5
Estudo de Caso

O teste funcional pode ser aplicado em contextos mais amplos e mais complexos do que o retratado no exemplo no capítulo sobre teste funcional (capítulo 4). Este é o caso de sistemas baseados em componentes de software que interagem em ambiente distribuído. O estudo descrito neste capítulo reflete o teste funcional de um sistema deste tipo (Fonseca, 2007) e mostra como foi possível avaliá-lo funcionalmente por meio da decomposição do problema em problemas menores e mais simples.

Neste estudo de caso, o propósito do sistema avaliado é realizar a contagem do número de ocorrências de cada palavra no texto de um documento. Para atender a esta finalidade com um grau de eficiência satisfatório, a aplicação em análise foi elaborada segundo o modelo de programação *MapReduce* (Dean & Ghemawat, 2004).

O *MapReduce* é um modelo para processamento e geração de enormes conjuntos de dados. Usuários especificam uma função de mapeamento (`map`) e uma função de redução (`reduce`). A função `map` processa pares chave/valor de modo a gerar um conjunto intermediário de pares chave/valor. A função `reduce` integra todos os valores intermediários associados a uma mesma chave. Adotando este estilo funcional, é possível paralelizar o processamento e executar as tarefas de forma distribuída.

Muitas aplicações podem ser expressas segundo este modelo (Dean & Ghemawat, 2004), a exemplo de vários trabalhos conduzidos na empresa *Google Inc.*. Com isto, surgiu o interesse desta pesquisa em analisar funcionalmente um sistema que adotasse tal abordagem, dando origem ao estudo de caso proposto.

O estudo de caso foi dividido em quatro partes (seções 5.1, 5.2, 5.3, 5.4) que correspondem respectivamente às quatro fases do processo de teste funcional (capítulo 4). Ao final, são apresentados os resultados obtidos (seção 5.5).

5.1. Preliminares

Para começar a avaliar funcionalmente o sistema em teste proposto, a primeira atividade a ser realizada foi ***identificar documentos de especificação de requisitos***.

Os ***requisitos*** do problema de contagem do número de ocorrências de cada palavra no texto de um documento estão diretamente relacionados às funções de mapeamento e redução e às especificações do sistema e do ambiente de execução. Estes requisitos foram observados a partir da leitura de um artigo (Dean & Ghemawat, 2004) que descreve o problema.

A função de mapeamento deve emitir cada palavra (chave) associada a uma contagem (valor) de ocorrências, que no caso avaliado é simplesmente o "1". A função de redução deve somar todas as contagens emitidas (coleção de valores) para certa palavra (chave). Por exemplo, a função de mapeamento associa o valor "1" a cada palavra "requisitos" do parágrafo anterior, texto supostamente analisado (`"...requisitos|1...requisitos|1..."`). A função de redução soma todas as contagens da chave "requisitos" (`requistos|2`).

As especificações do sistema definem os parâmetros de sua configuração como o número de componentes utilizados para o trabalho de mapeamento/redução e o tamanho das partições do arquivo de entrada. As especificações do ambiente descrevem os nomes dos arquivos de entrada e saída utilizados pelo sistema assim como opções de conexão à infra-estrutura de componentes utilizada.

Os ***documentos*** separados foram um artigo sobre o sistema, arquivos IDL (*Interface Description Language*), arquivos HTML (*HyperText Markup Language*) e um artigo sobre o ambiente de execução do sistema.

O primeiro artigo (Dean & Ghemawat, 2004) representa o documento de visão do sistema, descrevendo o problema e a solução em linhas gerais. Aborda o modelo de programação, o qual acabou definindo a arquitetura da solução.

Os arquivos IDL definem a interface pública dos componentes utilizados. Por meio de sua descrição, é possível relacionar as funções do sistema e as estruturas de dados usadas por estas funções.

Os arquivos HTML destacados constituem a documentação das unidades de programação (classes e interfaces) do sistema construídas usando a linguagem de programação Java. Esta documentação é denominada *javadoc*.

O outro artigo (Augusto et al., 2007) descreve uma infra-estrutura leve e simples, projetada para distribuir, instanciar e executar aplicações baseadas em componentes de software segundo a arquitetura CORBA, cuja denominação é SCS – Sistema de Componentes de Software.

Uma vez identificados os documentos de especificação de requisitos, a próxima atividade foi ***definir o escopo de avaliação***. Nesta atividade, houve interferência do desenvolvedor que indicou alguns pontos críticos que precisariam ser testados.

Neste estudo de caso, o escopo de avaliação foi (1) verificar, para um dado arquivo texto de entrada, se os pares chave/valor intermediários são gerados corretamente pela função `map`. Além disso, foi preciso (2) checar se a contagem dos valores integrados por chave pela função `reduce` reflete a soma de todas as contagens emitidas. Outros aspectos também avaliados foram se (3) o número de componentes gerados correspondia ao configurado, se (4) o particionamento em tarefas foi feito segundo a especificação e se (5) a distribuição de tarefas (escalonamento) entre os componentes garante a execução de todas as tarefas.

5.2. Modelagem do Sistema em Teste

De posse dos documentos de especificação separados, e sabendo o escopo de avaliação definido preliminarmente, iniciou-se a fase de modelagem da aplicação em forma de modelos construídos a partir de gramáticas.

A primeira atividade foi ***especificar a estrutura*** do sistema que será utilizada pelos testes. Neste caso, fazem parte do escopo de teste:

- A classe `MasterApp` de controle do sistema, responsável pela instanciação dos componentes de trabalho (*workers*) e pelo escalonamento de tarefas (*tasks*).
- A classe `MapReduceServant`, que representa a de lógica do sistema e descreve as operações de `map` e `reduce` dos componentes de trabalho, as quais são definidas pela interface `MapReduce`;

- A classe `Task`, que representa a estrutura de dados de uma tarefa.

Note que a classe `MapReduceComponent` que descreve o componente de trabalho e é utilizada para encapsular a lógica do sistema segundo o sistema de componentes adotado (SCS) não será abordada, pois ela não é acessada diretamente. Repare também que a nomenclatura das classes segue a convenção deste mesmo sistema de componentes.

A Tabela 15 exibe o resultado da modelagem da estrutura. Atente para o fato de que a gramática empregada se restringe aos conceitos necessários à definição das classes `MasterApp`, `MapReduceServant`, `Task`. A gramática utilizada para descrever o modelo estrutural está descrita no capítulo sobre modelos gramaticais (capítulo 3).

Os arquivos gerados pelas funções de mapeamento e redução também foram avaliados e precisaram ser modelados como parte da estrutura. O formato especificado para os mesmos foi descrito segundo a gramática de estrutura. Para caracterizar o marcador de separação de chave e valor, foi utilizado um atributo.

Modelo Estrutural

```
/* Modelagem das Classes */
Task = ($string:fileExpected  $string:filePath
        $integer:id  $integer:taskId  $string:taskType);
MasterApp = ( Arraylist:reducedFiles
              TasksQueue:tasksQueue
              WorkersQueue:workersQueue
              HashTable:workingOnTable );

/* Modelagem dos arquivos */
mapFile = ( {$string:key ?|? $natural:value}:mapping );
reduceFile = ( { $string:key ?|? $natural:value }:mapping };

/* Notas:
- como o teste só utilizou as operações das classes, as superclasses
não foram descritas;
- a classe MapReduceServant que faz parte do escopo de avaliação só
apresenta operações e por isso não está retratada no modelo estrutural.
O mesmo acontece para ArrayList, BlockingQueue e HashTable;
- o atributo ?|? indica como os pares chave e valor estão separados. */
```

Tabela 15 – Modelo Estrutural do Estudo de Caso

A segunda atividade desta fase foi ***especificar os comportamentos*** do sistema que eram alvo dos testes. Neste caso, fazem parte do escopo de teste:

- As operações `buildWorkersQueue`, que monta a fila de trabalhadores (componentes de trabalho), `buildTasksQueue`, que monta a fila de tarefas a executar, `schedule`, que realiza a distribuição de tarefas por trabalhado, `getMasterChannel()`, que Todas são operações da classe `MasterApp`.
- As operações `map` e `reduce` da classe `MapReduceServant`, responsáveis, respectivamente, pelas funções de mapeamento e redução do modelo;
- As operações de construção das classes. Por padrão de notação, todos os construtores são descritos da seguinte forma `'makeClassName'`.

A Tabela 16 exibe o resultado parcial da modelagem dos comportamentos.

Modelo Comportamental

```
MasterApp = ( makeMasterApp () MasterApp;
              buildWorkersQueue ($integer:workersCount) $void;
              buildTasksQueue ($integer:fileParts
                               $string:path) $void;
              schedule () $void;
              getMasterChannel () IComponent; );
/* A operação getMasterChannel foi solicitada ao desenvolvedor e
disponibilizada somente para facilitar a realização do teste. */

WorkersQueue = ( getQueueSize () $integer; );

MapReduceServant = ( makeMapReduceServant () MapReduceServant;
                     map ( $string:inputPath
                           $string:outputPath
                           IComponent:channel
                           $integer:taskId ) $boolean;
                     reduce = ( $string:inputPath
                                $string:outputPath
                                IComponent:channel
                                $integer:taskId ) $boolean;
/* Nota: operações providas pela interface pública das classes. */
```

Tabela 16 – Modelo Comportamental do Estudo de Caso

5.3. Seleção do Cenário de Teste

Uma vez modelados a estrutura e o comportamento do sistema em teste, começou a fase de seleção do cenário de teste.

A primeira atividade desta fase foi ***definir o critério de seleção de teste*** segundo o escopo de avaliação identificado durante a fase de preliminares (seção 5.1). Neste estudo de caso, optou-se tanto por uma estratégia de ***teste randômico*** quanto por uma estratégia de ***teste de valores de contorno*** para a geração dos dados de teste. O teste randômico foi aplicado na verificação das operações `map` e `reduce` da classe `MapReduceServant`. O teste de valores de contorno foi aplicado na verificação dos métodos `buildWorkersQueue`, `buildTasksQueue` e `schedule` da classe `MasterApp`.

Por se tratar de uma aplicação codificada na linguagem de programação Java, foram selecionadas as seguintes classes da arquitetura de apoio: `JavaRandomStrategy` e `JavaBoundaryStrategy` (pacote `gbtx.java`) que estendem, respectivamente, `RandomStrategy` e `BoundaryStrategy` (pacote `gbt.functional`) abordadas no capítulo anterior (seção 4.4.1). Se não houvesse intenção de verificar a implementação do sistema, restringindo-se a um teste conceitual, as classes originais independentes de plataforma poderiam ter sido escolhidas.

Definido o critério de teste, iniciou-se a segunda atividade desta fase que é ***especificar os casos de teste*** utilizando o modelo gramatical correspondente. Para cada um dos itens avaliados (seção 5.1) há pelo menos um caso de teste descrito por um objetivo de teste (`testGoal`) e por um oráculo de teste (`testOracle`). Repare que a descrição deste modelo independe da plataforma utilizada e que cada critério define um cenário de teste diferente.

A Tabela 17 e a Tabela 18 apresentam os casos de teste associados a cada critério de teste. Exibem parcialmente os casos de teste projetados, destacando apenas aqueles relacionados às operações `map` e `buildWorkersQueue`.

Modelo de Caso de Teste – Primeiro Critério

```
/* Teste para garantir que os arquivo de mapeamento corresponde ao
especificado */
testGoal = ( makeMasterApp () ?master?;
             ?master? getMasterChannel() ?channel?;
             makeMapReduceServant () ?servant?;
             ?servant? map('e:/source.txt' 'e:/map.txt' ?channel? '1');
           ) @e:/map.txt;
testOracle = () @e:/mapOracle.txt;

/* Semântica da representação está descrita no capítulo 3. */
/* ... */
```

Tabela 17 – Modelo de Caso de Teste do Estudo de Caso – Primeiro Critério

Modelo de Caso de Teste – Segundo Critério

```
/* Teste p/ garantir que a fila realmente começa sem elementos. */
testGoal = ( makeMasterApp () ?master?;
             ?queue? = ?master? workersQueue;
           ) ?queue? getQueueSize();
testOracle = 0;

/* Teste p/ garantir que a fila terá o número de elementos desejado. */
testGoal = ( makeMasterApp () ?master?;
             ?master? buildWorkersQueue ( 10 ); )
             ?queue? = ?master? workersQueue;
           ) ?queue? getQueueSize();
testOracle = 10;

/* Semântica da representação está descrita no capítulo 3. */
/* ... */
```

Tabela 18 – Modelo de Caso de Teste do Estudo de Caso – Segundo Critério

A atividade seguinte foi ***especificar o ambiente de teste***. O modelo gramatical utilizado estabelece as configurações do ambiente de teste. Ele faz o mapeamento entre os conceitos utilizados nos demais modelos e os conceitos definidos pela plataforma de execução. Para isto declara os elementos não terminais dos modelos estrutural, comportamental e de casos de teste e define seu valor dependente de plataforma.

Neste estudo de caso, o modelo indicou a conversão entre conceitos da especificação e classes/interfaces Java do sistema em teste (Tabela 19).

O modelo de ambiente também é responsável pela definição do estado inicial. Mas neste estudo, o estado inicial foi estabelecido em cada caso de teste por uma opção do testador.

Modelo de Ambiente

```
/* Mapeamento da Estrutura para a Plataforma Java */

MasterApp = scs.demos.mapreduce.servant.MasterApp;
MapReduceServant = scs.demos.mapreduce.servant.MapReduceServant;
Task = scs.demos.mapreduce.servant.Task;
IComponent = scs.core.IComponent;

ArrayList = java.util.ArrayList;
TasksQueue = java.util.concurrent.linkedBlockingQueue
  <scs.demos.mapreduce.servant.Task>;
WorkersQueue = java.util.concurrent.linkedBlockingQueue
  <scs.demos.mapreduce.MapReduce>;
HashTable = java.util.HashTable;

$string = java.lang.String;
$natural = int;
/* ... */

/* Mapeamento dos Comportamentos para a Plataforma Java */

MaterApp.makeMasterApp = MasterApp;
MaterApp.buildWorkersQueue = buildWorkersQueue;
MaterApp.buildTasksQueue = buildTasksQueue;
MaterApp.schedule = schedule;
MaterApp.getMasterChannel = getMasterChannel;

MapReduceServant.map =  map;
MapReduceServant.reduce = reduce;

WorkersQueue.getQueueSize= size;
/* ... */
```

Tabela 19 – Modelo de Ambiente do Estudo de Caso

A quarta e última atividade desta fase foi ***gerar o cenário de teste*** para cada critério identificado. Esta geração corresponde à interpretação dos modelos de casos de teste e de ambiente, com auxílio dos modelos do sistema (estrutural e comportamental). A geração é feita por um construtor de testes (`TestBuilder`) que é ajudado por um interpretador (`TestParser`) de cada modelo de teste. Cada interpretador realiza sua tarefa de acordo com a estratégia de teste selecionada. A gramática especificada pelo modelo orienta a geração dos dados de teste. Os atributos descritos nestas gramáticas representam possibilidades de variação na geração destes dados. Estes atributos são considerados ou não pela estratégia empregada. O resultado é um programa que descreve o cenário de teste.

Neste estudo de caso, como havia necessidade de avaliar efetivamente o código fonte, foi usado um construtor de testes destinado a gerar programas na linguagem Java (`JavaTestBuilder`). Este construtor (Figura 59) é uma extensão do construtor original (`TestBuilder`) citado na descrição do teste funcional (seção 4.4.4). O programa por ele gerado (Tabela 20) apresenta uma classe de cenário de teste com métodos correspondentes a cada caso de teste. Os objetivos de teste são projetados como métodos chamados pelo caso de teste e os dados gerados são usados como parâmetros destes métodos. Os oráculos de teste são usados em assertivas presentes no caso de teste. Se o oráculo de teste equivaler a uma seqüência de chamadas, outro método é criado tendo como valor de retorno um resultado que será confrontado com o valor do objetivo de teste por uma expressão booleana que compõe a assertiva.

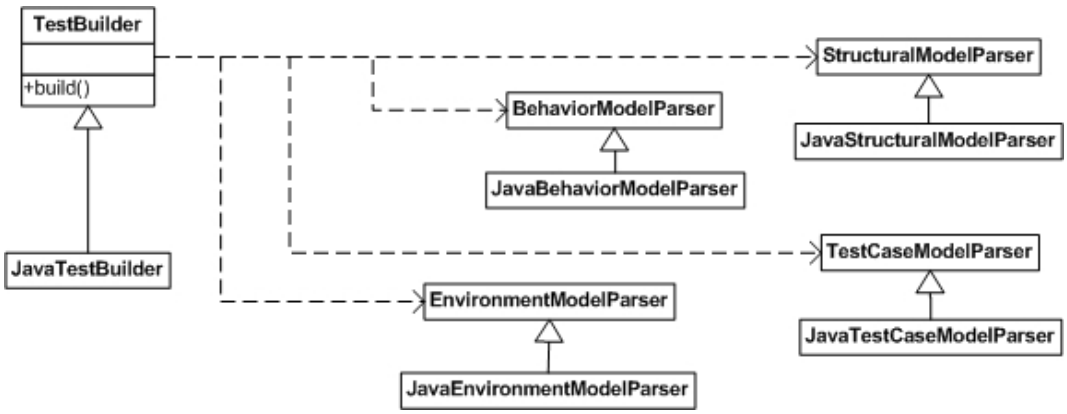

Figura 59 – Extensão do Gerador de Teste.

Código Gerado

```
package test;


import java.util.List;


public class TestScenario1
{

    public enum Verdict
    {
        pass, fail, error
    }


    public class TestLog
    {
        public TestLog(Verdict verdict)
        {
            this.verdict = verdict;
        }

        private Verdict verdict;

        public Verdict getVerdict()
        {
            return verdict;
        }
    }


    private List<TestLog> testLog = new java.util.ArrayList<TestLog>();


    public List<TestLog> getTestLog()
    {
        return testLog;
    }
```

```
public abstract class TestCase
{

    public boolean assertCase()
    {
        boolean result = false;
        TestLog log;
        try
        {
            result = testGoal().equals(testOracle());

            if (result)
                log = new TestLog(Verdict.pass);
            else
                log = new TestLog(Verdict.fail);
        }
        catch (Exception e)
        {
            log = new TestLog(Verdict.error);
        }
        testLog.add(log);
        return result;
    }

    public abstract Object testGoal();

    public abstract Object testOracle();
}


private List<TestCase> testSuite =
    new java.util.ArrayList<TestCase>();


public List<TestCase> getTestSuite()
{
    return testSuite;
}
```

```
    public class TestCase1 extends TestCase
    {
        public Object testGoal()
        {
            scs.demos.mapreduce.servant.MasterApp master =
                new scs.demos.mapreduce.servant.MasterApp();
            java.util.concurrent.LinkedBlockingQueue
                <scs.demos.mapreduce.MapReduce>
                queue = master.workersQueue;
            return queue.size();
        }
        public Object testOracle()
        {
            return 0;
        }
    }

    public class TestCase2 extends TestCase
    {
        public Object testGoal()
        {
            scs.demos.mapreduce.servant.MasterApp master =
                new scs.demos.mapreduce.servant.MasterApp();
            master.buildWorkersQueue(10);
            java.util.concurrent.LinkedBlockingQueue
                <scs.demos.mapreduce.MapReduce>
                queue = master.workersQueue;
            return queue.size();
        }
        public Object testOracle()
        {
            return 10;
        }
    }

    public TestScenario1()
    {
        this.testSuite.add(new TestCase1());
        this.testSuite.add(new TestCase2());
    }
}
```

Tabela 20 – Código Gerado

5.4. Execução e Avaliação do Cenário de Teste

Com o cenário de teste gerado no formato de um programa, passou-se à última fase do processo de teste. Nesta fase, o programa é executado e o resultado de sua execução é avaliado.

A primeira atividade desta fase foi ***executar o cenário de teste*** de cada critério identificado. O programa equivalente ao cenário de teste é aproveitado por um executor de testes (`TestRunner`) como descrito no processo de teste funcional (seção 4.5.1). Este é responsável pela execução do programa que consiste na execução de todos os métodos representantes dos casos de teste. Assim, a atividade é fragmentada em tarefas, cada uma corresponde a ***executar um caso de teste***. O resultado da execução de cada caso de teste é armazenado num laudo de teste (`TestLog`).

Neste estudo de caso, uma extensão (Figura 60) do executor de testes foi empregada por se tratar de um programa descrito na linguagem Java (`JavaTestRunner`). Este dispositivo é responsável por interagir com a aplicação para execução dos testes e por capturar os resultados obtidos.

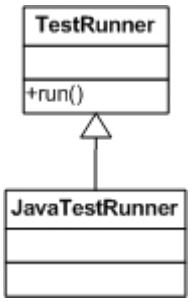

Figura 60 – Extensão do Executor de Teste.

Depois da execução, a próxima e última atividade foi ***gerar o veredicto de teste***. Para isso, os resultados dos casos de teste são avaliados por um juiz (`TestJudge`). Este juiz se baseia numa estratégia (`TestJudgement`) para atribuir um veredicto a todo cenário de teste. Este veredicto varia conforme esta estratégia de julgamento.

Para avaliar os resultados de um programa Java, foi preciso usar uma estratégia de julgamento específica (`JavaTestJudgement`) desta linguagem,

capaz de interpretar e comparar os resultados obtidos nos laudos de teste. A estratégia de julgamento usada neste caso de teste foi a padrão, onde o cenário de teste: “falhou” se pelo menos um caso de teste for inválido; “passou” se todos os casos de teste forem válidos; “errado” se a arquitetura de teste apresentar problemas durante sua execução.

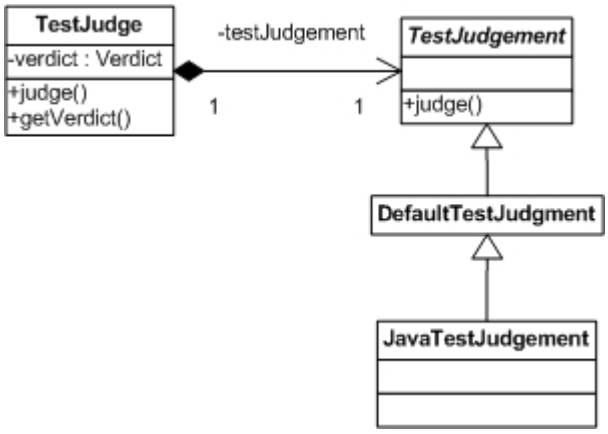

Figura 61 – Extensão do Julgamento de Teste.

5.5. Resultados

Após a realização dos testes, analisa-se qualitativamente o emprego do processo de teste com sua arquitetura de apoio e os resultados obtidos com o teste funcional do sistema em estudo.

Para avaliar o arquivo gerado pela função de mapeamento do estudo de caso, foi necessário modelar com a gramática de teste a estrutura do arquivo de entrada e do arquivo de saída desta função. Assim, por meio de uma estratégia de teste randômico, foram gerados arquivos de entrada e saída correspondentes e usados respectivamente como parâmetro do objetivo de teste e como oráculo de teste. O resultado da execução deste caso de teste no programa Java é um arquivo de texto que pôde ser comparado com o arquivo criado como oráculo de teste. A mesma abordagem foi realizada para verificação da função de redução.

No caso da conferência de componentes gerados, particionamento em tarefas e distribuição de tarefas, a estratégia empregada foi a de teste de valores de contorno. Neste caso, os valores das variáveis de ambiente correspondentes aos parâmetros de configuração do sistema em teste sofreram ligeiras mudanças

(acréscimos e decréscimos de unidades) para verificar se as estruturas de dados da aplicação se comportavam adequadamente. Com essas variações foi possível analisar a correta alocação e liberação de recursos.

Para garantir que a estrutura de teste estava funcionando corretamente e não influenciou nos resultados obtidos, foi solicitado ao desenvolvedor introduzir erros propositais em sua codificação. Assim, tendo controle sobre estes erros, foi possível determinar se a arquitetura de testes identificava ou não sua presença.

O saldo dos testes funcionais foi positivo na medida em que permitiu a identificação de todos os problemas existentes na programação do sistema em teste e possibilitou confrontar o programa criado com sua especificação. Outro aspecto interessante foi observar que os modelos gramaticais permitiram avaliar a especificação por meio de uma nova linguagem independente de implementação.