UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS APLICADAS E EDUCAÇÃO

DEPARTAMENTO DE CIÊNCIAS EXATAS

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Teste de Sistemas de Informação Assíncronos: Um Estudo de Caso

DIEGO SOUSA DE AZEVEDO

Profa. Dra. Ayla Débora Dantas de Souza Rebouças
(Orientadora)

RIO TINTO - PB
2013

DIEGO SOUSA DE AZEVEDO

Teste de Sistemas de Informação Assíncronos: Um Estudo de Caso

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharelem Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE), Campus IV da Universidade Federal da Paraíba.

Orientadora: Profa. Dra. Ayla Débora Dantas de Souza Rebouças.

RIO TINTO - PB
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca da UFPB

A994t *Azevedo, Diego Sousa de.*

Testes de Sistemas de Informação Assíncronos: Um estudo de caso / Diego Sousa de Azevedo. – Rio Tinto: [s.n.], 2013.

55f.: il. –

Orientadora: Ayla Débora Dantas de Souza Rebouças.
Monografia (Graduação) – UFPB/CCAE.

1. Teste de software. *2. Sistemas assíncronos. 3. Sistemas concorrentes. I. Título.*

UFPB/BS-CCAE *CDU: 004.415.53 (043.2)*

DIEGO SOUSA DE AZEVEDO

Teste de Sistemas de Informação Assíncronos: Um Estudo de Caso

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal da Paraíba, Campus IV, como parte dos requisitos necessários para obtenção do grau de BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Assinatura do autor:_____

APROVADO POR:

_____

Orientadora: Profa. DSc. Ayla Débora Dantas de Souza Rebouças
Universidade Federal da Paraíba – Campus IV

_____

Prof. MSc. Rodrigo Almeida Vilar
Universidade Federal da Paraíba – Campus IV

_____

Prof. MSc. Marcus Williams Aquino de Carvalho
Universidade Federal da Paraíba – Campus IV

RIO TINTO - PB
2013

A minha mãe, Maria de Lourdes, que foi a peça fundamental na conclusão de mais este desafio.

AGRADECIMENTOS

Primeiramente a Deus por ter me proporcionado força para superar todos os desafios. A minha mãe, a qual não tenho nem palavras para descrever tudo o que ela significa para mim. A minha namorada Raquel, pelo seu apoio e incentivo nas horas mais difíceis. A todos os meus amigos adquiridos durante esta jornada acadêmica, dos quais lembrarei pelo resto da minha vida. A todos os professores da UFPB que contribuíram para minha formação, em especial a professora Ayla Rebouças, que em todos esses anos esteve me orientando em atividades acadêmicas, repassando valiosos ensinamentos, os quais se tornaram as principais referências que possuo hoje. A toda equipe da INDRA e do projeto SIM, que além de colegas de trabalho, são grandes amigos. A todas as outras pessoas que não me vieram à memória agora mais que contribuíram fortemente para a minha formação pessoal e profissional.

RESUMO

Considerando as muitas dificuldades existentes em se testar sistemas assíncronos, o objetivo geral deste trabalho é apresentar os resultados de um estudo de caso realizado no intuito de divulgar formas para que desenvolvedores de testes possam evitar falsos positivos na execução de seus testes automáticos para sistemas de informação assíncronos. Alguns desses falsos positivos ocorrem por causa do problema das asserções antecipadas e tardias. Ou seja, as verificações de estado do sistema ou dos resultados que este produz acabam não sendo verdadeiras no momento em que são executadas. Isso é sinalizado através de uma falha obtida como resultado da execução do teste, o qual tem como finalidade indicar a existência de um defeito no software sendo testado e não um defeito no teste.

Ainda como objetivo desta pesquisa, foi planejado mostrar que se pode amenizar o tempo gasto com a execução de testes de sistemas assíncronos por meio do uso do *arcabouço* ThreadControl, comparando-o com o uso de atrasos explícitos, como também, propor futuras investigações relativas ao teste de sistemas assíncronos. A metodologia empregada neste trabalho incluiu levantamentos bibliográficos e análise de trabalhos relacionados aos principais assuntos discutidos, entre eles, sistemas *multi-threaded*, teste de software, sistemas distribuídos e sistemas assíncronos. A metodologia deste trabalho também incluiu o projeto e construção de um sistema a ser utilizado no estudo de caso, totalmente assíncrono e distribuído para que fossem utilizados os seus testes como objeto deste estudo. Com base neste uso, este trabalho discute alguns dos principais desafios e lições aprendidas no teste de sistemas assíncronos.

Palavras chave: teste de software, sistemas assíncronos, sistemas concorrentes.

ABSTRACT

Considering that it is difficult to test asynchronous systems, the objective of this work is to present the results of a case study intended to show how test developers can avoid false positives in the execution of automated tests for asynchronous information systems. Some of those false positives occur because of late or early assertions (verifications of the system that are not performed at an appropriate time). This is signaled by a test failure, whose purpose is to indicate the existence of a defect in the software under test and not in the test itself.

Besides, another objective of this research was to show that we can reduce the time spent with the execution of asynchronous systems tests through the use of the ThreadControl framework instead of alternatives such as explicit delays. Moreover, we also propose some future investigations related to the test of asynchronous systems. The methodology used in this study included literature reviews and analysis of related work, including topics as multi-threaded systems, software testing, distributed systems and asynchronous systems. The methodology of this work also included the design and development of a system to be used in the case study, which had to be asynchronous and distributed and whose tests were the object of this study. Based on this use, this study discusses some of the key challenges and lessons learned in testing asynchronous systems.

Keywords: software testing, asynchronous systems, concurrent systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

SUT - System Under Test

CRM - Customer Relationship Management

EUA – Estados Unidos da América

REST - Representational State Transfer

HTTP - Hypertext Transfer Protocol

JSON - JavaScript Object Notation

V & V - Verificação e Validação

API - Application Programming Interface

AJDT - AspectJ Development Tools

IDE - Integrated Development Environment

SOA - Service-Oriented Architecture

DAO - Data Access Object

JPA – Java Persistence Api

TC - ThreadControl

RAM - Random-access memory

CPU - Central Processing Unit

JAR - Java Archive

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta a proposta deste trabalho. Inicialmente são abordadas as motivações que levaram à escolha deste tema, seguido da metodologia que descreve os passos sistemáticos executados durante esta pesquisa. Além disso, também são apresentados os objetivos que se deseja alcançar e por fim se descreve a estrutura deste documento.

1.1 MOTIVAÇÃO

A falta de testes apropriados de software pode causar diversos problemas, como por exemplo, o que ocorreu em 1990, onde 75 milhões de telefones nos EUA ficaram sem resposta devido a uma linha de código escrita na linguagem C que apresentava um *BREAK* no local incorreto. Em 1996, em menos de um minuto do lançamento do foguete francês Ariane 501 ele se autodestruiu devido a uma conversão de um número de 64bits em um espaço de 16bits ocasionando um erro de compatibilidade, pois o inteiro de 64bits era maior do que o espaço de 16bits poderia suportar, gerando um prejuízo em torno de oito bilhões de dólares. Em 1998 um erro de software da nave espacial Mars Polar Lander fez com que ela voasse muito baixa provocando uma colisão com o solo [Colin 2007].

Sendo assim, observa-se que testar software é importante. No entanto, testar software pode ser bem desafiante. Quando testamos um sistema, queremos estimular as operações e verificar se os resultados produzidos por tais operações são como esperado. Testes e especialmente testes automáticos são realmente importantes para a evolução dos sistemas, principalmente hoje, quando essa evolução deve ser muito rápida. Grande parte dos sistemas on-line que usamos hoje são frequentemente atualizados e precisamos de testes que também possam ser executados rapidamente para verificar se mesmo depois de algumas mudanças, como a introdução de um novo recurso ou uma correção de um defeito (*bug),* o sistema ainda está funcionando conforme o esperado (pelo menos para os cenários cobertos pelos testes).

Quando um teste falha é um sinal de que algo está errado no sistema e que um defeito deve ser encontrado. No entanto, quando testamos sistemas assíncronos, alguns falsos positivos (falsas sinalizações de erros no sistema) podem ocorrer, ou seja, o teste falha, mas o erro não está no código, mas sim no sincronismo entre o teste e o sistema sendo testado, o que pode causar vários problemas para a equipe de desenvolvimento. Às vezes, o teste pode falhar porque a fase de verificação do teste é feita muito cedo, quando os resultados esperados por meio do teste não foram produzidos ainda, ou muito tarde, quando o sistema já atingiu outro estado já diferente do estado esperado para suas *thread*s no momento da asserção. Uma causa

comum para este problema é o uso de atrasos explícitos (como *Thread.sleep(timeout)*) ou esperas ocupadas (atrasos explícitos dentro de loops) nos testes entre as fases de estimulação e de verificação.Os atrasos explícitos são alternativas mais simples do que os mecanismos de sincronização entre o teste e o Sistema sob Teste (*System Under Test* - SUT), as quais podem ser muito invasivas, considerando a necessidade de alterações no SUT.Segundo Goetz and Peierls (2006) essa prática é conhecida como "easier said than done". Em um trabalho de estágio realizado por Gaudencio (2009), foi constatado o uso de atrasos explícitos e esperas ocupadas em testes de alguns sistemas de código aberto populares, como o Apache Hadoop, o Tomcat, o Ant e o JBoss.

Com práticas assim, os resultados dos testes acabam sendo falhas em algumas de suas execuções, e estas são na verdade falsos positivos.Dessa forma, uma falha obtida como resultado da execução do teste, que tem como finalidade indicar a existência de um defeito no software sendo testado, acaba sendo na verdade um resultado de um problema com o teste e não com o software sendo testado [Dantas 2010]. Quando o problema encontrado pelo teste é resultante do teste e não do software, ele pode ser um erro difícil de encontrar,fazendo com que os programadores percam muito tempo tentando encontrar um erro que não existe no software. O *arcabouço* proposto por Dantas (2010), chamado ThreadControl, busca evitar asserções antecipadas e tardias e é uma alternativa aos atrasos explícitos e espera ocupada ou a mecanismos explícitos de sincronização.

Neste trabalho de conclusão de curso pretende-se comparar o uso de testes com ThreadControl e com atrasos explícitos. Esta comparação será realizada por meio de um estudo de caso com um sistema de informação assíncrono, melhor documentando a forma de usar esse *arcabouço* e algumas lições aprendidas com seu uso, além de suas vantagens e limitações.

O sistema que será utilizado como estudo de caso é um sistema CRM (*Customer Relationship Management*) simples, chamado MySimpleCRM, e que apresenta uma interface do tipo REST (*Representational State Transfer*), a qual basicamente utiliza-se de mensagens HTTP (*Hypertext Transfer Protocol*) para comunicação dos dados em formato de arquivo JSON (JavaScript Object Notation). Segundo Fielding (2012) REST é "*Acoordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations*", ou seja, REST tem o objetivo de criar uma comunicação de rede inserindo requisitos que aperfeiçoam a ligação dos componentes, como escalabilidade e redução da latência, tornando os componentes menos dependentes. Tal sistema foi escolhido como estudo

de caso por utilizar tecnologias comuns na atualidade e por tentar explorar bons princípios da engenharia de software. Vale ressaltar que a escolha de desenvolver um software completamente novo e não utilizar algum outro já existente, surgiu da busca frustrada de sistemas de informação assíncronos de código aberto e bem documentados que permitissem um bom controle de possíveis falhas nos seus resultados de teste.

1.2 OBJETIVO GERAL

O objetivo geral deste trabalho é auxiliar desenvolvedores de testes de sistemas assíncronos a evitar testes que gerem falsos positivos em sua execução, como também, amenizar o enorme tempo gasto com a execução deste tipo de teste por meio do *arcabouço* ThreadControl, comparando-o com o uso de atrasos explícitos. Pretende-se com isso explicitar algumas lições aprendidas, vantagens e desvantagens, que foram identificadas por meio do estudo de caso realizado em um sistema de informação assíncrono.

1.3 OBJETIVOS ESPECÍFICOS

- Realizar um levantamento bibliográfico de abordagens propostas para se testar sistemas assíncronos;
- Desenvolver um sistema de informação com tecnologias comuns da atualidade que seja assíncrono e simples para utilizar como estudo de caso;
- Desenvolver testes para o sistema de informação desenvolvido utilizando o *arcabouço* ThreadControl e utilizando atrasos explícitos avaliando tempo de execução dos testes produzidos, existência de falsos positivos na execução dos testes vantagens e desvantagens das duas abordagens;
- Identificar lições aprendidas durante a realização do estudo de caso.

1.4 METODOLOGIA

A metodologia adotada para este trabalho é baseada no modelo exploratório de pesquisa com foco em análise comparativa por meio de um estudo de caso, dado que a pesquisa exploratória envolve levantamento bibliográfico de experiências práticas com o problema pesquisado e análise de exemplos que estimulem a compreensão [Clemente et al. 2007]. O objetivo desse tipo de estudo é procurar padrões, idéias ou hipóteses, como também avaliar quais teorias ou conceitos existentes pode ser aplicados a um determinado problema ou se novas teorias e

conceitos devem ser desenvolvidos [Collis et al. 2005]. A partir deste modelo este trabalho engloba as seguintes atividades:

- Levantamento bibliográfico nos principais engenhos, como a busca Google, Google Acadêmico e bibliotecas digitais da ACM e IEEE, sempre dando maior ênfase aos principais assuntos discutidos nessa pesquisa, entre eles, sistemas *multi-threaded*, teste de software, sistemas distribuídos e sistemas assíncronos, com o objetivo de compreender um pouco mais estes tópicos. Como critério de sucesso e resultados esperados para esta atividade, esperava-se encontrar nas referências bibliográficas o conteúdo necessário para sanar as possíveis duvidas que surgiram durante o desenvolvimento desta pesquisa.
- Levantamento de trabalhos relacionados, nos principais engenhos citados na atividade acima, para se testar sistemas assíncronos. Como critério de sucesso e resultados esperados para esta atividade, esperava-se encontrar outras formas de se testar sistemas assíncronos além do uso do arcabouço ThreadControl.
- Construção de um sistema a ser utilizado como estudo de caso e que seja totalmente assíncrono, *multi-threaded* e distribuído para que seus testes fossem o objeto deste estudo. Como critério de sucesso e resultados esperados para esta atividade, esperava-se desenvolver todo o sistema proposto em um tempo hábil, com isso, sendo possível utilizar seus testes na avaliação de algumas abordagens para o teste de sistemas assíncronos (o uso do arcabouço *ThreadControl* e o uso de atrasos explícitos).
- Avaliação de algumas alternativas de se testar o sistema proposto nesta pesquisa demonstrando vantagens e desvantagens de cada uma. Para isso, para cada abordagem foram desenvolvidos casos de testes e foram definidas métricas de comparação, como o tempo para execução total de cada caso de teste e a quantidade de falsos positivos durante algumas execuções.Como critério de sucesso para esta atividade, esperava-se obter dados que demonstrassem vantagens, desvantagens e lições aprendidas no teste de sistemas assíncronos.

1.5 ESTRUTURA DO TRABALHO

O restante deste trabalho está organizado da seguinte forma. No capitulo dois é apresentado o referencial teórico com o intuito de familiarizar o leitor com alguns tópicos que são discutidos durante este trabalho. No capitulo três é descrito o sistema que foi utilizado no estudo de caso

e como se pode fazer seus testes que envolvem operações assíncronas. No capitulo quatro, apresentamos os principais resultados observados e as lições aprendidas com o estudo de caso realizado. Finalmente, no capítulo cinco são apresentadas as considerações finais e direções para pesquisas futuras juntamente com alguns trabalhos relacionados.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta alguns dos principais temas diretamente relacionados com esta pesquisa.

2.1 TESTE DE SOFTWARE

Teste de Software é uma área da engenharia de software destinada a melhorar a confiança de que o software irá se comportar corretamente e para verificar se o produto produzido atende às especificações exigidas pelo cliente. De acordo com Dijkstra (1969), teste de programas pode ser usado para mostrar a presença de defeitos, mas nunca para mostrar a sua ausência. Testes estão entre as abordagens de verificação e validação (V &V) de sistemas e eles são caracterizados como uma técnica em que o software é exercitado em busca de possíveis defeitos, para aumentar a confiança de que ele irá se comportar corretamente quando em produção [Sommerville 2006]. Um teste pode ser feito manualmente ou por meio de ferramentas de teste automatizado, o que pode criar um ambiente de teste mais eficiente e reproduzível, melhorando a qualidade do teste [Dustin et al.1999] e minimizando a execução de testes manuais.

2.2 TESTE DE SISTEMA ASSÍNCRONO

Operações assíncronas são operações que ocorrem, mas não são executadas obrigatoriamente no mesmo momento em que são invocadas. Segundo Martins (1985), um sistema assíncrono, ou sistema sem relógio, baseia-se no fato de que não existe qualquer garantia sobre o momento em que a tarefa que corresponde a um funcionamento assíncrono é executada.

Ao escrever testes para código assíncrono, teremos que lidar com a falta de sincronização entre o sistema e o teste. Por outro lado, para as operações síncronas, os testes aguardam pelas respostas assim que as requisitam, e assim, os erros são detectados imediatamente, o que torna este tipo de teste muito mais simples. Por exemplo, depois de invocar uma operação síncrona, tal como *insertNewProduct(productId, amount)*, é possível criar um teste que verifique se o produto foi inserido sem ter a necessidade de incluir no teste qualquer tipo de atraso. Os testes assíncronos normalmente não obtêm as respostas das

operações assim que as invocações a estas operações são concluídas,o que pode resultar em falsos positivos (falsos erros), já explanados no tópico de motivação, que são erros que não estão no código, mas sim no sincronismo entre o teste e o sistema sendo testado, o que poderá confundir os desenvolvedores de teste fazendo com que percam muito tempo procurando um defeito (*bug*) que não existe. Os falsos positivos não são apenas encontrados em testes que apresentam erros, também existem casos de testes assíncronos que algumas vezes podem passar e outras não, ou seja, às vezes é possível ler as respostas das operações assíncronas em tempo hábil e às vezes a execução do teste da mesma operação não será possível ser executada em um tempo hábil, o que irá gerar uma falha no teste. Como uma possível forma de solucionar o problema dos falsos positivos, muitos desenvolvedores ainda utilizam em alguns pontos dos seus testes a adição de atrasos explícitos (*Thread.sleep(timeout*)) para que as operações possam tentar receber e executar as respostas em tempo hábil. Infelizmente, esta solução não é sempre eficiente, porque além de poluir o código com diferentes esperas, que podem deixar o teste confuso e lento, vai ser muito difícil determinar o melhor tempo de espera. Caso o tempo seja muito pequeno, pode não ser possível receber e executar as operações de resposta em tempo hábil e se for muito grande pode deixar a execução do teste muito demorada, levando às vezes uma bateria de testes a durar horas ao invés de minutos.

Na Figura 1, podemos ver a execução de um teste que possui operação síncrona e assíncrona. Este teste é dividido em cinco partes (*PART*), a *PART 1* e a *PART 2* inserem os objetos *Object 1* e *Object 2* respectivamente de forma síncrona, já a *PART 3* insere o objeto *Object 3* de forma assíncrona. A *PART 4* verifica se todos os objetos foram inseridos (*Object size = 3*) e por fim, a *PART* 5 demonstra o momento em que o *Object 3* (inserido de forma assíncrona) foi realmente inserido na base de dados.

Inicialmente o teste executa as operações síncronas *PART1* e a *PART2* na qual inserem o O*bject 1 e* O*bject 2* respectivamente na base de dados (representado pelo cilindro visto na imagem). Como estas são operações síncronas, seus resultados são gerados antes que elas retornem. Já a *PART3,* que possui sua implementação assíncrona, inicia a inserção do O*bject 3* na base de dados, mas antes que a *PART3* seja concluída, ou seja, que *Object 3* seja inserido, a *PART4* inicia e conclui seu processamento, que verifica a quantidade de objetos inseridos na base. Como a *PART3* ainda está em execução, à verificação realizada na *PART 4* gera um erro, que na verdade foi um falso positivo.

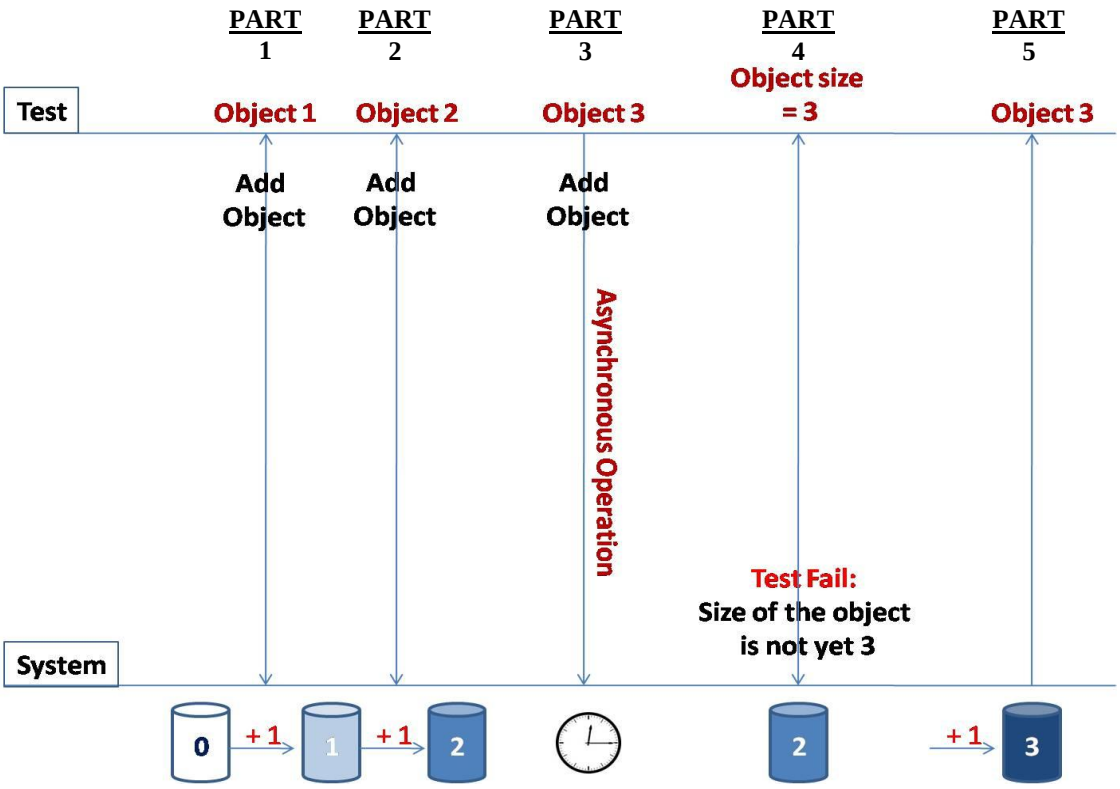

Figura 1 - Exemplo de falha em teste por verificação feita em momento não apropriado.

2.3 O ARCABOUÇO DE TESTES THREADCONTROL

Segundo Dantas (2010), ThreadControl é um arcabouço (*framework)* de teste criado seguindo as definições contidas na abordagem de teste de sistema assíncrono, chamada "*Thread Control For Tests*". Este arcabouço é baseado no monitoramento e controle das threads do sistema durante a execução de testes automáticos, tais como os implementados usando o *arcabouço* Junit [Massol 2004]. O ThreadControl oferece uma *API* com primitivas de teste para ajudar no desenvolvimento de testes automáticos para sistemas *multi-threaded*, evitando asserções realizadas cedo ou tarde demais, tais como as que ocorrem quando o testador prefere usar atrasos explícitos (Thread.sleep()) ou esperas ocupadas (atrasos explícitos dentro de loops (*wait()*) [Sousa et al. 2013]. Este arcabouço é implementado usando a linguagem AspectJ [Kickzales et al. 1997], uma extensão orientada a aspectos da linguagem Java, com o intuito de ser não-invasivo quanto à necessidade de alterações no sistema sendo testado.

As versões iniciais deste *arcabouço* estão disponíveis como código aberto em http://code.google.com/p/threadcontrol/ e na plataforma de versionamento de código GitHub, pelo seguinte link https://github.com/ayladebora/threadcontrol. O ThreadControl utiliza das

principais operações relacionadas a *Threads* da API (*Application Programming Interface*) versão 1.4 da linguagem de programação Java e de algumas das operações da API de concorrência introduzidas na versão 5 (pacote java.util.concurrent), o que demonstra que vários sistemas podem se beneficiar deste arcabouço para a preparação de seus testes mesmo sem a necessidade de alterar qualquer trecho de código do ThreadControl. Para que seja possível utilizarem seus testes algumas primitivas providas pelo *ThreadControl*, é necessário instalar o *plugin* intitulado *AspectJ Development Tools plugin* (AJDT) na IDE sendo usada. Além disso, é necessário baixar o pacote do código ThreadControl acessando http://code.google.com/p/threadcontrol/downloads/list ou realizando um clone em https://github.com/ayladebora/threadcontrol.git e incluir esse código no código-fonte no projeto. Segundo Dantas (2010), para que seja possível o uso do *arcabouço* ThreadControl, os seguintes métodos de teste (oferecidos por meio dos métodos da classe fachada chamada ThreadControl) são fornecidos para desenvolvedores de teste quando eles lidam com operações assíncronas em testes automáticos:

- **prepare**: É usado para especificar as configurações desejadas para o estado do sistema (em termos de estado das threads) pelas quais o teste deve esperar antes de fazer asserções (*assertions*). Estas configurações representam fases na execução do sistema,correspondentes aos estados de certas threads e questão passadas como parâmetro para a operação de prepare;
- **waitUntilStateIsReached:**Permite que as asserções sejam realizadas em um momento seguro, quando o estado especificado através da operação *prepare* tenha realmente sido atingido. Se uma thread monitorada tenta perturbar o sistema depois que este estado foi atingido, ela será bloqueada a fim de evitar uma asserção tardia. Quando esta operação é chamada, a thread de teste espera até que a aplicação alcance o estado esperado e em seguida, executa as asserções.
- **proceed:**Esta é a operação responsável por liberar o sistema para proceder com a sua execução normal. Qualquer thread que não foi autorizada a continuar porque asserções estavam sendo realizadas, serão nesse momento liberadas. Dessa forma, o teste pode terminar ou continuar com outras asserções.

O código 1, extraído de Sousa et al. (2013), demonstra o uso do ThreadControl em um caso de teste baseado no exemplo do produtor e do consumidor, utilizando o arcabouço de testes JUnit.

Código fonte 1: Caso de teste JUnit referente a um exemplo simples de Produtor/Consumidor.

```
1 public class AppTest{
2 @Test
3 public void testUsingJustAProducer(){
4   ThreadControltc = new ThreadControl();
5   Buffer buffer = new Buffer(Buffer.LIMIT);
6   Producer prod1 = new Producer(buffer);
7   tc.prepare(getSysConfigOfProducerWaiting());
8   new Thread(prod1).start();
9   tc.waitUntilStateIsReached();
10  assertEquals(Buffer.LIMIT,buffer.size());
11  tc.proceed();
12  producer.stop();
13 } // End of the test method;
14
15 public SystemConfiguration getSysConfigOfProducerWating() {
16  ThreadConfiguration config = new ThreadConfiguration(
17   Producer.class.getCanonicaName(), ThreadState.WAITING,
18   ThreadConfiguration.AT_LEAST_ONCE,
19   ThreadConfiguration.ALL_THREADS_TO_BE_IN_STATE);
20  ListOfThreadConfigurations sysConfig = new
21   ListOfThreadConfigurations();
22  sysConfig.addThreadConfiguration(config);
23  return sysConfig;
24 } // End of the method
```

Neste exemplo, a classe *Producer* é uma classe *Runnable* que gera mensagens para serem armazenados em um *buffer* de objetos de tempos em tempos, utilizando um `sleep` entre estes intervalos. Se o limite de *buffer* for atingido, o *Producer* espera até que um objeto consumidor remova um item do *buffer* antes de produzir mais mensagens.

O método de teste acima verifica se quando o *Runnable Producer* atinge o estado esperando (*WAITING*), o limite do buffer foi realmente atingido. Para que seja possível realizar este procedimento, uma instância da classe ThreadControl deve ser criada (linha 4) para permitir a chamada dos métodos de teste do ThreadControl para que as threads do teste sendo testado esperem até que o estado de espera tenha sido alcançado. Além disso, o teste deve inicializar os objetos referentes às classes a serem testadas. Neste teste, é preciso inicializar as classes *Buffer* e a classe *Producer* (linhas 5-6).

A seguir, o ThreadControl informa qual estado deverá ser atingido antes da asserção (assertion) ser executada. Isto é feito através da invocação do método *tc.prepare*, que recebe como parâmetro um objeto do tipo *SystemConfiguration*. A partir deste momento, o ThreadControl começa a monitorar o sistema sendo testado considerando a configuração especificada das *threads* do sistema. Neste caso, o estado esperado é o momento no qual todas as instâncias da classe *Producer* estão no estado *WAITING* e onde este estado já tenha sido atingido pelo menos uma vez (linhas 17-19) pelas threads do tipo *Producer*. Depois de indicar ao ThreadControl o estado de interesse para o teste, o teste atual inicia a *thread* que fará com que o *Producer* inicie sua execução (linha 8). Em seguida, o teste deve verificar se o limite do *buffer* é atingido quando o *Producer* atinge o estado *WAITING*. A fim de fazer o teste esperar até que este estado seja atingido antes da verificação, inserimos uma invocação ao método *waitUntilStateIsReached*() (linha 9). Isto bloqueia a thread do teste até que o estado esperado seja atingido. Então, a asserção do teste verifica se o buffer atingiu seu limite (linha 10).

Finalmente, após a realização da asserção do teste que dependia de ter sido alcançado o estado esperado, o método *proceed*() do ThreadControl é chamado (linha 11), a fim de liberar qualquer *thread* que poderia ter sido previamente bloqueada ao tentar mudar o seu estado enquanto asserções estariam sendo executadas.

O método mostrado entre as linhas 15-24 é usado para criar um objeto do tipo `SystemConfiguration`, uma interface que representa as configurações de estados esperados determinadas pelo desenvolvedor do teste. O objeto criado neste método é uma instância da classe *ListOfThreadConfigurations*, uma implementação da interface *SystemConfiguration*. Esta classe apresenta o método `addThreadConfiguration` (linha 22), que recebe como parâmetro instâncias da classe `ThreadConfiguration` (*config*). Cada `ThreadConfiguration` representa a configuração de um certo tipo de thread pela qual se quer esperar (ex: todas as threads de um tipo X no estado WAITING). No construtor da classe `ThreadConfiguration` ilustrado no Código 1 são passados quatro parâmetros (linhas 17-19). O primeiro parâmetro (linha 17)é o nome da classe que representa o tipo de *Runnable* ou *Thread*(*Producer.class.getCanonicaName()*) cujo estado pelo qual se deseja esperar está sendo especificado. O segundo parâmetro é uma constante do tipo `ThreadState` e que representa o estado em que deve estar aquele tipo de thread. O terceiro parâmetro é o número de vezes em que o estado especificado pela constante da classe `ThreadState` deve ter sido atingido por instâncias da classe passada no primeiro parâmetro (*Producer.class.getCanonicaName()).* Por fim, no quarto parâmetro é passada a

quantidade de instâncias ativas da classe passada no primeiro parâmetro (*Producer.class.getCanonicaName()),* que deverá estar no estado especificado. Ao invés de especificar um certo número, pode-se também passar no terceiro e quarto parâmetro as constantes *AT_LEAST_ONCE* e *ALL_THREADS_TO_BE_IN_STATE* da classe *ThreadConfiguration*, e que informam ao ThreadControl que ele deverá aguardar que pelo menos uma vez as threads daquele tipo tenham alcançado o estado especificado e que todas estejam naquele estado para que o teste possa prosseguir com as asserções.

2.4 PRINCIPAIS TRABALHOS RELACIONADOS

Para identificar as principais obras relacionados a este trabalho de conclusão de curso, foram realizadas pesquisas exaustivas em alguns dos principais engenhos acadêmicos, entre eles, o Google Acadêmico[1], utilizando as chaves de busca: "*Test Asynchronous Systems OR Software -chip -chips -circuit –circuits*" (s/aspas), vale ressaltar que o trecho da chave "*-chip -chips -circuit –circuits*" foi utilizado para diminuir a ocorrência de trabalhos relacionados à Engenharia elétrica que contém as palavras "*chips"* e "*circuits"*, esta busca foi utilizada com ênfase em "qualquer parte do artigo". Também foram utilizadas as bibliotecas digitais *ACM Digital Library*[2] e IEEE[3] utilizando a chave de busca "Test Asynchronous Systems".

Utilizando as chaves de busca acima, foram encontradas inúmeras obras relacionados a teste de software, porém uma pequena parte se tratava de teste de software assíncrono. Entretanto, foi possível selecionar trabalhos que possuem uma maior ênfase nos principais assuntos discutidos nessa pesquisa, ou seja, dificuldades de se testar sistemas assíncronos e o uso do *arcabouço* ThreadControl. Ao final da pesquisa, foram destacados dois trabalhos que serão discutidos a seguir.

O artigo de Salas e Krishnan (2009), intitulado "Automated Software Testing of Asynchronous Systems" descreve uma abordagem, que possui o objetivo geral de permitir que ferramentas de testes possam realizar testes assíncronos através de modelos. Embora esta abordagem já seja utilizada em algumas ferramentas de teste assíncrono, o trabalho dos autores foca em testes caixa-preta (*black-box*), isto é, lida principalmente com interfaces disponíveis ao invés do funcionamento interno dos sistema. Para isso, segundo Salas e Krishnan, é preciso modelar os sistemas distribuídos assíncronos utilizando de estruturas baseadas em eventos, que também são demonstradas em seu trabalho. Este trabalho também

[1] Google Acadêmico: http://scholar.google.com.br/
[2] ACM DL: http://dl.am.org/
[3] IEEE: http://ieeexplorer.ieee.org/

tem o objetivo de estender tais ferramentas existentes para implementar o conhecimento teórico sobre como lidar com a comunicação assíncrona expresso em modelos.

O artigo de Dantas et. al (2008), intitulado “Obtaining Trustworthy Test Results in Multi-threaded Systems” descreve o arcabouço ThreadControl, focando principalmente na abordagem para teste que ele suporta, nos detalhes da sua implementação e em um estudo de caso para demonstrar que o arcabouço evita falsos positivos.

Analisando as obras apresentadas e relacionando-as com o presente trabalho, foi observado que apesar de possuírem características similares, possuem objetivos distintos. O trabalho de Sales e Krishnan (2009) tem como objetivo específico utilizar a abordagem em testes de caixa-preta, o que difere deste trabalho que apresenta abordagens que podem ser utilizadas em testes de caixa-branca (white-box), os quais lidam com o funcionamento interno do sistema. O presente trabalho de conclusão de curso se diferencia do trabalho de Dantas (2008), o qual se focava em detalhes internos e funcionalidades do ThreadControl, por se concentrar mais nos aspectos práticos do uso do arcabouço em sistemas de informação assíncronos e nas observações da quantidade de falsos positivos e tempo de execução dos casos de teste comparando com a abordagem de atrasos explícitos, que pode ser encontrada na prática em testes de sistemas reais.

3 O SISTEMA DE INFORMAÇÃO MYSIMPLECRM

Como forma de comparar por meio de estudo de caso o uso do *arcabouço* ThreadControl com atrasos explícitos no teste de sistemas de informação assíncronos, foi desenvolvido um sistema totalmente assíncrono, distribuído e que se comunica via *WebService* REST chamado MySimpleCRM. Os seguintes tópicos desta seção detalham esse sistema com a intenção de que outros pesquisadores possam evoluir suas funcionalidades ou adaptá-lo às suas necessidades. Para o desenvolvimento do MySimpleCRM foram utilizados arcabouços e bibliotecas utilizadas atualmente na indústria de software, já que nosso objetivo é demonstrar o comparativo entre formas de testar sistemas assíncronos em um ambiente controlável, mas que seja o mais próximo possível de um sistema real.

O MySimpleCRM pode ser encontrado na plataforma de versionamento de código GitHub, pelo seguinte link: https://github.com/diegosousa/mysimplecrm.

3.1 VISÃO GERAL DO MYSIMPLECRM

O sistema utilizado para o estudo de caso apresentado neste trabalho é um sistema de informação simples para a gestão de relacionamento com clientes (CRM), que foi chamado MySimpleCRM. Ele basicamente apresenta as seguintes operações assíncronas:

- Gerenciamento de clientes, produtos e promoções;
- Gerenciamento de informações de compras e promoções;
- Envio automático de e-mails aos clientes informando sobre as compras realizadas;
- Envio automático de e-mails desejando parabéns aos clientes aniversariantes;
- Envio automático de e-mails sobre promoções de acordo com o perfil de cada cliente.

MySimpleCRM é um sistema web. Sua arquitetura é dividida em três camadas: Camada de interface do usuário (*User Interface Layer*), camada de serviços de negócio (*Services Business Layer*) e Camada de dados (*Data layer*). Na Figura 2 é apresentado o diagrama arquitetural, no qual a *User Interface Layer* é representada por uma aplicação cliente responsável por consumir os recursos do MySimpleCRM. Essa aplicação poderá ser um browser ou qualquer outro serviço que utiliza a interface REST disponibilizada pelo MySimpleCRM. A *Service Business Layer* contém todo o código responsável por receber e processar os dados da *User Interface Layer*, como também interagir com a *Data layer* para armazenar os dados. Dentro da *Services Business Layer* foi definida uma subcamada chamada "*Logic Layer"*, que concentra toda a lógica de negócio da aplicação, e uma "*Asynchronous Layer*" responsável pela execução das operações de forma assíncrona usando um pool de threads e construída baseada no padrão de projeto Command [Gamma et al. 1994]. A lógica do sistema é acessada através da classe da fachada, que interage com a camada assíncrona e que segue o padrão de projeto Facade [Gamma et al. 1994].

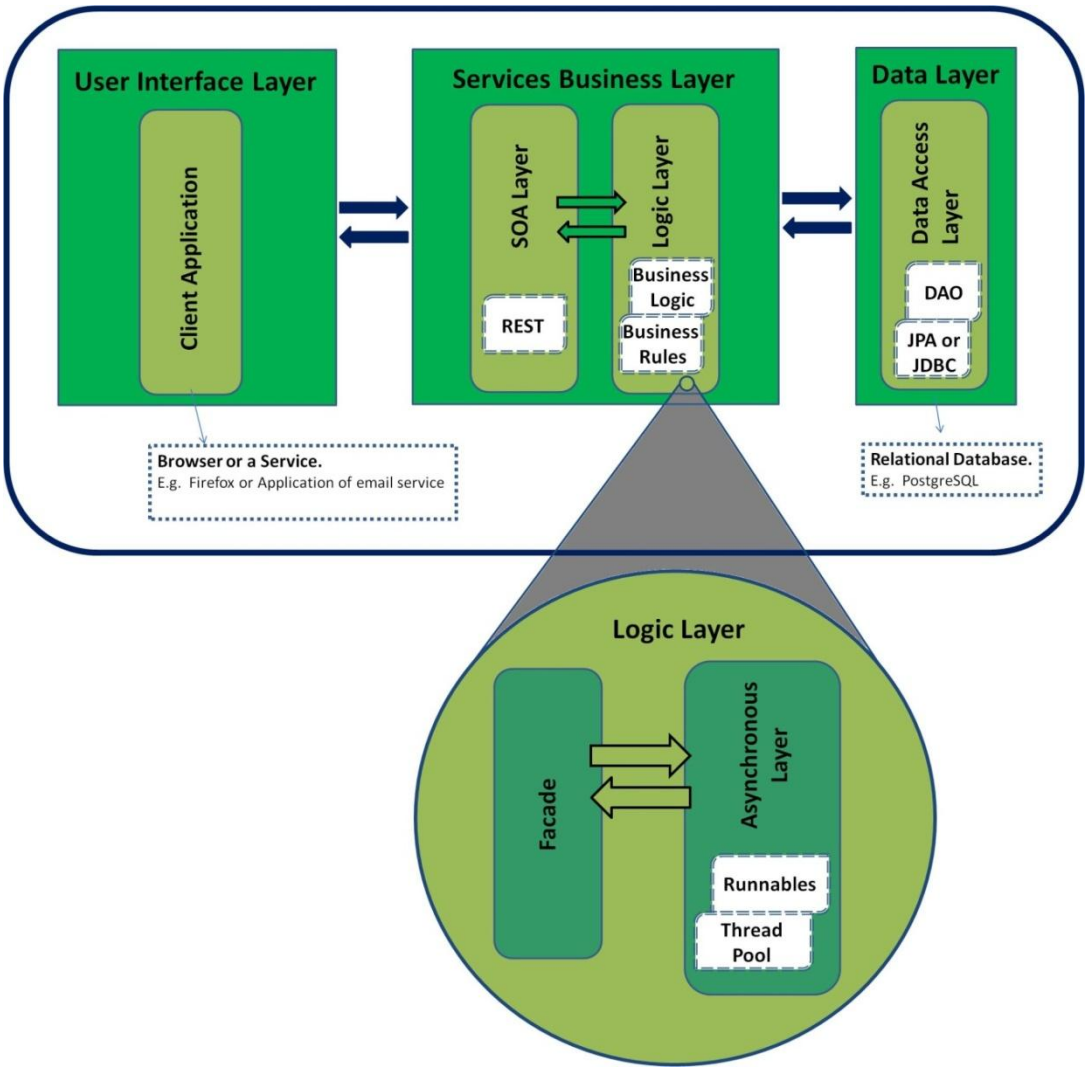

Figura 2 - Diagrama Arquitetural

O MySimpleCRM segue o estilo de arquitetura de software *SOA* (*Service-Oriented Architecture*) e seus serviços são oferecidos através de interfaces genéricas fornecidas via *REST* (*Representational State Transfer*), que usa mensagens *HTTP* (*Hypertext Transfer Protocol*) para comunicação de dados. Usando essa abordagem, podemos obter módulos desacoplados, facilidade de reutilização e escalabilidade, entre outras vantagens. A fim de compreender melhor o sistema MySimpleCRM, são ilustrados na Figura 3 algumas das principais classes relacionadas à operação de adição de uma nova compra (*addPurchase()*).

Com o intuito de auxiliar o entendimento da comunicação existente entre as classes do sistema, foram inseridas no diagrama abaixo apenas associações binárias e suas navegabilidades, que identifica o sentido em que as informações são transmitidas entre os objetos das classes envolvidas, ou seja, o sentido em que os métodos poderão ser disparados.

Também foi utilizado o relacionamento de dependência, que identifica o grau de dependência de uma classe em relação à outra. [Guedes 2009].

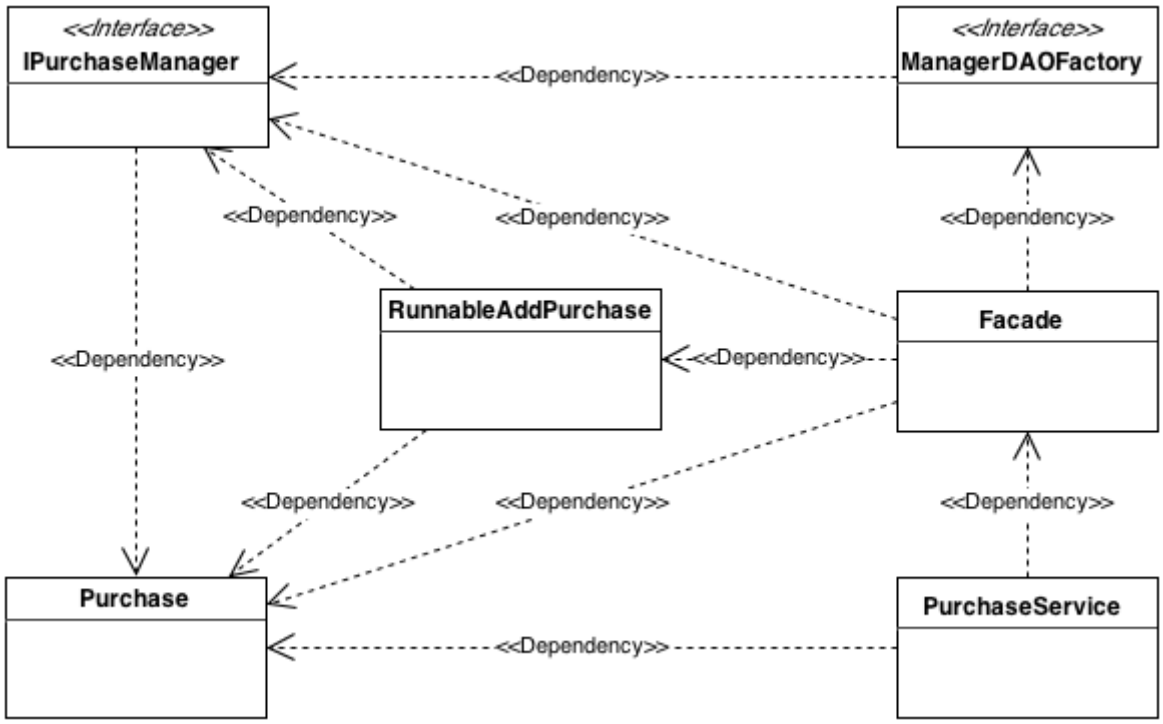

Figura 3 - Diagrama de classe com as classes envolvidas na funcionalidade addPurchase()

A classe *PurchaseService* é uma classe da camada *SOA* que é responsável pela comunicação entre a *Logic Layer* e a *User Interface Layer*. A classe *Facade* fornece uma interface simplificada para acessar as funcionalidades do sistema. A classe *Purchase* é a entidade que representa a compra de um cliente. O MySimpleCRM utiliza o padrão *DAO* (*Data Access Object*) que possui a finalidade de separar as regras de negócio do sistema das regras de acesso ao banco de dados. O *DAO* fornece interfaces, tais como a interface IpurchaseManager ilustrada na Figura 3. Esta interface, em particular, representa as operações relativas à persistência da entidade *Purchase*. Alguns exemplos de implementações desta interface são as classes PurchaseDAOJPAe a PurchaseDAOFILE, que utiliza *JPA* (*Java Persistence API*) e simples arquivos como mecanismos de persistência, respectivamente. O padrão de projeto *Abstract Factory* [Gamma etal. 1994] é utilizado no sistema, a fim de construir estas diferentes implementações. Por exemplo, quando o mecanismo de persistência utilizado é o JPA, o sistema utiliza uma instância da classe *ManagerDAOFactoryJPA*, que implementa a Interface *ManagerDAOFactory* (mostrada na Figura 3), para construir o objeto *IpurchaseManager* específico.

A fim de oferecer as suas operações de forma assíncrona, o MySimpleCRM utiliza algumas classes que implementam a interface *Runnable* da linguagem *Java*, como por exemplo, a classe *RunnableAddPurchase* também mostrada na Figura 3. Quando este objetos iniciam a sua execução, ou seja, quando o seu método `run()` for executado, estes irão utilizar o objeto *IpurchaseManager* (tal como um instância da classe *PurchaseDAOJPA*) para realmente executar a funcionalidade de compra.

3.2 MODELO DE DADOS

Esta seção é destinada a descrição de todo o funcionamento do banco de dados do MySimpleCRM, como uma forma de documentar o sistema utilizado no estudo de caso. O primeiro tópico é demonstrado o modelo relacional com seus elementos, chaves candidatas e estrangeiras. No segundo tópico é demonstrado o modelo entidade relacionamento, onde é possível observar todo o modelo de dados com alto nível de abstração. Por fim, no terceiro tópico desta seção é apresentado o dicionário de dados deste sistema, onde é possível visualizar toda a descrição e definição de toda a informação usada na construção deste sistema.

3.2.1 MODELO ENTIDADE RELACIONAMENTO

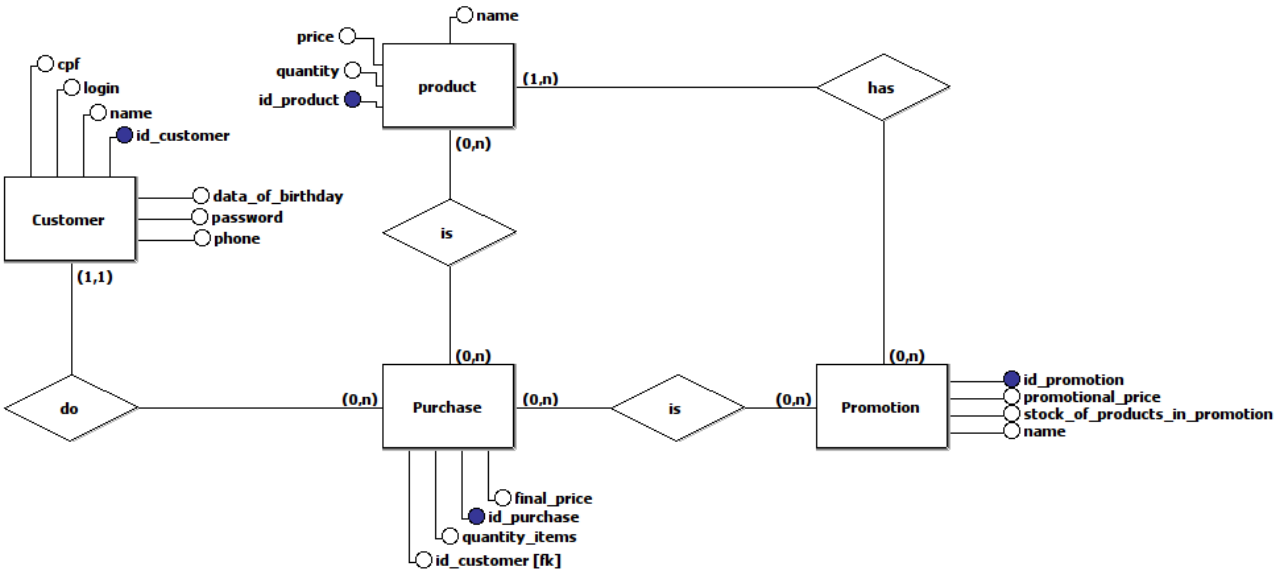

Figura 4 - Diagrama Entidade Relacionamento do MySimpleCRM.

3.2.2 MODELO RELACIONAL

- **customers** (id_customer[pk], date_of_birthday, cpf, is_active, login, name, password, phone)
- **products** (id_product[pk], is_active, name, price, quantity)
- **promotions** (id_promotion[pk], is_active, name, promotional_price, stock_of_products_in_promotion)
- **purchases** (id_purchase[pk], finalprice, id_customer[fk], quantity_items)
- **products_in_promotion** (id_promotion[fk], id_product[fk])
- **purchased_products** (id_purchase[fk], quantity_product, id_products[fk])
- **purchased_promotions** (id_purchased[fk], quantity_promotions, id_promotions[fk])

3.2.3 DICIONÁRIO DE DADOS

TABLE CUSTOMERS

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id_customer	NUMERIC	NÃO	Customer Identifier Code	Auto-increment	X		
Name	TEXT	NÃO	Name of Customer				
Phone	ALPHANUMERIC	SIM	Customer Phone				
Login	ALPHANUMERIC	NÃO	Customer Login				X
Password	ALPHANUMERIC	NÃO	Password of the Customer				
date_of_birth	DATA	NÃO	Date of Birth from Client				
Cpf	ALPHANUMERIC	NÃO	Cpf of Customer				X
isActive	BOOLEAN	NÃO	Indicates if the object was deleted	If true the customer is active, if false the customer is disabled			

Tabela 1 – Referente à entidade Customer

TABLE PRODUCTS

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id _product	NUMERIC	NAO	Product Identifier Code	Auto-increment	X		
name	TEXT	NAO	Name Product				X
Price	NUMERIC	NAO	Product Price				
quantity	NUMERIC	NAO	Quantity of products in stock				

isActive	BOOLEAN	NÃO	Indicates if the object was deleted	If true the product is active, if false the product is disabled			

Tabela 2 - Referente à entidade Product

TABLE PURCHASES

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id_purchase	NUMERIC	NAO	Purchase Identifier Code	Auto-increment	X		
id_customer	NUMERIC	NAO	Customer Identifier Code	Table Customer		X	
finalPrice	NUMERIC	NAO	Final Price of Purchase	Sum of the prices of items			

Tabela 3 - Referente à entidade Purchase

TABLE PROMOTIONS

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id_promotion	NUMERIC	NÃO	Promotion Identifier Code	Auto-increment	X		
name	TEXT	NÃO	Promotion Name				X
promotional_price	NUMERIC	NÃO	PromotionPrice	Less than the price of the product			
stock_of_products_in_promotion	NUMERIC	NÃO	Quantity of product in stock.	Less or equals quantity of product			
isActive	BOOLEAN	NÃO	Indicates if the object was deleted	If true the promotion is active, if false the promotion is disabled			

Tabela 4 - Referente à entidade Promotion

TABLE PRODUCTS_IN_PROMOTION

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id_promotion	NUMERIC	NAO	Promotion Identifier Code	Auto-increment		X	
id_product	NUMERIC	NAO	Product Identifier Code	Table Product		X	

Tabela 5 - Referente à junção entre a tabela Product e a tabela Promotion

TABLE PURCHASED_PROMOTIONS

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN

id_purchased	NUMERIC	NAO	Purchased Identifier Code	Auto-increment		X	
id_promotion	NUMERIC	NAO	Promotion Identifier Code	Table Promotions		X	
quantity_promotion	NUMERIC	NÃO	Quantity of promotions sold in a purchase				

Tabela 6 - Referente à junção entre a tabela Purchase e a tabela Promotion

TABLE PURCHASED_PRODUCTS

COLUMN	TYPE	NULL	DESCRIPTION	DOMAIN	ID		
					PK	FK	CAN
id_purchased	NUMERIC	NAO	Purchased Identifier Code	Auto-increment		X	
id_product	NUMERIC	NAO	Product Identifier Code	Table Product		X	
quantity_product	NUMERIC	NÃO	Quantity of products sold in a purchase				

Tabela 7 - Referente à junção entre a tabela Purchase e a Product

4 ESTUDO DE CASO: ANÁLISE DE ALTERNATIVAS PARA TESTAR O SISTEMA DE INFORMAÇÃO ASSÍNCRONO MYSIMPLECRM

Neste capitulo é apresentado o estudo de caso relativo ao teste do sistema de informação assíncrono MySimpleCRM utilizando duas alternativas de se testar sistemas assíncronos, levantando vantagens e desvantagens de cada uma e lições aprendidas.

4.1 TESTANDO O MYSIMPLECRM COM ATRASOS EXPLICITOS

Foi exercitada em testes a utilização do MySimpleCRM durante a compra de um produto e utilizando-se como estratégia de espera nos testes os atrasos explícitos (*Thread.sleep(timeout)*), seguindo os passos abaixo:

1. Criar uma instância da classe Fachada do MySimpleCRM para se exercitar as operações do sistema.
2. Registrar um novo cliente chamado de "customer1".
3. Esperar até que o intervalo (*timeout*) explicitado no atraso explicito(*sleep*) tenha se passado.
4. Verificar se o cliente registrado pode ser encontrado no sistema.
5. Registrar um novo produto chamado "product1".
6. Esperar pelo tempo definido no atraso explícito.
7. Verificar se o novo produto pode ser encontrado no sistema.

8. Registrar uma nova compra identificada por “purchase1” em relação ao “product1” para o cliente “customer1”
9. Esperar pelo tempo definido no atraso explicito.
10. Verificar se a compra chamada "purchase1" foi corretamente incluída no sistema.

O código a seguir ilustra esses passos em um teste JUnit utilizando *Thread.sleep(timeout);*

Código fonte 2: Caso de teste Junit referente aos passos iniciais de uma compra utilizando Thread.sleep().

```
1@Test
2 public void testAddPurchase() {
3
4  facade = Facade.getInstance(
5    new ManagerDaoFactoryJPA()); //STEP 1
6
7 /*----------Starting operations of the customer-------*/
8
9  takerClientList = new LinkedBlockingQueue<Customer>();
10  Customer customerAuxOne = null;
11 Customer customer1 = new Customer("Diego", "07278910112",
12  "3422-1010","diego.sousa@dce.ufpb.br", "S3cr3t",
13   18, 9, 1988);
14
15 facade.addCustomer(customer1); // STEP 2
16
17 try {
18  Thread.sleep(3000); // STEP 3
19 } catch (InterruptedException ie) {
20  ie.printStackTrace();
21 }
22
23 facade.searchCustomerByCpf("07278910112",
24   takerClientList); //STEP 4
25
26 try {
27  customerAuxOne = takerClientList.take(); //this remains
28 } catch(InterruptedException e) {         //synchronous due
29  e.printStackTrace();                     //to the use of a
30 }                                         //BlockingQueue.
31
```

```
32 assertEquals("Diego", customerAuxOne.getName());
33 assertEquals("diego.sousa@dce.ufpb.br",
34     customerAuxOne.getLogin());
35
36 /*--------------End of customer operations---------------*/
37
38 /*----------Starting operations of the product-----------*/
39
40 takerProductList = new LinkedBlockingQueue<Product>();
41 Product productAuxOne = null;
42 Product product1 = new Product("IPod", 1200.00, 100);
43
44 facade.addProduct(product1); // STEP 5
45
46 try {
47   Thread.sleep(3000); // STEP 6
48 } catch (InterruptedException ie) {
49   ie.printStackTrace();
50 }
51
52 facade.searchProductByName("IPod",
53     takerProductList); // STEP 7
54
55 try {
56   productAuxOne = takerProductList.take();
57 } catch (InterruptedException e) {
58   e.printStackTrace();
59 }
60
61 assertEquals("IPod", productAuxOne.getName());
62 assertFalse(productAuxOne.getPrice() == 1200.0);
63// ...
64   /*--------------End of Product operations--------------
*/
65
66 }
```

O teste com atrasos explícitos invocou as operações do sistema através da classe *Facade*, que possui todas as operações assíncronas, assim também foi necessário que a *Thread* do teste JUnit aguardasse algum tempo antes de executar o código responsável pelas asserções (*assertions*).

Nesse caso, a abordagem utilizada para fazer com que a Thread do teste aguardasse antes de executar as asserções foram os atrasos explícitos de 3 segundos, o que nem sempre pode ser adequado para esperar antes de fazer as asserções. Nesse exemplo de código foi utilizado um atraso de 3000 milissegundos nos passos (*STEPS*) 3 e 6, e não há nenhuma garantia de que este tempo será suficiente para que as operações assíncronas(como `addCustomer` e `addProduct`) concluam toda a sua execução antes do teste executar as pesquisas utilizadas nas asserções. Por outro lado, 3000 milissegundos poderá ser um tempo demasiado grande o que deixará a máquina de teste ociosa em vários momentos, gerando um maior tempo da execução da bateria de testes.

O código descrito acima é correspondente aos passos (*STEPS*) 1-7 anteriormente mencionados. As etapas 8-10 representam a finalização da operação de compra e não são aqui explicados por serem semelhantes aos passos 2-4.

4.2 TESTANDO O MYSIMPLECRM COM O THREADCONTROL

De modo a testar um cenário em relação à utilização do MySimpleCRM durante a compra de um produto, podemos usar o *Arcabouço* JUnit e o ThreadControl seguindo os passos abaixo:

1. Criar uma instância de ThreadControl (TC), a fim de utilizar as primitivas de fazer o teste esperar até que certo estado seja atingido antes de executar alguma asserção.
2. Criar uma instância da classe Fachada do MySimpleCRM para se exercitar as operações do sistema.
3. Registrar um novo cliente chamado de "customer1".
4. Esperar até que a *Thread* responsável pela adição do cliente tenha terminado o seu trabalho.
5. Verificar se o cliente registrado pode ser encontrado no sistema.
6. Registrar um novo produto chamado "product1".
7. Esperar até que a *Thread* responsável pela adição de produtos tenha terminado o seu trabalho.
8. Verificar se o novo produto pode ser encontrado no sistema.
9. Registrar uma nova compra identificada por “purchase1” em relação ao “product1” para o cliente “customer1”
10. Esperar até que a *Thread* responsável pela adição da compra tenha terminado o seu trabalho.
11. Verificar se a compra chamada "purchase1" foi corretamente incluída no sistema.

O código a seguir ilustra esses passos em um teste JUnit utilizando o *arcabouço* ThreadControl.

Código fonte 3: Caso de teste Junit referente aos passos iniciais de uma compra utilizando o ThreadControl.

```
1 @Test
2 public void testAddPurchase() {
3
4  threadControl = new ThreadControl(); //STEP 1
5 facade = Facade.getInstance();//STEP 2
6
7 /*----------Starting operations of the customer----------*/
8
9 takerClientList = new LinkedBlockingQueue<Customer>();
10 CustomercustomerAuxOne = null;
11 Customer customer1 = new Customer(“Diego”, “07278910112”,
12   “3422-1010”, “diego.sousa@dce.ufpb.br”, “S3cr3t”,
13    18, 11, 1988);
14
15 //Add Customer(previously specifying the test waiting
16 condition)
17 threadControl.prepare(getAddCustomerFinishedState(1));
18 facade.addCustomer(customer1); //STEP 3
19
20 threadControl.waitUntilStateIsReached(); //STEP 4
21 //getting customer by CPF
22 facade.searchCustomerByCpf(“07278910112”,
23  takerClientList); //STEP 5
24
25 try {                                    //thisremains
26customerAuxOne = takerClientList.take() //synchronous due
27 } catch (InterruptedException e) {       //to the use of a
28e.printStackTrace();                      //BlockingQueue.
29 }
30
31 assertEquals(“Diego”, customerAuxOne.getName());
32 assertEquals(“diego.sousa@dce.ufpb.br”,
33threadControl.proceed();
34 /*------------End of customer operations-----------------*/
35
```

```
takerProductList = new LinkedBlockingQueue<Product>();
Product productAuxOne = null;
Product product1 = new Product (“IPod”, 1200.00, 100);

//Add Product
ThreadControl.prepare(getAddProductFinishedState(1));
facade.addProduct(product1); //STEP 6

threadControl.waitUntilStateIsReached(); //STEP 7

//Searching by product name
facade.searchProductByName(“IPod”,
takerProductList);//STEP 8

try{
productAuxOne = takerProductList.take();
} catch (InterruptedException e){
e.printStackTrace();
}

assertEquals(“IPod”, productAuxOne.getName());
assertTrue(productAuxOne.getPrice() == 1200.0);
threadControl.proceed();//…
/*------------End of Product operations------------------*/
```

Como as operações do sistema invocadas através da classe Fachada são assíncronas, a *Thread* do teste deve esperar algum tempo antes de executar o código responsável pelas asserções (*assertions*). Para evitar falsos positivos, pelo uso de atrasos explícitos inadequados (como chamadas a *Thread.sleep*), com tempos de espera insuficientes, foi utilizado aqui o arcabouço de testes ThreadControl, como ilustrado pelo trecho de código comentado acima. Este código corresponde aos passos (*STEPS*) 1-8 anteriormente mencionados. As etapas 9-11 representam a finalização da operação de compra e não são aqui explicados por serem semelhantes aos passos 3-5.

A fim de definir o estado esperado para o sistema no momento em que as asserções são executadas, o desenvolvedor do teste deve construir um objeto do tipo *SystemConfiguration*. À medida que o mesmo estado de espera pode ser utilizado para vários testes, é interessante o desenvolvimento de métodos para a criação de tais objetos, como o método *getAddCustomerFinishedState*, apresentado no seguinte trecho de código.

Código fonte 4: Método auxiliar para ser utilizado pelo caso de teste JUnit referente à invocação da operação addCustomer().

```
1 public SystemConfiguration getAddCustomerFinishedState(
2  int timesToBeInState){
3  ThreadConfiguration config = new ThreadConfiguration(
4    RunnableAddCustomer.class.getCanonicalName(),
5    ThreadState.FINISHED, timesToBeInState);
6  ListOfThreadConfigurations sysConfig = new
7  ListOfThreadConfigurations();
8  sysConfig.addThreadConfiguration(config);
9  return sysConfig;
10 }
```

Este método cria uma instância da classe *ListOfThreadConfigurations*, que é uma implementação da interface *SystemConfiguration*. A configuração representada pelo presente objeto corresponde a um estado no qual as instâncias da classe *RunnableAddCustomer* devem atingir o estado *FINISHED* e o número de vezes que os Runnables dessa classe deve alcançar este estado, representado pela variável *timesToBeInState*. No código de teste, o valor utilizado para este parâmetro foi de "1" (linha 17 do código fonte *3*), o que significa que o teste deve esperar até que uma instância do *RunnableAddCustomer* termine sua execução.

4.3 ANÁLISE DOS TEMPOS DE EXECUÇÃO E DA OCORRÊNCIA DE FALSOS POSITIVOS

Para a realização do estudo de caso, foram implementadas versões de teste do MySimpleCRM com atrasos explícitos (*Thread.sleep(timeout)*) de 3000, 500 e 250 milissegundos (ms) respectivamente e uma versão com o ThreadControl, para que assim fosse possível obter dados que facilitassem o estudo comparativo. Vale ressaltar que a métrica de escolha da quantidade de valores (três), surgiu da observação dos resultados analisados de execução de testes em diferentes máquinas, sendo constatado que para o primeiro dificilmente se obtinha falsos positivos, com o segundo se obtinha em algumas máquinas falsos positivos e que com o terceiro algumas máquinas nunca obtinham sucesso em seus testes e outras sim. É evidente, porém, que inúmeros valores poderiam ser utilizados para deixar o estudo mais completo, mas por restrições de tempo se considerou apenas estes. Para que tenhamos confiança nos

resultados, é demonstrado o cálculo do intervalo de confiança da média obtida para o tempo de execução dos testes. O objetivo deste estudo é responder às seguintes questões de pesquisa:

i. Em média, quanto tempo em milissegundos é gasto para executar as versões de teste do MySimpleCRM utilizando o ThreadControl e versões com `Thread.sleep(timeout)` de 250ms, 500ms e 3000ms com 95% de confiança no calculo da média?
ii. Qual a quantidade de falsos positivos em cada abordagem analisada?

Para responder a estas perguntas, a análise será realizada na seguinte ordem:

a) Obter o tempo total de 30 iterações (em milissegundos), já que esta é a quantidade mínima para se utilizar da tabela normal padrão (Tabela Z).
b) Calcular o intervalo de confiança da média com as premissas pedidas no enunciado da questão i.
c) Calcular o tamanho da amostra com as premissas pedidas no enunciado da questão i.
d) Caso seja necessário, obter o tempo total das iterações faltantes (em milissegundos).
e) Calcular a média do tamanho da amostra.

4.3.1 OBTENDO O TEMPO TOTAL DAS ITERAÇÕES REALIZADAS.

Inicialmente foram feitas 30 iteração com o testes do MySimpleCRM utilizando os Script Shell apresentados no apêndice A deste trabalho, para que fosse possível automatizar ainda mais a execução destes testes.

Os tempos de execução dos testes em milissegundos (ms) podem ser vistos na Tabela 8. A primeira coluna representa uma identificação de cada iteração dos testes. Na segunda, terceira e quarta coluna, estão representados os testes com `Thread.sleep(timeout)` de 3000, 500 e 250 milissegundos respectivamente. Por fim, na quinta coluna é apresentado o tempo total do teste com o ThreadControl. Nas duas últimas linhas da Tabela 8 são apresentadas a média ($\overline{x}$) aritmética de cada caso de teste e o desvio padrão (s), que serão utilizados para o cálculo do intervalo de confiança da média, como também, para identificar o tamanho da amostra a utilizar considerando a margem de erro esperada para se atingir 95% de confiança nos resultados obtidos.

ID	Sleep 3000 ms	Sleep 500 ms	Sleep 250 ms	ThreadControl
# 1	14685	4572	3596	3137
# 2	14689	4677	3672	3159
# 3	14711	4651	3667	3193
# 4	14656	4723	3627	3097
# 5	14631	4674	3713	3129
# 6	14071	4631	3585	3219
# 7	14066	4634	3632	3164
# 8	14651	4632	3719	3094
# 9	14643	4666	3628	3102
# 10	14621	4633	3603	3161
# 11	14686	4662	3638	3119
# 12	14699	4601	3654	3114
# 13	14603	4668	3599	3192
# 14	14673	4706	3653	3196
# 15	14066	4669	3688	3155
# 16	14626	4661	3657	3138
# 17	14678	4073	3679	3175
# 18	14064	4675	3647	3185
# 19	14066	4609	3704	3018
# 20	14697	4064	3646	3116
# 21	14659	4681	3617	3172
# 22	14683	4622	3651	3075
# 23	14685	4647	3572	3119
# 24	14062	4599	3651	3198
# 25	14067	4593	3007	3133
# 26	14615	4662	3698	3104
# 27	14726	4635	3687	3184

# 28	14673	4647	3659	3011
# 29	14661	4621	3658	3173
# 30	14673	4625	3599	3109
$\bar{x}$	**14526**	**4607**	**3627**	**3138**
s	**260**	**150**	**123**	**50**

Tabela 8 - Tempo de execução das amostras para o cálculo do intervalo de confiança da média

4.3.2 CALCULANDO O INTERVALO DE CONFIANÇA DA MÉDIA.

Para este cálculo, inicialmente é preciso encontrar o valor correspondente a utilizar da tabela normal padrão (Tabela Z), sendo possível realizar o cálculo do intervalo de confiança da média, do qual também resulta a margem de erro máxima. Para isso é utilizada a equação 1 abaixo:

$$z = \frac{\mathbf{1} - [\boldsymbol{nível\ de\ confiança\ desejada}(\%)/\mathbf{100}]}{\mathbf{2}}$$

***Equação 1**. Formula utilizada para encontrar o valor da tabela Z.*

- Utilizando a equação acima, obtivemos o seguinte resultado:

$$z = \frac{1 - [95 / 100]}{2}$$

$$z = \frac{1 - 0{,}95}{2}$$

$z = \frac{0{,}05}{2}$ => **0,025** ≅ (equivalente) na tabela Z a **1,96**.

- Para calcular o intervalo de confiança de cada caso de teste é utilizada a equação abaixo, onde o "$\bar{x}$" representa a média de tempo obtida nos casos de teste e o "s" corresponde ao desvio padrão (ambos na Tabela 8), o Z é o valor correspondente a tabela normal padrão (Tabela Z), descrito no início deste tópico e o "n" é a quantidade de iterações de teste realizada (neste caso trinta).

$$\bar{\boldsymbol{x}} \pm \boldsymbol{Z} * \frac{\boldsymbol{s}}{\sqrt{\boldsymbol{n}}}$$

***Equação2**. Formula utilizada para encontrar o intervalo de confiança da média.*

Na Tabela 9 abaixo, é demonstrado o cálculo do intervalo de confiança da média (IM) dos casos de testes exercitados neste trabalho. Para cada caso de teste foi utilizada a equação 2 para que fossem encontrados tais intervalos com 95% de confiança nos resultados.

Sleep 3000 ms	**Sleep 500 ms**
$IM = 14526 \pm 1,96 * \frac{260}{\sqrt{30}}$ $IM = 14526 \pm 93$ $\boldsymbol{IM = (14433\ ;\ 14619)}$	$IM = 4607 \pm 1{,}96 * \frac{150}{\sqrt{30}}$ $IM = 4607 \pm 54$ $\boldsymbol{IM = (4553\ ;4661)}$
Sleep 250 ms	**ThreadControl**
$IM = 3627 \pm 1,96 * \frac{123}{\sqrt{30}}$ $IM = 3627 \pm 44$ $\boldsymbol{IM = (3583\ ;3671)}$	$IM = 3138 \pm 1,96 * \frac{50}{\sqrt{30}}$ $IM = 3138 \pm 18$ $\boldsymbol{IM = (3120\ ;3156)}$

Tabela 9 - Calculo do intervalo de confiança dos casos de teste.

4.3.3 CÁLCULO DO TAMANHO DA AMOSTRA.

Para calcular o tamanho da amostra é utiliza da equação 3 abaixo. O "E" representa a margem de erro, ou seja, a diferença máxima provável (com probabilidade 1-α) entre a média amostral observada e a verdadeira média da população.Para calcular o E,utilizamos a seguinte equação,$\boldsymbol{E = \left(z * \frac{s}{\sqrt{n}}\right)}$, no qual o "s" corresponde ao desvio padrão (descrito na Tabela 8), o "Z" é o valor correspondente a tabela normal padrão (Tabela Z), descrito no inicio do tópico anterior e o "n" é a quantidade de iterações de teste realizada (neste caso trinta).O "N" da equação 3 será o resultado final com a quantidade de iterações que deveremos realizar para atingir o grau de confiança descrito na pergunta i do inicio deste tópico. As demais variáveis já foram descritas nos tópicos anteriores desta seção:

$$\boldsymbol{N = \left(\frac{Z\ *\ s}{E}\right)^2}$$

Equação3*. Formula utilizada para encontrar o tamanho da amostra.*

A tabela 10 abaixo demonstra o calculo da amostra, que resultará na quantidade de iterações necessárias para atingir 95% de confiança nos resultados.

Sleep 3000 ms	**Sleep 500 ms**
$E = \left(1{,}96 * \frac{260}{\sqrt{30}}\right) \cong 93\ ou\ 0{,}64\%$ $N = \left(\frac{1{,}96\ *\ 260}{93}\right)^2$ $N = \left(\frac{509{,}6}{93}\right)^2$ $\boldsymbol{N \cong 30\ itera\text{ç}\tilde{o}es}$	$E = \left(1{,}96 * \frac{150}{\sqrt{30}}\right) \cong 54\ ou\ 1{,}17\%$ $N = \left(\frac{1{,}96\ *\ 150}{54}\right)^2$ $N = \left(\frac{294}{54}\right)^2$ $\boldsymbol{N \cong 30\ itera\text{ç}\tilde{o}es}$
Sleep 250 ms	**ThreadControl**
$E = \left(1{,}96 * \frac{123}{\sqrt{30}}\right) \cong 44\ ou\ 1{,}21\%$ $N = \left(\frac{1{,}96\ *\ 123}{44}\right)^2$ $N = \left(\frac{241{,}08}{44}\right)^2$ $\boldsymbol{N \cong 30\ itera\text{ç}\tilde{o}es}$	$E = \left(1{,}96 * \frac{50}{\sqrt{30}}\right) \cong 18\ ou\ 0{,}57\%$ $N = \left(\frac{1{,}96\ *\ 50}{18}\right)^2$ $N = \left(\frac{97{,}5}{18}\right)^2$ $\boldsymbol{N \cong 29\ itera\text{ç}\tilde{o}es}$

Tabela 10 - Calculo do tamanho da amostra para todos os casos de teste.

4.3.4 RESULTADOS DA ANÁLISE

A partir dos cálculos obtidos na análise realizada, foi verificado que para atingir 95% de confiança no intervalo da média, seriam necessárias 30 iterações para os testes que utilizam de atrasos explícitos (`sleep(timeout)`) com 3000 ms, 500 ms e 250 ms e 29 iterações para o teste que utiliza do ThreadControl, que obtiveram média final de 14526 ms para o teste de 3000 ms, 4607 ms para o teste de 500 ms, 3627 ms para o teste de 250 ms e por fim 3138 ms para o teste com ThreadControl. A versão com ThreadControl, que não possui invocações a *`Thread.sleep(timeout)`*, foi 78,4% menor do que a versão com *sleep* com um tempo de espera de 3 segundos, e até 13,48% menor do que a versão com o sleep utilizando um tempo de espera de 250 ms, o que pode levar a muitos falsos positivos em algumas máquinas.

Os dados coletados correspondentes ao tempo de execução de cada caso de teste podem ser encontrados na tabela 12 no apêndice B deste documento. Para manter a conformidade dos

campos da tabela, foi inserida a sigla “n/c” significando que “não consta” valor, já que os casos de teste tiveram quantidade de iterações distintas, como por exemplo, para o teste com ThreadControl que foram necessárias 29 iterações para que fossem atingidas as exigências da questão i do inicio desta seção, enquanto o teste com sleep de 3000 ms, necessitou de apenas 30 iterações. O gráfico 1 abaixo demonstra com mais clareza os dados obtidos. A barra em azul representa a média obtida da quantidade de iterações, exigidas no calculo do tamanho da amostra e as barras em vermelho e verde são os valores do intervalo de confiança de cada caso de teste consecutivamente:

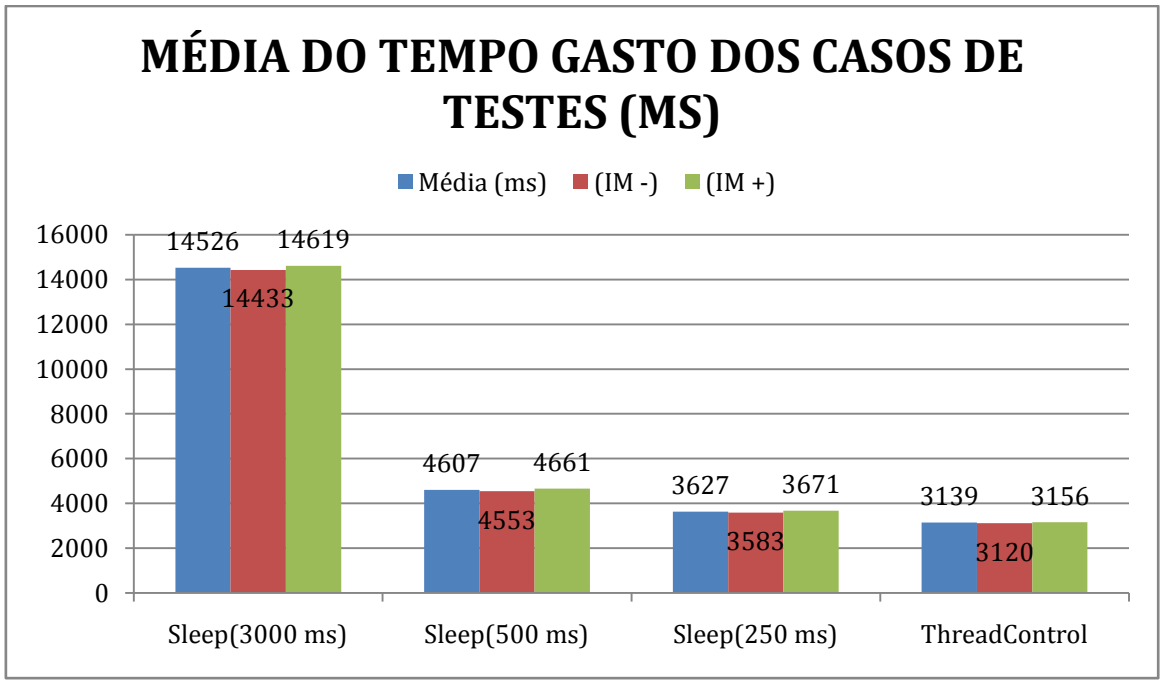

Tabela 11 - Média do tempo gasto dos casos de testes (ms)

Além de analisar o tempo de execução dos testes, investigou-se a ocorrência de falsos positivos durante a execução de cada caso de teste (respondendo a questão ii desta seção). Para isso, foi feita uma execução exaustiva de 1000 iterações para cada caso e o resultado referente a esta segunda investigação foi que apenas o `Thread.sleep(250)` gerou falsos positivos. Para 1000 iterações desta versão dos testes, ocorreram 80 falsos positivos. O restante dos testes não apresentou falsos positivos. Porém, vale ressaltar que o tempo gasto para o processamento do teste depende da configuração da maquina utilizada, processos em execução no momento do teste e do algoritmo de escalonamento utilizado pelo processador. Considerando isso, este resultado poderá divergir se for executado em outro ambiente de trabalho com características diferentes. Para os resultados descritos neste trabalho, em específico, foi utilizada uma máquina com processador Intel(R) Core(TM) i3, versão CPU M

330, clock 2.13GHz e 4 gigas de memória RAM e todos os programas não utilizados para o teste foram encerrados antes do inicio da execução.

4.4 LIÇÕES APRENDIDAS

Além de utilizar o ThreadControl para controlar o tempo antes de fazer as asserções do teste, foi analisada como alternativa o uso de invocações a `Thread.sleep(timeout)`, cuja implementação é bem simples em relação ao uso do ThreadControl, já que não necessita importar biblioteca externa e é utilizado apenas uma chamada ao método estático `sleep()` da classe Thread, sendo portanto uma pratica seguida por vários desenvolvedores de testes. Porém, como esperado, às vezes tivemos execuções onde os testes falharam porque o tempo limite escolhido não era suficiente. Para uma versão estendida do `testAddPurchase()`, anteriormente mostrada, foi percebido que a utilização de um tempo limite de 500 milissegundos, não encontrou qualquer falha em um mil execuções do mesmo teste na máquina em que foi executado. No entanto, usando um tempo limite de 250 milissegundos (ms), tivemos 80 falhas em mil execuções nesta mesma máquina. É importante notar que, em algumas máquinas, até mesmo o tempo de espera de 500 ms também levaria a algumas falhas. Por exemplo, nós executamos o teste como tempo limite de 250 ms em outra máquina e, em apenas uma execução de uma centena o teste passou. Portanto, quanto maior o tempo de espera utilizado nos testes, diminuirá a probabilidade de falhas por asserções antecipadas mas será maior o tempo de execução dos testes. Por exemplo, o tempo médio das execuções de teste com um tempo limite (*timeout*) de 3 segundos foi de 14526 segundos, enquanto as execuções de teste em relação ao teste com 500 ms de tempo de espera levou, em média, 4607. Ao usar um tempo limite de 250 ms, o teste todo levou em torno de 3627 segundos para ser executado.

A fim de evitar os falsos positivos ou execuções de teste muito longas, implementou-se também uma versão de testes com o ThreadControl e suas primitivas, tais como o `waitUntilStateIsReached()`, que exige uma invocação anterior do `prepare(expectedState)` para informar sobre o estado esperado e em seguida uma invocação à operação `proceed()` para eliminar quaisquer threads que possam ter sido bloqueadas. Esta última abordagem apresentou um melhor comportamento neste este estudo de caso, onde foi observado que seus testes foram executados 13,48% mais rápido do que a versão com o sleep utilizando um tempo de espera de 250 ms (o menor tempo utilizando atrasos explícitos desta pesquisa). Além disso, não foi constatado nenhum falso positivo, ao

contrário dos testes utilizando o atraso explícito de 250 ms, onde se observaram 80 falsos positivos em 1000 execuções.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

Neste trabalho, foi possível fazer um comparativo entre duas abordagens para se testar automaticamente um sistema com operações assíncronas: o uso do *arcabouço* ThreadControl e o uso de atrasos explícitos (*Thread.sleep()*). Além disso, se pôde observar como estas abordagens são utilizadas na prática em um sistema de informação assíncrono simples.

No contexto do uso do ThreadControl, a principal constatação é que para utilizá-lo, é preciso ter uma boa compreensão do comportamento das threads do sistema sob teste (*System Under Test* - SUT) e da criação de alguns métodos auxiliares para a criação de especificações dos estados esperados pelas threads do sistema. O ThreadControl demonstrou um melhor comportamento em relação à abordagem utilizando atrasos explícitos, considerando que houve uma redução significativa no tempo de execução dos métodos de teste e se conseguiu evitar os falsos positivos nos testes representados neste trabalho.

5.2 SUGESTÕES DE TRABALHOS FUTUROS

Como trabalho futuro, pretende-se desenvolver novos estudos de caso e comparar outras abordagens e técnicas utilizadas para testes de sistemas assíncronos. Entre as principais estão:

- O padrão de teste “Distributed Test Agents”;
- A técnica Busy-Wait (`while(!result) busy-wait`);
- O método de atraso explícito `wait()` da linguagem Java;
- O uso de implementações da classe `Future` da linguagem Java;
- O uso da técnica de Callbacks.

Também pretende-se analisar a presença de falsos negativos utilizando as abordagens acima. Consideramos falsos negativos os resultados de testes que passam na verificação das asserções embora possa haver um defeito no sistema e que só se manifesta em alguns escalonamentos das threads.

Alem disso pretende-se evoluir o ThreadControl para que sua importação seja feita por meio de um arquivo no formato *jar* ao invés de incluir seu código fonte no projeto. Outro trabalho futuro planejado é realizar testes de stress com varias requisições do mesmo tipo, ou seja, executando a mesma operação inúmeras vezes, para que seja possível obter e analisar os dados em busca de possíveis erros. Pretende-se também analisar casos de *bugs* que podem levar a estados que não são atingidos, o que poderá fazer com que o ThreadControl fique infinitamente esperando um estado que nunca será alcançado, como também, se o uso de timeouts pode solucionar este problema. Por fim pretende-se ampliar os dados desta pesquisa calculando também o intervalo de confiança da proporção para os falsos positivos.

REFERÊNCIAS BIBLIOGRÁFICAS

COLIN, B. Top 10 worst IT disasters of all time. Novembro 2007. Disponível em: http://www.zdnet.com.au/top-10-worst-it-disasters-of-all-time-339284034.htm. Acesso em: 13 maio de 2013.

COLLIS, Jill; HUSSEY, Roger. Pesquisa em Administração (2 ed.). Porto Alegre: Bookman, 2005.

CLEMENTE, Fabiane apud GIL, A. C. (2007). Pesquisa qualitativa, exploratória e fenomenológica: Alguns conceitos básicos. Sítio Administradores <http://www.administradores.com.br/informe-se/artigos/pesquisa-qualitativa-exploratoria-e-fenomenologica-alguns-conceitos-basicos/14316/>. Acessado em: 11 de junho de 2013.

DANTAS, A., GAUDENCIO, M., BRASILEIRO, F. e CIRNE, W. (2008) "Obtaining trustworthy test results in multi-threaded systems". In SBES 2008: Proceedings of the XXII Simpósio Brasileiro de Engenharia de Software.

DANTAS, A. Aumentando a Confiança nos Resultados de Testes de Sistemas Multi-threaded: Evitando Asserções Antecipadas e Tardias. Campina Grande, 2010. Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande.

DIJKSTRA, E.W. (1969). Notes on structured programming. (T.H.-Report 70-WSK-03). Technische Hogeschool Eindhoven.

DUSTIN, E., RASHKA, J. and PAUL, J. (1999) Automated Software Testing: Introduction, Management and Performance. Addison-Wesley.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

GAUDENCIO, M. Estudo sobre Técnicas de Espera Utilizadas em Testes envolvendo Assincronia. Technical report, Relatório de Estágio Integrado, DSC / UFCG, 2009.

MARTINS, W. (1985) Waneck. Estação (n-m-p): Um Computador não-Von Neumann. São Paulo: Cartgraf, 1985.

MASSOL, V.(2004). JUnit in Action.Ed. Manning.

SOMMERVILLE, I. (2006) Software Engineering. Addison-Wesley, 2006.

SALAS, P. P.; KRISHNAN, P. 2009. Automated Software Testing of Asynchronous Systems. Electron. Notes Theor. Comput. Sci. 253, 2 (Oct. 2009), 3-19. DOI= http://dx.doi.org/10.1016/j.entcs.2009.09.048

SOUSA, D., DANTAS, A., LOPES, E. (2013) Testing Asynchronous Information Systems with ThreadControl: a Case Study. In: Simpósio Brasileiro de Sistemas de Informação, 2013, João Pessoa. Anais do III Simpósio Brasileiro de Sistemas de Informação. João Pessoa, PB, maio de 2013. p. 194-205.

FIELDING. ROY. T. AND TAYLOR. N. RICHARD Principled design of the modern webarchitecture.ACM Trans. Inter. Tech., 2(2):115–150, 2002.

GUEDES, GILLEANES T. A. UML 2 - Uma Abordagem Prática. 1. ed. São Paulo: Novatec, 2009. v. 1. 488p.

APÊNDICE

Apêndice A – Arquivos Script Shell utilizado para automatizar a execução dos testes do MySimpleCRM.

Script 1: Script Main.sh, inicia a execução de todos os casos de teste.

```
#!/bin/sh

# /**
#  * Script Main
#  *
#  * @author Ayla Rebouças - ayla@dce.ufpb.br
#  * @author Diego Sousa - diego.sousa@dce.ufpb.br
#  *
#  */
sh execute.sh testWith3000.sh 1000 saida/testWith3000/
sh execute.sh testWith500.sh 1000 saida/testWith500/
sh execute.sh testWith250.sh 1000 saida/testWith250/
sh execute.sh testWithTC.sh 1000 saida/testWithTC/
```

Script 2: Script Execute.sh, inicia a execução de 1 caso de teste e é executado pelo Script Main.

```
#!/bin/bash

# /**
#  * Example usage of this script:
#  * sh execute.sh testWith3000.sh 1000 saida/testWith3000/
#  *
#  * Use: sh execute.sh [nameScript] [cont] [OutputDirectory]
#  *
#  * @author Ayla Rebouças - ayla@dce.ufpb.br
#  * @author Diego Sousa - diego.sousa@dce.ufpb.br
#  *
#  */

script=$1
cont=$2
OutputDirectory=$3
```

```
# Checking if the folder that is stored text files with time
# and the result exists. If there is no folder is created.

DIR_RESULTS=$OutputDirectory/Results
VERI_DIR_RESULTS=`ls -l $DIR_RESULTS 2> /dev/null`

if [ $? = 2 ]; then
     mkdir $DIR_RESULTS
fi

# The test runs and stores the output of time and result in
#separate files to facilitate manipulation of data.

for i in `seq 1 $contFim`
do
     sh $script > $OutputDirectory/$script$i
     cat $OutputDirectory/$script$i | grep 'Time' | awk
'{print
     $2}'
     >> $OutputDirectory/Results/$script"_Temp.txt"
     cat $OutputDirectory/$script$i | grep 'OK' | awk '{print
     $1}' >>
     $OutputDirectory/Results/$script"_Result.txt"
     cat $OutputDirectory/$script$i | grep 'FAILURES' | awk
     '{print
     $1}' >> $OutputDirectory/Results/$script"_Result.txt"
done
```

Script 3: Script utilizado para executar os casos de teste com sleep().

```
#!/bin/sh

# /**
#  * Script to run test with Sleep()
#  *
#  * @author Ayla Rebouças - ayla@dce.ufpb.br
#  * @author Diego Sousa - diego.sousa@dce.ufpb.br
#  *
#  */
```

```
echo
echo "Enter the quantity of times that the test must be run:"
echo
read quant
echo

for i in `seq 1 $quant`
do

     java           -cp           :target/classes/:target/test-
     classes/:$HOME/.m2/repository/org/hibernate/hibernate-
     core/4.1.4.Final/hibernate-core-
     4.1.4.Final.jar:$HOME/.m2/repository/org/hibernate/hiberna
     te-entitymanager/4.1.4.Final/hibernate-entitymanager-
     4.1.4.Final.jar:$HOME/.m2/repository/org/hibernate/common/
     hibernate-commons-annotations/4.0.1.Final/hibernate-
     commons-annotations-
     4.0.1.Final.jar:$HOME/.m2/repository/org/hibernate/javax/p
     ersistence/hibernate-jpa-2.0-api/1.0.1.Final/hibernate-
     jpa-2.0-api-
     1.0.1.Final.jar:$HOME/.m2/repository/antlr/antlr/2.7.7/ant
     lr-2.7.7.jar:$HOME/.m2/repository/dom4j/dom4j/1.6.1/dom4j-
     1.6.1.jar:$HOME/.m2/repository/org/javassist/javassist/3.1
     5.0-GA/javassist-3.15.0-
     GA.jar:$HOME/.m2/repository/org/jboss/logging/jboss-
     logging/3.1.0.GA/jboss-logging-
     3.1.0.GA.jar:$HOME/.m2/repository/org/jboss/spec/javax/tra
     nsaction/jboss-transaction-api_1.1_spec/1.0.0.Final/jboss-
     transaction-api_1.1_spec-
     1.0.0.Final.jar:$HOME/.m2/repository/postgresql/postgresql
     /9.1-901.jdbc4/postgresql-9.1-
     901.jdbc4.jar:$HOME/.m2/repository/junit/junit/4.8.2/junit
     -4.8.2.jar:aspectjrt-
     1.7.2.jarorg.junit.runner.JUnitCorebr.edu.ufpb.threadContr
     ol.mySimpleCRM.FacadeTestJpaWithSleep
done
```

Script 4: Script utilizado para executar o caso de teste com ThreadControl().

```
#!/bin/sh

# /**
#  * Script to run test with ThreadControl
#  *
#  * @author Ayla Rebouças - ayla@dce.ufpb.br
```

```
#  * @author Diego Sousa - diego.sousa@dce.ufpb.br
#  *
#  */

echo
echo "Enter the quantity of times that the test must be run:"
echo
read quant
echo

for i in `seq 1 $quant`
do

     java           -cp           :target/classes/:target/test-
     classes/:$HOME/.m2/repository/org/hibernate/hibernate-
     core/4.1.4.Final/hibernate-core-
     4.1.4.Final.jar:$HOME/.m2/repository/org/hibernate/hiberna
     te-entitymanager/4.1.4.Final/hibernate-entitymanager-
     4.1.4.Final.jar:$HOME/.m2/repository/org/hibernate/common/
     hibernate-commons-annotations/4.0.1.Final/hibernate-
     commons-annotations-
     4.0.1.Final.jar:$HOME/.m2/repository/org/hibernate/javax/p
     ersistence/hibernate-jpa-2.0-api/1.0.1.Final/hibernate-
     jpa-2.0-api-
     1.0.1.Final.jar:$HOME/.m2/repository/antlr/antlr/2.7.7/ant
     lr-2.7.7.jar:$HOME/.m2/repository/dom4j/dom4j/1.6.1/dom4j-
     1.6.1.jar:$HOME/.m2/repository/org/javassist/javassist/3.1
     5.0-GA/javassist-3.15.0-
     GA.jar:$HOME/.m2/repository/org/jboss/logging/jboss-
     logging/3.1.0.GA/jboss-logging-
     3.1.0.GA.jar:$HOME/.m2/repository/org/jboss/spec/javax/tra
     nsaction/jboss-transaction-api_1.1_spec/1.0.0.Final/jboss-
     transaction-api_1.1_spec-
     1.0.0.Final.jar:$HOME/.m2/repository/postgresql/postgresql
     /9.1-901.jdbc4/postgresql-9.1-
     901.jdbc4.jar:$HOME/.m2/repository/junit/junit/4.8.2/junit
     -4.8.2.jar:aspectjrt-1.7.2.jar  org.junit.runner.JUnitCore
     br.edu.ufpb.threadControl.mySimpleCRM.FacadeTestJpaWithTC

done
```

Apêndice B – Tabela abaixo com os dados coletados correspondentes ao tempo de execução de cada caso de teste.

Tabela 12 - Tempo de execução dos casos de teste e média final.

ID	Sleep 3000 ms	Sleep 500 ms	Sleep 250 ms	ThreadControl
# 1	14685	4572	3596	3137
# 2	14689	4677	3672	3159
# 3	14711	4651	3667	3193
# 4	14656	4723	3627	3097
# 5	14631	4674	3713	3129
# 6	14071	4631	3585	3219
# 7	14066	4634	3632	3164
# 8	14651	4632	3719	3094
# 9	14643	4666	3628	3102
# 10	14621	4633	3603	3161
# 11	14686	4662	3638	3119
# 12	14699	4601	3654	3114
# 13	14603	4668	3599	3192
# 14	14673	4706	3653	3196
# 15	14066	4669	3688	3155
# 16	14626	4661	3657	3138
# 17	14678	4073	3679	3175
# 18	14064	4675	3647	3185
# 19	14066	4609	3704	3018
# 20	14697	4064	3646	3116
# 21	14659	4681	3617	3172
# 22	14683	4622	3651	3075
# 23	14685	4647	3572	3119

# 24	14062	4599	3651	3198
# 25	14067	4593	3007	3133
# 26	14615	4662	3698	3104
# 27	14726	4635	3687	3184
# 28	14673	4647	3659	3011
# 29	14661	4621	3658	3173
# 30	14673	4625	3599	n/c
Média Final	**14526**	**4607**	**3627**	**3139**