

Geração de código escalável para o padrão de workflow *Deferred Distribution*¹

Matheus Guedes Crispim¹, Rodrigo de Almeida Vilar de Miranda¹

{matheus.guedes, rodrigovilar}@dcx.ufpb.br

Abstract. *Workflow Management Systems use rule engines to interpret high-level rules and provide functionality at runtime. However, this solution can represent a bottleneck for information systems that use them. Therefore, this work proposes the abstraction of workflow concepts through code generation. The main result of this work is a idiom proposal for the implementation of a Workflow pattern of the Resource patterns type, called Deferred Distribution. The proposed idiom represents a reference code for implementing the Deferred Distribution pattern in the technologies: Angular and Typescript for frontend; and Spring Boot and Java for backend - which can be used as the basis for a code generator. The idiom was analyzed qualitatively in order to assess its scalability.*

Resumo. *Os Sistemas Gerenciadores de Workflow utilizam motores de regras para interpretar regras de alto nível e prover funcionalidades em tempo de execução. Todavia essa solução pode representar um gargalo para sistemas de informações que as utilizem. Assim sendo, este trabalho propõe a abstração dos conceitos de workflow através da geração de código. O principal resultado deste trabalho é uma proposta de idioma para a implementação de um padrão de Workflow do tipo Resource patterns, chamado o Deferred Distribution. O idioma proposto representa um código de referência para implementação do padrão Deferred Distribution nas tecnologias: Angular e Typescript para frontend; e Spring Boot e Java para backend - que pode ser utilizado como base para um gerador de código. O idioma foi analisado qualitativamente com o objetivo de avaliar sua escalabilidade.*

¹Trabalho de conclusão de curso, sob orientação do professor Rodrigo de Almeida Vilar submetido ao Curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAIE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de BACHAREL EM SISTEMAS DE INFORMAÇÃO."

1. Introdução

Um dos principais desafios de empresas é a automação e melhoria de processos de negócios, sistematizando e otimizando as atividades referentes a objetivos corporativos, com o intuito de padronizar e também reduzir gargalos. Diante desse contexto, *workflow*, pode ser entendido como "fluxo de trabalho", mas também serve como base para a realização da reengenharia de negócios em organizações.

Workflow diz respeito a sequência de passos utilizados para a realização de um trabalho, segue padrões definidos por um conjunto de regras e procedimentos criteriosos, que são estabelecidos baseando-se em todas as atividades envolvidas em um fluxo, e também nas regras inerentes ao negócio. De um modo geral, pode ser definido como a sistematização do processo organizacional, parcialmente ou completamente.

As ferramentas de Sistemas de Gerenciamento de *Workflow* (SGW) possuem papel fundamental na organização e automatização do fluxo de trabalho, servindo também como uma abstração mais alta para implementação dessa funcionalidade. A partir de fluxos montados pelo usuário em sua interface, os SGW produzem em tempo de execução, funcionalidades (telas e lógica de negócio). No entanto, Mowbray (1995) aponta que ferramentas de SGW não prezam por métricas relacionadas a escalabilidade, pois geralmente o *software* produzido por essas ferramentas têm grande dependência de seu motor de execução, dificultando na maioria dos casos a manutenção, extensão e o desempenho de funcionalidades, gerando assim, gargalos.

Visto isso, esse trabalho levanta a seguinte questão: **É possível utilizar uma abstração para o padrão de *workflow* *Deferred distribution* de modo que a funcionalidade provida seja escalável?**

Nesse sentido, o presente trabalho tem como objetivo propor uma solução intermediária para geração de código baseando-se em padrões de *workflow*, de modo que o código gerado siga o aspecto proposto pela questão de trabalho. Sendo assim, é utilizado como objeto principal de viabilização, o padrão *Deferred Distribution* - explicado na seção 3.3.1. As implementações de código realizadas serão feitas nas tecnologias *Angular*, *Typescript*, *Spring boot* e *Java*.

Este trabalho está estruturado da seguinte maneira: a Seção 2 representa a fundamentação teórica, em que são explicados os conceitos base para o desenvolvimento do trabalho, sendo esses *workflow*, padrões de *workflow*, o padrão *Deferred Distribution*, e expressões idiomáticas e por fim o conceito de escalabilidade; na Seção 3 é discutida a metodologia do trabalho; na Seção 4 é discutido como foram desenvolvidas e analisadas as expressões idiomáticas propostas; na Seção 5, apresenta-se os resultados; e na Seção 6, é feita a conclusão e a discussão de trabalhos futuros.

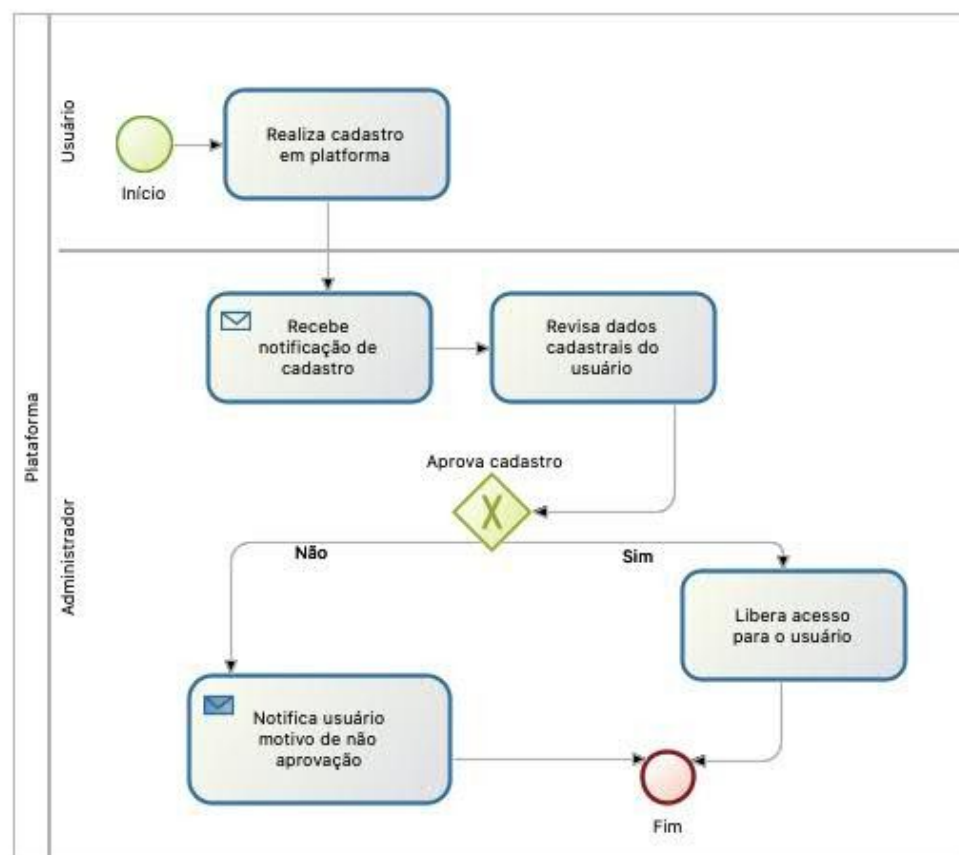
2. Fundamentação teórica

Segundo Pereira e Casanova (2003) um *workflow* é o que diz respeito a uma sequência de passos necessários para automatizar processos de negócio, parcialmente ou completamente.

De acordo com Cruz (2004) a constituição de um *workflow* é baseada em atividades, indivíduos, rotas e regras. Atividades são definidas como objetos ou entidades que necessitam da ação de um indivíduo para serem executadas. No que diz respeito aos indivíduos, esses são os agentes responsáveis por executarem alguma ação necessária dentro do *workflow*, podendo ser um agente humano, ou não, são definidos também como recursos. Rota é a conexão entre as ações ou objetos que fazem parte do *workflow*, dividem-se em caminhos paralelos, lineares ou circulares. Regras são definidas como o conjunto de princípios que devem ser atendidos para que seja possível passar de uma atividade para outra.

Para consolidar melhor o tema *workflow*, a figura abaixo exemplifica um caso em que um usuário tem de realizar o cadastro em uma plataforma online, mas para isso é necessário que seu cadastro seja autorizado por algum administrador no sistema.

Figura 1. Cadastro com aprovação de administrador



Fonte: O autor

No fluxo acima é possível identificar as partes constituintes de um *workflow*, as setas representam as **rotas** definidas, as **atividades** são especificada como as caixas de texto, os

indivíduos são representados por as raia e as **regras** são representadas por as premissas para a realização de alguma atividade.

2.1 Sistemas de gerenciamento de *workflow*

Sistemas de gerenciamento de *workflow* são definidos por Georgakopoulos et al. (1995) (*SGW*) como ferramentas que especificam, gerenciam, e auxiliam na execução de processos. Isso se torna funcional por meio de sua interface, onde é possível montar fluxos em alto nível, especificando todas as partes envolvidas em um. Além disso, as *SGW* também funcionam como ferramentas para desenvolvimento de software, geram códigos funcionais e prontos para serem executados, estes baseados em todo o fluxo montado na interface.

Sua execução é guiada com base em rotas estabelecidas, havendo sempre a influência de outras partes constituintes do fluxo, como as regras de negócio que devem ser atendidas e as ações que necessitam de um indivíduo para acontecer. É papel também de um *SGW*, garantir que as atividades serão executadas sempre na ordem correta, tentando minimizar ao máximo os erros que possam vir a ocorrer durante a execução do *workflow* estabelecido.

Nesse sentido, é válido salientar algumas das principais vantagens que um sistema de gerenciamento de *workflow* pode trazer, sendo essas as seguintes: redução de custos operacionais, em que os custos que estão intrinsecamente ligados às atividades, são reduzidos; aumento da eficiência e eficácia nos processos; redução de erros humanos; geração de artefatos de forma automática e digital; automação e melhoria contínua de processos.

Todavia, como apontado por Mowbray (1995), existem diversos problemas acerca de artefatos gerados por as ferramentas de *SGW*. Em muitos casos os softwares gerados não prezam por fatores essenciais para um bom funcionamento, deixando de lado aspectos de grande relevância, como, por exemplo, aspectos relacionados a escalabilidade, pois a maior parte das ferramentas que geram esses códigos, causam dependência com o motor de execução do *workflow*. Assim, esse tema é de extrema relevância para o trabalho, pois nele principalmente, é baseada a problemática que se busca resolver.

Um exemplo de *SGW* é o *BonitaSoft*², que foi utilizado para a criação dos diagramas de *workflow* exibidos neste trabalho. É possível utilizar também esta ferramenta para o desenvolvimento de aplicações *web*, onde o código gerado na linguagem *Java* é executado apenas no portal do *Bonita*, gerando assim, a relação de dependência entre código e o *SGW*.

2.2 Padrões de Workflow

Enquanto os *SGW* são responsáveis pela organização e gerenciamento de fluxos, os padrões de *workflow* buscam tratar de soluções elaboradas de forma genérica, com o foco em problemas comuns de negócios. Esses padrões independem de linguagem ou tecnologia, e

² <https://www.bonitasoft.com/>

buscam de um modo geral padronizar e documentar as soluções para os problemas mais recorrentes referentes a regras de negócio.

A *Workflow Management Coalition (WfMC)* (2019), organização responsável por catalogar e classificar os *padrões de workflow*, os divide em quatro tipos principais, sendo estes os seguintes: *Control-flow* (Padrões de fluxos de controle), *Resource Patterns* (Padrões de fluxo de recurso), *Data Patterns* (Padrões de fluxos de dados) e *Exception Handling Patterns* (Padrões de fluxos de exceções).

O tipo *Control-flow* classifica os padrões referentes a fluxos de controle, que visam principalmente regras que referem-se às rotas entre atividades. *Resource patterns* classifica os padrões que tratam da alocação de recursos para tarefas ou indivíduos. O tipo *Data Patterns* categoriza os padrões que visam a comunicação entre os componentes de um fluxo. E *Exception Handling Patterns* categoriza os padrões referentes a tratamento de erros e exceções, tratando de possíveis erros que podem acontecer durante a execução do *workflow*.

No próximo tópico é discutido um padrão da família *Resource Patterns*, que foi escolhidos para estudo e implementação neste trabalho.

2.2.1 Deferred Distribution

O *Deferred Distribution* é um padrão classificado como *Resource Patterns*, cujo seu propósito principal é a habilidade de identificar e especificar em tempo de execução o indivíduo que irá executar uma determinada atividade dentro do *workflow*.

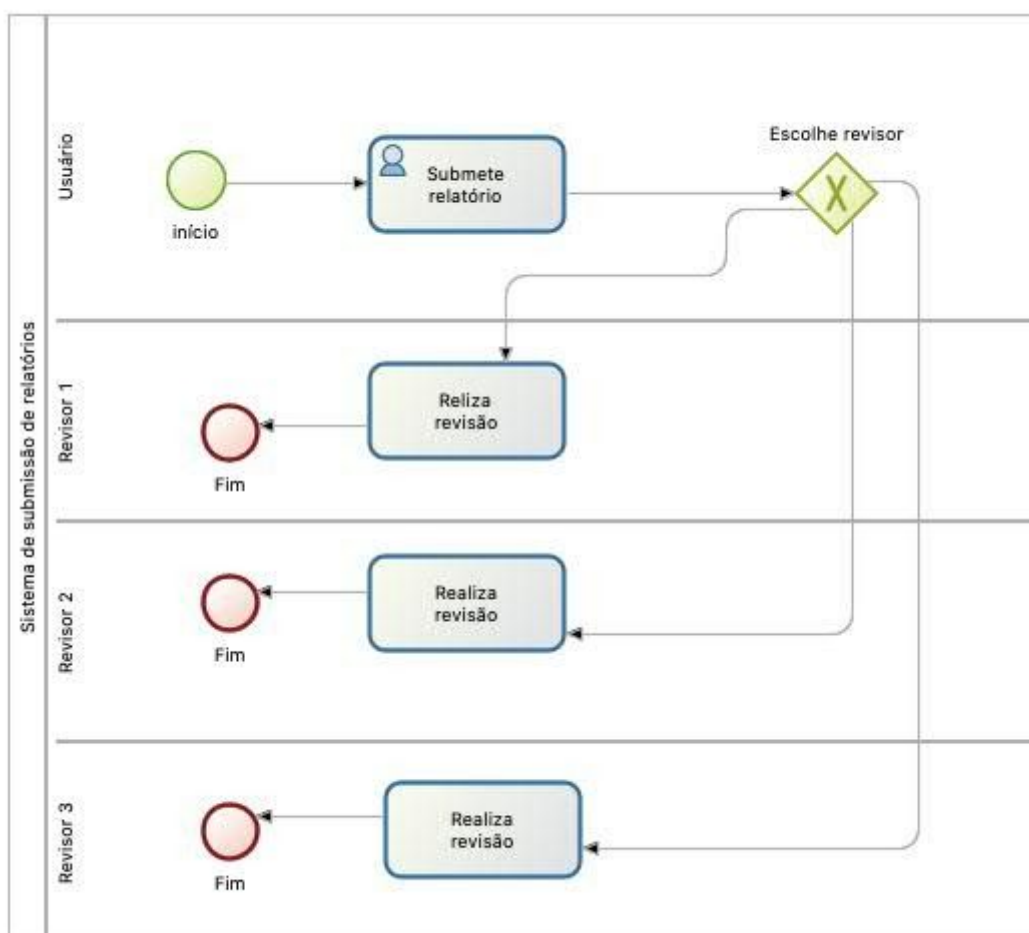
Portanto, esse padrão tem utilidade quando existem atividades que podem ser distribuídas para indivíduos de uma determinada categoria, ou até mesmo quando a atividade pode ser executada por qualquer um. Sendo assim, é feito em tempo de execução uma verificação acerca dos indivíduos disponíveis, onde a atividade é alocada para o que for selecionado.

Um problema em que pode ser aplicado este padrão, é a situação na qual um usuário tem que selecionar um revisor para seu relatório. Se o *Deferred Distribution* for aplicado corretamente, é possível que a pessoa escolha em tempo de execução o revisor que desejar. A Figura 2 mostra o *workflow* para esta situação.

2.3 Geração de código

No desenvolvimento de *software*, um dos principais problemas é o que se refere a atividades repetitivas, em que muitas vezes tem de se criar diversas funcionalidades que seguem o mesmo padrão, realizar configurações, definição de estruturas de acesso ao banco, etc. São etapas do desenvolvimento que na maioria dos casos demandam tempo e outros recursos, mas que poderiam ser solucionadas rapidamente se houvesse ferramentas eficientes para a automatização de tais.

Figura 2. Deferred Distribution



Fonte: O autor

Mediante isso, existem geradores de código, que tratam-se de ferramentas responsáveis pela configuração e geração inicial de código referente a alguma funcionalidade. Geradores podem ser utilizados sem que haja a necessidade de repetir o trabalho exaustivo de estar codificando funcionalidades monótonas como as de *CRUD*³, ou realizando configurações básicas em todo o projeto. Esse processo é automatizado e viabilizado com base em metadados e *templates*, que são a base para a geração do código. Portanto, é possível que o desenvolvedor execute o projeto em questão de minutos, sem a necessidade de um grande esforço (Herrington, 2014).

Um exemplo de gerador de código é a ferramenta *Ruby on Rails*⁴, que gera código para aplicações *web* na linguagem *Ruby*, baseando-se em meta dados.

³ Acrônimo do termo "*CREATE, READ, UPDATE, DELETE*", que diz respeito a 4 operações básicas que podem se realizar em um banco de dados.

⁴ <https://rubyonrails.org/>

2.4 Expressões idiomática

Visto que padrões de *workflow* documentam como problemas são resolvidos dentro de um fluxo de trabalho, expressões idiomáticas dizem respeito a padrões de baixo nível, próprios de uma linguagem de programação.

Em uma expressão idiomática, são descritos aspectos implementados de um problema, como, por exemplo, as regras de negócio inerentes a determinada funcionalidade. Para obter uma expressão idiomática a partir dessa funcionalidade, é necessário fazer uma generalização do código escrito, onde são parametrizadas as partes peculiares da implementação, como nomes de classes, funções, variáveis, trechos que de modo geral não têm impacto nas regras de negócio envolvidas.

Desse modo, pode ser obtido um modelo genérico do código, onde o objetivo da funcionalidade é mantido, mas partes não relevantes de forma operacional são substituídas. Podem assim ser facilmente replicadas, manualmente ou automaticamente (em geradores de código), em sistemas que necessitem de uma funcionalidade de mesmo objetivo. Sendo assim, Buschmann (1996) define uma expressão idiomática como a capacidade de descrever características peculiares de componentes ou o relacionamento entre eles com os recursos de uma determinada linguagem.

2.5 Escalabilidade

Segundo Almeida (2014), escalabilidade não é um termo encontrado facilmente em dicionários de nossa língua, mas que em um sentido geral pode ser traduzido como a qualidade de crescer ou escalar algo. No que diz respeito a engenharia de *software*, não é diferente, sendo assim, esse termo é comumente aplicado em diversas vertentes da área, desde aspectos relacionados à qualidade do código, até aspectos que dizem respeito a operabilidade de um sistema.

Tendo em vista que não há uma definição ou fórmula padrão para medir a escalabilidade de um código ou *software*, Duboc et al. (2007) buscou desenvolver técnicas para medir e prever esse aspecto, onde ficou entendido que existe mais de uma dimensão para tal. Sendo assim, é perceptível que esse é um termo que pode ser aplicado a quase qualquer característica de um *software*, mas que para medir o que é escalável, é necessário definir bem o contexto ao qual se busca investigar.

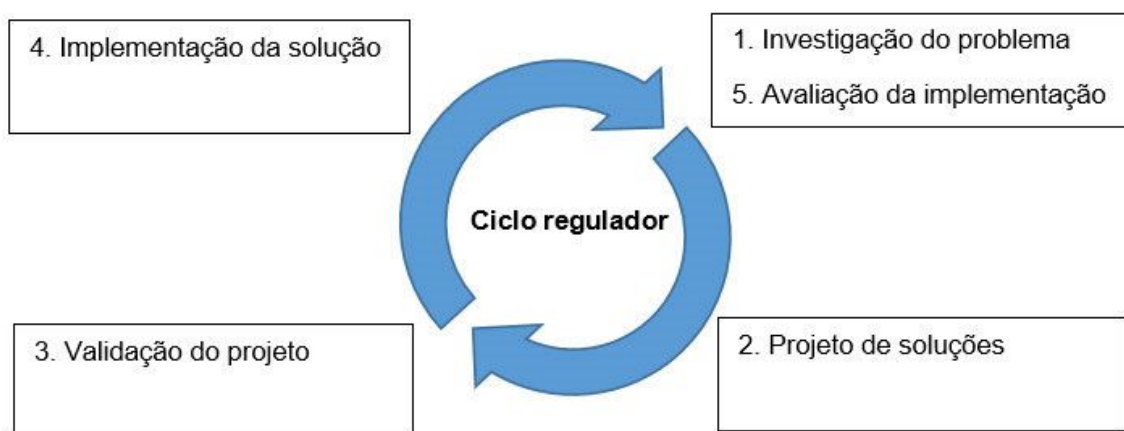
Um exemplo de métrica de escalabilidade é a quantidade de requisições que um sistema é capaz de atender em um determinado período de tempo. Tendo essa em vista, é possível projetar no sistema a capacidade de receber muitas requisições, sem degradar o seu tempo de resposta. A escalabilidade também pode ser observada em aspectos de um *software*, como, por exemplo, na possibilidade de estender ou implementar funcionalidades em uma aplicação, por meio de seu código fonte. Desse modo, percebe-se que escalabilidade é um tema de grande importância no contexto arquitetural e funcional de sistemas que tendem a evoluir constantemente.

3. Metodologia

Com o propósito de investigar a possibilidade de abstrair o código do padrão de Workflow Deferred distribution, mantendo o aspecto de escalabilidade, foi adotada a metodologia *Design Science Research* (DSR). Segundo Dresch et al. (2015), trata-se de um método que serve como base para a realização de uma pesquisa quando o objetivo do estudo é um artefato ou prescrição.

Para o desenvolvimento dos artefatos propostos pelo trabalho, se deu a utilização do ciclo regulador de Wieringa, um ciclo de cinco etapas que visa regular o desenvolvimento dos objetos produzidos na pesquisa. Inicialmente é feito uma investigação acerca do problema abordado. Em seguida são propostas soluções para esse problema. Dando sequência, é realizada uma avaliação das soluções propostas. Seguindo, é realizada a implementação das soluções e por fim é feita uma avaliação da implementação dos artefatos. Finalizando, esse mesmo ciclo pode ser repetido.

Figura 3. Ciclo regulador de Wieringa



Fonte: <[ResearchGate](#)>

A metodologia do trabalho tendo como base o ciclo de Wieringa, é estruturada da seguinte forma:

1. Investigação do problema no contexto deste trabalho;
2. Implementação do padrão de *workflow Deferred distribution*;
3. Análise por especialistas acerca da solução implementada;
4. Transcrição do padrão implementado em expressões idiomáticas;
5. Análise de escalabilidade acerca dos idiomas criados.

3.1 Investigação do problema

A etapa inicial de investigação foi iniciada com uma pesquisa acerca de trabalhos relacionados.

Sendo assim, foi utilizada a ferramenta *Google Scholar*⁵, onde os termos pesquisados foram "*Workflow*", "*Workflow patterns*", "*Geração de código*", "*Expressões idiomáticas*" e "*Escalabilidade*". Desse modo, foram selecionados os trabalhos relacionados com maior relevância para revisão bibliográfica, onde a métrica para seleção foi o maior grau de relação com a temática deste trabalho. A divisão da revisão se deu da seguinte forma:

1. Estudo sobre o que é e como funciona *workflow*;
2. Investigação acerca de padrões de *workflow*;
3. Investigação sobre expressões idiomáticas e geração de código;
4. Investigação acerca da métrica escalabilidade;
5. Busca por ferramentas que provêm uma base para trabalhar na implementação de padrões de *workflow*;

Os resultados para esta investigação são exibidos na seção 2, onde são apresentadas os temas bases para a realização deste trabalho.

3.2 Implementação de um padrão de *workflow*

Após a investigação inicial, foi selecionado um padrão de *workflow* para dar sequência ao trabalho. O padrão escolhido foi o *Deferred Distribution*, onde a escolha se deu com base na sua baixa quantidade de regras de negócios. Desse modo, o site *WfMC* serviu como fonte para melhor entendimento sobre esse.

Seguindo, foi realizado o mapeamento acerca de problemas comuns que podem ser resolvidos com o *Deferred Distribution*. A implementação do padrão escolhido foi feita utilizando o exemplo apresentado no *WfMC*, por nesse ser abordada a problemática central resolvida.

Para implementação do padrão foram utilizadas as linguagens *Java* e *Typescript*, com implementações nos *frameworks Angular* e *Spring boot*. *Angular* aplicado nas telas e *Spring boot* na lógica operacional. A escolha de tais tecnologias se deve ao fato de que neste trabalho é utilizado uma estrutura⁶ de código fornecida por Honorato (2019), que também estudou a implementação de idiomas para *workflow* e utiliza as mesmas tecnologias escolhidas neste trabalho.

A estrutura fornecida opera com o estilo arquitetural: *Representational State Transfer (REST)* (Fielding, 2000), que opera entre cliente e servidor. A estrutura fornecida conta com um Sistema de autenticação de usuários e também uma estrutura de *dashboard*, que foi adaptada para implementação do padrão *Deferred Distribution*.

⁵ <https://scholar.google.com.br/>

⁶ <https://github.com/JohnnyHonorato/workflow-patterns-back>
<https://github.com/JohnnyHonorato/workflow-patterns-front>

3.3 Análise por especialistas

A etapa de análise por especialistas teve início com a disponibilização do código na plataforma Github⁷, que além da hospedagem, forneceu também funcionalidades para versionamento, comentários e controle de mudanças no código.

Feito isso, um engenheiro de *software* com mais de 10 anos de experiência, profissional do laboratório *Virtus*⁸, deu início a análise. Feedbacks foram passados com o objetivo de melhorar a qualidade e maturidade do código, o tornando assim mais propício para a transcrição em templates genéricos que podem ser utilizados por geradores de código.

Os principais feedbacks fornecidos foram acerca da melhor generalização do código, como por exemplo nomes mais genéricos para variáveis, funções e classes. Além disso foi também de contribuição do especialista, o modelo para de parametrização código para transcrição em expressões idiomáticas.

3.4 Transcrição do padrão implementado em expressões idiomáticas

Após a análise de especialistas, se deu início ao passo de transcrição do padrão implementado. Para a consolidação desta etapa, foi necessário parametrizar todo código implementado para o padrão *Deferred Distribution*. Sendo assim, todo o texto considerado generalizável dentro do código foi parametrizado e transformado em idiomas que, futuramente, podem ser utilizados em templates genéricos para gerar código automaticamente.

3.5 Análise de escalabilidade

A fim de analisar a qualidade do código que se pode gerar com templates baseados em *workflow*, foi realizada uma etapa de estudos qualitativos acerca do código. O objetivo de tal foi testar e comprovar se de fato é possível gerar um código que responda ao aspecto escalabilidade. Sendo assim, esta etapa foi dividida da seguinte forma:

1. Investigação acerca do aspecto escalabilidade;
2. Definição das métricas para análise;
3. Análise das implementações com base nas métricas escolhidas;

Por fim, os resultados desta análise são apresentados e discutidos nas próximas seções deste trabalho.

4. Resultados

Visto que é possível gerar código baseando-se em padrões de *workflow*, foram implementadas funcionalidades de um sistema real, onde este serviu como base para toda a elaboração e desenvolvimento de templates de expressões idiomáticas que se baseiam no

⁷ <https://github.com/MatheusCrispim/workflow-patterns-back>
<https://github.com/MatheusCrispim/workflow-patterns-front>

⁸ <https://www.virtus.ufcg.edu.br/>

padrão *Deferred Distribution*. Por fim, pôde ser realizada uma avaliação acerca do aspecto escalabilidade nos códigos implementados. Portanto, são mostrados e discutidos a seguir, os resultados alcançados com a realização deste trabalho.

4.1 Implementação manual do padrão *Deferred Distribution*

Sendo feito o estudo acerca do padrão *Deferred Distribution*, a implementação de uma solução para o padrão foi realizada, sendo essa o sistema Tarefas educacionais. Ao fim dessa implementação, que é abordada de maneira mais abrangente na seção 4.2.1, foram obtidos os seguintes resultados.

4.1.1 Funcionalidades reais de um sistema

São mostradas a seguir as telas com funcionalidades que foram implementadas na estrutura fornecida.

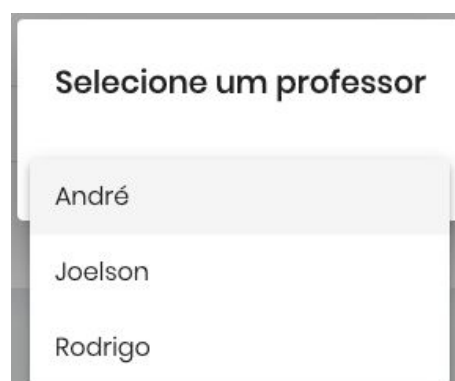
Figura 4. Componente de apresentações de atividades

Id	Date	Atividade	Ações
3	10.10.1000	Revisar Artigo	   <button>Atribuir Professor</button>
2	10.10.1000	Dar Nota de Artigo	   <button>Atribuir Professor</button>

Fonte: O autor

Como pode-se observar na imagem acima, é através deste componente que podem ser visualizadas as atividades cadastradas no sistema, há um botão de título "Atribuir Professor", que tem como funcionalidade a capacidade de atribuir aquela atividade a um usuário que for selecionado na tela mostrada a seguir:

Figura 5. Componente de seleção de usuários



Fonte: O autor

Nesta janela que aparece ao ser clicado o botão "Atribuir Professor", é possível selecionar o usuário ao qual a atividade será atribuída, ao ser feito, é enviada uma requisição para o back-end onde é realizada a atribuição daquele usuário a atividade. Foram realizados testes manuais nesta funcionalidade, com finalidade de ser comprovar o seu correto funcionamento, onde todos os testes, também realizados por o especialista, passaram. Desse modo, a validação da implementação manual do padrão *Deferred Distribution* foi finalizada.

4.2 Desenvolvimento de Expressões idiomáticas

Nesta seção são discutidos os passos necessários para a implementação das expressões idiomáticas que se encontram na seção de Apêndice.

4.2.1 *Deferred Distribution*

O padrão *Deferred Distribution* oferece a capacidade de atribuir dentro de um *workflow*, um indivíduo para uma instância de atividade a ser realizada. Sendo assim, para a implementação desta funcionalidade foi necessário inicialmente estudar a estrutura fornecida por Honorato (2019). No estudo realizado, foram analisados aspectos arquiteturais e funcionais acerca de como poderia ser realizada tal implementação.

A base principal para o estudo foi a análise sobre como foram codificados outros padrões de *workflow*. Feito isso, tinha-se o conhecimento necessário para codificar o padrão escolhido. A partir daí, seguindo a metodologia também abordada por Honorato (2019), foram tomadas decisões acerca da implementação do padrão *Deferred Distribution*. Os questionamentos para tomadas de decisões estão listados a seguir:

1. Como o *back-end* atribui um usuário para realizar uma atividade?
2. Como o *back-end* identifica e retorna as atividades atribuídas a um usuário?
3. Como o *front-end* atribui um usuário para realizar uma atividade?
4. Como o *front-end* deve mostrar as atividades atribuídas a um usuário?

Resposta para o **item 1**: O *back-end* deve realizar a atribuição de usuário a atividades por meio de identificadores para os mesmos. Este mapeamento é realizado através do *front-end* e recebido pelo *back-end* via requisição *REST*.

Resposta para o **item 2**: O *back-end* deve recuperar por meio da sessão de autenticação do usuário, o autor da sessão, feito isso é possível identificar as atividades que pertencem ao mesmo e retornar por meio de uma resposta *REST* para o *front-end*.

Resposta para o **item 3**: O *front-end* deve mostrar uma tabela de listagem de atividades, onde para cada atividade listada, é possível clicar em um botão correspondente à mesma. Feito isso, é exibida uma modal de seleção, nesta estão listados os usuários disponíveis no sistema para realização da atividade. Quando clicado, o frontend identifica a

atividade e o usuário que lhe está sendo atribuído, por fim realiza uma requisição *REST* no *back-end* enviando os identificadores.

Resposta para o **item 4**: O *front-end* deve realizar uma requisição *REST* em um *endpoint* específico que é responsável por retornar todas as atividades atribuídas ao usuário que está logado, quando retornadas, essas são listadas em uma tabela.

Tendo as respostas para os questionamentos em vista, foi dado início a implementação do código, onde se aplicou uma arquitetura semelhante à abordado por Honorato (2019) em suas implementações de padrões de *workflow*. Buscou-se implementar o padrão *Deferred Distribution* utilizando padrões de projeto de *software*, seguindo como base também o modelo de camadas.

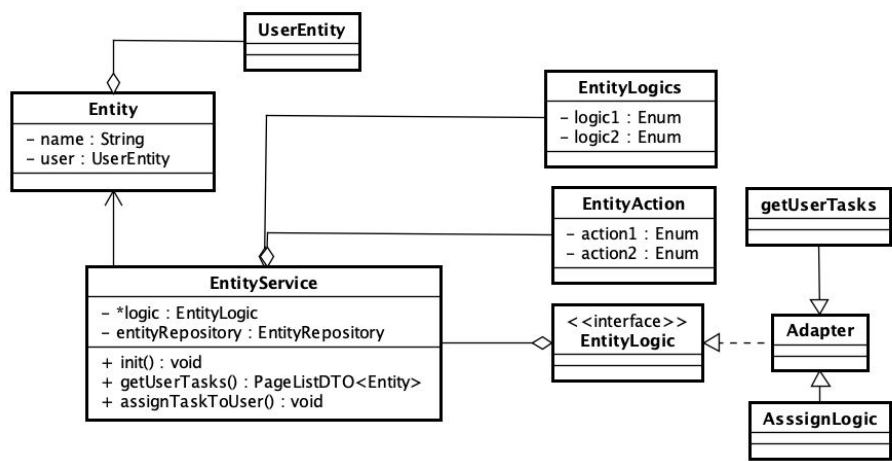
O código foi dividido principalmente em *Controllers*, *Services*, *Entities*, *Dtos*, *Repositories* e *Enumerations*. Na camada de *Controllers* são implementados os pontos de entrada dos dados, onde estes são recebidos em formato de *Dtos*. Quando recebidos pelos *Controllers*, os *Dtos* são validados e convertidos para *Entities*. Feito isso, são enviados para os *Services*, esses são responsáveis por conter as regras de negócios do sistema. Quando processados pelo *Services*, as *Entities* são passadas para *Repositories* e por fim são persistidas em um banco de dados. Os *Enumerations* dizem respeito a uma forma de criar constantes, que podem ser utilizadas no código. O *front-end* é estruturado principalmente baseando-se em módulos.

Visto que grande parte da complexidade deste trabalho está contida nas regras operacionais do padrão, foi dado maior ênfase na estruturação do código *back-end*. Sendo assim, para garantir a melhor manutenibilidade e extensibilidade do código, se utilizou uma estrutura de encapsulamento para as lógicas envolvidas.

Para isto, foi criada uma interface *TaskLogic* (Listagem A.3) onde nesta estão as assinaturas das ações que podem ser realizadas para a entidade *Task*. Esta interface tem uma implementação padrão em um *Adapter* (Listagem A.4). O objetivo dessa implementação, é para que uma classe concreta que implementa a interface *TaskLogic*, não tenha a obrigação de implementar todos os métodos listados na interface.

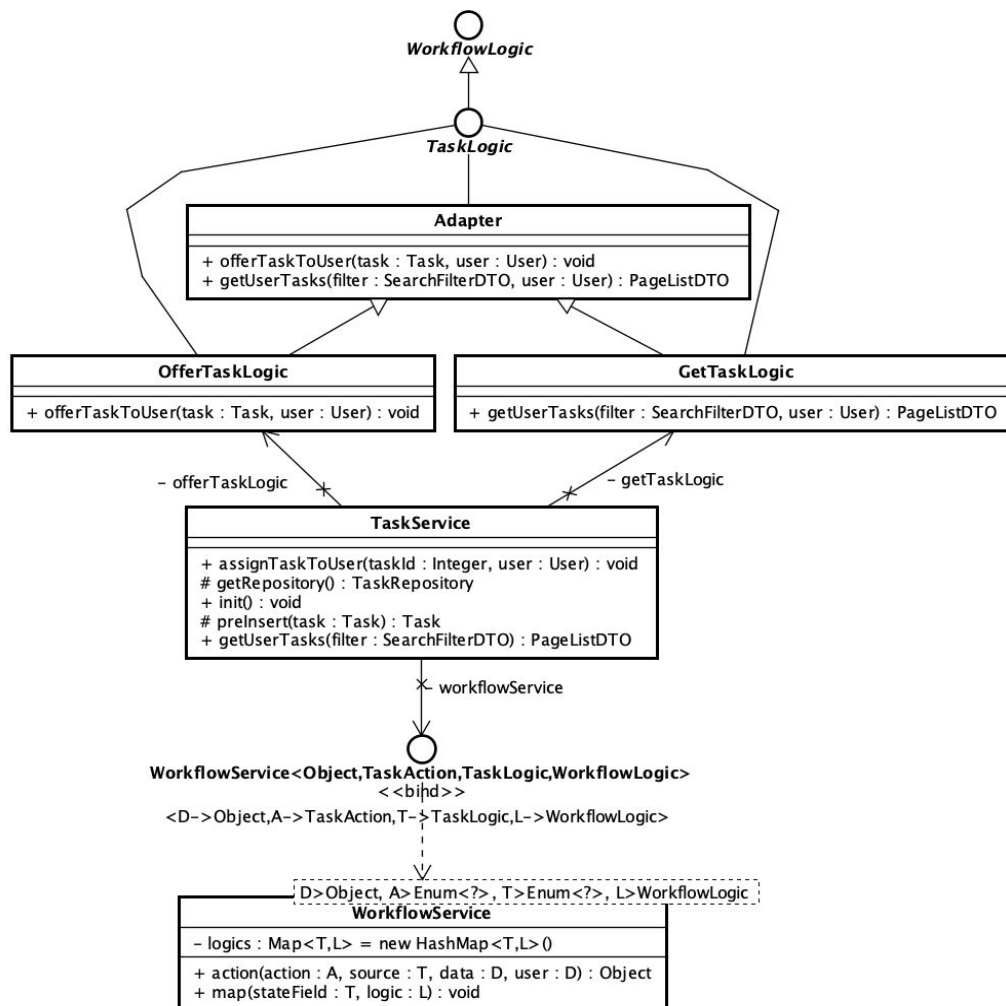
Feito isso, foram criados os *Enums* *TaskAction* (Listagem A.7) e *TaskLogic* (Listagem A.8) com a finalidade de se mapear todas as lógicas que implementam a interface *TaskLogic*, deixando assim de fácil implementação a criação de uma nova lógica concreta para essa interface. Todo o gerenciamento de tais implementações, se dá por meio do *TaskService* (Listagem A.1), que conta com uma funcionalidade de mapeamento, que se baseia nos *Enums* criados. Esse serviço pode ser injetado em um outro e utilizado, mas para que funcione corretamente, é necessário mapear as lógicas que se pretende utilizar, como se observa na função *init* do *TaskService* (Listagem A.1). Para ilustrar melhor, abaixo são exibidas as modelagens para o padrão *Deferred Distribution*, e a de código estático implementado :

Figura 6. UML da expressão idiomática *Deferred Distribution*



Fonte: O autor

Figura 7. UML do código implementado



Fonte: O autor

4.2.2 Expressões idiomáticas

Ao fim da implementação do código estático, se deu início ao processo de transcrição da solução implementada em expressões idiomáticas, o diagrama utilizada para identificação dos componentes transcritos é apresentado na figura 6.

A transcrição foi feita substituindo pontos generalizáveis do código por expressões genéricas que indicam ao gerador o que pode ser substituído dentro do *template*. Gerando por fim, um novo código.

Sendo assim, a seguir é mostrado uma função estática em *Java* e logo após a expressão idiomática que corresponde a função.

```
public void assignTaskToUser(Integer taskId, User user) throws BusinessException {
    Task task = getOne(taskId);
    workflowService.action(TaskAction.offerTaskToUser, TaskLogics.offerTask, task, user);
}
```

Listagem 1. Exemplo de código estático na linguagem Java.

```
public void assign{{entity}}ToUser(Integer {{entityVar}}Id, User user) throws BusinessException {
    {{entity}} {{entityVar}} = getOne({{entityVar}}Id);
    workflowService.action({{entity}}action.assign{{entity}}ToUser,
    {{entity}}Logic.assign{{entity}} {{entityVar}}, user);
}
```

Listagem 2. Exemplo de template na linguagem Java.

Como pode-se observar na imagem acima, é possível notar trechos de códigos onde as características não operacionais foram substituídas por notações, assim geradores que trabalham com essa linguagem podem utilizar o template implementado para gerar um novo código. Desse modo, desenvolvedores não necessitam implementar funcionalidades que normalmente tendem a ser repetitivas, e exaustivas, podendo assim ser rapidamente implementadas com templates de expressões idiomáticas e o auxílio de um gerador de código.

4.3 Avaliação de escalabilidade para as expressões idiomáticas criadas

Visto que escalabilidade é um termo amplo e complexo de ser medido, esta etapa teve início com a definição de um escopo exato para a extração de métricas. Para a definição de tal, foi necessário fazer o seguinte questionamento: Em quais aspectos de um *software*, pode ser medida a escalabilidade?. Feito isso, foi possível mapear um contexto e métricas para análise.

O contexto escolhido para trabalho está relacionado com a área de engenharia de *software*. Este escolhido por ter maior relação com o objetivo do presente trabalho. A partir disso, foi possível mapear métricas presentes nesse contexto. As métricas escolhidas se baseiam na problemática abordada por Mowbray (1995), ao apontar que na maioria dos casos SGW inibem a extensibilidade e portabilidade das aplicações, gerando assim, gargalos. Desse modo, as métricas escolhidas para análise foram extensibilidade e portabilidade. Com a definição de tais, foi possível realizar análises acerca do idioma implementado.

Para verificar a existência do aspecto escalabilidade de acordo com a métrica extensibilidade, foi inicialmente realizada a análise do código e de dos diagramas de classe exibidos nas figuras 6 e 7. Nestes pode se perceber que o código é dividido em camadas, sendo as principais *Controllers*, *Services* e *Repositories*. Desse modo, cada uma dessas têm uma responsabilidade, o que faz com que caso seja realizada alguma ação de extensão, como implementação de uma nova funcionalidade, ou até mesmo a modificação de uma já existente. Esta vai ser feita de forma mais direta, seguindo o padrão de responsabilidades do código. Um exemplo prático, pode ser a implementação de uma nova regra de negócio para

o cadastro de atividades, para que seja implementado um nova regra de negócio, o desenvolvedor só necessita realizar as implementações necessárias na camada *Service* (que trata de tais regras), sem que seja necessário se preocupar com as demais camadas.

Outro ponto em que pode ser observado o aspecto escalabilidade de acordo com a métrica extensibilidade, é no acoplamento de lógicas feito pela interface *TaskLogic* (Listagem A.3) e pela classe *Adapter* (Listagem A.4). Desse modo, caso o desenvolvedor deseje criar alguma nova lógica para utilizar no *TaskService* (Listagem A.1), não é necessário que ele modifique as lógicas já existentes no sistema, basta adicionar uma assinatura na interface *TaskLogic* (Listagem A.3), fazer uma implementação padrão no *Adapter* (Listagem A.4) e criar uma classe concreta que implementa a interface e estende o *adapter*, desse modo, diversas lógicas podem ser adicionadas facilmente ao *Service*, sem que haja a necessidade de modificar outras já existentes, facilitando assim a possibilidade de se estender o sistema e até mesmo de modificar lógicas já existentes.

A métrica portabilidade pode ser observada na não dependência entre código e *engine*. Pois visto que as SGWs causam tal problema, por o código na maioria dos casos não ser executado fora da ferramenta, o idioma implementado possibilita que o desenvolvedor tenha acesso ao código diretamente em linguagem nativa, sem a necessidade de uma engine intermediária. Isso permite com que seja possível realizar modificações no código, e também que esse seja executado no ambiente em que se desejar, podendo ser facilmente portado de uma arquitetura para outra.

Desse modo é possível perceber que pode ser gerado um código escalável de acordo com as métricas extensibilidade e portabilidade, utilizando como base as expressões idiomáticas apresentadas neste trabalho.

6. Conclusão e trabalhos futuros

De um modo geral, foi realizado neste trabalho uma análise acerca de expressões idiomáticas baseadas no padrão de *workflow Deferred Distribution*. sendo possível a partir destes implementar uma solução para o mesmo. Com base na solução implementada, foi possível criar expressões idiomáticas, as quais podem ser utilizados para o projeto de geradores código.

Dando sequência, foi realizada uma análise qualitativa acerca dos idiomas propostos, se baseando nas métricas de extensibilidade e portabilidade de um *software*, englobadas por o aspecto escalabilidade. Esta etapa teve o intuito de analisar como o aspecto discutido está presente nas expressões idiomáticas.

Sendo assim, ao final da realização desta etapa, foi possível obter a respostas para a questão central levantada neste trabalho. Esta, a seguinte: **É possível utilizar uma abstração para o padrão de *workflow Deferred distribution* de modo que a funcionalidade provida seja escalável?** A resposta obtida a partir dos estudos e análises realizadas, foi a seguinte: Sim, é possível utilizar uma abstração para o padrão de *workflow*

Deferred distribution, de modo que o código gerado seja escalável, pois o idioma pode ser implementado em linguagem nativa Java, sem o uso de motores de workflow que tornam o código dependente e pode ser um gargalo. Além disso, ao se ter a solução em Java puro e bem estruturado, é possível implementar facilmente novas funcionalidades, e também modificar outras já existentes.

Para trabalhos futuros pretende-se explorar mais a temática acerca de expressões idiomáticas e geração de código, onde serão utilizados outros padrões de *workflow* catalogadas por o *WfMC*. Também pode ser abordado a criação de idiomas para *Deferred distribution* em outras tecnologias, tais como *React Js*, *Node*, *PHP*.

7. Referências

ALMEIDA, Sandra. (2014). Dossiê Escalabilidade. São Paulo: Portal de Periódicos Mackenzie, Disponível em: <http://editorarevistas.mackenzie.br/index.php/tint/article/view/7751/5170>. Acesso em: 15 de Fevereiro de 2020.

Dresch, A.; Lacerda, D. P.; Antunes Júnior, José Antônio Valle (2015). Design science research: método de pesquisa para avanço da ciência e tecnologia. Bookman Editora.

Duboc, L., Rosenblum, D., e Wicks, T. (2007). A framework for characterization and analysis of software system scalability. In ESEC-FSE 07: Proc. of the 6th joint meeting of the European software engineering conference.

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.

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

Honorato, M., Johnny. (2019). Geração De Código de Sistemas web com base em padrões de workflow. Disponível em: <https://si.dcx.ufpb.br/wp-content/uploads/2015/12/JOHNNY-HONORATO.pdf>. Acesso em: 15 de Fevereiro de 2020.

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

Wieringa, R. J. (2014). Design Science Methodology for Information Systems and Software Engineering. Springer-Verlag Berlin Heidelberg, 1 edition.

Workflow Management Coalition. The Workflow Reference Model. Disponível Em <<http://www.wfmc.org>>. Acesso em: 4 dez. de 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 - Expressão idiomática de um *Controller* para o padrão *Deferred Distribution* baseada no *TaskController*

```
public class {{entity}}Controller extends CrudBaseController<Task, Integer, {{entity}}DTO> {
    @Autowired
    private {{entity}}Service service;

    @Override
    protected {{entity}}Service getService() {
        return service;
    }

    @Override
    public ResponseEntity<{{entity}}DTO> updatePartial(HttpServletRequest request,
        @PathVariable Integer id, @RequestBody {{entity}}DTO modelDTO) throws Exception {

        Task model = toModel(modelDTO);
        Task dbModel = getService().getOne(id);
        NullAwareBeanUtils.getInstance().copyProperties(dbModel, model);
        getService().assign{{entity}}ToUser(id, model.getAssignedUser());
        return ok(modelDTO);
    }

    public ResponseEntity<ModelDTO> search(HttpServletRequest request, @RequestParam("filter")
        String filterJson) throws BusinessException {
        SearchFilterDTO filter = JSonUtil.fromJson(filterJson, SearchFilterDTO.class);
        filter = customSearchFilter(request, filter);
        PageListDTO response = getService().getUser{{entity}}(filter);
        response.setItems(toSearchListDTO(response.getItems()));
        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}
```

Listagem A.2 - Expressão idiomática de um *Service* para o padrão *Deferred Distribution* baseada no *TaskService*

```
public class {{entity}}Service extends CrudService<{{entity}}, Integer> {

    @Autowired
    private {{entity}}Repository repository;
    @Autowired
    private UserService userService;
    @Autowired
    private WorkflowService<Object, {{entity}}Action, {{entity}}Logic, WorkflowLogic>
        workflowService;

    @Autowired
    private Assign{{entity}}Logic assign{{entity}}Logic;
    @Autowired
    private Get{{entity}}Logic get{{entity}}Logic;
```

```

@Override
protected {{entity}}Repository getRepository() {
    return repository;
}

@PostConstruct
public void init() {
    workflowService.map({{entity}}Logic.offer{{entity}}, assign{{entity}}Logic);
    workflowService.map({{entity}}Logic.get{{entity}}, get{{entity}}Logic);
}

@Override
protected Task preInsert({{entity}} {{entityVar}}) throws BusinessException {
    {{entityVar}}.setAssignedUser(userService.getOne(getCurrentUserId()));
    return super.preInsert(task);
}

public void assign{{entity}}ToUser(Integer {{entityVar}}Id, User user) throws
BusinessException {
    {{entity}} {{entityVar}} = getOne({{entityVar}}Id);
    workflowService.action({{entity}}action.assign{{entity}}ToUser,
{{entity}}Logic.assign{{entity}} {{entityVar}}, user);
}

public PageListDTO getUser{{entity}}(SearchFilterDTO filter) throws BusinessException {
    User user = userService.getOne(getCurrentUserId());
    return (PageListDTO) workflowService.action({{entity}}Action.getUser{{entity}},
{{entity}}Logic.get{{entity}}, filter, user);
}
}

```

Listagem A.3 - Expressão idiomática de uma interface para o padrão *Deferred Distribution* baseada na TaskLogic

```

public interface {{entity}}Logic extends WorkflowLogic {

    public void offer{{entity}}ToUser(Task task, User user) throws BusinessException;

    public PageListDTO getUser{{entity}}(SearchFilterDTO filter, User user) throws
BusinessException;
}

```

Listagem A.4 - Expressão idiomática de um Adapter para o padrão *Deferred Distribution* baseada no Adapter

```

public class Adapter implements {{entity}}Logic{
    @Autowired {{entity}}Repository repository;

    @Override
    public void offer{{entity}}ToUser(Task {{entity}}, User user) throws BusinessException{
        throw new BusinessException({{messageExceptionOffer}});
    }
}

```

```

        @Override
        public PageListDTO getUser{{entity}}(SearchFilterDTO filter, User user) throws
        BusinessException {
            throw new BusinessException({{messageExceptionGet}});
        }
    }
}

```

Listagem A.5 - Expressão idiomática de lógica para o padrão *Deferred Distribution* baseada no OfferTaskLogic

```

public class Offer{{entity}}Logic extends Adapter implements {{entity}}Logic {

    @Override
    public void offer{{entity}}ToUser({{entity}} {{entityVar}}, User user)
    throws BusinessException {
        {{entityVar}}.setAssignedUser(user);
        repository.save({{entityVar}});
    }

}

```

Listagem A.6 - Expressão idiomática de lógica para o padrão *Deferred Distribution* baseada no GetTaskLogic

```

public class Get{{entity}}Logic extends Adapter implements {{entity}}Logic {

    @Override
    public PageListDTO getUser{{entity}}(SearchFilterDTO filter, User user) throws
    BusinessException{
        FilterDTO filterDTO = new FilterDTO("assignedUser", user, Comparison.EQ);
        filter.getFilters().add(filterDTO);
        return repository.search(filter);
    }

}

```

Listagem A.7 - Expressão idiomática de enum para o padrão *Deferred Distribution* baseada no TaskAction

```

public enum {{entity}}Action {

    offer{{entity}}ToUser("offer{{entity}}ToUser"), getUser{{entity}}("getUser{{entity}}");

    private String description;

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

    public String getDescription(){

```

```

        return this.description;
    }
}

```

Listagem A.8 - Expressão idiomática de enum para o padrão *Deferred Distribution* baseada no TaskLogics

```

public enum {{entity}}Logics {
    offer{{entity}}("Offer{{entity}}"), get{{entity}}("Get{{entity}}");

    private String description;

    {{entity}}Logics(String description){
        this.description = description;
    }

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

```

Listagem A.9 - Função no *front-end* para atribuição de atividade a um usuário

```

assignTaskToUser(taskId) {
    const dialogConfig = new MatDialogConfig();
    dialogConfig.disableClose = false;
    dialogConfig.autoFocus = true;
    dialogConfig.data = {
        taskId: taskId
    }
    this.dialog.open(UserSelectComponent, dialogConfig);

    this.dialog.afterAllClosed.subscribe(()=>{
        this.listItems();
    })
}

```

Listagem A.10 - Representação no *front-end* para o botão de atribuição da atividade ao usuário

```

<button (click)="assignTaskToUser(item.id)" id="button-offerTaskToUser-{{item.id}}">
    <span>Atribuir Professor</span>
</button>

```

Listagem A.11 - Representação no *front-end* para a caixa de seleção de usuários

```

<form [formGroup]="editForm">
    <h2 mat-dialog-title>{{ 'User.user-select' | translate }}</h2>
    <mat-dialog-actions>

```

```

        <mat-select placeholder="{{ 'User.title.plural' | translate }}"
formControlName="assignedUser">
            <mat-option *ngFor="let user of users" [value]="user" (click)="selectUser()" >
                {{user.name}}
            </mat-option>
        </mat-select>
    </mat-dialog-actions>
</form>

```

Listagem A.12 - Representação no *front-end* da ação que envia os dados para o *back-end*

```

selectUser(): void{
    this.crudService.updatePartial(TaskURL.BASE, this.taskId, this.getFormValue()).subscribe(result => {
        this.notification.updateSuccess();
    }, error => {
        this.notification.error('Erro ao executar ação');
    });
    this.dialogRef.close();
}

```