Universidade Federal de Pernambuco

Graduação em Ciência da Computação
Centro de Informática

2013.2

Um Estudo Sobre a Transição para Arquiteturas Multicore em uma Aplicação de Código Aberto

---

Trabalho de Graduação

Aluna - Jéssica de Carvalho Barbalho {jcb@cin.ufpe.br}
Orientador - Fernando José Castor de Lima Filho {fjlcf@cin.ufpe.br}

Recife, Março de 2014.

Agradecimentos

Primeiro agradeço a Deus, àquele que me deu a vida e me proporcionou todos os momentos, recursos e pessoas que me ajudaram a chegar onde estou. Sem Ele, nada teria acontecido, e se não fosse para Ele, nenhum trabalho valeria a pena, por isso agradeço antes de tudo ao Senhor.

Agradeço à minha família, meus pais, Zé e Fátima, e a minha irmã Vanessa, pelo apoio, pelo respeito, principalmente pelo suporte, pelas várias marmitas feitas antes mesmo de eu acordar, por tantas vezes fazerem meus serviços domésticos em meu lugar para que eu pudesse ter mais tempo de estudo, essas são poucas das grandes coisas que fizeram por mim, amo vocês.

Agradeço a Celso, meu namorado, amigo, companheiro, foi ele quem mais aguentou meus estresses e suportou as tantas vezes que tiver que ficar no CIn nos fins de semana. Mas valeu a pena amor, obrigada pelo incentivo e por me fazer enxergar o lado positivo das situações, estamos juntos.

Agradeço imensamente à tia Dinha, pelas centenas de carona da UFPE até em casa, obrigada tia pela ajuda e por várias vezes em esperar até tarde para que eu não fosse sozinha. Agradeço aos meus companheiros de projetos Wellton e Húgaro pelos projetos feitos, viradas de noite, apoio e muito trabalho em equipe! Á minha amiga Cynthia, pela amizade inestimável e por seguir junto comigo nesta caminhada da graduação!

Tenho muito que agradecer aos meus amigos e irmãos da Igreja Adventista do Sétimo Dia, especialmente pelas orações e pela torcida, à família que Deus me deu, muito obrigada! Aos meus familiares, avós, primos, tios, valeu pelo carinho, pelas mensagens pelo incentivo, pois vocês me acompanham muito antes da faculdade!

Agradeço ao meu orientador Fernando Castor, por ter aceitado fazer este trabalho junto comigo e por ter se disposto a me ajudar mesmo diante de tantos compromissos. Também não posso deixar de agradecer à infraestrutura oferecida pelo Centro de Informática, pois me proporcionou recursos que não estavam acessíveis a mim e também um ambiente onde pudemos desenvolver nossos projetos.

Aos meus amigos, colegas, vizinhos, conhecidos, irmãos, agradeço a Deus pela vida de cada um de vocês e por estarem presentes na minha vida, me ajudando a crescer e a conquistar meus objetivos.

Obrigada!

Resumo

A arquitetura multicore tem se tornado predominante para as necessidades computacionais gerais, por estas razões muitas vezes é necessária uma mudança de arquitetura dentro do projeto, estas mudanças, ou transição de arquitetura, podem ser analisadas através do código criado, das diferenças entre os trechos de códigos ao longo do tempo e dos metadados associados a estas alterações (mensagens de *commits*, *bugtracking*) [3].

Muitos projetos grandes, que passaram por modificações deste tipo estão disponíveis em repositórios públicos e são aplicações de código aberto que podem ser acessados facilmente. É através dos recursos disponíveis nos repositórios públicos que será analisada a eficácia da transição para arquitetura multicore, tanto pela verificação das construções implementadas como também pela análise de surgimento de falhas decorrentes das modificações.

Palavras-chave: paralelismo, concorrência, mineração de repositório, Github, JUnit, multicore, código aberto, Java, arquitetura.

Abstract

A multicore architecture has become prevalent for general computing needs, for these reasons it is often necessary to change the architecture within the project, these changes or transition architecture can be analyzed through the code created, the differences between sections of codes over time and the metadata associated with these changes (commits messages, bugtracking) [3] .

Many large projects that have undergone modifications of this type are available in public repositories, are open source and can be easily accessed. It is through the resources available in public repositories that will analyze the effectiveness of the transition to multicore architecture, by checking the buildings as well as for examining the emergence of failures arising from the changes.

Keywords : parallelism, concurrent, mining repository, Github, JUnit, multicore, open source, Java, architecture.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

A indústria de hardware computacional recorreu à construção de unidades de processamento (CPUs) multicore para manter a previsão da Lei de Moore. A revolução multicore coloca uma pressão nos desenvolvedores de software para que os mesmos utilizem paralelismo a fim de aproveitar todos os benefícios das inovações recentes de hardware [1]. Antes a arquitetura multicore era aplicada para fins computacionais específicos, complexos que requeriam grande poder computacional como, por exemplo, previsão do tempo, porém o hardware multicore está em praticamente todos os computadores pessoais atuais [2].

Paralelismo é uma operação difícil, porém novos métodos se tornam mais fáceis de serem manuseados com o passar do tempo e surgimento de ferramentas que auxiliam o processo. Historicamente o desenvolvedor que deseja programar paralelamente enfrenta as seguintes dificuldades:

1. Alto custo e raridade de sistemas paralelos
2. Falta de pesquisa e experiência com sistemas paralelos
3. Falta de acesso público a códigos paralelos
4. Não aplicação de engenharia de software em programação paralela
5. Grande overhead de comunicação no processo de programação paralela

Muitas dessas dificuldades já foram superadas. O próprio avanço do hardware elimina a primeira dificuldade, colocando computadores pessoais com sistemas paralelos ao acesso de todos. A pesquisa avança nesta área, muitos artigos e livros já foram publicados baseados em programação paralela.

Grandes projetos e bibliotecas de grande porte têm seus códigos liberados ao acesso público, facilitando a disseminação código paralelo. A grande quantidade de sistemas paralelos, produzidos nas décadas de 80 e 90, proporcionou à comunidade de software, desenvolvedores com habilidades de aplicar engenharia ao desenvolvimento de sistemas paralelos. O quinto ponto mencionado não foi superado com o passar do tempo e ainda representa uma dificuldade atual.

Ao desenvolver paralelamente alguns objetivos devem ser considerados para o ambiente de desenvolvimento: desempenho, produtividade e generalização. O desempenho do sistema é medido considerando o número de transições realizadas por *clock*. A produção de software paralelo deve ter uma alta produtividade e quanto mais generalizado for mais objetivos podem ser alcançados na produção usando o mesmo sistema.

Existe um grande desafio para alcançar o equilíbrio entre os três objetivos, especialmente entre generalização e produtividade, pois existem ambientes para objetivos muito específicos que tem alto desempenho e ótima produtividade. Já ambientes generalizados são mais difíceis de configurar e alinhar o ambiente com o objetivo do desenvolvimento. Estes objetivos devem reger o desenvolvimento de sistemas paralelos.

Nem sempre adotar programação paralela é a melhor solução para um projeto. Existem alternativas que melhoram o desempenho de uma aplicação e simulam um ambiente paralelo. Alguns deles:

- Rodar múltiplas instâncias de uma aplicação sequencial
- Fazer a aplicação usar software paralelo
- Aplicar otimização de desempenho

Diante de todas as contextualizações aqui feitas, o que realmente faz programar paralelamente difícil está do lado do ser humano que está programando. A habilidade de dizer à máquina o que deve ser feito paralelamente é a maior dificuldade no desenvolvimento. Para esta dificuldade ser sanada e os objetivos de desempenho, produtividade e generalização serem alcançados quatro tarefas devem ser bem desempenhadas pelo desenvolvedor.

1. Particionar o trabalho. Uma análise do que será programado deve ser feita a fim de dividir o trabalho da melhor forma possível. Porém esta divisão deve ser feita de forma muito cuidadosa. O código particionado não deve ser executado sequencialmente. Esta divisão de tarefas dificulta a captura de erros e eventos e deve haver uma preocupação com a comunicação entre *threads* assim como o controle da quantidade de *threads* em execução.
2. Controle de acesso paralelo. A coordenação ao acesso de recursos de forma paralela torna-se um problema, pois a forma de acesso de cada recurso depende da localização do mesmo e o acesso pelas diversas *threads* deve ser controlado
3. Partição de recursos e replicação. O desenvolvedor deve entender como particionará os recursos da aplicação, bem como deve ser feita sua replicação.
4. Interação com hardware. Esta interação geralmente cabe aos Sistemas Operacionais, mas dependendo do tipo de funcionalidade que está sendo desenvolvida a aplicação irá interagir com o hardware, cabendo ao programador analisar como isto será feito pelas várias *threads*.

Alguns ambientes de programação facilitam a vida do programador, e não o obrigam a se preocupar com todos os pontos citados acima. As quatro tarefas quando

pensadas de forma conjuntam minimizam a comunicação entre *threads* bem como aumentam a produtividade do desenvolvedor. [4]

Este trabalho está dividido em quatro capítulos, sendo este o primeiro. O capítulo dois deste estudo trata de paralelismo, de como o hardware se comporta diante da arquitetura multicore, suas implicações e custos, bem como alguns conceitos básicos para lidar com programação paralela.

O terceiro capítulo fala da metodologia usada neste estudo, todas as etapas seguidas. O quarto capítulo diz respeito ao objetivo geral deste trabalho que é o estudo da transição de um projeto para um aproveitamento de estruturas paralelas. Analisando um projeto *open source* que tenha passado por mudanças estruturais adotando paralelismo. Avaliando as mudanças, suas consequências para a aplicação envolvida e os resultados da transição. O último capítulo traz as conclusões deste estudo.

2. Background

Antes de iniciar o estudo sobre a transição de arquitetura é importante empenharmos um capítulo entendendo as implicações de um ambiente paralelos no nível de hardware, bem como estudarmos noções básicas de programação paralela a fim de compreender a complexidade desta operação em todas as camadas da computação.

2.1. Revolução do hardware

O hardware passou por uma revolução em sua arquitetura graças à Lei de Moore, houve uma explosão no número de transistores em uma CPU bem como sua diminuição de tamanho em contraste com a sua capacidade de processamento. Um processador típico da década de 80 levava três ciclos de *clock* para completar uma instrução, enquanto no início dos anos 2000 várias instruções eram executadas simultaneamente, devido ao surgimento das *pipelened* CPUs. Porém com este grande avanço também surgiu alguns obstáculos.

- Fluxo do *pipeline*. Dificuldade de prever quando uma instrução acaba. Se uma aplicação executa vários loops e possui muitas instâncias de objetos, o hardware não tem a capacidade de prever qual será a próxima instrução a ser executada e por isso fica à espera do término da execução, não obtendo o desempenho esperado.

- Acesso à memória. Um processador executa centenas de instruções no tempo de um acesso à memória. Este fato acontece pelo aumento do tamanho da memória, fazendo com que a CPU não consiga mapear a memória inteira, sendo obrigada a gastar bem mais ciclos de *clock* para acessá-la.

- Operação atômica. Instruções, determinadas pelo programador, que não podem ser quebradas limitam o uso do *pipeline* pela CPU.

- Barreiras de memória. O programador pode proteger instruções e variáveis, esta proteção cria barreiras na memória fazendo com que a CPU não possa reordenar as instruções, limitando o *pipeline*.

- Erro de *cache*. Quando uma variável é compartilhada entre várias CPUs a mesma não é buscada da *cache* causando aumentando a taxa de erro de *cache* e tendo que buscar a referência na memória.

- Operações de I/O. Semelhante ao problema de erro de *cache*, as operações de entrada e saída surgem quando se trata da comunicação entre CPUs de sistemas paralelos distribuídos.

As diversas CPUs, juntamente com suas *caches*, se comunicam por meio de sistema computacional que as liga entre si por meio de vias. Estas vias ligam todas as CPUs à memória. É este mecanismo de comunicação que eleva o custo das operações, elas limitam o desempenho das diversas CPUs que trabalham paralelamente.

2.2. Custo de operações

Dos diversos obstáculos apresentados alguns são mais significantes do que os outros, esta diferenciação é feita pela comparação do custo de cada operação. As operações relacionadas aos erros de *cache*, proteção de variáveis e I/O são as de maior custo. O custo é baseado no tempo de cada operação, e o tempo gasto numa operação deste tipo é o mesmo gasto na execução de cerca de quinhentas instruções básicas pela CPU. Existem diversas tecnologias, tanto em nível de hardware quanto de software que podem melhorar as métricas de custo.

- Integração 3D. A integração tridimensional permite que a integração entre as CPUs seja feita na vertical também diminuindo a distância entre os blocos, porém esta prática traz desafios significantes aos fabricantes de hardware.

- Novos materiais e processos. Incentivo a pesquisas que busquem novos materiais para os semicondutores que gastem menos energia e façam a comunicação de forma mais eficiente.

- Luz, não elétrons. A condução de elétrons dentro dos semicondutores possui uma baixa velocidade se comparada à velocidade da luz. Uma alternativa seria usar luz, ao invés de elétrons na comunicação, porém cm esta modificação surge o problema do gasto da conversão entre eletricidade e luz.

- Aceleradores de propósito especiais. Hardware especializado em alguns tipos de operações, de vetores, por exemplo, de criptografia, compressão de arquivos, etc. Hardware especializado não é barato e pode trazer gasto de energia quando não utilizados.

- Software paralelo existente. Softwares paralelos existem no mercado a mais de um quarto de século, softwares já testados e aprovados pela comunidade, e estes resolvem boa parte dos problemas de programação paralela e não custa nada usa-los.

Os custos das operações de hardware são altos e impõem limitações à programação paralela e seu avanço. Isto mostra o grande problema decorrente da comunicação entre threads e mostra que uma aplicação paralela é mais eficiente se suas threads forem mais independentes, ou seja, houver o mínimo de comunicação entre elas.

2.3. Software paralelo

Esta seção traz algumas noções básicas para programação paralela. O primeiro conceito que deve ser entendido é o de processo. Um processo de software é responsável por executar uma aplicação. Um processo pode ter processos filhos que são responsáveis por uma parte independente da aplicação, manipulando arquivos, segmentos de memória e recursos, por exemplo. É importante notar que processos não compartilham memória entre si e nem entre os seus processos filhos.

Dentro de um processo é permitida a criação de uma ou mais *threads*. Estas compartilham memória entre si e podem ser executadas em paralelo, tudo dentro do mesmo processo. As variáveis compartilhadas dentro das *threads* criam uma condição de corrida entre elas, por isso esta memória compartilhada deve ser cuidadosamente manipulada, necessitando de um mecanismo que garanta um meio seguro para este acesso.

Para evitar a condição de corrida entre as *threads* foi criado o conceito de *lock* que permite que uma variável seja adquirida por uma thread e usada por ela de forma exclusiva, sem a possibilidade de outra thread manipular a mesma variável. Para que a variável fique novamente disponível, a thread detentora do *lock* deve liberá-la após o seu uso, enquanto esta liberação não acontecer as demais threads ficam na espera. Este é chamado de *lock* exclusivo.

Existe o conceito de *lock* que apenas adquire o direito de escrever a variável, permitindo a leitura da mesma pelas demais *threads* concorrentemente. Neste caso o *lock* não é exclusivo e para a instrução de leitura não é necessário o uso do *lock*, apenas para a escrita. Este conceito deve ser usado com muito critério e num contexto não crítico da aplicação, ou seja, onde a leitura incorreta de uma variável não implicará em grandes transtornos durante a execução. Portanto, recomenda-se que em áreas de risco, o *lock* exclusivo seja usado.

O programador deve ser criterioso e estudar bem os conceitos antes de decidir que abordagem utilizar em seu projeto de software. Deve saber exatamente que tipo de algoritmo deseja criar e se realmente é necessário o uso de paralelismo. Os problemas causados pela comunicação entre threads não são dependentes apenas do hardware, mas também das estruturas de software escolhidas pelo programador.

3. Metodologia

O estudo sobre a transição para arquitetura “multicore” consistiu em uma mineração do repositório de um projeto que passou por uma modificação de arquitetura para tirar proveito dessas arquiteturas. O objetivo da mineração de um repositório de software é extrair informação pertinente a partir de repositórios, analisar informações e entregar conclusões dentro do contexto de evolução do software [3]. A metodologia do estudo consistiu em três etapas básicas.

3.1. Busca de repositório

O primeiro passo pra a realização do estudo foi encontrar um repositório. Foi decidido buscar um sistema em Java pelo fato de75% dos projetos nessa linguagem utilizarem programação concorrente [6]. Assim, o repositório desejado deveria atender às seguintes características:

1. Ser um sistema de grande porte, Java, *open source* e que possua uma boa lista de discussões, bem como um *bugtracking* atualizado.

2. Haver passado por uma modificação de arquitetura que caracteriza a adoção de estruturas paralelas ao invés de sequenciais e que afetem várias classes.

A procura do repositório foi feita dentro da comunidade do Github que conta com muitos projetos *open source* conhecidos e de fácil manipulação e recuperação de dados. A princípio foram feitas buscas usando palavras chaves como: *concurrent*, *synchronized*, *paralelism* e *thread*. Porém o resultado não foi satisfatório, pois a busca devolvia um grande numero de resultados e os projetos encontrados não atendiam as características acima. Como o Github foi criado em 2008 os projetos hospedados lá não tem muito tempo de vida e isto reduz as chances de encontrar um bom repositório na pesquisa.

A segunda opção de busca foi analisar projetos conhecidos por adotarem paralelismo e realizar uma pesquisa direcionada aos repositórios desses projetos. Utilizando o conhecimento de especialistas em projetos Java que indicaram algumas possibilidades de sistemas. Alguns foram testados como o Apache/lucene-solr e o Netflix/RxJava. Porém os resultados mais uma vez não atenderam as expectativas.

A terceira e melhor opção foi analisar projetos Java de grandes empresas como Google, Amazon e Twitter e projetos Java mais conhecidos no mundo do código aberto. Foi feita uma pesquisa com os projetos mais populares que estavam hospedados no Github e o resultado da pesquisa foi bem satisfatório.

Dado que um projeto que atendia à característica número um foi encontrado, uma busca também foi realizada dentro do repositório, especificamente no *bugtracking* do

projeto em busca das palavras chaves: *concurrent*, *synchronized*, *paralelism* e *thread*. Esta busca especifica é feita para encontrar um item que representa a modificação que procuramos. Este item deve atender a característica número dois do repositório desejado e compreende a modificação que será estudada.

O passo seguinte consistiu em identificar os módulos afetados pela modificação e determinar o momento da mesma dentro do projeto. Uma modificação grande é inicialmente proposta e passa pelo crivo da comunidade para ser aceita, ganha melhorias e recebe sugestões sendo resubmetida. Assim, é importante identificar o momento inicial da proposta, bem quanto tempo ela levou para ser aceita no projeto, se houve propostas anteriores que desencadearam a estrutura que foi incorporada.

Após a identificação dos módulos, o passo seguinte é a análise do código antes e depois das alterações visando à utilização das estruturas concorrentes em substituição às estruturas sequenciais. As diferenças de código, mais comumente chamadas de *diff*, são disponibilizadas junto com a descrição e discussão de um item no Github e são elas que serão analisadas neste passo.

3.2. Análise de bugs

Tendo em mãos o momento em que as alterações foram admitidas ao projeto, a segunda etapa do estudo é a análise de *bugs*, que consiste em realizar um levantamento de todos os itens do *bugtracking* que foram reportados nos seis meses anteriores e posteriores à modificação. Este levantamento é feito a fim de mapear os problemas relacionados à paralelização do código e também os seus consertos [7].

Antes de olhar cada item em particular é necessário fazer uma classificação quanto ao tipo, categoria, status e relação com a transição. A necessidade desta categorização surgiu durante a análise, e foi proposta por nós para melhor mapear todos os itens de modo a relacioná-los entre si.

Existem dois tipos de itens levando em consideração a classificação feita pelo próprio Github, *issue* e *pull request*. O primeiro é um relato de possível problema do projeto com uma descrição que não possui nenhum código relacionado com a resolução, *issues* são apenas descritivos e pode gerar discussões que avaliam se aquele relato é realmente um problema e como pode ser solucionado.

Pull requests são propostas de alteração do código do projeto, seja para melhorar algum ponto ou resolver um *bug* reportado ou não, ou seja, o *pull request* pode reportar um problema e já propor a solução. Esta proposta pode gerar discussões que levam a correções na proposta e, se aprovada pela comunidade, esta alteração é incorporada ao código do repositório do projeto. Os *pull requests* também possuem descrições que foram analisadas durante o levantamento, semelhantemente às *issues*.

Algumas categorias foram criadas para avaliar estes itens (*issues* e *pull requests*), problema, conserto, problema resolvido, discussão, funcionalidade, dúvida, melhoria. Um item é avaliado como "problema", se este reporta um problema real do sistema que não foi consertado ainda. Um "conserto" representa um reparo de um problema reportado anteriormente. Um item é avaliado como "problema resolvido" quando representa um problema que já foi resolvido, neste caso está relacionado a um item de conserto. Um problema resolvido também representa problemas além do código que já foi consertado, como por exemplo, um erro no site do JUnit, que diz respeito ao projeto, mas não ao código do sistema.

Além de itens relacionados a problemas no sistema do JUnit outras categorias surgem na lista. Alguns itens são criados para promover um debate sobre alguma ideia nova para o projeto, ou sobre melhorias que podem ser feitas. Assim várias pessoas opinam sobre o assunto até chegarem a um acordo. Um item avaliado como "funcionalidade" representa um novo comportamento do sistema, ele é caracterizado pela adição de novas classes ou módulos dentro do projeto. Os itens avaliados como "dúvida" são abertos com o intuito de saber como usar alguns recursos do JUnit para determinados fins. As "melhorias" são reparos no código, estilo de código, legibilidade ou documentação, são caracterizados por modificações em classes já existentes e que não trazem mudança de comportamento ao sistema.

Todos os itens que são cadastrados no Github possuem um status que pode ser *open*, *closed* ou *merged*. Quando um item é aberto ele fica com o status open e permanece assim até que alguém finalize a discussão sobre o item marcando-o como *closed* ou *merged*. O status *closed* se aplica às *issues* que foram resolvidas ou tiveram sua discussão encerrada, pois se chegou a uma conclusão sobre o assunto, e também a *pull requests*, quando as modificações propostas não foram aceitas, precisam de retrabalho ou quando o usuário que propôs a alteração prefere postergar a integração da sua proposta. O status *merged* se aplica apenas a *pull requests* quando os mesmos são introduzidos ao código com sucesso.

Todos os itens foram também são avaliados quanto à relação dos mesmos com o estudo. O item tem relação com a transição se o mesmo trata dos módulos que foram afetados, ou seja, o módulo de notificações do JUnit ou se o item trata de discussões sobre paralelismo, concorrência e estruturas concorrentes. Os demais não tem relação com a modificação ou com o estudo.

Tendo estruturado a classificação dos itens, é necessário varrer a lista de itens e classificar cada item de acordo com as descrições acima explicadas. Para este estudo o levantamento dos *bugs* foi feito à mão, ou seja, cada item é acessado pelo menos uma vez e é lida sua descrição e alguns comentários posteriores, o suficiente para determinar a sua categoria. Quando *pull request*, é necessário olhar a lista de arquivos modificados e criados também. Para determinar a relação do item com o nosso estudo, vamos chamá-los de "itens de interesse", são necessárias duas verificações:

1. O item tem relação com os módulos afetados pela modificação?

2. O item tem relação com o objeto do estudo, paralelismo ou concorrência?

Para determinar se o item se encaixa no primeiro ponto é preciso que uma busca superficial por palavras chaves como "*RunNotifier*", "*notifier*", "*listeners*", "*Runner*" seja feita, se algum desses termos for encontrado, então o item se encaixa na primeira descrição. Para determinar se o item corresponde ao segundo ponto, é feita uma busca superficial à procura de termos como, "*parallel*", "*concurrent*" e "*threads*", se encaixando no ponto quando a busca exibir resultados positivos. Quando um item se trata de um *pull request* a mesma busca também é feita sobre o código que foi modificado, a fim de classificar se o item é ou não relacionado ao estudo. As demais classificações são tiradas do próprio status oferecido pelo Github.

3.3. Análise temática

A última etapa do estudo foi realizar uma análise temática das discussões originadas da proposta inicial de transição. A análise de temas é um exame dos principais tópicos que estão sendo tratados dentro de um conjunto de dados. É focada nas percepções dos participantes, sentimentos e experiências. Num conjunto de dados, vários temas são abordados. Um tema dá significado a esses dados [8].

O processo para realizar a análise temática consiste em seis fases. A primeira fase é reservada à familiarização dos dados pelo analista. Nesta fase recomenda-se que o analista leia todos os dados pelo menos duas vezes para se familiarizar com os termos e linguagem usados, bem como os participantes e seus papéis dentro da discussão. A segunda fase trata da geração inicial dos identificadores dos dados. Esta fase tem como resultado uma lista de palavras-chave que são extraídas dos dados, palavras que possuem um padrão repetido. Esta lista deve conter o identificador e seu significado. É importante que mesmo que os identificadores pareçam irrelevantes eles sejam adicionados à lista.

Depois de completada a lista de identificadores no terceiro passo é preciso fazer uma combinação de identificadores de forma a criar temas com eles. Juntando palavras que possuem o mesmo sentido, de acordo com os seus significados. Os temas não necessitam estar escritos.

O quarto passo tem como objetivo revisar os temas propostos no passo anterior e revisar a combinação dos identificadores procurando por temas. Os temas criados devem ser mutuamente exclusivos, ou seja, os identificadores podem pertencer apenas a um tema. Neste passo devem-se relacionar os temas com os identificadores dentro do conjunto de dados e verificar se os mesmos são coerentes com o conjunto. Caso haja incoerência, o passo deve ser repetido.

Dado que foi feita a divisão dos identificadores por temas, o quinto passo trata de definir e nomear os temas e identificar a essência do tema, ou seja, do que o tema consiste. O sexto passo é a produção do relatório com os temas levantados na análise. Antes, porém da produção do relatório é necessário refinar ainda mais os temas e apresentar a análise iniciando com os temas mais importantes e explicando os resultados.

4. Resultados

Os resultados da aplicação da metodologia apresentada no capítulo 3 do trabalho estão descritos neste capítulo. Cada seção deste capítulo representa uma seção do capítulo anterior e cada etapa do processo é detalhada nas subseções.

4.1. Busca de repositório

Dentre os projetos que se encaixavam nas características propostas na seção 3.1, o JUnit foi escolhido pela sua relevância na comunidade e também pelo seu largo uso pelos desenvolvedores Java [9]. O JUnit é um framework de teste orientado a programação para Java [10]. Dentro do Github é um dos projetos mais populares em Java, sendo que é a segunda biblioteca mais usada entre os projetos Java hospedados na comunidade [5].

4.1.1. Identificação dos módulos

A alteração foi proposta dentro de uma requisição chamada de *pull request*, esta alteração foi descoberta através da busca feita dentro do *bugtracking* do projeto. A proposta foi aberta no dia 3 de fevereiro de 2013 com o título: ***Make RunNotifier code concurrent #625***, porém as discussões iniciais sobre a transição foram iniciadas em um *pull request* anterior aberto no dia 19 de dezembro de 2012 com o título: ***Improvement: Concurrent Queue in RunNotifier instead of Old Synchronized ArrayList #578***. A modificação foi incorporada ao projeto de fato no dia 6 de março de 2013 [11]. Esta informação foi tirada do *commit* que integra as alterações ao repositório. A evolução da proposta está descrita na Figura 1.

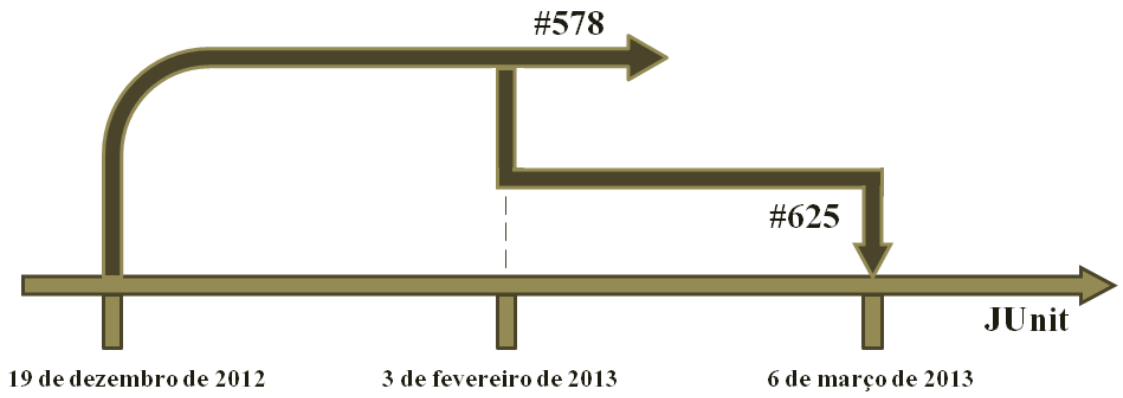

Figura 1 – Evolução da proposta de modificação ao projeto

As classes afetadas pela alteração fazem parte do módulo de notificações. Este módulo é responsável por notificar os eventos dos testes que estiverem em execução.

Um teste pode disparar alguns eventos durante sua execução, por exemplo: falha, sucesso, término.

A classe `Test` representa um teste no JUnit. Cada teste está associado a um `Runner`, que é uma interface que gerencia a execução de um ou mais testes. O `Runner` é responsável por executar os testes e gerar as notificações dos eventos significantes dos mesmos. Um `Runner` pode executar os testes associados em paralelo.

Cada `Runner` possui uma central de notificações, o `RunNotifier`, responsável por gerenciar as notificações e também os ouvintes do teste que são chamados de `RunListeners`, que são interfaces em Java. Estes ouvintes são cadastrados no `RunNotifier` para receberem as notificações de evento dos testes em execução. Estes módulos e o fluxo das notificações estão descritas na Figura 2.

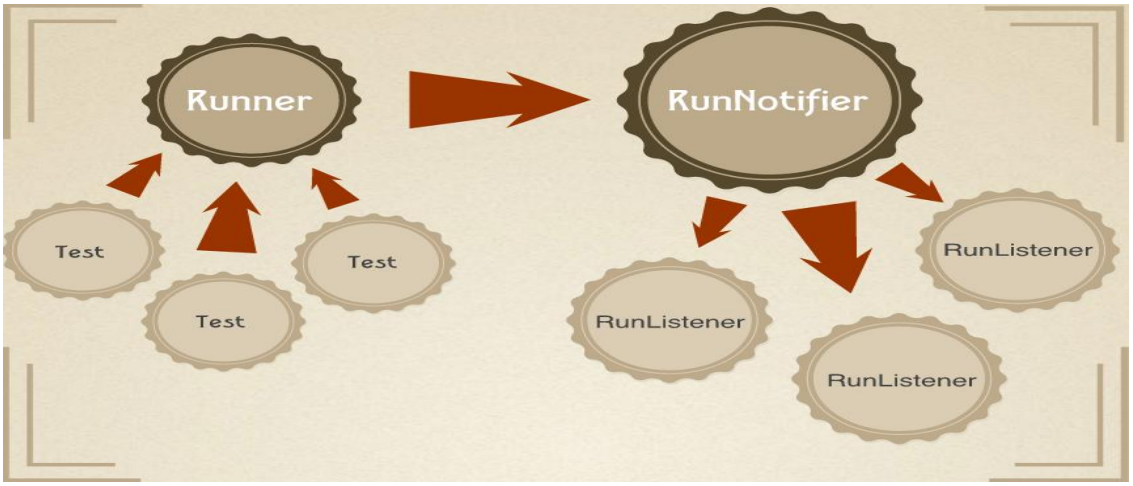

Figura 2 - Descrição de módulos e fluxo de notificações

O módulo de testes que cobrem o módulo de notificações também foi alterado. Todas as classes dentro do projeto do JUnit possuem testes associados e para as alterações serem aceitas era necessário a criação de novos testes e modificações.

4.1.2. Análise da transição

Antes da transição de arquitetura, a central de notificações matinha uma lista de ouvintes numa estrutura básica de lista em Java, os `RunListeners` que estavam cadastrados para receberem as notificações do teste, e quando o teste disparava um evento, o `RunNotifier` criava um bloco `synchronized` e percorria a lista de ouvintes notificando um a um dos ouvintes. A Figura 3 mostra como o `RunNotifier` realizava a notificação.

```
53          void run() {
54              synchronized (fListeners) {
55                  List<RunListener> safeListeners = new ArrayList<RunListener>();
56                  List<Failure> failures = new ArrayList<Failure>();
57                  for (Iterator<RunListener> all = fCurrentListeners.iterator(); all
58                          .hasNext(); ) {
59                      try {
60                          RunListener listener = all.next();
61                          notifyListener(listener);
62                          safeListeners.add(listener);
63                      } catch (Exception e) {
64                          failures.add(new Failure(Description.TEST_MECHANISM, e));
65                      }
66                  }
67                  fireTestFailures(safeListeners, failures);
68              }
69          }
```

Figura 3 - Notificação do RunNotifier antes da modificação

O bloco sincronizado era executado a cada evento, assim os eventos deviam esperar que o bloco de notificação estivesse liberado para realizar a nova notificação. Desta forma um `RunListener` não podia ser notificado concorrentemente quando os testes estão rodando em paralelo e sim pela mesma thread que estiver executando os testes, cada evento de uma vez.

A adoção de paralelismo consistiu em fazer com que esses ouvintes possam ser notificados paralelamente assim como os testes podem ser executados em paralelo. Desta forma o notificador, ao receber um evento de teste, percorre uma estrutura de lista concorrente de `RunListeners` e notifica cada um deles deixando a sincronização a cargo dos mesmos, como mostrado na Figura 4.

```
68          void run() {
69              int capacity = fCurrentListeners.size();
70              ArrayList<RunListener> safeListeners = new ArrayList<RunListener>(capacity);
71              ArrayList<Failure> failures = new ArrayList<Failure>(capacity);
72              for (RunListener listener : fCurrentListeners) {
73                  try {
74                      notifyListener(listener);
75                      safeListeners.add(listener);
76                  } catch (Exception e) {
77                      failures.add(new Failure(Description.TEST_MECHANISM, e));
78                  }
79              }
80              fireTestFailures(safeListeners, failures);
81          }
```

Figura 4 - Notificação do RunNotifier depois da modificação

Para que este comportamento fosse possível, foi criada uma anotação (`@ThreadSafe`) que deve ser sinalizada no `RunListener`. O `RunNotifier` então verifica se o ouvinte possui esta anotação e decora a classe. Ou seja, ela recebe uma implementação adicional que vai gerenciar as suas chamadas de notificações. Este decorador é a classe `SynchronizedRunListener` (Figura 5).

```
47      /**
48       * Wraps the given listener with {@link SynchronizedRunListener} if
49       * it is not annotated with {@link RunListener.ThreadSafe}.
50       */
51      RunListener wrapIfNotThreadSafe(RunListener listener) {
52          return listener.getClass().isAnnotationPresent(RunListener.ThreadSafe.class) ?
53                  listener : new SynchronizedRunListener(listener, this);
54      }
```

Figura 5 - RunNotifier decora RunListener com a classe SynchronizedRunListener

A classe `SynchronizedRunListener` estende a interface do `RunListener`, proporcionando a ela, o controle das notificações que recebe. Desta forma o bloco `synchronized` é retirado do notificador e movido para dentro do `RunListener`, por meio do decorador, que ficará responsável por sincronizar suas próprias notificações (Figura 6).

```
32      @Override
33      public void testRunStarted(Description description) throws Exception {
34          synchronized (fMonitor) {
35              fListener.testRunStarted(description);
36          }
37      }
```

Figura 6 - RunListener sincroniza suas notificações por meio da classe SynchronizedRunListener

4.2. Análise de bugs

Dado que as modificações foram incorporadas ao projeto no dia 3 de fevereiro de 2013, o levantamento dos *bugs* surgidos nos seis meses anteriores e posteriores a esta data foi feito, como descrito anteriormente (seção 3.2), com base na lista de *issues* e *pull requests* do repositório do JUnit no Github, esta lista é o *bugtracking* do projeto. Dentro da lista, o id que marca a transição é o #625, o item da lista que marca seis meses antes da transição é o #497 e o que marca seis meses depois é o #731. O levantamento dos *bugs* está neste intervalo de itens.

No levantamento de *bugs* para os seis meses anteriores à transição foram analisados 128 itens a partir do item #497 datado no dia 4 de setembro de 2012. A classificação geral desses itens está descrita na Tabela 1.

Classificação		**#Itens**
Tipo	ISSUE	44
	PULL REQUEST	84
Categoria	PROBLEMA	5
	CONSERTO	35
	PROBLEMA CONSERTADO	23
	DISCUSSÃO	13

	FUNCIONALIDADE	17
	DÚVIDA	8
	MELHORIA	27
Status	OPEN	26
	CLOSED	49
	MERGED	53
Relacionada	SIM	19
	NÃO	109

Tabela 1 - Classificação dos itens anteriores à modificação

No levantamento de *bugs* para os seis meses posteriores à transição foram analisados 106 itens, até o item #731 datado no dia 3 de setembro de 2013. A classificação geral desses itens está descrita na tabela 2.

Classificação		**#Itens**
Tipo	ISSUE	43
	PULL REQUEST	46
Categoria	PROBLEMA	5
	CONSERTO	14
	PROBLEMA CONSERTADO	13
	DISCUSSAO	12
	FUNCIONALIDADE	13
	DÚVIDA	17
	MELHORIA	30
Status	OPEN	29
	CLOSED	37
	MERGED	37
Relacionada	SIM	8
	NÃO	96

Tabela 2 - Classificação dos itens posteriores à modificação

4.2.1. Identificação dos itens de interesse

A etapa seguinte do processo do estudo é extrair os itens que tem relação com a modificação dos demais itens e fazer a sua classificação, ou seja, extrair os itens de interesse, ver Seção 3.2. No estudo foram identificados 29 itens de interesse e para estes a classificação se encontra na Tabela 3.

Classificação		#Itens
Tipo	ISSUE	9
	PULL REQUEST	20
Categoria	PROBLEMA	5
	CONSERTO	3
	PROBLEMA CONSERTADO	4
	DISCUSSAO	7
	FUNCIONALIDADE	2
	DÚVIDA	8
Status	MELHORIA	4
	OPEN	12
	CLOSED	13

Tabela 3 - Classificação dos itens de interesse

Estes itens vão desde atualização de documentação para questões de concorrência até a adoção de estruturas concorrentes no projeto. Os itens relacionados estão divididos em três categorias para uma melhor análise dos resultados: itens de modificação, itens de módulos afetados e itens de paralelismo.

Os itens de modificação compreendem os itens que estão diretamente relacionados à transição para estruturas paralelas. Esta categoria será analisada mais profundamente na próxima subseção deste capítulo. Os itens de módulos afetados consistem nos itens que tem relação com os módulos afetados, antes e depois da modificação abordada na subseção 4.1.2 e que não necessariamente tratam de concorrência e paralelismo. Estes estão descritos na Tabela 4.

Item ID	Título
#606	Fix for `Internal use only` in RunNotifier
#644	RunListener testRunStarted method not will not be run.
#713	Unit Tests are running concurrently

Tabela 4 - Descrição dos itens de módulos afetados

O item #606 trata-se de um *pull request* propondo que um método interno do `RunNotifier` se tornasse público, porém a proposta de modificação foi rejeitada visto que uma necessidade da aplicação teria que ser tratada pelo usuário, o que não satisfez os avaliadores do código. Os itens #644 e #713 tratam de dúvidas quanto à utilização de testes em paralelos. O primeiro foi resolvido e o segundo ainda está em aberto esperando por mais detalhes do usuário, desde agosto de 2013.

Os itens de paralelismo referem-se a itens que se encaixam na segunda descrição apresentada na seção 3.2 e envolve paralelismo e concorrência, tanto para conserto de

erros quanto para melhorias ao código, que não tratam dos módulos afetados pela modificação. Nesta categoria estão 25 itens. Ao ler a descrição dos itens desta categoria foi possível agrupá-las, pois muitos itens se tratam de soluções ou discussões do mesmo problema. Os itens descritos na Tabela 5 são os itens mais importantes desses grupos, aqueles que representam as soluções finais, ou os itens que expõem o problema, caso o mesmo ainda não tenha uma solução.

Alguns itens tratam de pequenas modificações no projeto, como documentação, estilo de código, pequenas dúvidas e resoluções, estes não foram agrupados e nem descritos na Tabela 5. Diante disto, foram levantados sete grupos, dos quais os itens mais relevantes estão descritos.

Item ID	Título
#510	Improved implementation of ParallelComputer
#540	org.junit.runner.Result : Treatment Of Long Variables and thread-safe constructor for List
#544	forked from #450 and improved
#569	Measure runtime in Stopwatch rule for Performace tests
#617	Abstraction for parallel computer
#640	Issue #162 Implementation
#666	Treatments for parallel execution

Tabela 5 - Descrição dos itens de paralelismo mais relevantes

O item #510 faz parte de um grupo que propõe melhorias aos testes e a uma classe experimental que tem por objetivo computar paralelamente os testes que serão rodados usando o framework. Já o item #540 trata de uma mudança na classe `Result` fazendo com que suas estruturas sejam *thread safe*.

O item #544 faz parte de um grupo que aplica mudanças na regra de Timeout do JUnit. Que faz com que o teste falhe se o mesmo extrapolar um limite de tempo de execução. Estas mudanças fizeram com que o timeout fosse usado corretamente quando testes estivessem sendo executados em paralelo. Assim foram adicionadas estruturas concorrentes às classes relacionadas, e também houve um melhoramento nos testes para abrangerem os casos com testes concorrentes.

O item #569 é uma continuação do item #552 e trata de uma funcionalidade que permite contar o tempo de execução de um teste. Esta funcionalidade foi pensada de modo a computar o tempo de testes concorrentes também. Esta funcionalidade foi proposta a amplamente aceita sendo facilmente incorporada ao JUnit.

O item #617 é o desfecho de um conjunto de discussões e *pul requests* sobre uma possível flexibilização da computação paralela dentro do JUnit. A discussão foi baseada numa implementação de um usuário e sugerida para o código do JUnit, entretanto após

muitas ponderações foi decidido que a modificação causaria um impacto grande ao projeto e que poderia ser melhor aproveitada se isto fosse uma extensão do projeto, ou seja, ficou acordado que esta funcionalidade seria incorporada ao projeto em forma de *plugin* ou biblioteca adicional.

O item #640 trata da resolução de um problema reportado no item #162 que consiste numa sugestão de aplicação de uma estrutura concorrente para melhoramento de desempenho para testes parametrizados.

O item #666 traz uma série de melhorias com respeito à concorrência, pois altera várias assinaturas de variáveis e métodos para garantir a correta sincronização dos mesmos. Dentro desta proposta há mudanças de estruturas de lista simples para listas concorrentes e adição de blocos sincronizados em várias classes do projeto. A proposta visa reduzir o risco de erros e aplicar corretamente as estruturas concorrentes dentro do seu devido contexto.

4.3. Análise temática da transição

Para a análise temática deste estudo foram levados em consideração os itens de modificação, definidos na subseção 4.2.1, que são os itens que compõem as propostas de modificação e estão descritos na tabela 6. É importante ressaltarmos que a quantidade de itens de interesse, sejam *issues* ou *pull requests*, encontrados do período estudado, é baixo e não permite que análises estatísticas sejam feitas. A quantidade de itens não permite análises descritivas ou que relacionem duas ou mais variáveis.

Item ID	Título
#528	fixed and improved Java Concurrency in RunNotifier
#578	Improvement: Concurrent Queue in RunNotifier instead of Old Synchronized ArrayList
#625	Make RunNotifier code concurrent
#641	Twaeks on pull requests # 625 and # 578

Tabela 6 - Descrição dos itens de modificação

O item #528 é um *pull request* que foi revertido por ter sido integrado antes do fim das revisões, existe uma descrição sobre a proposta inicial e algumas discussões sobre as estruturas utilizadas. Já o item #641 é uma tentativa de trazer o melhor dos itens #578 e #625, porém ele foi fechado por ser redundante e desnecessário.

Assim, os itens #578 e #625 concentram a maior parte das discussões da proposta e a análise temática foi feita em cima dessas conversas. Dentro do conjunto de dados foi feito o mapeamento de identificadores e suas combinações para encontrar os temas. Os temas encontrados foram:

1. ***Features***. Tema que aborda a lógica de cada mudança nas classes, explicando seus objetivos, bem como o objetivo de cada classe criada. Este tema é representado pelas palavras `RunNotifier`, *notifier*, *notifications*, *listerners*, `RunListener`, por exemplo e aparecem logo no início das discussões sendo os primeiros termos a surgirem.

2. **Concorrência e Paralelismo**. Tema que trás explicações sobre concorrência, blocos de sincronização, garantia de *thread-safety* e execuções paralelas. Este assunto aborda questões mais teóricas sobre fundamentos de paralelismo e concorrência. É representado pelas palavras *synchronized*, *non-blocking*, *asynchronous*, *thread safe*, *parallel*, por exemplo. O tema aparece nas argumentações quando havia uma tomada de decisão.

3. **Estrutura de dados**. Tema abrange o debate sobre a melhor estrutura de dados de lista, da biblioteca `java.util.cocurrent`, a ser usada. Expressões como `ConcurrentLinkedQueue`, `CopyOnWriteArrayList`, *collection* caracterizam este tema. É um dos temas que mais aparece, sendo que foi tema da maior discussão da lista de comentários.

4. **Tipos de erros e riscos**. Tema propõe a discussão sobre prevenção de erros e garantia de que não haveria nenhuma quebra do fluxo de execução dos módulos. É caracterizado pelo uso das palavras *locks*, *block*, *unsafe*, *risk*, *broken*, *consistent*. Mostrou ser uma grande preocupação dos revisores do código.

5. **Testes**. Tema envolve os cenários mais apropriados a serem testados, bem como onde e como os testes deveriam ser executados. O tema é representado pelas palavras *tests*, *passing*, *number of listeners*, *failed*.

6. **Desempenho**. Este tema trás justificativas sobre o desempenho dos módulos após a transição, benchmarks que comprovam a aceleração da execução e argumentos que apoiam a aplicação da modificação. Representado pelas expressões *speed up*, *performance*, *benchmarks*, *real-world gains*. Trazem dados numéricos que os ganhos de velocidade dos módulos.

7. **Dependência de terceiros**. Tema mostra o debate sobre a utilização de uma estrutura de projeto terceiro ou criação de uma estrutura própria. Houve muitas argumentações dentro deste assunto, mas para que não existisse dependência de estruturas externas foi decidido criar uma estrutura própria. Algumas palavras que caracterizam o tema são: *dependency*, *license*, `jcip`, *annotations*.

8. **Melhorias e consertos**. Tema que expressa opiniões e questionamentos em cima das mudanças feitas, e também sugestões de melhoria de código baseadas na experiência dos revisores. Palavras que caracterizam este tema são *disagree*, *would*, *propose*, *improvement*, *solution*, *need*, *fixed*, *add*, *remove*. Este tema está espalhado por toda a base de dados.

9. **Colaboração no Github**. Este tema envolve explicação sobre *commits* e o que seria aceito no *pull requests*, como a integração aconteceria. O tema se encontra no fim das discussões, fechando a proposta com a integração da mesma ao código do JUnit. Algumas expressões que caracterizam o tema são *pull request*, *commit*, *collaboratively*, *merge*.

O idealizador da proposta foi o usuário @Tibor17 e os revisores que fizeram parte da discussão foram @kcooney e @dsaff Alguns temas já eram esperados pela natureza da transição e pela característica *open source* do projeto JUnit. Porém alguns temas merecem uma explicação mais aprofundada.

O tema "Estrutura de dados" aborda a questão de qual a melhor estrutura de dado concorrente a ser usada. Inicialmente foi proposto o uso da classe ConcurrentLinkedQueue da biblioteca `java.util.cocurrent`, e posteriormente foi sugerido o uso da classe CopyOnWriteArrayList que também pertence a biblioteca `java.util.cocurrent`. A proposta da segunda veio na seguinte afirmação.

"Unless listeners are added and removed often, a CopyOnWriteArrayList should be much faster. The read operations for COWAL are non-blocking; in iterator() it relies on a single volatile read to ensure safety; after that, it's just indexing off an array. In contrast, the CLQ might need to skip over nulls in the queue." por @kcooney.

Algumas argumentações e testes de desempenho foram feitas usando as duas estruturas. Uma delas está descrita a seguir.

"*I made tests in order to decide between COWAL/CLQ.*
The scenario
- *registering new listener*
- *notify the RunNotifier*
- *unregister listener*

3000 and 300 test-cases/threads
Result
- *the mean execution time of parallel runners is shorter with CLQ on all JVM versions*
- *bigger variations with CLQ*
- *on JVM 1.6 and 1.7 the variations getting same in both collections*"

por @Tibor17

Assim ficou decidido o uso do CopyOnWriteArrayList no seguinte comentário.

"The COWAL has method #add(int, Object), and the CLQ does not. Therefore parallel calls on #addListener and #addFirstListener may break with CLQ, which is still unlikely use. To be absolutely on safe side [...], we may accept COWAL, which avoids these doubts." por @Tibor17

O tema "Tipos de erros e riscos" trata da preocupação em quebrar usuários do JUnit com esta transição. O debate foi longo e foi iniciado pelo fato da retirada do bloco *synchronized* do `RunNotifier`, que prejudicaria os usuários que não suportam RunListeners *thread-safe*. Esta preocupação está indicada no seguinte comentário.

"My bigger concern is that calling existing listeners with no synchronization will break existing users in very subtle ways if their listeners are not thread-safe. You can't just say that those users should know better; they may be using third-party listeners. We have historically been very careful to not break existing users. Please consider my proposal for making existing listeners synchronized while allowing new listeners to be called by multiple threads concurrently." por @kcooney

A partir desta preocupação foi acordado que os `RunListeners` teriam um decorador caso estivessem marcados com uma anotação específica, deste modo as classes que não são *thread-safe* não sofreriam impactos na execução como está descrito no comentário.

"The prior RunNotifier synchronization would never call two RunListeners concurrently. So if I add two listeners that call common non-thread-safe code, everything will work fine with the old code and my code." por @kcooney

4.3.1. Discussão sobre a análise temática

A análise temática revelou uma grande preocupação dos participantes em usar as estruturas concorrentes que diminuíssem o risco de falhas após a incorporação do código ao sistema. A principal preocupação revelada é a de não quebrar os usuários existentes, nem de forçá-los a adotar uma estrutura nova, tendo que modificar sua implementação antiga. Esta preocupação está explícita no seguinte comentário.

"*Regarding breaking existing users. I'm not concerned about the methods being called in a non-sequential order; that was the case before. I'm concerned about thread-safety issues. I think that reasonable people might not consider the current synchronization an implementation detail. Someone had to go out of their way to add synchronization to RunNotifier, and the Javadoc doesn't mention that RunListener implementations need to be thread-safe. Whether or not the developer that wrote the listener should have known better, the developer who upgrades their project to JUnit 4.12 will be left tracking down seemly random behavior due to concurrency issues in the listeners.*" por @kcooney

Esta preocupação foi o ponto chave para todos os temas que surgiram. A partir desta preocupação, que só foi explicitamente revelada no meio das discussões, é que os demais assuntos foram desencadeados. Preocupação com testes e desempenho visaram atender necessidades dos usuários do JUnit, como mostrado na discussão a seguir.

"*Taking a step back, what are the expected real-world gains over the current code for either of the proposed solutions? Under what circumstances will the time saved be user-visible?*" por @dsaff

"*Having hunderds of short parallel test. If you use parallel computer in order to speed up all test rounds, the fireTestStarted and fireTestFinished is bottleneck. If the listener uses other than memory or file access, e.g. remote logger, or DB connection, then the parallel tests should slow down. The reason of paralle tests is to speed up.*" por @Tibor17

Os problemas relacionados à estrutura de dados e prevenção de *bugs* foram os mais importantes e a fim de não prejudicar os usuários. Sempre tendo a preocupação de trazer uma boa sensação aos usuários, para que eles pudessem perceber a diferença após as modificações. Assim a análise de tema revelou uma grande preocupação em evitar erros com a nova implementação, diminuir os riscos e aumentar o desempenho da aplicação. Infelizmente após a incorporação das mudanças as discussões também foram finalizadas.

5. Conclusão

A adoção de estruturas paralelas, dentro dos projetos atuais de software, é a solução esperada para suprir a demanda dos processadores multicore, com seu aumento de núcleos e velocidade de processamento. Este estudo teve como objetivo analisar e estudar um projeto que passou por este tipo de transição.

O estudo também propôs a mineração de um repositório de software, *open source*. O conhecimento sobre a comunidade de repositório, o Github, foi fundamental para uma mineração mais específica e detalhada do sistema, pois já sabíamos o que íamos procurar e o meio mais fácil de conseguir a informação desejada.

Neste trabalho foi proposta uma metodologia de mineração de repositório com foco em projetos que passaram a adotar estruturas concorrentes. A metodologia descrita no capítulo 3 foi apresentada por nós, de modo a facilitar o estudo de um projeto deste tipo, bem como extrair informações relevantes para a pesquisa.

Os resultados responderam alguns questionamentos que motivaram este estudo. A modificação ocorrida no JUnit foi uma alteração bem-sucedida, pois não apresentou nenhum crescimento de *bugs*, nem reclamação por parte dos usuários da biblioteca. Foi possível extrair das discussões no *bugtracking* do repositório o que levou esta modificação a obter este resultado, que foi a preocupação com o valor que seria gerado pela alteração e a prevenção insistente contra riscos e erros inesperados. Os revisores das alterações tiveram cuidados especiais também com os testes e com o desempenho do sistema, para que o mesmo não sofresse penalização.

As discussões revelaram dados interessantes a respeito de estruturas da biblioteca `java.util.cocurrent`, apresentando dados que comprovam as vantagens e desvantagens de cada estrutura. Uma questão que também ficou clara durante as discussões foi a preocupação com os usuários antigos do sistema, para que eles não sofressem nenhum tipo de retrabalho em suas implementações.

Por fim, este estudo traz à comunidade um método de mineração de repositório com o objetivo de extrair informações relevantes, de como grandes projetos se comportam diante de alterações que trazem paralelismo ao sistema. O estudo visou uma alteração que aconteceu com o sistema JUnit, apresentando suas características, decisões tomadas e seu impacto com relação aos *bugs* do projeto.

6. Referências

[1] Semih Okur, Danny Dig: “How do developers use parallel libraries?” SIGSOFT FSE 2012: 54

[2] Saleh M. Alnaeli, Abdulkareem Alali, and Jonathan I. Maletic: Empirically Examining the Parallelizability of Open Source Software System. WCRE, page 377-386. IEEE Computer Society, (2012)

[3] Huzefa H. Kagdi, Michael L. Collard, Jonathan I. Maletic: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance 19(2): 77-131 (2007)

[4] Paul E. McKenney: Is Parallel Programming Hard, And, If So, What Can You Do About It? 2011

[5] GitHub’s 10,000 most Popular Java Projects. Disponível em: http://www.takipiblog.com/2013/11/26/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/
Acesso em: Dezembro, 2013

[6] Weslley Torres Gustavo Pinto, Benito Fernandes, Fernando Castor, Roberto S. M. Barros: A Large-Scale Study on the Usage of Java’s Concurrent Programming Constructs. SOFTWARE - PRACTICE AND EXPERIENCE, page 1-36. (2013)

[7] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou: “Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics” , In the Proceedings of the 13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'08), March 2008

[8] Thematic analysis. Disponível em: http://en.wikipedia.org/wiki/Thematic_analysis
Acesso em: Janeiro, 2014

[9] junit-team/junit. Disponível em: https://github.com/junit-team/junit
Acesso em: Dezembro, 2013

[10] JUnit. Disponível em: http://junit.org/
Acesso em: Fevereiro, 2014

[11] Make RunNotifier code concurrent #625. Disponível em: https://github.com/junit-team/junit/pull/625
Acesso em: Dezembro, 2013