UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE E INOVAÇÃO

FERNANDO HENRIQUE CANTO

Proposta de Método de Reengenharia de Sistemas Legados Desenvolvidos em PHP

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software e Inivação

Orientadora: Prof. Dr. Ingrid Nunes

Porto Alegre
2021

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Canto, Fernando Henrique

Proposta de Método de Reengenharia de Sistemas Legados Desenvolvidos em PHP / Fernando Henrique Canto. – Porto Alegre: PPGC da UFRGS, 2021.

47 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Curso de Especialização em Engenharia de Software e Inovação, Porto Alegre, BR–RS, 2021. Orientadora: Ingrid Nunes.

1. Engenharia de Software. 2. Reengenharia de Sistemas. 3. Refatoração. 4. UFRGS. I. Nunes, Ingrid. II. Título.

RESUMO

Sistemas legados são frequentemente fonte de problemas em diversas instituições, embora tratem-se de recursos valiosos e estratégicos. Tais problemas incluem custos elevados de manutenção e dificuldade de alterações. Eventualmente, a melhor maneira de lidar com tais sistemas é um projeto de reengenharia, que possa reduzir os impactos negativos de um sistema legado que seja de difícil evolução. Há evidências que sugerem que tais processos são vantajosos, porém eles podem ser complexos, custosos e propensos a erros. No contexto da Universidade Federal do Rio Grande do Sul (UFRGS), já foram realizados alguns projetos de reengenharia de sistemas legados desenvolvidos na linguagem PHP. Porém, observa-se a falta de um processo consolidado de reengenharia, que não só seja adaptado a realidade dos sistemas da universidade, mas que seja capaz de mitigar problemas observados em tais projetos. O objetivo deste trabalho é propor e validar um processo de reengenharia gradual de sistemas PHP, a ser realizado em duas etapas: a primeira etapa realiza apenas uma melhoria interna do código-fonte das aplicações, sem nenhuma repercussão para o usuário final, enquanto a segunda etapa incorpora melhorias de interface e interação do usuário, construídas em cima das melhorias produzidas na primeira etapa. Tal processo foi aplicado em dois módulos de sistemas da UFRGS. Foram produzidas novas versões desses módulos, incorporando novos recursos e práticas de desenvolvimento adotados pela UFRGS ao longo dos anos, e cujo comportamento é idêntico ao dos sistemas originais. O processo reduziu a incidência de erros durante a reescrita dos sistemas e aumentou o nível de confiança no resultado obtido, e permitiu a definição de prioridades claras em cada uma das duas etapas.

Palavras-chave: Engenharia de Software. Reengenharia de Sistemas. Refatoração. UFRGS.

A Proposal of a Method for Modernization of Legacy Systems Developed in PHP

ABSTRACT

Legacy systems are frequent sources of problems in many institutions, though they're valuable and strategic resources. Such problems include high maintenance costs and difficulty in making changes. Eventually, the best way to deal with such systems is a modernization project, which can reduce the negative effects of a legacy system with difficult maintenance. There's evidence that suggests such projects are advantageous, but they can be complex, costly and error prone. Within the Federal University of Rio Grande do Sul (UFRGS), there have been some modernization projects for its legacy systems developed in the PHP language. However, there's a lack of a well established modernization process, which is not only adapted to the university's systems, but is able to minimize the problems observed in such past projects. This goal of this study is to propose and validate a process of gradual modernization of PHP systems, to be done in two steps: the first step produces only an improvement in the internal quality of the source code of the applications, with no visible effects for the end user, while the second step produces interface and user experience improvements, which are built on top of the improvements made in the first step. This process was applied to two software modules in UFRGS. New versions of those modules were produced, incorporating new tools and development practices adopted by UFRGS through the years, and whose behavior is identical to the original modules. The process reduced the number of errors during the code rewrite and increased the level of confidence in the final result, and allowed the definition of clear priorities in each of its two steps.

Keywords: Software engineering, Software modernization, refactoring, UFRGS.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas legados são uma realidade observada no mundo do desenvolvimento de software em geral. Tanto a academia quanto as empresas vêm investigando os desafios e riscos de se lidar com sistemas de informação desenvolvidos em tecnologias antigas e metodologias inadequadas de desenvolvimento. Em particular, a manutenção desses sistemas tende a tornar-se cada vez mais difícil e custosa (BRODIE; STONEBRAKER, 1993). Uma das maneiras de se atacar esse problema é a reengenharia, que busca eliminar a degradação gradual da qualidade do código-fonte, tornando a manutenção mais fácil e menos custosa (KHADKA et al., 2014).

A reengenharia de um sistema, porém, apresenta uma série de desafios, relacionada em grande parte com características desses sistemas antigos como a ausência de documentação, a dificuldade de extrair regras de negócio do código-fonte, e a não conformidade com a arquitetura planejada (KHADKA et al., 2014). Para isso, é necessário ter à disposição um processo bem definido, que permita lidar com essas dificuldades de maneira eficaz, e produzir um novo sistema que reproduza exatamente o comportamento do sistema original, utilizando arquiteturas, plataformas e metodologias de desenvolvimento da melhor forma possível.

Na Universidade Federal do Rio Grande do Sul (UFRGS), esse problema é encontrado na extensa base de código PHP de seus sistemas Web. O surgimento de novas tecnologias e práticas de desenvolvimento permitem a produção de sistemas mais robustos e flexíveis a mudanças. Assim, este trabalho apresenta a proposta e validação de um processo de reengenharia desses sistemas antigos desenvolvidos em PHP, que permite a evolução do backend dos sistemas de forma gradual. Tal processo torna a manutenção e alteração desses sistemas mais ágil e confiável, e também permite a melhoria das interfaces e da experiência do usuário, garantindo que as funcionalidades do sistema permaneçam inalteradas.

O Capítulo 2 deste documento apresenta um embasamento teórico para as práticas apresentadas neste trabalho. O Capítulo 3 apresenta o processo de reengenharia proposto, detalhando e justificando seus passos. O Capítulo 4 detalha a aplicação desse processo em dois módulos legados dos sistemas da UFRGS. No Capítulo 5, são discutidos os resultados da experiência da aplicação do processo proposto. Por fim, no Capítulo 6, são apresentadas as conclusões obtidas a partir da realização deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta resultados de pesquisas sobre o tópico de reengenharia, de forma a embasar e justificar as escolhas feitas neste trabalho. Em primeiro lugar, são apresentados trabalhos sobre o processo de reengenharia em si (Seção 2.1). Em seguida, são citados alguns exemplos conhecidos de tal processo (Seção 2.2). Por fim, o capítulo descreve a situação encontrada na UFRGS, de forma a contextualizar e explicar a elaboração deste trabalho (Seção 2.3).

2.1 Processos e Métodos de Reengenharia

Segundo Khadka et al. (2014), a reengenharia de sistemas legados, ou modernização de software, consiste no processo de evolução de software pela substituição, reescrita, reuso ou migração de componentes ou plataformas de sistema, aplicadas quando os processos tradicionais de manutenção não atingem as qualidades de software desejadas. Segundo Brodie and Stonebraker (1993), sistemas legados costumam ter manutenção cara, e são inflexíveis e frágeis. O objetivo principal da reengenharia seria que esses sistemas legados não se tornem legados.

Conforme apontado por Khadka et al. (2014), sistemas legados não consistem em sistemas obsoletos, mas são vistos como úteis e críticos para o negócio, e utilizados frequentemente. Esse é seguramente o caso da UFRGS, onde tais sistemas legados atendem a demandas cruciais da comunidade acadêmica em seus três eixos (ensino, pesquisa e extensão), e é justamente por essa criticidade que esses sistemas precisam ser robustos e flexíveis a mudanças. Segundo Khadka et al. (2015), o aumento da flexibilidade e manutenibilidade de sistemas legados pode ser observado em casos reais de reengenharia.

Uma das preocupações deste trabalho é permitir que a modernização de um sistema seja feita de forma gradual. Brodie and Stonebraker (1993) citam duas estratégias para esse processo: *"Cold Turkey"*, na qual o sistema é refeito inteiramente do zero, com alto risco associado, e *"Chicken Little"*, na qual pequenas alterações são feitas gradualmente. Hasselbring et al. (2004) e Knoche and Hasselbring (2018) também indicam uma preferência por processos graduais de reengenharia.

Existe uma diversidade de métodos de reengenharia propostos na literatura. Deve-se considerar sempre o contexto e a realidade para as quais cada método é desenvolvido, e qual seu principal objetivo. Ainda assim, observa-se certas práticas comuns nos projetos

de refatoração. Tanto Santos et al. (2018) quanto Bruneliere et al. (2015) destacam três fases distintas no processo: engenharia reversa, para a compreensão dos requisitos do sistema original; reestruturação dos modelos obtidos para a nova plataforma; geração de código a partir dos novos modelos.

O padrão Dublo, apresentado por Hasselbring et al. (2004), visa transformar sistemas de camada única em sistemas multicamadas através da inserção de uma camada intermediária entre a apresentação e a lógica de negócios. Essa camada inicialmente comunica-se com o código legado através de adaptadores, de forma que a lógica de negócios pode ser gradualmente migrada para dentro da nova camada. Nota-se que essa proposta permite que o banco de dados permaneça o mesmo durante a migração.

Wendland et al. (2013) apresenta um método de reengenharia baseado em testes. Inicialmente, um processo de engenharia reversa é utilizado no sistema a ser migrado a fim de obter uma modelagem dos requisitos do sistema, e os testes são derivados destes modelos. No processo proposto neste trabalho, os testes são gerados diretamente da engenharia reversa do sistema. Assim, pode-se considerar que os testes em si são a própria modelagem, e permitem que o processo de migração produza um sistema com comportamento idêntico ao original. Uma decisão importante citada no trabalho de Wendland et al. (2013) é em qual o nível de abstração esses testes irão operar. Existem vantagens e desvantagens tanto na escolha de um nível alto de abstração quando em um nível baixo.

2.2 Exemplos Conhecidos de Reengenharia

Além do estudo teórico sobre o processo de reengenharia, também é útil conhecer casos concretos de reengenharia de software, para aprender com seus resultados e dificuldades.

Na literatura disponível, encontramos exemplos de casos bem sucedidos de modernização de sistemas legados, como o da empresa de vestuário Carhartt[1], a empresa de telefonia Claro[2], a empresa de data centers Equinix[3], a empresa financeira Goldman Sachs[4], a empresa de serviços aéreos Quantum Aviation Solutions[5] (GEORGIOU, 2019), a empresa de mídias digitais Ask Media Group[6] e a empresa automotora Jaguar Land

[1] www.carhartt.com
[2] www.claro.com.br
[3] www.equinix.se
[4] www.goldmansachs.com
[5] quantum-aviation.com
[6] www.askmediagroup.com

Rover[7] (BUCHANAN, 2019). Khadka et al. (2015) apresenta cinco casos anônimos de reengenharia de sistemas legados. Dentre os benefícios observados pelas empresas, observa-se melhoria na usabilidade, maior manutenibilidade e flexibilidade e redução de custos de manutenção.

Em contrapartida, existem casos documentados de projetos de reengenharia que não foram bem sucedidos. Um desses casos é o do Netscape 6.0, que passou por uma reescrita completa do código-fonte, depois que a versão 5.0 foi cancelada (CAUDILL, 2019). O projeto demorou dois anos, e, quanto o produto foi lançado, ele apresentava problemas de performance, e alguns recursos estavam faltando. Isso acabou levando ao fim do Netscape, e o projeto foi considerado desastroso.

2.3 Sistema de Software da UFRGS

A elaboração de um processo de reengenharia, além de levar em conta a literatura disponível e casos concretos, deve considerar o contexto específico de onde ele vai ser aplicado. Certas particularidades e idiossincrasias dos sistemas legados devem ser estudados e compreendidos, para que o processo seja o mais eficaz possível.

A UFRGS tem um catálogo extenso de sistemas e módulos desenvolvidos em PHP. O PHP foi adotado em meados dos anos 2000 como plataforma institucional de sistemas Web, substituindo a plataforma ASP. Vários dos sistemas em PHP são migrações de sistemas ASP, e também de sistemas ainda mais antigos, desenvolvidos na plataforma cliente-servidor Delphi.

Os sistemas PHP legados da UFRGS têm uma estrutura essencialmente monolítica, consistindo de amálgamas de PHP bruto, SQL, HTML e JavaScript, com alto índice de redundâncias e duplicação de código, alto acoplamento, baixa coesão e baixa modularidade. Isso se deve tanto a características das primeiras versões da linguagem PHP, quanto à inexperiência dos desenvolvedores. Com a constante necessidade de alterações e desenvolvimento de novas funcionalidades, esses sistemas foram acumulando débito técnico (FOWLER, 2019), e quaisquer atividades de manutenção e alteração tendem a ser difíceis, longas e complexas. Dessa forma, os sistemas encontram-se frágeis e inflexíveis. Porém, por se tratarem de sistemas críticos da universidade, tais mudanças costumam ser atividades prioritárias.

[7] www.jaguarlandrover.com

Em meados dos anos 2010, a UFRGS adotou o *Yii Framework*[8] como plataforma institucional de desenvolvimento. Isso permitiu a adoção de melhores práticas de desenvolvimento e componentes reutilizáveis, tornando os sistemas mais robustos, e com manutenção mais fácil. Isso tornou atrativa a ideia de modernizar os sistemas PHP legados utilizando a nova plataforma, e, de fato, alguns projetos de reengenharia foram realizados. Porém, não foi utilizado nenhum processo formal, e a reengenharia desses sistemas envolvia a cópia direta de muito desse código legado, sem a aplicação das melhores práticas do *Yii Framework*, como a arquitetura *Model-View-Controller* (MVC) (LEFF; RAYFIELD, 2001) com a devida separação de responsabilidades, e o reuso de código. Além disso, o desenvolvimento dependia de testes manuais, que é um procedimento longo, tedioso e propenso a erros.

É importante ressaltar que, de modo geral, há dois objetivos principais a serem obtidos quando se fala em modernização de sistemas da UFRGS. O primeiro objetivo é a melhoria interna do código-fonte em si, a fim de simplificar e agilizar as tarefas de manutenção e alteração. Idealmente, essas alterações devem ser transparentes ao usuário final. O segundo objetivo é a melhoria da usabilidade do sistema, com alterações nas interfaces visíveis aos usuários. Ambos benefícios foram observados no trabalho de Khadka et al. (2015).

Diante disso, viu-se a necessidade de adotar um processo de reengenharia de sistemas, que leve em conta a realidade dos sistemas PHP da universidade e das ferramentas e práticas utilizadas, e permita a melhoria gradual desses sistemas sem comprometer o seu funcionamento no dia-a-dia da universidade.

2.4 Considerações Finais

Os trabalhos apresentados neste capítulo, juntamente com a análise do contexto da universidade, ajudam a definir algumas das escolhas feitas neste trabalho (por exemplo, a reengenharia baseada em testes, e a migração gradual de funcionalidades). Além disso, os casos reais estudados demonstram que existem vantagens a serem obtidas com a definição de um projeto de reengenharia, o qual será exposto e explicado no capítulo seguinte.

[8] www.yiiframework.com

3 PROCESSO DE REENGENHARIA

A partir dos dados levantados no capítulo anterior, este capítulo apresenta a proposta do processo de reengenharia de sistemas, explicando e justificando as escolhas realizadas. Em primeiro lugar, é apresentada uma visão geral do processo (Seção 3.1), em uma descrição de seus passos (Seção 3.2). Em seguida, cada passo de cada fase do processo é explicada individualmente, dando uma visão aprofundada do processo do início ao fim (Seções 3.3 e 3.4).

3.1 Visão Geral do Processo

O processo aqui proposto tem, como objetivo principal, fornecer uma migração de sistemas legados PHP da UFRGS em duas fases. A primeira fase consiste de uma migração gradual do *backend* da aplicação, transferindo a base de código para o Yii Framework, sem afetar a interação do usuário com o sistema. Esta fase permite que o sistema seja refatorado, de forma a aumentar a modularização e a separação de responsabilidades, aproveitando ao máximo os recursos da nova plataforma. O resultado desta fase é chamado neste trabalho de sistema intermediário.

A segunda fase consiste na remodelagem das interfaces da aplicação, para melhoria da usabilidade e adequação aos padrões da universidade e de necessidades de acessibilidade. Esta fase se utiliza dos artefatos produzidos e das melhorias obtidas na primeira fase. O resultado desta fase é chamado de sistema final.

A divisão em duas fases permite que a reengenharia seja gradual, com cada fase focada em um aspecto especídifo. Além disso, a primeira fase em si também é gradual, pois as funcionalidades do sistema original podem ser substituídas uma a uma pelo sistema intermediário. Outra característica é que a base de dados dos sistemas não são afetadas. Isso é importante, pois a UFRGS utiliza uma base de dados institucional, compartilhada por diversos sitemas, e não há interesse no momento em alterar esse molde.

A reengenharia se baseia fortemente no desenvolvimento de testes automatizados, escritos na ferramenta Codeception[1]. O uso de testes automatizados facilita a garantia da preservação do comportamento do código após a realização de modificações, dando confiança ao processo de reengenharia e refatorações. O Codeception permite o desenvolvimento de testes funcionais, que são executados em um navegador simulado no próprio

[1] codeception.com

servidor Web, reproduzindo a interação de um usuário. O uso dos recursos e das estruturas do Codeception permitirá o reuso dos testes em ambas as fases da reengenharia. Foi utilizada a versão 2.5.6 do Codeception, por ela ser compatível com a versão 1.1 do Yii Framework. Essa compatibilidade foi removida na versão 3 (CODECEPTION. . . , 2021).

3.2 Descrição do Processo

Além da compreensão geral das duas fases do processo, é necessário explicar cada um dos passos de cada fase.

A primeira fase do processo é repetida para cada script PHP do sistema legado que é acessado pelo navegador. Via de regra, cada script representa uma unidade distinta dentro do sistema, podendo ser uma página de visualização de dados, um formulário, ou o processamento de um formulário gerado em outro script. É importante notar que alguns scripts consistem em coleções de funções ou classes, ou cabeçalhos e rodapés. Esses scripts ficam de fora do processo.

Assim, para cada página, realiza-se este procedimento:

1. Engenharia reversa da página, com mapeamento de todas as suas entradas;
2. Geração de testes funcionais que cubram o máximo possível do código do script, com geração de *fixtures*, e separação entre o script de teste, o *actor* e os *page objects*;
3. Exportação dos testes e fixtures para o local onde será desenvolvido o sistema intermediário, com as adaptações necessárias dos scripts de apoio;
4. Implementação do script no sistema intermediário, de forma que todos os testes sejam satisfeitos;
5. Substituição do script original por uma chamada ao sistema intermediário.

Na medida em que os scripts são migrados, pode ser necessário refatorar as classes do sistema intermediário. Os testes devem garantir a integridade do sistema a cada refatoração.

Para a segunda fase do processo, para cada página do sistema, os seguintes passos são realizados:

1. Exportação dos testes e fixtures para o local onde será desenvolvido o sistema final;
2. Adaptação do *actor* e dos *page objects* para a nova interface;

3. Implementação do script do sistema final, de forma que todos os testes sejam satisfeitos.

A substituição do sistema intermediário pelo sistema final fica a critério da equipe de desenvolvimento, pois pode ser preferível substituir todas as interfaces de uma vez, a fim de manter a consistência.

3.3 Descrição dos Passos da Primeira Fase

Esta seção descreve cado passo da primeira fase do processo. A execução desta fase em um script do sistema legado resultará em sua migração para o sistema intermediário.

3.3.1 Engenharia Reversa da Página

O processo começa com o estudo e a compreensão do funcionamento do script, através da engenharia reversa. Este passo inclui o estudo do código-fonte em si, e testes manuais executados em um navegador.

Um dos objetivos principais deste passo é identificar os diferentes parâmetros de entrada do script, bem como sua influência sobre a execução do código. A análise dos sistemas da UFRGS revela que as entradas de um script PHP concentram-se em quatro áreas: variáveis de sessão, variáveis Get, variáveis Post, e dados oriundos do banco de dados.

Variáveis de sessão podem ser criadas e lidas em qualquer página PHP, e seus valores são específicos a cada instância de usuário autenticado no servidor. Usualmente as variáveis de sessão armazenam dados que identificam o usuário (no caso da UFRGS, o código do cartão, por exemplo), mas também acabam sendo usadas para a passagem de valores entre diferentes páginas (embora seja recomendado o uso de Post e Get para esse caso). Já as variáveis Get e Post são recebidas diretamente do navegador do usuário. As variáveis Get são passadas na URL, enquanto as variáveis Post vêm no corpo da requisição, e costumam ser passadas por meio de um formulário HTML.

É necessário ser minucioso na busca da utilização desses valores de entrada. Na linguagem PHP, esses valores encontram-se nas variáveis $_SESSION, $_GET e $_POST, que são consideradas variáveis “superglobais”, ou seja, acessíveis de qualquer ponto do

sistema. Ou seja, é possível encontrar casos de dependência implícita no código, onde um valor de sessão, Get ou Post é lido dentro de uma função, ou dentro de um método de uma classe, e é impossível identificar essa dependência sem olhar o código-fonte de tal função ou método.

Por fim, existem os valores oriundos do banco de dados, que são acessados através de consultas. Nos sistemas legados da UFRGS, tais consultas são escritas em SQL, e é necessário analisar o conteúdo dessas consultas para identificar as tabelas e colunas utilizadas na execução do script, bem como o modo que esses dados são utilizados: por exemplo, uma consulta pode gerar registros que são iterados para a exibição de listas ou tabelas, enquanto outra retorna valores usados em cláusulas condicionais.

Outro objetivo da engenharia reversa é entender o resultado da execução da página. De forma geral, um script PHP pode gerar uma página HTML, que é exibida no navegador do usuário, ou pode realizar alterações no banco de dados. Note-se que essa divisão não é absoluta: é comum que páginas que alteram o banco de dados apresentem uma página HTML com uma mensagem de sucesso ou de erro para o usuário. Também há outras ações possíveis, como alterações em variáveis de sessão e disparos de e-mail. É importante mapear tais ações e como elas se relacionam aos valores de entrada, pois isso será a base para a geração dos testes no passo seguinte.

3.3.2 Geração de Testes Automatizados

A partir da engenharia reversa do script, com o mapeamento de suas entradas e compreensão de suas saídas, o passo seguinte é a implementação de uma suíte de testes automatizados, que cubram o máximo possível de situações previstas no código. Este passo não necessariamente precisa ser feito após o término do primeiro passo; em muitos casos, pode ser desejável realizar os dois passos de forma concomitante, usando a elaboração dos próprios testes como maneira de compreender o funcionamento do código.

Os testes são elaborados na ferramenta Codeception, desenvolvida especialmente para teste de aplicações PHP. O Codeception inclui três níveis de teste: testes unitários, que executam métodos internos do sistema, verificando seu retorno; testes funcionais, que executam chamadas HTTP ao sistema, simulando o comportamento de um navegador Web; e testes de aceitação, que utilizam a ferramenta Selenium para instanciar um navegador Web e executar a aplicação. Para este processo de reengenharia, os testes unitários não são adequados, em parte porque os sistemas legados não são desenvolvidos no

paradigma orientado a objetos, e, mesmo que fossem, a arquitetura interna do sistema não seria mantida. No momento da elaboração deste trabalho, a ferramenta Selenium ainda não está consolidada na UFRGS. Por isso, este passo utiliza a elaboração de testes funcionais.

O objetivo deste trabalho não é definir como os testes são gerados, mas é importante especificar a maneira como eles são implementados na ferramenta, pois isso é crucial para a segunda fase do processo.

Normalmente, os testes funcionais são escritos com uma sintaxe bastante simplificada, utilizando verbos que tentam reproduzir ações e reações de um usuário hipotético. Porém, este método de escrever testes funcionais não é adequado para nenhuma das duas fases da reengenharia. Na primeira, é necessário garantir que o sistema intermediário reproduza exatamente o comportamento do sistema original. Já na segunda, é necessário que os testes sejam flexíveis o suficiente para que as interfaces sejam refeitas e aprimoradas. Isso é obtido separando cada teste em três camadas.

A camada mais externa e mais abstrata do teste consiste nos métodos de teste em si, que são sequências de verbos ordenados, de forma semelhante à sintaxe original do Codeception, mas com a diferença de que esses verbos refletem especificamente as regras de negócio da página sendo testada. Ou seja, em vez do teste verificar a presença de uma string em um determinado elemento HTML, o teste verifica, por exemplo, o título de uma atividade de extensão em uma determinada posição de uma lista (ver Figura 3.1). Isso é específico o suficiente para satisfazer a primeira fase do processo, e flexível para a segunda fase. É importante que o nome de cada verbo seja claro e preciso, para facilitar a compreensão de seu propósito ao longo do processo.

Cada verbo da camada anterior é implementado na segunda camada, que consiste em uma classe *Actor*. O Codeception inclui uma classe *Actor* diferente para cada nível de teste (unitário, funcional e de aceitação), e cada uma pode ser estendida, com a inclusão de métodos novos. Na execução de cada método de teste, o Codeception detecta qual *Actor* deve ser passado como parâmetro, e seus métodos tornam-se disponíveis dentro do teste.

Dentro do *Actor*, os verbos do método de teste serão implementados com maior nível de especificidade, e é possível que um único verbo seja desmembrado em mais de uma ação ou verificação. Por exemplo, um verbo que verifica a existência de uma lista com N itens pode verificar, além do número de itens, a existência de um título para a lista e um botão para habilitar sua exibição. Também é importante que cada verbo receba o

Figura 3.1: Exemplo de classe de teste

```php
public function testAcaoEmAnalise(SolVisualizacaoTester $eu)
{
    $eu->estouLogadoComo(10, 1);

    $eu->acessoVisualizacao(101, 1);

    $pageObject = new SolVisualizacaoPageObject($eu);

    $eu->vejoDadosAcao($pageObject, "Coordenador Numero Zero", 10, 101, "Ação com solicitação em análise");

    $eu->vejoRecursoQuantidade($pageObject, 1, "Bolsa-Evento", 3);
    $eu->vejoPessoaEspecificacaoRecurso(
        $pageObject,
        1,
        null,
        "Especifique o evento, o período e o número de bolsas necessárias (máximo de 3 bolsas por pedido, por um "
            . "período de até 2 meses cada, em um total de até 10 bolsas por ano)",
        "Obs obs obs"
    );
    $eu->vejoNumDocumentos($pageObject, 1, 0);

    $eu->vejoRecursoValorSolicitado($pageObject, 2, "Espaço Físico - Salão de Atos", "12,34");
    $eu->vejoPessoaEspecificacaoRecurso(
        $pageObject,
        2,
        null,
        "Especifique Atividade e Evento (isenção ou redução das taxas de manutenção)",
        "Obs recurso espaço físico"
    );
    $eu->vejoNumDocumentos($pageObject, 2, 1);
    $eu->vejoDocumentoRecurso($pageObject, 2, 1, "Arquivo do recurso 2", "chave1");
```

Fonte: O Autor

mínimo possível de parâmetros, e que estes sejam relacionados às regras de negócio; por exemplo, um verbo que testa a existência de um link para visualizar um registro não deve receber a URL completa do link, e sim apenas os valores usados na geração do link, neste caso, a chave primária do registro em questão.a verbo seja claro e preciso, para facilitar a compreensão de seu propósito ao longo do processo.

O Codeception permite que qualquer verificação possua uma mensagem customizada, caso a verificação falhe. É vantajoso utilizar esse recurso para melhorar a inteligibilidade de cada erro possível, melhorando não só a compreensão do código de teste, mas facilitando os próximos passos do processo.

Para que a implementação desses verbos seja específica, é necessária a referência a elementos HTML da página. Essas referências devem ficar restritas à terceira camada do teste, que consiste nos *page objects*. *Page objects* são um padrão de desenvolvimento de teste, que consistem em objetos que interpretam o código HTML de uma página e extraem as informações que são usadas pelas camadas anteriores de teste (FOWLER, 2013). Isso permite que o *Actor* fique desacoplado da sintaxe HTML da página, e fique focado apenas nas informações contidas nela.

Durante o desenvolvimento dos testes, é importante manter em mente que essas três camadas serão mantidas idênticas no desenvolvimento do sistema intermediário.

Figura 3.2: Exemplo de classe Actor

```
public function vejoNumRecursos(SolVisualizacaoPageObject $pageObject, $numRecursos)
{
    $this->assertEquals(
        $numRecursos,
        $pageObject->getNumRecursos(),
        "Eram esperados " . $numRecursos . " recursos, mas foram obtidos " . $pageObject->getNumRecursos()
    );
}

public function vejoRecursoQuantidade(SolVisualizacaoPageObject $pageObject, $numRecurso, $nome, $quantidade)
{
    $this->assertEquals(
        $nome,
        $pageObject->getConteudoNomeRecurso($numRecurso),
        "Era esperado o nome " . $nome . " para o recurso " . $numRecurso . ", mas foi obtido "
            . $pageObject->getConteudoNomeRecurso($numRecurso)
    );
    $this->assertEquals(
        "Quantidade: " . $quantidade,
        $pageObject->getConteudoValorRecurso($numRecurso),
        "Era esperada a quantidade " . $quantidade . " para o recurso " . $numRecurso . ", mas foi obtido "
            . $pageObject->getConteudoValorRecurso($numRecurso)
    );
}
```

Fonte: O Autor

Portanto, quaisquer aspectos dos testes que precisarão sofrer modificações deverão ser mantidos isolados em classes externas. Existem duas maneiras principais de fazer isso: a primeira é implementar métodos na classe *FunctionalTester*, os quais podem ser invocados dentro dos testes. A outra é desenvolvendo módulos, que são classes separadas, cujos métodos são incorporados pelo *Actor*.

Um desses aspectos que mudam é a carga de *fixtures* no banco de dados de testes. O Yii Framework conta com o seu próprio mecanismo de *fixtures*, mas, para as aplicações legadas, foi necessário implementar um mecanismo customizado, com as classes de conexão ao banco já previamente existentes. Esses métodos de carga e limpeza de tabelas são mantidos em um módulo chamado *DataHelper*. Outro aspecto são as URLs do sistema, necessárias tanto para acessar as páginas durante os testes quanto para verificar os endereços dentro dos links. Os métodos que retornam as URLs também podem ser mantidos em um módulo separado, ou no próprio *FunctionalTester*.

3.3.3 Exportação dos Testes

Quando a geração dos testes automatizados for considerada satisfatória, os scripts que contém as três camadas de testes, bem como os arquivos de fixtures, devem ser copiados para o diretório onde fica o projeto com o sistema intermediário. É essencial garantir que o conteúdo desses arquivos seja mantido idêntico durante todo o processo, pois isso

é crucial para que o sistema intermediário comporte-se de maneira idêntica ao sistema original. Qualquer mudança que seja eventualmente realizada nos testes do sistema original deverão ser transferidas para os testes do sistema intermediário e vice versa. Apenas as classes de apoio, como *FunctionalTester* e os módulos, precisarão ser devidamente adaptadas para garantir a integração dos testes com o ambiente do Yii Framework.

Naturalmente, neste momento, todos os testes falharão quando executados, pois o sistema ainda não existe. Porém, é recomendável a execução desses testes, para certificar que não existem erros no scripts de teste.

3.3.4 Implementação do Script no Sistema Intermediário

Uma vez que os testes estão devidamente implantados no sistema intermediário, a implementação da página propriamente dita pode começar. Devido ao fato de que o sistema deve reproduzir exatamente o comportamento do sistema original, é recomendado que o código HTML da página original seja preservado sem alterações. Porém, devido à mistura de código HTML com PHP nas aplicações legadas, muitas vezes pode ser delicado dissociar a lógica de apresentação da lógica de negócio.

Há dois objetivos fundamentais neste passo: em primeiro lugar, o código desenvolvido deve satisfazer plenamente todos os testes gerados na segunda fase. Para isso, é recomendado que o sistema seja implementado aos poucos, rodando os testes com frequência, para validar cada passo dado. É importante ressaltar que os testes funcionais executam um script PHP inteiro e verificam integralmente a página resultante (ao contrário de testes unitários, que testam pequenas unidades do sistema, de forma mais granular). Isso significa que os testes nesta fase resultarão em erro durante a maior parte do desenvolvimento da página. Portanto, o uso de mensagens de erro customizadas torna-se bastante útil, pois permite ao desenvolvedor acompanhar melhor o andamento do processo.

O segundo objetivo é que o código desenvolvido siga ao máximo as boas práticas da engenharia de software e do Yii Framework, incluindo a separação de responsabilidades entre as diferentes camadas, o baixo acoplamento entre as classes, a alta modularização e baixa repetição de código. Embora o sistema intermediário seja eventualmente suplantado pelo sistema final, o objetivo do desenvolvedor é implementar o sistema de forma que o máximo possível de código seja reusado na segunda fase.

É importante destacar que o código gerado pode ser aprimorado gradualmente com refatorações, e os testes gerados no segundo passo servem para validar cada uma

das melhorias. Além disso, o desenvolvedor deve considerar a geração de testes unitários para os métodos criados, adicionando assim uma camada adicional de testes ao sistema (lembrando que esses métodos serão reaproveitados no sistema final, e, portanto, os testes não serão descartados).

Uma vez que todos os testes foram satisfeitos, é recomendável a realização de alguns testes manuais, para detectar de problemas que não foram cobertos nos passos anteriores. Sempre que for necessário realizar alguma alteração nos testes, é necessário manter todas as alterações sincronizadas no sistema original e no intermediário, rodando os testes em ambos para verificar a correção e a integridade do processo.

3.3.5 Substituição do Script Original

Uma vez que a página está validada e de acordo com os testes, ela pode então substituir o script do sistema original. A maneira mais simples de realizar isso é alterar os links que apontam para o script original. No caso do script original não receber nenhum parâmetro Post, pode ser incluída uma redireção HTTP para o sistema intermediário.

Como este processo prevê a reengenharia gradual do sistema, realizada página por página, estes passos serão repetidos, nesta mesma ordem, para todas as páginas subsequentes do sistema original. Assim, os links também precisam ser atualizados com cada script migrado, e isso impacta nos testes. Por isso, é importante que as URLs utilizadas nos testes sejam geradas por classes de apoio, e não pelos métodos de teste, *actors* ou *page objects*.

Com a geração de testes para as outras páginas do sistema, pode ser necessário refatorar as classes de teste e as classes de apoio, para evitar redundâncias e inconsistências no código. É sempre importante garantir que essas mudanças sejam reproduzidas exatamente no sistema intermediário. Além disso, após tais alterações, é importante executar os testes novamente, tanto no sistema original quanto no intermediário, pois erros não detectados podem prejudicar a geração de novos testes.

3.4 Descrição dos Passos da Segunda Fase

Esta seção descreve cada um dos passos da segunda fase do processo. A execução desta fase resulta na migração de um script do sistema intermediário para o sistema final.

3.4.1 Exportação dos Testes e Fixtures

Com a consolidação do sistema intermediário, a migração para o sistema final pode ser iniciada. Ela também consiste na reengenharia de uma página por vez, de forma gradual, embora o sistema final provavelmente será implantado todo de uma vez. Cada migração começa pela exportação dos arquivos de teste e fixtures do sistema intermediário para o final, com as devidas adaptações nas URLs.

3.4.2 Adaptação do Actor e dos Page Objects

Conforme mencionado anteriormente, o script de teste em si não é alterado. Isso é fundamental para a execução correta desta fase, pois é isso que garante a correção semântica do sistema. Apenas o *actor* e os *page objects* são alterados, para adequá-los aos novos padrões de interface do sistema. Idealmente, a maior parte das alterações ocorrerá nos *page objects*, mas isso depende da profundidade das mudanças sofridas pela interface.

É importante notar aqui que há uma grande dificuldade em adaptar os *page objects* para ficarem de acordo com uma página HTML inexistente. Nesta etapa, a funcionalidade ainda não está implementada no sistema final, portanto, a página não existe. Para esse problema, são propostas duas soluções possíveis.

A primeira solução é implantar, provisoriamente, páginas estáticas para um ou mais dos testes, com as mesmas informações geradas no sistema intermediário, mas dentro do novo padrão de interface. Desta forma, cada teste enxergará uma página idêntica a que será eventualmente gerada pelo sistema final, e o *actor* e os *page objects* podem ser adaptados de acordo. Quando os testes passarem, a página estática pode então ser transformada em um script dinâmico, conforme a especificação do sistema. Essa transformação pode ser feita de forma gradual, com execuções periódicas dos testes, para garantir a correção do processo. A segunda solução é realizar esta etapa paralelamente com a etapa seguinte, ou seja, desenvolver o script final gradualmente, e adaptando os testes na mesma medida.

A primeira solução é recomendada para páginas complexas, e/ou com um grande volume de elementos, pois a complexidade do *actor* e dos *page objects* pode tornar muito difícil a sua adaptação simultânea com o desenvolvimento do sistema. A desvantagem é a ocorrência de um certo retrabalho, pois deve ser desenvolvido o modelo estático da página, que então é substituído pelo script dinâmico. A segunda solução é menos custosa,

e pode ser aplicada em páginas mais simples. De qualquer forma, o objetivo desta fase é a obtenção de scripts de testes que estão totalmente de acordo com o sistema final.

3.4.3 Implementação do Script no Sistema Final

Dependendo da opção realizada na etapa anterior, a concretização desta fase é o desenvolvimento do script no sistema final, de forma a satisfazer todos os testes. É importante ressaltar que o sucesso desta etapa não depende apenas da existência dos testes, mas também do trabalho realizado na primeira fase. Uma evidência do sucesso da execução do processo é que o trabalho de desenvolvimento neste passo seja relativamente pequeno, pois deve-se reutilizar ao máximo possível os componentes e recursos desenvolvidos na primeira fase. Se o desenvolvedor sentir a necessidade de implementar novamente a lógica de negócio no sistema final, é sinal de que ainda há oportunidade de refatoração no sistema intermediário.

O benefício do reuso de componentes é observado principalmente no processamento de formulários, pois isso não envolve alterações de interface. Assim, os componentes desenvolvidos na primeira fase podem ser reusados em sua totalidade.

É necessário frisar que, ao contrário do que é feito na primeira fase, não se deve manter a sincronia das classes de teste, pois os *actors* e *page objects* serão diferentes para o sistema final. Se for observada a necessidade de mudança nos testes, apenas os scripts de teste e as fixtures devem ser sincronizados.

3.5 Considerações Finais

Neste capítulo, foi apresentado e detalhado o processo de reengenharia proposto para a Universidade Federal do Rio Grande do Sul. Ele foi definido com base na literatura e experiência com as tecnologias utilizadas no contexto da UFRGS. Este processo foi aplicado na reengenharia de dois módulos de sistemas da UFRGS para validação, a qual é apresentada no capítulo seguinte.

4 VALIDAÇÃO DO PROCESSO

Este capítulo apresenta a validação do processo descrito no capítulo anterior, descrevendo detalhadamente como as duas fases apresentadas foram executadas em módulos de sistemas reais. Cada módulo é apresentado, e em seguida é descrito como foi realizada a reengenharia de cada uma das telas dos módulos escolhidos (Seções 4.1 e 4.2).

Como validação do processo, foram realizados dois projetos de reengenharia em sistemas da UFRGS. O primeiro foi a reengenharia total do módulo de pareceres de extensão, e o segundo foi a reengenharia parcial do módulo de solicitação de fomento à extensão. Os critérios para a escolha destes módulos foram:

1. Ele deve ser pequeno o suficiente para que fosse possível uma reengenharia de uma porção significativa do módulo no escopo deste trabalho, com toda a sua lógica;
2. Ele deve possuir características típicas de projetos PHP legados, conforme eles eram desenvolvidos até antes da adoção do Yii Framework pela universidade, e;
3. Ele deve possuir uma complexidade suficiente para por à prova a eficácia do processo de reengenharia, e medir o nível de dificuldade de sua aplicação.

Os dois casos são descritos em detalhe a seguir.

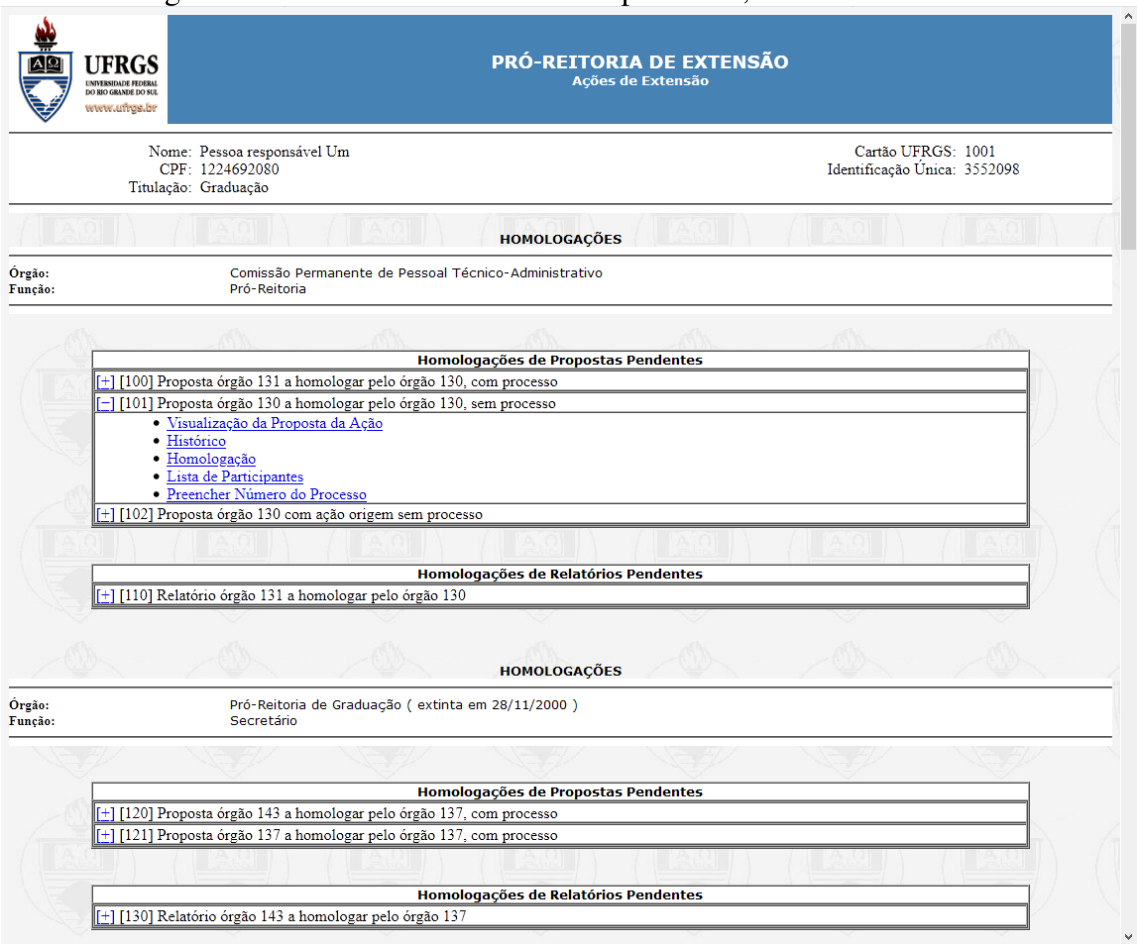
4.1 Reengenharia do Módulo de Pareceres

Esta seção descreve a aplicação do processo de reengenharia no módulo de pareceres do Sistema de Extensão da UFRGS, com a geração de todas as telas no sistema final.

4.1.1 Descrição do Módulo

O módulo de pareceres do Sistema de Extensão é responsável por passos importantes no trâmite de uma atividade de extensão. As propostas de atividades submetidas por algum docente da UFRGS passam pela chefia imediata do coordenador, que pode tanto autorizar a proposta quanto rejeitá-la, retornando-a para o coordenador responsável. Depois, elas precisam da homologação do responsável pela unidade na qual a atividade será executada, e a homologação pode ser aceita ou rejeitada. Um processo análogo

Figura 4.1: Tela inicial do módulo de pareceres, com dados de teste



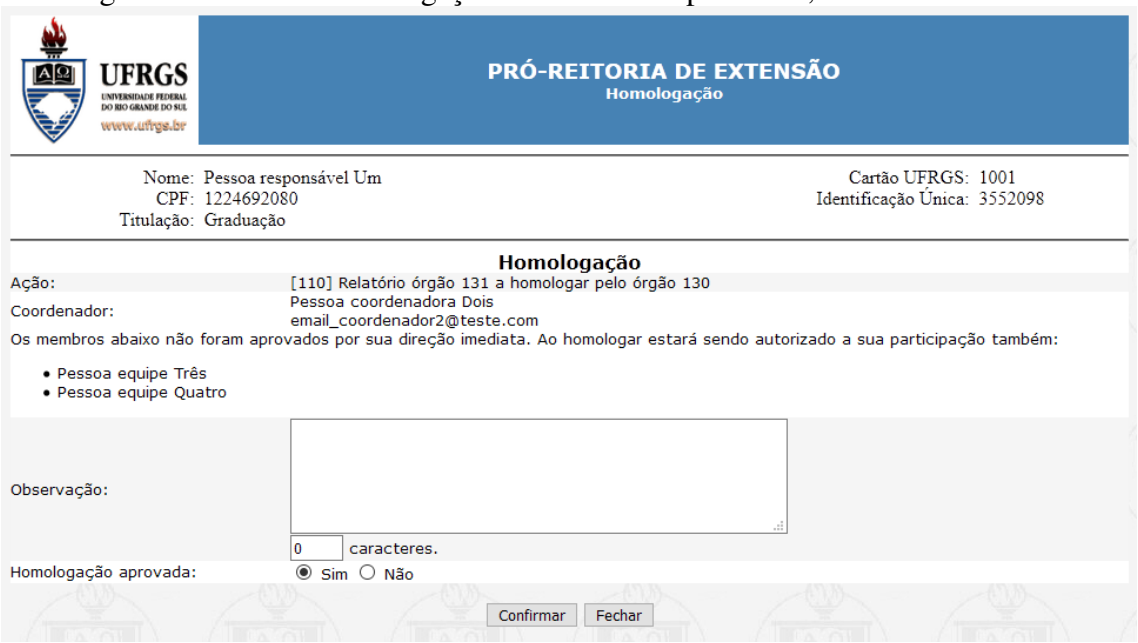
Fonte: O Autor

ocorre quando o relatório de execução da atividade é submetido. Também é necessária a autorização das participações dos membros da equipe pela chefia.

Na tela inicial do módulo (Figura 4.1), o usuário tem acesso a todas as atividades de extensão que possuem alguma atividade pendente, nesta ordem: propostas ou relatórios para homologar como responsável pela unidade, e propostas, relatórios e participações de membros para autorizar como chefia imediata. Além disso, o usuário também pode ver um histórico de atividades que já passaram por ele. Cada proposta, relatório ou membro de equipe disponível leva a um formulário específico (Figura 4.2), no qual o usuário pode selecionar sua opção (aceitar ou rejeitar) e preencher uma observação.

Assim, o módulo consiste basicamente de duas telas, porém ambas possuem uma complexidade considerável. A listagem inicial de atividades deve contemplar o fato de que o usuário pode ser responsável por mais de uma unidade, e a existência de quatro situações distintas na qual uma atividade pode estar: homologação de proposta, homologação de relatório, autorização de proposta e autorização de relatório. Além disso, existem as participações de membros de equipe, sendo que uma atividade pode ter vários membros

Figura 4.2: Tela de homologação do módulo de pareceres, com dados de teste



Fonte: O Autor

aguardando autorização.

Assim, o módulo precisa contemplar cinco situações distintas, sendo que para cada uma o usuário dispõe de duas opções: aceitar ou negar. Embora o formulário seja muito parecido para todos os casos, a sequência de operações é diferente. Ou seja, tratam-se de dez operações totalmente distintas. Todas essas dez operações estão implementadas em um único script PHP, organizadas em grandes estruturas condicionais. Apenas algumas poucas operações pontuais estão implementadas em funções reusáveis, como os envios de e-mail.

Na tela do menu principal, houve uma tentativa de criar classes para listar as atividades, para tentar reduzir a redundância de código. Porém, a organização hierárquica entre unidades e atividades, bem como as diferenças entre as situações das atividades, adicionam complexidade ao código, tornando sua compreensão difícil para um desenvolvedor iniciante. Com tudo isso, foi observado que o módulo se encaixa nos critérios de seleção listados acima.

É importante destacar que, a partir do ano de 2020, as atividades não precisam mais da autorização da chefia imediata. Porém, essa funcionalidade não foi removida do módulo, em grande parte pela possibilidade de que ela seja reincluída no futuro. Portanto, a reengenharia do módulo contempla essa funcionalidade.

4.1.2 Primeira Fase da Reengenharia

Inicialmente, foi realizada a primeira fase da reengenharia em todas as funcionalidades do módulo. Todos os passos da reengenharia de cada funcionalidade são descritos a seguir.

4.1.2.1 Menu Inicial

Antes de iniciar o processo em si, foram feitas algumas modificações nos requisitos do módulo. Algumas funcionalidades pontuais não foram incluídas na reengenharia, como a simulação de usuários por parte de um conjunto restrito de administradores do sistema, e o preenchimento de números de processo nas atividades. Essas funcionalidades não adicionam valor à validação do processo, portanto serão desconsideradas neste trabalho.

O menu inicial do módulo foi a primeira funcionalidade a ser selecionada, justamente por tratar-se do ponto de entrada do usuário no módulo. Como determina o processo, o primeiro passo dado foi a engenharia reversa do código-fonte, começando pelas suas variáveis de entrada. Constatou-se que apenas duas entradas são utilizadas: o identificador do usuário logado, e uma data de início para exibição do histórico de atividades passadas. Essa data de início pode ser informada pelo usuário em um campo exibido na tela. Quando o sistema não recebe nenhuma data, ele assume por padrão o primeiro dia do mês atual.

A maior parte da lógica deste script depende inteiramente do usuário logado, de seu relacionamento com as unidades da UFRGS, e dos órgãos aos quais as atividades de extensão estão associados. Esses relacionamentos são resolvidos em consultas SQL no sistema original, portanto foi fundamental compreender essas consultas. Além disso, tais consultas utilizam visões do banco de dados, as quais não podem ser manipuladas diretamente. Felizmente, foi possível intuir o comportamento dessas visões observando os dados disponíveis no banco de desenvolvimento.

Após isso, foi necessário entender como o sistema monta as listas de órgãos e os menus de atividades de extensão a partir dos dados disponíveis. Foi necessário uma atenção especial ao comportamento do sistema em relação à não existência de nenhuma atividade de extensão em alguma etapa específica, e como as mensagens são apresentadas.

A partir dessas observações, foram gerados os dados de teste (fixtures) e as classes de teste em si. A separação em três camadas foi seguida rigorosamente, de forma que

a camada mais alta de testes verifica apenas os dados exibidos na página, e os valores usados para gerar os links para outras páginas (ex. códigos identificadores das atividades de extensão, códigos de órgãos, etc).

Os testes foram então transferidos para o local do sistema intermediário, dando início ao processo de reengenharia propriamente dito. Para garantir que o sistema intermediário produza um resultado igual ao do sistema original, foram usados como base os códigos HTML resultantes das execuções dos testes do sistema original. A lógica em si, porém, foi totalmente refeita. A busca dos órgãos, bem como das atividades de extensão dentro de cada órgão, ficaram sob responsabilidade de classes separadas. A arquitetura de classes usada no sistema original para montagem dos menus não foi utilizada, pois foi preferível manter o código mais legível e claro.

4.1.2.2 Formulário de Parecer

O formulário no sistema original era separado em dois scripts: o primeiro apenas exibia o HTML do formulário em si, e o segundo realizava o processamento dos dados enviados por Post. A engenharia reversa de cada script foi feita separadamente.

A exibição do formulário utiliza como variáveis de entrada, além do identificador do usuário logado, o identificador da atividade de extensão, o identificador do órgão responsável pela atividade, um código referente ao tipo de ação a ser realizada (autorização de proposta, homologação de proposta, etc.), e mais o identificador do membro da equipe executora, usado apenas no caso de autorização de membro. Esses mesmos dados são mandados como entrada para o script de processamento, junto com a opção selecionada pelo usuário (aprovar ou negar) e uma observação textual (exceto para a autorização de membro de equipe).

Algumas peculiaridades do código da aplicação tornaram a engenharia reversa bastante complexa. Por exemplo, o identificador do órgão responsável, recebido como parâmetro de entrada, não era utilizado em nenhum momento. Além disso, havia uma quantidade considerável de código morto causado por funcionalidades que foram abandonadas ao longo do tempo, e situações que, em tese, nunca deveriam ocorrer. Além disso, foi necessário mapear exatamente a sequência de passos tomada em cada uma das dez opções possíveis, incluindo o envio de e-mails para destinatários diferentes em cada caso.

Devido ao problema do envio de e-mails em um sistema em ambiente de desenvolvimento, o sistema exibe na tela os dados do e-mail a ser enviado. Isso tornou possível

testar se tais informações estão sendo geradas corretamente, mas não foi possível testar se, na prática, os e-mails são enviados corretamente. Já as operações no banco de dados podem ser testadas com um nível alto de confiança, com a exceção da execução de uma função do banco de dados para encerrar uma pendência do usuário.

A geração das fixtures e dos testes foi um trabalho extenso, devido não só ao número de situações possíveis, mas a situações adicionais existentes em algumas dessas opções (por exemplo, a homologação de propostas de atividades classificadas como MOOCs, ou "Massive Open Online Courses").

A reengenharia do formulário envolveu uma reescrita completa do código, usando também uma arquitetura de classes para separar cada tipo de operação sem depender de estruturas de controle complexas. Foi dada uma atenção especial para implementar essa arquitetura de forma reusável e desacoplada, para que ela pudesse ser utilizada pelo sistema final, sem alterações. Além disso, como o processamento de cada opção envolve operações parecidas, mas sutilmente diferentes (ex. inserções em uma mesma tabela, porém com dados diferentes), foi feito um esforço para que o código responsável por cada opção fosse claro e facilmente compreensível, facilitando assim quaisquer alterações futuras no processo burocrático da Extensão.

4.1.3 Segunda Fase da Reengenharia

Após o término da primeira fase, foi realizada a segunda fase do processo para ambas as funcionalidades. Os passos realizados são descritos a seguir.

4.1.3.1 Tela Inicial

A reengenharia da tela principal para o sistema final envolveu basicamente uma mudança nos elementos HTML utilizados para a montagem do menu, utilizando os estilos já padronizados dos sistemas do portal da UFRGS. Em termos de lógica, a única mudança significativa foi a remoção do identificador do órgão responsável pela atividade como parâmetro de entrada ao formulário, pois foi constatado tratar-se de um parâmetro redundante, que prejudica a clareza do código.

Uma mudança substancial no layout da página foi a remoção do cabeçalho com os dados pessoais de usuário. Como já é padrão que o nome do usuário autenticado é exibido sempre no mesmo lugar, foi considerado desnecessário exibir informações adicionais no

corpo da página em si.

A abordagem escolhida para a reengenharia desta tela foi utilizar uma página HTML estática como base (foi escolhida a página resultante do primeiro teste), alterando os elementos HTML até ficarem de acordo com o padrão desejado. Após isso, os *page objects* e *actor* do teste automatizado foram alterados até o teste passar. Conforme determinado pelo processo, o script de teste em si permaneceu inalterado. Isso teve algumas consequências curiosas na lógica de testes. Com a remoção do cabeçalho, o método que verifica os dados pessoais verifica apenas o nome do usuário, e ignora todos os outros dados. Além disso, o código identificador do órgão, que é verificado em cada um dos links para o formulário, também é ignorado pelo *actor*.

No momento em que o teste passou, os elementos estáticos da página HTML foram substituídos pela lógica de busca de órgãos e atividades. Esse processo foi relativamente simples, pois foram reusadas as classes produzidas para o sistema intermediário, sem necessidade de alterações ou adaptações. Na medida em que a lógica foi desenvolvida, o teste era rodado novamente, para manter a correção de cada parte da página, até que toda a lógica estava implementada. Após isso, os outros testes foram executados, e a página foi adaptada para que todos os testes passassem. O resultado é exibido na Figura 4.3.

4.1.3.2 Formulário de Parecer

Pelo fato de o formulário ser mais simples e pequeno, não foi utilizada a mesma abordagem da tela inicial. Em vez disso, o formulário foi implementado do zero, e a adaptação das classes de teste foi realizada de forma paralela. Foram necessárias mais adaptações desta vez, pois a geração dos elementos do formulário foi feita usando as classes *helper* do Yii Framework, mudando os nomes dos elementos, e necessitando alterações no *actor*. A reengenharia desta tela foi bem mais simples, pois tanto o mecanismo de geração de formulários do Yii, quanto as classes geradas para o sistema intermediário, já providenciavam grande parte da implementação necessária.

4.2 Reengenharia do Módulo de Solicitação de Fomento

Esta seção relata a aplicação do processo em uma parte do módulo de solicitação de fomento à extensão, com a migração de três funcionalidades para o sistema final.

Figura 4.3: Tela inicial da versão final do módulo, com dados de teste

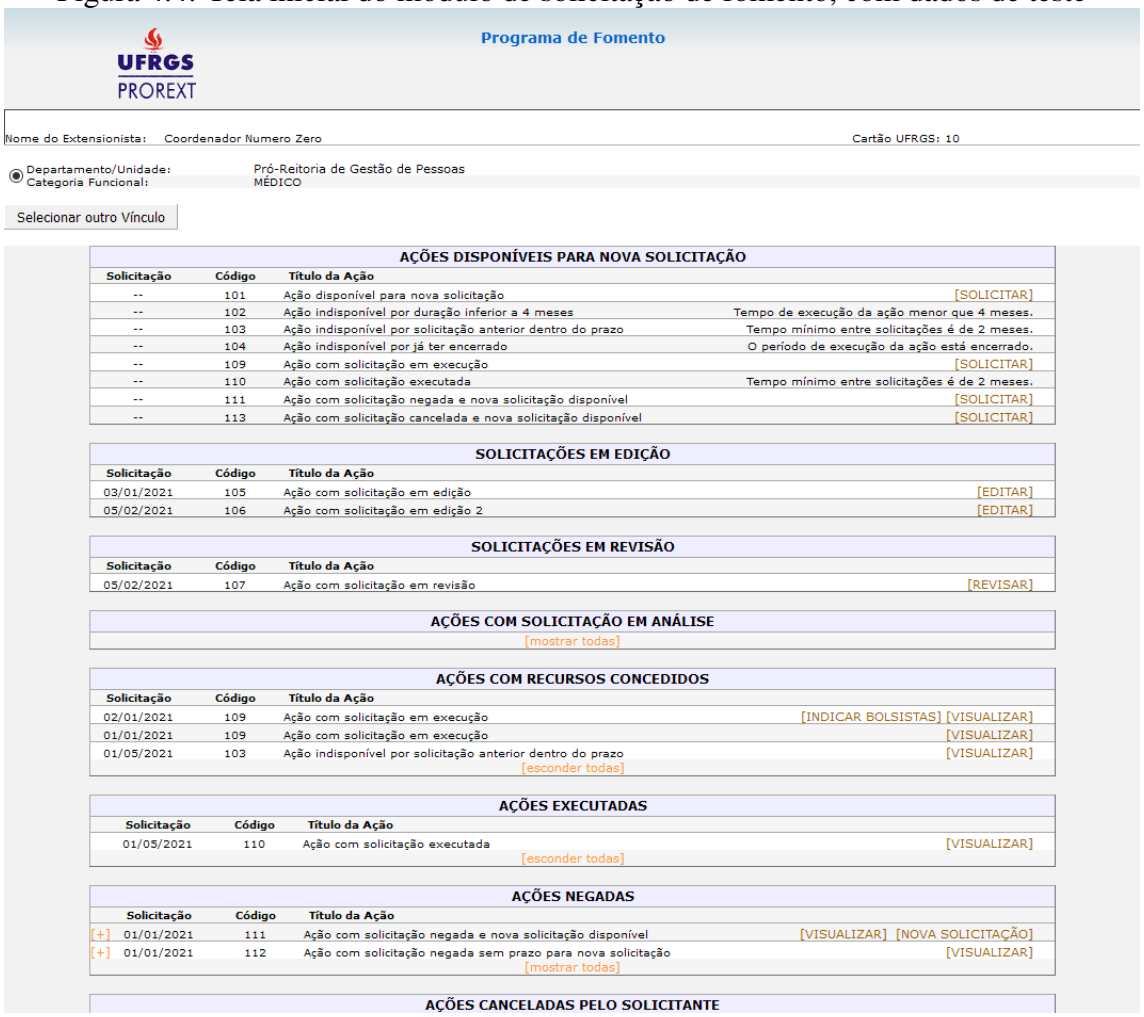
Fonte: O Autor

4.2.1 Descrição do Módulo

O módulo de solicitação de fomento é uma parte do módulo de fomento à extensão, que é responsável pelo trâmite da concessão de recursos financeiros e materiais para a execução de atividades de extensão na universidade. O módulo gerencia todos os passos do processo, começando pela solicitação em si, a análise e aprovação por parte da PROREXT, a geração do pagamento, a informação dos valores executados, e outros.

Este trabalho focou-se em três funcionalidades do módulo de solicitação, no qual o extensionista tem acesso às atividades para as quais a solicitação está disponível, bem como às solicitações já realizadas, em todos os seus diferentes estágios. Dependendo da situação de cada solicitação, o extensionista pode realizar diferentes ações, como revisão,

Figura 4.4: Tela inicial do módulo de solicitação de fomento, com dados de teste



Fonte: O Autor

execução, visualização e indicação de bolsistas (Figura 4.4).

Na tela de visualização, o extensionista pode ver, discriminadamente, cada tipo de recurso solicitado. Dependendo da situação, o usuário verá o valor solicitado, o valor concedido ou o valor executado para cada recurso. Os recursos disponíveis dependem ainda da modalidade da atividade, e cada recurso pode ter características específicas associadas, como a necessidade de preencher a pessoa que receberá o recurso, uma descrição textual, e um ou mais arquivos a serem anexados pelo solicitante.

A tela de solicitação é mais complexa, pois o usuário pode realizar novas solicitações, ou realizar a edição e revisão de solicitações já existentes. Em ambos os casos, o usuário pode selecionar os recursos desejados e informar dados como o valor monetário ou a quantidade (dependendo do tipo de recurso), a justificativa, a pessoa à qual o recurso é destinado, o período, e um ou mais documentos anexos.

Pela complexidade do módulo, o processo de reengenharia focou-se apenas nessas três funcionalidades. O processo será descrito abaixo.

4.2.2 Primeira Fase da Reengenharia

4.2.2.1 Menu Inicial

Da mesma forma que no projeto anterior, algumas funcionalidades do menu inicial foram desconsideradas, de forma que o trabalho focasse na parte mais complexa do processo.

As variáveis de entrada utilizadas neste script são variáveis de sessão correspondente ao usuário logado. Além do código identificador da pessoa, a página exige a seleção de um de seus vínculos. O vínculo selecionado não afeta nenhuma parte da lógica desta página, embora seja determinante na permissão do acesso às telas específicas de cada atividade de extensão, as quais são acessíveis através deste menu. Isso poderia ser considerado uma inconsistência na lógica do sistema, mas é importante destacar que este projeto não visa corrigir erros nos sistemas em questão. Isso deve sempre ser analisado em separado.

Foi mais fácil compreender a lógica das consultas utilizadas nesta página, pois todas elas envolvem um número reduzido de tabelas. A dificuldade maior estava relacionada às diferentes situações nas quais cada solicitação pode estar, e nas opções disponibilizadas para cada situação, bem como na formatação de cada tabela exibida. No total, além da tabela que exibe atividades de extensão disponíveis para novas solicitação, há sete situações possíveis para solicitações existentes, e, para cada situação, há diferenças nas tabelas. Além disso, para as novas solicitações, há três motivos distintos pelos quais uma solicitação não pode ser realizada.

Isso exigiu uma quantidade considerável de dados de teste, pois era necessário verificar isoladamente cada possibilidade dentro de cada tabela de atividades. Além disso, foram gerados casos de teste para usuários sem solicitações realizadas, sem atividades disponíveis para novas solicitação, e sem nenhuma das duas coisas.

A semelhança entre as estruturas HTML das tabelas foi explorada, de forma que o *page object* ficou relativamente simples. A classe *actor*, porém, ficou bastante extensa, devido à grande quantidade de casos diferentes a serem testados. Ainda assim, o processo foi seguido à risca, o que permitiu que a classe de testes ficasse descritiva e abstrata conforme necessário.

O desenvolvimento da página no sistema intermediário exigiu uma reformulação de como a consulta de solicitações é realizada. No sistema original, todas as solicitações são retornadas em uma única consulta, ordenadas de acordo com a situação. A situação é

representada em um código numérico, cuja ordem não corresponde à ordem de exibição das tabelas na página. Embora isso teve uma solução bastante simples no sistema original, a reescrita da consulta no Yii exigiu uma abordagem diferente: o sistema realiza a chamada a um método, que retorna um vetor com até sete posições, cada uma correspondendo a uma situação, e com a lista de solicitações correspondente. A solução passou por uma série de refatorações desde sua implementação inicial até a solução final adotada, o que foi possível graças aos testes automatizados.

4.2.2.2 Visualização da Solicitação

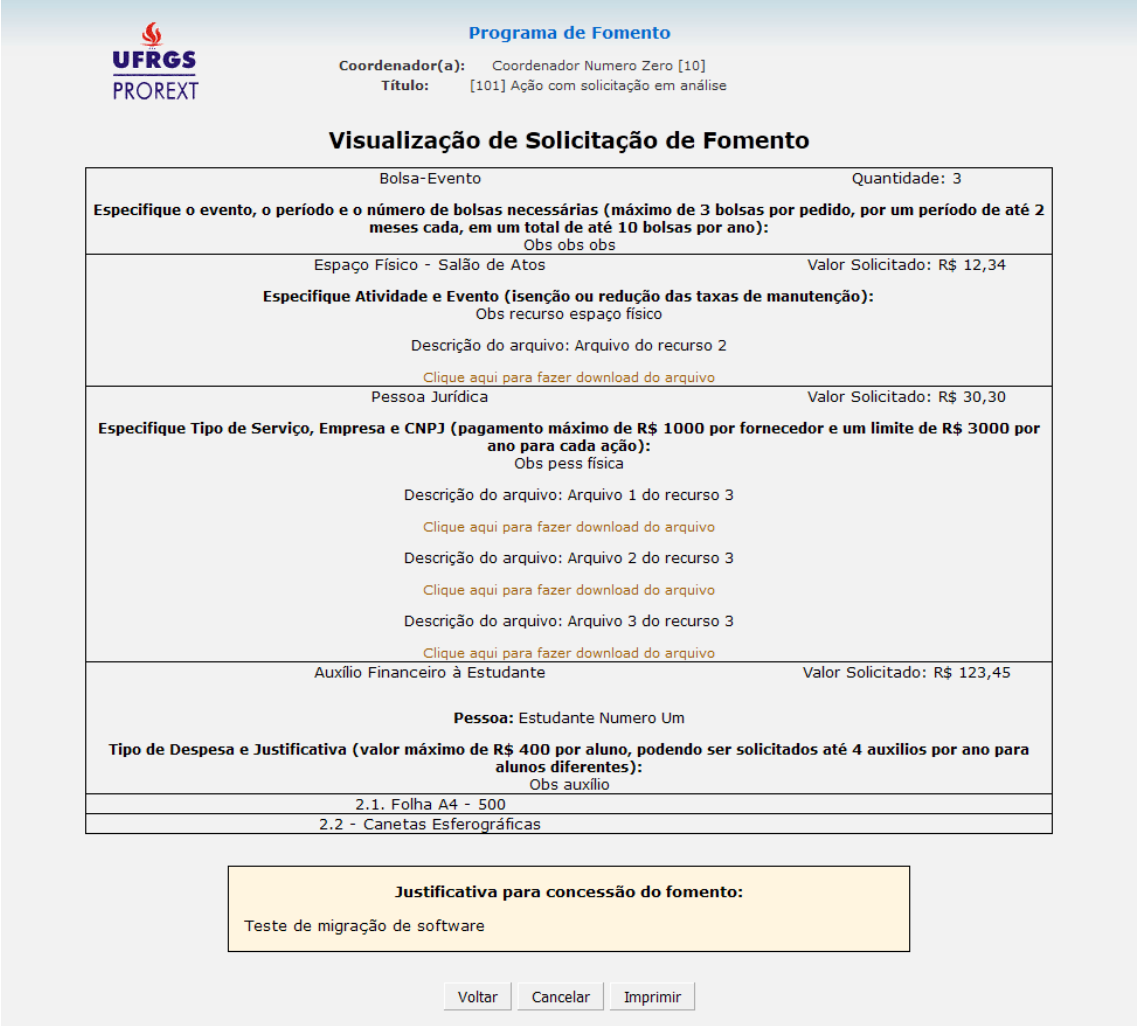
A tela de visualização, por sua vez, apresentou um conjunto diferente de desafios. O mapeamento das entradas foi simples, pois trata-se apenas dos códigos identificadores da atividade de extensão e da solicitação, além dos dados de autenticação do usuário.

Para a solicitação selecionada, a tela exibe um detalhamento de cada recurso incluído, juntamente com dados informados para cada um, que variam dependendo do tipo do recurso. Isso inclui a pessoa que receberá o pagamento, uma descrição textual, e um ou mais arquivos anexos (Figura 4.5). Portanto, foi necessário gerar dados de teste que contemplem todas as possibilidades de exibição de recursos. Além disso, a página é desenvolvida para exibir solicitações em diferentes situações, o que acarreta em pequenas mudanças textuais na página. Além disso, foram gerados testes para os casos de acesso por um usuário não autorizado, ou para uma solicitação inexistente.

O desafio principal nesta tela diz respeito à exibição dos dados dos recursos. As várias informações relativas a cada recurso são colocados em linhas diferentes de uma mesma tabela; ou seja, a estrutura dos elementos HTML não segue a hierarquia dos dados, e os elementos ficam todos planificados. Assim, para verificar os dados de cada recurso, é necessário saber em qual linha da tabela os seus dados começam, e isso varia com o caso. A solução mais óbvia seria que a classe de teste informasse para o *actor* em qual linha da tabela cada recurso começa; porém, isso cria um vínculo indesejado entre a semântica da página e a sua implementação específica em termos de HTML; ou seja, a classe de teste não ficaria mais responsável apenas por verificar as informações contidas na páginas, mas sim sua disposição específica na estrutura da página.

Para evitar essa característica indesejada, foi necessário implementar o *page object* de maneira inteligente: no momento em que ele é instanciado, ele varre cada linha da tabela, examinando qual tipo de informação é exibida (nome do recurso, justificativa textual, pessoa ou documento anexo. Essas informações são armazenadas em atributos

Figura 4.5: Tela de visualização de solicitação de fomento, com dados de teste



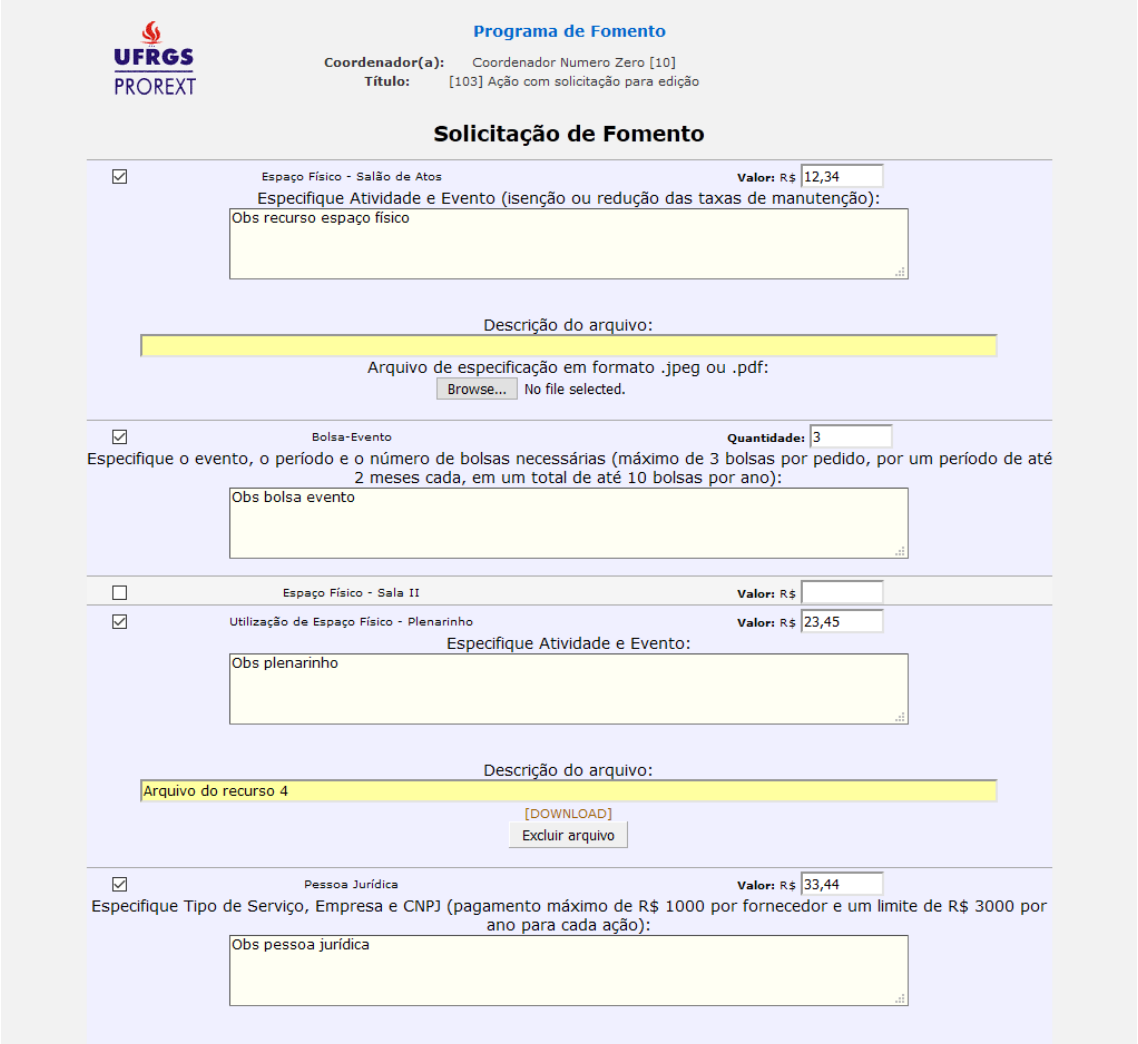
Fonte: O Autor

do objeto, os quais são utilizados não apenas para informar ao *actor* se cada uma dessas informações existe para cada recurso, mas em qual linha da tabela cada recurso está localizado. Assim, basta o *actor* informar o número do recurso, e o *page object* localiza exatamente onde os seus dados estão incluídos.

Dessa forma, a implementação específica do sistema original forçou uma complexidade adicional nas classes de teste; complexidade essa que pode ser eliminada no sistema final, pois basta uma organização mais estruturada das informações na página para que o *page object* fique mais simples.

Porém, no sistema intermediário, a estrutura da página foi mantida, conforme requerido pelo processo. Casualmente, a implementação da página no sistema intermediário foi relativamente simples, devido à baixa complexidade da página em si. Não foi necessário fazer nenhuma implementação complexa em termos de consultas, pois as classes do Yii Framework já são naturalmente adaptadas para esse tipo de manipulação de dados.

Figura 4.6: Formulário de solicitação de fomento, com dados de teste



Fonte: O Autor

4.2.2.3 Formulário de Solicitação

A funcionalidade do formulário, responsável pela realização da solicitação, apresentou o maior desafio, tanto pela complexidade da funcionalidade em si, quanto pela sua implementação em termos de elementos HTML. Assim como na visualização, o formulário apresenta diversos tipos de recursos, os quais podem ser selecionados pelo usuário. Porém, em vez de informações estáticas, esta tela apresenta campos de entrada, que são diferentes de acordo com o tipo de cada recurso.

De forma análoga à tela de solicitação, esses elementos são apresentados de forma planificada, o que exigiu uma complexidade ainda maior na implementação do *page object*. Não só isso, mas o formulário é processado por um segundo script, que realiza as operações no banco de dados correspondentes à opção do usuário, o que significa um volume ainda maior de testes.

Em termos de dados de entrada, além do identificador do usuário logado armazenado em sessão, a tela recebe o código da atividade de extensão à qual a solicitação está associada. Além disso, a tela pode receber o código de uma solicitação existente, em caso de edição, ou nenhum código, no caso de criação de nova solicitação. Em cada caso, há um comportamento diferente do sistema. Além disso, os recursos cadastrados em uma solicitação em edição também contam como casos de entrada, pois os testes precisam verificar como esses dados são exibidos no formulário. Isso gerou uma complexidade bastante grande no caso de recursos financeiros que exigem cadastro de arquivos anexos: quando já existe algum documento cadastrado, isso altera a quantidade de elementos ocultos (*hidden*) para aquele recurso, o que deve ser calculado pelo *page object*.

Fora isso, o usuário possui três opções de ação: salvar, encaminhar ou cancelar. O salvamento registra as alterações no banco de dados, mas mantém a solicitação em edição. O encaminhamento, além de salvar, envia a solicitação para análise. Por fim, o cancelamento simplesmente marca a solicitação como cancelada, tornando-a indisponível para posterior edição. Também existe a possibilidade de exclusão de um arquivo previamente anexado. Além dessa variedade de opções, também existe um caso específico para solicitações em revisão. Estas só podem ser encaminhadas dentro de um prazo de 60 dias após a análise. Após esse prazo, o encaminhamento torna-se indisponível.

Se, por um lado, essa tela gerou uma grande quantidade de testes e um grande volume de código, por outro lado, essa tela esbarrou em algumas limitações da ferramenta escolhida. A primeira é que a versão utilizada do Codeception não dá suporte à anexação de arquivos, o que tornou impossível a automatização do teste dessa funcionalidade. Junto a isso, o formulário depende de algumas funcionalidades implementadas em JavaScript, como a busca de pessoa e a possibilidade de replicar um tipo de recurso, cadastrando dois um mais recursos do mesmo tipo. Devido ao fato dos testes funcionais serem realizados sem o uso de um navegador, também não foi possível automatizar os testes dessas funcionalidades.

A reengenharia desta funcionalidade, portanto, exigiu uma implementação cuidadosa no sistema intermediário. Para reduzir a dependência em testes manuais e minimizar a possibilidades de erros na interface, o código JavaScript legado foi mantido sem alterações, e os testes automatizados verificam que os nomes e identificadores dos elementos HTML foram preservados, mantendo a compatibilidade do código. Consequentemente, como os nomes dos elementos não aderem ao padrão do Yii Framework, foi necessário implementar uma camada intermediária entre os dados recebidos do formulário HTML e

a classe implementada no sistema intermediário para processar tais dados. Mesmo assim, a classe foi feita de forma a ser reutilizada no sistema final. A lógica do processamento, porém, foi totalmente refeita e refatorada com o apoio dos testes automatizados. Mesmo assim, foi necessário executar testes manuais para garantir que a busca de pessoas e a anexação de arquivos estivesse funcionando.

4.2.3 Segunda Fase da Reengenharia

Com o fim da primeira fase, foi realizada a segunda fase do processo nas três funcionalidades do módulo. Os passos são descritos a seguir.

4.2.3.1 Tela Inicial

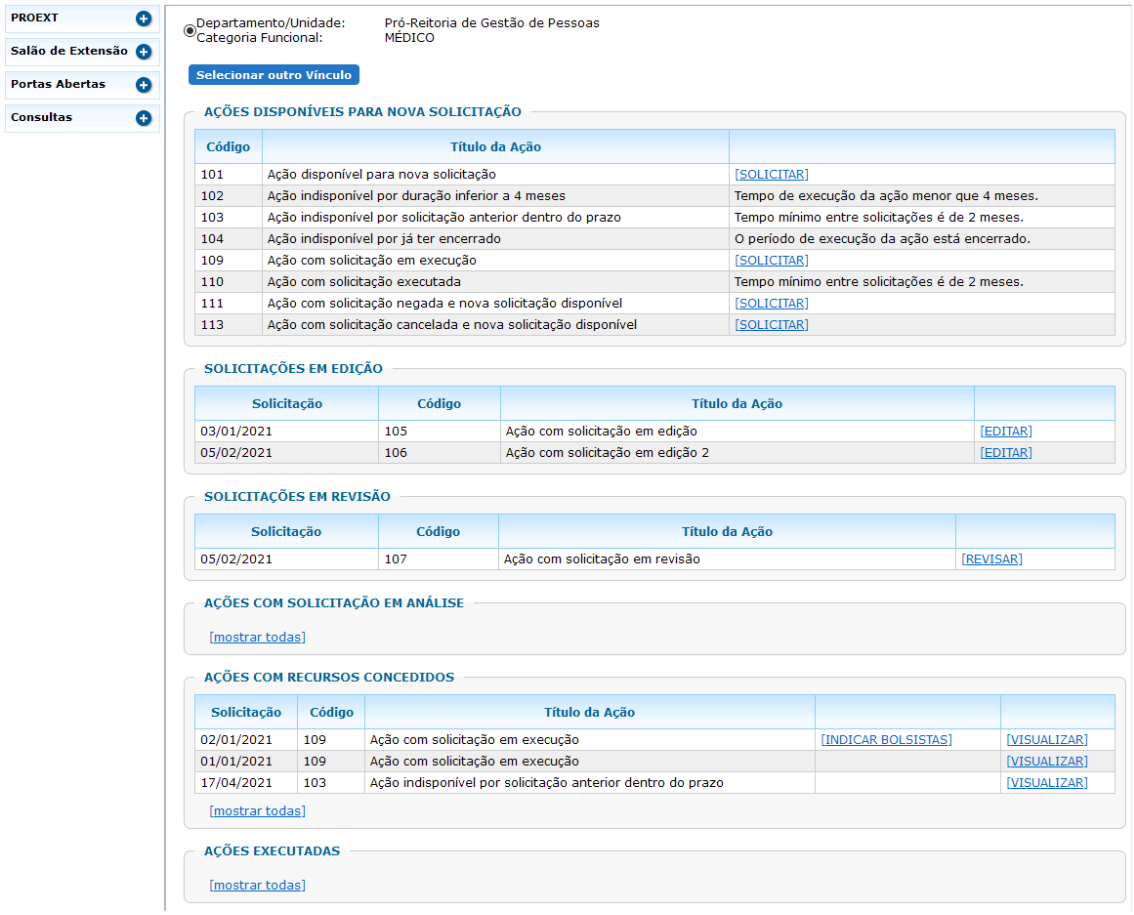
A mudança nos elementos HTML da tela inicial exigiu menos esforço em relação ao primeiro projeto, mas foram realizadas mudanças na disposição dos dados, para eliminar certas redundâncias. Foi necessário principalmente alterar as classes CSS das tabelas, para adequá-las ao padrão do portal da UFRGS (Figura 4.7), o que não necessitava nenhuma alteração nas classes de teste.

Desta vez, a abordagem escolhida para a implementação foi diferente da utilizada no primeiro projeto: ou seja, não foi utilizada uma página HTML estática, pois foi considerado que isso seria mais trabalhoso do que simplesmente implementar o script conforme ele deveria ser. Após a implementação ser concluída, as classes de teste foram então adaptadas. Devido ao fato de que a estrutura da página não foi radicalmente alterada, foram necessárias poucas alterações nas classes de teste, e o processo de reengenharia foi relativamente rápido. De fato, o desenvolvimento dos testes foi o passo mais custoso deste processo.

4.2.3.2 Visualização da Solicitação

Devido à baixa complexidade da lógica de negócios, já citada na seção anterior, a implementação da visualização da solicitação foi também simples. Da mesma forma que na tela anterior, a tela foi implementada sem o uso de uma página estática. Comforme indicado na seção anterior, foi escolhida uma disposição mais adequada dos elementos HTML, embora sua exibição na tela não tenha sido significativamente modificada (Figura 4.8).

Figura 4.7: Tela inicial do novo módulo de solicitação de fomento, com dados de teste



Fonte: O Autor

As devidas adaptações das classes de teste confirmaram a correção do desenvolvimento da página. Foi observado, após o termino do processo, que o desenvolvimento das classes de teste foi a parte mais complexa e custosa da reengenharia desta tela.

4.2.3.3 Formulário de Solicitação

Ao contrário da funcionalidade anterior, esta funcionalidade foi mais desafiadora nesta fase do processo. Por um lado, os elementos HTML foram reformulados de forma a simplificar as classes de teste, resultado em código mais inteligível. Por outro lado, para tornar a implementação mais compatível com o Yii Framework, os nomes dos elementos foram alterados. Isso exigiu que o código JavaScript fosse quase totalmente refeito, o que dependeu do uso de testes manuais. Além disso, a classe de processamento do formulário também precisou passar por algumas pequenas mudanças, e os testes do sistema intermediário foram executados para garantir que a compatibilidade não fosse perdida.

A estratégia adotada foi, em primeiro lugar, implementar a funcionalidade no sistema final de forma que todos os testes passassem. Assim como nas outras funcionali-

Figura 4.8: Tela nova de visualização de solicitação de fomento, com dados de teste

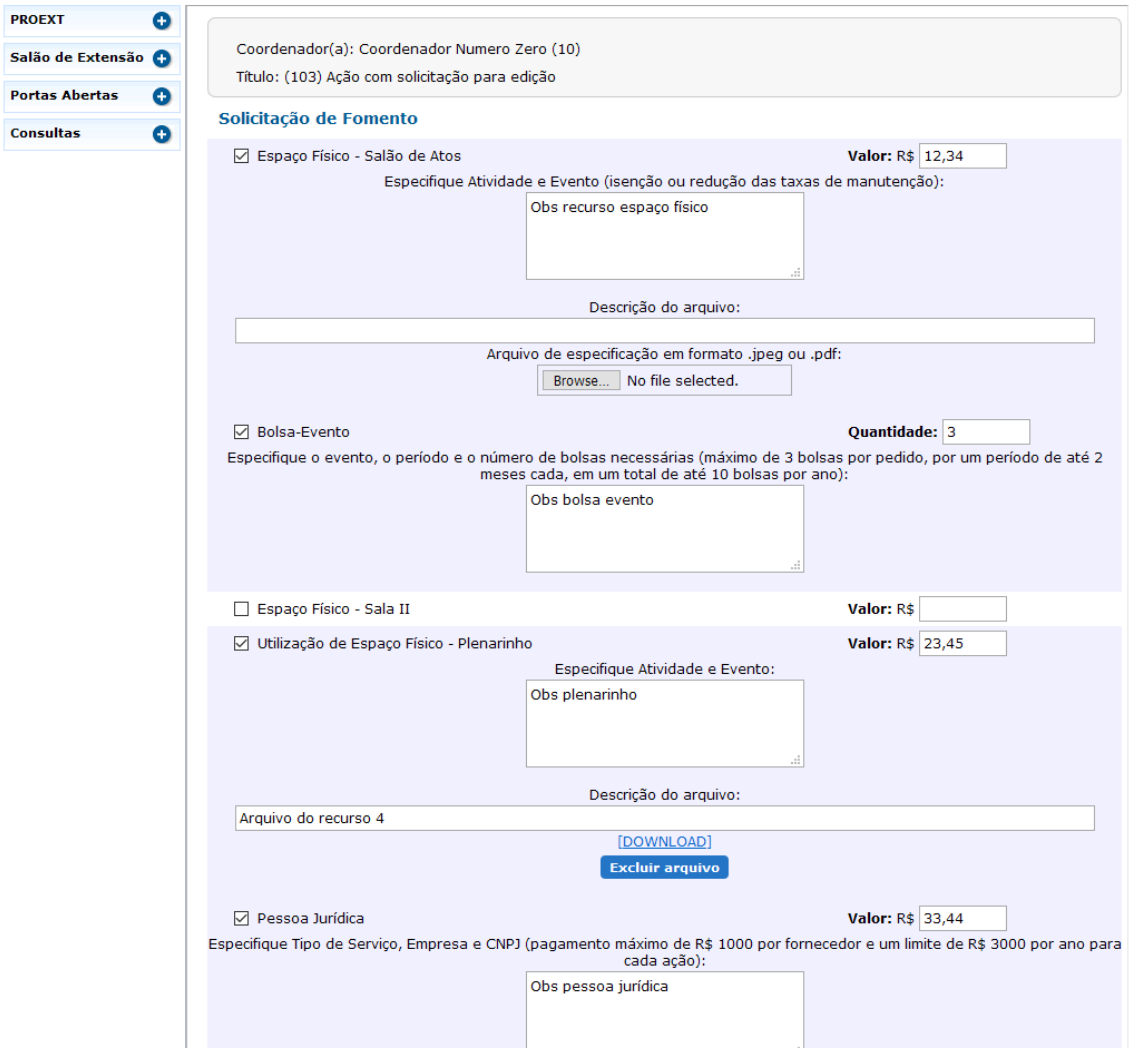
Fonte: O Autor

dades, o *actor* e o *page object* foram adaptados, até os testes passarem. Apenas depois disso é que as alterações no JavaScript foram realizadas para implementar as funcionalidades não cobertas pelos testes (busca de pessoa, anexação de documentos e replicação de recursos). Porém, os testes ainda mostraram-se importantes, pois eles podiam acusar qualquer alteração que introduzisse bugs na implementação já realizada. Desta forma, os testes manuais ficaram focados apenas nessas funcionalidades específicas.

4.3 Considerações Finais

A experiência da aplicação do processo nas funcionalidades acima providenciou uma série de desafios e questionamentos sobre a natureza e as características do processo. A diversidade de aspectos de desenvolvimento de sistemas e a complexidade das regras de negócio das funcionalidades foi essencial para dar uma visão aprofundada de como o processo se importa em um caso concreto. As observações colhidas durante a validação são discutidas no capítulo a seguir.

Figura 4.9: Formulário nova de solicitação de fomento, com dados de teste



Fonte: O Autor

5 DISCUSSÃO

A partir da experiência da validação do processo descrita no capítulo anterior, foi possível levantar algumas questões que serão discutidas neste capítulo. Tais questões são agrupadas em três grandes temas: o uso de testes automatizados (Seção 5.1), a melhoria da qualidade do sistema (Seção 5.2), e o uso da ferramenta Codeception (Seção 5.3).

5.1 Uso de Testes Automatizados

Uma das suposições que foram verificadas na validação do processo é a de que a elaboração dos testes automatizados serviria como um tipo de documentação das regras do sistema, obtida a partir da engenharia reversa. Na prática, verificou-se que isso não acontece, pois o código dos testes não fornece uma leitura fácil e objetiva do que o sistema deve fazer exatamente. Ou seja, os testes automatizados não substituem a especificação e documentação do sistema, e não servem para cobrir a lacuna de uma modelagem inexistente.

A geração dos testes automatizados é um processo longo e trabalhoso. Porém, a existência dos testes tornou o processo de reengenharia mais rápido e mais seguro, por reduzir drasticamente a necessidade de testes manuais, que, além de mais lentos, são menos confiáveis. Não só isso, mas os testes gerados no processo permanecem no sistema, podendo ser utilizado em quaisquer processos de manutenção evolutiva ou corretiva no futuro.

Por outro lado, o processo de geração de testes em si também é propenso a falhas. Durante a reengenharia da tela inicial do módulo na fase intermediária, foi observado um erro de execução durante um acesso manual ao sistema. Esse erro não foi observado pelos testes automatizados, pois a maneira como os fixtures foram gerados mascarava o erro. Além disso, a verificação da lista de atividades disponíveis em cada bloco também apresentava falha, pois um bug no sistema intermediário que resultava na duplicação de um desses registros não foi detectado pelos testes. O teste em si foi corrigido de forma que o erro era observado, e só então o bug foi corrigido. Desta forma, a utilização deste processo não elimina a necessidade de testes manuais, apesar de reduzir bastante sua frequência.

Além disso, algumas mudanças feitas no módulo de pareceres no sistema final (por exemplo, a remoção dos cabeçalhos de dados pessoais, e a remoção do código iden-

tificador do órgão como parâmetro de entrada do formulário) significa que os scripts de testes realizam a verificação de informações que não existem no sistema, e esse mascaramento só fica claro no código da classe *actor*. Podemos argumentar que, após o término do processo de reengenharia, os testes podem então ser alterados para melhor refletir o estado do sistema, porém essa dissonância persistirá por algum tempo durante o processo.

5.2 Melhoria da Qualidade do Sistema

A geração dos sistemas intermediários abriu uma oportunidade muito grande de refatoração e melhoria da arquitetura interna dos sistemas. Apesar do processo ter sido realizado em módulos pequenos, suas regras são bastante complexas e específicas, e era importante que o código-fonte expressasse de forma mais explícita o que está sendo realizado em cada momento, para facilitar alterações futuras. A possibilidade de aprimorar a qualidade interna do código, sem afetar o funcionamento do sistema para o usuário final, já representa por si só um avanço significativo, e pode-se considerar que apenas isso já justifica a existência deste processo.

Por outro lado, o fato de os testes automatizados se manterem iguais entre as versões do sistema a fim de manter a consistência de seu comportamento significa que o processo de reengenharia pode ficar comprometido com decisões de desenvolvimento que não sejam ideais. Por exemplo, no módulo de pareceres original, a ação a ser realizada pelo usuário no formulário é associada a um código numérico, que não tem nenhum significado em si, e o processamento desses diferentes casos é realizado em um mesmo script. Isso precisou ser mantido no sistema intermediário, e embora a complexidade do código tenha sido mitigada com a criação de uma estrutura de classes, esse código numérico persiste no sistema. Já no módulo de fomento, algumas melhorias puderam ser implementadas sem nenhuma interferência das classes de teste.

A divisão em duas fases providenciou também uma oportunidade para a melhoria da interface dos sistemas. O fato de o sistema final ser construído em cima das melhorias obtidas no sistema intermediário, com o embasamento dos testes automatizados, permite que o trabalho seja focado especificamente nas interfaces, tornando-o mais eficiente e menos propenso a erros.

5.3 Aspectos da Utilização do Codeception

O uso da própria ferramenta Codeception também apresenta alguns desafios adicionais. Devido ao sistema de organização dos ambientes de desenvolvimento e teste da UFRGS, os scripts de testes são executados com o acesso a uma URL via navegador, e isso faz com que alguns tipos de mensagens de erro não sejam exibidos. Isso possivelmente poderia ser solucionado com a adoção de um frontend para o Codeception, mas, até o presente momento, nenhuma solução foi adotada.

Além disso, a maneira como as mensagens de erro são exibidas podem fazer com que a investigação e depuração do problema seja longa e tediosa. Isso em parte é causado pela separação em camadas do código de teste, pois a sintaxe quase verbal dos testes funcionais tradicionais tendem a ter fácil depuração, mas as classes intermediárias aumentam o número de passos na investigação da causa do erro. Isso pode ser mitigado com o uso de mensagens de erro customizadas no *actor*, mas isso implica em trabalho adicional por parte do desenvolvedor. Além disso, a manipulação do código HTML pelos *page objects* pode resultar em alguns erros, cuja leitura não pode ser facilitada com esse recurso.

Foram encontradas outras limitações impostas pela ferramenta, tanto pela natureza dos testes funcionais, quanto pela versão utilizada. Essas limitações impediram a automação de testes de interface que utilizam código JavaScript, bem como funcionalidades como o envio de arquivos, as quais continuam dependendo de testes manuais. A primeira limitação poderá ser contornada com a adoção da ferramenta Selenium como complemento ao processo, enquanto a segunda deixará de existir com a eventual migração para versões posteriores do Yii Framework. Mesmo assim, os testes automatizados ainda providenciaram um apoio crucial para a reengenharia, por reduzir bastante a necessidade de testes manuais.

O desenvolvimento dos testes em si também acaba sendo mais complexo do que seria a escrita de um teste funcional tradicional. Normalmente, os testes funcionais do Codeception verificam a existência de textos fixos em elementos HTML específicos, e realizam interações simples com links, botões e elementos de formulário. Dessa forma, a escrita de um teste tende a ser muito fácil e intuitiva. Porém, a necessidade de tornar os testes reusáveis pelo sistema intermediário e final significa que, às vezes, os *page objects* precisam ser bastante inteligentes e flexíveis, contemplando a complexidade e algumas escolhas idiossincráticas do código HTML dos sistemas legados. Isso pode adicionar bastante complexidade à geração dos testes, e pode não ser uma tarefa adequada a um

desenvolvedor de pouca experiência, e pode ser custoso inclusive para um desenvolvedor experiente. Assim, a geração das classes de teste pode ocupar um tempo considerável do processo como um todo. Isso foi especialmente observado nas funcionalidades de solicitação e de visualização de solicitações de fomento, mas mesmo em páginas mais simples existe um esforço associado.

5.4 Considerações Finais

No geral, a reengenharias dos dois módulos foi bem sucedida, e o código resultante tem uma qualidade superior ao código original, demonstrada pela separação de responsabilidades entre as classes, desacoplamento das diferentes camadas do sistema (geração de interfaces, operações no banco de dados, etc.), o uso de recursos nativos do Yii Framework para simplificar operações como a manipulação do banco de dados, e maior facilidade de manutenção. Isso fornece evidências de que este processo de reengenharia tem potencial de ser aplicado em outros sistemas da UFRGS, independente de sua idade e complexidade, e resultará não só em uma melhoria de qualidade dos sistemas, mas no aumento da cobertura de testes e na adoção da cultura de automação de testes e refatoração de código por parte dos desenvolvedores e analistas de desenvolvimento.

6 CONCLUSÃO

A reengenharia de sistemas legados é um tópico importante na área de desenvolvimento de sistemas, devido ao problema da degradação da qualidade de sistemas e a complexidade do processo em si, com a necessidade de planejar os projetos de reengenharia e compreender suas dificuldades e os benefícios a serem obtidos. A existência de software legado na UFRGS, embora seja um desafio às equipes de manutenção, acaba oferecendo uma oportunidade de elaborar um processo de reengenharia adaptado à realidade da universidade.

A necessidade de reengenharia e modernização de tais sistemas já era observada há bastante tempo, e a elaboração de um processo confiável tornava-se cada vez mais importante. Isso está de acordo com os casos observados no estudo bibliográfico deste trabalho, não se tratando, portanto, de uma impressão equivocada da realidade da UFRGS. Outra observação é a de que tais sistemas não se tratam de código morto e obsoleto, mas sim de recursos críticos e frequentemente utilizados, o que aumenta a necessidade de manter a qualidade e a manutenibilidade do código.

Este trabalho apresentou um processo completo e gradual de reengenharia de sistemas legados em PHP, utilizando os recursos disponíveis para as equipes. A validação do processo foi feita em sistemas reais da universidade, fornecendo uma visão realista do seu uso. O desenvolvimento do processo tem embasamento não só no estudo teórico, mas na compreensão da realidade da UFRGS e de seus sistemas, e é orientado especificamente para seu contexto e tecnologia. Isso não significa que o processo não pode ser utilizado por outras instituições que encontram-se em uma realidade semelhante, isto é, com uma grande quantidade de código legado em PHP, com alto nível de dificuldade de manutenção e alteração. O processo foi apresentado à equipe de desenvolvimento da UFRGS, e teve boa aceitação, o que indica sua provável adoção no futuro.

A validação do processo apontou alguns de seus desafios e dificuldades, mas também demonstrou que ele apresenta grandes oportunidades para melhorias extensas e profundas na qualidade do código, e no aprimoramento da cultura de desenvolvimento de software da UFRGS, com a absorção de conceitos como refatoração, melhoria contínua e automação de testes. A substituição dos sistemas legados por versões mais flexíveis e robustas se traduz em economia de tempo e esforço, que podem ser investidos no desenvolvimento de novas soluções, que permitam a evolução contínua dos sistemas de informação da UFRGS, gerando benefícios a toda a comunidade acadêmica.

REFERÊNCIAS

BRODIE, M. L.; STONEBRAKER, M. Darwin: On the incremental migration of legacy information systems. **Distributed Object Computing Group, Technical Report TR-0222-10-92-165, GTE Labs Inc**, v. 28, 1993.

BRUNELIERE, H. et al. Software modernization revisited: Challenges and prospects. **Computer**, v. 48, 08 2015.

BUCHANAN, C. **4 Successful examples of application modernization**. 2019. <https://about.gitlab.com/blog/2019/03/14/application-modernization-examples/>. Acesso em 15/02/2021.

CAUDILL, H. **Lessons from 6 software rewrite stories**. 2019. <https://medium.com/@herbcaudill/lessons-from-6-software-rewrite-stories-635e4c8f7c22>. Acesso em 07/07/2021.

CODECEPTION Changelog. 2021. <https://codeception.com/changelog>. Acesso em 12/07/2021.

FOWLER, M. **PageObject**. 2013. <https://martinfowler.com/bliki/PageObject.html>. Acesso em 06/07/2021.

FOWLER, M. **TechnicalDebt**. 2019. <https://martinfowler.com/bliki/TechnicalDebt.html>. Acesso em 16/07/2021.

GEORGIOU, M. **Why Businesses Should Modernize Their Legacy Applications?** 2019. <https://www.imaginovation.net/blog/why-businesses-should-modernize-legacy-applications/>. Acesso em 15/02/2021.

HASSELBRING, W. et al. The dublo architecture pattern for smooth migration of business information systems: an experience report. In: **Proceedings. 26th International Conference on Software Engineering**. [S.l.: s.n.], 2004. p. 117–126.

KHADKA, R. et al. How do professionals perceive legacy systems and software modernization? In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 36–47. ISBN 9781450327565. Available from Internet: <https://doi.org/10.1145/2568225.2568318>.

KHADKA, R. et al. Does software modernization deliver what it aimed for? a post modernization analysis of five software modernization case studies. In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2015. p. 477–486.

KNOCHE, H.; HASSELBRING, W. Using microservices for legacy software modernization. **IEEE Software**, v. 35, n. 3, p. 44–49, 2018.

LEFF, A.; RAYFIELD, J. Web-application development using the model/view/controller design pattern. In: **Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference**. [S.l.: s.n.], 2001. p. 118–127.

SANTOS, B. M. et al. Software refactoring for system modernization. **IEEE Software**, v. 35, n. 6, p. 62–67, 2018.

WENDLAND, M.-F. et al. Model-based testing in legacy software modernization: An experience report. In: **Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation**. New York, NY, USA: Association for Computing Machinery, 2013. (JAMAICA 2013), p. 35–40. ISBN 9781450321617. Available from Internet: <https://doi.org/10.1145/2489280.2489291>.