas, destacamos a abstração de *componentes* como o principal achado e no qual muito esforço tem sido empregado em termos de sua definição e modos de elaboração (Vale et al. 2016), ao longo de décadas.

Entretanto, apesar de todo o esforço, construir componentes de software não parece ser uma tarefa facilmente dominada. A falta de uma definição clara e universal a respeito do que seja e de como realizá-los (Szyperski 2003) contribui em muito como desafio. Além de que afirmar seu emprego como garantia para alcançar a entrega de *artefatos reusáveis* não se sustenta facilmente. Embora o conceito e sua implementação no campo das engenharias, de modo geral, seja algo difundido e amadurecido, não parece ser o caso no campo da Engenharia de Software. Numa *revisão sistemática* do tema apresentado em (Vale et al. 2016) a conclusão é a de que não há evidências fortes o suficiente que justifiquem a simples adoção de técnicas de componentização e a consequente *eficiência* relativa se construir e reusar componentes de software. De maneira geral, segundo esses estudos, ainda permanecem em aberto a construção de soluções de problemas relativos à construção de software com vistas à redução de custos pelo emprego de técnicas de componentização, pelo menos.

É neste particular que o proposto nesse artigo encontra oportunidade para oferecer-se como alternativa de metodologia de construção de software, entre outras tais. Aqui propõe-se uma metodologia de desenvolvimento chamada *Metadata Interpretation Driven Development* (MIDD). Esta proposição tem como pressuposto uma abordagem que, prescindindo da componentização no que tange à implementação dos requisitos funcionais, resolve construir *interpretadores* de requisitos funcionais – em termos de metadados – pela perspecitva dos requisitos não funcionais. A primeira e mais relevante implicação disso, como será visto, é que dispensa-se o gerenciamento dos requisitos funcionais e seus impactos em quaisquer que sejam as fases do ciclo de vida do sistema de software, quando das demandas por transformações desses requisitos – ou seja, nos conceitos do domínio.

Este artigo segue com uma brevemente discussão crítica a respeito do tema da componentização. Em seguida, faz-se uma apresentação da abordagem MIDD, de forma sintética descrevendo um framework a partir do qual seja possível realizá-la. E, por fim, as conclusões e encaminhamentos relativos à pesquisa que segue sendo realizada quanto a essa abordagem proposta.

## Fundamentos da Crítica à Abordagem de Componentes

A despeito do que já foi apresentado anteriormente, quanto às dúvidas que são lançadas a respeito da eficiência do emprego de abordagens baseadas no uso de componentes, julgamos necessário discutir brevemente os desafios do emprego da abordagem de componentes para melhor estruturar os motivos da abordagem proposta de MIDD.

### *Ausência de Equivalência entre Níveis de Representação*

Uma observação relevante feita em (Qureshi & Hussain 2008) destaca que é no *nível do código* em que os desafios da componentização mais se manifestam e, portanto, onde mais se realizam. Contudo, componentes são abstrações que não existem como construtos em linguagens de programação, embora existam no nível da modelagem de sistemas. Eis o primeiro desafio relativo à realização de componentes em níveis mais baixos de abstração. Componentes, no nível das linguagens de programação – ou seja, no

nível de código – são construídos a partir da composição de outros construtos tais quais: classes, métodos, estruturas de dados, etc., inclusive, sem que haja uma regra de construção universalmente válida.

### *Desafios Relativos a Acoplamento e Coesão*

Dois conceitos aqui são subjacentes e relevantes para compreender a abstração de componentes: *acoplamento* e *coesão*. São conceitos basilares que guiam a construção de componentes de software de maneira a que se potencialize reuso, embora as várias definições não explicitem, ou divirjam quanto a como realizá-los a nível de código. Segundo esses dois princípios, o que há de concordância entre as definições é: tanto maior será a qualidade de um componente, quanto menos houver acoplamento entre os tais e quanto mais coesão interna houver em um componente. Ver (Gulia & Palak 2017) e (Przybylek 2011).

Quanto ao desafio de minimizar o acoplamento entre componentes, engenheiros precisam fazer frente ao fenômeno do *entrelaçamento de interesses* de software – fenômeno tratado e descrito em (Kiczales et al. 1997) – que em muito dificulta a abordagem de componentização. Sistemas de software tendem a ser construídos em camadas, e estas tais entrelaçam-se umas nas outras de modo a dificultar a quebra da solução em partes, a ponto de que cada uma dessas venham a tornarem-se componentes reusáveis. A esse respeito ver também (Tarr et al. 1999).

Já relativamente ao problema da coesão, será melhor alcançada tanto mais haja maturidade ontológia relativa ao domínio do sistema de software em questão. Aqui, mudanças ontológicas no domínio – seja no nível de definição de dados, seja no nível da definição das operações – implicam custos que, por vezes, põem em xeque o emprego da abordagem. Tal desafio, via de regra, só é superado quando se pressupõe maturidade ontológica no domínio de qualquer sistema a ser abordada a construção. Ou seja, só há potencial ganho efetivo do emprego de abordagens baseadas na construção e uso de compoenentes, quanto mais maduro seja o domínio do sistema em termos de sua ontologia.

Por fim, esses são os desafios mais relevantes para abordagens baseadas em componentização, tendo por finalidade a busca por reduções de custos ao longo do processo de construção e evolução de sistemas de software. A esse respeito, pelos trabalhos de (Vale et al. 2016), (Jalender et al. 2010) e (Frakes & Kang 2005), pode-se sustentar que o campo ainda está aberto a novas possibilidades para a abordagens que impliquem maior eficiência em termos de custo.

## Abordagem MIDD

A proposta *Metadata Interpretation Driven Development* (MIDD) parte da admissão de que são legítimas e viáveis abordagens de desenvolvimento que prescindem do uso de componentes. Algumas pesquisas, tais quais as apresentadas anteriormente, sustentam que o emprego de compoenentes como estratégia para construir softwares com baixo custo, a despeito de larga adoção pela indústria do software e de todo o esforço de pesquisa e elaboração ao longo de décadas, apontam ainda certas carências relativas às evidências de sua efetividade.

Uma vez que, conforme dito anteriormente, os seguintes desafios mantém-se postos quanto ao emprego da componentização:

1. ausência de definição universal e vazio da representação do construto *componente* a nível do código – onde os compoenentes são realizados de fato;

2. compreensão de que a existência de sistemas pressupõe ter que lidar com a pressão de demandas por evoluções no domínio em que esses existem, e de maneira cada vez mais veloz;

Resolveu-se por uma abordagem que evite o uso desse construto tanto quanto possível. E isso no sentido de uma aborgagem que, ao invés de representar em código o domínio do sistema, seja capaz de construir sistemas de software mediante a capacidade de interpretar os requisitos funcionais – isto é, os metadados do domínio – operando-os conforme o que for definido atrávés dos requisitos não funcionais.

Uma abordagem tal qual esta oferece a oportunidade para que os sistemas sejam pensados e construídos, e mesmo evoluídos, sem o acoplamento exigido entre os requisitos funcionais e requisitos não funcionais, acoplamento comum nas abordagens atuais. Deste modo, é de se esperar que os sistemas sejam um

conjugado entre duas peças distintas em finalidade no domínio dos sistemas de software. Quais sejam: as peças cuja finalidade seja representação de domínios e peças que implicam a responsabilidade de dar a computação esperada para aquelas, interpretando-as.

Outra particularidade de MIDD, dado a forma como aborda a construção de sistemas de software, é que as instâncias que implicam as peças capazes de interpretar representações do domínio – modelos – podem ser inteiramente reutilizadas ainda que seja demandada uma mudança interna a um domínio, ou mesmo uma mudança inteira de domínio, sem que seja preciso alterações no nível de código das instâncias de interpretação dos modelos de domínio.

Essa tal característica da abordagem MIDD – destacada no parágrafo anterior – a posiciona como uma aborgagem que, pressupondo testes e validações, oferece boas oportunidades para atender requisitos como os relativos às demandas de aplicações *multitenancy,* evocando assim potencial de escalabilidade de uso dessas instâncias para oferta de serviços de software. Algo que em muito encontra relavância no cenário de *nuvens computacionais*, sobretudo, em *arquiteturas serverless* ou uso de serviços baseados no emprego de *funcões lambdas*.

### *Uma Síntese da Realização de MIDD*

Do ponto de vista de sua realização, a abordagem estabelece a necessidade de haverem representações de domínio e interpretadores para essas representações. Desse modo, um framework possível para a realização da abordagem precisa definir minimamente:

1. uma ontologia de domínio, capaz de descrevê-lo com expressividade suficiente para descrevê-lo em termos do que e como deve ser computado;

2. um suporte que recepcione e informe as representações conforme a ontologia escolhida;

3. um interpretador da ontologia contida no suporte, de modo que possa dar a devida computação pela perspecitiva das demandas dos requisistos não funcionais;

Explorando esta lista, é possível constatar que dois dos três requisitos da realização da abordagem são amplamente conhecidos e empregados nas atuais abordagens de construção de sistemas de software. Ontologias e modelos – e seus suportes – são realidades conhecidas e difundidas a ponto de dispensarem-se sérias considerações a respeito. Assim, fica tanto mais facilitado o desafio da construção de sistemas conforme a abordagem MIDD.

A respeito do último requisito, e a despeito de certa novidade da aborgagem, não pode-se dizer que seja algo inteiramente desconhecido. O conceito de *interpretação* não é novo no campo da Engenharia de Software, haja vista linguagens de programação estruturadas segundo este conceito, como é o caso de PHP (GMichiaki et al. 2010), e mesmo Java com seus bytecodes (Gsavrun-Yeniçeri et al. 2013).

O interpretador, como requisito listado acima, deverá receber em algum instante do tempo de sua execução o que estiver definido no suporte que descreve o domínio e, então, criar os *statements* necessários a que realizem-se as computações necessárias sobre os dados e metadados do domínio – computações tais quais, por exemplo, recuperação e persistência de dados de domínio em caso de aplicação para SI, ou a renderização de formulários em uma aplicação web, ou mesmo a aplicação de regras de controle de acesso a recursos computacionais. A criação desses *statements* podem ser realizadas em tempo de montagem, ou em tempo de execução, ou uma combinação de ambas a partir do que, desse instante em diante, estarão disponíveis para realizar as operações do serviço de software demandados ao sistema.

## Conclusões

Retomando o conjunto de tudo o que foi exposto anteriormente, por um viés de comparação entre abordagem de desenvolvimento baseada em componetes e a abordagem MIDD, destacamos: (a) o esforço de construção de sistemas, a nível de código, está descolado do esforço de representação de domínio; (b) a partir de (a), é possível pensar o desenvolvimento e a evolução das instâncias de softwares – a implementação dos interpretadores – de forma independente de seu uso, uma vez que não constituem elas mesmas representações em nível de código do domínio do sistema; (c) a partir de (a), a existência de

uma única instância de software, não importando o domínio de negócios a que os interpretadores sejam aplicados, ou as mudanças demandadas no domínio; (d) depreendido de (c) ganha-se com a maximização de reuso, saltando à vista o potencial caráter escalável de soluções que sejam construídas por meio da abordagem MIDD.

Por meio do emprego de MIDD, tudo é feito com o fim de dotar o software da capacidade de interpretar os metadados de um domínio qualquer. Portanto, ficam eliminadas, no nível de código, as preocupações relativas a como melhor definir ontologicamente um domínio e a como melhor relacionar os conceitos do domínio, de modo a que se chegue à elaboração e realização de componentes de softwares que potencializem o reuso. Este último, conceito chave na abordagem baseada em componentes para que se alcance redução de custos relativos à construção e uso de sistemas de software.

Nossa dedicação, neste instante, concentra-se sobretudo em estudar aspectos relacionados à *construção* e o *desempenho* do uso dessa abordagem em casos concretos. Além disso, cocomitantemente, investigar seu emprego em arquiteturas baseadas em nuvens computacionais (destacadamente, em arquiteturas *serverless*) – buscando medir o potencial escalável anotado neste texto.

Por fim, como prova de conceito, está posta uma instância de código e sistema construído inteiramente segundo a metodologia MIDD. Um interpretador de metadados com o código fonte escrito em Java. O acesso público pode ser feito via o seguinte sítio Github: https://github.com/anonymous-sbes/midd.

# REFERÊNCIA


Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J. 1997. Aspects Oriented Programming. In: ECOOP'97 — Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1241, pp. 220–242.

D. A. Rezende. 2005. Engenharia de Software e Sistemas de Informação, Bras-port, Rio de Janeiro, Brasil.

Jalender, B., Govardhan, A., & Premchand, P. 2010. A Pragmatic Approach to Software Reuse. Journal of Theoretical & Applied Information Technology, 14.

J. B. de Vasconcelos, C. Kimble, P. Carreteiro, ´A. Rocha. 2017. The Application of Knowledge Management to Software Evolution, InternationalJournal of Information Management 37 (1).

C. Szyperski. 2003. "Component technology - what, where, and how?," 25th International Conference on Software Engineering, 2003. Proceedings. pp. 684-693.

Przybylek, A. 2011. Systems evolution and software reuse in object-oriented programming and aspect-oriented programming. In: Objects, Models, Components, Patterns, Lecture Notes in Computer Science, vol. 6705, pp. 163–178.

M.R.J. Qureshi, S.A. Hussain, 2008. A reusable software component-based development process model. Advances in Engineering Software, Volume 39, Issue 2, Pages 88-94.

Vale, T., Crnkovic, I., Almeida, E.S., Neto, P.A., Cavalcanti, Y.C., & Meira, S.R. 2016. Twenty-eight years of component-based software engineering. J. Syst. Softw., 111, 128-148.

W. B. Frakes, K. Kang, 2005. Software reuse research: Status and future, IEEE Transactions on Software Engineering 31 (7) 529–536.doi:10.1109/TSE.2005.85.

Gulia, P., & Palak, P. 2017. Component Based Software Development Life Cycle Models: A Comparative Review. Oriental journal of computer science and technology, 10, 467-473.

GMichiaki Tatsubori, Akihiko Tozawa, Toyotaro Suzumura, Scott Trent, and Tamiya Onodera. 2010. Evaluation of a just-in-time compiler retrofitted for PHP. SIGPLAN Not. 45, 7, 121–132. DOI:https://doi.org/10.1145/1837854.1736015.

GSavrun-Yeniçeri, G., Zhang, W., Zhang, H., Li, C., Brunthaler, S., Larsen, P., & Franz, M. 2013. Efficient interpreter optimizations for the JVM. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (pp. 113-123).