Geração de Testes Automatizados: Uma análise comparativa no processo de garantia de qualidade de software[1]

Waldemar Jr. Dias Coimbra, Rodrigo de Almeida Vilar de Miranda

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

waldemar.junior@dcx.ufpb.br, rodrigovilar@dcx.ufpb.br

Abstract. *To ensure quality, teams always seek the path that provides the most productivity in the execution of activities, one of which is the automation of tests. Automating tests for certain systems can result in a large number of repeated codes. However, currently there are solutions that can solve the problem of high code repetition, especially the use of code generators, such as JHipster. This generator uses metadata and templates to generate its files, however, it became evident that there is a complexity in the implementation of templates for generating code for this tool. Aiming at an alternative to this model, we sought to implement templates using the fine-grained template approach, in order to compare which template approach demonstrates better encapsulation, allowing for better code reuse. Through the research it was possible to highlight that the fine-grained approach can improve the maintenance of templates for code generation, considering that through this approach it is possible to fragment the responsibilities of templates, thus improving the encapsulation, allowing the best code reuse.*

Resumo. *Para garantir qualidade, as equipes buscam sempre o caminho que proporcione mais produtividade na execução de atividades, sendo uma delas a automatização de testes. Automatizar testes para determinados sistemas pode acarretar uma grande quantidade de códigos repetidos. No entanto, atualmente existem soluções que podem resolver o problema de alta repetição de código, destacando-se o uso de geradores de código, como o JHipster. Esse gerador utiliza metadados e templates para gerar seus arquivos, porém, evidenciou-se que há uma complexidade na implementação de templates para a geração de código para essa ferramenta. Visando uma alternativa a esse modelo, buscou-se implementar templates utilizando a abordagem de templates de granularidade fina, com o intuito de comparar qual abordagem de template demonstra ter melhor encapsulamento, permitindo o melhor reuso de código. Através da pesquisa foi possível destacar que a abordagem de granularidade fina pode melhorar a manutenção de templates para geração de código, haja vista que através dessa abordagem é possível fragmentar as responsabilidades de templates, melhorando assim o encapsulamento, permitindo o melhor reuso de código.*

[1] Trabalho de Conclusão de Curso (TCC) na modalidade Artigo apresentado como parte dos pré-requisitos para a obtenção do título de Licenciado em Ciência da Computação pelo curso de Licenciatura em Ciência da Computação 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

O Teste de software é um processo que faz parte do ciclo de desenvolvimento, sendo este extremamente importante para garantir a qualidade de um sistema desenvolvido [MYERS,1979]. Atualmente existem diferentes maneiras de testar um software, se destacando as abordagens de testes manuais e testes automatizados. Os testes manuais são realizados por seres humanos através de casos de testes que possuem o passo a passo para obter o resultado esperado. No teste manual não há auxílio de um *script* automatizado para execução dos testes. Por outro lado, a automação de testes consiste na programação de um roteiro que exercita o software em desenvolvimento nos cenários de uso, comparando resultados esperados com os resultados reais. Através dessa abordagem, é possível que o teste seja repetido várias vezes, sendo mais fácil encontrar novos erros através da repetição e da simulação de cenários específicos, reduzindo o esforço humano em atividades manuais repetitivas. Além disso, os testes automatizados aumentam a produtividade do time de qualidade, pois podem focar em outras atividades mais relevantes do ciclo de testes, por exemplo, a formulação de casos de testes e planejamento.

A despeito dos benefícios advindos da automação de testes, uma questão principal pode tornar esse processo menos produtivo, com baixa qualidade e mais custoso. Ao automatizar testes para determinados sistemas, o testador pode escrever uma grande quantidade de código repetido. Isso acontece porque alguns cenários de teste repetem partes do mesmo processo e, consequentemente, do mesmo código. Por exemplo, na Figura 1:

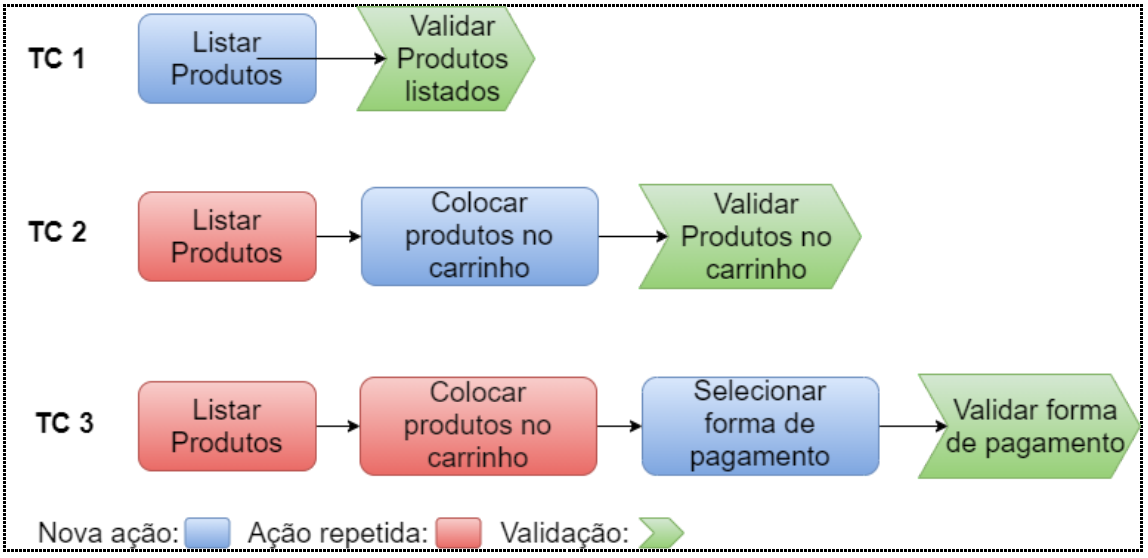

Figura 1. Diagrama para modelagem de casos de teste de exemplo

Verifica-se que, antes de colocar produtos no carrinho, é necessário listá-los. Da mesma forma que, para selecionar a forma de pagamento, é necessário listar os produtos e colocá-los no carrinho. Para executar esses três cenários de teste será necessário a repetição de atividades já programadas. Sendo assim, ao automatizar esse teste, eventualmente haverá repetição de código em diversas seções do roteiro, com basicamente a mesma funcionalidade. Esse problema pode impactar diretamente os

times de Qualidade de corporações TI, por haver um esforço repetitivo para escrever a mesma parte de roteiro, em testes diferentes.

Não obstante, existe outra causa que pode gerar repetição de código. Por exemplo, um sistema para biblioteca que contenha três entidades diferentes, sendo que para cada uma é possível realizar as ações de criar, visualizar, editar e excluir, também conhecido como CRUD (*Create, Read, Update e Delete*). Portanto haverá ações semelhantes porém com finalidades diferentes para cada uma dessas entidades, como exemplificado na Figura 2.

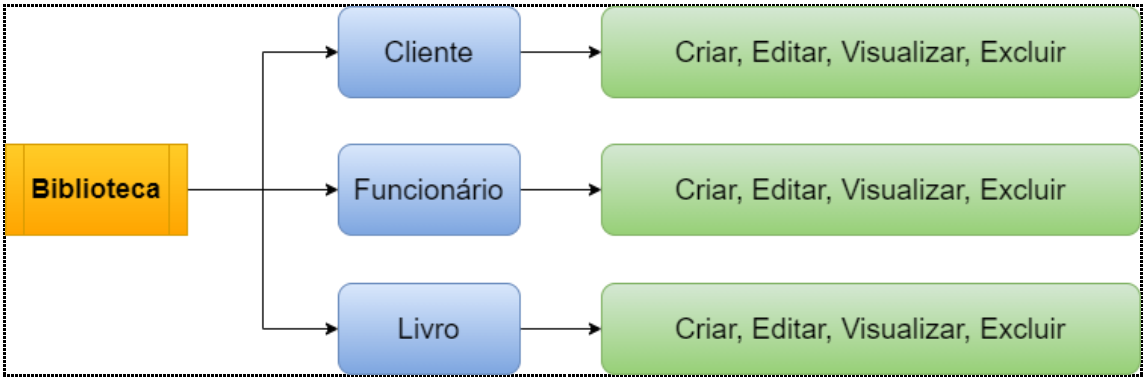

Figura 2. Diagrama do sistema para biblioteca

Nesse sistema, seria possível notar a repetição da estrutura de interface para os CRUDs, tendo como diferença apenas os campos dos formulários de cada entidade e as colunas das tabelas de listagem. A elaboração do *script* para os testes automatizados desses CRUDs apontaria para um alto número de linhas de código repetidas, diferindo apenas nas entradas e saídas em cada *script*.

É possível notar, a partir desses exemplos, que mesmo com a automação de testes, ainda pode haver limitadores na produtividade das equipes de teste. A repetição de código aumenta o custo do projeto de desenvolvimento.

Atualmente existem soluções para resolver o problema de alta repetição de código, destacando-se o uso de herança, composição, padrões de projeto, *frameworks*, bem como geradores de código, sendo esse o escopo deste trabalho.

Foi feito um levantamento no mercado, onde foi possível destacar a existência de geradores que se mostram promissores: Ruby on Rails[2], Spring Roo[3], Grails[4], Yeoman[5] e JHipster[6]. Porém, desenvolver *templates* para geradores de código torna-se uma tarefa complexa, levando em consideração os modelos existentes até o momento. *Templates* de granularidade grossa, mistura de responsabilidades e omissão em casos de teste são barreiras que tornam os geradores presentes no mercado complexos e de difícil implementação. Além de que são de difícil compreensão e, portanto, com difícil aplicação em grandes sistemas empresariais.

[2] https://rubyonrails.org/
[3] https://projects.spring.io/spring-roo/
[4] https://grails.org/
[5] https://yeoman.io/generators/
[6] https://www.jhipster.tech/

Como objetivo principal deste trabalho, busca-se aplicar os conceitos de *templates* de granularidade fina, propostos por Vilar et al (2015) nos *templates* do JHipster, com o intuito de comparar qual *template* demonstra ter melhor encapsulamento. O escopo será testes e2e (fim a fim), para execução através do *framework* Protractor[7], sobre aplicações com *front-end* Angular[8].

De modo concreto serão analisadas as métricas: quantidade total de linhas de código, quantidade de mudanças de responsabilidades, quantidade de artefatos, a média de linhas por artefatos, a média de mudanças de responsabilidades por artefatos e a cobertura de casos de teste (funções do sistema), sendo para isso utilizado um comparador de códigos. Além disso, este trabalho possui os seguintes objetivos específicos:

- O levantamento dos geradores de código de testes presentes como solução atualmente, através de uma pesquisa bibliográfica;
- O desenvolvimento de *templates* para geração de código Protractor (e2e), utilizando o modelo de *templates* com granularidade fina, que deverá ser capaz de cobrir o mesmo escopo do JHipster;
- Criação de uma metodologia para que seja feita a comparação entre os *templates* gerados pelo JHipster e o *Template* de granularidade fina.

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica, contendo um estudo sobre a área de testes, um estudo sobre *templates* de geração de código com granularidade fina e um levantamento do estado da arte (academia) e estado da prática (indústria) para geração de código. Na Seção 3 é apresentada a metodologia utilizada, contendo os passos para a elaboração da pesquisa. A Seção 4 mostra uma análise dos *templates* do gerador de código JHipster, que se mostrou a melhor ferramenta encontrada para o escopo deste trabalho. A Seção 5 descreve a criação de *templates* de granularidade fina para cobrir o mesmo escopo de testes e2e do JHipster. Na Seção 6 é realizada uma comparação entre os dois tipos de abordagens, e por fim, a Seção 7, de conclusão do trabalho, fazendo um apanhado geral sobre a pesquisa.

2 Fundamentação Teórica

Esta Seção apresenta conceitos que servirão de base para o desenvolvimento deste trabalho. A Seção é iniciada com uma introdução a Testes de Software, apresentando os tipos utilizados por equipes de qualidade de software, seguido de um levantamento do estado da arte e estado da prática para geração de códigos. No final é apresentado um estudo sobre *templates* de geração de código com granularidade fina.

2.1 Teste de Software

Segundo Delamaro et al. (2007) o objetivo do teste de software é executar programas ou produtos intermediários com entradas específicas, analisando se o comportamento desses produtos está de acordo com o resultado esperado. O

[7] https://www.protractortest.org/
[8] https://angular.io/

planejamento de teste, o projeto de casos de teste, a execução de testes e a avaliação dos resultados dos testes são os passos básicos do ciclo de teste.

2.1.1 Tipos de Teste de Software

Dentre as formas de teste mais usadas atualmente estão os testes manuais e testes automatizados. Como o próprio nome sugere, testes manuais são realizados por seres humanos através da execução de casos de testes, que possuem um passo a passo ou modelo com entradas e saídas esperadas, objetivando um determinado resultado. No teste manual de software não costuma haver a utilização de *scripts* automatizados. Esse modelo de teste é amplamente seguido em corporações TI e é basicamente o mais utilizado pelas equipes de qualidade de software no mercado, principalmente porque há alguns testes requisitados que necessitam de humanos executando, por exemplo: Testes exploratórios, Testes de Usabilidade e Testes Ad-hoc. Porém, nota-se que planejar casos de teste bem definidos, planejar ciclos de teste e ainda executar os testes demandam muito tempo das equipes de qualidade. Além de que executar determinados casos de teste se torna uma tarefa extremamente repetitiva. Nesse sentido, uma solução que tem sido muito difundida é a automação de testes.

Bartié (2002, p. 196), diz que os testes automatizados são "[...] a utilização de ferramentas de testes que possibilitem simular usuários ou atividades humanas de forma a não requerer procedimentos manuais no processo de execução dos testes.". Para Molinari (2003, p. 104), o teste automatizado "[...] permitirá aumentar a profundidade e abrangência dos casos de testes envolvidos.". Segundo Bartié (2002, p. 181), "A automação exige um esforço inicial de criação, porém possibilita uma incomparável eficiência e confiabilidade, impossível de ser atingida com procedimentos manuais.". Pressman (2011, p. 404) afirma que o teste "[...] pode ser executado manualmente, re-executando um subconjunto de todos os casos de teste ou usando ferramentas automáticas de captura/reexecução". Isso denota que os testes automatizados podem agregar grande valor nos ambientes das equipes de qualidade.

A execução de Testes de regressão, Testes de carga e Testes de desempenho demanda muito esforço por parte dos testadores, pois são atividades altamente repetitivas e, portanto, consomem muito tempo. Com o uso de testes automatizados há uma redução do tempo de execução de atividades dessa natureza.

Uma ferramenta muito utilizada atualmente nas empresas por equipes de qualidade para automatização de testes é o Protractor. Esse *framework* é usado para testes e2e (*end-to-end*) de aplicações desenvolvidas em Angular[9], e interage com *browsers* reais como um ser humano faria. Protractor executa sobre a plataforma Node[10] e foi criado baseado no Selenium[11] para interagir com elementos da página web DOM (*Document Object Model*). Além disso, o Protractor usa o Jasmine[12], que é uma ferramenta de testes de comportamento para Javascript[13], e é através dele que ocorre a escrita dos casos de teste. O Protractor permite automatizar casos de teste em que a ação

[9] https://angular.io/cli/e2e
[10] https://nodejs.org/en/
[11] https://www.selenium.dev/
[12] http://jasmine.github.io/
[13] https://www.javascript.com/

humana não é exclusivamente necessária, além de executá-los. Vale ressaltar que esse trabalho pretende comparar *templates* para geração de código fonte de testes e2e para execução no Protractor.

2.2 Geração de Código

O ciclo de desenvolvimento de uma aplicação demanda um tempo significativo com tarefas altamente repetitivas, tais como o desenvolvimento de CRUDs básicos e avançados, implementação de interface gráfica, configurações de *framework*, etc. Uma solução que vem sendo cada vez mais amadurecida é a geração de código. Através desse recurso, é possível automatizar as atividades repetitivas do ciclo de vida de uma aplicação.

Através do uso de geradores, é possível gerar automaticamente o código e as configurações iniciais de uma aplicação, possibilitando assim a sua execução pelo desenvolvedor em poucos minutos. Exemplos de geradores de código são as ferramentas *Ruby on Rails, Spring Roo, Grails, Yeoman e JHipster,* como citado em sessões anteriores. Todas essas ferramentas possibilitam a geração de código a partir de metadados e *templates*. A Figura 3 ilustra o funcionamento de um gerador dessa natureza.

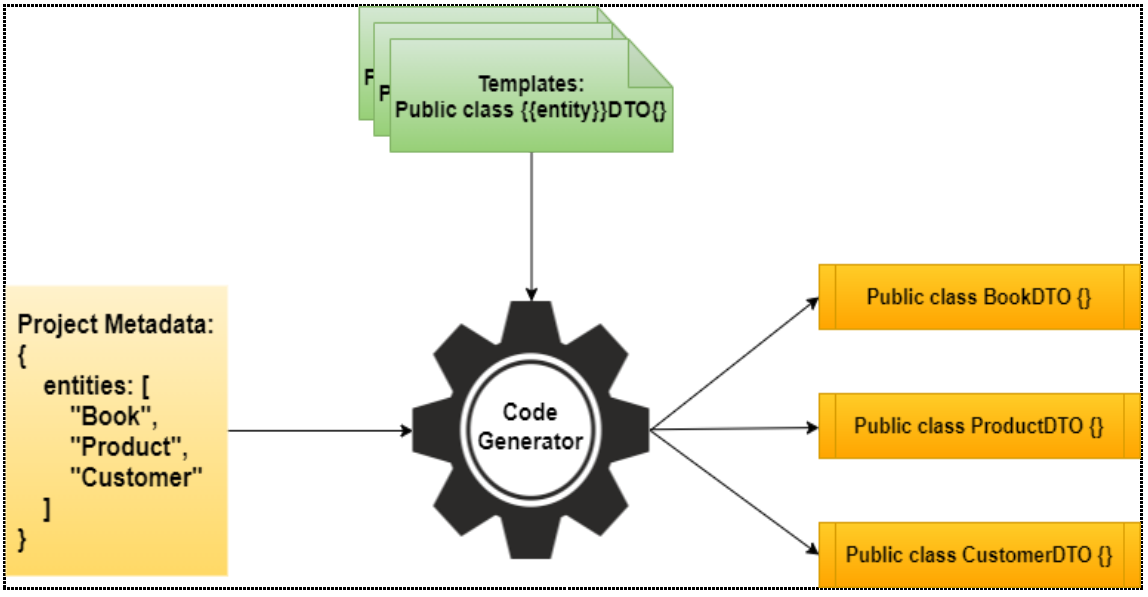

Figura 3. Exemplo do funcionamento de um gerador de código

O *Ruby on Rails*, também conhecido como *Rails* ou *RoR* é um projeto de código aberto, escrito na linguagem de programação *Ruby*[14]. As aplicações criadas utilizando o *framework* Rails são desenvolvidas com base no padrão de arquitetura MVC (*Model-View-Controller*). O Spring Roo é uma ferramenta para a criação de aplicativos web *Spring*. O Grails é um framework para desenvolvimento de aplicações web que utiliza a linguagem de programação *Groovy (*linguagem dinâmica para a plataforma Java[15]). O JHipster é uma plataforma de desenvolvimento para gerar, desenvolver e implantar rapidamente aplicativos web e arquiteturas de microsserviço.

[14] https://www.ruby-lang.org/pt/
[15] https://www.java.com/pt-BR/

Vilar et al. (2015) propuseram o gerador de código *Potter*[16], que funciona também a partir de metadados e *templates*, no entanto esse gerador, diferentemente dos demais, possui uma abordagem de *templates* de granularidade fina. Essa abordagem é descrita no tópico a seguir.

2.3 *Templates* de Geração de Código

Existem tipos diferentes de abordagem utilizadas em *templates* de geradores de código. O JHipster por exemplo, utiliza a abordagem de *template* de granularidade grossa. Neste trabalho, entende-se o modelo de *template* de granularidade grossa sendo a junção de várias responsabilidades em um só artefato de código. Na Seção 4, é mostrado parte de um *template* de granularidade grossa utilizado no gerador JHipster. Esse modelo pode se tornar complexo à medida que evolui, pois a mistura de responsabilidades pode piorar a manutenibilidade do *template*.

Por outro lado, a abordagem de *templates* de granularidade fina, utilizado no gerador *Potter,* visa o encapsulamento das responsabilidades. Essa abordagem, diferente da anterior, permite o reuso de código, visando melhorar a manutenibilidade dos *templates* propriamente ditos. Vilar (2017, p. 144) aponta que "[...]é possível aumentar o reuso sem prejudicar a manutenibilidade no desenvolvimento de (...) aplicações corporativas web, através do encapsulamento das responsabilidades (...) em artefatos de granularidade fina com base nos metadados do modelo do domínio [...]".

Com a abordagem de granularidade fina, é possível haver a separação de cada responsabilidade dentro do *template*. De forma parecida a orientação a objetos, o modelo permite que um *template* incorpore o código de outro. Assim, cada micro *template* tem sua responsabilidade e então é utilizado quando e se for necessário. Evoluir o *template* se torna menos complexo, resultando em um aumento de produtividade.

3. Metodologia

Visando alcançar o objetivo principal dessa pesquisa, que é aplicar os conceitos de *templates* de granularidade fina nos *templates* do JHipster, evidenciando o *template* que demonstra melhor encapsulamento, foram necessários seis passos:

1. Estudo sobre a área de testes de software;
2. Levantamento do estado da arte (academia) e estado da prática (indústria) para geração de código;
3. Estudo sobre *templates* de geração de código com granularidade fina;
4. Análise dos *templates* da melhor ferramenta encontrada;
5. Criação de *templates* de granularidade fina para cobrir o mesmo escopo de testes e2e do JHipster;
6. Comparação entre os *templates* de granularidade grossa e fina.

Para realizar o estudo sobre a área de testes, foi utilizado o motor de busca do Google. A pesquisa foi feita levando em consideração também conhecimentos prévios sobre o assunto. Dessa forma, buscou-se entender o objetivo do teste de software no

[16] encurtador.com.br/nsQY2

ciclo de desenvolvimento. Além disso, também procurou-se entender os tipos de teste de software e sua importância no contexto do mercado atualmente.

Também foi imprescindível executar um levantamento do estado da arte, bem como do estado da prática para geração de código. Esse levantamento foi feito através de uma pesquisa bibliográfica utilizando o motor de busca do Google Scholar, com as *Strings* "Geração automática de código", "Geração automática de código e2e", "Geração automática de código web" e "Geração automática de teste". O intervalo de tempo foi entre os anos 2007 a 2020.

Com esse levantamento buscou-se explorar trabalhos e ferramentas desenvolvidas na academia para geração automática de código. Um dos problemas encontrados nessa pesquisa foi a dificuldade para adquirir acesso a grande parte dos trabalhos existentes na academia para esse contexto, por estarem bloqueados pelos autores. Outro problema foi que, devido ao tempo de publicação de alguns trabalhos, o conteúdo se mostrou pouco eficiente para colaborar nesta pesquisa. Dos trabalhos que foi possível o acesso, somente dois demonstraram ter uma ferramenta que pudesse gerar código. Turner et al (2008) e Sakamoto et al (2013) propõem geradores de código que podem fornecer uma estrutura capaz de se mostrar solução para o problema, porém, não foi possível ter acesso a essas ferramentas.

Ao fazer o levantamento do estado da prática, foi possível destacar cinco ferramentas capazes de gerar código, sendo estas: Ruby on Rails, Spring Roo, Grails, Yeoman e JHipster, como já mencionado no tópico 2.2 deste trabalho. Esse levantamento foi feito através do motor de busca do Google, com as *Strings* "Geração automática de código", "Gerador automático de código e2e" e "plataforma de *Scaffolds*". Os resultados foram semelhantes aos encontrados por Vilar (2017, p. 3) que apontou "[...] Os templates de geração de código possuem estrutura baseada em metadados e são utilizados em plataformas de *Scaffolding* como Ruby on Rails, Grails, Spring Roo e Yeoman.[...]". Todas essas ferramentas demonstraram ter boa aceitação pelas comunidades e, como citado, utilizam metadados e *templates* para gerar código. O JHipster é um dos geradores de código baseados no Yeoman e foi a ferramenta que mais se destacou, principalmente porque, diferente das demais, é capaz de gerar, desenvolver e implantar aplicativos web e arquiteturas de microsserviços, sendo ainda possível gerar testes E2E para o framework Protractor, que é o escopo deste trabalho.

Porém, mesmo com a grande eficiência dessas ferramentas para o mercado, apurou-se que há uma complexidade na implementação dos *templates* para a geração de código. Para Vilar (2017, p. 5), [...]As abordagens baseadas em metadados são eficazes em relação ao reuso, uma vez que são independentes do domínio e podem ser reutilizadas em diversos projetos, porém são de difícil manutenção devido à generalidade excessiva, complexidade e má divisão de responsabilidades[...]. Visando encontrar uma alternativa aos modelos de *template* utilizados por essas ferramentas, buscou-se utilizar neste trabalho o modelo de *templates* de granularidade fina, proposto por Vilar *et al* (2015). O objetivo é a criação de *templates* de granularidade fina para cobrir o mesmo escopo de testes e2e do JHipster. Por fim, visando analisar e comparar qual dos *templates* demonstra ter menor média de tamanho (número de linhas de código por template/artefato), melhor encapsulamento, cobrindo a mesma quantidade

de casos de teste, será feita uma análise comparativa. Para isso será utilizada a metodologia de pesquisa quantitativa, objetivando verificar a quantidade total de linhas de código, a quantidade de mudanças de responsabilidades, quantidade de artefatos, a média de linhas por artefatos e a média de mudanças de responsabilidades por artefatos. Além disso, os códigos gerados a partir dos dois modelos de *template* foram submetidos a um comparador de código, para evidenciar a semelhança entre os arquivos gerados.

4. O JHipster

JHipster é uma plataforma de desenvolvimento para gerar, desenvolver e implementar rapidamente aplicativos web modernos e arquiteturas de microsserviço Spring Boot[17] + Angular (ou React[18]), mantido e atualizado por meio de uma comunidade. Uma das características do JHipster é o uso baseado em linha de comando para a criação de toda a estrutura do projeto, sendo essa também chamada de *Scaffolding* (termo em inglês da engenharia civil, que denota a colocação de andaimes e outras estruturas). Para a programação, Vilar (2017, p. 47) aponta que *Scaffolding* "[...] consiste em gerar automaticamente o código e a configuração inicial da aplicação, a partir dos metadados do modelo conceitual. Desse modo, o programador pode executar alguma funcionalidade da aplicação, mesmo que primitiva, após poucos minutos de desenvolvimento. [...]". Para o Scaffolding é usado o Yeoman, que é uma tecnologia baseada em metadados para o desenvolvimento de sistemas que utiliza *templates* de geração de código.

O Yeoman é uma ferramenta que oferece suporte na inicialização de novos projetos, estabelecendo práticas e ferramentas recomendadas para maior produtividade. Esse suporte é feito através de um ecossistema de geradores, sendo o JHipster um destes. Cada um dos geradores é uma ferramenta customizada construída em cima da arquitetura do Yeoman para gerar um resultado desejado.

Para os testes de integração UI Angular ou React do JHipster, é utilizado de forma opcional o framework Protractor, sendo esse o escopo deste trabalho. Atualmente o JHipster é capaz de gerar o *script* básico para execução de testes e2e (fim a fim) Protractor, sendo possível a geração de testes para validação de login e CRUD (*Create, Read, Update e Delete*) de entidades, cobrindo a validação básica dos atributos de cada CRUD, além do relacionamento entre entidades.

O gerador produz *scripts* de teste utilizando Typescript[19] como linguagem e contém em seus arquivos os *Specs*, bem como os *Page Objects*. Os arquivos *spec* contém a estrutura do teste e suas asserções, enquanto que o *Page Object* modela os objetos contidos na interface do usuário, dentro do código de teste. O acesso aos elementos da página passa a ser de responsabilidade de um *Page Object* e este provê métodos que tornam possível realizar ações sobre esses elementos, sendo esse o conceito de encapsulamento. O teste interage com esses objetos e então simula um usuário real interagindo com o sistema. Nos Trechos de código 1 e 2, é possível ver parte de um código de arquivos *spec* e *page object* respectivamente, gerados a partir do

[17] https://spring.io/projects/spring-boot
[18] https://pt-br.reactjs.org/
[19] https://www.typescriptlang.org/

JHipster, para uma entidade hipotética chamada *Generic* que tem campos os *FieldString, FieldInteger*, etc utilizada durante este trabalho de pesquisa. Vale ressaltar que o nome dessa entidade e seus atributos foram escolhidos de forma estratégica, a fim de tornar mais clara a exploração de todas as possibilidades de cobertura do JHipster, para teste e2e Protractor. O código fonte completo se encontra no repositório jhipster-generic-application[20] do Github.

```
it('should create and save Generics', async () => {
    const nbButtonsBeforeCreate = await
        genericComponentsPage.countDeleteButtons();
    await genericComponentsPage.clickOnCreateButton();
    await promise.all([
      genericUpdatePage.setFieldStringInput('fieldString'),
      genericUpdatePage.setFieldIntegerInput('5'),
      genericUpdatePage.setFieldLongInput('5'),
      genericUpdatePage.setFieldBigDecimalInput('5'),
      genericUpdatePage.setFieldFloatInput('5'),
```

Trecho de código 1. Arquivo *spec* gerado a partir do JHipster para definir valores em um formulário

```
export class GenericComponentsPage {
  createButton = element(by.id('jh-create-entity'));
  deleteButtons = element.all(by.
      css('jhi-generic div table .btn-danger'));
  title = element.all(by.css('jhi-generic div
      h2#page-heading span')).first();
  noResult = element(by.id('no-result'));
  entities = element(by.id('entities'));

  async clickOnCreateButton(): Promise<void> {
    await this.createButton.click();
  }

  async clickOnLastDeleteButton(): Promise<void> {
    await this.deleteButtons.last().click();
  }
```

Trecho de código 2. Arquivo *Page Object* com os componentes do formulário

Para gerar os scripts de teste, o JHipster utiliza *templates* para cada tipo de arquivo gerado, *specs* e *page objects*. Porém, ao analisar os *templates* responsáveis pela geração desses scripts, nota-se que há um alto nível de complexidade na sua compreensão. A estrutura do *template* reflete a mistura de várias responsabilidades, aumentando a dificuldade de implementação da expansão do gerador, além de apresentar um alto número de linhas de código por *template*. Isso demonstra que os *templates* do JHipster tem alto nível de acoplamento. Quanto maior o acoplamento, mais complexo se torna o *template*. No Trecho de código 3, que demonstra parte do *template* responsável pela geração do *spec* de entidades do JHipster, é possível verificar

[20] https://github.com/walcoimbra19/jhipster-generic-aplication

a mistura de responsabilidade em um único caso de teste "*should create and save*". No texto na cor Azul é possível notar o código relacionado a propriedades, de acordo com seus tipos. Já no trecho onde a cor do texto é Vermelho demonstra o código relacionado a entidades. Por último, os trechos na cor verde representam os relacionamentos entre entidades.

```
<%= openBlockComment %>it('should create and save <%= entityClassPlural %>', async () =>
{
    const nbButtonsBeforeCreate = await <%= entityInstance
            %>ComponentsPage.countDeleteButtons();
    await <%= entityInstance%>ComponentsPage.clickOnCreateButton();
    ...
    <%_ if (['Integer', 'Long', 'Float', 'Double', 'BigDecimal'].includes(fieldType)) {
_%>
    <%= entityInstance %>UpdatePage.set<%= fieldNameCapitalized %>Input('5'),
    <%_ } else if (fieldType === 'LocalDate') { _%>
    <%= entityInstance %>UpdatePage.set<%= fieldNameCapitalized %>Input('2000-12-31'),
    <%_ } else if (['Instant',
            'ZonedDateTime'].includes(fieldType)) { _%>
    <%= entityInstance %>UpdatePage.set<%=
            fieldNameCapitalized %>Input('01/01/2001' +
            protractor.Key.TAB + '02:30AM'),
    <%_ } else if (fieldType === 'Duration') { _%>
    <%= entityInstance %>UpdatePage.set<%=
            fieldNameCapitalized %>Input('PT12S'),
    <%_ } else if (['byte[]',
            'ByteBuffer'].includes(fieldType) &&
            fieldTypeBlobContent === 'text') { _%>
    <%= entityInstance %>UpdatePage.set<%=
            fieldNameCapitalized %>Input('<%= fieldName %>'),
    ...

<%_ relationships.forEach((relationship) => {
      const relationshipType = relationship.relationshipType;
      const ownerSide = relationship.ownerSide;
      const relationshipName = relationship.relationshipName;
      const relationshipFieldName =
            relationship.relationshipFieldName;
_%>
```

Trecho de código 3. *Template* de caso de teste com as diferentes responsabilidades destacadas

Vale ressaltar que dentro desse mesmo *template* ainda há outros casos de teste, e portanto, refletindo em um alto número de linhas (219), como citado anteriormente. Esse alto número de linhas e responsabilidades dificulta a expansão do gerador, bem como a compreensão do mesmo. Além disso, pode haver o surgimento de anomalias no código (*code smells*) em decorrência dessa problemática. O repositório completo com os códigos dos *templates* de *spec* e *page objects* se encontra em generator-jhipster[21] no Github.

5. *Templates* de granularidade fina

Visando apresentar uma abordagem de *template* diferente da utilizada no JHipster, objetivando melhorar assim o encapsulamento, permitindo o melhor reuso de

21 https://github.com/jhipster/generator-jhipster/tree/master/generators/entity-client/templates/angular/src/test/javascript/e2e/entities

código para ganho em produtividade, optou-se por utilizar o modelo de *template* de granularidade fina proposto por Vilar et al (2015), como citado anteriormente. Para que fosse possível interpretar e processar os *templates* desenvolvidos, foi utilizado o gerador de código baseado em *templates* de granularidade fina *Potter*.

Para Magno (2015), o ciclo de vida de um gerador baseado em *templates* tem 4 fases, sendo estas:

1. Criar ou utilizar o código fonte de projetos bem-sucedidos que implementam uma determinada funcionalidade desejada;
2. Parametrizar todo o texto genérico no código, como nome de classes, métodos e etc;
3. Implementar o gerador para atribuir os valores do metamodelo nos *templates* parametrizados da fase 2;
4. Gerar o código fonte.

Seguindo o modelo proposto por Magno (2015), optou-se por utilizar o código fonte gerado a partir do *template* do JHipster. Além disso, o código de teste e2e, que é o escopo deste trabalho, cobre validações básicas para CRUDs. Também foi utilizada a parametrização incorporada no *template* do JHipster, para que fosse possível identificar todos os casos de testes e suas funções.

Como citado anteriormente, para o desenvolvimento deste trabalho, foi utilizado o gerador de código *Potter*. Neste existem três tipos de objetos que são chamados de Worker, sendo estes os *Templates, Designs e Shaping*. Os *templates* são responsáveis por gerar os códigos para qualquer linguagem ou formato. Os *designs* definem as regras para conectar os *workers* e o *shaping* processa e transforma os metadados. O Potter dispõe de uma plataforma na nuvem para tornar mais simples a interação com o gerador. Para esse trabalho, após definir o metadado, foi necessário criar um projeto na plataforma para então incorporar o metadado e os workers ao projeto e então executá-los gerando, como resultado, os arquivos finais *spec* e *page objects*.

Para esse projeto foram criados seis *designs*, sendo estes "*Protractor:1*", "*RootE2E:1*", "*ProtractorEntities:1*", "*EntitySpec*" e "*EntityPO*". O "*Protractor:1*" basicamente remete a ferramenta Protractor, sendo esta o escopo do trabalho, além de rodar um comando *shell* utilizando o Prettier[22] (formatador de código) para formatar os códigos dos arquivos resultantes. Além disso, o design "*Protractor:1*" chama o "*RootE2E:1*" e cria uma compilação dos arquivos do projeto em formato compactado. O *design* "*RootE2E:1*" define a criação dos diretórios dos arquivos de *spec* e *page objects* que serão gerados, além de executar o *template* "*PageObjects:1*", que é o arquivo de *page object* genérico, não sendo necessária modificação. Além disso, o "*RootE2E:1*"ainda é responsável por chamar o *design* "*ProtractorEntities:1*".

A partir do *design* "*ProtractorEntities:1*", é criado um diretório para cada entidade, contendo as pastas e arquivos para *spec* e *page objects* resultantes, além da execução dos *designs* "*DesignSpec:1*" e "*DesignPO:1*". Vinculados a estes *designs* estão os *sub-templates* que serão chamados e executados através de portas. Vale ressaltar que os *designs*, assim como os metadados, são definidos como arquivo

[22] https://prettier.io/

JSON[23]. Na Figura 4 é possível ter uma visão geral da organização dos workers (*designs e templates*) através do diagrama do projeto. Para este trabalho, não foi necessária a utilização de *Shape*.

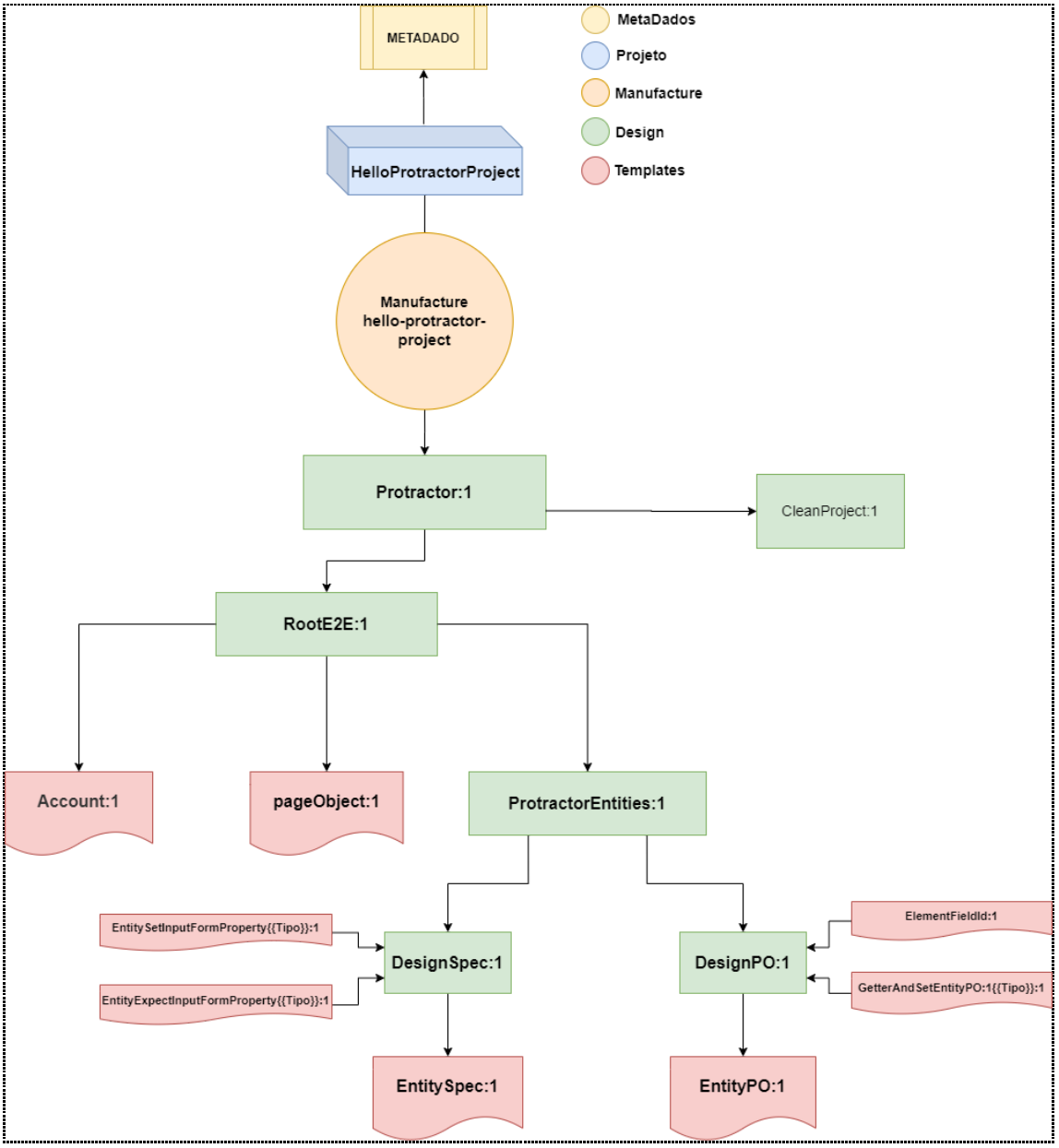

Figura 4. Diagrama do projeto

A criação do metadado para leitura a partir do *Potter* foi o primeiro passo para que fosse possível realizar todo o processo de geração. Nele foram definidas as informações básicas do projeto, bem como as entidades e seus atributos. Este foi definido como arquivo JSON (Javascript *object notation*), sendo esse um requisito do *Potter*. O trecho de código 4 demonstra parte do formato do metadado para o projeto. É possível notar que o nome das entidades foi determinado de forma genérica. Já o nome

[23] https://www.json.org/json-pt.html

dos atributos remete aos tipos utilizados em *TypeScript*[24], para que fosse possível validar todos os tipos cobertos pelo JHipster.

```
"project": {
      "id": "e2eprotractor1",
      "description": "e2e protractor 1",
      "version": "0.0.1"
    },
    "entities": [
      {
        "id": "001",
        "name": "generic",
        "labels": {
          "singular": "generic",
          "plural": "generics"
        },
        "properties": [
          {
            "name": "fieldString",
            "label": "FieldString",
            "type": "string"
          },
          {
            "name": "fieldInteger",
            "label": "FieldInteger",
            "type": "integer"
          }, (...)
```

Trecho de código 4. Metadados do projeto

Como o objetivo desta etapa é desenvolver *templates* com granularidade fina, o design "*DesignSpec:1*" tem como função executar o *template* "*EntitySpec:1*" e, quando necessário, executar um *subtemplate* para cada tipo suportado pelo JHipster. Da mesma forma, o design "*DesignPO:1*" executa o *template* "*EntityPO:1"* e, quando necessário, executa um *subtemplate* para cada tipo. De maneira objetiva, os *templates* "*EntitySpec:1*" e "*EntityPO:1"* são responsáveis por gerar o código fonte final dos arquivos *specs e page objects* para cada entidade dos metadados. Porém, diferente do *template* utilizado pelo JHipster, objetivou-se separar em sub *templates* as responsabilidades contidas nos *templates*.

[24] https://www.typescriptlang.org/

As funções para definir valores nos campos são abordadas de maneira diferente nos *templates* de granularidade fina. A função para cada tipo é chamada a partir de outro *template*, que é invocado através de uma porta. No *template* de *spec* é utilizado esse contexto para as funções de preencher valores nos formulários e as funções *expect*, com os possíveis resultados esperados contidos nos formulários. No *template* do *page objects* funciona da mesma forma, porém para a chamada das funções de captura de elementos da tela e de *Getter* e *Setter.* No Trecho de código 6 é demonstrado parte de um código do *template* "*EntitySpec:1*", tornando evidente a separação das responsabilidades.

```
it('should create and save {{ capitalize labels.plural }}', async () =>
{
    const nbButtonsBeforeCreate = await {{ labels.singular }}
      ComponentsPage.countDeleteButtons();
    await {{ labels.singular }}ComponentsPage.clickOnCreateButton();

    await promise.all([
      {{ set-input-form-property-e2e properties }} //Porta para os set
    ]);

    {{ expect-input-form-property-e2e properties }} //Porta p/ expects

    await {{ labels.singular }}UpdatePage.save();
    expect(await{{ labels.singular }}UpdatePage.getSaveButton()
      .isPresent(),'Expected save button disappear').to.be.false;

    expect(await {{ labels.singular }}ComponentsPage.
      countDeleteButtons()).to.eq(nbButtonsBeforeCreate + 1,
      'Expected one more entry in the table');
  });
```

Trecho de código 5. *Template* EntitySpec:1 com separação de responsabilidades através de portas

Nos Trechos de código 6 e 7, respectivamente, são apresentados os códigos dos *templates* que são invocados pelo *template* "*EntitySpec:1*", quando há a requisição através da porta "*set-input-form-property-e2e*", para os tipos *String e Integer*.

```
{{labels.singular}}UpdatePage.set{{capitalize name}}Input('fieldString'),
```

Trecho de código 6. Sub *Template da função set* para o tipo String

```
{{ labels.singular }}UpdatePage.set{{capitalize name}}Input('5'),
```

Trecho de código 7. Sub *Template da função set* para o tipo Integer

O código contido nesses *templates* não explicita o tipo para que foi solicitado. Essa definição é feita através de regras contidas no design "DesignSpec:1" para os *specs* e no "DesignPO:1" para os *page objects*. No *Potter* é possível definir o contexto em que o *template* será aplicado, e utilizar seletores para designar uma regra. Nesse caso, foi utilizado o contexto entidade e seletores para restringir a chamada dos sub *templates* pelo tipo de atributo. Por exemplo, para utilizar o atributo nome (*String*), o *template "EntitySpec:1*" chama o sub *template* responsável por essa função através da porta "*set-input-form-property-e2e*", após fazer uma busca pelo atributo nome e

verificar através do seletor qual o tipo desse atributo. O Trecho de código 10 demonstra a regra contida no design “EntitySpec:1”.

```
{
      "port": "set-input-form-property-e2e",
      "selector": {
          "type": "string"
      },
      "commands": [
          {
           "template": "Template:EntitySetInputFormPropertyString:1"
          }
      ]
 }
```

Trecho de código 8. Regra para o selecionar um *template* para os atributos *string*

No *template “EntityPO:1”* a separação de responsabilidades funciona exatamente da mesma forma, uma vez que todo código responsável pelas funções de *getter* e *setter,* representado no Trecho de código 12, é chamado através da porta “*getter-and-set-entity-po*”, como também acontece na função “*getPageTitle*”. Para identificar os atributos e então invocar outros *templates* do *page obejct*, o *template* executa a porta “*element-field-id*” passando o contexto, que nesse caso são as propriedades “*properties*”. O trecho de código contido nos *templates* invocados é incorporado ao arquivo final, de acordo com as propriedades e seus tipos. Os *templates* dessa forma ficam com o número de linhas notavelmente menor em comparação ao mesmo trecho de código nos *templates* do JHipster. O código Trecho de código 12 demonstra o *template* “*getter-and-set-entity-po*”, responsável pelos *getter* e *setter*. O repositório *Protractor-entity-potter-generator*[25] contém todos os códigos desenvolvidos para esse projeto.

```
async get{{capitalize name}}Input(): Promise<string> {
    return await this.{{name}}Input.getAttribute('value');
  }

async set{{capitalize name}}Input({{name}}: string): Promise<void> {
    await this.{{name}}Input.sendKeys({{name}});
  }
```

Trecho de código 9. *Template* getter-and-set-entity-po

6. Comparação

Uma vez apresentada a sistemática de cada gerador e seus *templates*, é importante avaliar comparativamente as abordagens de granularidade grossa e fina. Para essa comparação foram analisadas em cada uma das abordagens a quantidade total de linhas de código, de mudanças de responsabilidades, de artefatos, a média de linhas por artefatos e a média de mudanças de responsabilidades por artefatos. Além disso, os arquivos finais de *spec* e *page object* gerados através dos *templates* de granularidade fina e granularidade grossa foram submetidos a um comparador de arquivos textuais, a

[25] Inserir o código do github

fim de verificar as semelhanças e diferenças entre ambos. Vale ressaltar que as linhas de código relacionadas ao relacionamento entre entidades, presentes nos *templates* do *JHipster,* foram desconsideradas nesse primeiro momento e farão parte de futuras análises.

Para analisar a quantidade total de linhas de código dos *templates* de *spec* e *page object* do JHipster foi necessário avaliar os arquivos base contidos no repositório GitHub da ferramenta. É importante destacar que o próprio repositório informa a quantidade de linhas de código em cada um dos *templates*, porém, foram desconsideradas na contagem as linhas referentes ao relacionamento entre entidades em ambos os *templates*, haja vista que essa funcionalidade não foi implementada nos *templates* de granularidade fina. Além disso, foram desconsideradas as linhas de comentários presentes nos arquivos base do JHipster.

Diferente do JHipster, os *templates* de granularidade fina utilizam a abordagem de subdividir as responsabilidades em *workers*. Nesse sentido, foram consideradas as linhas de código dos *templates* principais de *page object* e *spec,* além das linhas de código de todos os *sub-templates* e *designs* relacionados aos *templates* principais na abordagem de granularidade fina. A Figura 5 demonstra um gráfico contendo o número de linhas de código nas abordagens de granularidade fina e granularidade grossa. Os *templates* de *spec* e *page object* do JHipster contém 182 linhas e 137 linhas respectivamente em seus arquivos base. Por outro lado, os utilizados no Potter somam 475 linhas nos arquivos de *spec* e 339 linhas nos arquivos de *page object.* Isso demonstra que a abordagem de granularidade fina tende a ter um alto número de linhas de código, se considerado todos os *templates* e *designs* utilizados. Porém, é importante salientar que, ao considerar somente os *templates* esse número de linhas decresce consideravelmente. Nesse caso, os arquivos de *spec* e *page object* somam respectivamente 110 linhas e 88 linhas.

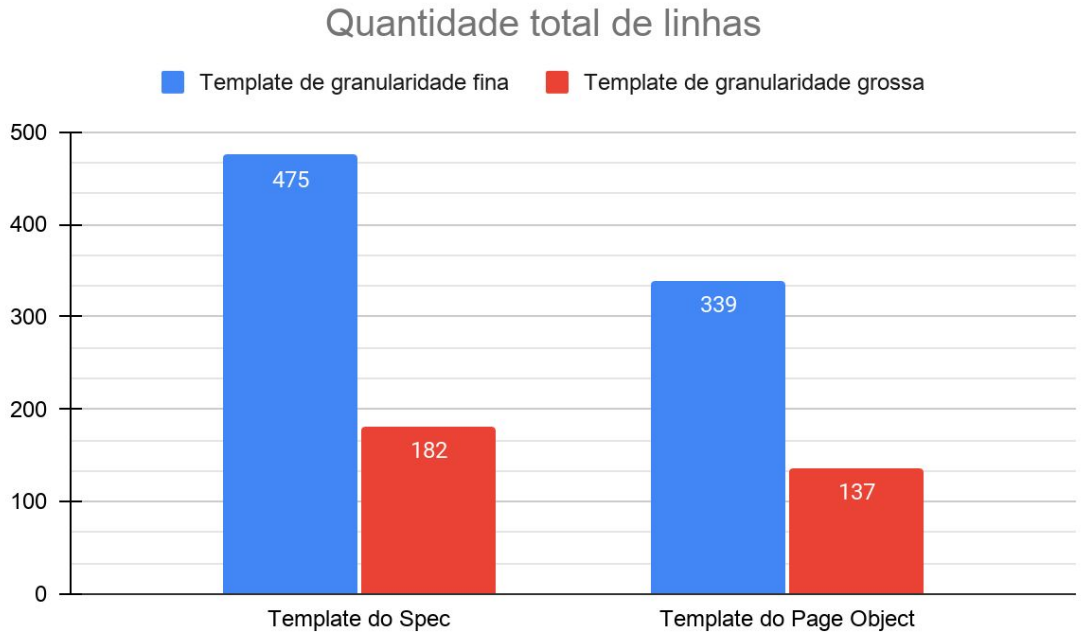

Figura 5. Quantidade total de linhas

Para analisar a quantidade de mudanças de responsabilidades nos templates do JHipster foi necessário destacar nos códigos dos templates de *spec* e *page object* todos os trechos relacionados aos metadados de entidades e propriedades de entidades. A partir disso, foram contadas todas as ocorrências em que havia uma mudança entre metadados de entidades e propriedades de entidades. Nos *templates* de granularidade fina, todos os sub-templates foram considerados na contagem. Nesse sentido, foi possível destacar 39 ocorrências nos templates de *spec* utilizados no Potter e 50 ocorrências no template de *spec* do JHipster. Porém é importante ressaltar que, ao desconsiderar os sub-templates do Potter, há somente 1 ocorrência no template principal de *spec*, sendo esta relacionada a metadados de entidades. Todas as linhas relacionadas a propriedades de entidades foram inseridas nos sub-templates. Nos templates de *page object* utilizados no Potter foram somente 2 ocorrências. Da mesma forma ocorreu no *template* de *page object* do JHipster, sendo somente 2 ocorrências. A Figura 6 demonstra a comparação entre as abordagens.

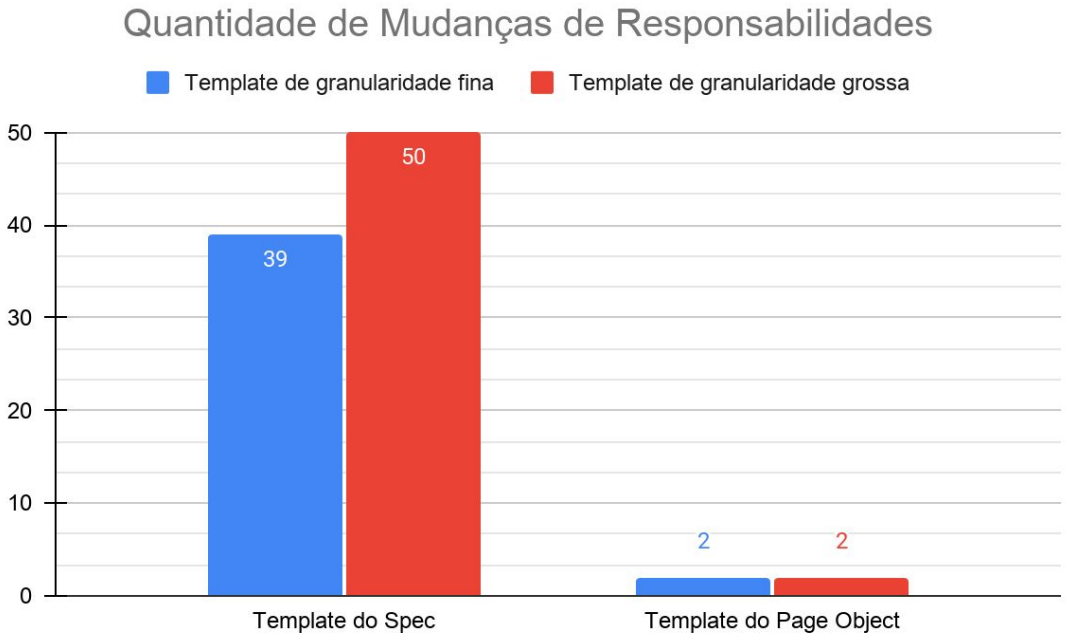

Figura 6. Quantidade de mudanças de responsabilidades

Para os *templates* de *spec* utilizado no Potter foram contados 36 artefatos, sendo 1 *template* principal, 30 *sub-templates* e 5 *designs*. Para os *templates* de *page object* foram contados 12 artefatos, sendo 1 *template* principal, 6 *sub-templates* e 5 *designs*. Já nos *templates* do JHipster foram contados apenas 1 artefato para *spec* e 1 artefato para *page object*. A Figura 7 evidencia a diferença numérica entre as abordagens. Através desses dados é possível evidenciar a granularidade fina na abordagem utilizada pelo Potter.

Ainda foi possível identificar a média de linhas de código por artefatos em ambas as abordagens. No JHipster, por haver somente 1 artefato para spec e *page object* respectivamente, o número de linhas permaneceu o mesmo apresentado na Figura 5. Já os *templates* de granularidade fina obtiveram a média de 13,19 nos arquivos de spec e

28,25 nos arquivos de *page object*. Para alcançar essa média, dividiu-se o número de linhas de código pelo número de artefatos. Isso demonstra que o número de artefatos pode aumentar consideravelmente na abordagem de granularidade fina, porém a média de linhas por artefatos reduz fortemente. A Figura 8 evidencia a diferença entre as abordagens.

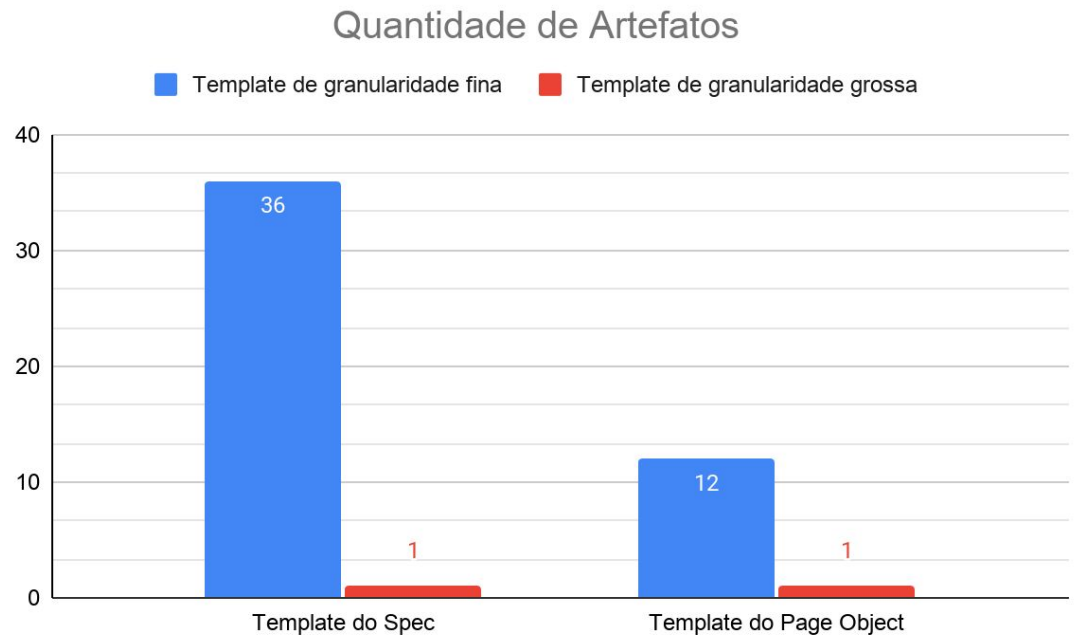

Figura 7. Quantidade de artefatos

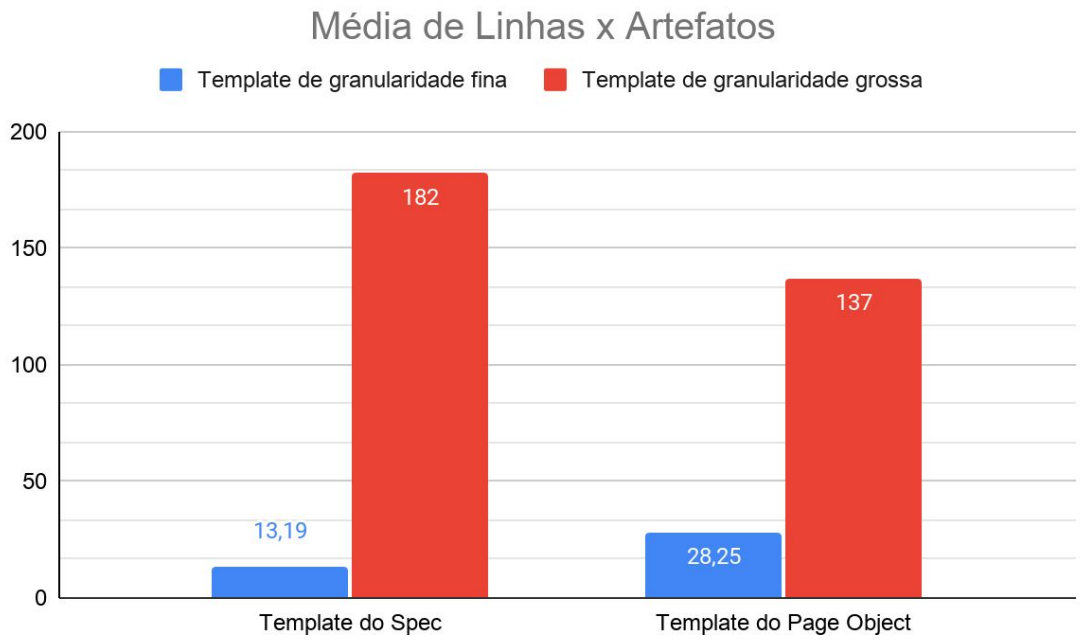

Figura 8. Média de linhas por artefatos

Foi possível ainda evidenciar a média de mudanças de responsabilidades por artefatos. Para isso, foi necessário dividir o número de responsabilidades dos

templates do JHipster e do Potter pelo número de artefatos. Como no caso anterior, por os *templates* do JHipster conterem somente 1 artefato para *spec* e *page object* respectivamente, a média foi o próprio número de mudanças de responsabilidade, sendo 50 para o *spec* e 2 para o *page object*. Já nos *templates* de granularidade fina, foi possível destacar uma média de 1,86 mudanças de responsabilidades por artefatos para o *spec* e 0,16 para o *page object*. Esses dados demonstram que a média de mudanças de responsabilidade tende a ser muito menor nos *templates* de granularidade fina. Sendo assim, é possível destacar que os *templates* de granularidade fina propostos por Vilar et al (2015) demonstram ter melhor encapsulamento em comparação aos *templates* de granularidade grossa utilizados no JHipster, permitindo o melhor reuso de código e a melhor manutenibilidade dos *templates*. A Figura 9 demonstra o gráfico com as médias em escala logarítmica.

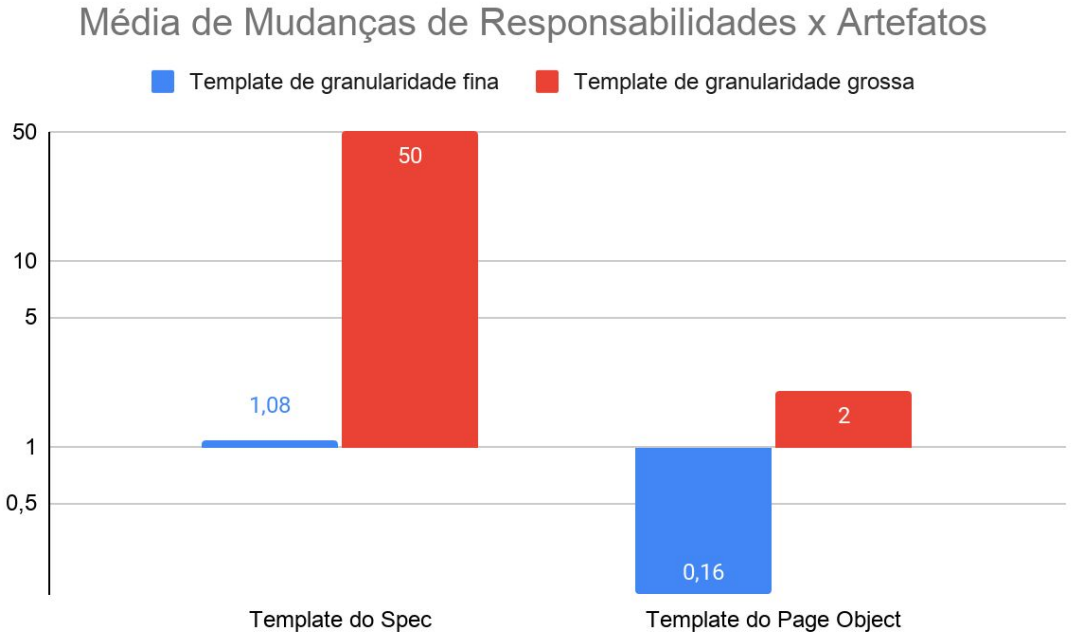

Figura 9. Média de mudanças de responsabilidades por artefatos

Por último, os arquivos finais de spec e *page object*, gerados a partir do JHipster e do Potter foram submetidos ao comparador de códigos Economaster[26] para verificar as diferenças entre eles. Ao desconsiderar as linhas de código referentes ao relacionamento entre entidades e os comentários contidos nos *templates* do JHipster, foi possível notar exatamente o mesmo conteúdo em ambos os arquivos gerados. Dessa forma, evidenciou-se a existência da cobertura igual de casos de teste nos *templates* gerados a partir de ambas as abordagens.

7. Conclusão

Essa pesquisa realizou uma análise comparativa entre as abordagens de granularidade grossa, presentes nos *templates* do gerador de código JHipster e granularidade fina, presentes nos *templates* do gerador de códigos Potter, com o

[26] https://economaster.com.br/todas/web-ferramentas/comparador-de-textos-e-codigos/

objetivo geral de demonstrar qual abordagem possui melhor encapsulamento, além de comparar a quantidade total de linhas de código, a quantidade de mudanças de responsabilidades, quantidade de artefatos, a média de linhas por artefatos e a média de mudanças de responsabilidades por artefatos nos *templates* entre as abordagens.

Através dessa pesquisa foi possível observar indícios de uma solução para o problema da complexidade na geração de código para testes automatizados, melhorando a manutenção de *templates* para geração de código. Haja vista que através dessa abordagem é possível fragmentar as responsabilidades de *templates*, melhorando assim o encapsulamento, permitindo o melhor reuso de código. Mesmo havendo maior número total de linhas de código nos *templates* de granularidade fina em comparação aos de granularidade grossa, na média geral, há uma redução significativa no número de mudanças de responsabilidades nessa abordagem. Sendo assim, pode haver um ganho em tempo de manutenção, reduzindo o custo final para criar e manter *templates*.

Dessa forma, pode-se afirmar que o encapsulamento das responsabilidades em artefatos de granularidade fina permite o reuso de código e melhora a manutenibilidade, pelo menos em termos do tempo gasto para realização de tarefas de customização.

Como trabalhos futuros, propõem-se evoluir os *templates* de granularidade fina para que estes sejam capazes de cobrir todo escopo dos *templates* do JHipster para testes e2e Protractor e criar novos templates para testar funcionalidades não cobertas pelo JHipster, como por exemplo, validações de campos de formulários.

Referências

Bartié, Alexandre (2002), Garantia da qualidade de software: adquirindo maturidade organizacional, Campus.

Delamaro, M. E., Maldonado, J. C., e Jino, M. (2007). Introdução ao Teste de Software. Elsevier

Magno, Danillo Goulart (2015). Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD. Disponível em: <encurtador.com.br/nIP39>

Molinari, Leonardo (2003), Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis, Érica.

Myers (1979), G. The Art of Software Testing. New York: Wiley.

Pressman, Roger S. (2011), Engenharia de software: Uma abordagem profissional, AMGH, 7ª edição.

Vilar, Rodrigo. Oliveira, Delano. Almeida, Hyggo. (2015). Rendering patterns for enterprise applications. Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15). Association for Computing Machinery, New York, NY, USA, Article 22, 1–17.

Vilar, Rodrigo A.V. Decomposição e reúso de componentes baseados em metadados para interfaces gráficas do usuário em aplicações corporativas web. Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, Campina Grande, 2017. Disponível em: <encurtador.com.br/ciBZ4> Acesso em: 21 nov. 2020.

Sakamoto, K., Tomohiro, K., Hamura, D., Washizaki, H., and Fukazawa, Y. (2013). Pogen: A test code generator based on template variable coverage in graybox integration testing for web applications. Fundamental Approaches to Software Engineering

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshy- vanyk, D. (2015). When and why your code starts to smell bad. In 37th Int. Conference on Software Engineering (ICSE), pages 403–414.

Turner D., Park M., Kim J., Chae J (2008). An Automated Test Code Generation Method for Web Applications using Activity Oriented Approach. in 23rd IEEE/ACM International Conference on Automated Software Engineering.