Desenvolvimento Guiado por Interpretação de Metadados

Autor 1
autor1@email
instituição
cidade, estado

Autor 2
autor2@email
instituição
cidade, estado

Autor 3
autor3@email
instituição
cidade, estado

RESUMO

O campo de pesquisa de construção de software, pela perspectiva do reuso, é considerado um campo 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 nesse cenário, 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, destacadamente as baseadas no Paradigma da Orientação a Objetos, para efeito de construção de softwares com baixo custo pela maximização do reuso de seus construtos.

CCS CONCEPTS

• **Engenharia de Software** → Metodologia de Desenvolvimento; *Componentes*; *Reusabilidade*; • **Multitenancy** → Computação na Nuven.

PALAVRAS CHAVE

entrelaçamento de interesses, reuso, metadado, interpretação, *multitenancy*

1 INTRODUÇÃO

Construir Sistemas de Informação (SI) implica planejá-los para que sejam feitos com o mínimo de entrelaçamento de interesses (do inglês *cross-cutting concerns* [Kiczales et al. 1997]). Duas são as razões para minimizar esse entrelaçamento: a primeira é a facilidade de desenvolvimento e manutenção; a segunda, garantia de maior grau de reuso das partes que constituem o software. Portanto, preocupar-se com entrelaçamento relaciona-se diretamente com redução de custo de processo de construção de software.

Tais custos possuem impacto mais destacado nas fases posteriores à fase inicial de desenvolvimento, a da manutenção [B.Jalender et al. 2011], ou evolução [de Vasconcelos et al. 2017]. É aqui que entram as dificuldades conhecidas na literatura por *entrelaçamento de interesses*, que manifesta-se sobretudo no nível do código.

SBES '20, Setembro 03–05, 2020,

Para endereçar soluções a essa questão, a do entrelaçamento de interesses, Programação Orientada a Objetos (POO) e Programação Orientada a Aspectos (POA), destacadamente em relação a várias outras tecnologias, surgiram e com elas vem-se fazendo frente a esse desafio até os dias atuais.

Em termos da produção de artefatos de software, o uso dessas tais tecnologias implica em: (a) existência de pelo menos uma instância de software para cada domínio de negócios; (b) a construção se torna tanto mais complexa, tantas mais camadas de tecnologia sejam adicionadas a POO (exemplo, adição de POA a POO para efeito de redução de custo com reuso); (c) consequência de (b): aumento de complexidades e exigência de maior qualificação técnica de pessoal para lidar com as várias camadas tecnológicas; e (d), o artefato construído, uma vez concluído, dedica-se exclusivamente à tarefa para a qual foi concebido.

Neste trabalho propõe-se uma metodologia chamada *Desenvolvimento Guiado por Interpretação de Metadados* (de agora em diante referida como DGIM) que dispensa o gerenciamento dos requisitos funcionais no nível de código quanto à dimensão dos dados – aqui assumimos o que [Cibrán 2007] diz a respeito das regras de negócios, sobre tratá-las em dimensão distinta da de dados. Tal gerenciamento ocorre a ponto de ficar inteiramente dispensado mesmo com mudança radical na definição dos dados de domínio de negócios, ou com a mudança do próprio domínio. Com DGIM a mesma instância de código pode ser aplicável a qualquer conjunto de requisitos funcionais, de quaisquer domínios de negócios. Tal característica oferece a oportunidade de aproximar as instâncias de software construídas pelo emprego desta metodologia à qualidade *multitenancy* [Odun-Ayo et al. 2017], cujo valor verifica-se, sobretudo, no contexto de uso de nuvens computacionais.

Ao final do artigo está posta uma referência a um site que disponibiliza um binário que implementa uma instância de software segundo a metodologia DGIM.

O artigo está organizado de maneira a que segue-se uma apresentação breve de críticas às metodologias atuais, sobretudo a POO e POA, pela perspectiva da componentização; uma apresentação da abordagem DGIM, conclusões e, ao final, apontamentos sobre recursos externos.

2 CRÍTICA

O princípio que emergiu da indústria e da academia que consolidou-se como sendo estruturante para a construção de software, em mais de três décadas de esforço, é o da componentização [Szyperski 2003]. Seu fim é, organizando e empacotando os artefatos, maximizar o reuso desses de maneira a que possam ser empregados em circunstâncias outras distintas das quais foram concebidos e, assim, reduzir o custo de construção e evolução.

Dois conceitos aqui são subjacentes e relevantes par compreender o achado da abstração de componentes: acoplamento e coesão. São

conceitos basilares que definem a construção de componentes de software. Por esses, maior será a qualidade de um componente, tanto mais este seja desacoplado e coeso. Entretanto, não parece algo fácil de se alcançar, justamente devido a como os interesses de softwares se entrelaçam uns aos outros.

2.1 Criticas à POO e a sua Componentização

As primeiras duas décadas das pesquisas relacionadas à Engenharia de Software, a respeito da busca por metodologias de componentização, legou-nos POO como a principal metodologia subjacente à construção de software. As duas décadas seguintes legou-nos uma série de críticas, que apesar de reconhecer a contribuição de POO, revelou também suas fraquezas quanto a oferecer um caminho bem ajustado para resolver o problema da coesão e do desacoplamento entre componentes. No mesmo período dessas últimas duas décadas as críticas fizeram surgir novas soluções que, em geral, implicam a adição de novas camadas de tecnologia a POO – seja ao processo, seja aos construtos – a fim de minimizar suas fraquezas.

O legado crítico a respeito dessas pesquisas, por todo esse período, pode ser classificado em dois eixos: as críticas quanto à forma com que POO assimila e representa fenômenos do mundo real e as críticas quanto à forma com que as ferramentas de codificação POO implementam a noção de relacionamento.

A principal crítica a POO, no primeiro eixo, é que, em POO, os objetos tratam todos os seus atributos e comportamentos como intrínsecos ao próprio fenômeno que define o objeto que daí surge por meio do processo de análise. Contudo, esses atributos e comportamentos, não são [Harrison and Ossher 1993] intrínsecos. Isso dá causa às dificuldades para elaborar ou modificar artefatos de software quando as circunstâncias obrigam mudanças no nível da ontologia, por exemplo. Como consequência, para melhor lidar com essas mudanças, todo um campo de estudos dedicado à ontologia dos conceitos, ou de objetos, precisa emergir para auxiliar a análise e construção de Sistemas de Informação – o que, tipicamente, adiciona camadas de conhecimento e tecnologia aos artefatos e aos processos de construção de software. Por óbvio, esse incremento de complexidade, trás junto uma pressão por auemnto de custos que, por vezes, pode por em xeque seus benefícios [Pohl et al. 2008]. Estas são algumas tecnologias que surgiram no esteio desse esforço: [Harrison and Ossher 1993], [Kristensen and Østerbye 1996] e [Rayside and Campbell 2000].

No segundo eixo, a crítica está centrada em como as relações entre os conceitos são realizadas, ou seja, como as construções são impactadas quando ocorrem mudanças no nível das relação entre os conceitos de um domínio de negócios. [Pearce and Noble 2006], [Bierman and Wren 2005] e [Osterbye 1999] enfatizam que a noção de relacionamento, sobretudo em linguagens orientadas à objetos, não favorece reutilização de componentes. A consequência é uma série de dificuldades que impedem os componentes daí surgidos serem facilmente reutilizados em contextos distintos dos quais foram criados.

O que se destaca aqui é: relacionar conceitos, tanto quanto definir ontologia de conceitos, é necessário e, por conta da fenomenologia em POO, torna-se um desafio para cientistas e engenheiros da software chegarem a boas definições de componentes capazes de favorecer facilidade de construção e reuso dos tais componentes. Nesse sentido POA foi o ferramental que surgiu que mais sucesso alcançou. A abordaremos brevemente a seguir.

2.2 Críticas à POA

POA é hoje possivelmente a mais importante solução que surgiu como forma de executar desacoplamento entre interesses de software, mitigando o fenômeno do entrelaçamento de interesses. Ocorre, contudo, que esse sucesso foi alcançado para desacoplar mais especificamente requisitos funcionais de requisitos não funcionais [Kiczales et al. 1997]. Portanto, seu sucesso é relativo e, no geral, aplicado apenas a esse contexto de separação de interesses.

E aqui baseia-se a crítica mais relevante para esse artigo: é preciso entender os artefatos de software como sendo construções n-dimensionais e, assim, fazer separações entre todas essas n dimensões – não apenas entre as funcionais e não funcionais. A considerar-se, por exemplo, as várias dimensões de interesses de dados que há se avaliarmos apenas os requisitos funcionais [Tarr et al. 1999].

Ademais, apesar de suas realizações, POA não se tornou unanimidade. Em [Pohl et al. 2008], por exemplo, desencoraja-se sua adoção nos produtos da empresa SAP. Outros autores declararam que POA falha em fornecer melhorias significativas para facilitar a construção dos sistemas e mantê-los quando comparado aos sistemas construídos com POO puro [Przybylek 2011]. [Steimann 2006] afirma que o uso da POA é paradoxal porque sua realização contradiz dois requisitos relevantes da construção de componentes de software: o primeiro diz respeito à capacidade de desenvolvê-los independentemente uns dos outros; o segundo, diz da legibilidade/inteligibilidade de seus artefatos.

Com base nas críticas feitas e brevemente apresentadas acima, a respeito de POO e POA, a abordagem DGIM surge apresentando uma alternativa de construção que tem como fundamento o pressuposto de que códigos de softwares não devem conter, em si, a definição dos conceitos tanto quanto possível, sob pena de surgirem limitações e dificuldades quanto a sua reutilização. Ou seja, não devem ter nos artefatos de código do software a definição dos objetos que constituem os domínios de negócios. Veremos a seguir.

3 ABORDAGEM DGIM

A proposta de DGIM, tomada como metodologia de desenvolvimento, endereça solução aos dois eixos do criticismo discutido na sessão anterior. Ou seja, a solução proposta atua para mitigar os problemas advindos de mudanças nas ontologias e/ou nos relacionamentos entre conceitos.

3.1 Fundamentação Filosófica

Em [Northover et al. 2008] constata-se que POO endereça os objetos no mundo segundo a noção de *forma ideal*. Aqui, portanto, o ponto central é responder à questão: como conciliar a inteligibilidade das coisas no mundo com o fato de que as coisas mudam?

Para POO, a resposta está na crença da existência de um caráter universal nas coisas, algo que transcende a singularidade dos vários indivíduos e atinge a todos. As categorias que surgem a partir daí são as de *classe* e *objeto*. O conceito de objeto representa a singularidade do fenômeno, a extensão dos indivíduos. Enquanto o de classe representa a forma inteligível mais abstrata na qual os

objetos são classificados, ou seja, aquilo que entre os objetos há de comum e os define.

Em face ao que é dito em [Rayside and Campbell 2000], sugerimos que a dificuldade quanto a remover o acoplamento entre os vários interesses, que constituem as classes de objetos em uma construção de software, tem a ver com o fato de que nunca alcançamos a forma ideal das coisas, dos fenômenos, das singularidades. Tudo que alcançamos da investigação de qualquer fenômeno sempre está no nível de uma representação que é sempre uma *aproximação da idealidade*. Portanto, nunca alcançamos a forma ideal que define o objeto.

E essa condição de *representação aproximada* é transportada para o código do software que pretende ser uma representação de um domínio de negócios qualquer. Isso ocorre quando inscreve-se a definição ontológico-relacional de um conceito no código do software por meio de uma linguagem de programação, no caso, orientada a objetos. O que se tem nesse instante é, também, no máximo uma boa abstração possível do conceito que poderá sofrer alterações devido a aquilo que antes não fora observado, ou compreendido. Portanto, o que est áescrito no código do software não pode ser chamada de representação ideal, ou universal. As representações, quaisquer que sejam seus níveis de abstração, não são a implementação definitiva desejada. E isso porque são sempre condicionadas a uma particular circunstância de observação.

Resta claro, portanto, a razão do custo: conceitos mudam, por isso, pela forma da fenomenologia em POO, códigos mudam. E então sente-se a necessidade de gerenciar as demandas por mudanças, a fim de minimizarem-se os custos. Contudo, como discutido anteriormente, esse é o tipo de necessidade que faz surgir agregações de novas tecnologias ao processo de construção, e ao próprio construto de classes e objetos, que são potencialmente geradores de pressão por aumento de custos.

Por fim, se entende-se que construir Sistemas de Informação é lidar sistematicamente com conceitos que representam fenômenos do mundo real, que a compreensão desses tais podem ser (e são) frequentemente modificadas em muitos casos, que os compoenetes são estruturados a partir de linguagem de programação e neles são inscritas as definições dos conceitos – definindo portanto sua funcionalidade e escopo – esclarece-se que, usando componentes ou não, inscrever definição dos conceitos em código de aplicações de software tem consequências que geram dificuldades difíceis de superar, ou mesmo impossíveis de superar.

A dificuldade, portanto, parece, não está na tecnologia em si, certamente está na forma como pensa-se, ou realizam-se, representações de domínio de negócios em nível de código. Muito possivelmente a dificuldade está em uma abordagem metodológica que deseja encontrar estabilidade conceitual para a representação do que não é. A dificuldade está em que o objeto da análise escapa ao controle de definições que se queira transcendente, ideal. Esse é o pressuposto dos fundamentos da formatação da metodologia que aqui é sinteticamente discutida.

3.2 Fundamentos de DGIM

Diante do que foi exposto, o cerne da metodologia DGIM está em que deverá ser boa prática desenvolver software dotando-os da capacidade de interpretar metadados – as definições de classes e objetos – em tempo de execução a fim de que interprete e opere quaisquer dados de quaisquer domínio de negócios, sem que com isso seja necessária a alteração das linhas de código de qualquer instância de software construída para operar esses dados. Assim, abandonamos a busca das formas ideais que pretendem representar os fenômenos do mundo a fim de inscrevê-los em código, conformando-os a um objetivo de uso específico, pretensamente bem definido.

Portanto, buscar um modo de aceitar as várias formas de representação de um domínio de negócio qualquer e, no contexto de cada uma, usar a que for mais adequada possível ao interesse de uso da aplicação em questão, parece ser o caminho mais adequado à finalidade da reutilização, ou da maximização de reuso de artefatos de softwares. Com isso resolveu-se trabalhar no nível do metametadado em código, de maneira a interpretar os metadados dos dados expressos nas respectivas representações de domínios de negócios. Sendo assim, não importa a forma (o domínio de negócios) em termos de código. Havendo um interpretador de formas, das representações em termos de seus metadados, as mudanças sempre serão aceitas sem quaisquer alterações no nível do código.

De resto, não trata-se de automatizar a construção de sistemas por meio da geração de código, mas a construção de um código que, executando, possa interpretar os metadados de um dado domínio de negócios. Eis o caráter qualitativo da DGIM que favorece construções de instâncias de software caracterizadas como *multitenancy* .

4 CONCLUSÃO

Retomando o que foi posto na primeira sessão do artigo a respeito do uso das atuais tecnologias, em comparação com POO e POA, versus o uso do DGIM, destacamos: (a) há com DGIM uma única instância de código não importando o domínio de negócios da aplicação; (b) desnecessário adição de novas camadas de tecnologias a DGIM, como em POO com POA – ou mesmo devido a rastreio em código relativo a mudanças; (c) depreendido de (b), demanda-se menos custos com tecnologias e capacitação de pessoal; e (d) depreendido de (a) ganha-se com a maximização de reuso, saltando à vista o caráter escalável de soluções impresso por sua característica *multitenancy*.

Ressalte-se, tudo é feito com o fim de dotar o software da capacidade de interpretar os metadados – não tem a ver aqui com geração de código. Ficam eliminadas, no nível de código, as preocupações relativas a como melhor definir ontológica e relacionalmente os conceitos/dados e, então, as preocupações quanto a como implementar componentes de dados, principalmente, para serem maximamente coesos e desacoplados. Note-se que POO, como metodologia, segue relevante para representar fenômenos no mundo, portanto, pode definir o próprio modo de operação do interpretador quanto a reconhecer os fenômenos/objetos com os quais precisa lidar.

Por fim, nessa fase da pesquisa, a dedicação está em estudar aspectos relacionados ao desempenho do uso dessa metodologia e, também, a investigar seu emprego em arquiteturas baseadas em nuvens computacionais - destacadamente, em meio às arquiteturas *serverless* – dado seu potencial escalável.

Atualmente testa-se dois tipos distintos de interpretadores a fim de compará-los com implementações de software tradicional (com

escrita de definições ontolócigo-relacional em seus códigos), tendo por base um mesmo domínio de negócios. Esperamos em breve publicar os resultados.

REFERENCES

Gavin Bierman and Alisdair Wren. 2005. First-Class Relationships in an Object-Oriented Language. In *ECOOP 2005 - Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 3586. Springer Berlin Heidelberg, 262–286. https://doi.org/10.1007/11531142_12

B.Jalender, A.Govardhan, and P.Premchand. 2011. Breaking the Boundaries for Software Component Reuse Technology. *International Journal of Computer Applications* 13, 6 (2011), 107–119.

María Agustina Cibrán. 2007. *Connecting High-Level Business Rules with Object-Oriented Applications: An approach using Aspect-Oriented Programming and Model-Driven Engineering*. Ph.D. Dissertation. Vrije Universiteit Brussel.

José Braga de Vasconcelos, Chris Kimble, Paulo Carreteiro, and Álvaro Rocha. 2017. The application of knowledge management to software evolution. *International Journal of Information Management* 37, 1 (2017), 1499–1506.

William Harrison and Harold Ossher. 1993. Subject-oriented Programming: A Critique of Pure Objects. *SIGPLAN Not.* 28, 10 (Octuber 1993), 42–54. https://doi.org/10.1145/165854.165932

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspects oriented programming. In *ECOOP'97 — Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 1241. Springer Berlin Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

Bent Bruun Kristensen and Kasper Østerbye. 1996. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems* 2, 3 (1996), 143–160. https://doi.org/10.1002/(SICI)1096-9942(1996)2:3<143::AID-TAPO2>3.0.CO;2-X

Mandy Northover, DerrickG. Kourie, Andrew Boake, Stefan Gruner, and Alan Northover. 2008. Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering. *Journal for General Philosophy of Science* 39, 11 (2008), 85–113. https://doi.org/10.1007/s10838-008-9068-7

Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. 2017. Cloud multi-tenancy: Issues and developments. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. 209–214.

K. Osterbye. 1999. Associations as a language construct. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of.* 224–235. https://doi.org/10.1109/TOOLS.1999.779015

David J. Pearce and James Noble. 2006. Relationship aspects. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development (AOSD '06)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/1119655.1119668

Christoph Pohl, Anis Charfi, Wasif Gilani, Steffen Göbel, Birgit Grammel, Henrik Lochmann, Andreas Rummler, and Axel Spriestersbach. 2008. Ading aspect-oriented software development in business application engineering. In *7th International Conference on Aspect-Oriented Software Development (AOSD'08)*. Brussels, Belgium.

Adam Przybylek. 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. Springer Berlin Heidelberg, 163–178. https://doi.org/10.1007/978-3-642-21952-8_13

Derek Rayside and Gerard T. Campbell. 2000. An Aristotelian Understanding of Object-oriented Programming. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 337–353. https://doi.org/10.1145/353171.353194

Friedrich Steimann. 2006. The Paradoxical Success of Aspect-Oriented Programming. *SIGPLAN Not.* 41, 10 (Octuber 2006), 481–497. https://doi.org/10.1145/1167515.1167514

Clemens Szyperski. 2003. Component Technology: What, Where, and How?. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 684–693.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 107–119.

5 RECURSOS ONLINE

Está posta uma instância de software construída inteiramente segundo a metodologia DGIM. Um interpretador de metadados escrito em Java. O acesso público ao binário pode ser feito via o seguinte sítio: https://github.com/anonymous-sbes/midd.

Neste sítio estão postas, também, as instruções de instalação e uso da versão de um interpretador DGIM desenvolvido. No repositório está também a descrição de um modelo de dados a ser empregada durante a execução da aplicação.

De resto, pode-se descompactar o arquivo em formato jar e obter mais informações a respeito da estrutura do interpretador.