GERAÇÃO DE CÓDIGO DE SISTEMAS WEB COM BASE EM PADRÕES DE *WORKFLOW*[1]

Johnny M. Honorato[1], Rodrigo Vilar[1]

[1]Departamento Ciências Exatas (DCX) – Universidade Federal da Paraíba (UFPB)
Rua da Mangueira, s/n, Companhia de Tecidos Rio Tinto
CEP 58297-000 - Rio Tinto - PB - Brasil

{johnny.menezes,rodrigovilar}@dce.ufpb.br

Abstract. *Workflow engines and manual code implementation are the main methodologies for developing workflow in web information systems, however they have limitations. This way, systems can become too dependent of engines or will require a lot of effort for manual implementation. The Scaffolding technique is an alternative because it can generate workflow code without being coupled with an engine. However, most Scaffolding techniques do not produce code with workflow functionality. This work analyzes the feasibility of using Scaffolding to generate workflow code. Idiomatic expressions were developed in Java and TypeScript languages for three Workflow Patterns documented by the academic community: Deferred choice, User-based distribution and Role-based distribution. Finally, the idioms have been transformed into code generation templates. The generated code was verified through automated tests, based on workflow patterns. Thus, the feasibility of the workflow code generator was demonstrated.*

Resumo. *As metodologias atuais para implementar funcionalidades de workflow em sistemas de informação web, a saber o uso de workflow engines e a implementação manual do código, apresentam limitações. Dessa forma, o sistema pode se tornar dependente demais das engines ou demandará muito esforço para implementação manual. Diante deste cenário, uma alternativa é utilizar a técnica de Scaffolding para gerar código de workflow sem estar acoplado a uma engine. No entanto, a maioria das técnicas de Scaffolding não produz código com funcionalidades de workflow. Este trabalho busca analisar a viabilidade de utilizar um Scaffolding para gerar código com funcionalidades workflow. Para isso, foram desenvolvidas Expressões Idiomáticas em Java e TypeScript, a partir de três Workflow Patterns documentados pela comunidade acadêmica: Deferred choice, User-based distribution e Role-based distribution. Por fim, as expressões idiomáticas foram transformadas em templates de geração de código. O código gerado foi verificado através de testes automatizados, baseados nos workflow patterns. Assim sendo, se mostrou a viabilidade do gerador de código para workflow.*

---

[1] Trabalho de Conclusão de Curso (TCC) na modalidade Artigo apresentado como parte dos pré-requisitos para a obtenção do título de Bacharel em Sistemas de Informação pelo curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE), Campus IV da Universidade Federal da Paraíba, sob a orientação do professor Rodrigo de Almeida Vilar de Miranda.

1. Introdução

Automatizar e otimizar processos de fluxo de trabalho são umas das principais preocupações das empresas, a fim de aumentar a qualidade e eficiência no gerenciamento de suas atividades. Diante desse contexto, os sistemas de gerência de *workflow*[2] (SGW) são definidos por Georgakopoulos et al. (1995) como, "um conceito intimamente relacionado à reengenharia e automação de negócios e de processos de informação em uma organização".

Workflow é um conjunto coordenado de atividades (sequenciais ou paralelas) interligadas que podem ser realizadas por sistemas de computador, agentes humanos ou uma combinação de ambos, com o objetivo de alcançar uma meta comum (Gryphon, 2001). A organização internacional *Workflow Management Coalition (*WfMC*)* (2019) define *workflow* como "a automação total ou parcial de um processo de negócio, na qual documentos, informações e tarefas são passadas de um participante para outro através de ações, conforme um conjunto de regras procedimentais".

Todavia *SGW* e *Workflows* não se limitam a processos documentais, pois também podem abranger uma variedade de sistemas que são capazes de: criar, editar e especificar diagramas de processos; monitorar fluxos de trabalho e decisões na produção para descobrir, analisar e resolver problemas técnicos; e até implementar automaticamente funcionalidades de *workflow* para sistemas informatizados, a partir de um modelo de fluxo de trabalho (Camunda, 2019).

Esta implementação automática é baseada na interpretação, em tempo de execução, dos modelos de fluxo de trabalho, realizada por uma *engine* de *workflow* que controla desde a lógica de negócio até a interface gráfica do *SGW*. Um exemplo é a implementação de um processo de compra onde um funcionário preenche um formulário com as especificações do produto a ser comprado e o gerente aprova ou recusa a compra. A partir de modelos visuais, a *engine* produz e atribui automaticamente os formulários a cada tipo de indivíduo envolvido no *workflow,* sem a necessidade de escrever código fonte ou parar a aplicação para atualizar a versão do *workflow*. Dessa forma, pode-se reduzir o custo para desenvolvimento e implantação de fluxos de trabalho nas organizações.

No entanto, para se beneficiar das *engines* de *workflow,* geralmente é necessário utilizar ferramentas e *interface* específicas, o que causa uma forte dependência entre o sistema desenvolvido e a *engine.* Mowbray (1995) aponta que o uso dessas ferramentas inibe a extensibilidade do sistema desenvolvido e restringe a possibilidade de escolha de tecnologia. Além disso, existe o risco da *engine* se transformar em um gargalo no desempenho do sistema desenvolvido.

Uma outra alternativa para implementar funcionalidades de *workflow* em sistemas é desenvolver as funcionalidades do zero em código fonte escrito manualmente. Porém a implementação de cada uma das funcionalidades requer uma grande demanda de tempo e recurso em um projeto.

Neste trabalho, foi adaptada uma solução, utilizada por diversos *frameworks* web, para aumentar a produtividade no desenvolvimento de *workflows*. Trata-se da técnica de *Scaffolding* que, a partir de metadados do domínio do problema, gera diversos artefatos do sistema. Magno et al.(2015) lista esses artefatos como: componentes de CRUD[3] e segurança, integração de sistemas, testes unitários e etc. Entretanto, até onde foi pesquisado neste

[2] Em português, fluxo de trabalho

[3] As quatro operações básicas utilizadas sobre bases de dados: cadastrar, listar, editar e remover

trabalho, a técnica *Scaffolding* ainda não é utilizada para produzir código com funcionalidades *workflow*.

Partindo desta premissa, este trabalho levanta a seguinte questão: **É possível gerar código de *SGW* web com uma abordagem semelhante ao *Scaffolding*?**

Para resolução dessa questão, esta pesquisa tem como objetivo implementar e avaliar um gerador de código para *SGW* web. Todavia existem muitas funcionalidades de *workflow* documentadas na literatura. Por exemplo a organização WfMC que documentou 131 padrões de *workflow*. Dada a limitação de tempo, foram escolhidos três padrões de *workflow* para este trabalho: *Deferred choice, Direct Distribution, Role-Based Distribution*.

Além disso uma outra contribuição deste trabalho é a proposição de Expressões Idiomáticas para cada um dos padrões de *workflow* escolhidos, especificamente para as plataformas *Spring Boot* (que utiliza a linguagem *Java*) e *Angular* (que utiliza a linguagem *TypeScript*). As expressões idiomáticas podem servir de referência tanto para implementação manual de *workflow* com para a geração de código automática.

Este trabalho está organizado da seguinte forma: na Seção 2 encontra-se a fundamentação teórica, apresentando os conceitos e as tecnologias utilizadas; na Seção 3 são descritos os passos metodológicos do trabalho; na Seção 4 foram apresentadas as decisões para criação das expressões idiomáticas nas linguagens *Java* e *TypeScript* catalogadas no Apêndice A; na Seção 5 são descritos os resultados da pesquisa, e por fim, a Seção 6 é composta pela conclusão e trabalhos futuros.

2. Fundamentação teórica

Esta seção apresenta conceitos que serviram de base para o desenvolvimento deste trabalho. É iniciada com uma introdução a *Workflow*, seguida de uma explanação sobre Sistemas de *Workflow* e *Workflow Patterns*. No final são apresentados os conceitos de Expressões idiomáticas e Geração de código automática.

2.1 *Workflow*

Segundo Pereira e Casanova (2003), *Workflow* é uma coleção de atividades organizadas para realizar um processo, executadas por sistemas de computador ou por agentes humanos. O *workflow* define a ordem de execução, as pré-condições das atividades, a sua sincronização e o fluxo de informações entre si. Segundo Cruz (2004), *workflow* é composto por:

- **Rotas** - Caminhos entre conjuntos de objetos criando o *workflow*, podem ser lineares, circulares ou paralelos;
- **Regras** - Condições necessárias para permitir a passagem de um estado ao seguinte;
- **Indivíduos** ou **Papéis** - Funções das pessoas ou programas envolvidos;
- **Atividades** - Ações conduzidas pelos utilizadores ou programas definidos em Regra.

Para ilustrar esses conceitos, a Figura 1 mostra um exemplo de *workflow* para o processo de enviar formulário de solicitação de compra em uma organização. Os **indivíduos** desse *workflow* são representados por bonecos laranjas e possuem **papéis** de funcionário ou gerente, cada um deles pode efetuar algum tipo de **atividade** que é representada por retângulos. Um exemplo de **regra** é que o formulário só é salvo no inventário quando o gestor aprova a solicitação. Uma outra regra é que só o gestor pode aprovar ou reprovar uma solicitação de compra. É importante observar que existem vários caminhos possíveis que

representam todos os cenários atingíveis desse *workflow*. Além disso, várias solicitações de compras podem ser executadas sobre esse *workflow* de forma concomitante e independente, cada uma com seu estado e histórico diferente.

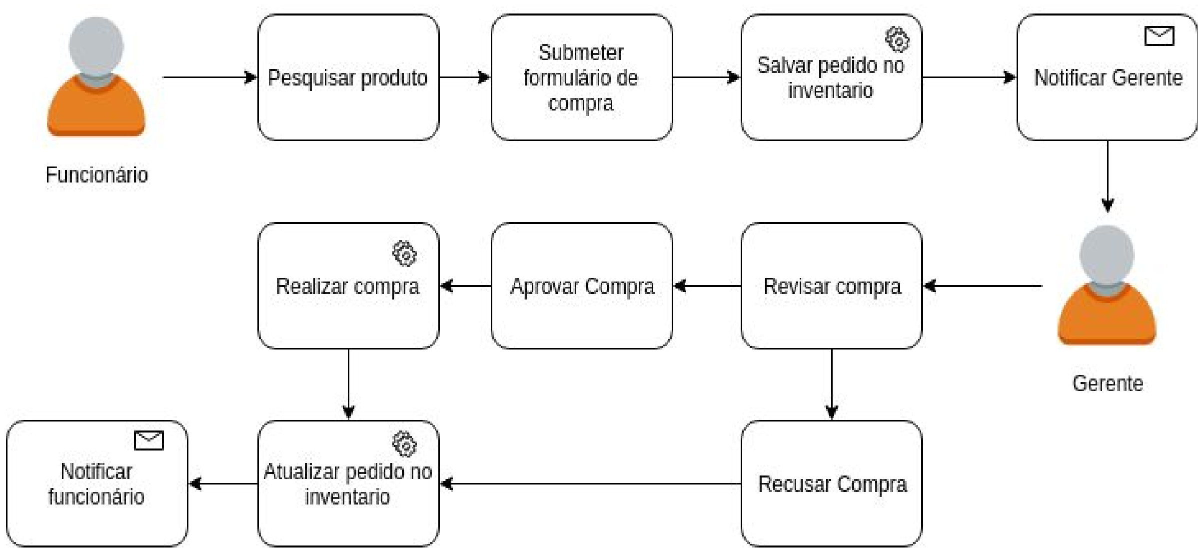

Figura 1 - *Workflow* de solicitação de uma compra.

2.2 Sistemas de *Workflow*

De acordo com Qing et al. (2001) sistemas de *workflow* são ferramentas capazes de automatizar a execução de atividades previamente estabelecidas de um processo. As principais funções de um sistema de *workflow* segundo a organização (WfMC, 2019) são: distribuir tarefas a diversos indivíduos; criar, alterar e manipular *workflows* graficamente; controlar *workflows* em tempo real e invocar ferramentas e aplicativos externos (ex.: banco de dados, leitor de pdf).

Um exemplo de sistema *workflow* é o *Bonita Open Solution*[4] que tem seu código fonte aberto. A sua documentação descreve alguns benefícios em usar a ferramenta, tais como: diminuição da circulação de documentos em papel; simplificação das atividades de arquivamento e recuperação de informações; manutenção das informações das atividade do processo sempre atualizadas; conhecimento do status do processo a cada instante.

Também pode se usar o *Bonita* para o desenvolvimento aplicações web na linguagem *Java*. O desenvolvedor precisa criar diagramas com as especificações do processo, *design* das páginas e formulários para depois especificar quais aplicativos externos devem ser usados (ex.: banco de dados e servidor de aplicativos) e, por fim, realizar o *deploy* no portal *Bonita BPM*. Vale destacar que esse processo só pode ser executado nas ferramentas do *Bonita,* causando assim dependência entre a aplicação e a ferramenta.

2.3 *Workflow patterns*

Uma vez que o uso de *workflow engines* pode causar problemas no desenvolvimento de software, uma alternativa é implementar os workflows manualmente. Porém existem várias maneiras de codificar cada funcionalidade desse tipo de *software.* Nesse contexto, os *Workflow Patterns* são descrições de funções recorrentes encontradas em processos de negócio (ex.: notificação, aprovação, decisão, solicitação de execução de tarefa) (Thom,

[4] www.bonitasoft.com

2006). Esses padrões são independentes de tecnologia e linguagem de modelagem específicas. Eles podem ser usados para: examinar a adequação de uma linguagem de processo em um projeto; avaliar pontos positivos e negativos de abordagens de processos; implementar determinados requisitos de negócios em um sistema de informações; e servir de base para desenvolvimento de linguagem e ferramentas (WfMC, 2019).

Os *Workflow Patterns* são catalogados pela WfMC em quatros tipos: *Control-Flow Patterns* (padrões de fluxo de controle) que definem os aspectos de fluxo entre várias tarefas (ex.: seqüencial, paralelo, condicional); *Data Patterns* (padrões de dados) que lidam com a passagem de dados e informações entre os componentes do *workflow*; *Resource Patterns* (padrões de recursos de fluxo) que representam as alocações de recursos para tarefas e indivíduos; e por fim *Exception Handling Patterns* (padrões de tratamento de exceções), que detalham as causas de exceções e ações que precisam ser tomadas como resultado delas.

A seguir são descritos três *workflow patterns* definidos pela WfMC*,* dos tipos *Control-Flow e Resource,* que foram abordados neste trabalho.

2.3.1 *Deferred choice*

Deferred choice é um padrão da família *Control-Flow Patterns* que especifica um ponto no *workflow* onde um indivíduo ou o próprio *software* podem realizar uma atividade dentre múltiplas possíveis. Por exemplo, em um processo para atender uma reclamação de um cliente, é possível escolher entre a tarefa de contato inicial da secretária com o cliente ou passar a tarefa para um atendimento direto com secretária. A Figura 2 representa esse exemplo do padrão *Deferred Choice*.

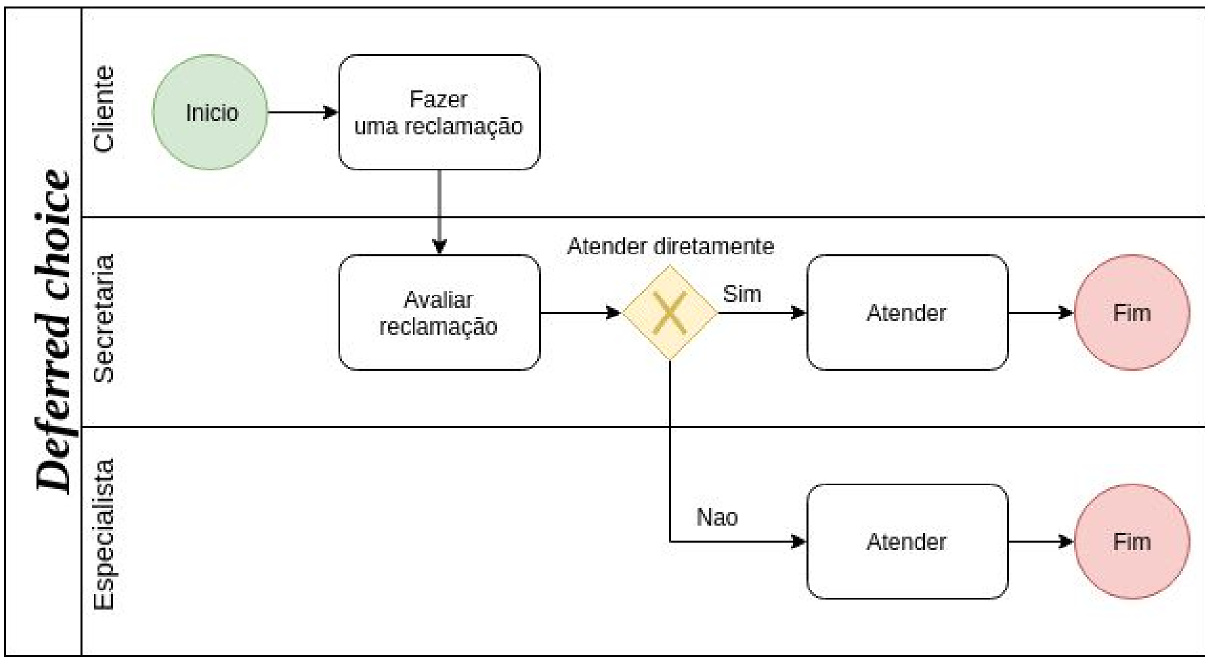

Figura 2 - *Deferred choice*

2.3.2 *Direct Distribution*

Direct Distribution é um padrão da categoria *Resource Patterns* que oferece a capacidade de atribuir uma tarefa a um indivíduo no *workflow*. Isso é útil quando se sabe que uma tarefa só pode ser realizada por um indivíduo específico. Por exemplo, em um processo de desenvolvimento de um software, existe a tarefa de revisar código, que só pode ser realizada pelo indivíduo *Fred.* A Figura 3 demonstra esse exemplo do padrão *Direct Distribution*.

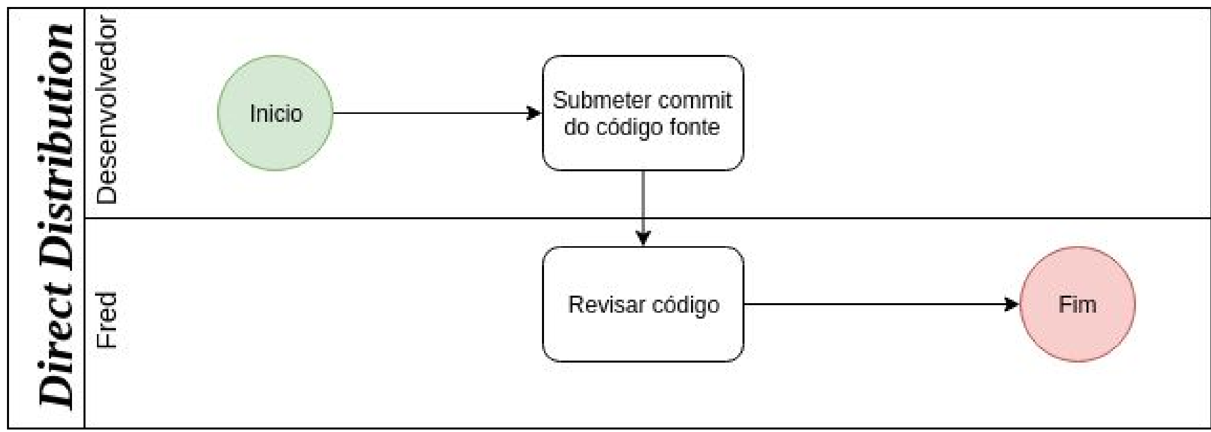

Figura 3 - *Direct Distribution*

2.3.3 *Role-Based Distribution*

Role-Based Distribution é um padrão da categoria *Resource Patterns* que oferece a capacidade de atribuir uma tarefa a um conjunto de indivíduos com características ou autoridades semelhantes no *workflow*.

A vantagem oferecida pelo *Role-Based Distribution* é que em vez de definir individualmente todas as pessoas que podem realizar uma determinada tarefa, o administrador do *workflow* pode, em um passo, determinar que todas as pessoas participantes de um grupo têm permissão de realizar uma certa tarefa. Por exemplo, em um banco existem vários gerentes e, no processo de realizar empréstimos, só eles podem avaliar as solicitações. A Figura 4 exemplifica esse exemplo do padrão *Role-Based Distribution*.

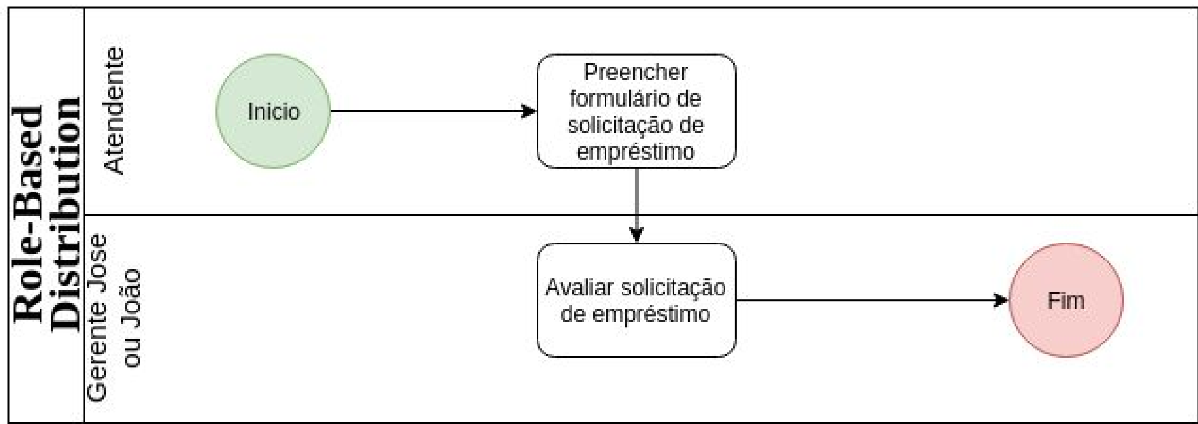

Figura 4 - *Role-Based Distribution*

2.4 Expressões idiomáticas

Enquanto os *workflow patterns* documentam como implementar os conceitos de fluxos de trabalho independente de tecnologia, as expressões idiomáticas são padrões de baixo nível específicos para uma linguagem de programação. Uma expressão idiomática descreve como implementar aspectos particulares de componentes ou as relações entre eles com os recursos da linguagem dada (Buschmann, 1996). Esse conceito também difere dos Padrões de projeto, que são descrições ou modelos genéricos de como resolver um problema que pode ser aplicados em situações diferentes (John et al, 1995).

Buschmann (*1996*) explica que, se um padrão for implementado em uma linguagem de programação específica, pode-se obter uma expressão idiomática. Por exemplo, o padrão de projeto *Singleton*, que garante que exista exatamente uma instância de uma classe em

tempo de execução, é implementada na linguagem *Smalltalk* por Kent Beck no livro *Smalltalk Best Practice Patterns*, criando uma expressão idiomática.

Este trabalho apresenta, no Apêndice A, uma coleção de expressões idiomáticas nas linguagens *Java* e *TypeScript* com base nos *Workflow Patterns* descritos na Seção 2.3. Essas expressões idiomáticas, além de servir como guia para desenvolvedores que queiram implementar as funcionalidades de *workflow* manualmente, também servem como base para criação de *templates* para geração de código do *workflow* automaticamente.

2.5 Geração de código

Uns dos problemas no desenvolvimento de uma aplicação web é a repetição de atividades como: criar entidades, configurar arquivos de persistência, implementar classes de interfaces gráficas, configurar funções específicas do *framework* e implementar operações de CRUD.

Essas tarefas consomem muito tempo e recurso em um projeto. Uma prática comum para evitar esse desperdício são os geradores de código que utilizam *templates* e metadados para gerar automaticamente o código fonte e as configurações iniciais da aplicação. Desse modo, o programador pode executar alguma funcionalidade da aplicação após poucos minutos de desenvolvimento (Herrington, 2014).

Um exemplo de gerador de código é o *Ruby on Rails*[5], que produz aplicações web na linguagem *Ruby* a partir de instruções baseadas em metadados. A Figura 5 ilustra como funciona a geração de código para aplicações no *Ruby on Rails*.

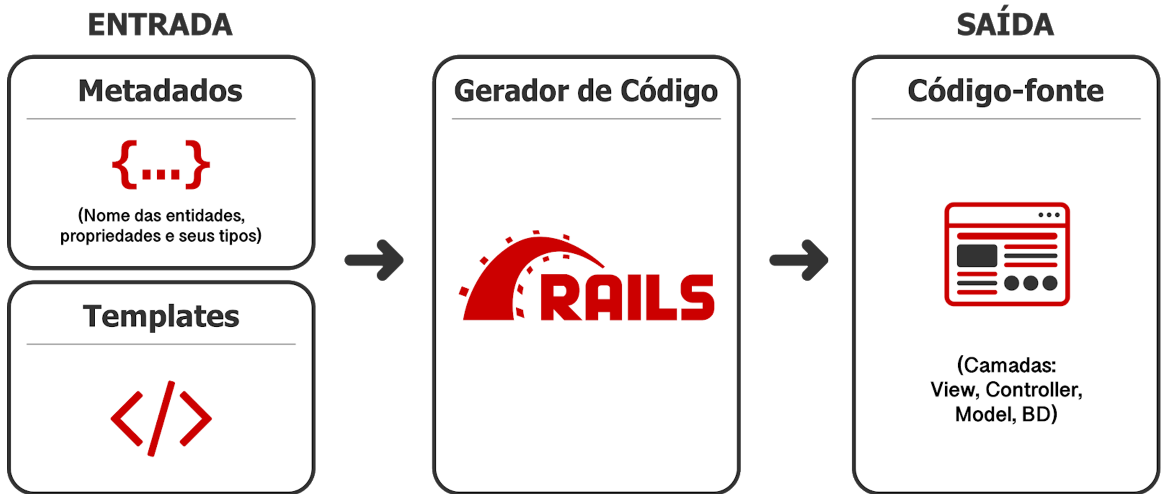

Figura 5 - Exemplo do funcionamento do gerador de código *Ruby on Rails*.

Além do *Ruby on Rails*, existem outras ferramentas que possibilitam gerar código a partir de metadados e *templates*, por exemplo, *Spring Roo, Grails* e *jHipster*. O gerador de código *Potter*[6], proposto por Vilar et al. (2014), possui o diferencial de utilizar uma abordagem de granularidade fina para os *templates*, deixando-os com poucas operações e funcionalidades bem específicas, facilitando a customização dos *templates* para geração de código mais complexo do que simples CRUDs. Além disso, o *Potter* é independente de linguagem, portanto pode ser utilizado para gerar os códigos *Java* e *TypeScript*. Por esses motivos, o *Potter* foi escolhido para implementar a solução deste trabalho.

[5] https://rubyonrails.org/
[6] https://github.com/potterjs/potter-project

3 Metodologia

A fim de verificar se é possível gerar código para sistemas web com base em *workflow patterns* com uma abordagem semelhante ao *Scaffolding*, foi adotada a metodologia *Design Science Research* como paradigma para esta pesquisa.

Design Science Research procura soluções de base tecnológica para problemas importantes (Creswell, 2010), como é o caso da implementação automática de funcionalidades *workflow* em sistemas web, e se concentra no desenvolvimento de artefatos inovadores (Wastell, 2009).

Para desenvolver esses artefatos, Wieringa (2009) propõe o ciclo regulador, conforme pode ser visto na Figura 6, que consiste em uma estrutura lógica para a resolução de problemas. O ciclo se inicia com a investigação de informações para entender o problema. A etapa seguinte são especificados os projetos de soluções. Logo após, na terceira etapa, o pesquisador valida os potenciais resultados e então passa para próxima etapa onde o projeto é implementado. Os resultados são avaliados na etapa 5 e então se pode ter o início de uma nova volta no ciclo regulador.

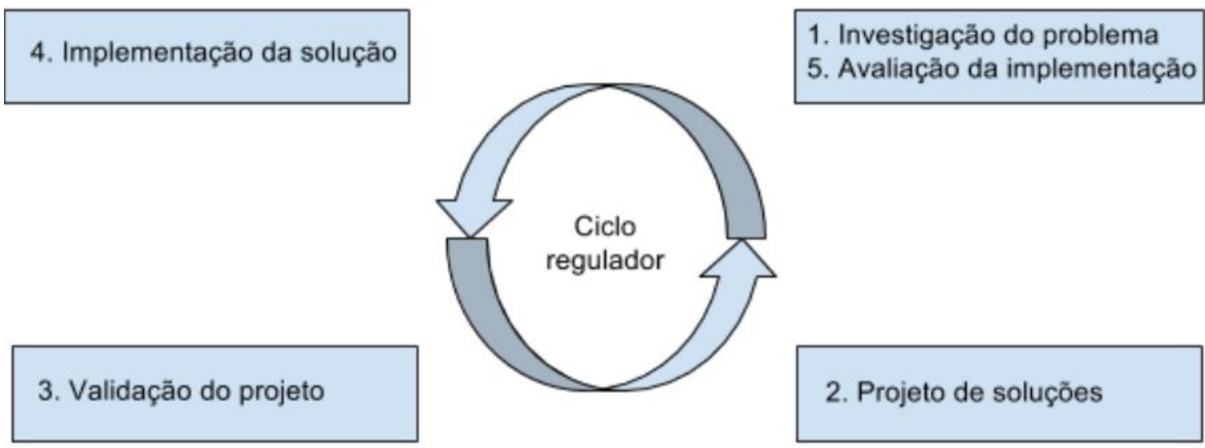

Figura 6 - Ciclo regulador de Wieringa.

Assim sendo, a metodologia deste trabalho é composta por cinco etapas, baseadas no ciclo regulador de Wieringa: (i) investigação do problema, especificamente sobre como implementar os *workflow patterns* escolhidos; (ii) definição de expressões idiomáticas para os três *workflow patterns*; (iii) revisão do código para as expressões idiomáticas; (iv) implementação de um gerador de código baseado nas expressões idiomáticas; (v) teste automatizado do código gerado, que deve ter comportamento semelhante ao código feito manualmente a partir das expressões idiomáticas.

3.1 Investigação do problema

Esta etapa foi iniciada com buscas no site *Google* Acadêmico[7] com as palavras chave "*workflow*", "*workflow engine*", "fluxo de trabalho", "expressões idiomáticas" e "geração de código" e foi dividida em cinco etapas: (i) busca preliminar do que é e como funciona *workflow*; (ii) verificação de ferramentas que provêem suporte para sistemas web com funções de *workflow*; (iii) investigação de padrões de projeto específicos para *workflow*; (iv) análise bibliográfica sobre expressões idiomáticas; (iv) estudo sobre geração de código.

---

[7] https://scholar.google.com

Os resultados das pesquisa encontram-se na Seção 2 onde foram abordados os conceitos de *workflow*, sistemas de *workflow,* expressões idiomáticas, geração de código e *workflow patterns* junto com especificação dos padrões: *Deferred Choice*, *Direct Distribution* e *Role-Based Distribution*.

3.2 Definição de expressões idiomáticas para três *workflow patterns* em um sistema web - *Angular* e *Spring Boot*

Logo após a pesquisa sobre *workflow patterns*, as linguagens *Java* e *TypeScript* foram escolhidas para implementar expressões idiomáticas por estarem bem posicionadas no ranking *RedMonk*[8] e por haver vasta disponibilidade de programadores experientes nessas linguagens no ambiente de validação deste trabalho.

Foram implementados os *workflows patterns,* seguindo os exemplos providos pelo site da WfMC e utilizando os *frameworks Angular,* responsável pelas as telas da aplicação e *Spring Boot* para regra de negócio; e a arquitetura *Representational State Transfer* (*REST) (Fielding, 2000)* para comunicação entre *front-end* e *back-end*. Por fim a implementação foi generalizada e documentadas criando as expressões idiomáticas.

3.3 Revisão do código

Após a implementação do código, iniciou a etapa de validação do projeto, onde foi utilizado a ferramenta *GitHub*[9] para realizar o gerenciamento do código. O *GitHub* além de fornecer armazenamento do código ele permitiu o controle de versão, ajudando a acompanhar as mudanças feitas no código.

Com isso o código foi passado pela a prática de revisão de código, onde dois arquitetos de *software* com mais de 10 anos de experiência e um programador júnior com experiência em geradores de código, profissionais do laboratório Virtus[10], realizaram as revisões de código com a finalidade de avaliar a solução proposta e melhorar a qualidade do código. Ao final dessa fase, tem-se um código maduro o suficiente para propor, de fato, uma expressão idiomática para cada *workflow pattern* nas linguagens *Java* e *TypeScript*. O fluxo do processo de desenvolvimento está na Figura 7.

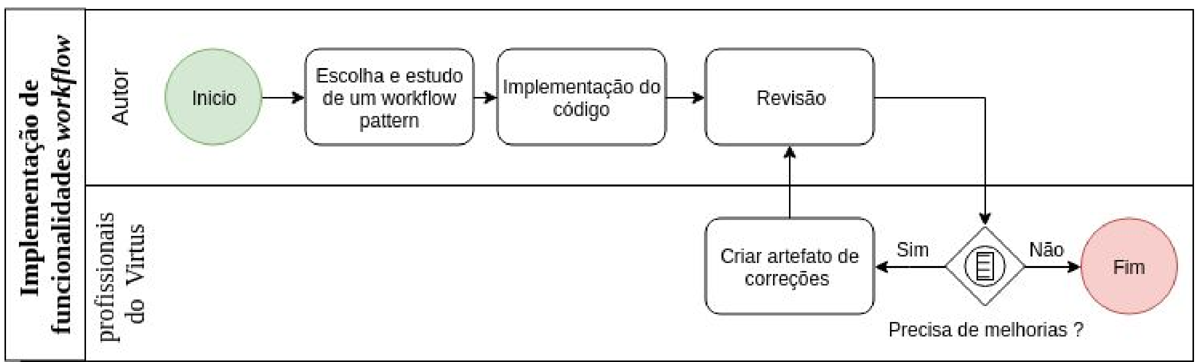

Figura 7- Diagrama de processo de desenvolvimento.

[8] https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/
[9] https://github.com/JohnnyHonorato/workflow-patterns-back, https://github.com/JohnnyHonorato/workflow-patterns-front
[10] https://www.virtus.ufcg.edu.br/

3.4 Implementação de um gerador de código baseado nas expressões idiomáticas

Para a quarta fase foi utilizado o gerador de código *Potter*, que produz códigos em diferentes linguagens baseado em *templates* e metadados, onde cada *template* contém um pequena funcionalidade do código. Apesar do *Potter* possuir uma grande biblioteca de *templates* eles só englobam funcionalidades de *CRUD* e Segurança. Sendo assim foi necessário adicionar o código parametrizado na fase anterior à sua biblioteca e criar metadados com a especificações de *workflow*.

Para parametrização, foi realizada a extração de *templates* genéricos a partir do código criado na Seção 3.2 para, em seguida, ser possível gerá-lo utilizando o novo metadado criado, como especificado na Seção 5.2. Essa etapa foi dividida em três fases: (i) Parametrizar todo o texto genérico no código, como nome de classes, métodos e etc; (ii) implementar o gerador para atribuir os valores do metadados nos *templates* parametrizados; e por fim (iii) gerar o código fonte.

3.5 Criação de testes automatizados para cada *workflow pattern*

Para avaliar as funcionalidades do código gerado automaticamente, foram criados testes automatizados fundamentados nos exemplos dos *workflows patterns* no site da WfMC. O código fonte dos testes estão publicados na ferramenta *GitHub*[11], e foram implementados utilizando o *framework Protractor* que cria testes funcionais, capazes de checar se as funções do *workflow pattern* estão sendo realizadas corretamente pelo *software*.

Os testes automatizados garantem que o código gerado possui o mesmo comportamento do código feito manualmente a partir das expressões idiomáticas propostas neste trabalho.

4 Desenvolvimento de Expressões Idiomáticas

Nessa seção explica-se decisões tomadas para criar as expressões idiomáticas mostrada no apêndice A, a partir dos *workflow patterns* especificados na Seção 2.3, através do passo metodológico descrito na Seção 3.2.

4.1 *Deferred choice*

O padrão *Deferred choice* especifica que, a partir de um ponto do *workflow*, o usuário pode realizar mais de uma ação sobre algum registro de uma entidade do sistema. Para implementar essa funcionalidade, é necessário haver um atributo que represente os estados possíveis, definidos pelos pontos do *workflow*, em cada registro. O usuário precisa saber quais ações podem ser realizadas sobre um registro em cada estado do *workflow* e deve informar qual delas será executada naquele momento. O diagrama de sequência da Figura 8 ilustra como *Deffered choice* se comporta em um sistema web.

A partir desse diagrama de sequência, as seguintes decisões precisam ser tomadas no projeto de baixo nível da solução:

I. Como representar o estado no registro no *back-end*?
II. Como o *back-end* deve informar ao *front-end* quais ações estão disponíveis para um registro em um devido momento?

[11] https://github.com/JohnnyHonorato/workflow-patterns-teste

III. Como o *front-end* deve mostrar ao usuário quais ações estão disponíveis para um registro em um devido momento?

IV. Como o *front-end* deve informar ao *back-end* a ação que deve ser executada sobre um objeto?

V. Como o *back-end* deve encapsular a lógica de negócio da execução de uma ação a ser executada sobre um registro?

VI. Como o *front-end* deve informar ao usuário se ação foi recusada ou aprovada?

Respondendo ao **item I**, o estado dos registros foi representado através de um atributo do tipo *Enumeration* no *back-end* Java (Listagem A.1) e atribuído à entidade relacionada aos estados (Listagem A.2).

Enumerations são uma forma simples de criar constantes para os N possíveis estados de qualquer *workflow*. Porém, alguns estados podem conter espaços nos seus nomes, por isso se utilizou *String Enumerations* para fornecer tanto um identificador de código fonte aos estados como um nome amigável para o usuário.

A pergunta do **item II** é sobre como o *back-end* deve informar ao *front-end* quais ações estão disponíveis para um registro. Conforme a (Listagem A.3), as ações possíveis de serem realizadas no *workflow* também são representadas com e*numerations.* Como as ações possíveis podem ser calculadas a partir do estado atual do registro, a lista de ações de um registro pode ser representada por um campo *transiente*, que não é persistido no banco de dados (Listagem A.4).

Na arquitetura base do *back-end* existem duas maneiras de representar uma entidade: Entity que é utilizado pelo o *service* e *repository* para persistir os dados e EntityDTO que é usado pelo o *controller* para transferir os dados entre o *front-end* e o *back-end* da aplicação. Dessa forma foi criada uma classe EntityActionDTO (Listagem A.5) a fim de retornar as ações para o *front-end* no estilo HATEOAS de REST[12].

Para povoar essa classe, deve ser criado um mapa na camada de serviços da entidade (Listagem A.6). Esse mapa associa os estados do *workflow* com as ações que podem ser realizadas e é inicializado pelo o método init(). Quando o *front-end* requisita as possíveis ações, o método get() (Listagem A.7) no *Controller* aciona o *Service* que depois de povoar a entidade executa o método getAvailableActions(), ele recebe como parâmetro o estado do registro e retorna quais ações estão disponíveis. Após esse processamento, o *controller* pode retornar o JSON[13] (Listagem A.8) com as ações que estão disponíveis, além dos dados do próprio registro.

[12] Restrição que provê informações para navegar entre seus endpoints de forma dinâmica: https://spring.io/projects/spring-hateoas

[13] Javascript object notation é um formato utilizados para comunicação entre serviços web

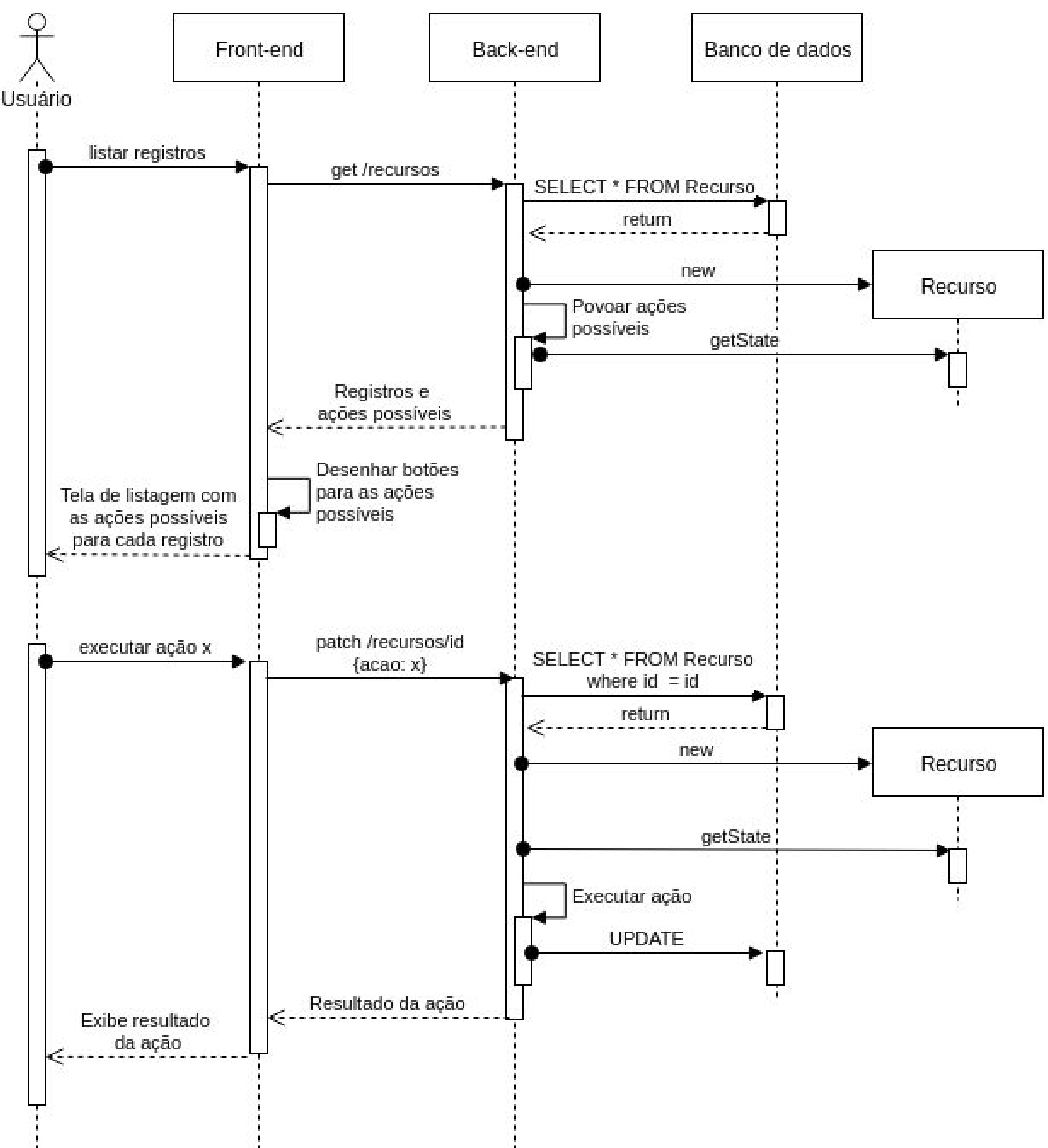

Figura 8 - Diagrama de sequência do *Deferred choice* nas camadas de uma aplicação web.

A pergunta do **item III** refere se a como o f*ront-end* mostra ao usuário quais ações estão disponíveis para um objeto**.** O *front-end* desenha um botão para cada ação possível retornada pelo *back-end* como demonstrado na (Listagem A.9).

A pergunta do **item IV** se refere de como o comando de execução da ação deve ser enviado ao *back-end,* na (Listagem A.10) se exibe uma função responsável por enviar um JSON definido na (Listagem A.11) e como resultado retorna uma notificação ao usuário respondendo também à pergunta do **item VI** a partir do id da entidade escolhida e da ação que o usuário deseja executar.

A pergunta do **item V** refere-se a como *back-end* deve executar uma ação requisitada pelo o *front-end*. É preciso criar uma interface chamada StateLogic (Listagem A.12) para representar a lógica de cada ação a partir dos estados do *workflow*. Para cada estado, deve haver uma implementação da interface StateLogic e, para cada ação, a implementação deve conter as regras de negócio relacionadas às passagem de estado (Listagem A.13). Para evitar linhas de códigos repetitivas em regras de negócios iguais, se criou a classe de StateLogicAdapter (Listagem A.14), que pode implementar um comportamento *default* para cada ação e apenas os métodos com lógica específica precisam ser implementados nos estados concretos.

Para o gerenciamento dessas classes é utilizado um *Service*, onde se instancia todas as interfaces e com auxílio de um mapa relaciona com seus respectivos estados no método init(). Quando o *front-end* requisita uma ação, o *Controller* invoca o método *changeState()* onde, via reflexão, acessa a interface de nome igual o estado da entidade e executa o método referente a ação. Dessa forma, se reduz o acoplamento do *Service* com as classes concretas de lógica.

Para melhor entendimento, a Figura 9 ilustra o diagrama de classes com os componentes da expressão idiomática *Deferred choice* no back-end.

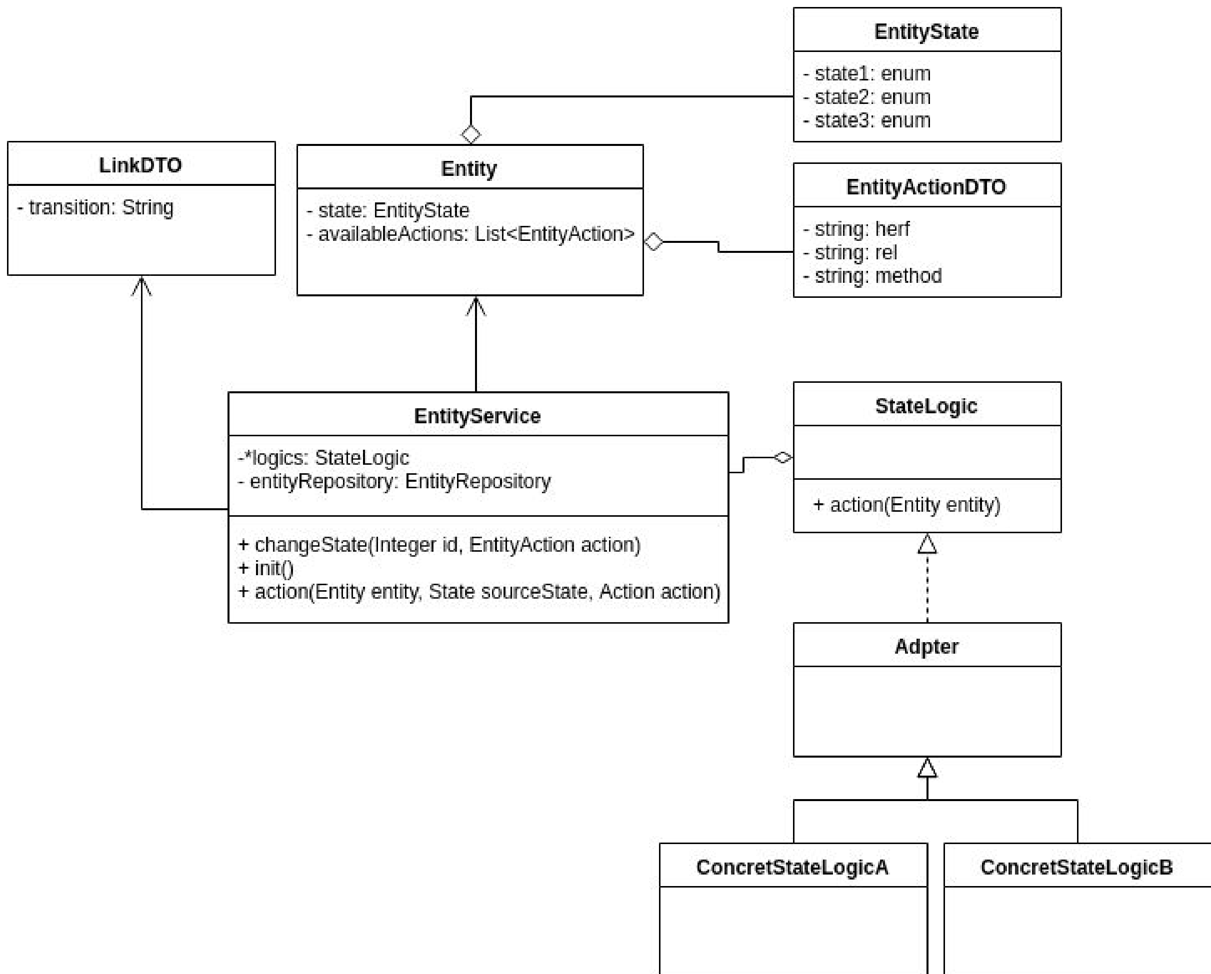

Figura 9 - UML da expressão idiomática *Deferred choice*

4.2 *Direct distribution* e *Role-based distribution*

Os padrões *Direct distribution* e *Role-based distribution* oferecem a capacidade de atribuir dentro de um *workflow,* uma instância de uma tarefa para determinado recurso que pode ser um usuário ou um grupo, a partir de seus identificadores. Para implementar essa funcionalidade, o *back-end* deve verificar se há um identificador de recurso na ação, antes de executá-la,e se o usuário logado está associado a esse recurso.

Para complementar esses dois padrões na solução *Deferred choice,* as seguintes decisões precisam ser tomadas no projeto de baixo nível da solução:

I. Como o *back-end* deve atribuir as instâncias de tarefa aos usuários.
II. Como o *back-end* verifica se os usuários estão aptos a realizar as tarefas.

Respondendo ao **item I**, o *back-end* deve possuir uma anotação para que seja capaz de marcar um método (Listagem A.16), de maneira que essa marcação possa ser verificada em tempo de execução. Logo após deve sinalizar os método que precisam de verificação passando como parâmetro o identificador, como demonstrado na (Listagem A.17).

A pergunta do **item II** refere se a como *back-end* verifica se o usuário está apto a realizar uma tarefa. O back deve recuperar as informações da anotação e do usuário quando método marcado é executado, para isso acontecer o *back-end* deve adicionar linhas de código da (Listagem A.18) ou (Listagem A.19) no método *action()* da classe *Service* da (Listagem A.15) e implementar o método isAllowed() da (Listagem A.20) que a partir do identificador do usuário logado e o identificador da anotação verifica se o usuário está apto para realizar a tarefa ou não.

5 Resultados

A partir do desenvolvimento das expressões idiomáticas, nas linguagens *Java* e *TypeScript,* a partir dos padrões de *workflow* demonstrados na Seção 2.3, foi implementado um gerador de código que produz automaticamente funcionalidades *workflow* para sistemas web. O gerador recebe como parâmetros de entrada os metadados demonstrado na Seção 5.2 e *templates* criados a partir das expressões idiomáticas demonstradas na Seção 4.

Com isso, a solução desenvolvida possibilita criar um sistema web automaticamente, onde os usuários cadastrados podem executar uma série de tarefas dentro de um *workflow* pré definido. O sistema também atribui uma determinada tarefa a ser executa para um usuário ou grupo específico.

A seguir, na subseção 5.1, tem-se uma demonstração do código-fonte do sistema gerado.

5.1 Trecho de código transformado em template

Nesta subseção, mostra-se um exemplo com o código da aplicação *Java* desenvolvido manualmente na Listagem 1 e o seu respectivo *template* na Listagem 2. Esta demonstração visa esclarecer em quais pontos no código pode ser substituído.

```
public void changeState(Integer id, TaskStateAction action){
  TaskState sourceState = task.getTaskState();
  service.action(task, sourceState, action);
}
```

Listagem 1 - Exemplo de código estático na linguagem Typescript.

```
public void changeState(Integer id, <%= entity.name %>StateAction action) {
  <%= entity.name%>State sourceState = <%=entity.name()%>.get<%=entity.name%>State();
  service.action(<%= entity.name.toLowerCase() %>, sourceState, action);
}
```

Listagem 2 - Exemplo de template na linguagem Typescript.

Observa-se que em alguns pontos do código, as propriedades da entidade foram substituídas por funções na linguagem EJS[14] utilizada pelo *Potter*. O nome da entidade enviada por parâmetro a função *changeState*, por exemplo, ao invés de inseri-lo diretamente, usa-se o seguinte comando: <%= entity.name %> que em seguida será substituído pelo o nome da entidade descrito no metadados da listagem 3.

Como isso o desenvolvedor não precisa escrever todo o código, que normalmente é repetitivo e requer uma grande quantidade tempo, ele só precisa se preocupar em criar os metadados com a configuração do *workflow* que serão utilizados para gerar a aplicação.

5.2 Metadado com especificações Workflow

O metadado para *workflow* foi definido como um arquivo JSON. Na Listagem 3, podemos observar um metadado que é capaz produzir todas as expressões idiomáticas definidas na Seção 4, onde uma entidade de nome Solicitação possui um objeto chamado *workflow*, que é composto por dois atributos: *actions*, responsável por indicar quais ações podem ser executadas no fluxo; e *status* que possui todos os estados com a indicação das ações que podem ser executadas, do estado destino se a ação for executada e da condição (usuário ou papel) para executar a ação.

```
"entities": [
  {
    "name": "Solicitacao",
    "labels": { "sigular": "Solicitacao", "plural": "Solicitacoes" },
    "id": "id",
    "properties": [ ],
    "workflow": {
      "actions": ["Abrir", "Responder", "Finalizar"],
      "states": {
        "Esperando":[ { "action": "Abrir",
                        "target": "Aberto",
                        "role": "secretaria"
                      }],
         "Aberto": [ {  "action": "Responder",
                        "target": "Respondido",
                        "user": "sue"
                    }],
         "Respondido": [ { "action": "Finalizar", "target": "Finalizado" }],
         "Fechado": [ ]
      }
    }
  } ]
```

Listagem 3 - Exemplo de metadado

[14] Linguagem de templates que permite gerar código com JavaScript

5.3 Tela da aplicação

O componente de listagem exibido na Figura 10 é responsável por apresentar os atributos e ações de cada entidade.

Id	Task.date	Task.taskState	system.actions
2	13.02.2019	initial	OpenOffice
3	21.03.2019	opened	DoBanking
4	24.03.2019	banked	CloseOffice
5	03.06.2019	closed	

Figura 10 - Componente de apresentação de entidades.

É através deste componente que o usuário executa ações. As ações são representadas por botões e são exibidas conforme a permissão dada a cada usuário. Estes botões ao serem clicados enviam requisições ao *back-end* com a identificação da entidade e a ação a ser executada. Se a ação for realizada o sistema informa um mensagem de confirmação, é mudado o estado da entidade e pode aparecer uma nova ação, caso contrário, é mostrada uma mensagem de erro e nada é alterado na entidade.

Para validar esse componente que foi criado manualmente e depois gerado, foram criados testes automatizados onde os caso de teste seguiam exemplos de *workflow* do site WfMC. Na Listagem 4 mostra um exemplo de resultado, onde Sue e Bob são usuários e cada um tem um acesso específico as tarefas. Ex: Somente Sue tem o papel de abrir o arquivo.

```
Spec started
  Casos de teste workflow
    ✓ Sue tenta abrir arquivo e é aceito
    ✓ Sue tenta fazer deposito e é negado
    ✓ Sue tenta fechar arquivo e é negado
    ✓ Bob tenta abrir arquivo depois que Sue abre e é negado
    ✓ Bob tenta fazer deposito depois que Sue abre e é aceito
    ✓ Bob tenta fechar arquivo depois que Sue abre e é negado
    ✓ Sue tenta abrir arquivo depois que ele foi aberto e foi feito o deposito e é negado
    ✓ Sue tenta fazer deposito depois que ele foi aberto e foi feito o deposito e é negado
    ✓ Sue tenta fechar arquivo depois que ele foi aberto e foi feito o deposito e é aceito

9 specs, 0 failures
Finished in 125.688 seconds

Executed 9 of 9 specs SUCCESS in 2 mins 6 secs.
```

Listagem 4 - Resultados dos casos de teste

6 Conclusão e trabalhos futuros

Neste trabalho, foi feito um estudo de *workflow* para sistemas web, a fim de viabilizar a geração de código para esse tipo de sistema. Com esse estudo foram escolhidos três *workflow patterns* documentados pela organização *Workflow Management Coalition*. A partir desse *patterns* criou se expressões idiomáticas revisadas por especialistas e que foram utilizadas para desenvolver um gerador de código. Para validar o gerador e as expressões

idiomáticas, foram criados testes automatizados fundamentados nas descrições dos *Workflows Patterns* onde cada caminho do *workflow* definia um caso de teste. Com isso mostrou que é sim possível gerar código de *SGW* web com uma abordagem semelhante ao *Scaffolding*.

Como trabalhos futuros, pretendemos criar *templates* e expressões idiomáticas para outras linguagens de programação além de *Java* e *TypeScript*, como *Python* e *Ruby*. Outro possível trabalho futuro é elaborar mais expressões idiomáticas a partir de diferentes *workflow patterns* tendo em vista a grande quantidade documentada, além de realizar experimentos para comparar o desenvolvimento de *workflow* manual, com *engines* e com o gerador de código aqui proposto.

7 Referências

BECK, Kent. Smalltalk Best Practice Patterns. Volume 1: Coding. Prentice Hall, Englewood Cliffs, NJ, 1997.

BUSCHMANN, Frank et al. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. 1996. Cited on, p. 346.

Camunda, (2019). Acesso em 15 de 05 de 2019, disponível em: https://camunda.com/products/bpmn-engine/

CRUZ, Tadeu. Workflow II: a tecnologia que revolucionou processos. Editora E-papers, 2004.

FA, HE Qing et al. RELATION BASED LIGHTWEIGHT WORKFLOW ENGINE [J]. Journal of Computer Research and Development, v. 2, 2001.

FIELDING, Roy T .; TAYLOR, Richard N. Estilos arquitetônicos e o design de arquiteturas de software baseadas em rede . Tese de Doutorado: University of California, Irvine, 2000.

Georgakopoulos, D., Amit, S., An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases, no. 3, 119-152, 1995.

GRYPHON, Robert L .; CLEMENTS, Michael R .; MAKAGON, Kira R. Linguagem de modelagem de fluxo de trabalho . Patente dos EUA n. 6.233.537, 15 de maio de 2001.

HERRINGTON, Jack. Code generation in action. Manning Publications Co., 2003.

JOHNSON, Ralph; VLISSIDES, John. Design patterns. Elements of Reusable Object-Oriented Software Addison-Wesley, Reading, 1995.

MAGNO, Danillo Goulart. Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD. 2015.

MYERS, Glenford J. et al. The art of software testing. Chichester: John Wiley & Sons, 2004.

PEREIRA, Luis Antônio M.; CASANOVA, Marco Antonio. Sistemas de gerência de workflows: Características, distribuição e exceções. PUC–Rio de Janeiro: Inf. MCC, 2003.

PRESSMAN, Roger S. Software engineering: a practitioner's approach. Palgrave Macmillan, 2005.

SCHMIDT, DOUGLAS et al. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. 1996.

THOM, Lucinéia Heloisa; IOCHPE, Cirano. Applying Block Activity Patterns in Workflow Modeling. In: ICEIS (3). 2006. p. 457-460.

VILAR, Rodrigo; OLIVEIRA, Delano; ALMEIDA, Hyggo. Rendering patterns for enterprise applications. In: Proceedings of the 20th European Conference on Pattern Languages of Programs. ACM, 2014. p. 33.

WIERINGA, Roel. Design science as nested problem solving. In: Proceedings of the 4th international conference on design science research in information systems and technology. ACM, 2009. p. 8.

Workflow Management Coalition. The Workflow Reference Model. http://www.wfmc.org Access: Jun, 2019

ZAHAVI, Ron; MOWBRAY, Thomas J. The essential CORBA: systems integration using distributed objects. Wiley, 1995.

APÊNDICE A - Catálogo de Expressões Idiomáticas

Listagem A.1 - Representação dos estados de *workflow* através de *String Enumerations* em Java.

```
enum EntityState {

  State1("State 1"), State2("State 2"), StateN("State N");

  private String description;

  EntityState(String description) {
    this.description = description;
  }

  public String getDescription() {
    return this.description;
  }
}
```

Listagem A.2 - Representação de uma Entidade que contém um campo para os estados do *workflow* através de uma classe Java.

```
public class Entity {
  private Integer id;
  private EntityState state;
}
```

Listagem A.3 - Representação de possíveis ações em um *workflow* através de *String Enumerations* em Java.

```
public enum EntityAction {
  Action1("Action 1"), Action2("Action 2"), ActionN("Action N");
  private String description;

  EntityAction(String description) {
    this.description = description;
  }

  public String getDescription() {
    return this.description;
  }
}
```

Listagem A.4 - Representação de uma Entidade que contém um campo transiente para representar possíveis ações em um *workflow* em uma classe Java.

```
public class Entity {
  private Integer id;
  private EntityState state;
  @Transient
  private List<EntityAction> availableActions;
}
```

Listagem A.5 - EntityActionDTO em Java.

```
public class EntityActionDTO {
  private EntityAction rel;
  private String description;
  private String method;

  public EntityActionDTO(EntityAction action, String method){
    this.rel = action.toString();
    this.description = action.getDescription();
    this.method = method;
  }
}
```

Listagem A.6 - EntityService em java.

```
public class EntityService {
  private Map<EntityState, List<Action>> transacao = new HashMap();

  @PostConstruct
  public void init() {
    private List<Action> actions = new ArrayList<>();
    actions.add(Action.action1);
    actions.add(Action.action2);
    actions.add(Action.actionN);
    transacao.put(EntityState.State1, actions);
  }

public List<Transition> getAvailableActions(EntityState state) {
  if (transacao.get(state) != null)
    return transacao.get(state);
  return new List<Action>();
}
protected Pedido postGet(Pedido pedido) {
  pedido.setAvailableTransitions(getAvailableTransitions(pedido.getStatusPedido()));
  return pedido;
}
```

Listagem A.7 - Controller aciona Service para povoar ações.

```
public class EntityControlle {
  @GetMapping("/{id}")
  public D get(@PathVariable Interge id) {
    return toDTO(getOneModel(id));
  }
}
```

Listagem A.8 - JSON de comunicação entre *Front-end* e *Back-end.*

```
GET /registers Response Body
[
  {
    "id":1,
    //other fields
    "status":“State 1”,
    "availableActions": {
      {
        "rel":"action1",
        "description" = "Action 1",
        "method": "POST"
      },
      {
        "rel":"action2",
        "description" = "Action 2",
        "method": "POST"
      },
      {
        "rel":"actionN",
        "description" = "Action N",
        "method": "POST"
      }
    }
  }, ...
]
```

Listagem A.9 - Representação para ações no *Front-end*.

```
<button (click)="sendAction(item.id, ‘action1’)">
  <span> Action 1 </span>
</button>
<button (click)="sendAction(item.id, ‘action2’)">
  <span> Action 2 </span>
</button>
<button (click)="sendAction(item.id, ‘actionN’)">
  <span> Action N </span>
</button>
```

Listagem A.10 - Função no *front-end* para envio de ação ao back-end.

```
sendAction(id, action) {
  this.POST(URL, id, { "entityAction": action}).subscribe(result => {
    this.notification.success();
    this.listItems();
  }, error => {
    this.notification.error('Error during ${entityAction}: ${error}');
  });
}
```

Listagem A.11 - JSON de envio de ação.

```
{ "action": "action1" }
```

Listagem A.12 - Interface para representar a lógica de cada ação a partir de cada estado do *workflow.*

```
public interface StateLogic {
  public void action1(Entity entity);
  public void action2(Entity entity);
  public void actionN(Entity entity);}
```

Listagem A.13 - Interface implementadas.

```
public class State1Logic extends Adapter implements Logic {
  public void action1(Entity entity){
    //regra de negócio
  }
  public void action2(Entity entity){
    //regra de negócio
  }
  public void actionN(Entity entity){
    //regra de negócio
  }
}
```

Listagem A.14 - Classe Adapter para diminuir quantidade de linhas em classe concretas em java.

```
public class Adapter implements Logic{
  public void action1(Entity entity){ … }
  public void action2(Entity entity){ … }
  public void actionN(Entity entity){ … }
}
```

Listagem A.15 - EntityService configura os StateLogics e executa a ação do *workflow*.

```
public class EntityService {

  private final Map<Enum, Logic> logics = new HashMap<Enum, EntityLogic>();

  @Autowired
  private State1Logic state1Logic;
  @Autowired
  private State2Logic state2Logic;

  @PostConstruct
  public void init(){
    logics.map(EntityState.State1, State1Logic);
    logics.map(EntityState.State2, State2Logic);
  }

  public void changeState(Integer id, EntityAction action){
    Entity entity = getEntityById(id);
    EntityState sourceState = entity.getState();
    action(task, sourceState, action);
  }

  public void action(Entity entity, EntityState sourceState, Action action) {
    Logic logic = logics.get(sourceState);
    try {
      Method method = logic.getClass().getMethod(action.toString(), entity.getClass());
      method.invoke(logic, entity);
    } catch (Exception e) {
      throw new BusinessException("Erro", e);
    }
  }
}
```

Listagem A.16 - Definição de uma Anotação para declarar que um usuário tem permissão para usar uma ação

```
@Documented
@Retention(RUNTIME)
@Target({ TYPE, METHOD })
public @interface UsersAllowed {
  String[] value()
}
```

```
@Documented
@Retention(RUNTIME)
@Target({ TYPE, METHOD })
public @interface RolesAllowed {
  String[] value()
}
```

Listagem A.17 - Definição de uma Anotação para representar que apenas um usuário pode executar uma ação.

```
@UsersAllowed("USERNAME or LOGIN USER")
public void action1(Entity entity) throws BusinessException {
  entity.setState(EntityState.action2);
  entityRepository.save(entity);
  return;}

@RolesAllowed("ROLE")
public void action2(Entity entity) throws BusinessException {
  entity.setState(EntityState.action3);
  entityRepository.save(entity);
  return;}
```

Listagem A.18 - Linhas de código para verificar pelo o *Username* se o usuário está apto a executar uma ação.

```
UsersAllowed usersallowed = method.getAnnotation(Allowed.class);
if (allowed != null && !isAllowed(this.getIdentificador(), usersallowed.value()))
  throw new BusinessException("Usuário não pode realizar essa ação");
```

Listagem A.19 - Linhas de código para verificar pelo o *Role* se o usuário está apto a executar uma ação.

```
RolesAllowed rolesAllowed = method.getAnnotation(Allowed.class);
String role = this.roleService.getOne(this.getCurrentIdRole());
if (allowed != null && !isAllowed(role, rolesallowed.value()))
  throw new BusinessException("Usuário não pode realizar essa ação");
```

Listagem A.20 - Método para auxiliar verificação se o usuário está apto a executar um ação.

```
private boolean isAllowed(String identificador, String[] identificadoresAutorizados) {
  for (String identificadorAutorizado: identificadoresAutorizados) {
    if (identificador.equals(identificadorAutorizado))
      return true;
  }
  return false;
}
```