UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESTUDO DE CASO SOBRE COMO A INTRODUÇÃO DE COMPORTAMENTOS ADAPTATIVOS EM UMA APLICAÇÃO *WEB* LEGADA IMPACTA A COBERTURA DE CÓDIGO

ROGERIO JERONIMO GENTIL

ORIENTADOR: PROF. DR. FABIANO CUTIGI FERRARI

São Carlos – SP
Setembro/2020

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESTUDO DE CASO SOBRE COMO A INTRODUÇÃO DE COMPORTAMENTOS ADAPTATIVOS EM UMA APLICAÇÃO *WEB* LEGADA IMPACTA A COBERTURA DE CÓDIGO

ROGERIO JERONIMO GENTIL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Fabiano Cutigi Ferrari

São Carlos – SP
Setembro/2020

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Rogerio Jeronimo Gentil, realizada em 16/09/2020.

Comissão Julgadora:

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Prof. Dr. Vinicius Humberto Serapilha Durelli (UFSJ)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Dedico ao meu filho, Dante.

AGRADECIMENTO

À minha esposa, Paula, pelo companheirismo e compreensão, à minha mãe, Margareth, pelo eterno o suporte incondicional, aos meus familiares pelas energias positivas, ao meu orientador, Fabiano, pelos valiosos ensinamentos, contribuição técnica e acadêmica e pelo amparo emocional nos momentos de incertezas e debilidade, aos membros das bancas de qualificação e defesa, Ingrid Oliveira de Nunes – Universidade Federal do Rio Grande do Sul (UFRGS), Vinícius Humberto Serapilha Durelli – Universidade Federal de São João Del-Rei (UFSJ) – e Valter Vieira de Camargo – Universidade Federal de São Carlos (UFSCar), pelas preciosas contribuições, aos meus colegas do Laboratório de Pesquisa em Engenharia de *Software* (LaPES), aos meus colegas de trabalho da Secretaria Geral de Informática (SIn) e à Pró-reitoria de Graduação (ProGrad) da UFSCar pela autorização para o uso do Sistema Integrado de Gestão Acadêmica (SIGA) neste trabalho.

"A ciência não é uma ilusão, mas seria uma ilusão acreditar que poderemos encontrar noutro lugar o que ela não nos pode dar."

Sigmund Freud

RESUMO

Contexto: Sistemas computacionais que se autogerenciam, também conhecidos como sistemas adaptativos, são uma alternativa para liberar profissionais de TI de detalhes de operação e manutenção. Aplicações *web* podem incorporar características de um sistema adaptativo, tais como comportamentos de adaptação em tempo de execução, para responder às necessidades do usuário, ou a mudanças de contexto, de modo a torná-las autônomas. Entretanto, um dos desafios da engenharia de *software* é transformar um sistema legado em um sistema adaptativo, tendo em vista os custos e riscos para substituir um sistema legado. Ademais, como em qualquer processo de evolução de *software*, a introdução de comportamentos adaptativos pode conduzir à inserção de defeitos no sistema legado. Desta forma, as informações de cobertura de código produzida por testes de *software* auxiliam nas atividades relacionadas à manutenção evolutiva do mesmo. **Objetivos:** Neste trabalho foi analisado, por meio da comparação de métricas, como a introdução de comportamentos adaptativos em uma aplicação *web* legada do "mundo real" impactou a cobertura de código. **Método:** Visando dispor de um percentual mínimo de cobertura, para cada cenário implementado foi criado um conjunto de casos de teste na versão legada da aplicação *web*. Em seguida, foram implementadas versões evolutivas que introduziram comportamentos adaptativos à aplicação. As versões (legada e adaptativas) foram organizadas em ramos do sistema de controle de versão de modo que as implementações adaptativas evoluíram seguidamente em duas vertentes tecnológicas. As métricas de cobertura de cada versão da aplicação *web* foram coletadas após a reexecução do conjunto de casos de teste criado e tabuladas para permitir a análise dos dados. **Resultados:** As coberturas de seis versões da aplicação *web* com comportamentos característicos de sistemas adaptativos, além da versão legada, foram analisadas e comparadas entre os respectivos cenários de implementação. **Conclusão:** Os resultados obtidos indicam que o impacto sobre a cobertura total é relevante, onde mesmo as modificações relativamente pequenas podem afetar consideravelmente, de forma negativa ou positiva, os percentuais de cobertura da unidade alterada.

Palavras-chave: Aplicações *web*, teste de software, cobertura de código, sistemas auto-adaptativos, estudo de caso

ABSTRACT

Context: *Self-managing computer systems, also known as self-adaptive systems, are an alternative to get IT professionals free from details of operation and maintenance. Web applications can incorporate characteristics of a self-adaptive system, such as adaptation behavior at runtime, to respond to user needs, or to changes in context, in order to make them autonomous. However, one of the challenges of software engineering is to transform a legacy system into a self-adaptive system, in view of the costs and risks to replace a legacy system. Furthermore, as with any software evolution process, the introduction of adaptive behavior can lead to the insertion of faults in the legacy system. This way, the code coverage information produced by software tests assist in activities related to the evolutionary maintenance of the system.* ***Objective:*** *In this work, we analyzed, through the comparison of coverage metrics, how the introduction of adaptive behaviors in a real world legacy web application impacted code coverage.* ***Method:*** *In order to have a minimum percentage of coverage, for each scenario implemented, a set of test cases was created in the legacy version of the web application. Then, evolutionary versions were develop with the introduction of the adaptive behaviors to the application. The versions (legacy and adaptive) were organized into branches of the version control system so that adaptive implementations have evolved in two technological ways. The coverage metrics for each version of the web application were collected, after re-running the set of test cases created, and tabulated to allow data analysis.* ***Results:*** *The coverage of six versions of the web application with behavior characteristic of self-adaptive systems, in addition to the legacy version, were analyzed and compared between the respective implementation scenarios.* ***Conclusion:*** *The results indicate that the impact on the total code coverage is relevant, where even relatively small changes can considerably affect, either positively or negatively, the percentage of coverage of the changed unit.*

Keywords: *Web applications, software testing, code coverage, self-adaptive systems, case study*

LISTAGENS DE CÓDIGOS

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AdvanSE - *Advanced Research Group on Software Engineering*
BPEL - *Business Process Execution Language*
DC - Departamento de Computação
EAR - *Enterprise Application aRchive*
HTML - *HyperText Markup Language*
HTTP - *HyperText Transfer Protocol*
ID - Identificador
JAR - *Java ARchive*
Java EE - *Java Enterprise Edition*
LaPES - Laboratório de Pesquisa em Engenharia de Software
LDAP - *Lightweight Directory Access Protocol*
LOC - *Lines of Code*
MAPE-K - *Monitor*, *Analyzer*, *Planner*, *Executor - Knowledge-based*
PPGCC - Programa de Pós-graduação em Ciência da Computação
SA - Sistema Adaptativo
SAGUI - Sistema de Apoio à Gestão Universitária Integrada
SGBD - Sistema Gerenciador de Banco de Dados
SIGA - Sistema Integrado de Gestão Acadêmica
SIn - Secretaria Geral de Informática
SOAP - *Simple Object Access Protocol*
TI - Tecnologia da Informação
UFSCar - Universidade Federal de São Carlos
URI - *Uniform Resource Identifier*
URL - *Uniform Resource Locator*
XML - *Extensible Markup Language*
WAR - *Web application ARchive*
WS - *Web Service*
WSDL - *Web Services Description Language*
www - *World Wide Web*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto e Motivação

Com a expansão da rede global de computadores, a *internet*, as mais variadas informações podem ser acessadas de diferentes lugares sob as mais diversas condições de conectividade (Brandon, 2008). Essa condição torna a *World Wide Web* (*www*), ou simplesmente *Web*, um ambiente propício para o consumo de bens e serviços. Diversas áreas de atuação, como o comércio, a indústria, a educação, o governo e o entretenimento, por exemplo, se beneficiam de aplicações de *software* para a *Web* – ou apenas *aplicações web* (Pressman e Lowe, 2009).

As aplicações *web* podem apresentar tanto conteúdo estático (quando o conteúdo é o mesmo, independentemente do acesso) quanto conteúdo dinâmico (quando o conteúdo é gerado a cada acesso) (Alonso et al., 2004). Características como concorrência, disponibilidade, segurança e estética, para não citar outras, diferem uma aplicação *web* de um *software* convencional. Além disso, o escopo e a complexidade deste tipo de aplicação variam amplamente, desde simples páginas pessoais até aplicações corporativas distribuídas através da *internet*, por exemplo.

Neste contexto, a dificuldade de gerenciar aplicações *web* mais modernas e complexas tem exigido dos profissionais de Tecnologia da Informação (TI) habilidades relacionadas à instalação, configuração, ajuste e manutenção desse tipo de *software*. Em sistemas mais complexos, em que há numerosas interconectividades, os limites da capacidade humana são rompidos em relação a antecipar e projetar interações entre os componentes de *software* (Kephart e Chess, 2003). Diante disso, sistemas computacionais que se autogerenciam são uma opção para alterar o foco dos profissionais de TI de modo a isentá-los de detalhes de operação e manutenção no que diz respeito à disponibilidade dos sistemas (Kephart e Chess, 2003). Por conseguinte, as aplicações *web* podem incorporar comportamentos característicos de um Sistema Adaptativo (SA) de modo a torná-las mais autônomas (em inglês, *autonomic*), ou seja,

com o menor número de intervenções humanas possível. Um SA é um sistema que se adapta em tempo de execução, de acordo com as mudanças de necessidade do usuário ou de contexto, e tem como característica fundamental pelo menos uma das seguintes propriedades em algum nível de adaptabilidade: autoconfiguração, autocura, auto-otimização e autoproteção (Kephart e Chess, 2003; Lalanda et al., 2013). Tais características incrementam a complexidade desses tipos de sistemas computacionais, e aplicações *web* são frequentemente utilizadas como objeto de estudo em pesquisas na área Sistemas Adaptativos (Castañeda et al., 2014; Moreno et al., 2018; Raufi, 2011).

Ainda que seja desejável modernizar as operações de TI, tendo em vista a incorporação de comportamentos adaptativos, substituir um sistema legado é um exagero em muitos casos e envolve custos e riscos (Warren, 1999). Assim, um dos desafios da engenharia de *software* é transformar um sistema legado em um sistema adaptativo (Salehie e Tahvildari, 2009) como alternativa à substituição de sistemas. Não diferente de um processo evolucionário de *software*, a introdução de comportamentos adaptativos em um sistema legado altera as particularidades de seu funcionamento, o que pode acarretar a introdução de defeitos também.

Desse modo, assim como em um *software* convencional, a atividade de teste de aplicações *web* tem como intuito revelar defeitos (Myers et al., 2011). Contudo, devido ao dinamismo e heterogeneidade característicos de aplicações *web* modernas, o teste desse tipo de *software* é mais complexo (Delamaro et al., 2016). Tal complexidade decorre do fato de que a atividade de teste de aplicações *web* aborda aspectos que normalmente não são abordados em um *software* convencional, tais como as características de disponibilidade, segurança e desempenho. Além disso, em razão da característica de evolução contínua das aplicações *web*, o teste de regressão é imprescindível como estratégia para garantir que partes do *software* desenvolvidas anteriormente ainda funcionem, e que os códigos recém desenvolvidos produzam o resultado o esperado (de S. Campos Junior et al., 2017).

Alinhado ao teste de regressão, as informações sobre a *cobertura de código* produzida pelos testes, isto é, a porção de código que é exercitada por um conjunto de casos de teste (também é conhecida como *cobertura de testes*), oferecem suporte às atividades relacionadas à manutenção evolutiva de *software*, dentre as quais pode ser citada a análise de impacto (Elbaum et al., 2001). Conforme o *software* evolui, as informações de cobertura de código calculadas para uma versão podem não refletir com precisão a cobertura das versões subsequentes.

Diante deste contexto, e em função da quantidade discreta de trabalhos que exploram a incorporação de comportamentos característicos de SAs em sistemas legados, um estudo de caso exploratório foi conduzido para avaliar o impacto sobre a cobertura de código produzida pelos testes causado pela introdução de comportamentos adapta-

tivos em uma aplicação *web* legada do “mundo real”. Os objetivos almejados para este estudo de caso são definidos de forma mais detalhada na próxima seção.

1.2 Objetivos

O objetivo primário deste trabalho residiu na análise, através da comparação dos dados quantitativos de métricas de cobertura entre as versões, tanto da versão legada quanto das suas respectivas evoluções adaptativas, de como a introdução de comportamentos adaptativos em uma aplicação *web* legada do “mundo real” impactou a cobertura de código. Além disto, neste trabalho buscou-se adicionar comportamentos característicos de SAs intrínsecos ao recurso gerenciado, ao invés de processos de adaptação restritos à infraestrutura de operação, como a autoescalabilidade, por exemplo.

1.3 Organização do Trabalho

Incluindo esta introdução, que contextualiza e apresenta a motivação e o objetivo deste trabalho de mestrado, este documento contém mais cinco capítulos. No Capítulo 2 são apresentados os fundamentos do teste de *software*, de cobertura de código, de aplicações baseadas na *Web* e de sistemas adaptativos. Além disso, alguns trabalhos encontrados na literatura que se relacionam com este trabalho também são apresentados no Capítulo 2. O Capítulo 3 apresenta o referencial metodológico, que inclui o objetivo do estudo, as questões de pesquisa e o fluxo de processo utilizado na condução do estudo de caso, além dos procedimentos, artefatos e métodos utilizados para o desenvolvimento deste trabalho. A execução do estudo de caso é detalhada no Capítulo 4. O Capítulo 5 apresenta a análise dos dados extraídos das atividades experimentais e a discussão dos resultados. Por fim, o Capítulo 6 conclui este trabalho apresentando as contribuições, limitações e trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Antes de detalhar os procedimentos e materiais utilizados na execução deste trabalho para atingir o objetivos especificados no capítulo introdutório, é necessário entender alguns conceitos e definir a terminologia que será utilizada ao longo do texto, uma vez que termos comuns assumem significados particulares quando usados em um contexto teórico. Além disto, é preciso revisitar a literatura para identificar estudos que se relacionam com este trabalho, mas diferem em algum aspecto. Assim sendo, este capítulo almeja propiciar um melhor entendimento sobre os conceitos utilizados nos capítulos subsequentes e apresentar as similaridades e os diferenciais deste trabalho em relação a alguns estudos da literatura.

Desta forma, a Seção 2.2 introduz os conceitos básicos sobre teste de *software*. Tal seção aborda as técnicas de teste, bem como os seus critérios, e as fases do processo de teste. Na Seção 2.3 são abordados conceitos em relação ao teste de regressão. A Seção 2.4 trata sobre o conceito de cobertura de código produzida por testes. Os conceitos básicos sobre aplicações *web* e testes relativos a este tipo de aplicação são apresentados na Seção 2.5. A apresentação dos conceitos básicos encerra-se na Seção 2.6, na qual são abordados os Sistemas Adaptativos. Na Seção 2.7, este capítulo ainda sumariza e compara trabalhos relacionados a este trabalho.

2.2 Conceitos Básicos sobre Teste de Software

A tarefa de desenvolver um *software* pode ser complexa dependendo da dimensão e das características do *software* a ser desenvolvido. Consequentemente, o processo de desenvolvimento torna essa tarefa suscetível a problemas que podem resultar em um produto diferente do especificado. Embora possam ser utilizados métodos e ferramentas da engenharia no desenvolvimento de *software*, erros podem ocorrer. Dentre os fatores que podem ser identificados como causadores desses erros está o erro humano.

Para atenuar erros, uma série de atividades podem ser aplicadas com a finalidade de garantir que o produto esteja em conformidade com o especificado. Tais atividades são divididas em dois conjuntos: Validação e Verificação. Validação se refere a um conjunto de atividades que asseguram que o *software* foi criado de acordo com o especificado. Verificação se refere ao conjunto de atividades que visam a garantir que o *software* está sendo criado corretamente. As atividades de Validação e Verificação podem e devem ser conduzidas durante todo o processo de desenvolvimento do *software*. Um dos elementos dessas atividades é o teste de *software* (Pressman e Maxim, 2016).

O teste de *software* é um processo sistemático que objetiva asseverar que o programa não faz algo indesejado e, por consequência, faz o que foi projetado para fazer. Logo, a atividade de teste tem o intuito de revelar defeitos no *software* (Myers et al., 2011). Quando a execução do *software* não condiz com o esperado, diz-se que foi revelado um defeito (Delamaro et al., 2016). O termo erro pode ser entendido e utilizado de forma distinta pelas pessoas. Contudo, na literatura esse e outros termos relacionados, tais como defeito e falha, têm seus significados bem definidos.

A não conformidade de um *software* é denominada “defeito” (em inglês, *fault*). O defeito pode ocasionar um “erro” (em inglês, *error*). O erro se caracteriza por um estado inconsistente ou inesperado em um determinado instante da execução de um *software*. Tal estado pode causar uma “falha” (em inglês, *failure*). A falha decorre da divergência entre o resultado esperado e o resultado produzido pela execução de um *software*. Tanto o termo erro quanto o termo falha são considerados conceitos dinâmicos, dado que ambos ocorrem e são percebidos apenas quando o *software* está em execução. Por sua vez, o termo defeito é estático, pois sua existência não está associada à execução de um *software* (Delamaro et al., 2016). Embora uma falha possa ocorrer em um determinado estado de um *software*, cenários de uso rotineiro podem deixar um defeito oculto por toda a vida do *software*. Assim sendo, pode-se descrever casos de teste com o objetivo de evidenciar tipos particulares de defeitos.

Para uma determinada execução de um programa, um caso de teste é um par formado por: (i) um dado de teste (entrada) e (ii) um resultado esperado (saída). Um

dado de teste é um elemento do conjunto de todos os valores possíveis que podem ser utilizados como entrada para executar um programa. Esse conjunto de valores é denominado domínio de entrada.

O instrumento que decide se o resultado de uma determinada execução para um dado de teste coincide com o resultado esperado é denominado oráculo de teste (Delamaro et al., 2016).

Idealmente, o processo de teste de *software* deveria fazer uso de todos os casos de teste possíveis. Entretanto, o teste exaustivo (como é conhecido o método de utilização de todos os possíveis valores do domínio de entrada de um programa) é impraticável na maioria dos casos, já que há diversas situações em que o programa pode ter diversas combinações de entradas e saídas por mais simples que seja. A criação de um subconjunto de casos de teste adequado deve então priorizar os elementos de *software* (denominados requisitos de teste) que serão exercitados. Assim sendo, técnicas e critérios devem ser utilizados para definir um subconjunto de casos de testes que vise a ter a mais alta probabilidade de detecção de defeitos (Myers et al., 2011).

2.2.1 Técnicas e Critérios de Teste

Quando feita de modo aleatório, a seleção de dados de teste para a criação de um subconjunto de casos de teste tem a probabilidade reduzida em relação a detecção de defeitos (Myers et al., 2011). Neste sentido, critérios são utilizados para selecionar casos de teste a fim de gerar um subconjunto eficaz e satisfatório. Para identificar quais dados de teste comporão um subdomínio, os critérios são estabelecidos com base nos requisitos de teste (Delamaro et al., 2016). Os critérios de teste se associam a técnicas de teste que utilizam fontes variadas de informação para representar o *software* em teste. As principais técnicas identificadas na literatura são: (i) Funcional; (ii) Estrutural; e (iii) Baseada em Defeitos. Tais técnicas são brevemente descritas a seguir.

Teste Funcional: A técnica de Teste Funcional baseia-se nas especificações dos requisitos do programa e tem o objetivo de encontrar situações nas quais o programa não se comporta de acordo com o especificado (Myers et al., 2011). Nessa técnica, o programa é considerado uma caixa preta, pois os detalhes de implementação não são importantes para o teste. Apesar de ser comumente aplicada ao nível de sistema, essa técnica pode ser aplicada em todos os níveis de teste. Para o teste funcional, as entradas são fornecidas e as saídas geradas são avaliadas para verificar se estão em conformidade com os resultados esperados (Delamaro et al., 2016). Alguns dos principais critérios funcionais são: (i) Particionamento de equivalência; e (ii) Análise de Valor Limite.

O critério de Particionamento de Equivalência almeja reduzir o domínio de entrada para um conjunto finito e ao mesmo tempo eficiente no intuito de revelar defeitos. Nesse critério, o domínio de entrada é dividido em classes (partições) de dados a partir das quais é possível gerar casos de teste. Duas ou mais classes podem produzir o mesmo resultado, indicando que ambas são equivalentes. Uma classe de equivalência representa um conjunto de estados válidos e inválidos para condições de entrada de um programa. Uma condição de entrada pode ser um valor numérico, um intervalo de valores ou uma condição booleana (Pressman e Maxim, 2016).

A Análise do Valor Limite utiliza-se do critério de Particionamento de Equivalência com o intuito verificar os limites das classes de equivalência. Nesse critério são selecionados elementos nas "fronteiras" da classe e casos de testes são obtidos a partir do domínio de saída também (Pressman e Maxim, 2016).

Teste Estrutural: Baseada no conhecimento da estrutura interna do programa, a técnica de Teste Estrutural propõe exercitar os caminhos lógicos do programa (Myers et al., 2011). Nessa técnica o programa é considerado uma caixa branca, uma vez que demanda a execução de partes do programa e os aspectos de implementação são fundamentais para gerar e selecionar os casos de teste (Delamaro et al., 2016). De forma geral, os critérios dessa técnica utilizam um Grafo de Fluxo de Controle (GFC) para representar um programa, no qual os nós e as arestas representam, respectivamente, os blocos de código e os possíveis caminhos lógicos. Os principais critérios dessa técnica são: (i) Todos os Comandos; (ii) Todos os Ramos; e (iii) Todos os Caminhos.

O critério Todos os Comandos (também conhecido como Todos os Nós) requer que a execução do programa passe por todos os nós do grafo pelo menos uma vez. Ou seja, cada comando deve ser executado ao menos uma vez. O critério Todos os Ramos (também conhecido como Todos os Arcos) exige que a execução do programa passe por todos as arestas grafo. Isto é, os desvios que representam o fluxo de controle do programa devem ser executados ao menos uma vez. O critério Todos os Caminhos demanda que todos os possíveis caminhos do programa sejam executados.

Teste Baseado em Defeitos: Por fim, a técnica de Teste Baseado em Defeitos utiliza-se de defeitos típicos introduzidos durante a construção do *software*. Da mesma forma que um defeito pode ser introduzido no *software* de forma não intencional, os casos de testes podem ser ineficientes em não revelar um defeito. O objetivo é derivar os requisitos de teste que podem ser utilizados para revelar defeitos (Delamaro et al., 2016). O principal critério é a Análise de Mutantes. Nesse critério, versões de um programa com pequenas alterações sintáticas de código são utilizadas para verificar a capacidade dos casos de teste em revelar as diferenças comportamentais entre a versão original do programa e as versões alteradas, denominadas "mutantes".

O emprego das técnicas e dos critérios de teste desacompanhados de um conjunto de medidas previamente estabelecidas pode não aumentar a probabilidade de detecção de defeitos em um nível satisfatório, uma vez que cada método tem seus pontos fortes e seus pontos fracos (Myers et al., 2011). Por isso, é importante definir um processo de teste de *software* eficiente.

2.2.2 Processo de Teste e Níveis de Teste

O teste de *software* é um processo amplo e, portanto, deve ser conduzido de forma sistemática. Consequentemente, organizá-lo em etapas torna-o mais lógico. Com esse objetivo, o processo de teste de *software* pode ser dividido em etapas da seguinte forma (Camargo, 2012):

1. Planejamento: etapa na qual é criado um plano de teste que visa a definir quais são os objetivos e como atingi-los em cada etapa do processo. Ademais, o plano deve conter informações sobre o escopo, a abordagem a ser seguida, os riscos inerentes, os recursos necessários (recursos humanos, máquinas - *hardware*, ferramentas - *software*, etc.), dentre outros.

2. Configuração de Dados e de Ambiente: etapa em que são implementados e configurados os dados e ajustados os itens necessários para que os casos de teste sejam executados (por exemplo, máquina - virtual ou física, *container*, Sistema Gerenciador de Banco de Dados (SGBD), dentre outros).

3. Projeto: etapa na qual as técnicas e os critérios definidos no plano de teste são utilizados para especificar os casos de teste. É importante destacar que a especificação dos casos de teste pode demandar a revisão dos requisitos, por exemplo, nas situações em que a especificação dos requisitos esteja incompreensível ou dúbia. A rastreabilidade entre os casos de teste e os requisitos verificados por eles também é um ponto importante a se manter nessa etapa.

4. Execução e Avaliação: etapa em que os casos testes são aplicados e as revelações de defeitos do programa são registradas de acordo com a avaliação do oráculo.

5. Monitoramento e Controle: etapa na qual o progresso das atividades de teste desempenhadas são comparadas com o plano de teste a fim de identificar necessidades de alteração no plano.

Níveis de Teste: Embora possam existir várias estratégias de teste de *software*, as quais também fazem parte do Plano de Teste, muitas equipes têm adotado uma

abordagem incremental (Pressman e Maxim, 2016). Nessa abordagem, os testes são implementados em níveis, verificando desde a menor parte até a completude do *software*. Os níveis de teste são definidos em termos de unidade, de integração e de sistema.

Normalmente considerado um processo auxiliar na etapa de codificação, o teste em nível de unidade tem o foco na verificação da menor unidade de um programa (Pressman e Maxim, 2016). Entende-se por unidade uma função, um procedimento, um método ou mesmo uma classe (Delamaro et al., 2016). O objetivo do teste de unidade é revelar defeitos relacionados a algoritmos incorretos, estruturas de dados mal implementadas ou simples erros de programação (como enganos na digitação de um código de programa). Como cada unidade é testada separadamente, a construção do teste pode ocorrer de forma incremental antes mesmo da conclusão do *software*.

Uma vez que cada unidade funciona de forma adequada individualmente, não é necessariamente verdade que elas continuarão a funcionar adequadamente quando combinadas. À medida que as diversas partes do *software* são executadas de forma conjunta, é preciso verificar se a integração entre as partes funciona de acordo com o especificado e sem falhas. O teste em nível de integração é a forma sistemática para verificar essa combinação de unidades.

O teste em nível de sistema pode ser aplicado quando o *software* está completo. Esse nível de teste objetiva validar se os requisitos foram implementados corretamente e explora tanto aspectos não funcionais, tais como segurança e desempenho, quanto funcionais (Delamaro et al., 2016).

2.3 Teste de Regressão

Cada nova adição ou cada alteração (melhoria ou correção, por exemplo) realizada no *software* gera uma nova versão do mesmo. Ainda que se adote um processo meticuloso e os testes sejam implementados em níveis, efeitos colaterais associados à evolução do *software* podem fazê-lo retroceder. O que funcionava corretamente na versão anterior pode apresentar falhas na nova versão. Nesse contexto, o teste de regressão é a estratégia de reexecução de um subconjunto de casos de teste (Myers et al., 2011) para verificar se um novo defeito foi introduzido.

Muitos tipos de testes podem ser incorporados ao conjunto de testes de regressão, incluindo testes de unidade, de integração e de sistema. Conforme o *software* evolui, consequentemente o conjunto (ou suíte) de testes tende a crescer, elevando os custos para executá-lo inteiramente (Yoo e Harman, 2012). Algumas abordagens têm sido estudadas para determinar o subconjunto de casos testes que deve ser reexecutado

com eficiência, que seja financeiramente viável e com baixo consumo de tempo. Dentre as abordagens que têm sido estudadas no processo de teste de regressão, Yoo e Harman (2012) destacam: (i) Minimização da Suíte de Testes (em inglês, *Test Suite Minimization*); (ii) Seleção de Casos de Teste (em inglês, *Test Case Selection*); e (iii) Priorização dos Casos de Teste (em inglês, *Test Case Prioritization*).

Minimização da Suíte de Testes: Também conhecida como Redução de Suíte de Testes (em inglês, *Test Suite Reduction*), a técnica de Minimização visa a identificar casos de teste obsoletos ou redundantes e removê-los da suíte de testes com a finalidade de reduzir seu tamanho. Segundo Yoo e Harman (2012), essa técnica é definida como: dados uma suíte de testes, *T*, um conjunto de requisitos de testes {r_1, r_2, ..., r_n} que deve ser satisfeito para proporcionar o teste adequado de um programa, e subconjuntos de *T*, T_1, T_2, ..., T_n, associados com cada r_i, tal que qualquer um dos casos de teste t_j pertencente a T_i pode ser usado para satisfazer o requisito r_i. A questão é focada em determinar um subconjunto representativo, T', de casos de testes de *T*, que satisfaça todos os requisitos de teste, r_i. O critério de teste é satisfeito quando cada requisito de teste do conjunto é satisfeito. Por exemplo, considerando-se o cenário no qual a suíte de testes, composta por casos de teste t_i, que satisfaça os requisitos de teste, r_i, conforme a Tabela 2.1, um subconjunto mínimo dessa suíte que satisfaz todos os requisitos de teste é composto pelos casos de teste t_2 e t_3.

Tabela 2.1: Exemplo Adaptado de Yoo e Harman (2012) de um Cenário para Seleção de Casos de Teste

Requisitos de Teste / Casos de Teste	r_1	r_2	r_3	r_4	r_5
t_1		✓			✓
t_2	✓		✓		
t_3		✓		✓	✓
t_4	✓			✓	

Seleção de Casos de Teste: A Seleção de Casos de Teste é a técnica que busca identificar um subconjunto de casos que serão utilizados para testar partes modificadas do programa em relação à versão que funcionava. Formalmente, o problema de seleção é definido como (Yoo e Harman, 2012): dados um programa *P*, a versão modificada de *P*, P', e uma suíte de teste, *T*, a questão consiste em encontrar um subconjunto de *T*, T', com o qual P' pode ser testado.

Priorização dos Casos de Teste: Por fim, a técnica de Priorização dos Casos de Teste visa a ordenar a sequência de execução dos casos de teste de forma ideal de tal

modo que a detecção de defeito seja antecipada ao máximo. Nessa técnica, assume-se que todos os casos de teste podem ser executados na ordem da sequência produzida. Entretanto, o processo de teste pode ser encerrado de forma arbitrária em algum ponto. Mais formalmente, essa técnica é definida da seguinte forma (Yoo e Harman, 2012): dada uma suíte de testes, *T*, um conjunto de permutações de *T*, *PT*, e uma função de *PT* para números reais $f : PT \rightarrow \mathbb{R}$:, a questão é encontrar T' pertencente a *PT* tal que $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$. Nessa definição, a função f, aplicada a qualquer ordem da suíte de testes, produz um valor para a ordem (Rothermel et al., 2001). O valor produzido pode ser compreendido como um menor número de casos de teste executados ou um tempo menor de execução da suíte de testes para detectar defeitos, por exemplo. Para ilustrar a técnica, considere o cenário no qual uma suíte de testes, composta por casos de testes t_i, revele defeitos, D_i, conforme a Tabela 2.2.

Tabela 2.2: Exemplo Adaptado de Rothermel et al. (1999) de um Cenário para Priorização dos Casos de Teste

Casos de Teste \ Defeitos	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}
t_1		✓								✓
t_2							✓	✓	✓	
t_3	✓			✓			✓			✓
t_4	✓	✓	✓	✓	✓					
t_5						✓				✓

Pertinente a técnica de priorização, um critério comum utilizado para ordenar casos de teste é a detecção de defeitos (de S. Campos Junior et al., 2017). Dessa forma, considere a seguinte ordenação dos casos de teste: t_1–t_2–t_3–t_4–t_5. Ao executar essa suíte de testes nessa ordem, a taxa percentual de detecção de defeitos após executar o caso de teste t_1 é de 20%. Ou seja, dois dentre dez defeitos foram detectados. Após executar o caso de teste t_2, 50% dos defeitos são detectados. Seguindo com a execução dos casos de teste nessa ordenação, a taxa de detecção de defeitos atingirá 100% apenas após a execução do caso de teste t_5. Isto é, são necessários cinco casos de testes para detectar todos os defeitos. Por outro lado, se a sequência de execução dos casos de teste for t_4–t_2–t_5–t_3–t_1, em que os casos de teste t_4, t_2 e t_5 são priorizados nesta ordem, a taxa de detecção de defeitos atingirá 100% após a execução desses três casos de testes.

2.4 Cobertura de Código

O conceito de *cobertura de código* – também conhecido como *cobertura de teste* (Cornett, 2014) – é empregado à porção de código de um programa que é exercitada por um conjunto de casos de teste (Graham et al., 2008). Este conceito está associado à etapa de Monitoramento e Controle do processo de teste de *software*, tendo em vista que os artefatos produzidos nesta atividade possibilitam analisar a eficiência do conjunto de casos teste.

O processo de análise da cobertura de código visa a procurar áreas de código que não são exercitadas por um subconjunto de casos de teste, criar casos de teste que objetivem aumentar a cobertura, e determinar a quantidade de código coberto. Neste sentido, a cobertura de código é pertinente à qualidade do conjunto de testes, não à qualidade do código.

A cobertura de código pode ser medida com base em um número de diferentes elementos estruturais em um *software*, em diferentes níveis como, por exemplo, unidade, integração e sistema (Graham et al., 2008). A medida de cobertura é frequentemente expressa em valores percentuais; contudo, dados sobre os elementos individuais do programa ou casos de teste também podem ser apresentados como valores detalhados (Tengeri et al., 2016). Ademais, a cobertura de código pode ser considerada uma medida de abrangência do conjunto de casos de teste. Todavia, há diversas métricas de cobertura bem conhecidas que fornecem dados para adequar o conjunto de casos de teste (Ivanković et al., 2019). As duas principais métricas de cobertura são:

Cobertura de Comandos (em inglês, *Statement Coverage*): Embora existam ferramentas e métodos que contabilizem os comandos de formas diferentes, de modo geral, o princípio básico desta métrica considera cada linha como um comando (Graham et al., 2008).

Cobertura de Decisão (em inglês, *Decision Coverage*): Essa métrica considera cada estrutura de controle como um ponto de decisão, onde há duas ou mais possibilidades de saída ou resultados (Graham et al., 2008).

No que diz respeito ao processo evolutivo de um *software*, as informações de cobertura podem auxiliar em algumas atividades, tais como análise de impacto, avaliações de adequação de teste, minimização da suíte de testes e priorização dos casos de teste (Elbaum et al., 2001).

2.5 Conceitos Básicos sobre Aplicações *Web*

Criada com o propósito de permitir o compartilhamento de conhecimento e recursos sobre uma rede de computadores (Shklar e Rosen, 2003), a *World Wide Web* (também conhecida como *W3* ou simplesmente *Web*) tornou-se onipresente com a expansão da rede global de computadores, a *internet* (Brandon, 2008; Ginige e Murugesan, 2001). Essa realidade permite que pessoas acessem e compartilhem diversas informações dos mais remotos lugares (Alonso et al., 2004), nas mais variadas condições, e torna a *Web* um meio factível para o consumo de bens e serviços e uma tecnologia indispensável para áreas como o comércio, a indústria, a educação, o entretenimento, a ciência, e a financeira (Pressman e Lowe, 2009).

A *Web* foi originalmente criada como uma biblioteca virtual, na qual um documento pode ser acessado como um recurso por um endereço único, denominado *Uniform Resource Locator* (URL). Um documento pode ser referenciado em outros documentos por *links* de hipertexto. Hipertexto é um termo que foi cunhado para descrever textos que ramificam, que levam a outras partes do documento ou outros documentos, e pode ser estendido sob o nome de hipermídia para referenciar documentos disponibilizados em outros formatos além de texto, tais como imagem, vídeo e som (Shklar e Rosen, 2003). Os documentos podem ser descritos semanticamente usando-se uma linguagem de marcação de hipertexto, denominada HTML (*HyperText Markup Language*), que especifica como a hipermídia deve ser exibida por navegadores *Web* (Alonso et al., 2004).

Com relação ao modelo arquitetural, a *Web* foi projetada como uma arquitetura cliente-servidor. Nesse modelo, um cliente faz uma requisição de uma informação para um servidor que devolve uma resposta para o cliente. A informação, trafegada na forma de documentos entre cliente e servidor, é normalizada pelo protocolo genérico conhecido como HTTP (*HyperText Transfer Protocol*). No tráfego de informações, os documentos são reconhecidos por identificadores denominados URIs (*Uniform Resource Identifiers*). O tráfego de informação pode ser estático, quando o próprio recurso é devolvido, ou dinâmico, quando o conteúdo do documento é gerado a cada acesso (Alonso et al., 2004).

Ainda que existam aplicações *web* simples como páginas com conteúdos estáticos, tal como nos primórdios da *World Web Wide*, as aplicações *web* atuais são complexas por serem dinâmicas, distribuídas e por se integrarem a sistemas gerenciadores de banco de dados, aplicações de indexação e recuperação de informação e até mesmo outras aplicações *web*.

Mesmo com padrões e tecnologias estabelecidos, a definição sobre o que exatamente é uma aplicação *web* não é constante ao longo do tempo (Abildgaard, 2018).

Shklar e Rosen (2003) definem uma aplicação *web* como uma aplicação cliente-servidor que usa um navegador de *Web* como um programa cliente, executa um serviço interativo conectando-se com servidores pela *internet* ou *intranet* e que entrega conteúdos dinamicamente de acordo com os parâmetros da requisição, comportamentos de usuários e considerações de segurança. Abildgaard (2018) sugere uma definição mais moderna e abrangente:

> "Uma aplicação *web* é uma aplicação cliente-servidor com um número qualquer de clientes e servidores, na qual a comunicação cliente-servidor acontece via HTTP, em que tanto cliente quanto servidor podem ser ambientes de execução e onde o servidor pode ser qualquer sistema arbitrariamente complexo em si".

Em suma, uma aplicação *web* pode ser considerada como um *software* que depende da *Web* para sua execução, contudo suas características se diferenciam de um *software* convencional. Pressman e Lowe (2009) descrevem algumas características que são encontradas na grande maioria deste tipo aplicação:

- **Intensidade de Rede**: toda aplicação *web* está presente em um rede de computadores e deve servir a uma variedade de clientes.
- **Concorrência**: as ações de um ou muitos usuários pode ter impacto sobre informações apresentadas a outros usuários ou suas ações, dado que diversos usuários podem acessar a aplicação *web* ao mesmo tempo.
- **Carga Imprevisível**: o número de usuários que utilizam a aplicação *web* pode variar em determinados períodos de tempo.
- **Desempenho**: o tempo de resposta para apresentação e processamento por parte de uma aplicação *web* é importante para percepção de utilização do usuário.
- **Disponibilidade**: uma aplicação *web* demanda acessos de modo contínuo, mesmo que 100% de disponibilidade pareça pouco razoável.
- **Orientada a Dados**: a função primária de muitas aplicações *web* é usar hipermídia para apresentar conteúdo de texto, gráfico, áudio e vídeo. Além disso, aplicações *web* são comumente usadas para acessar informações em bases de dados.
- **Sensibilidade ao Conteúdo**: a qualidade e a natureza estética do conteúdo são fatores importantes da qualidade de uma aplicação *web*.
- **Evolução Contínua**: não é incomum que aplicações *web* tenham seu conteúdo atualizado frequentemente e as próprias aplicações evoluam de forma incremental.

- **Imediatismo**: em muitos domínios de aplicação, a necessidade de levar o *software* ao mercado rapidamente demanda o uso de métodos de planejamento, análise, projeto, implantação e testes adaptados ao tempo reduzido requerido para o desenvolvimento de aplicações *web*.

- **Segurança**: dado que as aplicações *web* são acessíveis por uma rede, é difícil, se não impossível, limitar o conjunto de usuários que podem acessar a aplicação. Nesse contexto, medidas de segurança robustas devem ser implementadas na aplicação, assim como por meio da infraestrutura que a sustenta, para proteger conteúdos confidenciais e prover modos seguros de transmissão de dados.

- **Estética**: a aparência é uma parte inegável do apelo de uma aplicação *web*. A estética de uma aplicação pode estar relacionada com receita gerada para o qual ela foi projetada.

Além das características que diferem uma aplicação *web* de um *software* convencional, o escopo e a complexidade dessas aplicações variam amplamente. Os escopos variam desde aplicações de pequena escala até aplicações corporativas distribuídas através da *internet*, bem como por meio de *intranets* e *extranets* corporativas. Nota-se, dessa forma, que as aplicações *web* possuem uma vasta variedade de funcionalidades e têm diferentes requisitos (Brandon, 2008).

Apesar de não existir uma categorização de aplicações *web* que seja amplamente aceita, é possível identificar na literatura um classificação baseada em suas funcionalidades, conforme apresentado na Tabela 2.3.

Tabela 2.3: Categorias de aplicações *web*, extraídas dos trabalhos de Ginige e Murugesan (2001) e Brandon (2008).

Categoria	Exemplos
Informativa	Aplicações de notícias, de catálogo de produtos, de classificados, entre outras.
Interativa	Formulários de registro, jogos *online*, entre outros.
Transacional	Aplicações financeiras como bancos e meios de pagamento, comércio eletrônico – bens e serviços, entre outras.
Orientada a Fluxo	Aplicações de agendamento, planejamento, gestão de inventário, gestão cadeia de suprimentos, entre outras.
Ambientes de Trabalho Colaborativo	Sistemas de autoria distribuída, ferramentas de *design* colaborativas, entre outras.
Comunidades e *Marketplaces*	Sistemas de recomendação, grupos de discussão, sistemas de leilões, entre outros.

Embora grande parte das aplicações *web* sejam destinadas à interação humana, algumas são concebidas para serem acessíveis por outras aplicações através da *Web*. Um serviço *web* (em inglês, *web service*) é uma aplicação de *software* modular orientada à *internet* que pode ser identificada por um URI, da qual *interfaces* e *bindings* são capazes de serem definidos, descritos e descobertos. De forma mais lúcida, um serviço *web* pode ser compreendido como um modo de expor uma funcionalidade de um sistema que pode ser invocada através de padrões de tecnologia *Web* (Alonso et al., 2004). Serviços *web* suportam interações que são descritas semanticamente usando uma linguagem denominada WSDL (*Web Services Description Language*). A WSDL é uma linguagem descritiva baseada no formato XML (*Extensible Markup Language*), uma linguagem de marcação que define um conjunto de regras para codificar documentos que possam ser lidos tanto por humanos quanto por máquinas. Informações sobre como um serviço pode ser invocado, quais parâmetros são esperados e quais estruturas de dados são retornadas são providas por um serviço *web* no formato WSDL. As interações com os serviços *web* são normalizadas pelo protocolo SOAP (*Simple Object Access Protocol*), um protocolo para troca de informações estruturadas baseado no protocolo HTTP.

Mesmo que as aplicações *web* possam ser desenvolvidas usando uma abordagem orientada a serviço, normalmente um serviço *web* fornece uma quantidade limitada de funcionalidades. Para implementar cenários de processos de negócio mais flexíveis e complexos, serviços *web* podem ser compostos. Popularmente, a composição de serviços *web* utiliza uma linguagem para especificação de processos de negócio que estende o modelo de interação entre serviços *web* e a ordem de execução entre essas interações, denominada BPEL (*Business Process Execution Language*) (Sun et al., 2018). Dessa forma, aplicações que agrupam um conjunto de serviços *web* que precisam ser orquestrados são conhecidas como aplicações BPEL ou WS-BPEL.

Assim como em um software convencional, é importante que padrões e práticas sejam exercidos para atender as demandas de desenvolvimento de maneira eficiente em aplicações baseadas na *Web*. Dentre as práticas, a atividade de teste é destacada a seguir.

2.5.1 Teste de Aplicações *Web*

Ainda que os objetivos de teste de aplicações *web* sejam os mesmos de um *software* convencional (Brandon, 2008), o teste de aplicações *web* modernas é mais complexo em razão da heterogeneidade e do dinamismo característicos desse tipo de programa (Delamaro et al., 2016). Isso porque a estratégia de teste para este tipo de aplicação aborda diversos aspectos, tais como desempenho, segurança, disponibilidade, interoperabilidade e escalabilidade, que não são abordados em um *software* convencional.

As aplicações *web* possuem várias dimensões de qualidade. As dimensões mais relevantes e que ajudam a entender os objetivos e a estratégia de teste[1] no contexto desse tipo de aplicação são (Pressman e Maxim, 2016):

- **Conteúdo**: são avaliados os níveis sintático e semântico. No nível sintático, são examinados ortografia, pontuação e gramática em documentos baseados em texto. No nível semântico, são verificados a exatidão da informações apresentadas, a consistência e a ausência de ambiguidade.
- **Função**: testes são realizados para descobrir defeitos que indiquem a não conformidade com os requisitos estabelecidos. Cada função é avaliada em relação a exatidão, instabilidade e conformidade geral com os padrões apropriados de implementação.
- **Estrutura**: é avaliada de forma a certificar que a extensão e a manutenibilidade sejam mantidas na medida em que novas funcionalidades são acrescidas.
- **Usabilidade**: é testada de modo a garantir que a interface esteja adequada aos vários tipos de usuários e que a navegação esteja intuitiva.
- **Navegabilidade**: é testada para verificar que toda a sintaxe e semântica de navegação sejam avaliadas para descobrir links inativos ou incorretos.
- **Desempenho**: testes são realizados sob diversas condições de configuração, operação e carga para verificar se níveis de degradação inaceitáveis são atingidos.
- **Compatibilidade**: é testada em diferentes configurações de hospedagem da aplicação, tanto no lado do cliente quanto no lado do servidor.
- **Interoperabilidade**: é testada para garantir a integração adequada com outras aplicações, tais como, por exemplo, sistemas gerenciadores de banco de dados e sistemas de indexação e recuperação de informação.
- **Segurança**: é testada para investigar e tentar explorar vulnerabilidades em potencial de modo que cada penetração bem sucedida é considerada uma falha de segurança.

Além das dimensões de qualidade, o uso de diferentes tecnologias e as várias maneiras pelas quais os componentes de *software* se comunicam em aplicações *web* influenciam na complexidade do processo de desenvolvimento e na atividade de teste

[1] Nota-se que alguns itens que devem ser avaliados, e as respectivas avaliações sugeridas, estão mais alinhados com atividades de *inspeção* de software do que com teste de software. Exemplos são os itens relacionados a *Conteúdo* e *Estrutura*. A despeito disso, todas as atividades listadas pelos autores originais (Pressman e Maxim, 2016) foram incluídas na lista a seguir.

desse tipo de aplicação. Apesar disso, algumas abordagens de teste aplicadas em *software* convencional, tais como o teste estrutural e teste baseado em defeitos, podem ser adaptadas ou até mesmo modificadas ao contexto de aplicações *web* (Delamaro et al., 2016).

Além da técnica de teste funcional, o teste de nível de sistema em aplicações *web* pode ser alvo de testes não funcionais. Alguns testes não funcionais comuns estão relacionados à questão de verificação de desempenho, *stress* e usabilidade (Brandon, 2008). O teste de desempenho verifica a conformidade da aplicação em relação às especificações de desempenho, como verificar o tempo de resposta da aplicação a uma determinada ação do usuário, por exemplo. O teste de *stress* verifica o comportamento da aplicação em situações extremas ou anormais de carga até o ponto em que a aplicação deixa de funcionar. O teste de usabilidade visa a avaliar, entre outros aspectos, a habilidade do usuário em aprender a operar a aplicação. Em alguns casos, um especialista observa o humano interagir com o aplicação para avaliar a usabilidade.

2.6 Conceitos Básicos sobre Sistemas Adaptativos

A complexidade de sistemas de *software* modernos fez que com que tais sistemas demandassem profissionais de Tecnologia da Informação (TI) com as mais diversas habilidades, tais como instalar, configurar, ajustar e manter. Além disso, a dificuldade de gerenciar sistemas computacionais vai além da administração individual de ambientes de *software*. Sistemas de *software* modernos integram-se com diversos ambientes heterogêneos, estendendo-se para além dos limites de uma organização. Sistemas computacionais complexos, em que há numerosas interconectividades, rompem os limites da capacidade humana no que diz respeito a antecipar e projetar interações entre os componentes de *software* (Kephart e Chess, 2003). Neste contexto, a opção por sistemas computacionais que se autogerenciam altera o foco e objetivos dos profissionais de TI, liberando-os de detalhes de operação e manutenção do sistema em relação a sua disponibilidade (Kephart e Chess, 2003).

Um sistema que se adapta em tempo de execução para operar de acordo com as mudanças nas necessidades do usuário ou de contexto é denominado Sistema Adaptativo (SA). Um SA pode possuir grandes variações de configurações e adaptações em tempo de execução, tanto que há diferentes tipos de SAs e diversos níveis de adaptabilidade. Comumente, considera-se que estes tipos de sistemas podem possuir quatro características fundamentais (Kephart e Chess, 2003; Lalanda et al., 2013):

- **Autoconfiguração** (em inglês, *Self-Configuration*): capacidade de um sistema configurar e reconfigurar seus parâmetros internos, adaptando-se às mudanças de um ambiente dinâmico.

- **Autocura** (em inglês, *Self-Healing*): capacidade de um sistema ser resiliente, ou seja, descobrir que não está operando corretamente, e fazer as mudanças necessárias para restaurar-se para um estado normal ou desejado de modo a aumentar sua disponibilidade.

- **Auto-otimização** (em inglês, *Self-Optimization*): capacidade de um sistema se esforçar proativamente em busca de oportunidades para otimizar suas próprias operações e aumentar sua eficiência.

- **Autoproteção** (em inglês, *Self-Protecting*): capacidade de um sistema antecipar, identificar e prevenir diversos tipos de ameaças, tornando-o menos vulnerável diante das políticas de segurança e preservando sua integridade.

Em relação aos conceitos arquiteturais para descrever sistemas adaptativos, o modelo MAPE-K é amplamente utilizado (Klöes et al., 2017; Lalanda et al., 2013). Esse modelo combina propriedades como a separação de interesses e a escalabilidade, e seu propósito é indicar as principais funções que um gerenciador autônomo deve prover para administrar um sistema (Lalanda et al., 2013). Além disso, o MAPE-K é um modelo de referência de arquitetura modular que define claramente dois tipos de módulos: recursos gerenciados e gerenciador autônomo. Recursos gerenciados são as entidades de *software* e *hardware* que podem ser automaticamente administradas. Exemplos são um servidor *Web*, um sistema operacional, um *cluster* de máquinas, e uma unidade central de processamento (CPU). Gerenciador autônomo é a entidade responsável por administrar os recursos gerenciados em tempo de execução.

Ambos os módulos do modelo MAPE-K se comunicam por intermédio de interfaces providas pelo recurso gerenciado para seu controle e sua adaptação. Sensores (em inglês, *sensors*) e atuadores (em inglês, *effectors*) são os pontos de controle de um recurso gerenciado (Lalanda et al., 2013). Sensores fornecem informações sobre os recursos gerenciados, como o estado de seus elementos. Para um servidor *Web*, essa informação poderia ser os dados sobre o uso de disco ou rede, a utilização de memória ou CPU, ou mesmo o tempo de resposta para a requisição de um cliente. Atuadores fornecem maneiras para ajustar os recursos gerenciados e, consequentemente, modificar seu comportamento. A modificação de um arquivo de configuração ou a substituição de elementos desatualizados são exemplos de ações aplicáveis por um atuador.

O MAPE-K é composto por quatro componentes básicos (***M**onitor*, ***A**nalyze*, ***P**lan* e ***E**xecute* – em português, Monitor, Analisador, Planejador e Executor) e um componente

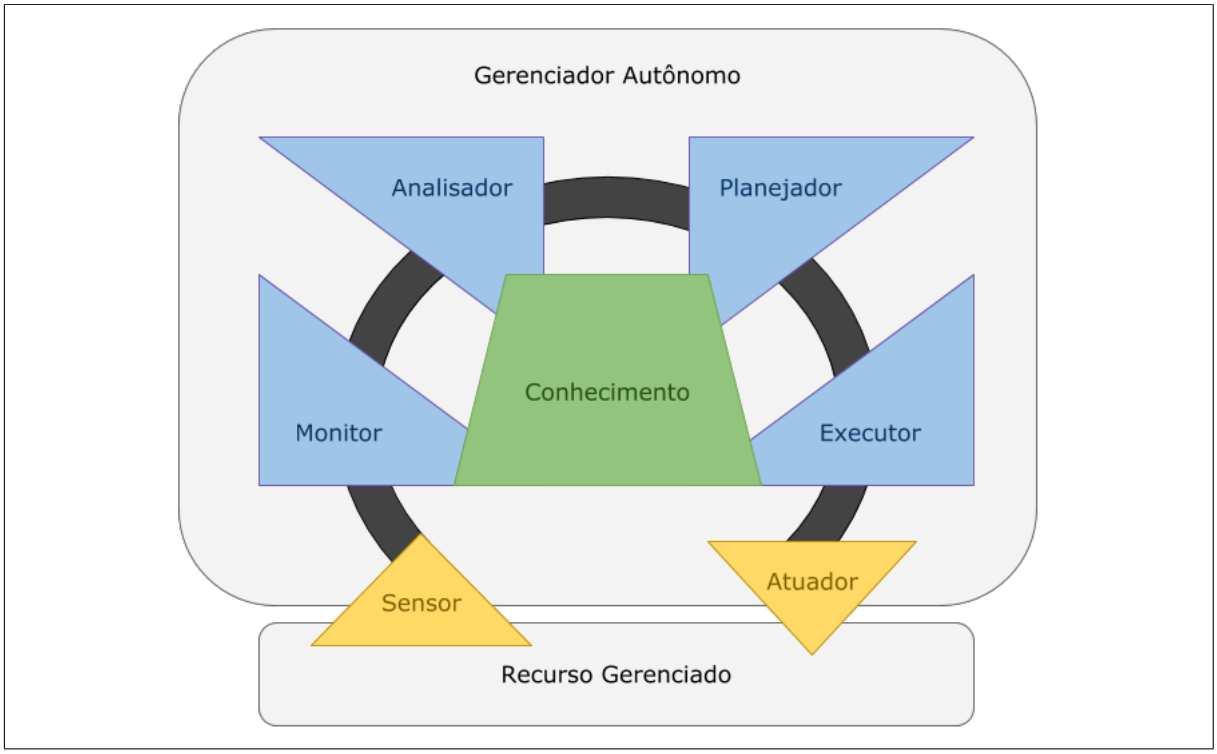

Figura 2.1: Estrutura Geral do Modelo MAPE-K - Adaptada de IBM (2005)

de conhecimento (***K**nowledge* – em português, Conhecimento) (IBM, 2005). O componente Monitor provê o mecanismo que coleta, agrega, filtra e transforma informações adquiridas do recurso gerenciado com o propósito de construir um modelo de contexto de execução. O componente Analisador provê o mecanismo que correlata e modela situações complexas como, por exemplo, previsão de séries temporais e modelos de filas, a fim de avaliar situações e determinar quaisquer anomalias ou problemas, ajudando a prever situações. Usando políticas de informação para guiar seu trabalho, o componente Planejador provê o mecanismo que desenvolve ações para alcançar as metas e objetivos estabelecidos com a finalidade de determinar um conjunto de ações que permitam a passagem de um estado atual para um estado desejado, conforme definido pelas atividades de monitoramento e análise. O componente Executor provê o mecanismo que controla a execução de um plano com considerações para atualizações dinâmicas. O componente Conhecimento provê o mecanismo com informações sobre o sistema e seu contexto. Ambos os componentes podem ser representados com parte de um Gerenciador Autônomo e trabalham juntos para prover a funcionalidade de *loop* de controle. Ou seja, o gerenciador autônomo percebe a situação de contexto interno ou externo, determina ações de gerenciamento e conduz suas execuções, nas quais a análise depende do monitoramento, o planejamento depende da análise, a execução depende do planejamento e todos dependem do conhecimento (Lalanda et al., 2013). A Figura 2.1 ilustra de forma generalizada a estrutura da arquitetura referente ao modelo MAPE-K.

As adaptações em SA podem ser efetuadas, por exemplo, em função do tempo ou pelo disparo de um evento (de la Iglesia, 2014). Na primeira abordagem, um temporizador é responsável por iniciar o processo de transição de comportamento após um determinado tempo. Comumente, essa abordagem usa informações armazenadas na base de conhecimento compartilhada. Na segunda abordagem, um mecanismo em um estado inicial aciona a transição de comportamento para um estado desejado.

2.7 Trabalhos Relacionados

Na literatura é possível identificar os trabalhos que avaliam a incorporação de comportamentos adaptativos em sistemas legados – sistemas que não foram projetados para assimilar características da computação autônoma. Para identificar alguns desses trabalhos, realizou-se uma busca *ad hoc* (isto é, não sistemática) utilizando-se o mecanismo de busca Google Scholar[2]. O conjunto de palavras-chave utilizado na busca foi composto pelos seguintes vocábulos: *evolving legacy system to self-adaptive system*. A partir dos resultados apresentados pelo mecanismo de busca, alguns trabalhos foram selecionados pelo título e pelo teor do resumo. Posteriormente, a estratégia de *snowballing* (Wohlin, 2014), tanto *backward snowballing* quanto *forward snowballing*, com um nível de recursão foi aplicada ao conjunto de trabalhos coletados para encontrar outros trabalhos relacionados. Ao final desse procedimento, seis trabalhos foram selecionados. Tais trabalhos são listados na Tabela 2.4 e serão sintetizados a seguir.

O Trabalho de Zhang e Cheng (2007)

Zhang e Cheng (2007) propuseram uma abordagem dirigida a modelagem UML para introduzir adaptações dinâmicas em sistemas legados não adaptativos. Na abordagem, a partir de uma análise de requisitos, um conjunto de programas Java legados não adaptativos são selecionados e traduzidos em diagramas de estado. Para cada programa, também são criados modelos de adaptação nos termos dos diagramas de estado. Tanto o modelo do programa quanto o modelo da adaptação são traduzidos para modelos formais e verificados utilizando-se ferramentas de análise. Por fim, os modelos são implementados para tornar o código legado adaptável e a implementação resultante é adicionada à base de código. Além disso, uma técnica orientada a aspectos é aplicada para alcançar a separação de interesses na implementação, mantendo o código para adaptação separado do código legado. Os autores relataram que a abordagem foi empregada em aplicações para dispositivos móveis e concluíram que

[2] `http://scholar.google.com` – acessado em agosto, 2020.

Tabela 2.4: Trabalhos Relacionados

Título	Autor(es)	Sistema Alvo	Objetivo
Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation	Zhang e Cheng (2007)	Aplicações experimentais para dispositivos móveis.	Propor uma abordagem dirigida a modelo para introduzir a adaptação dinâmica em sistemas legados não adaptativos, mantendo as propriedades de garantia.
Evolving Software Systems Towards Adaptability	Amoui (2009)	Uma aplicação de telefonia orientada a serviços e um sistema de controle de tráfego aéreo.	Propor uma abordagem sistemática e econômica para a evolução de um *software* existente através de um processo para auxiliar na adição de recursos adaptativos e de acordo com os requisitos de adaptação.
Autonomic Software Systems: Developing for Self-Managing Legacy Systems	Mulcahy e Huang (2014)	Uma aplicação empresarial legada de uma empresa varejista do ramo de instrumentos musicais.	Descrever um projeto de integração onde a reutilização de código, a arquitetura orientada a serviços e abordagens da computação autônoma foram empregadas para estender um sistema legado a fim de minimizar a quantidade de pessoas envolvidas no processo entre a compra *online* e a preparação para a entrega do produto.
An Autonomic Approach to Extend the Business Value of a Legacy Order Fulfillment System	Mulcahy e Huang (2015)	Um sistema integrado de gestão empresarial (em inglês, *Enterprise Resource Planning* - ERP).	Descrever um projeto de desenvolvimento no qual um sistema legado foi estendido com um *design* de *loop* de controle autônomo e uma arquitetura orientada a serviços para se comunicar com um provedor externo de triagem para reduzir o custo do trabalho humano necessário para rastrear volumes de pedidos e reduzir o potencial de erro humano no processo de rastreamento.
A Methodology Towards the Adaptization of LegacySystems using Agent-oriented Software Engineering	Wavresky e Lee (2016)	Um *software* de simulação de estações de abastecimento para veículos elétricos.	Apresentar uma abordagem, nomeada pelos autores de "adaptização", que consiste em converter o código legado em um sistema multiagente, para transformar gradativamente um sistema legado em um SA.
Adopting Autonomic Computing Capabilities in Existing Large-Scale Systems	Li et al. (2018)	Um sistema empresarial de larga escala do qual os autores não podem dar detalhes exatos.	Compartilhar a experiência de reengenharia de um sistema industrial de larga escala com a introdução de recursos de computação autônoma para reduzir a intervenção humana no ajuste de configuração de desempenho e melhorá-lo significativamente.

ela introduz atividades de engenharia de *software* a fim de possibilitar a introdução de adaptações com garantias.

O Trabalho de Amoui (2009)

Em seu trabalho, Amoui (2009) apresentou um processo de evolução para ajudar na adição de recursos da computação autônoma a um *software* existente com foco no custo-benefício. Denotado como um processo de co-evolução, tendo em vista que tanto a evolução do "Motor de Adaptação" (em inglês, *Adaptation Engine*) quanto do "*Software* Adaptável" (em inglês, *Adaptable Software*) são abordados (embora o trabalho esteja focado na evolução do "*Software* Adaptável"), este abrange a investigação das especificações do *software* e limites de domínio (desenvolvendo e analisando um modelo de objetivo de adaptação), a reconstrução de domínio da aplicação e a trans-

formação e o refinamento dos comportamentos adaptativos. Além disso, foi proposta uma métrica para indicar a efetividade do processo de evolução que se resume a um indicador de adaptabilidade em relação às especificações de adaptação fornecidas. Dois estudos de caso utilizando *softwares* já existentes, sendo uma aplicação de telefonia de código aberto orientada a serviços e um sistema privado de controle de tráfego aéreo, foram conduzidos com o objetivo de entender o processo de evolução de adaptabilidade. O autor concluiu indicando que o desenvolvimento de conceitos, metodologias e ferramentas para apoiar o processo de evolução da adaptabilidade, bem como sua validação em cenários do mundo real, são contribuições esperadas com a publicação de seu trabalho.

Os Trabalhos de Mulcahy e Huang (2014, 2015)

Mulcahy e Huang descreveram em seus trabalhos (Mulcahy e Huang, 2014, 2015) como utilizaram a arquitetura orientada a serviços para estender as funcionalidades de sistemas legados e adicionar recursos da computação autônoma. No trabalho de 2014, foi apresentada uma solução de integração entre um ambiente de sistemas legados de uma empresa varejista e o *Marketplace* da Amazon, visando minimizar a quantidade de pessoas envolvidas no processo entre a compra *online* e a preparação para a entrega do produto. A API do serviço *web* de *Marketplace* da Amazon (Amazon *Marketplace Web Service*, ou simplesmente Amazon MWS) e uma API já existente do sistema legado empresarial foram utilizadas para efetivar a integração, que incorporou o auto-monitoramento para detectar condições anômalas durante o processo e notificar os responsáveis para indicar a necessidade de intervenções manuais, a auto-configuração para ajustar a frequência na qual os arquivos de pedidos eram transmitidos entre os componentes do ambiente de execução, e auto-cura, no sentido passivo, para ser tolerante às falhas inesperadas. Transmissões de pedidos com dados de teste e variações no ambiente foram simuladas para avaliar o comportamento da solução, que foi considerada bem sucedida após ser posta e avaliada em produção. No trabalho publicado no ano seguinte, Mulcahy e Huang apresentaram uma solução auto-gerenciada para selecionar compradores e destinatários de uma linha de produtos sensível de uma empresa varejista, excluindo aqueles que não eram autorizados a fazer determinadas compras. A solução dispõe de um gerenciador autônomo que periodicamente é executado para interagir com os bancos de dados do sistema legado a fim de rastrear pedidos e, através de serviços *web* de terceiros, realizar a triagem destes para, posteriormente, classificá-los quanto à tentativa de compra inapropriada. No contexto destes trabalhos, os resultados apresentados em ambos apontam para redução do esforço humano e, consequentemente, a redução de custos provenientes de erros humanos nas atividades operacionais.

O Trabalho de Wavresky e Lee (2016)

A proposta de Wavresky e Lee (2016) teve por padrão primeiro converter o código legado em um sistema multiagente com o intuito de facilitar a comunicação com atores externos e, em seguida, gradualmente transformar o sistema legado em um SA. Para esse processo, foi apresentada uma metodologia para estabelecer os passos em direção ao que os autores chamam de "adaptização" e um *framework* de implementação. Além disso, um estudo de caso foi conduzido com o objetivo de "adaptizar" um *software* de simulação de uma estação de carregamento de energia elétrica para veículos. Para isso, três experimentos foram realizados para demonstrar como a "adaptização" foi gradualmente implementada no sistema alvo através da cooperação auto-organizada entre os agentes. Os autores concluem que a "agentificação" é adequada para salvar o investimento legado por reutilizar a lógica legada relevante e transformá-la gradualmente em um SA.

O Trabalho de Li et al. (2018)

Mais recentemente, Li et al. (2018) compartilharam os desafios encontrados e as experiências vividas ao adicionar recursos da computação autônoma em um sistema industrial de larga escala. Os autores descreveram as lições aprendidas com a implementação da solução que objetivou aprimorar o desempenho do sistema e que tem vieses de automonitoramento, onde as métricas de desempenho são coletadas e computadas, auto-otimização, em que buscas por parâmetros de uma configuração ideal são realizadas, e autoconfiguração, na qual os valores de parâmetros do sistema são ajustados. Além da utilização de orientação a aspectos para separar os interesses e minimizar os impactos da incorporação dos comportamentos adaptativos à base de código, o processo de trabalho realizado utilizou uma abordagem que visa a minimizar a quantidade de mudanças no código do sistema não adaptativo através do entendimento prévio do comportamento do sistema.

Ademais, foram usadas estratégias de teste para validar os requisitos, onde os autores indicam que foram necessárias considerar outras perspectivas além das tradicionais de teste funcional e teste de carga, tais como abordagens de teste da eficácia de adaptação, nas quais foram comparados os desempenhos do sistema com e sem a capacidade de adaptação (sob diferentes condições de ambiente e carga do sistema), e de teste de robustez, no qual a capacidade de adaptação foi submetida a cargas extremas com o intuito de evidenciar os impactos de uma possível falha no processo de adaptação.

Os resultados apresentados indicam que a ativação dos recursos de computação autônoma alcança um desempenho muito melhor do que o uso da configuração padrão do sistema e que não há uma configuração ideal universal (indicando o favorecimento

da incorporação dos recursos da computação autônoma para encontrar a configuração ideal de acordo com a carga do sistema e as condições do ambiente de operação).

2.7.1 Notas Sobre os Trabalhos Relacionados sob a Perspectiva do Presente Trabalho

Ainda que os trabalhos relacionados a este trabalho de mestrado apresentem alguma abordagem relativa à incorporação de comportamentos adaptativos a um sistema legado em um algum nível de adaptação, apenas nos trabalhos de Zhang e Cheng (2007), Mulcahy e Huang (2014) e Li et al. (2018) foi mencionada a aplicação de alguma atividade de verificação com o intuito de garantir a conformidade com os requisitos especificados. Ainda assim, somente no trabalho de Li et al. (2018) as abordagens de teste são descritas com enfoque.

Com relação ao sistema alvo utilizado, embora os estudos de Amoui (2009), Mulcahy e Huang (2014, 2015) e Li et al. (2018) tenham utilizado *softwares* do mundo real como objeto de estudo, em nenhum destes ficou evidente a utilização exclusiva de uma aplicação *web* similar a um sistema de informação.

No que diz respeito às similaridades de implementação, assim como nos trabalhos de Zhang e Cheng (2007) e Li et al. (2018), neste trabalho de mestrado uma abordagem que envolve conceitos e mecanismos de orientação a aspectos[3] é utilizada para alcançar a separação de interesses entre a lógica de adaptação e a lógica de negócio e para minimizar os impactos da incorporação dos comportamentos adaptativos à base de código existente.

Assim sendo, este trabalho se diferencia dos demais ao utilizar um sistema de informação enquadrado na categoria Transacional de aplicações *web* e, principalmente, por evidenciar os impactos para o conjunto de casos de teste de um sistema legado com a introdução de comportamentos adaptativos às funcionalidades existentes utilizando métricas de cobertura de código.

2.8 Considerações Finais

Neste capítulo foram apresentados os conceitos básicos sobre *teste de software*, bem como as técnicas, critérios, processo e níveis referentes à atividade de teste, e a estratégia de *teste de regressão*, tal qual algumas de suas abordagens. Sobre aplicações *web*, foram apresentados os seus conceitos básicos, assim como suas características, um

[3]Em específico, uma das implementações apresentadas neste trabalho de mestrado utiliza um recurso da API da Java EE que intercepta o código da lógica de negócio em execução para executar o código da lógica de adaptação.

tipo de categorização por funcionalidades, e *algumas questões relacionadas ao teste* para esse tipo de aplicação. *Sistemas adaptativos* também foram abordados neste capítulo, com a apresentação de suas características fundamentais e um modelo conceitual de estruturação desses tipos de sistema. Ademais, algumas publicações encontradas na literatura que se relacionam a este trabalho foram sumarizadas, destacando-se os seus aspectos de similaridade e os seus diferenciais.

O conhecimento relativo aos conceitos básicos é fundamental para a análise e compreensão da execução deste trabalho, que será apresentada no Capitulo 4. Antes, no Capítulo 3, serão apresentados o planejamento, o processo e os materiais utilizados na execução do trabalho.

Capítulo 3

PLANEJAMENTO DO ESTUDO

3.1 Considerações Iniciais

Este trabalho se trata de um estudo de caso. Estudos de caso são conduzidos com o objetivo de acrescer conhecimento e induzir reflexões sob o fenômeno em estudo. Em pesquisas de engenharia de *software*, o objetivo é entender melhor como e por quê a engenharia de *software* deve ser exercida para aperfeiçoar seu processo e seus resultados (Runeson et al., 2012).

Os passos de execução do estudo de caso deste trabalho foram previamente planejados e sintetizados em um fluxo de processo. A análise dos dados quantitativos foi elaborada com base em métricas de cobertura de código produzida pelos testes e coletadas de cada versão do sistema alvo, tanto a legada quanto as versões que incorporaram algum comportamento adaptativo.

Assim sendo, na Seção 3.2 apresenta-se o referencial metodológico; em particular, define-se o conceito de *estudo de caso* e a justificativa pela qual este trabalho enquadra-se neste método de pesquisa. Nesta seção também é apresentado o objetivo do estudo, bem como as questões de pesquisa, as variáveis do estudo, o planejamento do estudo de caso, incluindo o processo empregado, e os possíveis cenários de implementação. Em seguida, o sistema alvo do estudo é apresentado (Seção 3.3), onde são descritas as tecnologias usadas para o seu desenvolvimento e teste, sua arquitetura, suas dimensões e como os códigos estão organizados. O processo de desenvolvimento das implementações para este trabalho de mestrado é descrito na Seção 3.4. Um breve relato sobre a escolha de uma ferramenta para apoiar o desenvolvimento de algumas implementações é retratado na Seção 3.5, enquanto que na Seção 3.5.1 são apresentados detalhes da ferramenta escolhida. A Seção 3.6 apresenta a biblioteca utilizada para auxiliar a coleta e análise de cobertura de código enquanto que os procedimentos para coleta, organização e análise dos dados são apresentados na Seção 3.7. Ao final do capítulo são apresentadas as considerações finais.

Para os fins do estudo de caso, o uso do Sistema Integrado de Gestão Acadêmica (SIGA) – sistema alvo – foi solicitado à Pró-Reitoria de Gradução (ProGrad) e à Secretaria Geral de Informática (SIn), ambas unidades organizacionais da UFSCar, e autorizado por seus respectivos dirigentes à época.

3.2 Referencial Metodológico

Este trabalho se trata de um estudo de caso exploratório de introdução de comportamentos adaptativos no SIGA, uma aplicação *web* legada do "mundo real" (isto é, um *software* não experimental).

Na literatura, há várias definições para o termo "estudo de caso" como método formal de pesquisa. A título de exemplo, Runeson et al. (2012) constituíram uma definição orientada à área de engenharia de *software* para o termo, derivando de definições mais genéricas, que pode ser traduzida da seguinte forma:

> O estudo de caso em engenharia de *software* é uma investigação empírica que baseia-se em múltiplas fontes de evidência para investigar uma instância (ou um número pequeno de instâncias) de um fenômeno contemporâneo de engenharia de *software* dentro do seu contexto de vida real, especialmente quando a fronteira entre fenômeno e contexto não pode ser claramente especificada.

Esta definição equipara-se com uma definição mais abrangente e recente, cunhada por Yin (2018), que pode ser traduzida da seguinte maneira:

> Um estudo de caso é um método empírico que investiga um fenômeno contemporâneo (o "caso") em profundidade e dentro do seu contexto de mundo real, especialmente quando as fronteiras entre fenômeno e contexto podem não ser claramente evidentes.

Estudos de caso têm um alto grau de realismo por serem conduzidos em cenários do mundo real, nos quais a situação realista é constantemente complexa e não determinística (Runeson et al., 2012).

Este trabalho alinha-se às definições de estudo de caso apresentadas, pois se refere a uma investigação empírica de uma instância de fenômeno contemporâneo em um contexto real da engenharia de *software*. Sua execução foi planejada e realizada seguindo um processo sistêmico, no qual alguns cenários de implementação foram cogitados e, em seguida, selecionados e desenvolvidos. A análise do estudo baseia-se na comparação das métricas de cobertura de código entre a versão legada e as versões

evolutivas do SIGA que implementam alguma característica adaptativa, onde os dados quantitativos para a análise foram coletados com o apoio de uma ferramenta.

3.2.1 Objetivo do Estudo

Este estudo de caso tem por objetivo geral avaliar o impacto sobre a cobertura de código produzida por testes causado pela introdução de comportamentos adaptativos em uma aplicação *web* legada do "mundo real". Mais especificamente, o propósito deste trabalho é explorar empiricamente os resultados de cobertura de código entre a versão legada e as versões evolutivas que introduzem comportamentos adaptativos e compará-las com o intuito de delinear respostas para as questões de pesquisa (QP) definidas a seguir.

3.2.2 Questões de Pesquisa do Estudo

As seguintes questões de pesquisa conduziram o desenvolvimento do estudo de caso:

- **QP1**: Qual é o impacto sobre a cobertura de código das classes que já existiam em relação a funcionalidade que recebeu a introdução de um comportamento adaptativo?
- **QP2**: Qual é a cobertura de código das novas classes que foram introduzidas com a implementação do comportamento adaptativo?

3.2.3 Variáveis Dependentes e Independentes

Em uma pesquisa observacional como um estudo de caso, uma variável é um conceito que pode variar ou ter mais de um valor. Neste trabalho, o interesse reside em saber como determinadas variáveis estão relacionadas. Estas variáveis podem ser dependentes ou independentes. Uma variável independente representa a grandeza que pode ser manipulada ou apenas medida. Uma variável dependente representa a grandeza cujo valor depende, pode ser explicada ou estimada, pela variável independente. Durante a pesquisa, uma variável dependente nunca tem seus valores alterados pelos pesquisadores (Cruzes, 2007).

As variáveis independentes para este trabalho são os casos de teste e as classes Java. As variáveis dependentes, ou seja, aquelas a serem observadas e analisadas, são as métricas de cobertura de código.

3.2.4 Visão Geral

Para a execução do estudo de caso, um fluxo de processo foi definido para sistematizar os procedimentos que seriam realizados. Esse fluxo é representado na Figura 3.1.

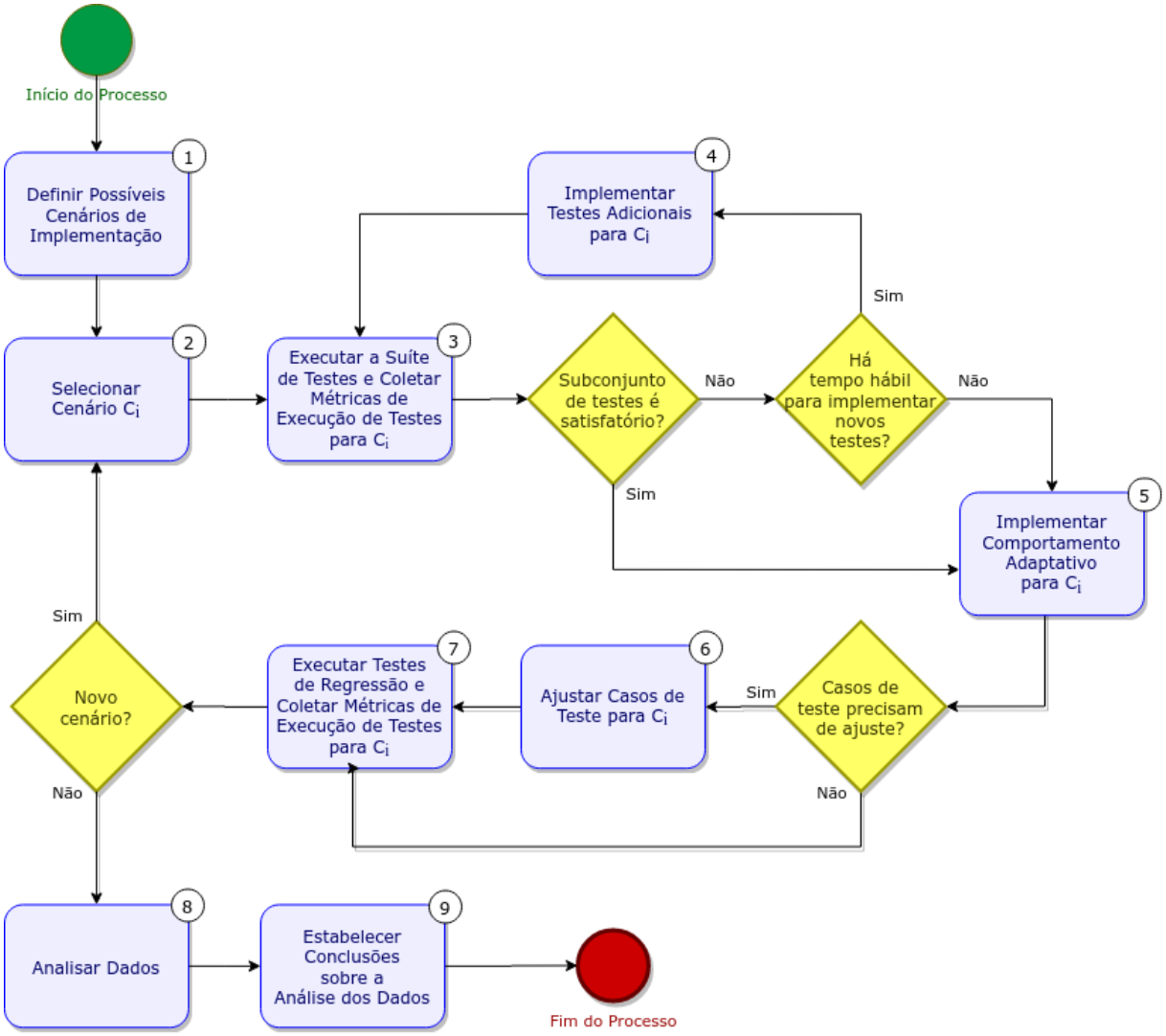

Figura 3.1: Fluxograma do processo do estudo de caso

A primeira etapa deste processo, representada pelo passo 1 da Figura 3.1, foi definir os possíveis cenários de implementação de comportamentos adaptativos. As possibilidades de implementação foram elaboradas de acordo com: (i) o conhecimento prévio do autor sobre as funcionalidades originais do SIGA; (ii) reuniões de *brainstorming* e discussões com membros de grupos de pesquisa do Departamento de Computação[1] (DC) da UFSCar, entre eles, o grupo de pesquisa do Laboratório de Pesquisa em Engenharia de Software[2] (LaPES) e grupo de pesquisa do laboratório *Advanced Research Group on Software Engineering*[3] (AdvanSE); e (iii) pesquisas em base de dados

[1] `https://www.dc.ufscar.br` – acessado em agosto, 2020
[2] `http://lapes.dc.ufscar.br` – acessado em agosto, 2020
[3] `http://advanse.dc.ufscar.br` – acessado em agosto, 2020

científicas com consecutivas leituras de publicações relativas à implementações de SAs, dentre as quais podem ser citados os trabalhos de Cheng et al. (2009), Carzaniga et al. (2010), Castañeda et al. (2014), Weyns e Calinescu (2015), Gerasimou et al. (2017) e Moreno et al. (2018). Neste contexto, quatro possíveis cenários de implementação foram elaborados. A Tabela 3.1 apresenta esses cenários que representaram a conclusão do passo 1 da Figura 3.1. Nesta tabela estão descritos o comportamento inicial, que descreve o modo como a funcionalidade opera originalmente, e o comportamento adaptativo, onde é especificada a forma desejada de operação da funcionalidade, bem como o título concedido ao cenário de implementação definido no passo 1.

Após o término da primeira etapa, o processo de trabalho seguiu para a fase de seleção de um dos possíveis cenários de implementação, descrita pelo passo 2 da Figura 3.1. A ordem de seleção dos cenários foi estabelecida conforme a maturidade dos requisitos e a familiaridade do autor com as funcionalidades consideradas. Uma vez que um cenário foi selecionado, a suíte de teste já existente foi executada por completo e as métricas de cobertura de código produzidas pelos testes foram coletadas (etapa representada pelo passo 3 da Figura 3.1). As métricas de cobertura de código foram utilizadas para observar se o subconjunto de casos de teste, pertinente à funcionalidade, possuía um nível satisfatório antes que fosse implementado algum comportamento adaptativo. Para tanto, esperava-se que pelo menos metade das instruções e dos desvios das classes mais essenciais da funcionalidade fossem cobertas para que o nível de satisfação fosse atingido.

Nas ocasiões em que o subconjunto de casos de teste não era satisfatório e havia um tempo considerável diante do cronograma de trabalho do projeto, o processo seguiu então para a etapa de implementação de casos de teste adicionais (passo 4 da Figura 3.1). Nessa etapa, novos casos de teste foram planejados e implementados com a finalidade de aumentar a cobertura de código da funcionalidade para o cenário selecionado e, por conseguinte, atender ao critério de satisfação. Além disso, a implementação de casos de teste em nível de sistema foi priorizada em relação aos outros níveis. Salienta-se novamente que os casos de teste implementados nessa etapa foram desenvolvidos no projeto original do SIGA, ou seja, aquele mantido pela equipe de desenvolvimento da SIn/UFSCar, e foram incorporados definitivamente ao conjunto de testes do projeto original.

Tendo um subconjunto de casos de teste satisfatórios, a suíte de testes foi executada novamente com o propósito de coletar métricas de cobertura de código da implementação da funcionalidade sem a introdução de algum comportamento adaptativo (etapa representada pelo passo 3 da Figura 3.1), visando compará-las com as métricas da implementação da funcionalidade com comportamento adaptativo. Em seguida, o processo seguiu para a implementação do comportamento adaptativo (passo 5 da Figura 3.1). Nesta etapa, a funcionalidade recebeu a introdução de novos trechos de

Tabela 3.1: Possíveis cenários de implementação definidos para o estudo de caso

Cenário	Comportamento Inicial	Comportamento Adaptativo
Autenticação Autocurável	Sistema valida credenciais do usuário através de um único mecanismo de autenticação externo.	Sistema valida credencias do usuário através de um mecanismo de autenticação externo, denominado principal, sendo capaz de trocar por outro(s) mecanismo(s) de autenticação, denominado(s) secundário(s), nas ocasiões em que identificar a indisponibilidade do mecanismo principal.
Processamento Auto-otimizado de Candidatos à Formatura e Formandos	Sistema processa as matrículas de estudantes de forma linear para verificar a necessidade de alteração de seus *status* em relação ao cumprimento de sua matriz curricular.	Sistema processa as matrículas de estudantes de forma concorrente ou paralela para verificar a necessidade de alteração de seus *status* em relação ao cumprimento de sua matriz curricular de acordo com o tamanho do lote de matrículas a serem processadas, a disponibilidade dos recursos computacionais (principalmente, memória) e o tempo despendido na execução atual e na execuções anteriores.
Autenticação e Autorização Autoprotetora	Sistema verifica se a requisição HTTP para uma determinada URL está autorizada com base nas credenciais armazenadas na sessão do usuário, caso o usuário esteja autenticado. Quando o usuário não está autenticado, todas as requisições às páginas que precisem de autorização são redirecionadas para a página de autenticação.	Sistema verifica o IP de origem da requisição HTTP e o usuário que está sendo utilizado nesta requisição, bem como a frequência de requisições provenientes desse IP ou desse usuário (que pode estar sendo utilizado para acessar o sistema de diversos dispositivos). Com base em uma política de segurança, o sistema bloqueia preventivamente e temporariamente a autenticação do usuário para esse determinado IP e notifica o usuário através de mensagem de e-mail.
Personalização Autoconfigurável	Sistema apresenta para o usuário autenticado uma lista de funcionalidades que ele tem acesso configurado por meio do gerenciamento de privilégios.	Sistema apresenta para o usuário uma lista de funcionalidades que ele tem acesso, com base nos seus privilégios, ordenada de acordo com os seguintes critérios: (i) intervalos de datas do período letivo vigente que se encaixam à data atual (por exemplo, a funcionalidade de digitação de notas pode ser priorizada somente nos períodos em que o calendário acadêmico prever sua utilização); e (ii) número de vezes que o usuário acessou uma determinada funcionalidade (semelhante à abordagem de funcionalidade favorita).

código que agregaram características de comportamentos adaptativos. Mesmo com a alteração de código, a introdução dessas características objetivou não descaracterizar a usabilidade e navegabilidade já existentes. Ou seja, as alterações realizadas com a introdução do comportamento adaptativo prezaram pela manutenção do modo de

operar o SIGA. Assim, as alterações focaram no lado servidor do SIGA com o propósito de evitar modificações nos casos de teste aplicados a elementos de interface.

Ao final da implementação da introdução do comportamento adaptativo, foi verificado se a suíte de teste precisava de ajuste para ser novamente executada por completo. Quando necessário, os ajustes nos casos de testes foram realizados (passo 6 da Figura 3.1), como a substituição do objeto dublê para alguns casos de teste em nível de unidade, por exemplo, em função de uma modificação na classe sob teste. A etapa seguinte do processo (passo 7 da Figura 3.1) foi executar a suíte de teste por completo a fim de coletar as métricas de cobertura de código da funcionalidade com o comportamento adaptativo para compará-las com as métricas coletadas anteriormente (passo 3 da Figura 3.1).

Neste ponto do fluxo de processo, pôde-se optar por reiniciar o ciclo a partir da seleção de um novo cenário de implementação (passo 2 da Figura 3.1), considerando o tempo disponível pelo cronograma de trabalho. No momento em que não se optou por implementar um novo cenário, o processo seguiu para a etapa de análise dos dados (passo 8 da Figura 3.1), onde os dados reunidos na etapas de coleta de métricas (passos 3 e 7) foram examinados e comparados entre as versões da mesma funcionalidade. Em outras palavras, os dados coletados da versão não adaptativa foram comparados aos dados coletados das versões com comportamentos adaptativos introduzidos. Na etapa final do processo (passo 9 da Figura 3.1), foram estabelecidas as conclusões com base nas análise realizadas sob os dados, concluindo o processo em seguida.

3.3 Aplicação *Web* Alvo (SIGA versão Legada)

O objeto utilizado no estudo de caso é uma aplicação *web* legada do “mundo real” que está em operação desde de 2015. O Sistema Integrado de Gestão Acadêmica (SIGA) é o *software* que automatiza processos e procedimentos de controle e gestão da graduação da UFSCar.

O principal motivo para a escolha do SIGA como alvo para o estudo de caso deste trabalho é devido a este ser uma aplicação *web* legada. Ou seja, é uma aplicação que foi desenvolvida há algum tempo, de acordo com práticas e tecnologias de desenvolvimento datadas, e que ainda permanece em operação (Warren, 1999). Por conseguinte, o SIGA é um *software* do “mundo real”, isto é, não experimental. Além disso, o fato do autor deste trabalho atuar no desenvolvimento corretivo e evolutivo desta aplicação *web* inibiu a etapa de compreensão de detalhes de projeto, tais como a arquitetura,

compilação, execução de testes, funcionalidades e, principalmente, comportamentos, que não é o foco deste estudo.

A concepção do projeto iniciou-se em meados de 2013 sendo uma reengenharia da aplicação *web* legada anterior de controle acadêmico da graduação da UFSCar. O SIGA foi concebido sobre a plataforma Java com a intenção utilizar tecnologias modernas da época. Desta maneira, ele foi desenvolvido de acordo as especificações da versão 6 da *Java Enterprise Edition* (Java EE)[4], utilizando tecnologias como *JavaServer Faces* (JSF), *Java Persistence API* (JPA), *Enterprise JavaBeans* (EJB) e *Context and Dependency Injection* (CDI).

Em relação à arquitetura, o SIGA está decomposto nas seguintes camadas:

- Apresentação: composta por páginas *web* criadas utilizando JSF, arquivos *Cascading Style Sheets* (CSS) para estilização das páginas *web*, e arquivos Javascript;
- Controle: composta por classes de controle de fluxo e filtros de requisição HTTP, conversão de dados entre páginas *web* e lógica de controle, entre outros componentes *web* das especificações Java EE;
- Serviço: composta por classes que implementam regras de negócios, classes para o controle de exceções, entre outras; e
- Acesso a Dados: classes de entidade que mapeiam as tabelas da base de dados e classes que acessam o SGBD para manipular dados.

A arquitetura do SIGA influencia a organização dos códigos fonte do projeto, que é dividido em módulos utilizando o Apache Maven[5]. O Maven é uma ferramenta utilizada para, entre outras tarefas, automatizar a compilação dos códigos fonte, o empacotamento dos artefatos e a execução de testes. Os principais módulos do projeto SIGA são: (i) `siga-web`; e (ii) `siga-ejb`. O primeiro módulo comporta as camadas de apresentação e de controle, enquanto que o segundo engloba as camadas de serviço e de acesso a dados. A Figura 3.2 apresenta de forma simplificada um esboço da arquitetura do SIGA.

Além dos módulos supracitados, o projeto SIGA é composto pelos seguintes módulos Maven:

- `siga-entity`: reúne essencialmente as classes de entidade que mapeiam as tabelas da base de dados;
- `siga-ear`: responsável por empacotar os artefatos gerados pelos módulos `siga-web` e `siga-ejb` (arquivos no formato WAR – *Web application ARchive*, e

[4] `https://www.oracle.com/technetwork/java/javaee/tech/javaee6technologies-1955512.html` – acessado em agosto, 2020

[5] `https://maven.apache.org` – acessado em agosto, 2020.

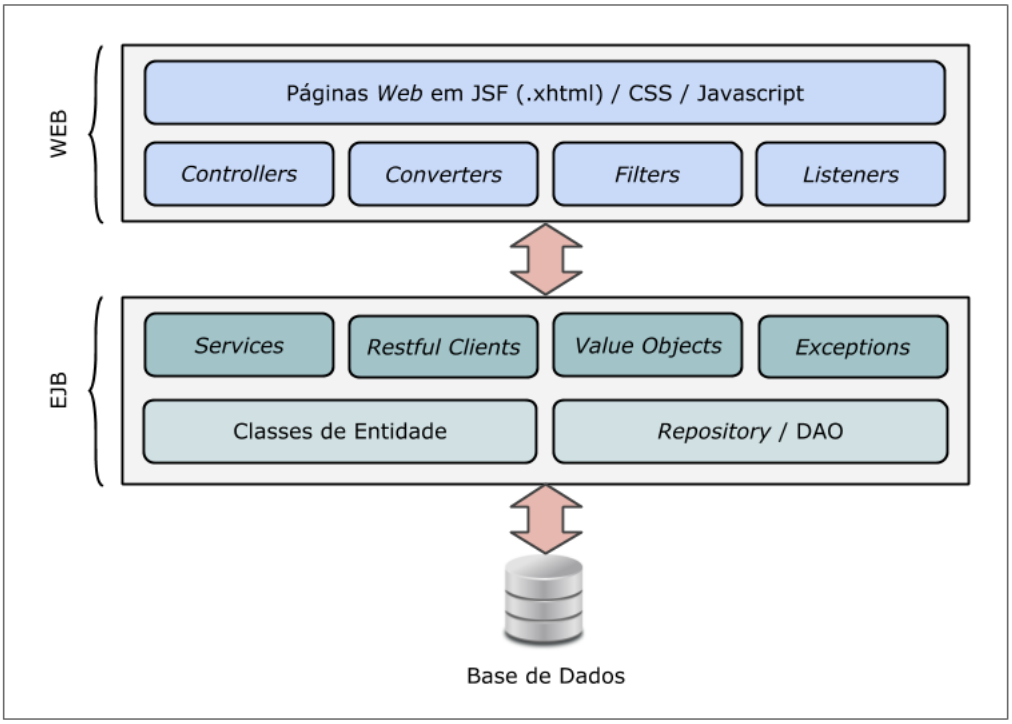

Figura 3.2: Arquitetura simplificada do SIGA

JAR – *Java ARchive*, respectivamente) em um arquivo único no formato EAR (*Enterprise Application aRchive*) com o objetivo de tornar simultânea a implantação destes módulos em servidores de aplicação *web*;

- `siga-test-funcional`: engloba tanto as classes de teste em nível de sistema quanto classes que implementam o padrão Page Object (Fowler, 2013), por exemplo;
- `springdbunit-utils`: formado basicamente por classes utilitárias a serem utilizadas na implementação e execução de testes dos módulos `siga-ejb` e `siga-test-funcional`; e
- `siga-jacoco`: responsável por agregar os arquivos com dados de execução dos testes dos módulos `siga-web`, `siga-ejb` e `siga-entity` e gerar um relatório unificado com as métricas de cobertura código produzida por testes.

Quando em execução, o SIGA também faz uso de serviços *web* de outro sistema mantido pela UFSCar, o Sistema de Apoio à Gestão Universitária Integrada (SAGUI)[6]. Este serviço *web* permite ao SIGA consumir e produzir dados comuns à UFSCar, uma vez que estes dados são integrados e também são utilizados por outros *softwares*, além de realizar a autenticação de usuários.

[6] `https://sistemas.ufscar.br/sagui` – acessado em agosto, 2020

Quanto às tecnologias empregadas pelo SIGA para a execução de testes, o *framework* JUnit[7] é utilizado para executar os testes em nível de unidade. O *framework* Mockito[8] também é aplicado nos testes de unidade para criar objetos dublês no intuito de simular os comportamentos das dependências e isolar a unidade a ser testada. Para os testes em nível de sistema, é utilizado o Arquillian[9], uma plataforma que permite que os testes no formato JUnit sejam executados em servidores de aplicação *web*. Ademais, o Graphene[10], uma extensão do projeto Arquillian que provê um conjunto de extensões para o Selenium Webdriver[11], é usado para apoiar a execução dos testes aplicados a elementos de interface. Ainda para os testes em nível de sistema, o Spring Test DBUnit[12] é utilizado para configurar os dados destinados aos cenários de teste no SGBD.

No que se refere ao porte do projeto SIGA, algumas métricas foram extraídas utilizando-se o *plugin* MetricsReloaded[13] para o ambiente de desenvolvimento integrado IntelliJ IDEA[14]. O projeto possui em torno de 1.400 classes de produto, isto é, classes que não são classes de teste como, por exemplo, classes JUnit. Estas classes contabilizam cerca de 143.000 linhas de código (LOC) não comentadas, onde aproximadamente 56.000 linhas são efetivamente de códigos de instrução. Isto quer dizer que não são contabilizadas, por exemplo, linhas com instrução de importação de outras classes e linhas em branco. Com relação à suíte de testes, 395 casos de teste[15] estão distribuídos entre os módulos do projeto, conforme exposto pela Tabela 3.2.

Tabela 3.2: Quantidade de casos de teste por módulo do SIGA

Módulo	Qtde. de Casos de Teste
`siga-ejb`	365
`siga-entity`	23
`siga-web`	6
`siga-test-funcional`	1
Total	395

Ainda em relação às dimensões, o SIGA chega a ter em torno de 10.000 visitantes únicos em um dia em períodos de pico de uso, conforme informações extraídas da

[7] `https://junit.org` – acessado em agosto, 2020
[8] `https://site.mockito.org` – acessado em agosto, 2020
[9] `http://arquillian.org` – acessado em agosto, 2020
[10] `http://arquillian.org/arquillian-graphene` – acessado em agosto, 2020
[11] `https://www.selenium.dev/projects/#selenium-webdriver` – acessado em agosto, 2020
[12] `https://springtestdbunit.github.io/spring-test-dbunit` – acessado em agosto, 2020
[13] `https://github.com/BasLeijdekkers/MetricsReloaded` – acessado em agosto, 2020
[14] `https://www.jetbrains.com/idea` – acessado em agosto, 2020
[15] Nesta contagem estão excluídos os casos de teste criados para os objetivos do desenvolvimento da pesquisa de mestrado.

plataforma analítica de acessos utilizada pela Secretaria Geral de Informática (SIn) para monitorar as aplicações *web* da UFSCar.

3.4 Processo de Desenvolvimento

O processo de preparação para as implementar os cenários definidos para o estudo de caso deu-se através da bifurcação (em inglês, *forking*) da base de código da versão 1.13.0 do SIGA, datada do início de julho de 2018 e mantida sob o sistema de controle de versão Git[16]. A bifurcação é processo de cópia de um repositório de código que possibilita que um projeto já existente seja utilizado como ponto de partida para outras experiências em um novo projeto. Essa abordagem permite que alterações sejam feitas no novo projeto sem comprometer o projeto original. Além disso, as alterações realizadas no projeto original podem ir sendo incorporadas ao novo projeto mesmo após a realização do processo de bifurcação.

Assim, o processo de desenvolvimento foi conduzido de modo que a base de código do sistema alvo para o estudo de caso foi regularmente atualizada com alterações efetuadas no projeto original do SIGA. Ou seja, os códigos submetidos pela equipe de desenvolvimento do projeto original do SIGA foram incorporados ao projeto de pesquisa sempre que estes foram disponibilizados de forma estável. Porém, salienta-se que os códigos desenvolvidos para a versão do projeto de pesquisa não foram integrados aos códigos do projeto original do SIGA. Essa abordagem foi estabelecida dado que os casos de teste que eventualmente precisaram ser criados para exercitar os elementos de *software* das funcionalidades que receberiam a introdução de comportamentos adaptativos foram desenvolvidos no projeto original e, só então, incorporados ao projeto de pesquisa.

O processo de incorporação dos códigos do projeto original no projeto de pesquisa prosseguiu até a versão 1.18.5 do SIGA, liberada no final abril de 2020.

No que se refere à organização das implementações, cada versão adaptativa do SIGA foi desenvolvida em um ramo (em inglês, *branch*) exclusivo do sistema de controle de versão. A ramificação é um recurso disponível na maioria dos sistemas de controle de versão mais modernos e permite derivar implementações já existentes. O ramo principal, denominado *develop*, foi mantido como uma cópia da base de códigos do projeto original do SIGA e recebeu as incorporações regularmente.

A organização por ramos permitiu que nas ocasiões em que foi necessário incorporar os códigos do projeto original ou alterar os códigos de uma versão adaptativa, foi possível reaplicar as alterações

[16] `https://git-scm.com` – acessado em agosto, 2020

3.5 Ferramentas de Apoio

Na literatura, podem ser encontradas diversas abordagens para a concepção de SAs (Krupitzer et al., 2015). Pesquisas e práticas nesta área abordam os desafios de projetar esses sistemas a partir de várias perspectivas e níveis de abstração. Uma abordagem amplamente reconhecida é a adaptação baseada na arquitetura (em inglês, *Architecture-based Adaptation*). Essa abordagem envolve mudanças de configuração de sistema em tempo de execução e atualização de comportamentos (Braberman et al., 2017).

Dentre as abordagens avaliadas por Krupitzer et al. (2016) que foram categorizadas com sendo do tipo de adaptação baseada na arquitetura e que ofereciam suporte à implementação através de um *framework*, Rainbow (Garlan et al., 2004) e StarMX (Asadollahi et al., 2009) foram selecionadas para serem avaliadas no uso deste trabalho. Ambas as abordagens foram exploradas e suas implementações via *framework* foram experimentadas. A tentativa de implementar uma aplicação *web* adaptativa simplista, isto é, sob uma arquitetura MVC (*Model-View-Controller*) e sem a adição de qualquer componente externo, como um SGBD, por exemplo, utilizando o Rainbow não foram bem sucedidas. A documentação incompleta à época, a falta de exemplos variados e as diversas configurações da solução, foram um entrave neste processo experimental. Em contrapartida, a documentação íntegra e objetiva, somado aos exemplos de código[17] disponibilizados na plataforma SourceForge, permitiram implementar uma aplicação *web* adaptativa simplista com o apoio do StarMX[18]. Deste modo, o *framework* StarMX foi destacado para condução de implementações deste trabalho.

3.5.1 O *Framework* StarMX

O StarMX é um *framework* que permite a criação de sistemas com características autogerenciáveis (*self-**). Baseado no princípio de separação de interesses, ele provê uma abordagem flexível para criar o processo de adaptação, separando a lógica de gerenciamento do recurso gerenciado. A lógica de gerenciamento é implementada sob um conjunto de entidades, denominadas “Processos” (em inglês, *Process*). Cada processo pode assumir uma ou mais funções de cada um dos quatro componentes básicos do modelo MAPE-K, ou seja, os componentes monitor, analisador, planejador e executor. A composição de processos forma o gerenciador autônomo.

[17] `https://sourceforge.net/projects/starmx` – acessado em agosto, 2020

[18] `https://github.com/rogeriogentil/starmx-managed-webapp` – acessado em agosto, 2020

A automatização das operações é inicializada através dos mecanismos de ativação definidos em um arquivo de configuração do *framework*. Os mecanismos de ativação suportados pelo StarMX são baseados em tempo e evento. Na primeira abordagem, as operações são desencadeadas em intervalos de tempo fixo, enquanto que na segunda, uma entidade dispara uma ação que ativa uma ou mais operações.

Em tempo de execução, o StarMX pode operar sobre sensores e atuadores de duas formas: (i) por meio de um simples objeto Java; ou (ii) através de interfaces de instrumentação da arquitetura *Java Management Extensions* (JMX)[19]. A tecnologia JMX é parte integrante do *Java Development Kit* (JDK) e é comumente utilizada em ambientes Java EE (Asadollahi et al., 2009). Vários recursos podem ser gerenciados através das interfaces de instrumentação JMX, chamadas de MBeans. Um MBean pode representar uma aplicação, um dispositivo ou qualquer outro recurso que precise ser gerenciado. Ademais, a JMX permite o gerenciamento remoto de recursos.

Em relação à definição dos processos, também é possível utilizar duas abordagens: (i) usando uma linguagem de regras de negócios (em inglês, *policy language*); ou (ii) implementando a interface `org.starmx.core.Process`, com comportamentos definidos pelo *framework*, utilizando a linguagem Java. Na primeira abordagem, os requisitos autogerenciáveis são definidos na forma de políticas de condição-ação. O StarMX pode interagir com *softwares* externos do tipo *policy engine* desde que sejam implementados os comportamentos exigidos pela interface `org.starmx.policy.PolicyAdapter`. Contudo, algumas restrições podem ser impostas à lógica de gerenciamento devido a alguma peculiaridade na construção ou sintaxe linguagem utilizada. Por conta disso, também é possível se beneficiar de todos os recursos da linguagem Java para especificar os requisitos de gerenciamento.

3.6 A Biblioteca JaCoCo

A JaCoCo (acrônimo de ***Ja**va **Co**de **Co**verage*)[20] é uma biblioteca que auxilia a análise de cobertura de código produzida por testes de *softwares* que são executados sob a Máquina Virtual Java (em inglês, *Java Virtual Machine* - JVM). Esta biblioteca possui funcionalidades que permitem instrumentar códigos em execução, onde são registradas as instruções realizadas durante a execução de um programa, e gerar relatórios em diversos formatos com métricas de cobertura de múltiplas execuções.

A instrumentação é realizada sob o código compilado (Java *bytecode*) do programa. Salienta-se que uma única linha de código fonte pode gerar diversas instruções no

[19] `https://www.oracle.com/java/technologies/javase/javamanagement.html` – acessado em agosto, 2020

[20] `https://www.jacoco.org/jacoco` – acessado em agosto, 2020

código compilado. A geração de várias instruções de código também ocorre para declarações de código fonte implícitas, tal como na linguagem Java em que não é preciso declarar um construtor vazio quando este é o único construtor de uma classe. A JVM é a responsável por interpretar os códigos compilados, independentemente da plataforma em que eles foram gerados, ou seja, independentemente da linguagem de programação e do sistema operacional.

Em relação aos relatórios, estes são gerados usando-se um conjunto de diferentes contadores para calcular as métricas de cobertura. Os contadores, derivados de informações contidas nos arquivos compilados, utilizados pela JaCoCo são:

- **Instruções**: são as menores unidades contabilizadas pela JaCoCo. As informações de cobertura deste contador fornecem dados sobre quantas e quais instruções de código foram ou não ativadas. A contagem independe da formatação do código.

- **Desvios**: contabiliza os desvios das declarações condicionais (*if* e *switch* da linguagem Java, por exemplo) conforme o valor ou a combinação de valores das condições. As informações de cobertura deste contador fornecem dados sobre quantos desvios existem e quais foram percorridos ou não. Ressalta-se que os tratamentos de exceções não são contabilizados por esse contador.

- **Complexidade Ciclomática**: contabiliza o número mínimo de caminhos que pode gerar todos os possíveis caminhos de um método. Os métodos abstratos não são contabilizados.

- **Linhas**: contabiliza o número de linhas de código fonte. Em virtude de uma única linha do código fonte tipicamente gerar várias instruções de código compilado, a contagem de linhas de métodos não é simplesmente o total de linhas que a classe contém. Assim, considera-se uma linha executada quando ao menos uma instrução atribuída à linha é executada.

- **Métodos**: contabiliza os métodos não abstratos que contenham ao menos uma instrução. Considera-se como um método executado quando ao menos uma de suas instruções foi executada. Construtores e inicializadores estáticos são considerados métodos pela JaCoCo. Salienta-se que instruções implícitas no código fonte, tal como construtores padrão não declarados, são gerados na compilação do código fonte.

- **Classes**: contabiliza a classe compilada (`.class`). Considera-se como uma classe executada quando ao menos um de seus métodos foi executado. Essa métrica tem relevância quando a análise é feita a nível de pacote.

Além disso, a JaCoCo possui integrações com várias ferramentas de construção; entre elas, o Apache Maven, utilizada no projeto SIGA.

3.7 Procedimentos para Coleta, Organização e Análise dos Dados

A coleta e organização dos dados para análise no escopo deste trabalho foi realizada de forma sistemática. Para cada versão do SIGA, tanto para a legada quanto para aquelas que receberam a introdução de algum comportamento adaptativo, foram coletados e separados os arquivos com dados da instrumentação de código realizada pela JaCoCo. Estes arquivos foram gerados em dois momentos:

1. Na compilação de cada módulo Maven do projeto onde existiam testes em nível de unidade (mais especificamente os módulos `siga-entity`, `siga-ejb` e `siga-web`); e

2. Na execução dos casos de teste em nível de sistema.

Agrupados por versão, estes arquivos foram combinados por meio de um processo de fusão utilizando uma ferramenta de linha de comando da JaCoCo para gerar um novo e único arquivo com dados da instrumentação de código. Utilizando a mesma ferramenta, foram gerados os relatórios de cobertura de código no formato HTML para cada versão a partir do novo arquivo gerado.

A tabulação de dados foi realizada em planilha eletrônica, onde as classes envolvidas em cada versão a ser analisada foram destacadas do relatório emitido pela ferramenta de linha de comando da JaCoCo. Além disto, a essa tabulação foram acrescidos alguns valores que foram calculados com base nos dados apresentados pelo relatório. Para as métricas Instruções e Desvios, foi calculado o total de cada métrica somando-se os valores das porções de código inatingida e coberta. Para as métricas Complexidade, Linhas, Métodos e Classes, foram calculados os valores que indicam a porção de código coberta e o percentual de cobertura. Deste modo, todas as métricas foram tabuladas apresentando os valores de porção de código inatingida, coberta e total e o valor de percentual de cobertura.

3.8 Considerações Finais

Ao longo deste capítulo foram apresentados o referencial metodológico utilizado na preparação para a execução do estudo de caso (Seção 3.2), assim como o fluxo de processo (Subseção 3.2.4) e o processo de desenvolvimento (3.4), os materiais (Seções 3.3, 3.5 e 3.6) e os procedimentos utilizados (Seção 3.7) com o propósito de atingir os objetivos definidos e apresentados na Subseção 3.2.1 e responder as questões de pesquisa (Subseção 3.2.2).

A compreensão do tipo de trabalho que foi desenvolvido, tanto quanto da execução do estudo de caso que será apresentada no próximo capítulo, colaboram para a discussão, análise e interpretação dos resultados que serão apresentados mais adiante.

Capítulo 4

EXECUÇÃO DO ESTUDO DE CASO

4.1 Considerações Iniciais

Neste capítulo é apresentada a execução do estudo de caso, com foco na definição dos cenários de implementação e suas realizações. Na Seção 4.2, os cenários selecionados e as versões geradas para cada implementação são expostos. As Subseções 4.2.1 e 4.2.2 elucidam, nesta ordem, como cada funcionalidade selecionada opera originalmente e qual é o comportamento adaptativo que foi implementado. Além disso, as implementações de cada cenário são detalhadas em suas respectivas subseções. Este capítulo encerra-se com as considerações finais onde são resumidos os procedimentos executados no desenvolvimento do estudo de caso.

4.2 Cenários de Implementação

Considerando o fluxo de processo apresentado no capítulo anterior (Seção 3.2.4) e a repetição dos passos a partir da seleção de algum novo cenário de implementação em função do cronograma de trabalho do projeto, as funcionalidades ***Autenticação*** e ***Processamento de Candidatos à Formatura e Formandos*** receberam a introdução de comportamentos adaptativos caracterizados, respectivamente, como **Autocurável** (*Self-Healing*) e **Auto-otimizado** (*Self-Optimized*). Durante a execução do estudo de caso, os cenários de implementação planejados e apresentados na Tabela 3.1 foram subdivididos com a finalidade criar versões minimalistas. Ou seja, mais de uma versão do SIGA com comportamento adaptativo foi gerada para um mesmo cenário. Desta forma, pôde-se coletar as métricas de cobertura e analisá-las de forma mais granular. As implementações também foram divididas em dois ramos de desenvolvimento. Um ramo utilizou apenas as tecnologias já inerentes ao SIGA, em especial as APIs das

especificações Java EE 6. No segundo ramo, o *framework* StarMX, apresentado na Seção 3.5.1, foi utilizado para apoiar as implementações dos comportamentos adaptativos. Assim, as implementações realizadas podem ser destacadas da seguinte forma:

- Autenticação Autocurável
  - com 2 mecanismos de autenticação
    - usando apenas APIs das especificações Java EE 6
    - com o apoio do *framework* StarMX
  - com 3 mecanismos de autenticação
    - usando apenas APIs das especificações Java EE 6
    - com o apoio do *framework* StarMX
- Processamento Auto-otimizado de Candidatos à Formatura e Formandos
  - com 1 elemento de conhecimento
  - com 2 elementos de conhecimento

As versões com comportamentos adaptativos para a funcionalidade "Processar de Candidatos à Formatura e Formandos" com o apoio do *framework* StarMX não puderam ser implementadas devido às limitações funcionais do *framework* em relação a característica mais dinâmica do comportamento adaptativo elaborado. Os detalhes de implementação do Processamento Auto-otimizado de Candidatos à Formatura e Formandos serão apresentados na Seção 4.2.2.

Ainda sobre os ramos de desenvolvimento, as implementações dos comportamentos adaptativos foram realizadas de modo incremental em cada ramo. Isto é, para cada ciclo de implementação, um novo elemento foi introduzido de forma sequencial, gerando uma nova versão do SIGA com comportamento adaptativo. A Figura 4.1 ilustra o resultado do processo de evolução do SIGA.

Evidenciados os cenários de implementação, nas seções subsequentes são descritos como cada funcionalidade opera originalmente, assim como, os respectivos casos de teste, tanto em nível de sistema quanto em nível de unidade, e os detalhes de implementações de cada cenário.

4.2.1 Autenticação

A autenticação em uma aplicação *web* é um mecanismo de verificação que valida a identidade de um usuário de forma a permitir que a comunicação cliente-servidor ocorra somente quando ambos são verificados. (Pressman e Lowe, 2009)

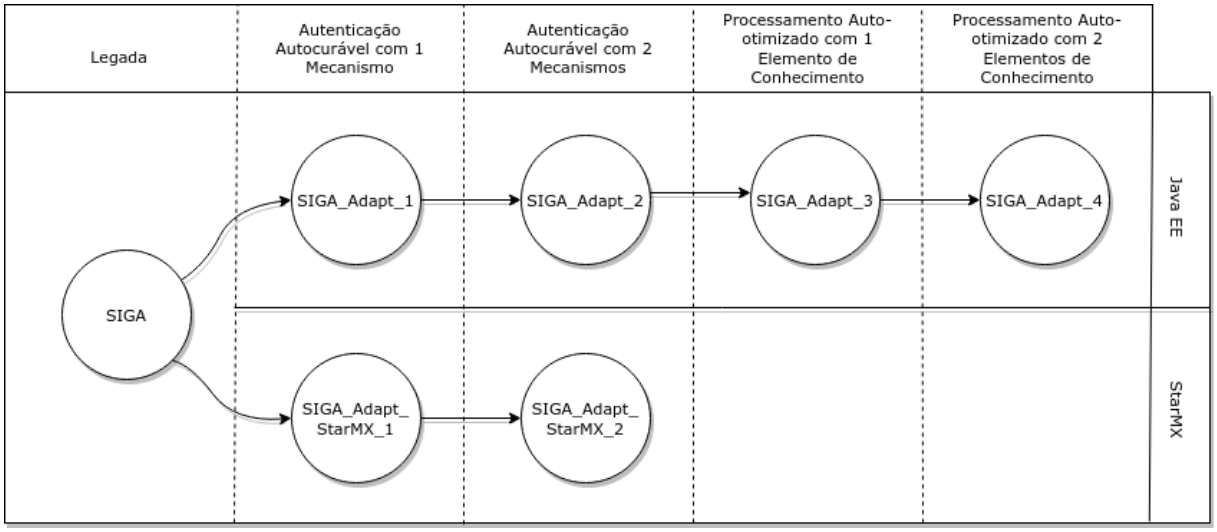

Figura 4.1: Diagrama de evolução do SIGA com a introdução de comportamentos adaptativos

Semelhante a outras aplicações *web*, a maioria das funcionalidades do SIGA exige que o usuário esteja autenticado para executá-las. Para autenticar-se, um usuário deve fornecer suas credenciais, ou seja, seu número UFSCar ou seu número de Cadastro de Pessoa Física (CPF), e sua senha, e submetê-las pelo formulário de autenticação. A Figura 4.2 apresenta o formulário de autenticação disponível na página inicial[1] do SIGA. O processo de submissão das credenciais é iniciado pelo cliente através de uma requisição HTTP. Por conseguinte, o servidor processa a requisição e devolve uma resposta para o cliente. O processamento da requisição no lado servidor utiliza o serviço *web* de autenticação do SAGUI para validar as credenciais.

Figura 4.2: Formulário de autenticação do SIGA

Com relação aos testes, a funcionalidade de autenticação do SIGA possuía apenas um caso de teste implementado. Somente o caso de teste em nível de sistema, no qual as credencias são válidas, estava implementado. Logo, com base nas especificações da funcionalidade e seguindo o critério de teste Particionamento de Equivalência (Myers

[1] `https://sistemas.ufscar.br/siga` – acessado em julho/2020.

et al., 2011), novos casos de teste funcionais em nível de sistema foram elaborados e desenvolvidos antes da implementação de qualquer comportamento adaptativo. A Tabela 4.1 sintetiza o conjunto casos de teste para a funcionalidade de autenticação que foram implementados onde as pré-condições são: (i) o usuário deve ser válido para o sistema; e (ii) o usuário não deve estar autenticado.

Tabela 4.1: Conjunto de casos de teste funcionais em nível de sistema para a funcionalidade de autenticação

ID	Descrição	Resultado Esperado
AUTH-FT-TC1	Preencher o campo Nº UFSCar ou CPF e deixar em branco o campo Senha - ambos campos do formulário de autenticação.	Informar que o campo “Senha” é obrigatório.
AUTH-FT-TC2	Preencher o campo Nº UFSCar ou CPF e deixar em branco o campo Senha - ambos campos do formulário de autenticação.	Informar que o campo “Nº UFSCar ou CPF” é obrigatório.
AUTH-FT-TC3	Deixar em branco os campos Nº UFSCar ou CPF e Senha - ambos campos do formulário de autenticação.	Informar que os campos “Nº UFSCar ou CPF” e “Senha” são obrigatórios.
AUTH-FT-TC4	Preencher o campo Nº UFSCar ou CPF com valor válido e preencher o campo Senha com valor inválido - ambos campos do formulário de autenticação.	Exibir a mensagem “Usuário e/ou senha inválidos.”.
AUTH-FT-TC5	Preencher o campo Nº UFSCar ou CPF com valor inválido e preencher o campo Senha com valor válido - ambos campos do formulário de autenticação.	Exibir a mensagem “Usuário e/ou senha inválidos.”.
AUTH-FT-TC6	Preencher o campo Nº UFSCar ou CPF com valor inválido e preencher o campo Senha com valor inválido - ambos campos do formulário de autenticação.	Exibir a mensagem “Usuário e/ou senha inválidos.”.
AUTH-FT-TC7	Preencher o campo Nº UFSCar ou CPF com valor válido e preencher o campo Senha com valor válido - ambos campos do formulário de autenticação.	Autenticar usuário.

Para exemplificar a implementação dos casos de teste descritos na Tabela 4.1, o fragmento de código 4.1 referente ao caso de teste identificado como “AUTH-FT-TC1” é apresentado a seguir.

```
1  @RunWith(Arquillian.class)
2  ...
3  @DatabaseSetup(value = {"/datasets/02-siga.minimun.dbunit.xml", ...})
4  public class LoginIT {
5      ...
6      private static final String MSG_VALIDACAO_INPUT = "Campo deve ser preenchido.";
7      ...
8      @Test
9      public void deveExibirMensagemDeValidacaoQuandoSenhaNaoForFornecida(
10             @InitialPage LoginPage loginPage) {
11         loginPage.preencherInputUsuario(USUARIO_VALIDO);
12         loginPage.clicarBotaoEntrar();
13         final String inputPasswordMessage = loginPage.getPasswordMessage();
14         assertThat(inputPasswordMessage, is(equalTo(MSG_VALIDACAO_INPUT)));
15     }
16     ...
17 }
```

Listagem de Código 4.1: Implementação do caso de teste intitulado AUTH-FT-TC1

A anotação `@RunWith`, na primeira linha, especifica que a execução dos métodos da classe `LoginIT` é coordenada pelo Arquillian. Na linha 3, a anotação `@DatabaseSetup`, bem como seus valores configurados com arquivos XML, indicam como as tabelas da base de dados devem ser preenchidas antes que qualquer caso de teste da classe seja executado. A anotação `@Test` na linha 8 caracteriza que o método é executado via JUnit. Na linha 10, a anotação `@InitialPage` para o parâmetro do método de teste configura que a execução deve iniciar pela página *web* que a classe `LoginPage` representa (seguindo o padrão Page Object), isto é, a página de autenticação. As linhas 11 e 12 executam, respectivamente, o preenchimento do campo usuário e o clicar no botão "Entrar", ambos elementos do formulário de autenticação. Por fim, a linha 13 obtém o valor do texto de mensagem para o campo de senha do formulário e o compara, na linha 14, com valor esperado e estipulado pela constante definida na linha 6.

Alguns casos de teste em nível de unidade também foram desenvolvidos com a intenção de exercitar os caminhos lógicos das classes mais essenciais que são invocadas durante a execução da funcionalidade de autenticação. Os casos de testes foram sendo definidos com base no resultado esperado, usando critério de Particionamento de Equivalência e observando-se a quantidade de instruções e desvios exercitados que são apresentados pelo relatório de cobertura. A técnica de teste funcional foi utilizada para descrever os casos de teste em nível de unidade. Ressalta-se que a implementação desses casos de teste ocorreu somente após todos os testes funcionais em nível de sistema (apresentados pela Tabela 4.1) terem sidos implementados. A Tabela 4.2 sintetiza os casos de teste para a classe `LoginServiceImpl` para exemplificar o resultado do processo de desenvolvimento do subconjunto de teste em nível de unidade. Essa é uma das classes que fazem parte da sequência de processamento de autenticação do SIGA.

Tabela 4.2: Casos de teste em nível de unidade para a classe `LoginServiceImpl`

ID	Descrição	Resultado Esperado
LS-UT-TC1	Passar credenciais (usuário e senha) válidas para o método `logon(String usuario, String senha)`.	Retornar String (*token* de autenticação) não nula.
LS-UT-TC2	Passar usuário inválido e senha válida para o método `logon(String usuario, String senha)`.	Lançar a exceção `FailedLoginException`.
LS-UT-TC3	Passar quaisquer valor de credenciais e fazer o método `autenticar(String usuario, String senha)` da classe `AutenticacaoSaguiApi` lançar a exceção `AutenticacaoSaguiApiException`.	Lançar novamente a exceção `AutenticacaoSaguiApiException`.

Para elucidar a implementação dos casos de teste em nível de unidade, o fragmento de código 4.2 é apresentado a seguir como a implementação do caso de teste identificado como “LS-UT-TC1”.

```
1  @RunWith(MockitoJUnitRunner.class)
2  public class LoginServiceImplTest {
3
4      @InjectMocks
5      private LoginService loginService = new LoginServiceImpl();
6
7      @Mock
8      private AutenticacaoSaguiApi autenticacaoSaguiApi;
9
10     @Test(expected = FailedLoginException.class)
11     public void deveRelancarExcecaoQuandoFailedLoginExceptionForLancada()
12             throws FailedLoginException, AutenticacaoException {
13         // 'blabla' eh tipicamente um valor de 'usuario' invalido
14         when(autenticacaoSaguiApi.autenticar(eq("blabla"), anyString()))
15             .thenThrow(FailedLoginException.class);
16
17         loginService.logon("blabla", "123546"); // lanca FailedLoginException
18     }
19     ...
20 }
```

Listagem de Código 4.2: Implementação do caso de teste intitulado LS-UT-TC1

Na primeira linha, a anotação `@RunWith` especifica que a execução do métodos da classe `LoginServiceImplTest` é coordenada pelo executor do *framework* Mockito. O atributo `loginService` é anotado com `@InjectMocks`, na linha 4, para que o objeto do tipo `LoginService` seja criado com seus respectivos dublês. A anotação `@Mock` na linha 7 designa que o atributo `autenticacaoSaguiApi` seja instanciado como um objeto dublê do tipo `AutenticacaoSaguiApi`. Na linha 10, a anotação `@Test` caracteriza o método como sendo um teste executado via JUnit. O parâmetro `expected` dessa anotação indica que é esperado que uma exceção do tipo da classe `FailedLoginException` seja lançada na execução do caso de teste. Na linha 14, o método `autenticar` do atributo `autenticacaoSaguiApi` é configurado para que a exceção `FailedLoginException` seja lançada quando a combinação de credenciais for o valor de usuário igual é a `blabla` e a senha seja qualquer valor de texto. Por fim, o método `logon`, alvo do teste, é invocado com os mesmo valores de credenciais esperados na linha 17.

Ao todo, foram implementados 15 casos de teste e aproximadamente 900 linhas foram codificadas entre arquivos XML necessários para a especificação dos cenários de teste e arquivos Java para a execução dos testes[2] (sendo que 473 linhas são exclusivamente de classes que inicializam a execução dos casos de teste). A Tabela 4.3 demonstra a quantidade de casos de teste implementados por classe, bem como

[2]A contagem da quantidade de casos de teste implementados e linhas de código foi realizada manualmente.

sua respectiva quantidade de linhas de código (LOC), e o nível de teste que a classe implementa. Essa tabela está ordenada por nível de teste, quantidade de casos de teste implementados e linhas de código.

Tabela 4.3: Casos de teste por classe para a funcionalidade Autenticação

Classe	Qtde.	LOC*	Nível
LoginIT	6	194	Sistema
AutenticacaoSaguiApiTest	3	69	Unidade
LoginServiceImplTest	3	55	Unidade
LoginBeanTest	2	90	Unidade
AutenticacaoSaguiApiParameterizedTest	1	65	Unidade

* Linhas de código Java

Com um subconjunto mínimo de casos de teste, pôde-se iniciar a implementação do comportamento adaptativo capaz de fazer as mudanças necessárias para restaurar-se para um estado desejado de forma a ampliar a disponibilidade da funcionalidade de autenticação, conforme descrito a seguir.

Autenticação Autocurável:

A introdução da característica de autocura (*self-healing*) na funcionalidade de autenticação almeja tornar o SIGA mais resiliente neste quesito. Para isso, além do mecanismo de autenticação já existente (serviço *web* do SAGUI), dois novos mecanismos de autenticação, denominados secundários, foram implementados, sendo eles: (i) *Lightweight Directory Access Protocol* (LDAP)[3], que é um protocolo para manter informações de forma distribuída e sob uma estrutura hierárquica de diretórios; e (ii) Sistema de Gerenciamento de Banco de Dados (SGBD).

Considerando o SIGA o sistema gerenciado em um SA, foi criado um gerenciador autônomo que pudesse trocar o mecanismo de autenticação em operação analisando os dados coletados pelos monitores de cada respectivo mecanismo (principal e secundários). O processo de adaptação permaneceu desacoplado da funcionalidade de autenticação. Isto quer dizer que o processo de adaptação pode ser inicializado e concluído diversas vezes sem que a autenticação seja executada uma única vez, tendo em vista que a autenticação faz uso apenas do resultado do processo de adaptação.

Ainda sobre os mecanismos de autenticação secundários, embora a funcionalidade de autenticação possa ser restaurada, isto é, opere com um mecanismo secundário de autenticação, o SIGA ainda não estará funcionando plenamente, pois apenas o mecanismo principal de autenticação fornece os dados necessários para carregar as autorizações de um usuário. Ainda assim, com a autenticação operando com um mecanismo secundário, o usuário tem a certeza de que suas credenciais serão válidas

[3] `https://tools.ietf.org/html/rfc1487` – acessado em julho, 2020.

e, por exemplo, que elas poderão ser utilizadas em outros serviços que utilizem as mesmas credencias. Essa característica de autocura é descrita por Rodosek et al. (2009).

A implementação dos mecanismos de autenticação secundários ocorreu de forma incremental. Primeiro, foi implementada a autenticação autocurável com apenas um mecanismo secundário, o LDAP, resultando na versão denotada como $SIGA\_Adapt\_1$. Subsequentemente, foi adicionado um segundo mecanismo de autenticação secundário, via SGBD, resultando na versão denotada $SIGA\_Adapt\_2$. Em paralelo, duas outras versões do SIGA foram geradas com o apoio do *framework* StarMX e utilizando os mesmos mecanismos (LDAP e SGBD). Essas versões foram implementadas também de forma incremental, resultando, respectivamente, nas versões denotadas como $SIGA\_Adapt\_StarMX\_1$ e $SIGA\_Adapt\_StarMX\_2$, conforme apresentado na Figura 4.1. Os detalhes de implementação dessas versões serão apresentados a seguir.

Implementação da Autenticação Autocurável com 1 Mecanismo Secundário:

A introdução da autenticação via LDAP foi facilitada pela simples adição de um artefato ao projeto, disponibilizado pela equipe de Coordenadoria de Sistemas de Informação (CoSI) da Secretaria Geral de Informática (SIn) da UFSCar. Este artefato já possui diversas funções implementadas que podem ser invocadas com pouco esforço. Entre essas funções encontra-se a autenticação por pessoa e senha.

A implementação da autenticação autocurável com 1 mecanismo secundário de autenticação, versão do SIGA denotada como $SIGA\_Adapt\_1$, introduziu elementos arquiteturais relacionados ao modelo MAPE-K. Tais elementos foram concretizados em unidades (classes ou métodos) a fim de representar os componentes básicos do modelo e levar em consideração a separação de interesses e as funções de um gerenciador autônomo. Também criou-se uma nova configuração de sistema para designar qual o mecanismo que deve ser utilizado no momento em que a funcionalidade de autenticação é executada. Essa configuração determina o mecanismo em operação no instante de execução da autenticação.

Assim, parte do modelo MAPE-K foi representado com a criação de classes que exercessem as funções: (i) de monitoramento e registro da situação dos mecanismos de autenticação (tanto principal quanto secundário) no elemento de conhecimento; (ii) de análise dos dados armazenados no elemento de conhecimento; e (iii) de execução da adaptação por meio da reconfiguração do sistema. As particularidades de tais classes são descritas a seguir.

Monitor: apura se o respectivo mecanismo (primário ou secundário) está operante. Tal apuração é inicializada por intermédio de um temporizador. As credencias são enviadas em intervalo de tempos pré-definidos para executar a autenticação. Quando

há uma resposta do mecanismo, sendo a autenticação bem sucedida ou mesmo quando a informação é de que as credenciais não são válidas, é considerado que o mecanismo está operante. Em qualquer outra situação, como *timeout* na execução, por exemplo, considera-se que o mecanismo de autenticação não está operante. O estado de operação do mecanismo é armazenado no elemento de conhecimento.

O fragmento de código 4.3 demonstra a implementação do monitor para o mecanismo de autenticação principal.

```
1  @Schedule(minute="*/1", hour="*") // a cada 1 minuto
2  public void get() {
3      final MecanismoAutenticacao apiRest = MecanismoAutenticacao.API_REST;
4
5      try {
6          autenticacaoSaguiApi.autenticar(USUARIO, SENHA);
7
8          knowledgeAutenticacao = new KnowledgeAutenticacao(apiRest, MecanismoStatus.UP);
9          service.create(knowledgeAutenticacao);
10     } catch (FailedLoginException e) {
11         knowledgeAutenticacao = new KnowledgeAutenticacao(apiRest, MecanismoStatus.UP);
12         service.create(knowledgeAutenticacao);
13     } catch (AutenticacaoSaguiApiException e) {
14         knowledgeAutenticacao = new KnowledgeAutenticacao(apiRest, MecanismoStatus.DOWN);
15         service.create(knowledgeAutenticacao);
16     }
17 }
```

Listagem de Código 4.3: Fragmento de código do monitor do mecanismo de autenticação principal

Na linha 1, o valores da anotação `@Schedule` definem que o método `get()` será executado em intervalos de um minuto. A autenticação por meio do mecanismo principal é executada na linha 6. A linha 8 expressa que a autenticação foi bem sucedida e, portanto, que o mecanismo de autenticação principal está operante. As linhas 10 e 11 expressam que, embora uma exceção do tipo da classe `FailedLoginException` tenha sido lançada pois a autenticação não possuía credenciais válidas, o mecanismo de autenticação principal está operante. As linhas 13 e 14 indicam que ocorreu uma exceção do tipo da classe `AutenticacaoSaguiApiException` ao executar a autenticação e, portanto, que o mecanismo principal não está operante. Nas linhas 9, 12 e 15, a condição do mecanismo principal é armazenada no elemento de conhecimento conforme o resultado da execução da autenticação.

Analisador: conforme um agendamento pré-determinado, checa a situação do mecanismo de autenticação em operação e se este é o mecanismo principal. Nas ocasiões em que o mecanismo configurado em operação for o principal e este não estiver operante, o analisador conclui que é necessário trocar o mecanismo de autenticação para o mecanismo secundário. Inversamente, nas oportunidades em que o mecanismo de operação não for o mecanismo principal e este estiver operante, a conclusão do analisador estabelece que é preciso colocar em operação novamente o mecanismo

de autenticação principal. Assim, quando é preciso efetuar a troca do mecanismo de autenticação, um evento é disparado. Quando o mecanismo em operação for o principal e este estiver operante, ou quando o mecanismo em operação for o secundário e o mecanismo principal não estiver operante, nenhum evento é disparado.

O fragmento de código 4.4 demonstra a implementação do analisador de mecanismos de autenticação.

```
1  @Schedule(minute="*/2", hour="*") // a cada 2 minutos
2  public void analyse() {
3      MecanismoAutenticacao mecanismoEmOperacao = getMecanismoEmOperacao();
4      boolean mecanismoPrincipalEstaEmOperacao = knowledgeAutenticacaoService
5              .isEmOperacao(MECANISMO_PRINCIPAL);
6
7      if (mecanismoEmOperacao.equals(MECANISMO_PRINCIPAL) && !mecanismoPrincipalEstaEmOperacao) {
8          trocarPara(MECANISMO_SECUNDARIO);
9      }
10
11     if (!mecanismoEmOperacao.equals(MECANISMO_PRINCIPAL) && mecanismoPrincipalEstaEmOperacao) {
12         trocarPara(MECANISMO_PRINCIPAL);
13     }
14 }
```

Listagem de Código 4.4: Fragmento de código do analisador de situação dos mecanismos de autenticação

Na linha 1, o valores da anotação `@Schedule` definem que o método `analyse()` será executado em intervalos de dois minutos. O configuração de sistema que indica qual é o mecanismo que deve ser utilizado para validar as credenciais de usuário no momento da execução de uma autenticação é obtida na linha 3. Nas linhas 4 e 5, é verificado se o mecanismo principal de autenticação está operante, ou seja, se ele está respondendo a execução da autenticação. Na linha 7 é avaliado se o mecanismo de autenticação principal é o que deve ser utilizado na execução da autenticação e se este está inoperante. Caso essa avaliação seja verdadeira, a configuração de sistema é alterada na linha 8 para que seja utilizado o mecanismo secundário na execução da autenticação. A avaliação inversa àquela realizada na linha 7 é praticada na linha 11. Por conseguinte, na linha 12 a configuração de sistema é alterada para que seja utilizado o mecanismo principal na execução da autenticação caso a avaliação da linha 11 seja verdadeira.

Executor: através de um evento disparado pelo analisador, efetiva a troca da configuração do sistema que indica qual deve ser o mecanismo utilizado na execução da autenticação.

O fragmento de código 4.5 expressa a implementação do executor. A anotação `@Observes` na linha 1 faz com que o método `execute` seja invocado sempre que um evento do tipo `TrocaMecanismoAutenticacaoEvent` for disparado. A alteração da configuração do sistema é invocada na linha 3, onde é enviado o mecanismo de autenticação que deverá ser utilizado conforme valor obtido na linha 2.

```
1 public void execute(@Observes TrocaMecanismoAutenticacaoEvent trocaMecanismoAutenticacaoEvent) {
2     MecanismoAutenticacao mecanismo = trocaMecanismoAutenticacaoEvent.getMecanismo();
3     atributoSistemaService.update(CHAVE_MECANISMO_AUTENTICACAO, mecanismo.getDBValue());
4 }
```

Listagem de Código 4.5: Fragmento de código do executor do processo de adaptação de autenticação autocurável

No que diz respeito a sensores e atuadores, buscou-se reutilizar trechos de códigos existentes como pontos de controle do recurso gerenciado (SIGA). Logo, as interfaces Java que definem os comportamentos de classes de serviço e de classes que executam regras de negócio, como as classes de autenticação, por exemplo, foram utilizadas para o controle e adaptação do recurso gerenciado.

Embora o gerenciador autônomo tenha sido desenvolvido de forma desacoplada do processamento da autenticação, ainda assim foram necessários ajustes para que a funcionalidade se aproveitasse do resultado de adaptação. Conforme a configuração do sistema, uma nova classe foi introduzida no processamento da autenticação para definir qual mecanismo deve ser utilizado durante a sua execução. O algoritmo 4.6 demonstra o comportamento dessa classe que implementa o padrão de projeto *Factory Method* (Gamma et al., 1994).

```
1  public Autenticacao getMecanismoAutenticacao() {
2      final String mecanismoAutenticacaoAtual = atributoSistemaService
3              .getAttribute(AtributoSistemaConstants.MECANISMO_AUTENTICACAO)
4              .getValor();
5
6      MecanismoAutenticacao mecanismoAutenticacao =
7              MecanismoAutenticacao.valueOf(mecanismoAutenticacaoAtual);
8
9      switch (mecanismoAutenticacao) {
10         case API_REST:
11             return this.autenticacaoSaguiApi;
12         case LDAP:
13             return this.autenticacaoLdap;
14         default:
15             logger.error("Valor invalido para atributo 'MECANISMO_AUTENTICACAO'.");
16             throw new RuntimeException("Mecanismo de autenticacao invalido.");
17     }
18 }
```

Listagem de Código 4.6: Fragmento de código da fábrica de autenticações

Entre as linhas 2 e 4, é obtido no formato de texto o valor da configuração do sistema que estabelece qual o mecanismo deve ser utilizado no processamento da autenticação. Nas linhas 6 e 7, o valor de texto é convertido para um dos valores válidos de mecanismos de autenticação determinados pela enumeração `MecanismoAutenticacao`. Entre as linhas 9 e 13, o valor convertido é avaliado pela instrução `switch` e, em seguida, comparado com os valores esperados de mecanismos de autenticação. Nas ocorrências em que o valor da configuração dos sistema for inválido, por padrão, será lançada uma exceção (indicada na linha 16).

Implementação da Autenticação Autocurável com 2 Mecanismos Secundários:

A implementação da autenticação autocurável com 2 mecanismos de autenticação, versão do SIGA denotada como $SIGA\_Adapt\_2$, se assemelha à versão da implementação com apenas 1 mecanismo secundário ($SIGA\_Adapt\_1$), pois a primeira é evoluída na segunda com a introdução da autenticação via SGBD. Essa introdução culminou na criação de um elemento de monitoramento exclusivo para esse mecanismo e a implementação do seu algoritmo é similar a apresentada pelo fragmento de código 4.3.

O modo como o analisador conclui qual mecanismo deve entrar em operação, quando for o caso, foi alterado. A análise entre os mecanismos principal e secundários é similar a apresentada pelo fragmento de código 4.4. Entretanto, nessa implementação o analisador define qual a melhor opção entre os mecanismos secundários (LDAP e SGBD) para substituir o mecanismo principal. O algoritmo 4.7 apresenta a forma com é realizada a definição entre as opções de mecanismo secundário.

```
1  private MecanismoAutenticacao definirMelhorOpcao() {
2      boolean isLdapEmOperacao = knowledgeAutenticacaoService.
3              isEmOperacao(MecanismoAutenticacao.LDAP);
4      boolean isDBEmOperacao = knowledgeAutenticacaoService.
5              isEmOperacao(MecanismoAutenticacao.DATABASE);
6
7      if (isLdapEmOperacao && isDBEmOperacao) {
8          return knowledgeAutenticacaoService.obterMelhorOpcaoAlternativa();
9      }
10
11     if (isLdapEmOperacao) {
12         return MecanismoAutenticacao.LDAP;
13     } else {
14         return MecanismoAutenticacao.DATABASE;
15     }
16 }
```

Listagem de Código 4.7: Método do analisador para definir a melhor opção entre os mecanismos secundários de autenticação

Entre as linhas 2 e 5 são obtidas situações dos mecanismos secundários de autenticação. A condição expressa na linha 7 avalia se ambos os mecanismos estão operantes. Caso a avaliação sejam verdadeira, o elemento de conhecimento é consultado na linha 8 para definir qual a melhor opção entres os mecanismos secundários. Internamente, o método `obterMelhorOpcaoAlternativa()` obtém os últimos 100 registros sobre a situação de cada mecanismo secundário de autenticação e estabelece que o mecanismo que deve ser utilizado é aquele que tiver o maior número de vezes registrados como operante. Por fim, entre as linhas 11 e 14, é decidido se a autenticação via LDAP ou via SGBD deve ser utilizada.

No que se refere ao executor do processo de adaptação, nenhuma alteração foi necessária. Contudo, a classe que define qual mecanismo será utilizado na execução

da autenticação, originalmente apresentado pelo fragmento de código 4.6, recebeu o acréscimo de mais uma condição em função da possibilidade de autenticação via SGBD. Essa condição adicional é executada quando o valor da configuração do sistema indicar que o SGBD deve ser utilizado como mecanismo de autenticação. Além disso, por se tratar de um forma de autenticação que utiliza um mecanismo inerente ao SIGA, códigos adicionais tiveram que ser implementados. Entre essas implementações estão as classes de autenticação via SGBD, de serviço (que também atua como sensor para o monitoramento do SGBD), de acesso a dados do usuário do sistema e de criptografia.

Implementação da Autenticação Autocurável com 1 Mecanismo Secundário e Apoio do *Framework* StarMX:

A implementação da autenticação autocurável com 1 mecanismo e com o apoio do *framework* StarMX, versão do SIGA denotada como $SIGA\_Adapt\_StarMX\_1$, se assemelha à implementação da versão $SIGA\_Adapt\_1$. Contudo, os dispositivos utilizados para executar o processo de adaptação foram os fornecidos pelo *framework*. Consequentemente, códigos adicionais que implementam as especificações da tecnologia JMX foram necessários. Apesar disso, os comportamentos das unidades de código que representam os componentes básicos do modelo MAPE-K não foram alterados.

Assim, foram criados monitores específicos para cada um dos mecanismos de autenticação (principal e secundário) com o propósito de verificar sua situação e registrá-la no elemento de conhecimento. Nesta versão do SIGA, a verificação dos monitores também é realizada de tempos em tempos e as suas implementações são similares à apresentada pelo fragmento de código 4.3. A diferença nessa implementação foi o uso do mecanismo de ativação por tempo fornecido pelo StarMX para inicializar a automatização do processo de monitoramento.

A similaridade também é válida em relação ao analisador. Ou seja, a implementação do analisador nesta versão foi similar à apresentada pelo fragmento de código 4.4, em que é verificado de tempos em tempos qual é o mecanismo em operação, bem como sua situação, com base nos dados armazenados no elemento de conhecimento. Todavia, tal como a implementação dos monitores, o mecanismo de ativação por tempo fornecido pelo *framework* foi utilizado na implementação desta versão do analisador.

Assim como o comportamento do executor da implementação sem o apoio do StarMX (apresentado pelo fragmento de código 4.5), o executor implementado nesta versão é invocado somente quando o analisador conclui que se deve alterar a configuração do sistema que estabelece qual mecanismo deve ser utilizado no processamento da autenticação.

Em relação aos pontos de controle do SIGA para a execução da lógica de gerenciamento, foram implementados códigos que utilizam recursos da tecnologia JMX para

permitir a comunicação entre o StarMX e o SIGA. A biblioteca Apache DeltaSpike[4] foi adicionada ao projeto SIGA para facilitar essa comunicação e a implementação de sensores e efetuadores no padrão MBean. O fragmento de código 4.8 demonstra a implementação do monitor do mecanismo de autenticação principal.

```
1  @MBean(
2          objectName = "siga.mbean:type=MonitorAutenticacaoSaguiApi",
3          description = "MBean JMX para monitorar o mecanismo de autenticacao da API do SAGUI"
4  )
5  public class MonitorAutenticacaoSaguiApiJMX extends MonitorAutenticacao
6          implements MonitorAutenticacaoSaguiApiJMXMBean {
7      ...
8      @JmxManaged(description = "Verifica o mecanismo principal de autenticacao")
9      @Override
10     public void get() {
11         ...
12     }
13 }
```

Listagem de Código 4.8: Monitor do mecanismo principal de autenticação

Na linha 1, a anotação `@MBean` estabelece que a classe que implementa a interface `MonitorAutenticacaoSaguiApiJMXBean` é um recurso que pode ser instrumentado via JMX e que o método `get()`, na linha 10, é uma operação válida deste recurso pois está anotado com `@JmxManaged` (linha 8).

Para a execução do processo de adaptação, foi preciso ainda constituir o arquivo de configuração do StarMX. O fragmento de código 4.9 ilustra a configuração do *framework* para a versão $SIGA\_Adapt\_StarMX\_1$.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE starmx PUBLIC "-//STAR Lab//StarMX Configuration DTD 1.0//EN" "starmx-1.0.dtd" >
3  <starmx>
4      <mbeanserver id="jboss-eap-6-ms" lookup-type="jmx">
5          <!-- JBoss EAP 6/JBoss AS 7 -->
6          <jmx-param service-url="service:jmx:remoting-jmx://localhost:9999"/>
7      </mbeanserver>
8
9      <mbean id="monitor-autenticacao-sagui-api-mb"
10         object-name="siga.mbean:type=MonitorAutenticacaoSaguiApi"
11         mbeanserver="jboss-eap-6-ms"
12         interface="br.ufscar.siga.ejb.mbean.MonitorAutenticacaoSaguiApiJMXMBean" />
13     ...
14     <execute>
15         <timer-info interval="1" unit="minute" />
16         <process id="sagui_api_auth_policy"
17             javaclass="br.ufscar.siga.manager.process.MonitorAutenticacaoSaguiApiProcess">
18                 <object name="monitorAutenticacaoSaguiApiJMXMBean"
19                     ref="monitor-autenticacao-sagui-api-mb" />
20         </process>
21     </execute>
22     ...
23 </starmx>
```

Listagem de Código 4.9: Arquivo de configuração do StarMX

[4] https://deltaspike.apache.org/

As configurações para comunicação com o servidor de aplicação são estabelecidas pela *tag* `mbeanserver` entre as linhas 4 e 7. A *tag* `mbean`, entre as linhas 9 e 12, é utilizada para especificar as informações sobre o MBean para a instrumentação do monitor do mecanismo de autenticação principal. Entre as linhas 14 e 21 são definidas as informações sobre execução do processo de adaptação. Dentre essas definições, consta que mecanismo de ativação por tempo inicializará o processo de monitoramento do mecanismos principal de autenticação em intervalos de um minuto.

Implementação da Autenticação Autocurável com 2 Mecanismos Secundários e Apoio do *Framework* StarMX:

A última versão que implementa a autenticação autocurável no SIGA é a versão que possui 2 mecanismos secundários e utiliza o *framework* StarMX para automatizar os processos de adaptação. A versão denotada $SIGA\_Adapt\_StarMX\_2$ é uma evolução da versão $SIGA\_Adapt\_StarMX\_1$, em que a principal diferença é a introdução da autenticação via SGBD como um mecanismo secundário. Consequentemente, essa introdução fez com que fosse necessário criar um novo elemento para monitorar a situação desse mecanismo, bem como defini-lo no arquivo de configuração do *framework*, e fosse preciso alterar o algoritmo do analisador do processo de adaptação.

A versão $SIGA\_Adapt\_StarMX\_2$ se assemelha à sua versão paralela, $SIGA\_Adapt\_2$, no que diz respeito a implementação dos comportamentos dos elementos de monitoramento, análise e execução do processo de adaptação. Em outras palavras, os monitores de cada mecanismo de autenticação implementam o comportamento apresentado pelo algoritmo 4.3, o analisador incorpora a decisão entre os mecanismos secundários, demonstrado pelo fragmento de código 4.7, e o executor continua alterando a configuração do sistema somente quando o analisador decide pela troca do mecanismo de autenticação. Todavia, os processos de adaptação nesta versão, assim como na sua versão antecessora ($SIGA\_Adapt\_StarMX\_1$), são inicializados com a utilização dos dispositivos fornecidos pelo StarMX.

Sumário da implementação da autenticação autocurável: Todas as versões que implementaram a autenticação autocurável (versões intituladas $SIGA\_Adapt\_1$, $SIGA\_Adapt\_2$, $SIGA\_Adapt\_StarMX\_1$ e $SIGA\_Adapt\_StarMX\_2$) foram devidamente avaliadas sob o aspecto da execução do processo de adaptação e de seu funcionamento. Testes manuais foram realizados mantendo apenas um dos mecanismos operante por vez para avaliar se o gerenciador autônomo foi capaz de alterar a configuração de sistema do SIGA. Para cada mecanismo em operação, também foram realizados testes *ad hoc* (isto é, não sistemáticos) com o propósito de assegurar que a autenticação continuava a ocorrer satisfatoriamente.

4.2.2 Processamento de Candidatos à Formatura e Formandos

A funcionalidade “Processar Candidatos à Formatura e Formandos” tem o propósito de ajustar o *status* da matrícula de estudantes conforme o cumprimento da matriz curricular em relação à sua conclusão. Assim sendo, durante a execução dessa funcionalidade, é avaliado se o estudante cumpriu determinados requisitos para se tornar um candidato à formatura (condição na qual o estudante estará apto a se tornar formando caso cumpra todos os requisitos de integralização da matriz curricular da sua matrícula) ou, até mesmo, formando (condição na qual o estudante cumpriu integralmente a matriz curricular do curso no qual está matriculado, mas ainda depende da celebração da colação de grau para se formar efetivamente).

Esta funcionalidade é considerada uma “rotina” de um período letivo. Em outras palavras, ela faz parte de uma sequência de procedimentos de gestão acadêmica que são executados periodicamente no SIGA. A periodicidade de execução depende da duração do período letivo. Além da separação por duração, os períodos letivos são divididos por modalidade de ensino (Presencial e Ensino à Distância - EaD, por exemplo), regime de inscrição (por atividade curricular ou seriado) e localidade educacional (quando a modalidade de ensino é presencial). Isto significa que essa “rotina” processa dados em lotes de acordo com a correspondência entre as configurações do período letivo e as informações das matrículas no que se refere à modalidade, ao regime de inscrição, e à localidade educacional do curso.

Em razão de processar dados em lote, esta funcionalidade tende a levar um tempo maior para concluir sua execução quando comparada com uma funcionalidade que não processa dados em volume. Portanto, o seu tempo de execução pode ser de poucos segundos ou até mesmo horas, dependendo do tamanho do lote a ser processado. Isto decorre também pelo fato dos dados não serem processados de forma paralela nem concorrente (tendo em vista que não existe especificação para a tecnologia utilizada pelo SIGA, nem há o uso de algum *framework* no projeto, por exemplo, que implemente o controle de execução concorrente).

No que se refere aos testes, a funcionalidade “Processar Candidatos à Formatura e Formandos” não possuía nenhum caso de teste implementado. Assim, as especificações da funcionalidade foram a base para a elaboração e implementação de um conjunto de casos de teste funcionais em nível de sistema seguindo o critério de teste Particionamento de Equivalência. A Tabela 4.4 sintetiza esse conjunto de casos de teste em que as pré-condições são: (i) o usuário deve estar autenticado; e (ii) o usuário deve ter privilégio para executar a funcionalidade.

Para efeitos de ilustração, o caso de teste da Tabela 4.4, identificado como “RCFF-FT-TC1”, pode ser visualizado pelo fragmento de código 4.10.

Tabela 4.4: Conjunto de casos de teste funcionais em nível de sistema para a funcionalidade de processamento de candidatos à formatura e formandos

ID	Descrição	Resultado Esperado
RCFF-FT-TC1	Usuário executa o processamento em que existe um estudante que possui inscrições com aprovação em quase todas as atividades curriculares da matriz curricular do curso no qual está matriculado. A matrícula desse estudante está com *status* "Cursando" e existem solicitações de inscrição com *status* "ESPERA" para as demais atividades curriculares da matriz curricular no período letivo vigente.	Alterar o *status* da matrícula desse estudante de "Cursando" para "Candidato a Formatura".
RCFF-FT-TC2	Usuário executa o processamento em que existe um estudante que possui inscrições com aprovação em todas as atividades curriculares da matriz curricular do curso no qual está matriculado. A matrícula desse estudante está com *status* "Cursando".	Alterar o *status* da matrícula desse estudante de "Cursando" para "Formando".
RCFF-FT-TC3	Usuário executa o processamento em que existe um estudante que possui inscrições com aprovação em quase todas as atividades curriculares da matriz curricular do curso no qual está matriculado. A matrícula desse estudante está com *status* "Cursando" mas não existem solicitações de inscrição com *status* "ESPERA" para as demais atividades curriculares da matriz curricular no período letivo vigente.	Não alterar o *status* da matrícula desse estudante mantendo-no como "Cursando".
RCFF-FT-TC4	Usuário desiste de iniciar a execução do processamento.	Fechar *popup* do navegador *web* sem iniciar o processamento.
RCFF-FT-TC5	Usuário cancela a execução do processamento depois de iniciá-la.	Cancelar a execução enquanto o processamento estava sendo executado.
RCFF-FT-TC6	Usuário desiste de cancelar a execução do processamento depois de iniciá-la e, em seguida, supor cancelá-la.	Fechar *popup* do navegador *web* e continuar a execução do processamento.

Na primeira linha, a anotação `@RunWih` especifica que a execução dos testes da classe `RotinaCandidatosFormaturaFormandosIT` é coordenada pelo Arquillian. Os dados preliminares para os cenários de teste de todos os casos de teste desta classe são inseridos na base de dados de acordo com as informações dos arquivos XML configurados nos parâmetros da anotação `@DatabaseSetup` (linha 3). O caso de teste é executado pelo *framework* JUnit, conforme especificado na linha 6 pela anotação `@Test`. Os dados a serem inseridos na base de dados para configurar o cenário de teste exclusivo do método são definidos pelos parâmetros da anotação `@DatabaseSetup` entre as linhas 7 e 12. A execução do teste é iniciada a partir da página *web* referenciada pela classe `LoginPage` (seguindo o padrão Page Object), conforme a anotação `@InitialPage` na linha 14. Entre as linhas 17 e 41, o código executa a navegação entre as páginas *web* e aciona a funcionalidade através de cliques pelos elementos de interface. Por fim, é realizada a asserção de que o *status* da matrícula é igual ao valor de texto "Candidato Formatura".

```
@RunWith(Arquillian.class)
...
@DatabaseSetup(value = {"/datasets/core/tipo-vinculo.dbunit.xml", ...})
public class RotinaCandidatosFormaturaFormandosIT {
    ...
    @Test
    @DatabaseSetup(
        type = DatabaseOperation.REFRESH,
        value = {
            "/datasets/siga/matricula-estudante-direito-com-nro-ufscar-2.dbunit.xml",
            ...
    })
    public void deveAlterarStatusDaMatriculaCursandoParaCandidatoFormaturaAoExecutarRotina(
        @InitialPage LoginPage loginPage
    ) {
        // Autentica como usuario admin
        loginPage.login(UsuarioTeste.ADMINISTRADOR);

        // Navega ate pagina de rotinas do periodo letivo
        paginaPrincipal.navegaParaPeriodosLetivosPeloMenu();
        paginaPeriodosLetivos.navegarParaRotinas();

        // Executa rotina "Processar candidatos a formatura e formandos" usando opcao "Todos"
        paginaRotinasPeriodoLetivo
                .clicaEmExecutarRotinaProcessarCandidatosFormaturaFormandos()
                .selecionaOpcao(TODOS)
                .executa();
        paginaRotinasPeriodoLetivo
                .getPopupProcessarCandidatosFormaturaFormandos()
                .executa();
        paginaRotinasPeriodoLetivo
                .getPopupConfirmacaoTerminoProcessamentoCandidatosFormaturaFormandos()
                .fechaAoTerminarProcessamento();

        // Vai para a pagina de matriculas (pois nao eh possivel
        // verificar o resultado pela pagina de rotinas de modo facil)
        paginaPrincipal.navegaParaMatriculasPeloMenu();

        paginaMatriculasDeUmEstudante.preencheNroUFSCar("2").buscaMatriculas();
        final String statusMatricula2 = paginaMatriculasDeUmEstudante
                .getDescricaoStatusMatricula();

        assertThat(statusMatricula2, is(equalTo("Candidato Formatura")));
        ...
    }
    ...
}
```

Listagem de Código 4.10: Implementação do caso de teste intitulado RCFF-FT-TC1

De forma análoga à feita para a funcionalidade de autenticação (Seção 4.2.1), além dos testes funcionais em nível de sistema, alguns casos de teste em nível de unidade foram implementados com a intenção de exercitar os caminhos lógicos das classes mais essenciais que são invocadas durante a execução da funcionalidade de processamento. Os casos de testes foram sendo definidos com base no resultado esperado, usando o

critério Particionamento de Equivalência e observando a quantidade de instruções e desvios. Salienta-se também que a implementação desses casos de teste só ocorreu depois que todos os testes funcionais em nível de sistema (apresentados pela Tabela 4.4) foram implementados. A Tabela 4.5 sintetiza uma amostra dos casos de teste para a classe `InformacoesIntegralizacaoVO` para exemplificar o resultado do processo de desenvolvimento do subconjunto de teste em nível de unidade. Essa classe faz parte da funcionalidade "Processamento de Candidatos à Formatura e Formandos" no SIGA.

Tabela 4.5: Amostra de casos de teste em nível de unidade para a classe `InformacoesIntegralizacaoVO`

ID	Descrição	Resultado Esperado
IIVO-UT-TC1	Fornecer uma matrícula com matriz curricular nula ao instanciar um objeto da classe `InformacoesIntegralizacaoVO` e invocar o método `matrizRequerAtividadesEletivas()` da mesma classe.	Indicar que não deve requerer o cumprimento créditos em atividades curriculares eletivas. Isto é, retornar `false`.
IIVO-UT-TC2	Fornecer uma matrícula com uma matriz curricular que exige o cumprimento mínimo de carga horária em atividades complementares ao instanciar um objeto da classe `InformacoesIntegralizacaoVO` e invocar o método `cumpriuAtividadesComplementares()` da mesma classe.	Indicar que a matrícula cumpriu a carga horária mínima em atividades complementares. Isto é, retornar `true`.

Para demonstrar a concretização do caso de teste da tabela 4.5 identificado como "IIVO-UT-TC1", o fragmento de código 4.11 apresenta sua respectiva implementação. Entre as linhas 1 e 3 são declarados o nome da classe bem como os atributos utilizados pelo caso de teste. Na linha 5, a anotação `@Test` designa o JUnit como executor do caso de teste. A construção e instanciação dos objetos necessários para a execução do caso de teste são realizadas nas linhas 7 e 8. Por fim, a asserção é concretizada na linha 11 com base no retorno do método alvo do teste (executado na linha 9).

```
1  public class InformacoesIntegralizacaoVOTest {
2      private InformacoesIntegralizacaoVO informacoesIntegralizacaoVO;
3      private Matricula matricula;
4      ...
5      @Test
6      public void naoDeveRequerAtividadesEletivasQuandoMatrizNaoSegueUmaMatrizCurricular() {
7          this.matricula = new MatriculaDataBuilder().comMatriz(null).build();
8          this.informacoesIntegralizacaoVO = new InformacoesIntegralizacaoVO(matricula);
9          final Boolean resultado = informacoesIntegralizacaoVO.matrizRequerAtividadesEletivas();
10
11         assertThat(resultado, is(false));
12     }
13     ...
14 }
```

Listagem de Código 4.11: Implementação do caso de teste intitulado IIVO-UT-TC1

Ao todo, foram implementados 33 casos de teste e aproximadamente 5000 linhas foram codificadas entre arquivos XML necessários para a especificação dos cenários de teste e arquivos Java para a execução dos testes[5] (sendo que 765 linhas são exclusivamente de classes que inicializam a execução dos casos de teste). A Tabela 4.6 demonstra a quantidade de casos de teste implementados por classe, bem como sua respectiva quantidade de linhas de código (LOC), e o nível de teste que a classe implementa. Essa tabela está ordenada por nível de teste, quantidade de casos de teste implementados e linhas de código.

Tabela 4.6: Casos de teste por classe para a funcionalidade de processamento

Classe	Qtde.	LOC*	Nível
RotinaCandidatosFormaturaFormandosIT	6	289	Sistema
InformacoesIntegralizacaoVOTest	16	219	Unidade
RotinaCandidatoFormandoServiceImplTest	10	211	Unidade
RotinaCandidatoFormandoServiceImplSpyTest	1	46	Unidade

* Linhas de código Java

Com um subconjunto mínimo de casos de teste, a implementação do comportamento adaptativo que pretende tornar o sistema capaz de buscar oportunidades para otimizar suas próprias operações e aumentar sua eficiência em relação ao recursos computacionais de forma proativa pôde ser iniciada.

Processamento Auto-otimizado de Candidatos à Formatura e Formandos:

A introdução da característica de auto-otimização (*self-optimization*) na funcionalidade "Processamento de Candidatos à Formatura e Formandos" almeja tornar o sistema capaz de buscar proativamente oportunidades para otimizar suas próprias operações e aumentar sua eficiência em relação aos recursos computacionais. Para isso, o lote de matrículas a serem processadas pela funcionalidade, ou seja, as matrículas que correspondem às configurações do período letivo, é subdividido em lote menores de acordo com o número de processadores do ambiente no qual o SIGA está sendo executado. Em outras palavras, um lote de 2.000 matrículas sendo processadas pelo SIGA em um ambiente computacional com 4 processadores, por exemplo, será dividido em sublotes de 500 matrículas. Esses sublotes são processados de forma paralela e assíncrona e levam em consideração a memória disponível no momento da execução, a quantidade de execuções similares em andamento e o tempo máximo de espera (*timeout*) para prosseguir com a execução do processamento nos casos em que os recursos forem escassos.

[5] A contagem da quantidade de casos de teste implementados e linhas de código foi realizada manualmente.

Diferentemente das versões anteriores (implementações da autenticação autocurável), o processo de adaptação nesta versão está intrinsecamente relacionado à execução da funcionalidade de processamento. Isto que dizer que o processo de adaptação é inicializado somente quando o processamento é executado.

A implementação das versões do processamento auto-otimizado ocorreu de forma incremental. Primeiramente, foi implementado o processamento auto-otimizado com apenas um elemento de conhecimento (*knowledge*), culminando na versão denotada como $SIGA\_Adapt\_3$. Nesta versão, o elemento de conhecimento lida com informações a respeito do *status* de execução dos sublotes de matrículas. Na implementação seguinte, um segundo elemento de conhecimento foi adicionado ao processamento auto-otimizado para lidar com informações referentes ao histórico de uso de memória, findando na versão denotada como $SIGA\_Adapt\_4$.

Os detalhes de implementação dessas versões, ambas apresentadas na Figura 4.1, são apresentados a seguir.

Implementação do Processamento Auto-otimizado com 1 Elemento de Conhecimento:

A implementação dessa versão, denotada como $SIGA\_Adapt\_3$, também introduziu elementos relacionados ao modelo MAPE-K. Para isso, o modelo foi representado em unidades de código que exercessem as funções: (i) de monitoramento das quantidades máxima e livre (disponível para uso) de memória e da quantidade de execuções simultâneas do algoritmo de processamento; (ii) de análise dos recursos computacionais referentes à memória e a quantidade de execuções simultâneas do algoritmo de processamento; (iii) de planejamento para a execução do processamento; e (iv) da execução do processamento. As particularidades de tais classes são descritas a seguir:

Monitor: apura as quantidades de máxima de memória, de memória disponível e de outros sublotes que estão sendo processados no mesmo instante. O fragmento de código 4.12 exibe como é realizada a apuração dos recursos computacionais para o processamento auto-otimizado.

```
1  public class MonitorRecursos {
2      ...
3      public void loadData() {
4          this.qtdeExecucoesParalelas = knowledgeLoteRotinaCandidatoFormandoService
5                  .findQtdeEmExecucao();
6          this.freeMemory = Runtime.getRuntime().freeMemory();
7          this.maxMemory = Runtime.getRuntime().maxMemory();
8      }
9      ...
10 }
```

Listagem de Código 4.12: Fragmento de código do monitor de recursos para o processamento auto-otimizado

A quantidade de outros sublotes sendo processados simultaneamente é obtida na linha 4 a partir do elemento de conhecimento e armazenada em uma variável de instância da classe. As quantidades de memória livre e máxima são obtidas, respectivamente, na linhas 6 e 7 e também armazenadas em variáveis de instância da classe.

Analisador: com base nos dados apurados pelo monitor do sublote, avalia se recursos computacionais estão minimamente disponíveis para a execução do processamento dos dados. Mais especificamente, pondera se a quantidade de memória disponível e se a quantidade de sublotes processados simultaneamente estão acima do limite estabelecido por parâmetros do sistema (definidos no código fonte). O fragmento de código 4.13 demonstra a implementação do analisador de recursos para o processamento auto-otimizado.

```
1  public class AnalyserRecursos {
2      ...
3      public boolean temPoucosRecursos() {
4          long freeMemory = monitorRecursos.getFreeMemory();
5          long maxMemory = monitorRecursos.getMaxMemory();
6          long qtdeExecucoesParalelas = monitorRecursos.getQtdeExecucoesParalelas();
7
8          return temPoucaMemoriaDisponivel(freeMemory, maxMemory)
9                  || temMuitasExecucoesParalelas(qtdeExecucoesParalelas);
10     }
11
12     private boolean temPoucaMemoriaDisponivel(long freeMemory, long maxMemory) {
13         final double percentFreeMemory = (freeMemory * 100) / maxMemory;
14         final boolean temPoucaMemoria = percentFreeMemory < 25.0;
15         ...
16         return temPoucaMemoria;
17     }
18
19     private boolean temMuitasExecucoesParalelas(long qtdeExecucoesParalelas) {
20         final boolean temMuitasExecucoesParalelas = qtdeExecucoesParalelas > 2;
21         ...
22         return temMuitasExecucoesParalelas;
23     }
24 }
```

Listagem de Código 4.13: Monitoramento dos recursos computacionais para o processamento auto-otimizado

O método `temPoucosRecursos`, estabelecido na linha 3, utiliza as informações sobre as quantidades de memória livre e máxima e de execuções simultâneas do processamento de sublotes obtidos pelo elemento de monitoramento para invocar os métodos que comparam essas informações com os parâmetros de sistemas definidos nas linhas 14 e 20. Nos casos em que o percentual de memória livre estiver abaixo do esperado ou a quantidade de execuções de processamento de sublotes estiver acima do estabelecido, o analisador conclui (linhas 8 e 9) que os recursos estão escassos no instante da análise para prosseguir com o processamento do sublote.

Planejador: determina as ações que possibilitam a mudança de estado do processamento de um sublote conforme a disponibilidade dos recursos computacionais e tempo máximo de espera (*timeout*). Com base nas informações prestadas pelo analisador, o planejador pode colocar a execução do sublote em um estado de espera, caso os recursos não atinjam os limites mínimos definidos pelos parâmetros de sistema, ou liberá-la. Nas ocasiões em que o tempo de espera pelos recursos mínimos atingir seu limite, a execução é liberada pelo planejador no intuito de não delongar seu processamento.

Executor: efetua o processamento do ajuste do *status* da matrículas do sublote. A execução do processamento o ocorrerá independentemente do processo de adaptação. Ou seja, o processo de adaptação não controla se o processamento será executado, mas sim quando ele será executado.

O processo de adaptação é inicializado através da interceptação da execução do código no ponto em que é invocada a unidade responsável pela verificação do ajuste nos *status* das matrículas. O fragmento de código 4.14 apresenta a maneira como a interceptação ocorre.

```
1  @Interceptors(value = ProcessRotinaCandidatoFormando.class)
2  @Asynchronous
3  public Future<LoteExecucao> calcularStatusMatricula(
4          Collection<Matricula> matriculas,
5          Map<StatusMatriculaEnum, StatusMatricula> statusFinais
6  ) {
7      // event
8      KnowledgeLoteRotinaCandidatoFormando knowledge =
9              knowledgeLoteRotinaCandidatoFormandoService
10             .create(new KnowledgeLoteRotinaCandidatoFormando());
11     ...
12     final LoteExecucao loteExecucao = new LoteExecucao(Thread.currentThread().getId());
13
14     for (Matricula matricula : matriculas) {
15         ... // algoritmo que verifica a necessidade de ajuste no status da matricula
16     }
17     knowledgeLoteRotinaCandidatoFormandoService.registrarConclusao(knowledge);
18     return new AsyncResult<>(loteExecucao);
19 }
```

Listagem de Código 4.14: Fragmento de código interceptado para inicialização do processo de adaptação do processamento auto-otimizado

Na linha 1, a anotação `@Interceptors` define que a classe `ProcessRotinaCandidatoFormando` seja executada antes do método `calcularStatusMatricula(...)` sempre que este for invocado. Essa classe controla o fluxo de execução da adaptação do processamento auto-otimizado. Além disso, a anotação `@Asynchronous` na segunda linha determina que o método pôde ser invocado assincronamente, isto é, sem bloquear o chamador. Isto possibilita que o método seja invocado várias vezes sem que a unidade responsável por sua

invocação precise aguardar por uma resposta para prosseguir com sua execução. Entre as linhas 8 e 12, as informações sobre andamento do processamento do sublote são armazenadas no elemento de conhecimento e as informações pertinentes a quantidade de matrículas processadas e o resultado dos ajustes nos *status* delas são armazenados no objeto `loteExecucao`. O trecho de código compreendido entre o laço `for` verifica a necessidade de ajuste no *status* de cada matrícula do sublote com base no cumprimento dos requisitos da respectiva matriz curricular. Por fim, a conclusão do processamento do sublote é registrado no elemento de conhecimento e as informações sobre as matrículas processadas são retornadas (linha 17 e 18, respetivamente).

A invocação do método `calcularStatusMatricula(...)`, que é interceptado pela unidade responsável pelo processo de adaptação, ocorre após a definição do tamanho dos sublotes de matrículas. O fragmento de código 4.15 apresenta como é definida a quantidade de matrículas por lote.

```
1  ...
2  final List<Matricula> list = new ArrayList<>(matriculas);
3  final double qtdeProcessadores = Runtime.getRuntime().availableProcessors();
4  final double listSize = list.size();
5  final int batchSize = (int) Math.ceil(listSize / qtdeProcessadores);
6  int offset = 0;
7
8  while (offset < matriculas.size()) {
9      ...
10     int limit = offset + batchSize;
11
12     if (limit > matriculas.size()) { // ultimo lote - evita IndexOutOfBoundsException
13         limit = matriculas.size();
14     }
15
16     List<Matricula> loteMatriculas = list.subList(offset, limit);
17     final Future<LoteExecucao> loteExecutado = integralizadora
18             .calcularStatusMatricula(loteMatriculas, statusFinaisMap);
19     ...
20     offset = limit;
21 }
22 ...
```

Listagem de Código 4.15: Algoritmo que define o tamanho dos sublotes para o processamento auto-otimizado

A quantidade de matrículas que cada sublote terá é definida com base na quantidade de processadores disponíveis no ambiente onde o SIGA está sendo executado e atribuída à variável intitulada `batchSize`. Quanto maior o número de processadores, menor o tamanho do sublote. Essa abordagem pode ser notada entre as linhas 2 e 5 do algoritmo 4.15. Na linha 6, a variável de deslocamento, denominada `offset`, recebe um valor que representa o índice inicial para obter um sublote de dados a partir lista de matrículas. A linha 8 controla o laço que define a composição de cada sublote.

Assim, ele será executado enquanto a variável de deslocamento não atingir um valor superior a quantidade total de matrículas a serem processadas. A definição do valor de índice final para o obter cada sublote é definido na linha 10 com base na variável de descolamento e no tamanho definido para o sublote. Quando o valor de índice final superar o tamanho da lista de matrículas a serem processadas, o valor do tamanho da lista de matrículas assumirá o valor de índice para evitar uma exceção na execução do programa. Essa condição pode ser observada entre a linhas 12 e 14. Com os valores inicial e final definidos, o sublote é obtido na linha 16. Na linha 17 e 18, esse sublote então é submetido ao algoritmo que verifica a necessidade de alteração de *status* de casa matrícula. Por fim, valor inicial de índice recebe o valor final na linha 20 para que a próxima iteração do laço obtenha um novo sublote.

Implementação do Processamento Auto-Otimizado com 2 Elementos de Conhecimento:

A evolução da versão do SIGA que corresponde à implementação do processamento auto-otimizado introduziu um novo elemento de conhecimento à execução do processo de adaptação. Exceto pelo objetivo de fazer com que o sistema fosse ainda mais capaz de otimizar sua próprias operações, a forma de execução da implementação da versão denominada $SIGA\_Adapt\_4$ — implementação do processamento auto-otimizado com 2 elementos de conhecimento — manteve-se inalterada. Os modos de definição do tamanho e de obtenção dos sublotes para o processamento assíncrono em função do número de processadores disponíveis no ambiente em que o SIGA é executado foram mantidos. O processo de adaptação continuou sendo inicializado com a interceptação da chamada do método que avalia a necessidade de alteração do *status* da matrícula e as unidades de código que representam os componentes de monitoramento, planejamento e execução do modelo MAPE-K também foram conservadas. O componente de análise foi a único que sofreu alguma alteração nessa versão. O fragmento de código 4.16 apresenta a implementação do analisador acerca da verificação de memória disponível.

O valor do parâmetro do sistema que define o percentual mínimo esperado para dar prosseguimento ao processo de adaptação do processamento auto-otimizado é obtido na primeiras linhas do método `temPoucaMemoriaDisponivel`. Em seguida, o elemento de conhecimento é consultado para obter o valor médio de memória livre das execuções passadas do processamento, a fim de avaliar se o valor configurado para o parâmetro do sistema não destoa dos valores registrados nas execuções. Nas ocasiões em que o valor configurado para o parâmetro do sistema for maior que a média dos valores de memória livre registrados, o valor da média é utilizado para a análise de recursos. Isto impede que o processamento fique em um estado de espera até atingir o limite máximo de tempo estabelecido (*timeout*) em uma eventual superestimação

na configuração do parâmetro do sistema. Ao final, o valor definido para análise é comparado com o percentual de memória livre calculado para indicar se há pouca memória disponível ou não.

```
1  private boolean temPoucaMemoriaDisponivel(long freeMemory, long maxMemory) {
2      final String percentualMinimoMemoriaTxt = atributoSistemaService
3              .getAttribute(PERCENTUAL_MINIMO_MEMORIA_LOOP_RECURSOS)
4              .getValor();
5      Double percentualMinimoMemoriaEstabelecido = Double.valueOf(percentualMinimoMemoriaTxt);
6
7      final OptionalDouble optionalDouble =
8              knowledgeMemoriaLoteRotinaCandidatoFormandoService.obterMediaMemoriaLivre();
9      if (optionalDouble.isPresent()) {
10         final double mediaPercentualMemoriaLivre = optionalDouble.getAsDouble();
11         percentualMinimoMemoriaEstabelecido =
12             percentualMinimoMemoriaEstabelecido.doubleValue() < mediaPercentualMemoriaLivre ?
13             percentualMinimoMemoriaEstabelecido :
14             mediaPercentualMemoriaLivre;
15     }
16
17     final double percentFreeMemory = Percentual.calcular(freeMemory, maxMemory);
18     final boolean temPoucaMemoria = percentFreeMemory < percentualMinimoMemoriaEstabelecido;
19
20     return temPoucaMemoria;
21 }
```

Listagem de Código 4.16: Algoritmo que indica se há pouca memória disponível para o processamento auto-otimizado na versão $SIGA\_Adapt\_4$

Sumário da implementação do processamento auto-otimizado: As versões que implementaram o processamento auto-otimizado (versões intituladas $SIGA\_Adapt\_3$ e $SIGA\_Adapt\_4$) foram devidamente avaliadas sob o aspecto da execução do processo de adaptação e de seu funcionamento. Teste manuais foram realizados para assegurar que a funcionalidade continuava a processar os dados satisfatoriamente. Ademais, ambas foram capazes de otimizar o tempo de execução. Em um dos testes, o tempo de execução foi reduzido de aproximadamente 9 minutos, na versão não adaptativa do SIGA, para cerca de 3 minutos com o processamento auto-otimizado para uma mesma amostra.

4.3 Considerações Finais

Resumem-se, nesta seção, os procedimentos executados no desenvolvimento do estudo de caso, bem como os artefatos produzidos. Ressalta-se que previamente à execução do estudo de caso, os procedimentos do estudo de caso foram sistematizados,

estabelecendo-se um fluxo de processo. A Figura 3.1, apresentada na Seção 3.2.4, ilustra o fluxo estabelecido.

A definição de possíveis cenários de implementação de comportamentos adaptativos (apresentados na Tabela 3.1) e a seleção de um cenário foram as primeiras etapas do processo a serem executadas. A "Autenticação Autocurável" foi o primeiro cenário selecionado. A proposta deste cenário foi tornar o SIGA capaz de descobrir que o mecanismo de autenticação não está operando corretamente e fazer as mudanças necessárias para restaurar a funcionalidade de autenticação para um estado operante. Além disso, a implementação da "Autenticação Autocurável' foi dividida em duas fases: (i) com um mecanismo de autenticação secundário; e (ii) com dois mecanismos de autenticação secundários.

Antes de iniciar a implementação da "Autenticação Autocurável" com um mecanismo de autenticação secundário, a suíte de testes existentes foi executada e identificou-se que o subconjunto de casos de teste dirigidos à funcionalidade de autenticação não era satisfatório. Ou seja, não seria possível comparar as métricas de cobertura da versão não adaptativa com a versão que implementaria a "Autenticação Autocurável". Assim sendo, um conjunto de casos de teste foi elaborado e implementado no intuito de elevar o percentual de cobertura de código no que se refere às instruções e aos desvios de código. A estratégia de implementação do conjunto de casos de testes priorizou os testes em nível de sistema (expostos na Tabela 4.1) em relação aos testes em nível de unidade. Em seguida, as métricas de cobertura de código das classes que estavam envolvidas na execução da funcionalidade de autenticação foram coletadas (resultados relacionados às coberturas estruturais obtidas são apresentados no próximo capítulo).

A primeira fase da implementação da "Autenticação Autocurável" introduziu o LDAP como um mecanismo secundário de autenticação, assim como institui as unidades de código que representaram os componentes de monitoramento (tanto para o mecanismo principal - já existente - quanto para o secundário), análise, execução, e o elemento de conhecimento, referentes ao modelo MAPE-K. Uma nova configuração de sistema também foi criada para indicar ao SIGA qual mecanismo deve ser utilizado no processamento da autenticação. Essa fase perfez a versão denominada $SIGA\_Adapt\_1$. Ao término da implementação, foram realizados ajustes nos casos de teste para retomar a execução da suíte de testes, e as métricas de cobertura de código desta versão foram coletadas.

Na segunda fase da implementação da "Autenticação Autocurável", outro mecanismo de autenticação foi adicionado. A incorporação da autenticação via SGBD como mecanismo secundário fez com que uma nova unidade de monitoramento fosse criada para esse mecanismo, bem como a implementação de unidades de código inerentes à criptografia e validação de credenciais. Além disso, o analisador precisou ser ajustado para que a escolha entre os mecanismos secundários fosse considerada. Para essa

versão do SIGA, chamada de $SIGA\_Adapt\_2$, não foram necessários ajustes no casos de teste. Assim, a suíte de testes foi executada novamente e as métricas de cobertura desta versão foram coletadas.

Paralelamente a essas duas primeiras versões adaptativas do SIGA, versões da "Autenticação Autocurável" com um e dois mecanismos de autenticação também foram implementadas com o apoio do *framework* StarMX. As implementações destas versões seguiram os mesmos passos das versões implementadas sem o apoio do *framework*. Ou seja, implementou-se uma primeira versão, chamada $SIGA\_Adapt\_StarMX\_1$, introduzindo-se o LDAP como um mecanismo secundário de autenticação, ajustando-se os casos de teste, e coletando-se as métricas de cobertura de código. Na sequência, implementou-se uma segunda versão, chamada $SIGA\_Adapt\_StarMX\_2$, adicionando-se o SGBD também como mecanismo secundário de autenticação e coletando-se as métricas de cobertura desta versão.

O segundo cenário selecionado foi o "Processamento Auto-otimizado de Candidatos à Formatura e Formandos". Este cenário propôs tornar o SIGA capaz de empenhar-se proativamente para otimizar o uso de recursos computacionais e aumentar sua eficiência em relação ao tempo de execução da funcionalidade. O ciclo do fluxo de processo foi reiniciado a partir da execução da suíte de testes existentes, onde identificou-se a inexistência de casos de testes dirigidos à funcionalidade de processamento. Desta forma, para que fosse possível comparar as métricas de cobertura da versão não adaptativa com a versão que implementa o processamento auto-otimizado, um conjunto de casos de testes foi elaborado e implementado a fim de elevar os percentuais de cobertura de instruções e de desvios de código. A estratégia de priorizar a implementação dos casos de teste em nível de sistema (expostos na Tabela 4.4) foi mantida e os casos de teste em nível de unidade foram implementados conforme o tempo disponível pelo cronograma de trabalho. Ao término da implementação do conjunto de casos de teste, foram coletadas as métricas de cobertura código das classes que estavam envolvidas na funcionalidade de processamento.

A implementação do "Processamento Auto-otimizado" introduziu novas unidades de código que representaram os componentes de monitoramento, análise e planejamento e o elemento de conhecimento do modelo MAPE-K. Considerou-se a unidade de código cerne do processamento como o componente de execução. Além disso, a implementação considerou a quantidade de processadores disponíveis no ambiente no qual o SIGA for executado para dividir o conjunto de dados submetido ao processamento em sublotes. Após o fim da implementação do comportamento adaptativo, a suíte de teste foi executada novamente e as métricas de cobertura desta versão, intitulada $SIGA\_Adapt\_3$, foram coletadas.

A última versão adaptativa do SIGA, nomeada $SIGA\_Adapt\_4$, adicionou um novo elemento de conhecimento ao "Processamento Auto-otimizado de Candidatos à Forma-

tura de Formandos". Este elemento lida com as informações históricas de memória do ambiente no qual o SIGA foi executado para tornar o elemento de análise mais autônomo. Depois de finalizada a etapa de implementação, a suíte de testes foi executada e as métricas de cobertura foram coletadas.

A evolução de versões do SIGA foi apresentada e pode ser revista pela Figura 4.1.

Capítulo 5

RESULTADOS E ANÁLISE DOS DADOS

5.1 Considerações Iniciais

No capitulo anterior foram apresentados os detalhes de cada implementação, tanto do comportamento original das funcionalidades quanto dos comportamentos adaptativos implementados para os cenários selecionados. Neste capítulo, os resultados são apresentados de forma sumária na Seção 5.2, onde também são analisados os dados e fornecidas respostas às questões de pesquisa que foram estabelecidas no Capítulo 3. O detalhamento do resultados referentes aos testes executados em cada versão do sistema alvo, assim como as análises correspondentes, são expostos no Apêndice A. Na Seção 5.3 são apresentadas as ameaças à validade dos resultados deste trabalho. Ademais, algumas discussões complementares são denotadas na Seção 5.4. As considerações finais são apresentadas no encerramento do capítulo.

5.2 Sumário dos Resultados

No Apêndice A são apresentadas as análises detalhadas da cobertura de código de cada cenário de implementação do SIGA, tanto da versão legada quanto das versões adaptativas. Nesta seção são apresentados os resultados sintetizados com a intenção de destacar os principais pontos da análise.

Nas Tabelas 5.1 a 5.6 apresentam-se as coberturas de código totais das classes envolvidas nas funcionalidades de cada cenário implementado neste estudo de caso. Para cada versão do SIGA, o percentual de cobertura total é representado pela divisão da soma da porção de código coberta (C) pela soma da porção total (T) de código exis-

tente de cada classe. Em termos matemáticos, a cobertura total pode ser representada como:

$$Cobertura\ Total = \frac{\sum C}{\sum T} * 100$$

Nestas tabelas, as versões do SIGA foram agrupadas de acordo com o conjunto de classes a ser analisado. Desta forma, na Tabela 5.3, é mais relevante comparar a cobertura total de código das classes das duas versões adaptativas do "Processamento Auto-otimizado de Candidatos à Formatura e Formandos" com a cobertura total de código das classes da funcionalidade "Processamento de Candidatos à Formatura e Formandos" da versão legada, ainda que a versão $SIGA\_Adapt\_3$ tenha evoluído da versão $SIGA\_Adapt\_2$.

Ressalta-se também que as coberturas de código das versões $SIGA\_Adapt\_2$ e $SIGA\_Adapt\_StarMX\_2$ sem considerar a classe `Bcrypt` (versões marcadas com *) foram calculadas e adicionadas às Tabelas 5.1 e 5.5 para fornecer uma visão mais fidedigna e não distorcer a análise dos resultados, dado que esta classe apresenta valores discrepantes (em inglês, *outliers*) em suas métricas totais quando comparados aos valores das métricas das outras classes da mesma versão, e por ter alto grau de cobertura. Nota-se que a classe `Bcrypt` concentra uma grande parte da quantidade total da maioria das métricas, conforme destacado nas descrições dos resultados de testes das implementações de Autenticação Autocurável.

Análise das Implementações utilizando Java EE:

A adição da capacidade de autocura à funcionalidade de autenticação utilizando apenas tecnologias Java EE acarretou em uma diminuição da cobertura de código total. Essa avaliação pode ser verificada na Tabela 5.1. O decremento na cobertura está associado ao tipo de implementação realizada, no qual o processo de adaptação independe do processo de autenticação. Desta forma, o conjunto de casos de teste direcionados à funcionalidade de autenticação legada em nada exercitou as classes envolvidas no processo de adaptação. A cobertura das classes envolvidas no processo de adaptação foi produzida pela execução dos componentes de monitoramento e análise, inicializados por meio de seus respectivos temporizadores, durante a realização dos testes em nível de sistema.

Além disto, também em função do tipo de implementação, pôde-se perceber, comparando as métricas entre as versões $SIGA\_Adapt\_1$ e $SIGA\_Adapt\_2$ *, que a adição de um novo mecanismo de autenticação degradou a cobertura de código.

De modo geral, a análise mais detalhada descrita no Apêndice A indica que as principais classes pertinentes ao fluxo de validação das credenciais e que já existiam desde a versão legada praticamente não foram impactadas nesta implementação.

Tabela 5.1: Cobertura total das classes envolvidas na Autenticação por versão (Java EE).

Versão	Instr.	Δ	Desvios	Δ	Complex.	Δ	Linhas	Δ	Métodos	Classes
legada	69%		46%		61%		69%		68%	100%
$SIGA\_Adapt\_1$	66%	↓4,4%	38%	↓17,4%	60%	↓1,6%	64%	↓7,3%	68%	96%
$SIGA\_Adapt\_2$	90%		48%		54%		69%		71%	94%
$SIGA\_Adapt\_2$ *	61%	↓11,6%	27%	↓41,3%	56%	↓8,2%	60%	↓13,0%	68%	94%

* não considera classe `BCrypt`

Entretanto, as classes que fornecem interfaces para o controle e a adaptação (que atuam como sensores e atuadores) do recurso gerenciado, isto é, o SIGA, foram impactadas de alguma forma.

As variações de cobertura da versão legada para a versão $SIGA\_Adapt\_1$ foram de perda de 4,4% das instruções e 17,4% de perda dos desvios (Tabela 5.1). As variações de instruções e desvios entre as versões $SIGA\_Adapt\_1$ e $SIGA\_Adapt\_2$ * foram, respectivamente, de 7,6% e 29,0% de perda. Entre as versões legada e $SIGA\_Adapt\_2$, a perda foi de 11,5% de instruções e 41,3% dos desvios.

No que se refere somente às classes criadas para representar os componentes e o elemento de conhecimento do modelo MAPE-K, a cobertura de código total da versão $SIGA\_Adapt\_1$ atingiu percentuais semelhantes à cobertura total em que são contabilizadas todas as classes envolvidas nesta versão que implementa a Autenticação Autocurável (expostas na Tabela 5.1). Em contrapartida, a cobertura de código total da classes alusivas ao modelo MAPE-K da versão $SIGA\_Adapt\_2$ apresentou percentuais abaixo da cobertura total em que todas as classes são consideradas. A queda de cobertura entre as versões está relacionada, principalmente, ao não exercício de métodos das classes de serviço e acesso a dados inerentes ao elemento de conhecimento da versão $SIGA\_Adapt\_2$ para definir a melhor opção entre os mecanismos secundários para substituir o mecanismo principal. A Tabela 5.2 apresenta os percentuais de cobertura de código total do conjunto de classes referentes ao modelo MAPE-K das versões do SIGA que implementam a Autenticação Autocurável.

Tabela 5.2: Cobertura total das classes referentes ao modelo MAPE-K da Autenticação Autocurável

Versão	Instruções	Desvios	Complexidade	Linhas	Métodos	Classes
$SIGA\_Adapt\_1$	66%	38%	61%	63%	68%	91%
$SIGA\_Adapt\_2$	55%	22%	53%	55%	64%	92%

Em relação à funcionalidade de processamento, a adição da capacidade de auto-otimização, tendo em vista que o processo de adaptação é intrínseco à execução do processamento de dados, elevou a cobertura de código total. A Tabela 5.3

ressalta essa análise. Desta forma, o conjunto de casos de teste (particularmente, os testes em nível de sistema) direcionados à funcionalidade, por consequência, exercitou as classes envolvidas no processo de adaptação. Além disso, a análise mais apurada no Apêndice A indica que a cobertura das classes consideradas essenciais para a funcionalidade de processamento foram pouco impactadas.

Tabela 5.3: Cobertura total das classes envolvidas no Processamento de Candidatos à Formatura e Formandos por versão

Versão	Instr.	Δ	Desvios	Δ	Complex.	Δ	Linhas	Δ	Métodos	Classes
legada	20%		13%		19%		21%		30%	97%
$SIGA\_Adapt\_3$	22%	↑10,0%	14%	↑7,7%	21%	↑10,5%	24%	↑14,3%	35%	95%
$SIGA\_Adapt\_4$	22%	↑10,0%	14%	↑7,7%	21%	↑10,5%	24%	↑14,3%	35%	93%

A alta estabilidade dos percentuais de cobertura entre as versões $SIGA\_Adapt\_3$ e $SIGA\_Adapt\_4$ está relacionada à alteração de menor porte que foi realizada entre elas. Entre a versão legada e as versões $SIGA\_Adapt\_3$ e $SIGA\_Adapt\_4$, a variação percentual de instruções e desvios foi elevada em 10,0% e 7,7%, respectivamente.

Quanto às classes criadas para representar os componentes e o elemento de conhecimento em referência ao modelo MAPE-K, as coberturas de código totais das versões $SIGA\_Adapt\_3$ e $SIGA\_Adapt\_4$ (apresentadas pela Tabela 5.4) foram bem superiores às coberturas totais na qual são contabilizadas todas as classes envolvidas no Processamento Auto-otimizado (apresentadas pela Tabela 5.3). Esses resultados estão associados ao fato de que cerca de um terço das classes envolvidas na funcionalidade de processamento (em sua maioria, classes de serviço e acesso a dados) disponibilizam funções para outras funcionalidades. Além disto, algumas destas classes possuem um número elevado de instruções e desvios, muitos destes não cobertos, quando comparadas às classes mais essenciais da funcionalidade processamento, agregando pouco ao percentual de cobertura total.

Em relação ao decremento nos percentuais de cobertura total da versão $SIGA\_Adapt\_3$ para $SIGA\_Adapt\_4$, o crescimento da classe que representa o componente de análise, a adição de uma nova classe de serviço específica para o elemento de conhecimento com baixa cobertura e a adição de uma nova classe de entidade sem cobertura, são os principais fatores que conduziram à perda.

Tabela 5.4: Cobertura total das classes referentes ao modelo MAPE-K da Processamento Auto-otimizado

Versão	Instruções	Desvios	Complexidade	Linhas	Métodos	Classes
$SIGA\_Adapt\_3$	70%	55%	53%	72%	67%	86%
$SIGA\_Adapt\_4$	58%	46%	44%	61%	53%	80%

Análise das Implementações com o apoio do StarMX:

Assim como nas implementações utilizando Java EE, a adição da capacidade de autocura à funcionalidade de autenticação com o apoio do *framework* StarMX também provocou um decremento na cobertura de código total. Do mesmo modo, a inserção de um novo mecanismo de autenticação degradou a cobertura de código (comparação entre as versões $SIGA\_Adapt\_StarMX\_1$ e $SIGA\_Adapt\_StarMX\_2$ *). A Tabela 5.5 quantifica essa análise.

Tabela 5.5: Cobertura total das classes envolvidas na Autenticação por versão (StarMX)

Versão	Inst.	Δ	Desvios	Δ	Complex.	Δ	Linhas	Δ	Métodos	Classes
legada	69%		46%		61%		69%		68%	100%
$SIGA\_Adapt\_StarMX\_1$	63%	↓8,7%	41%	↓10,9%	61%	–	61%	↓11,6%	68%	97%
$SIGA\_Adapt\_StarMX\_2$	90%		48%		55%		67%		71%	95%
$SIGA\_Adapt\_StarMX\_2$ *	57%	↓17,4%	26%	↓43,5%	58%	↓4,9%	58%	↓15,9%	69%	95%

* não considera classe `BCrypt`

De forma geral, tal qual as implementações da Autenticação Autocurável sem o apoio do StarMX, as principais classes pertinentes ao fluxo de validação das credenciais que já existiam desde a versão legada praticamente não foram impactadas nas implementações com o apoio do *framework* e as classes que fornecem interfaces para o controle e a adaptação do recurso gerenciado foram impactadas de alguma forma.

A cobertura total entre as versões legada e $SIGA\_Adapt\_StarMX\_1$ resultou em uma perda de 8,7% das instruções e 10,9% dos desvios. Entre as versões $SIGA\_Adapt\_StarMX\_1$ e $SIGA\_Adapt\_StarMX\_2$ *, a cobertura total das instruções e dos desvios degradou, respectivamente, 9,5% e 36,6%. Entre a versão legada e a versão $SIGA\_Adapt\_StarMX\_2$ *, a perda de cobertura foi de 17,4% das instruções e 43,5% dos desvios.

Em relação às classes que representam algum componente ou elemento de conhecimento, referentes ao modelo MAPE-K, quatro das seis métricas da versão $SIGA\_Adapt\_StarMX\_1$ atingiram percentuais de cobertura total um pouco abaixo daqueles em que foram contabilizados todas as classes envolvidas na Autenticação Autocurável (apontados na Tabela 5.5). Por outro lado, a métrica de complexidade que atingiu percentual superior. Com respeito à versão $SIGA\_Adapt\_StarMX\_2$, exceto para a métrica de complexidade, os percentuais de cobertura total deste conjunto de classes ficaram abaixo da cobertura total em que todas as classes são consideradas. Ademais, de forma destoante, a cobertura de desvios degradou em 57,7% (diferença entre os 38% da versão $SIGA\_Adapt\_StarMX\_1$ e 11% da versão $SIGA\_Adapt\_StarMX\_2$).

Tabela 5.6: Cobertura total das classes referentes ao modelo MAPE-K da Autenticação Autocurável (StarMX)

Versão	Instruções	Desvios	Complexidade	Linhas	Métodos	Classes
$SIGA\_Adapt\_StarMX\_1$	61%	38%	63%	60%	68%	92%
$SIGA\_Adapt\_StarMX\_2$	50%	11%	58%	52%	67%	93%

Revisitando as Questões de Pesquisa

A seguir revisitam-se as questões de pesquisa definidas no Capítulo 3 (Seção 3.2.2).

QP1: *Qual é o impacto sobre a cobertura de código das classes que já existiam em relação a funcionalidade que recebeu a introdução de um comportamento adaptativo?*

Notou-se, neste estudo de caso, que a cobertura da maioria das classes que já existiam não sofre impactos com a introdução de um comportamento adaptativo, e que as classes que sofreram algum impacto significativo na cobertura de código foram aquelas que fornecem interfaces para que o gerenciador autônomo exerça o controle e a adaptação do recurso gerenciado, ou seja, as que atuam como sensores e/ou atuadores, e aquelas que precisam de adequações na lógica de negócio para comportar a introdução do comportamento adaptativo. Ademais, mesmo não se tendo observado um padrão de impacto (negativo ou positivo) na cobertura de código, implementações em que a lógica de adaptação é independente da lógica de negócio parecem contribuir para a diminuição da cobertura geral, enquanto que nos casos em que a implementação da lógica de adaptação é inerente à lógica de negócio, a cobertura de código geral é elevada.

QP2: *Qual é a cobertura de código das novas classes que foram introduzidas com a implementação do comportamento adaptativo?*

A cobertura de código das classes que foram introduzidas com a implementação do comportamento adaptativo parece estar diretamente relacionada à existência de testes em nível de sistema, tendo em vista que os casos de teste em nível de unidade já existentes não estendem seu exercício a novas classes. Ainda assim, com exceção às classes de entidade (aquelas destinadas ao mapeamento objeto-relacional), observou-se que as classes que são intrínsecas à execução da funcionalidade adaptativa tendem a atingirem níveis de cobertura relevantes (metade ou mais das métricas com percentual igual ou superior a 50%) por incumbência da execução dos testes em nível de sistema já existentes, enquanto que as classes independentes da lógica de negócio só são exercitadas, também em nível de sistema, se forem parte da execução

da lógica de adaptação que é inicializada em função do tempo ou pelo disparo de um evento.

5.3 Ameaças à Validade

A validade dos resultados em um estudo de caso é uma questão que deve ser considerada (Wohlin et al., 2012). Quaisquer elementos que possam enfraquecer as conclusões ou a causalidade entre as variáveis, por exemplo, devem ser destacados. Desta forma, algumas ameaças à validade – de conclusão, de construção e externa – deste trabalho são apresentadas nesta seção.

A validade de conclusão diz respeito às questões que afetam a capacidade de tirar conclusões sobre as relações entre o tratamento e o resultado (Wohlin et al., 2012). Neste âmbito, as sutis diferenças de implementação dos mesmos cenários entre os ramos de desenvolvimento (Java EE e StarMX) podem ter comprometido, mesmo que de forma ínfima, a confiabilidade da implementação de tratamentos quando comparadas entre si as versões correlatadas destes ramos.

A validade de construção está relacionada a generalização dos resultados para o conceito ou teoria por trás do estudo (Wohlin et al., 2012). Neste contexto, existe a possibilidade das ações do pesquisador terem enviesado de alguma forma os resultados, tendo em vista que a pessoa que conduziu as implementações também perfez o papel de avalista dos resultados. Além disso, a ausência da casos de teste em nível de integração para a produção de cobertura de código pode ter causado ameaças à validade de viés mono-operação.

A validade externa corresponde a generalização dos resultados para outros domínios além daquele em que o estudo é conduzido (Wohlin et al., 2012). O fato de ter sido um utilizado apenas um sistema alvo no estudo deste trabalho é um elemento que pode ser considerado uma ameaça desta natureza.

5.4 Discussões Complementares

Como parte do processo natural de aprendizagem que envolve um estudo, uma parcela do conhecimento adquirido durante o período de desenvolvimento do projeto e análise dos resultados pode ser traduzido em algumas lições aprendidas, que são apresentadas a seguir.

Pode ser Difícil Aumentar a Cobertura de um Código Legado

Estudos como os de Steinberg (2001) e Aniche et al. (2013) indicam que a criação de testes de unidade pode tornar o código mais coeso. No sentido inverso, a ausência de testes neste nível pode resultar na construção de classes difíceis de testar posteriormente.

Por se tratar de um *software* escasso de testes, em relação ao seu porte, o *design* de algumas classes do SIGA dificultou a criação de testes. Estas classes apresentaram alguns problemas como alguns daqueles catalogados por Aniche e Gerosa (2012), em que podem ser citados “Muitos Testes Para Uma Classe”, “Cenário Muito Grande” “Testes em Método Que Não É Público” e “Objetos Dublê em Excesso”.

Desta forma, vários trechos de código (especialmente da funcionalidade “Processamento de Candidatos à Formatura e Formandos”) não foram exercitados por testes de unidade pela dificuldade de se criá-los.

O Processo de Desenvolvimento Paralelo com Incorporações Impactou no Andamento da Pesquisa

Embora o processo de desenvolvimento (Seção 3.4) tenha sido elaborado para encurtar a distância entre o projeto original e o projeto de pesquisa a fim de manter um alto grau de realismo para o estudo de caso, a incorporação regular de código criou dificuldades de evolução ao longo de sua execução. As tecnologias utilizadas e as decisões de *design* tomadas no projeto original influenciam diretamente no grau de dificuldade de desenvolvimento da pesquisa. Além disto, por algumas vezes foi preciso resolver conflitos antes inexistentes ao incorporar as alterações do projeto original nos ramos das versões do projeto de pesquisa. Este é um ponto que poderia ser mais árduo de resolver caso o autor não tivesse conhecimento prévio do código do sistema legado.

Implementações não Idênticas Dificultam a Análise

Em um trabalho com viés na avaliação diferencial entre implementações de um mesmo cenário, as sutis diferenças de código, mesmo que representem o mesmo algoritmo (como, por exemplo, uma quebra de linha entre chamadas de métodos de um objeto - ao invés de um encadeamento de chamadas *inline* - ou a extração de uma chamada para uma variável, a fim de melhorar a legibilidade do código), repercutiram nos resultados. Alguns trechos de código das implementações da Autenticação Autocurável (versões Java EE e StarMX) ficaram diferentes, embora representem o mesmo algoritmo. Essas diferenças tiveram que ser consideradas, revisitando o código, e minimizadas no momento da análise.

Reduzir as Diferenças entre o Sistema Utilizado no Processo Experimental do Sistema Alvo do Projeto

Um ponto integrante do planejamento do estudo foi avaliar as ferramentas de apoio (Seção 3.5) para a condução de algumas implementações para este trabalho. Para esta avaliação, projetou-se uma aplicação *web* com uma arquitetura similar, mas mais enxuta (sem as camadas de serviço e acesso a dados), e que utilizasse a mesma plataforma de operação do sistema alvo do projeto, isto é, a Java EE na versão 6. Embora o *framework* StarMX tenha se integrado bem a esta aplicação *web* mais simplista, a integração com o SIGA não foi plena, dado que a tecnologia CDI (utilizando a implementação Weld[1]) não é suportada pelo *framework*. Esta característica limitou um pouco a implementação da Autenticação Autocurável com o apoio do *framework*.

A Tecnologia Java EE Possui Recursos para Atingir Cobertura Equiparável à Utilização do *Framework* StarMX

Nas implementações em que foram empregadas as tecnologias já existentes atingiram-se níveis de cobertura de código equiparáveis àquelas implementações que utilizaram o *framework* StarMX. Além disto, foi preciso desenvolver uma quantidade levemente menor de código para implementar os cenários em que não foi utilizado o StarMX. Por outro lado, este *framework* possui outros recursos que não foram utilizados nas implementações deste trabalho como, por exemplo, a coleta de dados estatísticos sobre a execução de processo de adaptação e escopos de memória para armazenar dados para usos futuros ou para trocar dados com outros processos de adaptação. Caso esses recursos fossem utilizados, a criação de uma versão análoga apenas com tecnologias já existentes (isto é, Java EE) provavelmente demandaria a criação de mais código.

5.5 Considerações Finais

Neste capítulo foram apresentados os resultados sintéticos de cobertura de código das versões do SIGA, tanto da versão legada quanto das versões que implementam algum comportamento adaptativo. Além disso, as coberturas das funcionalidades das versões que implementam algum comportamento adaptativo foram comparadas com a cobertura da versão legada. Em seguida, respostas às questões de pesquisa foram elaboradas tendo em vista a análise sobre os resultados obtidos. Por fim, as

[1] `https://weld.cdi-spec.org/` – acessado em agosto, 2020

ameaças à validade dos resultados e discussões complementares sobre os processos de desenvolvimento e análise foram apresentados.

Capítulo 6

CONCLUSÃO

Conforme apresentado nos capítulos anteriores, a *Web* tornou-se ubíqua através da expansão da rede global de computadores (Brandon, 2008; Ginige e Murugesan, 2001). Pessoas podem acessar diversas informações de diferentes lugares sob as mais variadas condições de conectividade. Essa circunstância faz com que a *Web* seja um meio extremamente importante para o consumo de bens e serviços. Diferentes áreas de atuação usufruem amplamente de aplicações baseadas na *Web* para exercerem suas atividades (Pressman e Lowe, 2009).

Diante desse cenário, a exigência por aplicações *web* mais modernas tornou-se maior em relação a serem cada vez mais articuláveis, performáticas, resilientes e seguras. Pesquisas na área de Sistemas Adaptativos (SA) por vezes direcionam o alvo do estudo para as aplicações *web* no intuito de explorar novas abordagens (Castañeda et al., 2014; Moreno et al., 2018; Raufi, 2011). A incorporação de comportamentos adaptativos neste tipo de aplicação auxilia a torná-las mais autonômicas (com o menor número de intervenções humanas possível), liberando os profissionais de TI de detalhes de operação e manutenção de sistemas (Kephart e Chess, 2003) e adequando-se ao contexto de seus usuários.

Um dos desafios da engenharia de *software* é transformar um sistema legado em um sistema adaptativo (Salehie e Tahvildari, 2009), tendo em vista os custos e riscos para substituir um sistema legado (Warren, 1999). A introdução de comportamentos adaptativos em um sistema legado, como em qualquer processo de evolução de *software*, também pode conduzir à inserção de defeitos. Dessa forma, a atividade de teste é fundamental para revelar defeitos (Myers et al., 2011). Além disso, como as aplicações *web* evoluem de forma rápida (Brandon, 2008) e contínua (Pressman e Lowe, 2009), a estratégia de teste de regressão, alinhada às informações sobre a cobertura de código produzida pelos testes, auxilia nas atividades relacionadas à manutenção evolutiva do *software*.

Neste contexto, um estudo de caso foi conduzido para analisar como a introdução de comportamentos em uma aplicação web legada do “mundo real” impacta a

cobertura de código produzida por testes. Os dados quantitativos de cobertura de código de cada versão implementada neste trabalho foram comparados. Os resultados obtidos indicam que a introdução de comportamentos adaptativos em uma aplicação *web* legada têm impacto relevante sobre a cobertura total. A depender do tipo de implementação (inerente ou não à funcionalidade modificada) e dos níveis de testes da suíte existente, a cobertura pode degradar ou aumentar. Ressalta-se também que mesmo as modificações relativamente pequenas podem afetar consideravelmente os percentuais de cobertura da unidade alterada. Além disso, as implementações que se não utilizaram do *framework* StarMX foram mais enxutas em termos de linhas de código que foram adicionadas e obtiveram o mesmo resultado de funcionamento.

Por fim, foi observado que adaptações como aquela implementada no "Processamento Auto-otimizado" exigem que as ferramentas para apoiar o processo de adaptação utilizem dispositivos de ativação que vão além dos conhecidos (tempo e evento), tendo em vista que uma adaptação mais dinâmica não pode ficar limitada a um cronograma de execução nem iniciar uma *thread* em paralelo após o disparo de um evento.

6.1 Contribuições

O estudo de caso desenvolvido neste trabalho contribuiu tanto para o meio acadêmico quanto para a instituição que detém o código da aplicação *web* que foi utilizada como sistema alvo do projeto de mestrado.

De forma direta, a UFSCar, detentora do código fonte do SIGA, se beneficiou do incremento da suíte de testes da aplicação *web*, dado que os casos de teste criados para aumentar a cobertura de código (antes que fossem introduzidos comportamentos adaptativos à aplicação) foram desenvolvidos na base de código do projeto original do SIGA e são constante executados em ambiente de integração contínua de código. Além disto, oportunidades de melhoria de *design* de código foram identificadas e transmitidas à equipe de desenvolvimento do SIGA para que essas questões possam ser trabalhadas no devido tempo.

No que diz respeito às principais contribuições acadêmicas, destacam-se:

- A elaboração de cenários de implementação de comportamentos adaptativos que não são restritos exclusivamente à infraestrutura de operação;
- O detalhamento da introdução de dois comportamentos adaptativos em um sistema de informação legado do "mundo real" enquadrado na categoria Transacional de aplicações *web*; e

- A avaliação do impacto da introdução de comportamentos adaptativos sobre a cobertura de código – produzida pelos testes – através da comparação de métricas entre versões.

6.2 Limitações

Mesmo que os artefatos e resultados deste trabalho tenham promovido contribuições, este estudo de caso possui algumas limitações, das quais podem ser citadas:

- A utilização de um único sistema alvo atenuou a profundidade (fato de se basear em múltiplas fontes de evidência) do estudo de caso;
- O número reduzido de cenários implementados diminuiu a variação de versões para a análise de comparação cobertura de código;
- A criação dos casos de teste com o cenários de implementação já conhecidos podem, mesmo que inconscientemente, ter acarretado algum viés na sua elaboração;
- A ausência de casos de teste em nível de integração parece ter reduzido as chances das novas classes (que foram introduzidas com a implementação do comportamento adaptativo) atingirem níveis de cobertura maiores;
- A deficiência de integração entre o *framework* StarMX e a aplicação *web* alvo do estudo; e
- As pequenas diferenças de código, mesmo que representem o mesmo algoritmo, impactam na análise comparativa humana e manual dos resultados.

6.3 Trabalhos Futuros

O escopo deste trabalho é limitado às suas proposições. Ainda assim, há outras perspectivas que se relacionam e podem abranger outros aspectos para dar continuidade a este trabalho e contribuir para o complemento do mesmo. Tais perspectivas podem ser constituídas nos seguintes trabalhos futuros:

- Avaliar o esforço para implementar casos de teste após a introdução de comportamentos adaptativos para restabelecer a cobertura existente antes da introdução;

- Planejar testes adicionais para cobrir os casos da implementação do gerenciador autônomo; e
- Investigar outras variedades de dispositivo de ativação do processo de adaptação, além de tempo e evento.

REFERÊNCIAS

ABILDGAARD, N. R. *Perspectives on Architecture, Evolution, and Future of Web Applications*. MSc Dissertation, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, 2018.

ALONSO, G.; CASATI, F.; A. KUNO, H.; MACHIRAJU, V. *Web Services - Concepts, Architectures and Applications*. Springer-Verlag Berlin Heidelberg, 2004.

AMOUI, M. Evolving Software Systems Towards Adaptability. In: *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, Lille, France: IEEE, p. 299–302, 2009.

ANICHE, M. F.; GEROSA, M. A. How the Practice of TDD Influences Class Design in Object-Oriented Systems: Patterns of Unit Tests Feedback. In: *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)*, Natal, RN, Brazil: IEEE, p. 1–10, 2012.

ANICHE, M. F.; OLIVA, G. A.; GEROSA, M. A. What Do the Asserts in a Unit Test Tell Us about Code Quality? A Study on Open Source and Industrial Projects. In: *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, Genova, Italy: IEEE, p. 111–120, 2013.

ASADOLLAHI, R.; SALEHIE, M.; TAHVILDARI, L. StarMX: A Framework for Developing Self-Managing Java-based Systems. In: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Vancouver, BC, Canada: IEEE, p. 58–67, 2009.

BRABERMAN, V.; D'IPPOLITO, N.; KRAMER, J.; SYKES, D.; UCHITEL, S. An Extended Description of MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation. In: DE LEMOS, R.; GARLAN, D.; GHEZZI, C.; GIESE, H., eds. *Software Engineering for Self-Adaptive Systems III. Assurances*, Cham: Springer International Publishing, p. 377–408, 2017.

BRANDON, D. M. *Software Engineering for Modern Web Applications: Methodologies and Technologies*. Hershey, PA, USA: IGI Global, 2008.

CAMARGO, K. G. *Elaboração de um Processo de Teste com Base em um Estudo de Caso Real*. MSc Dissertation, Universidade Federal de São Carlos, São Carlos - SP - Brasil, 2012.

CARZANIGA, A.; GORLA, A.; PERINO, N.; PEZZÈ, M. Automatic Workarounds for Web Applications. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Santa Fe, NM, USA: ACM, p. 237–246, 2010.

CASTAÑEDA, L.; VILLEGAS, N. M.; MÜLLER, H. A. Self-adaptive Applications: On the Development of Personalized Web-tasking Systems. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Hyderabad, India: Association for Computing Machinery, p. 49–54, 2014.

CHENG, S.; GARLAN, D.; SCHMERL, B. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Vancouver, BC, Canada: IEEE, p. 132–141, 2009.

CORNETT, S. Code Coverage Analysis. Acesso em: 10 de jul. de 2020, 2014. Disponível em `https://www.bullseye.com/coverage.html`

CRUZES, D. S. *Análise Secundária de Estudos Experimentais em Engenharia de Software*. PhD Thesis, Universidade Estadual de Campinas, Campinas - SP - Brasil, 2007.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. 2 ed. Elsevier, 2016.

ELBAUM, S.; GABLE, D.; ROTHERMEL, G. The Impact of Software Evolution on Code Coverage Information. In: *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, Florence, Italy: IEEE, p. 170–179, 2001.

FOWLER, M. PageObject. Acesso em: 10 de ago. de 2020, 2013. Disponível em `https://martinfowler.com/bliki/PageObject.html`

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 ed. Addison-Wesley Professional, 1994.

GARLAN, D.; CHENG, S. .; HUANG, A. .; SCHMERL, B.; STEENKISTE, P. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, v. 37, n. 10, p. 46–54, 2004.

GERASIMOU, S.; CALINESCU, R.; SHEVTSOV, S.; WEYNS, D. UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles. In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Buenos Aires, Argentina: IEEE Press, p. 83–89, 2017.

GINIGE, A.; MURUGESAN, S. Web Engineering: An Introduction. *IEEE MultiMedia*, v. 8, n. 1, p. 14–18, 2001.

GRAHAM, D.; VAN VEENENDAAL, E.; EVANS, I. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008.

IBM *An Architectural Blueprint for Autonomic Computing*. White paper, IBM Corporation, Hawthorne, NY, USA, 2005.
Disponível em `https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf`

DE LA IGLESIA, D. G. MAPE-K Formal Templates for Self-Adaptive Systems: Specifications and descriptions. 2014.

IVANKOVIĆ, M.; PETROVIĆ, G.; JUST, R.; FRASER, G. Code Coverage at Google. In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia: Association for Computing Machinery, p. 955–963, 2019.
Disponível em `https://doi.org/10.1145/3338906.3340459`

KEPHART, J. O.; CHESS, D. M. The Vision of Autonomic Computing. *Computer*, v. 36, n. 1, p. 41–50, 2003.

KLÖES, V.; GOETHEL, T.; GLESNER, S. Parameterisation and Optimisation Patterns for MAPE-K Feedback Loops. In: *Proceedings of the 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Tucson, AZ, USA: IEEE, p. 13–18, 2017.

KRUPITZER, C.; ROTH, F. M.; PFANNEMÜLLER, M.; BECKER, C. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In: *Proceedings of the 13th IEEE International Conference on Autonomic Computing (ICAC)*, Würzburg, Germany: IEEE, p. 308–314, 2016.

KRUPITZER, C.; ROTH, F. M.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, v. 17, p. 184 – 206, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian, 2015.

LALANDA, P.; DIACONESCU, A.; JULIE, M. A. *Autonomic Computing - Principles, Design and Implementation*. Undergraduate Topics in Computer Science. Springer-Verlag London, 288 p., 2013.
Disponível em `https://hal.archives-ouvertes.fr/hal-00854882`

LI, H.; CHEN, T.; HASSAN, A. E.; NASSER, M.; FLORA, P. Adopting Autonomic Computing Capabilities in Existing Large-Scale Systems. In: *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, Gothenburg, Sweden: IEEE, p. 1–10, 2018.

MORENO, G. A.; SCHMERL, B.; GARLAN, D. SWIM: An Exemplar for Evaluation and Comparison of Self-adaptation Approaches for Web Applications. In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Gothenburg, Sweden: ACM, p. 137–143, 2018. Disponível em `http://doi.acm.org/10.1145/3194133.3194163`

MULCAHY, J. J.; HUANG, S. Autonomic Software Systems: Developing for Self-Managing Legacy Systems. In: *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Victoria, Canada: IEEE, p. 549–552, 2014.

MULCAHY, J. J.; HUANG, S. An Autonomic Approach to Extend the Business Value of a Legacy Order Fulfillment System. In: *Proceedings of the 9th Annual IEEE Systems Conference (SysCon)*, Vancouver, BC, Canada: IEEE, p. 595–600, 2015.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3 ed. John Wiley & Sons, 2011.

PRESSMAN, R. S.; LOWE, D. *Web Engineering : A Practitioner's Approach*. 1 ed. McGraw-Hill, 2009.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma Abordagem Profissional*. 8 ed. AMGH, 2016.

RAUFI, B. System Implementation and Adaptation Evaluation in Adaptive Web-based Systems. In: *Proceedings of the 12th International Conference on Computer Systems and Technologies (CompSysTech)*, Vienna, Austria: ACM, p. 286–291, 2011. Disponível em `http://doi.acm.org/10.1145/2023607.2023656`

RODOSEK, G. D.; GEIHS, K.; SCHMECK, H.; BURKHARD, S. Self-Healing Systems: Foundations and Challenges. In: *Self-Healing and Self-Adaptive Systems*, n. 09201, Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Test Case Prioritization: An Empirical Study. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Oxford, England: IEEE, p. 179–188, 1999.

ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, v. 27, n. 10, p. 929–948, 2001.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st ed. Wiley Publishing, 2012.

DE S. CAMPOS JUNIOR, H.; ARAÚJO, M. A. P.; DAVID, J. M. N.; BRAGA, R.; CAMPOS, F.; STRÖELE, V. Test Case Prioritization: A Systematic Review and Mapping of the Literature. In: *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES)*, Fortaleza, CE, Brazil: ACM, p. 34–43, 2017.
Disponível em `http://doi.acm.org/10.1145/3131151.3131170`

SALEHIE, M.; TAHVILDARI, L. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, v. 4, n. 2, 2009.
Disponível em `https://doi.org/10.1145/1516533.1516538`

SHKLAR, L.; ROSEN, R. *Web Application Architecture - principles, protocols and practices*. John Wiley & Sons Ltd, 2003.

STEINBERG, D. H. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. In: *XP Universe*, Citeseer, 2001.

SUN, C.; ZHAO, Y.; PAN, L.; LIU, H.; CHEN, T. Y. Automated Testing of WS-BPEL Service Compositions: A Scenario-Oriented Approach. *IEEE Transactions on Services Computing*, v. 11, n. 4, p. 616–629, 2018.

TENGERI, D.; HORVÁTH, F.; BESZÉDES, Á.; GERGELY, T.; GYIMÓTHY, T. Negative Effects of Bytecode Instrumentation on Java Source Code Coverage. In: *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan: IEEE, p. 225–235, 2016.

WARREN, I. *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. Practitioner Series. Springer London, 1999.

WAVRESKY, F.; LEE, S.-W. A Methodology towards the Adaptization of Legacy Systems Using Agent-Oriented Software Engineering. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, Pisa, Italy: Association for Computing Machinery, p. 1407–1414, 2016.

WEYNS, D.; CALINESCU, R. Tele Assistance: A Self-Adaptive Service-Based System Examplar. In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Florence, Italy: IEEE Press, p. 88–92, 2015.

WOHLIN, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, London, UK: ACM, p. 1–10, 2014.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

YIN, R. K. *Case study research and applications: Design and research*. 6 ed. Sage, 2018.

YOO, S.; HARMAN, M. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability*, v. 22, n. 2, p. 67–120, 2012. Disponível em `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430`

ZHANG, J.; CHENG, B. H. C. Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation. In: *Proceedings of the International Workshop on Modeling in Software Engineering (MISE)*, Minneapolis, MN, USA: IEEE, p. 10–10, 2007.

Anexo A

Resultados e Análise Detalhada dos Testes

As informações das métricas de cobertura de código fornecidas pela biblioteca JaCoCo (apresentada na Seção 3.6) serão utilizadas para a analisar a alteração da abrangência dos subconjuntos de casos de teste destinados às funcionalidades que receberam a introdução de comportamentos adaptativos. Os resultados de cobertura de cada versão do SIGA (ilustradas pela Figura 4.1) serão apresentados e discutidos.

As tabelas apresentadas nesta seção possuem cabeçalhos abreviados para permitir que todas as métricas coletadas sejam apresentadas de forma concentrada. As abreviações nestas tabelas podem ser compreendidas da seguinte forma:

- **I** (Inatingido): refere-se à porção de código que não foi coberta;
- **C** (Coberto):, refere-se à porção de código que foi coberta;
- **T** (Total): refere-se à porção total de código que pode ser coberta, sendo igual à soma do valor inatingido e do valor coberto; e
- **P** (Percentual): refere-se à fração por cento do valor coberto em relação ao valor total.

Em relação às classes apresentadas nas tabelas desta seção, as classes de entidade já existentes desde a versão legada não são listadas, tendo em vista que estas, em sua maioria, possuem apenas construtores com instruções de atribuição e métodos de obtenção e atribuição (conhecidos, respectivamente, como métodos *getters* e *setters* na linguagem de programação Java). Comumente, estes tipos de métodos não são alvos de testes por serem triviais. Além disso, essas classes não foram alteradas nas versões adaptativas do SIGA e, consequentemente, apresentam o mesmo resultado e não requerem análise. Assim sendo, para elucidar o conjunto de classes envolvidas na implementação, apenas as classes de entidade que foram criadas para implementar os

cenários detalhados no capítulo anterior são apresentadas nas tabelas de cobertura de código desta seção.

Um resumo das coberturas obtidas nas diversas versões consideradas neste trabalho é apresentado na Seção 5.2. Ressalta-se também que os pacotes nos quais cada classe está contida são apresentados no Anexo B.

Autenticação (Versão Legada):

A Tabela A.2 apresenta as métricas de cobertura de código que foram coletadas das classes envolvidas no processamento da autenticação da versão legada, isto é, a versão não adaptativa do SIGA. A implementação do subconjunto de teste, composto por casos de teste em nível de sistema (apresentados na Tabela 4.1), e de unidade (parcialmente apresentados na Tabela 4.2), teve êxito ao cumprir o critério de satisfação antes que algum comportamento adaptativo fosse introduzido à funcionalidade (passo 6 da Figura 3.1). O critério estabelecido esperava que ao menos metade das instruções e dos desvios das classes mais essenciais fossem exercitados. As classes `AutenticacaoSaguiApi` (inclusive sua classe interna, `CredendiaisSaguiApi`, que possui apenas um construtor e dois métodos de obtenção), `LoginBean`, `LoginServiceImpl` e `RestfulClientImpl` são consideradas essenciais pois executam o fluxo de envio das credenciais para o processamento da autenticação. Todas as instruções e os desvios (quando aplicável) das classes `AutenticacaoSaguiApi` e `LoginServiceImpl` foram exercitados, enquanto que nas classes `LoginBean` e `RestfulClientImpl` três quartos das instruções foram cobertas. Com relação aos desvios, metade foi coberto em `LoginBean` e todos foram cobertos em `RestfulClientImpl`. A complexidade ciclomática destas duas últimas classes, representado pelo valor total da métrica de “Complexidade”, é um fator preponderante para obter-se valores elevados de cobertura.

A Tabela A.1 abrevia as informações sobre o percentual de cobertura de instruções e de desvios das classes consideradas essenciais.

Tabela A.1: Percentual de cobertura de instruções e de desvios para a classes essenciais da funcionalidade de autenticação - versão legada

Classe	Percentual de Cobertura de Instruções	Percentual de Cobertura de Desvios
AutenticacaoSaguiApi	100%	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	67%	n/a
LoginBean	76%	50%
LoginServiceImpl	100%	n/a
RestfulClientImpl	76%	100%

Tabela A.2: Cobertura de código para a autenticação – versão legada

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AtributoSistemaDAO	93	38	131	29%	10	0	10	0%	9	2	11	18%	18	8	26	31%	4	2	6	33%	0	1	1	100%
AtributoSistemaServiceImpl	21	11	32	34%	n/a	n/a	n/a	n/a	4	3	7	43%	4	3	7	43%	4	3	7	43%	0	1	1	100%
AutenticacaoException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoSaguiApi	0	100	100	100%	0	4	4	100%	0	4	4	100%	0	17	17	100%	0	2	2	100%	0	1	1	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	6	12	18	67%	n/a	n/a	n/a	n/a	2	1	3	33%	2	4	6	67%	2	1	3	33%	0	1	1	100%
AutenticacaoSaguiApiException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AuthenticationBeanImpl	0	37	37	100%	1	3	4	75%	1	3	4	75%	0	9	9	100%	0	2	2	100%	0	1	1	100%
AuthenticationFilter	0	12	12	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
LoginBean	25	78	103	76%	2	2	4	50%	4	7	11	64%	10	22	32	69%	2	7	9	78%	0	1	1	100%
LoginServiceImpl	0	9	9	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
RestfulClientImpl	45	143	188	76%	0	2	2	100%	2	10	12	83%	10	33	43	77%	2	9	11	82%	0	1	1	100%

Em relação às classes subjacentes às classes essenciais, os percentuais das métricas de instruções e desvios das classes para o tratamento de exceções na execução da autenticação (`AutenticacaoException` e `AutenticacaoSaguiApiException`) não atingiram o valor para satisfazer o critério estipulado para a cobertura de código. Esses valores poderiam ser plenos se os construtores dessas classes, contabilizados como métodos pela JaCoCo, que foram declarados no código fonte mas não são utilizados em outras partes do projeto, não existissem. Os valores de cobertura de classes como `AtributoSistemaDAO` e `AtributoSistemaServiceImpl` ficaram abaixo dos cinquenta porcento. Isto era esperado, dado que estas classes atuam apenas na obtenção de um parâmetro do sistema que especifica a URI do serviço *web* do SAGUI que deve ser consumida para executar a autenticação. As classes `AuthenticationFilter` e `AuthenticationBeanImpl` também alcançaram valores expressivos de cobertura, embora a primeira classe atue apenas como um filtro de requisições e a segunda execute a lógica de controle desse filtro antes de processar a requisição.

Sob uma perspectiva geral, 69% das instruções e 46% dos desvios das classes envolvidas na funcionalidade de autenticação foram cobertos. Quando analisadas apenas as classes consideradas essenciais, foram cobertas 82% das instruções e 80% dos desvios.

Autenticação Autocurável com 1 Mecanismo Secundário:

A versão $SIGA\_Adapt\_1$ introduz unidades de código para implementar a "Autenticação Autocurável" com um mecanismo secundário. Em relação à versão antecessora, legada, quatorze novas classes foram criadas e outras duas passaram a ser utilizadas no processo de autocura ou na execução da autenticação. As métricas de cobertura de código das classes envolvidas no processamento da autenticação e no processo de adaptação desta versão estão apresentadas na Tabela A.3.

Com relação às classes consideradas essenciais para a execução da funcionalidade, tanto a classe responsável pelo processamento da autenticação utilizando o mecanismo principal (`AutenticacaoSaguiApi`) quanto a classe de controle de requisição (`LoginBean`) não sofreram alterações em suas implementações e os valores das métricas de cobertura permaneceram inalterados. No que diz respeito à classe `LoginServiceImpl`, embora tenha havido um simples incremento no número total de instruções, os valores percentuais de cobertura não variaram também. Em contrapartida, os percentuais de instruções e desvios cobertos da classe `RestfulClientImpl` foram decrementados em função de uma alteração para tornar funcional o componente de monitoramento do mecanismo de autenticação principal, relativo ao modelo MAPE-K e representado pela classe `MonitorAutenticacaoSaguiApi`.

Tabela A.3: Cobertura de código para a autenticação autocurável com 1 mecanismo secundário – versão $SIGA\_Adapt\_1$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AnalyserAutenticacao	5	50	55	91%	5	3	8	38%	4	5	9	56%	1	14	15	93%	0	5	5	100%	0	1	1	100%
AtributoSistemaDAO	76	74	150	49%	8	2	10	20%	7	5	12	42%	14	17	31	55%	2	5	7	71%	0	1	1	100%
AtributoSistemaServiceImpl	16	22	38	58%	n/a	n/a	n/a	n/a	3	5	8	63%	3	6	9	67%	3	5	8	63%	0	1	1	100%
AutenticacaoException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoFactory	16	24	40	60%	2	1	3	33%	2	2	4	50%	4	8	12	67%	0	2	2	100%	0	1	1	100%
AutenticacaoLdap	18	13	31	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	4	10	40%	0	2	2	100%	0	1	1	100%
AutenticacaoLdapException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoSaguiApi	0	100	100	100%	0	4	4	100%	0	4	4	100%	0	17	17	100%	0	2	2	100%	0	1	1	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	6	12	18	67%	n/a	n/a	n/a	n/a	2	1	3	33%	2	4	6	67%	2	1	3	33%	0	1	1	100%
AutenticacaoSaguiApiException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AuthenticationBeanImpl	0	37	37	100%	1	3	4	75%	1	3	4	75%	0	9	9	100%	0	2	2	100%	0	1	1	100%
AuthenticationFilter *	0	12	12	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
KnowledgeAutenticacao	52	0	52	0%	n/a	n/a	n/a	n/a	12	0	12	0%	22	0	22	0%	12	0	12	0%	1	0	1	0%
KnowledgeAutenticacaoDAO	0	22	22	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	5	5	100%	0	2	2	100%	0	1	1	100%
KnowledgeAutenticacaoServiceImpl	0	18	18	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	5	5	100%	0	3	3	100%	0	1	1	100%
LoginBean	25	78	103	76%	2	2	4	50%	4	7	11	64%	10	22	32	69%	2	7	9	78%	0	1	1	100%
LoginServiceImpl	0	10	10	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
MecanismoAutenticacao	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MecanismoStatus	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MonitorAutenticacao	0	3	3	100%	n/a	n/a	n/a	n/a	0	1	1	100%	0	1	1	100%	0	1	1	100%	0	1	1	100%
MonitorAutenticacaoLdap	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoSaguiApi	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%

Continuação da Tabela A.3 : Cobertura de código para a autenticação autocurável com 1 mecanismo secundário – versão $SIGA\_Adapt\_1$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
RestfulClientImpl	60	132	192	69%	2	0	2	0%	4	9	13	69%	13	32	45	71%	3	9	12	75%	0	1	1	100%
SigaDAO	40	9	49	18%	4	0	4	0%	6	2	8	25%	12	3	15	20%	4	2	6	33%	0	1	1	100%
SigaServiceImpl	15	8	23	35%	n/a	n/a	n/a	n/a	3	2	5	40%	5	2	7	29%	3	2	5	40%	0	1	1	100%
TrocadorMecanismoAutenticacao	0	24	24	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	5	5	100%	0	2	2	100%	0	1	1	100%
TrocaMecanismoAutenticacaoEvent	0	9	9	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	4	4	100%	0	2	2	100%	0	1	1	100%

Sobre a classe que executa a submissão das credenciais para o processamento da autenticação usando o mecanismo secundário, isto é, a classe `AutenticacaoLdap`, os valores percentuais de instruções atingiram números significantes. Essa classe foi introduzida nesta versão do SIGA para efetuar a autenticação via LDAP e foi exercitada quase que em metade de suas instruções. O exercício desta classe foi operado pelo componente de monitoramento do mecanismo, representado pela classe `MonitorAutenticacaoLdap`, durante a instrumentação dos testes de sistema.

As classes supracitadas e que representam os componentes de monitoramento dos mecanismos de autenticação desta versão do SIGA, isto é, `MonitorAutenticacaoLdap` e `MonitorAutenticacaoSaguiApi`, foram exercitadas com teste em nível de sistema, onde os respectivos temporizadores inicializaram suas execuções. Por terem implementações semelhantes, os valores de cobertura de código de ambas foram idênticos. Metade das linhas de ambas as classes foram cobertas e pouco mais de dois quintos das instruções foram exercitadas.

Pertinente às classes de monitoramento, `MonitorAutenticacao` é uma classe abstrata que possui apenas atributos utilizados por suas herdeiras. Assim, os valores de cobertura desta classe abstrata foram absolutos. O mesmo percentual de cobertura foi atingido pelas classes utilizadas na manipulação de informações relativas ao elemento de conhecimento (`KnowledgeAutenticacaoDAO` e `KnowledgeAutenticacaoServiceImpl`) e pela classes empregadas na adaptação do sistema (`TrocaMecanismoAutenticacaoEvent` e `TrocadorMecanismoAutenticacao`), tendo em vista que seus métodos não possuem desvios e foram propostos exclusivamente para a introdução do comportamento adaptativo.

A classe `AnalyserAutenticacao`, que representa o componente de análise referente ao modelo MAPE-K, também foi exercitada com teste em nível de sistema, onde seu temporizador inicializou sua execução. A cobertura de linhas foi quase plena, refletindo no alto valor percentual de cobertura de instruções. Entretanto, em virtude da diminuta quantidade de exercícios das possibilidades de saída das declarações condicionais, o percentual de desvios aproximou-se apenas de dois quintos de cobertura.

A classe `AutenticacaoFactory` é uma classe que também foi criada com a introdução do segundo mecanismo de autenticação. Sua função é produzir objetos de acordo com o valor da configuração do sistema que indica qual mecanismo deve ser utilizado para validar as credenciais enviadas na requisição. Como o subconjunto de casos de teste criado na versão legada foi direcionado ao mecanismo de autenticação principal, um dos desvios da classe foi coberto nesta versão. Os outros dois desvios (um relacionado à execução da autenticação via o mecanismo secundário e outro

para lidar com o valor inválido de configuração do sistema), naturalmente, não foram cobertos.

No tocante as demais classes da versão $SIGA\_Adapt\_1$, `AuthenticationFilter` e `AuthenticationBeanImpl` apresentaram os mesmos valores para todas as métricas em relação à versão legada, dado que suas implementações também não sofreram modificações. A mesma análise é válida para as classes destinadas ao tratamento de exceções que já existiam na versão legada (`AutenticacaoException` e `AutenticacaoSaguiApiException`). Ressalta-se ainda que a cobertura da classe criada nesta versão para o tratamento de exceções que eventualmente possam ser lançadas durante o processamento da autenticação via LDAP (`AutenticacaoLdapException`) foi igual à cobertura das classes de mesma responsabilidade. Por outro lado, embora a complexidade das classes `AtributoSistemaServiceImpl` e `AtributoSistemaDAO` tenha aumentado por conta do acréscimo de instruções, os respectivos valores das métricas de cobertura elevaram-se devido aos exercícios de código impostos pelos componentes de análise e de execução do processo adaptação, que fazem uso destas classes para obter e atualizar o valor da configuração de sistema.

De modo geral, 66% das instruções e 38% dos desvios das classes envolvidas na "Autenticaçao Autocurável" foram cobertos. Os mesmos valores de cobertura foram atingidos quando analisadas apenas as classes inerentes ao processo de adaptação - referentes ao modelo MAPE-K. Ademais, conclui-se que quatro das onze classes originalmente envolvidas na funcionalidade de autenticação foram impactadas com a introdução da característica de autocura.

Autenticação Autocurável com 2 Mecanismos Secundários:

Para a versão $SIGA\_Adapt\_2$, que acrescenta a autenticação via SGBD como um outro mecanismo secundário, oito novas classes foram acrescidas, em relação a versão antecessora ($SIGA\_Adapt\_1$), para implementar a "Autenticação Autocurável" com dois mecanismos secundários. A Tabela A.4 apresenta as métricas de cobertura de código para as classes vinculadas ao processamento da autenticação e ao processo adaptativo desta versão do SIGA.

Em relação às classes consideradas essenciais, as métricas de cobertura das classes desta versão, utilizadas no processamento da autenticação via o mecanismo principal (serviço *web*), permaneceram inalteradas quando comparadas à métricas da versão $SIGA\_Adapt\_1$.

Tabela A.4: Cobertura de código para a autenticação autocurável com 2 mecanismos secundários – versão $SIGA\_Adapt\_2$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AnalyserAutenticacao	15	64	79	81%	8	4	12	33%	6	6	12	50%	4	19	23	83%	0	6	6	100%	0	1	1	100%
AtributoSistemaDAO	76	74	150	49%	8	2	10	20%	7	5	12	42%	14	20	34	59%	2	5	7	71%	0	1	1	100%
AtributoSistemaServiceImpl	16	22	38	58%	n/a	n/a	n/a	n/a	3	5	8	63%	3	6	9	67%	3	5	8	63%	0	1	1	100%
AutenticacaoDB	7	13	20	65%	n/a	n/a	n/a	n/a	0	2	2	100%	2	4	6	67%	0	2	2	100%	0	1	1	100%
AutenticacaoDBException	9	0	9	0%	n/a	n/a	n/a	n/a	2	0	2	0%	4	0	4	0%	2	0	2	0%	1	0	1	0%
AutenticacaoException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoFactory	23	24	47	51%	3	1	4	25%	3	2	5	40%	6	8	14	57%	0	2	2	100%	0	1	1	100%
AutenticacaoLdap	18	13	31	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	4	10	40%	0	2	2	100%	0	1	1	100%
AutenticacaoLdapException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoSaguiApi	0	100	100	100%	0	4	4	100%	0	4	4	100%	0	17	17	100%	0	2	2	100%	0	1	1	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	6	12	18	67%	n/a	n/a	n/a	n/a	2	1	3	33%	2	4	6	67%	2	1	3	33%	0	1	1	100%
AutenticacaoSaguiApiException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AuthenticationBeanImpl	0	37	37	100%	1	3	4	75%	1	3	4	75%	0	9	9	100%	0	2	2	100%	0	1	1	100%
AuthenticationFilter *	0	12	12	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
BCrypt	115	5996	6111	98%	35	59	94	63%	34	31	65	48%	28	171	199	86%	2	16	18	89%	0	1	1	100%
BCryptPasswordEncoder	54	40	94	43%	14	2	16	13%	9	4	13	31%	11	15	26	58%	1	4	5	80%	0	1	1	100%
CriptografadorSagui	0	8	8	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
KnowledgeAutenticacao	52	0	52	0%	n/a	n/a	n/a	n/a	12	0	12	0%	22	0	22	0%	12	0	12	0%	1	0	1	0%
KnowledgeAutenticacaoDAO	18	22	40	55%	n/a	n/a	n/a	n/a	1	2	3	67%	4	5	9	56%	1	2	3	67%	0	1	1	100%
KnowledgeAutenticacaoServiceImpl	50	18	68	26%	6	0	6	0%	6	3	9	33%	12	5	17	29%	3	3	6	50%	0	1	1	100%
LoginBean	25	78	103	76%	2	2	4	50%	4	7	11	64%	10	22	32	69%	2	7	9	78%	0	1	1	100%
LoginServiceImpl	0	10	10	100%	n/a	na/	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%

Continuação da Tabela A.4 : Cobertura de código para a autenticação autocurável com 2 mecanismos secundários – versão $SIGA\_Adapt\_2$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
MecanismoAutenticacao	0	48	48	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	8	8	100%	0	3	3	100%	0	1	1	100%
MecanismoStatus	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MonitorAutenticacao	0	3	3	100%	n/a	n/a	n/a	n/a	0	1	1	100%	0	1	1	100%	0	1	1	100%	0	1	1	100%
MonitorAutenticacaoDB	34	25	59	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	8	14	57%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoLdap	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoSaguiApi	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%
RestfulClientImpl	60	132	192	69%	2	0	2	0%	4	9	13	69%	13	32	45	71%	3	9	12	75%	0	1	1	100%
SigaDAO	40	9	49	18%	4	0	4	0%	6	2	8	25%	12	3	15	20%	4	2	6	33%	0	1	1	100%
SigaServiceImpl	15	8	23	35%	n/a	n/a	n/a	n/a	3	2	5	40%	5	2	7	29%	3	2	5	40%	0	1	1	100%
TrocadorMecanismoAutenticacao	0	24	24	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	5	5	100%	0	2	2	100%	0	1	1	100%
TrocaMecanismoAutenticacaoEvent	0	9	9	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	4	4	100%	0	2	2	100%	0	1	1	100%
UsuarioDAO	4	19	23	83%	n/a	n/a	n/a	n/a	0	2	2	100%	2	4	6	67%	0	2	2	100%	0	1	1	100%
UsuarioServiceImpl	7	24	31	77%	n/a	n/a	n/a	n/a	0	3	3	100%	3	5	8	63%	0	3	3	100%	0	1	1	100%

Algumas classes foram impactadas com a introdução de um segundo mecanismo secundário de autenticação. Contudo, todas são classes que foram criadas na versão $SIGA\_Adapt\_1$, ou seja, não existiam na versão legada. A classe `AutenticacaoFactory` incorporou mais uma instrução condicional para tratar a instanciação de objetos quando o valor da configuração de sistema indicar que o SGBD deverá ser utilizado como mecanismo de autenticação. Desta forma, houve uma degradação na cobertura do seu código, tanto no tocante às instruções quanto aos desvios, à complexidade e às linhas.

Também foram degradados os valores das métricas de cobertura das classes utilizadas para manipular dados relativos ao elemento de conhecimento (`KnowledgeAutenticacaoDAO` e `KnowledgeAutenticacaoServiceImpl`), uma vez que os métodos criados nesta versão não foram invocados durante a execução do teste de sistema, e da classe que representa o componente de análise (`AnalyserAutenticacao`).

Dentre as classes que foram inseridas nesta versão, a cobertura da classe `AutenticacaoDB`, responsável pela autenticação via SGBD, atingiu números expressivos, embora seja um classe com poucas linhas. Por meio do teste em nível de sistema, esta classe foi exercitada por sua respectiva classe de monitoramento, `MonitorAutenticacaoDB`, que por sua vez teve sua execução inicializada por seu temporizador. No que se refere aos valores das métricas de cobertura, a classe de monitoramento do mecanismo de autenticação via SGBD equiparou-se às outras classes de monitoramento de mecanismos de autenticação (`MonitorAutenticacaoLdap` e `MonitorAutenticacaoSaguiApi`).

Outras classes que foram exercitadas por meio do teste de sistema em função da execução da classe `MonitorAutenticacaoDB`, inicializada por intermédio de seu temporizador, foram as classes pertinentes ao fluxo de validação das credenciais via SGBD. As classes `UsuarioDAO` e `UsuarioServiceImpl` foram cobertas de forma significativa, enquanto que quase metade das instruções do código da classe `Bcrypt` foi coberta. Em termos de instruções, a cobertura da classe `BCryptPasswordEncoder` foi quase plena, mesmo que pouco mais da metade dos desvios tenham sido exercitados. Por fim, a classe `CriptografadorSagui` foi plenamente coberta, uma vez que esta é uma classe que apenas delega uma chamada para a interface que define os comportamentos concretizados pela classe `BCryptPasswordEncoder`.

Quando comparada com a cobertura de código da versão antecessora, a cobertura das classes consideradas subjacentes à funcionalidade de autenticação permaneceu inalterada. Além do mais, também foram preservados os valores das métricas da classe introduzida para a validação das credenciais via LDAP (`AutenticacaoLdap`).

De forma sucinta, cinco classes foram impactadas com a inserção de um segundo mecanismo secundário de autenticação e a cobertura total de instruções e desvios atingiu, respectivamente, 90% e 48%. Salienta-se ainda que a elevação no percentuais de cobertura total das classes desta versão (apresentadas na Tabela A.4) quando comparada com o percentual de cobertura total das classes da versão antecessora (denotadas na Tabela A.3), deve-se ao fato da classe `BCrypt` possuir 80% do total de instruções e 59% de todos os desvios. Ao desconsiderar essa classe para o cálculo do valor total de cobertura das classes envolvidas na "Autenticação Autocurável", 61% das instruções e 27% dos desvios foram cobertos. Quando analisadas apenas as classes inerentes ao processo de adaptação (aquelas que representam algum item do modelo MAPE-K), a cobertura total de instruções e desvios atingiu, respectivamente, 55% e 22%.

Autenticação Autocurável com 1 Mecanismo Secundário e Apoio do *Framework* StarMX:

A versão $SIGA\_Adapt\_StarMX\_1$, a primeira a implementar a "Autenticação Autocurável" com o apoio do *framework* StarMX, insere dezesseis novas classes e utiliza outras duas já existentes para a execução da autenticação e no processo de autocura com um mecanismo secundário. A Tabela A.5 apresenta as métricas de cobertura das classes envolvidas nestes processos. Algumas classes adicionadas a esta versão são exatamente as mesmas que foram criadas na versão $SIGA\_Adapt\_1$. Logo, a descrição delas será abreviada na análise desta versão.

Em relação às classes consideradas essenciais para o processamento da autenticação, `AutenticacaoSaguiApi` foi a única classe que não teve alterações nos valores das métricas de cobertura. Assim como na versão $SIGA\_Adapt\_1$, a classe `LoginServiceImpl` não teve seus percentuais de cobertura alterados, contudo sofreu um simples acréscimo no número total de suas instruções em consequência da utilização da classe `AutenticacaoFactory`, que instancia objetos de acordo com o valor da configuração de sistema que indica qual deve ser o mecanismo utilizado para validar as credenciais.

Embora não tenha sofrido modificações, a cobertura da classe `LoginBean` aumentou, quando comparada com a cobertura da versão legada, em virtude do aumento no exercício de um de seus métodos com a execução manual de um teste de nível de sistema. Por outro lado, a classe `RestfulClientImpl` foi modificada para tornar funcional o componente de monitoramento do mecanismo de autenticação principal (`MonitorAutenticacaoSaguiApi`), mesma razão da modificação realizada na versão $SIGA\_Adapt\_1$. Desta forma, os valores da métricas de cobertura da classe `RestfulClientImpl` nesta versão são idênticos aos da versão $SIGA\_Adapt\_1$ e, por consequência, tem a mesma degradação em relação a versão legada.

Tabela A.5: Cobertura de testes para a autenticação autocurável com 1 mecanismo secundário e apoio do *framework* StarMX - versão $SIGA\_Adapt\_StarMX\_1$

Classe	Instruções				Desvios				Complexidade				Linhas				Methods				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AnalyserAutenticacaoJMX	13	39	52	75%	5	3	8	38%	5	4	9	44%	4	11	15	73%	1	4	5	80%	0	1	1	100%
AnalyserAutenticacaoProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
AtributoSistemaDAO	95	55	150	37%	8	2	10	20%	8	4	12	33%	20	12	32	38%	3	4	7	57%	0	1	1	100%
AtributoSistemaServiceImpl	22	16	38	42%	n/a	n/a	n/a	n/a	4	4	8	50%	5	4	9	44%	4	4	8	50%	0	1	1	100%
AutenticacaoException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoFactory	12	23	35	66%	2	1	3	33%	2	2	4	50%	3	6	9	67%	0	2	2	100%	0	1	1	100%
AutenticacaoLdap	18	13	31	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	4	10	40%	0	2	2	100%	0	1	1	100%
AutenticacaoLdapException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoSaguiApi	0	100	100	100%	0	4	4	100%	0	4	4	100%	0	17	17	100%	0	2	2	100%	0	1	1	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	6	12	18	67%	n/a	n/a	n/a	n/a	2	1	3	33%	2	4	6	67%	2	1	3	33%	0	1	1	100%
AutenticacaoSaguiApiException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AuthenticationBeanImpl	0	37	37	100%	1	3	4	75%	1	3	4	75%	0	9	9	100%	0	2	2	100%	0	1	1	100%
AuthenticationFilter *	0	12	12	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
KnowledgeAutenticacao	0	52	52	100%	n/a	n/a	n/a	n/a	12	0	12	0%	22	0	22	0%	12	0	12	0%	1	0	1	0%
KnowledgeAutenticacaoDAO	0	22	22	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	5	5	100%	0	2	2	100%	0	1	1	100%
KnowledgeAutenticacaoServiceImpl	0	18	18	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	5	5	100%	0	3	3	100%	0	1	1	100%
LoginBean	17	86	103	83%	1	3	4	75%	2	9	11	82%	7	25	32	78%	1	8	9	89%	0	1	1	100%
LoginServiceImpl	0	10	10	100%	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
MecanismoAutenticacao	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MecanismoStatus	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MonitorAutenticacao	0	3	3	100%	n/a	n/a	n/a	n/a	0	1	1	100%	0	1	1	100%	0	1	1	100%	0	1	1	100%

Continuação da Tabela A.5 : Cobertura de código para a autenticação autocurável com 1 mecanismo secundário e apoio do *framework* StarMX

Classe	Instruções				Desvios				Complexidade				Linhas				Methods				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
MonitorAutenticacaoLdapJMX	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoLdapProcess	4	5	9	56%	n/a	n/a	n/a	n/a	1	3	4	75%	2	3	5	60%	1	3	4	75%	0	1	1	100%
MonitorAutenticacaoSaguiApiJMX	29	30	59	51%	n/a	n/a	n/a	n/a	0	2	2	100%	6	8	14	57%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoSaguiApiProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
RestfulClientImpl	60	132	192	69%	2	0	2	0%	4	9	13	69%	13	32	45	71%	3	9	12	75%	0	1	1	100%
SigaDAO	40	9	49	18%	4	0	4	0%	6	2	8	25%	12	3	15	20%	4	2	6	33%	0	1	1	100%
SigaServiceImpl	15	8	23	35%	n/a	n/a	n/a	n/a	3	2	5	40%	5	2	7	29%	3	2	5	40%	0	1	1	100%
TrocadorMecanismoAutenticacao	18	3	21	14%	n/a	n/a	n/a	n/a	1	1	2	50%	3	1	4	25%	1	1	2	50%	0	1	1	100%

No que se refere às classes que foram adicionadas a esta versão, destacam-se as classes que representam os componentes básicos relativos ao modelo MAPE-K. As classes `AnalyserAutenticacaoProcess`, `MonitorAutenticacaoLdapProcess` e `MonitorAutenticacaoSaguiApiProcess` são àquelas que implementam a lógica de gerenciamento do StarMX. Estas são acionadas por seus respectivos temporizadores de acordo com as configurações do *framework*. Assim, os valores de cobertura destas classes foram atingidos através da instrumentação de código com testes em nível de sistema. Ademais, a simples responsabilidade destas classes em atuar sob as interfaces de instrumentação da tecnologia JMX proporcionou-lhes valores de cobertura expressivos.

Os valores de cobertura das classes que implementam as interfaces de instrumentação, tanto das classes que representam o componente de monitoramento (`MonitorAutenticacaoLdapJMX` e `MonitorAutenticacaoSaguiApiJMX`) quanto da classe que representa o componente de análise (`AnalyserAutenticacaoJMX`), são resultado da atuação das classes que implementam a lógica de gerenciamento do *framework*. Especificamente, pouco mais da metade das instruções e linhas da classe `MonitorAutenticacaoSaguiApiJMX` foram cobertas. Em contrapartida, menos da metade das instruções da classe `MonitorAutenticacaoLdapJMX` foram cobertas, embora metade das linhas tenham sido exercitadas. Quando estas duas classes são comparadas às respectivas classes de monitoramento da versão $SIGA\_Adapt\_1$, seus valores são equiparáveis.

Ainda que tenha a mesma quantidade de desvios, número de linhas e métodos, a classe `AnalyserAutenticacaoJMX` não foi tão exercitada em comparação com sua respectiva implementação da versão $SIGA\_Adapt\_1$. Mesmo assim, três quartos das instruções foram cobertas.

As demais classes que foram adicionadas nesta versão são as mesmas que foram implementadas na versão $SIGA\_Adapt\_1$. A título de conhecimento, podem ser citadas a classe que executa a submissão das credenciais para validação via LDAP (`AutenticacaoLdap`), assim como a classe para o tratamento de exceções que ocorram durante essa submissão (`AutentcacaoLdapException`), a classe abstrata que concentra atributos utilizados pelas classe de monitoramento (`MonitorAutenticacao`) e as classes relacionadas ao fluxo de dados relativos ao elemento de conhecimento (`KnowledgeAutenticacaoDAO` e `KnowledgeAutenticacaoServiceImpl`). Desta forma, o mesmo diagnóstico de comparação entre versão $SIGA\_Adapt\_StarMX\_1$ e a versão legada é aplicada na análise destas classes. Ressalta-se ainda que a classe `AutenticacaoFactory` desta versão foi implementada de forma semelhante à respectiva classe da versão $SIGA\_Adapt\_1$, embora o número de linhas e instruções sejam diferentes.

Com relação às classes subjacentes ao processamento da autenticação, tanto as utilizadas na interceptação de requisições HTTP (`AuthenticationFilter` e `AuthenticationBeanImpl`) quanto àquelas destinadas ao tratamento de exceções (`AutenticacaoException` e `AutenticacaoSaguiApiException`) atingiram os mesmos valores de cobertura atingidos na versão legada. Em contrapartida, ainda que a quantidade de instruções das classes `AtributoSistemaDAO` e `AtributoSistemaServiceImpl` tenham sido incrementadas e, consequentemente, elevado a complexidade total, a cobertura destas classes foi elevada pelo mesmo motivo apresentado na análise da versão $SIGA\_Adapt\_1$.

Quanto à cobertura total das classes envolvidas "Autenticação Autocurável" com um mecanismo secundário e apoio do *framework* StarMX, 63% das instruções e 41% dos desvios foram cobertos. Se analisadas apenas as classes inerentes ao processo de adaptação (aquelas relativas ao modelo MAPE-K), a cobertura total foi de 61% das instruções e 38% dos desvios.

Autenticação Autocurável com 2 Mecanismos Secundários e Apoio do *Framework* StarMX:

Para implementar a "Autenticação Autocurável" com dois mecanismos secundários e com o apoio do *framework* StarMX, nove classes, inerentes à validação de credenciais e ao monitoramento da autenticação via SGBD, foram adicionadas. As classes envolvidas na implementação desta versão podem ser visualizadas na Tabela A.6. Algumas classes adicionadas a esta versão, $SIGA\_Adapt\_StarMX\_2$, são exatamente as mesmas que foram criadas na versão $SIGA\_Adapt\_2$. Assim, a descrição e análise dessas classes será abreviada.

Em relação às classes consideradas essenciais, as métricas de cobertura das classes desta versão, utilizadas no processamento da autenticação via o mecanismo principal (serviço web), permaneceram inalteradas quando comparadas à métricas da versão $SIGA\_Adapt\_StarMX\_1$.

Assim como na versão $SIGA\_Adapt\_2$, as classes impactadas com a introdução do segundo mecanismo secundário de autenticação foram aquelas que foram criadas na versão antecessora (neste caso, na versão $SIGA\_Adapt\_StarMX\_1$). Além do mais, as classes `AutenticacaoFactory` e `AnalyserAutenticacaoJMX` sofreram degradação pelo mesmo motivo apresentado na análise da versão $SIGA\_Adapt\_2$. As demais classe não foram impactadas.

Tabela A.6: Cobertura de código para a autenticação autocurável com 2 mecanismos secundários e apoio do *framework* StarMX – versão $SIGA\_Adapt\_StarMX\_2$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AnalyserAutenticacaoJMX	43	33	76	43%	10	2	12	17%	8	4	12	33%	13	10	23	43%	2	4	6	67%	0	1	1	100%
AnalyserAutenticacaoProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
AtributoSistemaDAO	95	55	150	37%	8	2	10	20%	8	4	12	33%	20	12	32	38%	3	4	7	57%	0	1	1	100%
AtributoSistemaServiceImpl	22	16	38	42%	n/a	n/a	n/a	n/a	4	4	8	50%	5	4	9	44%	4	4	8	50%	0	1	1	100%
AutenticacaoDB	7	13	20	65%	n/a	n/a	n/a	n/a	0	2	2	100%	2	4	6	67%	0	2	2	100%	0	1	1	100%
AutenticacaoDBException	9	0	9	0%	n/a	n/a	n/a	n/a	2	0	2	0%	4	0	4	0%	2	0	2	0%	1	0	1	0%
AutenticacaoException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoFactory	23	24	47	51%	3	1	4	25%	3	2	5	40%	6	8	14	57%	0	2	2	100%	0	1	1	100%
AutenticacaoLdap	18	13	31	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	4	10	40%	0	2	2	100%	0	1	1	100%
AutenticacaoLdapException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AutenticacaoSaguiApi	0	100	100	100%	0	4	4	100%	0	4	4	100%	0	17	17	100%	0	2	2	100%	0	1	1	100%
AutenticacaoSaguiApi.CredendiaisSaguiApi	6	12	18	67%	n/a	n/a	n/a	n/a	2	1	3	33%	2	4	6	67%	2	1	3	33%	0	1	1	100%
AutenticacaoSaguiApiException	5	4	9	44%	n/a	n/a	n/a	n/a	1	1	2	50%	2	2	4	50%	1	1	2	50%	0	1	1	100%
AuthenticationBeanImpl	0	37	37	100%	1	3	4	75%	1	3	4	75%	0	9	9	100%	0	2	2	100%	0	1	1	100%
AuthenticationFilter *	0	12	12	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
BCrypt	115	5996	6111	98%	35	59	94	63%	34	31	65	48%	28	171	199	86%	2	16	18	89%	0	1	1	100%
BCryptPasswordEncoder	54	40	94	43%	14	2	16	13%	9	4	13	31%	11	15	26	58%	1	4	5	80%	0	1	1	100%
CriptografadorSagui	0	8	8	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
KnowledgeAutenticacao	52	0	52	0%	n/a	n/a	n/a	n/a	12	0	12	0%	22	0	22	0%	12	0	12	0%	1	0	1	0%
KnowledgeAutenticacaoDAO	18	22	40	55%	n/a	n/a	n/a	n/a	1	2	3	67%	4	5	9	56%	1	2	3	67%	0	1	1	100%
KnowledgeAutenticacaoServiceImpl	50	18	68	26%	6	0	6	0%	6	3	9	33%	12	5	17	29%	3	3	6	50%	0	1	1	100%

Continuação da Tabela A.6 : Cobertura de código para a autenticação autocurável com 2 mecanismos secundários e apoio do *framework* StarMX

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
LoginBean	17	86	103	83%	1	3	4	75%	2	9	11	82%	7	25	32	78%	1	8	9	89%	0	1	1	100%
LoginServiceImpl	0	10	10	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	2	2	100%	0	2	2	100%	0	1	1	100%
MecanismoAutenticacao	0	48	48	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	8	8	100%	0	3	3	100%	0	1	1	100%
MecanismoStatus	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	7	7	100%	0	3	3	100%	0	1	1	100%
MonitorAutenticacao	0	3	3	100%	n/a	n/a	n/a	n/a	0	1	1	100%	0	1	1	100%	0	1	1	100%	0	1	1	100%
MonitorAutenticacaoDBJMX	34	25	59	42%	n/a	n/a	n/a	n/a	0	2	2	100%	6	8	14	57%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoDBProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
MonitorAutenticacaoLdapJMX	35	24	59	41%	n/a	n/a	n/a	n/a	0	2	2	100%	7	7	14	50%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoLdapProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
MonitorAutenticacaoSaguiApiJMX	29	30	59	51%	n/a	n/a	n/a	n/a	0	2	2	100%	6	8	14	57%	0	2	2	100%	0	1	1	100%
MonitorAutenticacaoSaguiApiProcess	0	9	9	100%	n/a	n/a	n/a	n/a	0	4	4	100%	0	5	5	100%	0	4	4	100%	0	1	1	100%
RestfulClientImpl	60	132	192	69%	2	0	2	0%	4	9	13	69%	13	32	45	71%	3	9	12	75%	0	1	1	100%
SigaDAO	40	9	49	18%	4	0	4	0%	6	2	8	25%	12	3	15	20%	4	2	6	33%	0	1	1	100%
SigaServiceImpl	15	8	23	35%	n/a	n/a	n/a	n/a	3	2	5	40%	5	2	7	29%	3	2	5	40%	0	1	1	100%
TrocadorMecanismoAutenticacao	18	3	21	14%	n/a	n/a	n/a	n/a	1	1	2	50%	3	1	4	25%	1	1	2	50%	0	1	1	100%
UsuarioDAO	4	19	23	83%	n/a	n/a	n/a	n/a	0	2	2	100%	2	4	6	67%	0	2	2	100%	0	1	1	100%
UsuarioServiceImpl	7	17	24	71%	n/a	n/a	n/a	n/a	0	3	3	100%	3	5	8	63%	0	3	3	100%	0	1	1	100%

De forma geral, 90% de todas as instruções e 48% do total de desvios das classes envolvidas envolvidas “Autenticação Autocurável” com dois mecanismos secundários e apoio do *framework* StarMX (apresentadas na Tabela A.6) foram cobertos. Contudo, analogamente à versão $SIGA\_Adapt\_2$, é importante salientar que a classe `Bcrypt` possui 80% das instruções e 59% dos desvios totais destas classes. Desconsiderando-a para o cálculo do valor total de cobertura das classes envolvidas na “Autenticação Autocurável” da versão $SIGA\_Adapt\_StarMX\_2$, 57% das instruções e 26% dos desvios foram cobertos. A cobertura das classes específicas do processo de adaptação (aquelas que representam algum componente básico ou elemento de conhecimento relativo ao modelo MAPE-K) atingiu 50% das instruções e 11% dos desvios.

Processamento de Candidatos à Formatura e Formandos (Versão Legada):

As métricas de cobertura de código que foram coletadas das classes envolvidas no processamento de candidatos à formatura e formandos da versão legada do SIGA são apresentadas na Tabela A.8. O conjunto de casos de teste (níveis de unidade e sistema; Tabelas 4.5 e 4.4, respectivamente), que foi planejado e implementado não atendeu satisfatoriamente ao critério estabelecido para dar início à etapa de implementação do processamento auto-otimizado (passo 6 da Figura 3.1). Esperava-se que as classes mais essenciais da funcionalidade de processamento tivessem ao menos cinquenta por cento das instruções e dos desvios cobertos pelos testes. Os valores destas métricas cobertura para as classes `InformacoesIntegralizacaoVO` e `RotinaCandidatoFormandoDAO` atenderam ao critério de satisfação, enquanto que ao menos um desses valores para as classes `IntegralizacaoServiceImpl` e `RotinaCandidatoServiceImpl` ficou abaixo do esperado. Dentre as classes consideradas essenciais para a funcionalidade de processamento, `RotinaCandidatoFormandoBean` foi a única onde tanto o valor de instruções quanto de desvios não atingiu o mínimo esperado. Essas classes são consideradas essenciais para o processamento de candidatos à formatura e formandos já que demonstram terem sido constituídas exclusivamente para o propósito desta funcionalidade.

Um motivo para o não cumprimento critério de cobertura está relacionado ao esforço em função do tempo para implementar casos de teste dirigidos à classe `RotinaCandidatoServiceImpl`. Esta classe é pouco coesa e possui uma complexidade elevada, bem como um grande número de desvios, em virtude de concentrar toda a regra de negócio e o fluxo de obtenção de dados para a execução do processamento. Além disso, nenhum caso de teste de nível de unidade direcionado à classe `RotinaCandidatoFormandoBean` foi implementado em consequência do tempo investido na criação de casos de teste para as outras classes (apresentados

pela Tabela 4.6). A implementação de casos de teste destinados ao exercício desta classe poderiam ter auxiliado a elevar suas métricas de cobertura.

A Tabela A.7 abrevia as informações sobre o percentual de cobertura de instruções e de desvios das classes consideradas essenciais.

Tabela A.7: Percentual de cobertura de instruções e de desvios para a classes essenciais da funcionalidade de processamento - versão legada

Classe	Percentual de Cobertura de Instruções	Percentual de Cobertura de Desvios
InformacoesIntegralizacaoVO	79%	100%
IntegralizacaoServiceImpl	75%	38%
RotinaCandidatoFormandoBean	24%	44%
RotinaCandidatoFormandoDAO	99%	63%
RotinaCandidatoServiceImpl	68%	47%

As classes destinadas ao tratamento de exceções plausíveis de serem lançadas durante a execução do processamento (`RotinaCanceladaException` e `RotinaCandidatoFormandoException`) não superaram os 25% cobertura em suas métricas. Em uma análise particular, identificou-se que 3 dos 4 construtores dessas classes, contabilizados como métodos pela JaCoCo, não são utilizados em nenhuma outra parte do projeto. Sendo assim, as métricas dessas classes poderiam atingir valores plenos se esses construtores não existissem.

As métricas da classe com atributos para o acompanhamento do progresso do processamento (`RotinaCandidatoFormandoProgresso`) atingiram valores bastantes expressivos (todos acima dos 85% de cobertura). A compreensão para o alcance destes valores está relacionada a estrutura da classe, onde 13 dos 15 métodos (incluindo seu único construtor) exercem apenas a função de obtenção e atribuição dos atributos privados da classe. Além disso, esta classe está vinculada aos elementos de interface que foram exercitados pelos casos de teste de nível de sistema. Alguns casos de teste de nível de unidade também contribuíram indiretamente, dado que não foram usados objetos dublês em casos de teste em que foi preciso instanciá-la.

Com relação às classes subjacentes a verificação de integralização da matriz curricular, cerne da funcionalidade, os valores das métricas de cobertura indicam que estas foram pouco exercitadas. A classe `AbstractAtividadeCurricularFormula`, bem como suas herdeiras `DispensaFormula` e `EquivalenciaFormula`, foram exercitadas pelo subconjunto de teste de nível de sistema. Contudo, nenhum dos casos de teste deste subconjunto objetivou produzir cenários em que a concessão de créditos para a integralização por dispensa ou por equivalência fosse executada. Desta forma, a classe que avalia esta concessão, `SimpleStringFormulaResolver`, e a classe para tratar exceções que pudessem surgir durante essa avaliação, `FormulaException`, também não atingiram valores expressivos de cobertura.

Tabela A.8: Cobertura de código para o processamento de candidatos à formatura e formandos – versão legada

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AbstractAtividadeCurricularFormula	38	31	69	45%	2	0	2	0%	5	3	8	38%	12	6	18	33%	4	3	7	43%	0	1	1	100%
AtividadeComplementarDAO	106	71	177	40%	1	1	2	50%	3	2	5	40%	25	18	43	42%	2	2	4	50%	0	1	1	100%
AtividadeComplementarServiceImpl	137	10	147	7%	n/a	n/a	n/a	n/a	9	2	11	18%	32	3	35	9%	9	2	11	18%	0	1	1	100%
AtividadeCurricularDAO	2946	627	3573	18%	177	9	186	5%	118	5	123	4%	514	104	618	17%	26	4	30	13%	0	1	1	100%
AtividadeCurricularServiceImpl	204	21	225	9%	10	0	10	0%	26	5	31	16%	47	5	52	10%	21	5	26	19%	0	1	1	100%
CascataStringsServiceImpl	103	15	118	13%	8	0	8	0%	9	3	12	25%	20	3	23	13%	5	3	8	38%	0	1	1	100%
CoreDAO	91	11	102	11%	n/a	n/a	n/a	n/a	4	2	6	33%	18	2	20	10%	4	2	6	33%	0	1	1	100%
CoreServiceImpl	13	8	21	38%	n/a	n/a	n/a	n/a	3	2	5	40%	3	2	5	40%	3	2	5	40%	0	1	1	100%
DispensaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
EquivalenciaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
FichaCaracterizacaoServiceImpl	1589	27	1616	2%	157	1	158	1%	119	4	123	3%	381	9	390	2%	40	4	44	9%	0	1	1	100%
FormulaException	16	0	16	0%	n/a	n/a	n/a	n/a	4	0	4	0%	8	0	8	0%	4	0	4	0%	1	0	1	0%
InformacoesIntegralizacaoVO	40	149	189	79%	0	16	16	100%	8	28	36	78%	11	52	63	83%	8	20	28	71%	0	1	1	100%
IngressoServiceImpl	141	17	158	11%	31	1	32	3%	26	3	29	10%	38	4	42	10%	10	3	13	23%	0	1	1	100%
InscricaoDAO	2818	224	3042	7%	93	3	96	3%	76	3	79	4%	558	41	599	7%	29	2	31	6%	0	1	1	100%
InscricaoServiceImpl	765	11	776	1%	46	0	46	0%	63	3	66	5%	158	3	161	2%	39	3	42	7%	0	1	1	100%
IntegralizacaoServiceImpl	12	36	48	75%	5	3	8	38%	4	3	7	43%	2	7	9	78%	0	3	3	100%	0	1	1	100%
MatriculaDAO	1542	694	2236	31%	87	11	98	11%	69	8	77	10%	294	139	433	32%	22	6	28	21%	0	1	1	100%
MatriculaServiceImpl	1541	27	1568	2%	124	0	124	0%	109	6	115	5%	343	6	349	2%	46	6	52	12%	0	1	1	100%
RotinaCanceladaException	13	3	16	19%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoBean	417	131	548	24%	19	15	34	44%	33	17	50	34%	113	34	147	23%	21	12	33	36%	0	1	1	100%

Continuação da Tabela A.8 : Cobertura de código para o processamento de candidatos à formatura e formandos – versão legada

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
RotinaCandidatoFormandoDAO	2	173	175	99%	3	5	8	63%	3	4	7	57%	1	38	39	97%	0	3	3	100%	0	1	1	100%
RotinaCandidatoFormandoException	12	4	16	25%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoProgresso	8	80	88	91%	0	4	4	100%	2	15	17	88%	4	26	30	87%	2	13	15	87%	0	1	1	100%
RotinaCandidatoFormandoServiceImpl	414	887	1301	68%	68	60	128	47%	52	40	92	43%	91	211	302	70%	6	22	28	79%	0	1	1	100%
RotinaExecucaoServiceImpl	45	54	99	55%	4	0	4	0%	6	5	11	45%	8	13	21	62%	4	5	9	56%	0	1	1	100%
SigaDAO	30	19	49	39%	4	0	4	0%	5	3	8	38%	9	6	15	40%	3	3	6	50%	0	1	1	100%
SigaServiceImpl	10	13	23	57%	n/a	n/a	n/a	n/a	2	3	5	60%	4	3	7	43%	2	3	5	60%	0	1	1	100%
SimpleStringFormulaResolver	162	3	165	2%	14	0	14	0%	10	1	11	9%	33	1	34	3%	3	1	4	25%	0	1	1	100%
StatusMatriculaServiceImpl	17	11	28	39%	n/a	n/a	n/a	n/a	3	3	6	50%	5	3	8	38%	3	3	6	50%	0	1	1	100%

As métricas de cobertura das demais classes, no que se refere às instruções e aos desvios, apresentaram, em sua maioria, valores abaixo dos 50%. Isto deve-se ao fato destas classes não possuírem funções exclusivas para a funcionalidade "Processamento de Candidatos à Formatura e Formandos". Ou seja, algumas destas são classes segregam comportamentos das respectivas classes de modelo (no formato de classes de serviço ou de acesso a dados) e possuem diversos métodos para atender a outras funcionalidades. Ainda há classes que são abstratas com a função de agrupar comportamentos comuns às suas respectivas classes herdeiras. Por fim, assim como não foi encontrado, na versão legada, um subconjunto de casos de teste dirigido à funcionalidade de processamento e considerando a quantidade de casos de teste pré-existentes (apresentada pela Tabela 3.2), há uma grande possibilidade de não existirem muitos casos de testes direcionados ao exercício destas classes.

De modo geral, 20% das instruções e 13% dos desvios das classes envolvidas no "Processamento de Candidatos à Formatura e Formandos" foram cobertos. Se apreciadas apenas as classes consideradas essenciais para a funcionalidade, 61% das instruções e 51% dos desvios foram cobertos.

Processamento Auto-otimizado de Candidatos à Formatura e Formandos com 1 Elemento de Conhecimento:

As métricas de cobertura de código coletadas da versão $SIGA\_Adapt\_3$ são apresentadas pela Tabela A.9. Nesta versão, 10 novas classes foram criadas e uma já existente foi utilizada na implementação do "Processamento Auto-otimizado de Candidatos à Formatura e Formandos" com 1 Elemento de Conhecimento.

As métricas das classes consideradas essenciais pouco variaram em relação a versão legada. A única mudança significativa encontra-se nas métricas da classe `RotinaCandidatoServiceImpl`, na qual os valores de cobertura relativos às instruções e aos desvios foram elevados. Isto é justificável, uma vez que esta classe foi refatorada para permitir que chamadas assíncronas fossem realizadas ao algoritmo que determina os *status* das matrículas com base no cumprimento da matriz curricular. Essa refatoração transferiu parte dos métodos da classe `RotinaCandidatoServiceImpl`, reduzindo-a em 181 linhas, para a classe `MatriculaIntegralizadora`, criada nesta versão. Esta nova classe, por sua vez, teve percentuais de cobertura similares à classe da qual foi derivada.

A cobertura de código das classes que já existiam na versão legada permaneceram inalteradas nesta versão. Exemplos são as classes `RotinaCanceladaException`, `RotinaCandidatoFormandoProgresso` e `EquivalenciaFormula`.

Tabela A.9: Cobertura de código para o processamento auto-otimizado de candidatos à formatura e formandos – versão $SIGA\_Adapt\_3$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AbstractAtividadeCurricularFormula	38	31	69	45%	2	0	2	0%	5	3	8	38%	12	6	18	33%	4	3	7	43%	0	1	1	100%
AnalyserRecursos	28	60	88	68%	6	6	12	50%	6	4	10	40%	2	15	17	88%	0	4	4	100%	0	1	1	100%
AtividadeComplementarDAO	106	71	177	40%	1	1	2	50%	3	2	5	40%	25	18	43	42%	2	2	4	50%	0	1	1	100%
AtividadeComplementarServiceImpl	137	10	147	7%	n/a	n/a	n/a	n/a	9	2	11	18%	32	3	35	9%	9	2	11	18%	0	1	1	100%
AtividadeCurricularDAO	2946	627	3573	18%	177	9	186	5%	118	5	123	4%	514	104	618	17%	26	4	30	13%	0	1	1	100%
AtividadeCurricularServiceImpl	204	21	225	9%	10	0	10	0%	26	5	31	16%	47	5	52	10%	21	5	26	19%	0	1	1	100%
BaseDAO	21	29	50	58%	1	1	2	50%	3	5	8	63%	5	8	13	62%	2	5	7	71%	0	1	1	100%
BaseService	5	7	12	58%	1	1	2	50%	1	2	3	67%	1	3	4	75%	0	2	2	100%	0	1	1	100%
CascataStringsServiceImpl	103	15	118	13%	8	0	8	0%	9	3	12	25%	20	3	23	13%	5	3	8	38%	0	1	1	100%
CoreDAO	91	11	102	11%	n/a	n/a	n/a	n/a	4	2	6	33%	18	2	20	10%	4	2	6	33%	0	1	1	100%
CoreServiceImpl	13	8	21	38%	n/a	n/a	n/a	n/a	3	2	5	40%	3	2	5	40%	3	2	5	40%	0	1	1	100%
DispensaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
EquivalenciaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
FichaCaracterizacaoServiceImpl	1597	19	1616	1%	158	0	158	0%	120	3	123	2%	384	6	390	2%	41	3	44	7%	0	1	1	100%
FormulaException	16	0	16	0%	n/a	n/a	n/a	n/a	4	0	4	0%	8	0	8	0%	4	0	4	0%	1	0	1	0%
InformacoesIntegralizacaoVO	40	149	189	79%	0	16	16	100%	8	28	36	78%	11	52	63	83%	8	20	28	71%	0	1	1	100%
IngressoServiceImpl	141	17	158	11%	31	1	32	3%	26	3	29	10%	38	4	42	10%	10	3	13	23%	0	1	1	100%
InscricaoDAO	2818	224	3042	7%	93	3	96	3%	76	3	79	4%	558	41	599	7%	29	2	31	6%	0	1	1	100%
InscricaoServiceImpl	765	11	776	1%	46	0	46	0%	63	3	66	5%	158	3	161	2%	39	3	42	7%	0	1	1	100%
IntegralizacaoServiceImpl	12	36	48	75%	5	3	8	38%	4	3	7	43%	2	7	9	78%	0	3	3	100%	0	1	1	100%
KnowledgeLoteRotinaCandidatoFormando	38	0	38	0%	n/a	n/a	n/a	n/a	11	0	11	0%	16	0	16	0%	11	0	11	0%	1	0	1	0%

Continuação da Tabela A.9 : Cobertura de código para o processamento auto-otimizado de candidatos à formatura e formandos – versão $SIGA\_Adapt\_3$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
KnowledgeLoteRotinaCandidatoFormandoDAO	0	44	44	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	9	9	100%	0	2	2	100%	0	1	1	100%
KnowledgeLoteRotinaCandidatoFormandoServiceImpl	0	43	43	100%	n/a	n/a	n/a	n/a	0	5	5	100%	0	12	12	100%	0	5	5	100%	0	1	1	100%
LoteExecucao	26	33	59	56%	n/a	n/a	n/a	n/a	3	6	9	67%	5	13	18	72%	3	6	9	67%	0	1	1	100%
MatriculaDAO	1542	694	2236	31%	87	11	98	11%	69	8	77	10%	294	139	433	32%	22	6	28	21%	0	1	1	100%
MatriculaIntegralizadora	319	554	873	63%	59	47	106	44%	43	29	72	40%	70	139	209	67%	3	16	19	84%	0	1	1	100%
MatriculaServiceImpl	1541	27	1568	2%	124	0	124	0%	109	6	115	5%	343	6	349	2%	46	6	52	12%	0	1	1	100%
MonitorRecursos	0	26	26	100%	n/a	n/a	n/a	n/a	0	5	5	100%	0	8	8	100%	0	5	5	100%	0	1	1	100%
PlannerProcessRotinaCandidatoFormando	36	54	90	60%	2	2	4	50%	3	6	9	67%	7	16	23	70%	1	6	7	86%	0	1	1	100%
ProcessRotinaCandidatoFormando	9	37	46	80%	2	4	6	67%	2	3	5	60%	2	10	12	83%	0	2	2	100%	0	1	1	100%
RotinaCanceladaException	13	3	16	19%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoBean	421	127	548	23%	19	15	34	44%	34	16	50	32%	115	32	147	22%	22	11	33	33%	0	1	1	100%
RotinaCandidatoFormandoDAO	2	173	175	99%	3	5	8	63%	3	4	7	57%	1	38	39	97%	0	3	3	100%	0	1	1	100%
RotinaCandidatoFormandoException	12	4	16	25%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoProgresso	8	80	88	91%	0	4	4	100%	2	15	17	88%	4	26	30	87%	2	13	15	87%	0	1	1	100%
RotinaCandidatoFormandoServiceImpl	106	429	535	80%	11	19	30	63%	11	16	27	59%	24	97	121	80%	3	9	12	75%	0	1	1	100%
RotinaExecucaoServiceImpl	45	54	99	55%	4	0	4	0%	6	5	11	45%	8	13	21	62%	4	5	9	56%	0	1	1	100%
SigaDAO	30	19	49	39%	4	0	4	0%	5	3	8	38%	9	6	15	40%	3	3	6	50%	0	1	1	100%
SigaServiceImpl	10	13	23	57%	n/a	n/a	n/a	n/a	2	3	5	60%	4	3	7	43%	2	3	5	60%	0	1	1	100%
SimpleStringFormulaResolver	162	3	165	2%	14	0	14	0%	10	1	11	9%	33	1	34	3%	3	1	4	25%	0	1	1	100%
StatusLoteRotina	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	6	6	100%	0	3	3	100%	0	1	1	100%
StatusMatriculaServiceImpl	17	11	28	39%	n/a	n/a	n/a	n/a	3	3	6	50%	5	3	8	38%	3	3	6	50%	0	1	1	100%

No que se refere às classes introduzidas relacionadas ao MAPE-K, as métricas revelam que grande parte de seus códigos foram exercitados. Por exemplo, os testes cobriram metade dos desvios e mais da metade das instruções das classes `AnalyserRecursos` e `PlannerProcessRotinaCandidatoFormando`, que representam, respectivamente, os componentes de análise e planejamento. A classe `MonitorRecursos`, que representa o componente de monitoramento, por exemplo, foi completamente coberta, assim como as classes utilizadas no acesso a dados e para a manipulação de informações relativas ao elemento de conhecimento (respectivamente, `KnowledgeLoteRotinaCandidatoFormandoDAO` e `KnowledgeLoteRotinaCandidatoFormandoServiceImpl`). Os valores referentes a essas classes são resultados da instrumentação de código realizada pelos casos de teste em nível de sistema.

As classes remanescentes apresentadas na Tabela A.9 são classes abstratas, de serviço e de acesso a dados que já existiam desde a versão legada. Os valores das métricas de cobertura de todas permaneceram idênticos aos coletados da versão legada, exceto pela classe `FichaCaracterizacaoServiceImpl`, da qual os valores decresceram sutilmente. O motivo para esse decréscimo está relacionado à execução de um processo inicializado por meio de um temporizador que invocou um dos métodos desta classe durante a instrumentação dos testes em nível de sistema na versão legada e que não foi inicializado durante a instrumentação desta versão.

Resumidamente, apenas a classe que foi refatorada sofreu algum impacto relevante. Além disso, 22% de todas as instruções e 14% do total de desvios das classes envolvidas no “Processamento Auto-otimizado de Candidatos à Formatura e Formandos” com 1 Elemento de Conhecimento foram cobertos. Em contrapartida, se avaliadas apenas as classes referentes ao processo de adaptação (aquelas que representam algum item do modelo MAPE-K), 70% das instruções e 55% dos desvios foram cobertos.

Processamento Auto-otimizado de Candidatos à Formatura e Formandos com 2 Elementos de Conhecimento:

A última versão implementada, denominada $SIGA\_Adapt\_4$, refere-se ao “Processamento Auto-otimizado de Candidatos à Formatura e Formandos” com dois elementos de conhecimento, na qual foram adicionadas 3 novas classes. As métricas de cobertura de código desta versão podem ser visualizadas na Tabela A.10.

As coberturas das classes que já faziam parte da versão precedente, $SIGA\_Adapt\_3$, tantas as consideradas essenciais quanto as subjacentes à funcionalidade de processamento, permaneceram inalterados. A integridade destes resultados está relacionada à sutileza da implementação, que introduziu apenas mais um elemento de conhecimento ao processo de adaptação.

Tabela A.10: Cobertura de código para o processamento auto-otimizado de candidatos à formatura e formandos – versão $SIGA\_Adapt\_4$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
AbstractAtividadeCurricularFormula	38	31	69	45%	2	0	2	0%	5	3	8	38%	12	6	18	33%	4	3	7	43%	0	1	1	100%
AnalyserRecursos	73	95	168	57%	11	7	18	39%	9	5	14	36%	5	23	28	82%	0	5	5	100%	0	1	1	100%
AtividadeComplementarDAO	106	71	177	40%	1	1	2	50%	3	2	5	40%	25	18	43	42%	2	2	4	50%	0	1	1	100%
AtividadeComplementarServiceImpl	137	10	147	7%	n/a	n/a	n/a	n/a	9	2	11	18%	32	3	35	9%	9	2	11	18%	0	1	1	100%
AtividadeCurricularDAO	2946	627	3573	18%	177	9	186	5%	118	5	123	4%	514	104	618	17%	26	4	30	13%	0	1	1	100%
AtividadeCurricularServiceImpl	204	21	225	9%	10	0	10	0%	26	5	31	16%	47	5	52	10%	21	5	26	19%	0	1	1	100%
BaseDAO	21	29	50	58%	1	1	2	50%	3	5	8	63%	5	8	13	62%	2	5	7	71%	0	1	1	100%
BaseService	5	7	12	58%	1	1	2	50%	1	2	3	67%	1	3	4	75%	0	2	2	100%	0	1	1	100%
CascataStringsServiceImpl	103	15	118	13%	8	0	8	0%	9	3	12	25%	20	3	23	13%	5	3	8	38%	0	1	1	100%
CoreDAO	91	11	102	11%	n/a	n/a	n/a	n/a	4	2	6	33%	18	2	20	10%	4	2	6	33%	0	1	1	100%
CoreServiceImpl	13	8	21	38%	n/a	n//a	n/a	n/a	3	2	5	40%	3	2	5	40%	3	2	5	40%	0	1	1	100%
DispensaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
EquivalenciaFormula	29	11	40	28%	9	1	10	10%	5	2	7	29%	8	4	12	33%	0	2	2	100%	0	1	1	100%
FichaCaracterizacaoServiceImpl	1597	19	1616	1%	158	0	158	0%	120	3	123	2%	384	6	390	2%	41	3	44	7%	0	1	1	100%
FormulaException	16	0	16	0%	n/a	n/a	n/a	n/a	4	0	4	0%	8	0	8	0%	4	0	4	0%	1	0	1	0%
InformacoesIntegralizacaoVO	40	149	189	79%	0	16	16	100%	8	28	36	78%	11	52	63	83%	8	20	28	71%	0	1	1	100%
IngressoServiceImpl	141	17	158	11%	31	1	32	3%	26	3	29	10%	38	4	42	10%	10	3	13	23%	0	1	1	100%
InscricaoDAO	2818	224	3042	7%	93	3	96	3%	76	3	79	4%	558	41	599	7%	29	2	31	6%	0	1	1	100%
InscricaoServiceImpl	765	11	776	1%	46	0	46	0%	63	3	66	5%	158	3	161	2%	39	3	42	7%	0	1	1	100%
IntegralizacaoServiceImpl	12	36	48	75%	5	3	8	38%	4	3	7	43%	2	7	9	78%	0	3	3	100%	0	1	1	100%
KnowledgeLoteRotinaCandidatoFormando	38	0	38	0%	n/a	n/a	n/a	n/a	11	0	11	0%	16	0	16	0%	11	0	11	0%	1	0	1	0%
KnowledgeLoteRotinaCandidatoFormandoDAO	0	44	44	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	9	9	100%	0	2	2	100%	0	1	1	100%
KnowledgeLoteRotinaCandidatoFormandoServiceImpl	0	43	43	100%	n/a	n/a	n/a	n/a	0	5	5	100%	0	12	12	100%	0	5	5	100%	0	1	1	100%

Continuação da Tabela A.10 : Cobertura de código para o processamento auto-otimizado de candidatos à formatura e formandos – versão $SIGA\_Adapt\_4$

Classe	Instruções				Desvios				Complexidade				Linhas				Métodos				Classes			
	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P	I	C	T	P
KnowledgeMemoriaRotinaCandidatoFormando	47	0	47	0%	n/a	n/a	n/a	n/a	13	0	13	0%	19	0	19	0%	13	0	13	0%	1	0	1	0%
KnowledgeMemoriaRotinaCandidatoFormandoDAO	0	12	12	100%	n/a	n/a	n/a	n/a	0	2	2	100%	0	3	3	100%	0	2	2	100%	0	1	1	100%
KnowledgeMemoriaRotinaCandidatoFormandoServiceImpl	36	18	54	33%	n/a	n/a	n/a	n/a	2	3	5	60%	8	7	15	47%	2	3	5	60%	0	1	1	100%
LoteExecucao	26	33	59	56%	n/a	n/a	n/a	n/a	3	6	9	67%	5	13	18	72%	3	6	9	67%	0	1	1	100%
MatriculaDAO	1542	694	2236	31%	87	11	98	11%	69	8	77	10%	294	139	433	32%	22	6	28	21%	0	1	1	100%
MatriculaIntegralizadora	319	554	873	63%	59	47	106	44%	43	29	72	40%	70	139	209	67%	3	16	19	84%	0	1	1	100%
MatriculaServiceImpl	1541	27	1568	2%	124	0	124	0%	109	6	115	5%	343	6	349	2%	46	6	52	12%	0	1	1	100%
MonitorRecursos	0	26	26	100%	n/a	n/a	n/a	n/a	0	5	5	100%	0	8	8	100%	0	5	5	100%	0	1	1	100%
PlannerProcessRotinaCandidatoFormando	36	54	90	60%	2	2	4	50%	3	6	9	67%	7	16	23	70%	1	6	7	86%	0	1	1	100%
ProcessRotinaCandidatoFormando	9	35	44	80%	2	4	6	67%	2	3	5	60%	2	10	12	83%	0	2	2	100%	0	1	1	100%
RotinaCanceladaException	13	3	16	19%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoBean	421	127	548	23%	19	15	34	44%	34	16	50	32%	115	32	147	22%	22	11	33	33%	0	1	1	100%
RotinaCandidatoFormandoDAO	2	173	175	99%	3	5	8	63%	3	4	7	57%	1	38	39	97%	0	3	3	100%	0	1	1	100%
RotinaCandidatoFormandoException	12	4	16	25%	n/a	n/a	n/a	n/a	3	1	4	25%	6	2	8	25%	3	1	4	25%	0	1	1	100%
RotinaCandidatoFormandoProgresso	8	80	88	91%	0	4	4	100%	2	15	17	88%	4	26	30	87%	2	13	15	87%	0	1	1	100%
RotinaCandidatoFormandoServiceImpl	106	429	535	80%	11	19	30	63%	11	16	27	59%	24	97	121	80%	3	9	12	75%	0	1	1	100%
RotinaExecucaoServiceImpl	45	54	99	55%	4	0	4	0%	6	5	11	45%	8	13	21	62%	4	5	9	56%	0	1	1	100%
SigaDAO	30	19	49	39%	4	0	4	0%	5	3	8	38%	9	6	15	40%	3	3	6	50%	0	1	1	100%
SigaServiceImpl	10	13	23	57%	n/a	n/a	n/a	n/a	2	3	5	60%	4	3	7	43%	2	3	5	60%	0	1	1	100%
SimpleStringFormulaResolver	162	3	165	2%	14	0	14	0%	10	1	11	9%	33	1	34	3%	3	1	4	25%	0	1	1	100%
StatusLoteRotina	0	37	37	100%	n/a	n/a	n/a	n/a	0	3	3	100%	0	6	6	100%	0	3	3	100%	0	1	1	100%
StatusMatriculaServiceImpl	17	11	28	39%	n/a	n/a	n/a	n/a	3	3	6	50%	5	3	8	38%	3	3	6	50%	0	1	1	100%

Desta forma, somente a classe que representa o componente de análise (`AnalyserRecursos`) teve a cobertura de código alterada, tendo em vista que sua implementação também foi alterada. A redução da cobertura referente às métricas de instruções e desvios está relacionada ao aumento no total de número de linhas e de complexidade com a incorporação do novo elemento de conhecimento na execução da análise. Essa incorporação é resultado da criação de classes utilizadas para manipular dados pertinentes ao segundo elemento de conhecimento (`KnowledgeMemoriaLoteRotinaCandidatoFormando`, `KnowledgeMemoriaLoteRotinaCandidatoFormandoServiceImpl` e `KnowledgeMemoriaLoteRotinaCandidatoFormandoDAO`). Os valores das métricas de cobertura referentes a essas classes são resultados da instrumentação de código realizada pelos casos de teste em nível de sistema.

De modo geral, 22% de todas as instruções e 14% do total de desvios das classes envolvidas no "Processamento Auto-otimizado de Candidatos à Formatura e Formandos" com dois elementos de conhecimento foram cobertos. Em compensação, 58% das instruções e 46% dos desvios totais das classes pertinentes ao processo de adaptação, isto é, aquelas relativas ao modelo MAPE-K, foram cobertas.

Anexo B

CLASSES POR PACOTE

Tabela B.1: Classes por pacote do ramo Java EE

Pacote	Classe
br.ufscar.core.repository.dao	CoreDAO *
	UsuarioDAO
br.ufscar.core.service.impl	CoreServiceImpl *
	CriptografadorSagui
	UsuarioServiceImpl
br.ufscar.manager.autenticacao	AnalyserAutenticacao
	MonitorAutenticacao *
	MonitorAutenticacaoDB
	MonitorAutenticacaoLdap
	MonitorAutenticacaoSaguiApi
	TrocadorMecanismoAutenticacao
	TrocaMecanismoAutenticacaoEvent
br.ufscar.manager.rotina.candidatoformando	AnalyserRecursos
	MonitorRecursos
	PlannerProcessRotinaCandidatoFormando
	ProcessRotinaCandidatoFormando
br.ufscar.siga.aluno.repository.dao	AtividadeComplementarDAO
	RotinaCandidatoFormandoDAO
br.ufscar.siga.aluno.service.impl	AtividadeComplementarServiceImpl
	InformacoesIntegralizacaoVO
	RotinaCandidatoFormandoException
	RotinaCandidatoFormandoProgresso
	RotinaCandidatoFormandoServiceImpl
br.ufscar.siga.base.bean	AuthenticationBeanImpl
br.ufscar.siga.base.filter	AuthenticationFilter
br.ufscar.siga.cadastrosgerais.repository.dao	AtividadeCurricularDAO
	InscricaoDAO
br.ufscar.siga.cadastrosgerais.service.impl	AtividadeCurricularServiceImpl
	FichaCaracterizacaoServiceImpl

Continuação da Tabela B.1

Pacote	Classe
br.ufscar.siga.ejb.autenticacao	AutenticacaoDB
	AutenticacaoDBException
	AutenticacaoException
	AutenticacaoFactory
	AutenticacaoLdap
	AutenticacaoLdapException
	AutenticacaoSaguiApi
	AutenticacaoSaguiApi.CredendiaisSaguiApi
	AutenticacaoSaguiApiException
br.ufscar.siga.ejb.crypt.bcrypt	BCrypt
	BCryptPasswordEncoder
br.ufscar.siga.ejb.matricula	LoteExecucao
	MatriculaIntegralizadora
br.ufscar.siga.ejb.repository.dao	BaseDAO *
	KnowledgeAutenticacaoDAO
	KnowledgeLoteRotinaCandidatoFormandoDAO
	KnowledgeMemoriaLoteRotinaCandidatoFormandoDAO
	SigaDAO *
br.ufscar.siga.ejb.rest.client	RestfulClientImpl
br.ufscar.siga.ejb.service	BaseService *
br.ufscar.siga.ejb.service.impl	KnowledgeAutenticacaoServiceImpl
	KnowledgeLoteRotinaCandidatoFormandoServiceImpl
	KnowledgeMemoriaLoteRotinaCandidatoFormandoServiceImpl
	SigaServiceImpl *
br.ufscar.siga.entity.autenticacao	KnowledgeAutenticacao
	MecanismoAutenticacao **
	MecanismoStatus **
br.ufscar.siga.entity.periodoletivo.rotina	KnowledgeLoteRotinaCandidatoFormando
	KnowledgeMemoriaLoteRotinaCandidatoFormando
	StatusLoteRotina **
br.ufscar.siga.inscricao.bean	RotinaCandidatoFormandoBean
br.ufscar.siga.inscricao.service	FormulaException
	RotinaCanceladaException
br.ufscar.siga.inscricao.service.impl	AbstractAtividadeCurricularFormula *
	CascataStringsServiceImpl
	DispensaFormula
	EquivalenciaFormula
	InscricaoServiceImpl
	IntegralizacaoServiceImpl
	RotinaExecucaoServiceImpl
	SimpleStringFormulaResolver
br.ufscar.siga.login.bean	LoginBean

Continuação da Tabela B.1

Pacote	Classe
br.ufscar.siga.login.service.impl	LoginServiceImpl
br.ufscar.siga.matricula.repository.dao	MatriculaDAO
br.ufscar.siga.matricula.service.impl	MatriculaServiceImpl
	StatusMatriculaServiceImpl
br.ufscar.siga.processos.services.impl	IngressoServiceImpl
br.ufscar.siga.system.repository.dao	AtributoSistemaDAO
br.ufscar.siga.system.service.impl	AtributoSistemaServiceImpl

Tabela B.2: Classes por pacote do ramo StarMX

Pacote	Classe
br.ufscar.siga.base.bean	AuthenticationBeanImpl
br.ufscar.siga.base.filter	AuthenticationFilter
br.ufscar.siga.ejb.autenticacao	AutenticacaoException
	AutenticacaoFactory
	AutenticacaoLdap
	AutenticacaoLdapException
	AutenticacaoSaguiApi
	AutenticacaoSaguiApi.CredendiaisSaguiApi
	AutenticacaoSaguiApiException
br.ufscar.siga.ejb.mbean	AnalyserAutenticacaoJMX
	MonitorAutenticacao *
	MonitorAutenticacaoLdapJMX
	MonitorAutenticacaoSaguiApiJMX
	TrocadorMecanismoAutenticacao
br.ufscar.siga.ejb.rest.client	RestfulClientImpl
br.ufscar.siga.ejb.service.impl	KnowledgeAutenticacaoServiceImpl
	SigaServiceImpl *
br.ufscar.siga.entity.autenticacao	KnowledgeAutenticacao
	MecanismoAutenticacao **
	MecanismoStatus **
br.ufscar.siga.login.bean	LoginBean
br.ufscar.siga.login.service.impl	LoginServiceImpl
br.ufscar.siga.manager.process	AnalyserAutenticacaoProcess
	MonitorAutenticacaoLdapProcess
	MonitorAutenticacaoSaguiApiProcess
br.ufscar.siga.repository.dao	KnowledgeAutenticacaoDAO
	SigaDAO *
br.ufscar.siga.system.repository.dao	AtributoSistemaDAO
br.ufscar.siga.system.service.impl	AtributoSistemaServiceImpl

* classe abstrata
** enumeração