Um Estudo sobre Extração de Métodos para Reutilização de Código

Danilo Silva, Marco Túlio Valente, Eduardo Figueiredo

Universidade Federal de Minas Gerais, Departamento de Ciência da Computação,
Belo Horizonte – MG – Brasil
{danilofs,mtov,figueiredo}@dcc.ufmg.br

Resumo Refatoração de código é uma técnica amplamente utilizada e frequentemente estudada pela comunidade científica. Entretanto, ainda sabe-se pouco sobre os motivos que levam um desenvolvedor a refatorar o código. Neste trabalho, investigamos a relação entre a refatoração Extrair Método e a reutilização de código, com o fim de entender melhor as motivações por trás dela. Após analisar mais de 10 mil revisões de 10 sistemas de código aberto, encontramos indícios que, em 56,9% dos casos, Extrair Método é motivada pela reutilização de código. Além disso, em uma pequena porção desses casos (7,9%), a reutilização elimina código duplicado já existente no sistema. Por fim, também verificamos que existem casos nos quais a refatoração favoreceu a reutilização de código no longo prazo, mesmo sem ter essa intenção inicialmente.

1 Introdução

Refatoração de código é uma técnica amplamente conhecida e utilizada na manutenção e evolução de sistemas [10,3]. Devido a sua relevância, encontramos estudos empíricos na literatura relacionados a refatoração de código, que respondem questões como: com qual frequência desenvolvedores aplicam diferentes tipos de refatoração [8], qual a correlação entre atividades de refatoração e correções de *bugs* [1] e quão frequentemente refatorações são realizadas manualmente ou com suporte de ferramentas [9].

Entretanto, uma questão ainda pouco explorada é a motivação por trás da refatoração. Há um senso comum de que refatorações são motivadas principalmente por certos padrões de código conhecidos como *Bad Smells* [3], que sinalizam falhas no *design* do sistema. No entanto, faltam evidências empíricas que confirmem tal fato, especialmente para aquelas refatorações que podem servir para múltiplos propósitos. Extrair Método, que é uma das refatorações mais aplicadas na prática [8,9], possui tal característica. Por exemplo, Tsantalis et al. [14] encontraram nove motivações distintas para essa refatoração, algumas delas relacionadas a necessidade de extensão do sistema, e não a problemas existentes no código.

Por este motivo, este trabalho investiga a relação entre a refatoração Extrair Método e a reutilização de código. Suspeitamos que em muitos dos casos onde um método é extraído o intuito do desenvolvedor é reutilizar código. Por exemplo,

ao implementar uma nova funcionalidade, um desenvolvedor pode identificar um trecho de código de interesse existente no sistema, extrair esse código como um novo método e reutilizá-lo.

Mais especificamente, neste estudo investigamos três questões de pesquisa:

RQ1: Com qual frequência Extrair Método tem como motivação a reutilização de código?

RQ2: Com qual frequência Extrair Método tem como motivação a eliminação de código duplicado?

RQ3: Com qual frequência Extrair Método favorece a reutilização de código posteriormente, ainda que seu objetivo inicial não tenha sido esse?

Acreditamos que as respostas dessas questões podem contribuir para o melhor entendimento de como, e por quais motivos, as equipes de desenvolvimento refatoram o código. Tal conhecimento pode ser relevante para propor ou aperfeiçoar técnicas e metodologias empregadas no desenvolvimento de software. Em especial, pode ser possível aprimorar as ferramentas existentes especializadas na recomendação da refatoração Extrair Método [12,13], visto que novas heurísticas podem ser desenvolvidas para contemplar os cenários que ocorrem com mais frequência na prática.

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 descreve a metodologia utilizada no estudo. A Seção 3 apresenta os resultados obtidos e responde as questões de pesquisa levantada. A Seção 4 discute as ameaças a validade do estudo. A Seção 5 apresenta trabalhos relacionados encontrados na literatura. Por fim, a Seção 6 apresenta as conclusões e trabalhos futuros.

2 Metodologia

O processo utilizado neste estudo, ilustrado na Figura 1, pode ser dividido em três etapas:

1. Um conjunto de sistemas Java foi selecionado do GitHub, utilizando critérios pré-definidos.
2. As refatorações Extrair Método aplicadas nos sistemas foram detectadas com o auxílio de uma ferramenta. Para tal, cada alteração de código (*commit*) presente no histórico de versões foi analisada.
3. A contagem das invocações dos métodos extraídos foi obtida. Para tal, todas as revisões dos sistemas foram analisadas, identificando também os casos onde um método não é invocado na mesma revisão que ele é criado, mas sim posteriormente.

Todas as etapas do processo são feitas de forma automatizada, para permitir sua replicação e aplicação em uma grande massa de dados. Nas seções seguintes descrevemos os detalhes de cada etapa.

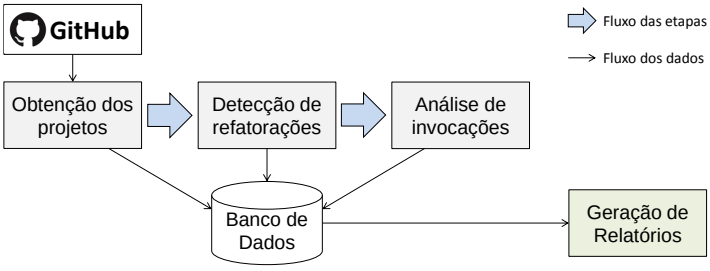

Figura 1. Visão geral da metodologia

2.1 Seleção dos Sistemas

A primeira etapa do estudo consistiu na escolha de sistemas e seus respectivos repositórios de código para análise. Para tal, utilizamos como fonte o GitHub, que é um site colaborativo de hospedagem de código baseado no sistema de controle de versões *git*. A escolha foi motivada pelo crescente relevância do GitHub, que conta com mais de 17 milhões de repositórios de código [1], em novembro de 2014.

Inicialmente, um conjunto de repositórios do GitHub foi levantado. Levando em conta boas práticas recomendadas na literatura [4], definimos os critérios de seleção listado abaixo:

- Os projetos devem ter Java como a linguagem primária, por limitações das ferramentas de análise utilizadas.
- Os projetos devem ter no mínimo seis meses de desenvolvimento, para evitar projetos que não tenham passado por um tempo de manutenção relevante.
- Os projetos devem ter no mínimo 200 revisões, pelos mesmos motivos da restrição anterior.
- Os projetos não devem ser ramificações (*forks*) de um outro projeto, para evitar dados duplicados.
- Os projetos obtidos devem ser os 100 mais populares que atendem aos demais critérios, utilizando como métrica o campo `stargazers_count`.

Do conjunto dos 100 projetos obtidos, selecionamos uma amostra de 10 sistemas para realizar este estudo, devido a limitações impostas pelo longo tempo necessário para executar a análise nos dados completos. A Tabela 1 apresenta a lista de sistemas selecionados e uma breve descrição de cada um deles. Note que entre eles podemos encontrar projetos conhecidos como o *framework* de testes *JUnit* e a linguagem de programação *Clojure*.

2.2 Detecção de Refatorações

Cada alteração de código presente no histórico de versões foi analisada para identificar as alterações entre duas revisões consecutivas do sistema. Para evitar

[1] `https://github.com/features`

Tabela 1. Sistemas analisados

Sistema	Descrição
android	GitHub Android App
android-async-http	An Asynchronous HTTP Library for Android
clojure	The Clojure programming language
facebook-android-sdk	Used to integrate Android apps with Facebook Platform
jsoup	jsoup: Java HTML Parser, with best of DOM, CSS, and ...
junit	A programmer-oriented testing framework for Java.
picasso	A powerful image downloading and caching API for Android
spark	A Sinatra inspired framework for java
storm	Distributed and fault-tolerant realtime computation ...
yuicompressor	YUI Compressor

a duplicação de informações, foram excluídas da análise aquelas alterações que representam uma operação de *merge*, ou seja, a junção de duas ou mais alterações na linha principal de desenvolvimento.

As refatorações realizadas em cada alteração foram detectadas utilizando a mesma técnica adotada por Tsantalis et al. [14], que por sua vez é baseada no algoritmo UMLDiff [15] e nas regras de detecção propostas por Biegel et al. [2]. Utilizamos uma implementação existente da técnica, com algumas adaptações para permitir seu uso na ausência do código compilado. Isso foi necessário pois compilar o código é uma tarefa que depende muito de especificidades de cada projeto, além de ser muito dispendiosa em termos de tempo.

Embora a ferramenta utilizada detecte 11 tipos diferentes de refatoração, nosso interesse recai na refatoração Extrair Método, cujas regras de detecção podem sem descritas da seguinte forma. Seja $M^{=}$ o conjunto de métodos que permaneceram entre duas revisões sucessivas de código v e v'. Analogamente, seja M^{+} o conjunto de métodos que foram adicionados entre as duas revisões. Um método m_j é extraído de outro método m_i quando as quatro condições a seguir são verdadeiras:

- $\exists\, m_i \in M^{=}$
- $\exists\, m_j \in M^{+}$
- $b_j \subseteq b_i$
- $m_j \in b'_i$

onde b_i é corpo de m_i antes da refatoração e b'_i é o corpo de m_i após a refatoração. Neste caso, o corpo b_i de um método m_i é representado pelo conjunto de membros (métodos e atributos) que são acessados no mesmo. Caso as condições acima sejam satisfeitas para mais de uma tupla (m_i, m_j) para um mesmo m_j, isso significa que o método foi extraído de mais de um local do sistema. Ou seja, o código encapsulado por m_j estava duplicado em cada m_i.

2.3 Contagem das Invocações de Métodos

Para detectar o número de invocações dos métodos de interesse, foi desenvolvida uma ferramenta baseada na API JDT Core, que é a API utilizada pela IDE Eclipse para analisar e manipular código Java. Tal API permite analisar código no nível de nodos da Árvore Sintática Abstrata (AST), possibilitando encontrar cada invocação de método existente no código.

Além disso, a API provê um mecanismo de resolução para vincular uma invocação de método a sua respectiva declaração. Vale ressaltar que não estamos interessados em invocações de métodos externos, que são definidos em API's de terceiros. Os métodos de interesse, ou seja, aqueles que foram refatorados, estão sempre definidos no próprio projeto. Isso viabiliza o uso do JDT mesmo na ausência de todas as dependências de compilação.

3 Resultados

Nesta seção descrevemos os resultados obtidos com a análise do histórico de versões dos sistemas levantados. Especificamente, 10.931 commits foram analisados ao longo do processo, conforme detalhado na Tabela 2. Adicionalmente, a tabela apresenta: o número de métodos distintos analisados (coluna 3) e o número de métodos distintos criados a partir da refatoração Extrair Método (coluna 4). Note que tais métodos representam apenas 2,2% de todos os métodos avaliados.

Tabela 2. Estatísticas dos sistemas analisados

Sistema	**Commits**	**Métodos Analisados**	**Métodos Extraídos**
android	2.351	3.386	100 (3,0%)
android-async-http	557	726	34 (4,7%)
clojure	2.629	6.276	122 (1,9%)
facebook-android-sdk	451	4.840	144 (3,0%)
jsoup	711	1.760	35 (2,0%)
junit	1.611	5.822	107 (1,8%)
picasso	461	1.357	41 (3,0%)
spark	281	738	19 (2,6%)
storm	1.491	4.987	38 (0,8%)
yuicompressor	388	269	9 (3,3%)
Total	10.931	30.161	649 (2,2%)

É importante ressaltar que o número de métodos analisados é sempre maior ou igual ao número de métodos total na versão atual de um sistema. Isso ocorre pois, ao longo da evolução do código, métodos são criados e também removidos. Mais especificamente, dos 30.161 métodos analisados, 17.707 (58,7%) deles

existem nas versões atuais de seus respectivos projetos. As próximas seções discutem em detalhes cada questão de pesquisa, levando em consideração apenas o conjunto dos 649 métodos extraídos.

3.1 RQ1: Com qual frequência Extrair Método tem como motivação a reutilização de código?

Nesta questão de pesquisa estamos interessados em identificar os casos onde um desenvolvedor, ao alterar o sistema, extrai um método e o invoca em mais de um ponto do código. Consideramos que isso é um indício forte o suficiente de que o desenvolvedor extraiu o método com o intuito de reutilizá-lo. A Tabela 3 apresenta a quantidade de métodos extraídos que são reutilizados na mesma revisão na qual ocorreu a refatoração (coluna 3). Analisando os sistemas individualmente, observamos que na pior das hipóteses a reutilização ocorre em 41,1% dos casos (junit), enquanto na melhor das hipóteses ela ocorre em 68,6% (jsoup). No geral, observa-se que 369 dos 649 métodos extraídos já nascem com mais de uma invocação, o que corresponde a 56,9% dos casos.

Tabela 3. Extrações de método motivadas pela reutilização

Sistema	Métodos Extraídos	Reutilizados	Com Duplicação
android	100	60 (60,0%)	10 (10,0%)
android-async-http	34	15 (44,1%)	3 (8,8%)
clojure	122	72 (59,0%)	11 (9,0%)
facebook-android-sdk	144	95 (66,0%)	9 (6,3%)
jsoup	35	24 (68,6%)	1 (2,9%)
junit	107	44 (41,1%)	11 (10,3%)
picasso	41	26 (63,4%)	4 (9,8%)
spark	19	10 (52,6%)	1 (5,3%)
storm	38	18 (47,4%)	0 (0,0%)
yuicompressor	9	5 (55,6%)	1 (11,1%)
Total	649	369 (56,9%)	51 (7,9%)

Tais resultados sugerem a seguinte resposta para a RQ1: em 56,9% dos casos há indícios de que os desenvolvedores aplicam Extrair Método motivados pela oportunidade de reutilizar um trecho de código.

3.2 RQ2: Com qual frequência Extrair Método tem como motivação a eliminação de código duplicado?

Nesta questão de pesquisa estamos interessados em investigar um cenário mais específico onde, além do método ser invocado em mais de um local, tais invocações foram introduzidas para eliminar código duplicado previamente existente no sistema. A última coluna da Tabela 3 mostra a frequência com que tal

cenário ocorre no total (7,9% dos casos) e também em cada sistema. O sistema storm foi o único onde não se encontrou caso algum. É importante ressaltar que os casos de duplicação estão incluídos nos casos de reutilização identificados na RQ1. Ou seja, dos 369 métodos extraídos com mais de uma invocação, 51 foram caracterizados como remoção de duplicação.

Tais resultados sugerem a seguinte resposta para a RQ2: em 7,9% dos casos há indícios de que os desenvolvedores aplicam Extrair Método com o intuito de eliminar código duplicado existente no sistema. Embora este caso seja menos frequente, ele pode ser observado de forma consistente em quase todos os sistemas.

3.3 RQ3: Com qual frequência Extrair Método favorece a reutilização de código posteriormente, ainda que seu objetivo inicial não tenha sido esse?

Enquanto nas questões de pesquisa anteriores foram analisados apenas cenários onde, em uma revisão do sistema, um método é extraído e invocado em mais de um local, nesta questão investigamos se métodos inicialmente com uma única invocação passam a ter mais invocações em revisões posteriores do sistema.

Tabela 4. Reutilização de métodos extraídos ao longo da evolução do sistema

Sistema	Imediatamente	Posteriormente	Nunca
android	60 (60,0%)	3 (3,0%)	37 (37,0%)
android-async-http	15 (44,1%)	5 (14,7%)	14 (41,2%)
clojure	72 (59,0%)	5 (4,1%)	45 (36,9%)
facebook-android-sdk	95 (66,0%)	2 (1,4%)	47 (32,6%)
jsoup	24 (68,6%)	2 (5,7%)	9 (25,7%)
junit	44 (41,1%)	9 (8,4%)	54 (50,5%)
picasso	26 (63,4%)	2 (4,9%)	13 (31,7%)
spark	10 (52,6%)	2 (10,5%)	7 (36,8%)
storm	18 (47,4%)	1 (2,6%)	19 (50,0%)
yuicompressor	5 (55,6%)	0 (0,0%)	4 (44,4%)
Total	369 (56,9%)	31 (4,8%)	249 (38,4%)

A Tabela 4 apresenta a quantidade de métodos que se enquadram nesse cenário (coluna 3), contrastando com os métodos que já iniciam com mais de uma invocação (coluna 2) e também com aqueles que não são reutilizados em momento algum (coluna 4). O resultado geral mostra que os métodos extraídos são reutilizados posteriormente em 4,8%dos casos. Além disso, este cenário ocorre em quase todos os sistemas, com exceção do yuicompressor, mas possivelmente pela pequena quantidade de métodos extraídos identificados no mesmo.

Tais resultados sugerem a seguinte resposta para a RQ3: em 4,8% dos casos que os desenvolvedores aplicam Extrair Método não há indícios de que o intuito seja reutilizar código mas, posteriormente, o método extraído favorece a reutilização. Embora não seja frequente, este caso também é observado de forma consistente.

3.4 Cenários de Reutilização

A luz dos resultados apresentados nas seções anteriores, podemos definir três cenários nos quais a refatoração Extrair Método está associada a reutilização de código: Duplicação, Reutilização Imediata e Reutilização Posterior, que são detalhados nas seções seguintes. A Figura 2 apresenta graficamente a proporção de cada um deles nos sistemas estudados. No geral, a soma dos três cenários representa 61,6% dos casos de Extrair Método. Os 38,4% complementares são aqueles casos onde os métodos envolvidos não são reutilizados em momento algum.

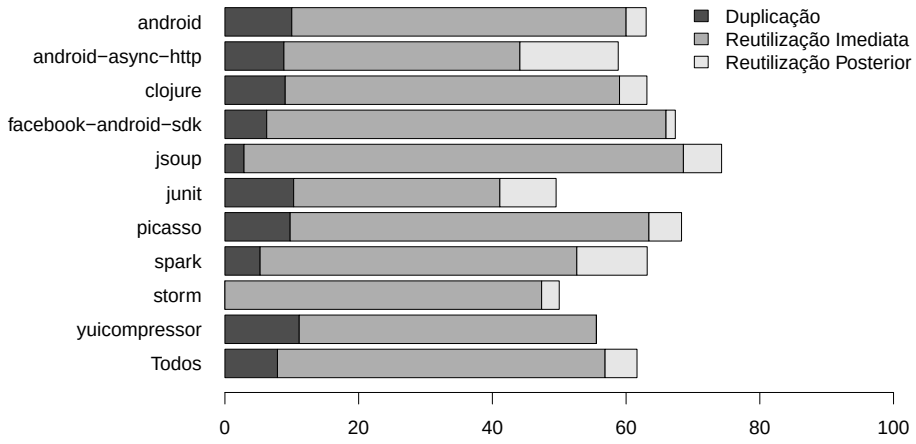

Figura 2. Distribuição entre os cenários de reutilização

Duplicação Neste cenário o desenvolvedor identifica a existência de código duplicado em dois ou mais pontos do sistema e aplica a refatoração Extrair Método, substituindo o código pela invocação do novo método, para sanar o problema. Portanto, a reutilização de código é introduzida para resolver um problema já existente no sistema (o *Bad Smell Duplicated Code*).

A Figura 3 apresenta um exemplo deste cenário no sistema junit. Neste caso, um desenvolvedor criou o método `createLoader`, contendo um trecho de código duplicado existente nos métodos `load` (linhas 8–9) e `reload` (linhas 12–13). O código duplicado foi então substituído pela invocação de `createLoader`.

Figura 3. Exemplo do cenário Duplicação

Figura 4. Exemplo do cenário Reutilização Imediata

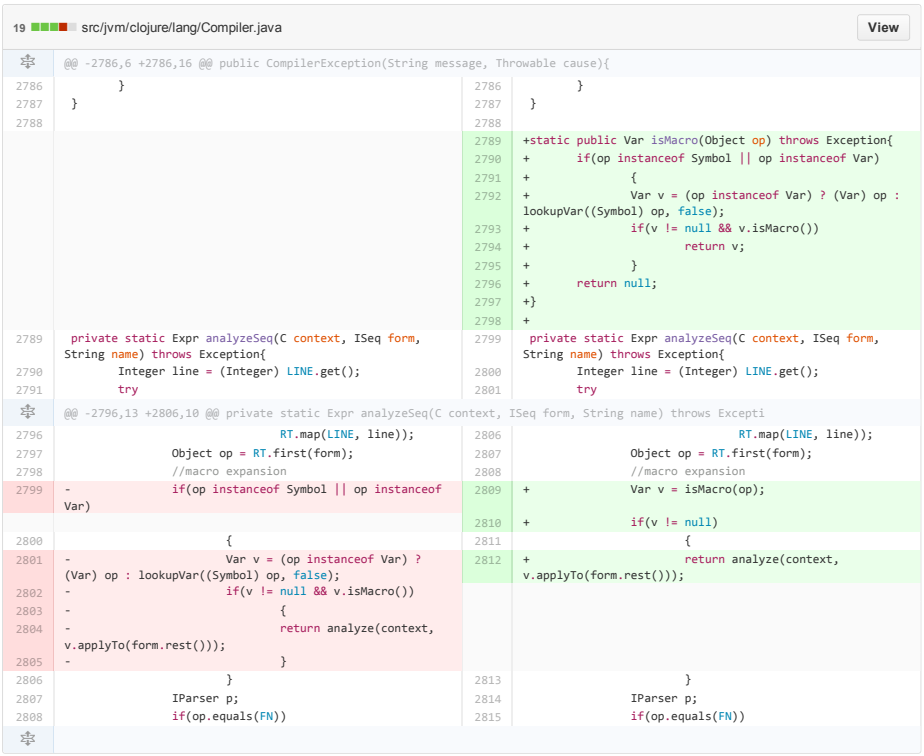

Figura 5. Exemplo do cenário Reutilização Posterior

Reutilização Imediata Neste cenário o desenvolvedor identifica a oportunidade de reutilizar código já existente ao modificar o sistema, seja para introduzir uma nova funcionalidade ou corrigir um problema. A refatoração Extrair Método é então aplicada para atender a uma necessidade imediata de reuso, sendo feita junto com a modificação.

A Figura 4 apresenta um exemplo deste cenário no sistema storm. Neste caso, o desenvolvedor criou um novo método `tryPublish`, cujo comportamento era muito semelhante ao método `publish` já existente. Para tal, um método sobrecarregado `publish(Object, boolean)` foi extraído do método original e reutilizado em `tryPublish`. Note que o parâmetro booleano introduzido no método extraído possibilita uma pequena variação na lógica, viabilizando a reutilização do mesmo em ambos os casos.

É interessante ressaltar que neste cenário o desenvolvedor prepara o sistema para receber o código que ele pretende introduzir, evitando que um problema de duplicação seja criado, enquanto no cenário anterior o desenvolvedor resolve um problema já existente de duplicação.

Reutilização Posterior Neste cenário o desenvolvedor aplica a refatoração Extrair Método, mas o método é invocado em apenas um local. No entanto, em futuras alterações no código, algum desenvolvedor se depara com a oportunidade de reutilizar esse método. Portanto, não há indícios de que a motivação inicial da refatoração seja a reutilização de código, mas ela favorece a reutilização posteriormente, possivelmente como um efeito colateral.

A Figura 5 apresenta um exemplo deste cenário no sistema clojure. Neste caso, um desenvolvedor extraiu o método `isMacro` de `analyzeSeq`, encapsulando o código das linhas 2799–2805. Podemos especular que o método provavelmente foi refatorado para melhorar sua legibilidade, visto que boa parte da lógica existente em `analyzeSeq` tinha como intenção determinar se um certo objeto era uma macro, o que ficou explicitado pelo nome do método extraído. Contudo, em uma modificação posterior, a mesma lógica foi necessária e outra invocação ao método `isMacro` foi adicionada.

4 Ameaças à Validade

Existem ao menos duas ameaças à validade interna do estudo apresentado:

- A precisão dos resultados obtidos depende do quão precisa é a ferramenta de detecção de refatorações utilizada. Embora os autores tenham reportado precisão de 96,4% [14], a precisão poderia ser diferente nas circunstâncias deste estudo. Para amenizar tal ameaça, inspecionamos manualmente uma amostragem dos resultados. De 50 instâncias da refatoração Extrair Método reportadas, escolhidas de forma aleatória, 4 foram consideradas falso positivos (92,0% de precisão). Além disso, existe a possibilidade de ocorrência de falso negativos, pois não há uma estimativa confiável da revocação da ferramenta.
- A precisão dos resultados também depende de quão precisa é a ferramenta de análise de invocações de métodos. Especificamente, acreditamos que nem toda invocação de método é identificada por dois motivos: (i) a metodologia usa apenas análise estática e não é capaz de identificar invocações por recursos dinâmicos da linguagem e (ii) a API JDT Core não é capaz de resolver vinculações de invocações de métodos em 100% dos casos, especificamente na ausência de todas as dependências de compilação. Embora o impacto dessas ameaças precise ser melhor avaliado, a existência de falso negativos não muda a conclusão principal do estudo, pois os casos de reutilização podem apenas aumentar caso essas ameaças de fato ocorram.

Também devemos considerar a ameaça de validade externa referente aos critérios de seleção dos sistemas, especialmente pelos seguintes pontos:

- Somente sistemas Java foram considerados, por limitações das ferramentas de análise. Os resultados podem ser diferentes se consideramos outras linguagens de programação.
- Os sistemas analisados são todos de uma única fonte (GitHub) e de código aberto. Os resultados podem ser diferentes para sistemas comerciais ou sistemas desenvolvidos em outras comunidades de desenvolvimento.

5 Trabalhos Relacionados

Estudos empíricos relacionados a atividade de refatoração são um tema de crescente interesse, que vem sendo abordado na literatura a alguns anos. Murphy et al. [7] investigaram dados coletados de 41 desenvolvedores usando a IDE Eclipse, para identificar quais recursos e *plug-ins* são mais utilizados. Entre outros pontos, o estudo abordou o uso das ferramentas de refatoração automatizada integradas a IDE, reportando que a operação mais utilizada pelos usuários é *Rename*, seguida de *Move* e *Extract*.

Ratzinger et al. [11] descobriram que é possível predizer a propensão a refatoração de código em um intervalo de poucos meses, utilizando o histórico de desenvolvimento do sistema. Para tal, os autores empregaram algoritmos de classificação, alimentados com informações como crescimento do código, relações entre classes, o número de autores que trabalharam no mesmo artefato, etc.

Murphy-Hill et al. [8] investigaram uma variedade de fontes de dados para entender como os desenvolvedores refatoram o código. Os autores constataram, entre outras coisas, que desenvolvedores refatoram em sessões dedicadas (*root-canal refactoring*), mas também de forma entrelaçada com outras mudanças no código (*floss-refactoring*). Além disso, a maior parte das refatorações é feita manualmente, especialmente aquelas feitas pelos desenvolvedores que não fazem parte da equipe de desenvolvimento das ferramentas de refatoração.

Bavota et al. [1] investigaram a correlação entre atividades de refatoração e a introdução de defeitos no software. Mais especificamente, os autores verificaram se classes que passaram por refatorações são mais propensas a conter erros do que classes que passaram por outros tipos de alterações. Como principal resultado, foi constatado que, embora algumas refatorações dificilmente introduzem defeitos, outras, como aquelas que manipulam a hierarquia (*pull up/push down*), são propensas a erros.

Negara et al. [9] investigaram quão frequentemente refatorações são aplicadas manualmente ou com suporte de ferramentas, utilizando uma técnica baseada na análise de alterações de código monitoradas por uma IDE instrumentada. Os autores relatam que mais da metade das refatorações é feita de forma manual. Além disso, apenas 30% delas chegam ao sistema de controle de versão.

Tsantalis et al. [14] reportam um estudo que investiga se atividades de refatoração sofrem influência de fatores como: a aplicação em código de teste e produção, a proximidade de datas de *release* e qual desenvolvedor está trabalhando no código. Além disso, diversas motivações distintas foram identificadas para refatorações dos tipos: Extrair Método, *Pull Up/Push Down Method/Field* e Extrair Superclasse/Interface.

Kim et al. [6,5] realizaram um estudo de campo na Microsoft para investigar qual a percepção dos desenvolvedores com relação aos benefícios e desafios da refatoração de código. Os autores reportaram que refatoração é vista como uma atividade que envolve custo e risco considerável. Além disso, uma análise quantitativa de dados do Windows 7 mostrou que os módulos mais refatorados tiveram uma redução em métricas de complexidade de código, mas ao mesmo tempo cresceram em tamanho.

6 Conclusão

Neste trabalhos foram analisados 10.931 commits de 10 sistemas Java de código aberto para investigar a relação entre a refatoração Extrair Método e a reutilização de código. Como principal resultado, encontramos indícios que 56,9% dos casos de aplicação de Extrair Método podem ter como motivação a reutilização do trecho de código encapsulado. Mais especificamente, isso ocorre em dois cenários distintos: **Duplicação** (7,9% dos casos) e **Reutilização Imediata** (49,0% dos casos). Além disso, verificamos a existência de um terceiro cenário, **Reutilização Posterior**, onde a refatoração favoreceu a reutilização de código no longo prazo, mesmo sem haver indícios de que essa era a intenção inicial (4,8% dos casos). Os três cenários foram encontrados em virtualmente todos os sistemas analisados e em proporções semelhantes.

Os resultados obtidos sinalizam que é incorreto assumir que a refatoração Extrair Método está sempre associada a resolução de um *Bad Smell*, tal como *Long Method* ou *Duplicated Code*. Aproximadamente metade dos casos avaliados se enquadra no cenário de **Reutilização Imediata**, onde o desenvolvedor refatora o código existente vislumbrando sua reutilização no novo código que ele está trabalhando ao modificar o sistema. Isso pode acontecer tanto na correção de um defeito quanto na implementação de uma nova funcionalidade.

Em particular, essa observação trás duas implicações para estudos relacionados a técnicas de recomendação da refatoração Extrair Método. Primeiro, independente da heurística de recomendação utilizada, certas decisões de um desenvolvedor levam em conta informações relativas ao código que ele ainda irá desenvolver, sendo bastante desafiador para uma ferramenta sugerir uma recomendação adequada. Portanto, isso deve ser levado em conta ao avaliar a precisão de uma heurística. Segundo, novas abordagens podem ser exploradas, levando em conta que a reutilização de código é um cenário frequente. Por exemplo, a extração de um trecho de código pode ser recomendada caso ele tenha alta probabilidade de ser reutilizado, com base em alguma heurística.

6.1 Trabalhos Futuros

Esse estudo pode ser ampliado nos seguintes pontos:

- Investigar com mais profundidade potenciais motivações para a refatoração Extrair Método que não se enquadram nos cenários de reutilização. Tais motivações podem estar relacionadas, por exemplo, a separação de interesses, a legibilidade do código ou a possibilidade de sobrescrita do método.
- Ampliar o estudo com mais sistemas, especialmente aqueles que já foram selecionados pelos critérios propostos (Seção 2.1).
- Avaliar com maior detalhe os possíveis casos de falso negativos na detecção de invocações de métodos.
- Avaliar a possibilidade de incluir outras refatorações como: Extrair Superclasse, Mover Método, entre outras.

Agradecimentos: Esta pesquisa foi apoiada pela FAPEMIG e CNPq.

Referências

1. Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O.: When does a refactoring induce bugs? an empirical study. In: 12th International Conference on Source Code Analysis and Manipulation (SCAM). pp. 104–113 (2012)
2. Biegel, B., Soetens, Q.D., Hornig, W., Diehl, S., Demeyer, S.: Comparison of similarity metrics for refactoring detection. In: 8th Working Conference on Mining Software Repositories (MSR). pp. 53–62 (2011)
3. Fowler, M.: Refactoring: Improving the design of existing code. Addison-Wesley (1999)
4. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining GitHub. In: 11th Working Conference on Mining Software Repositories (MSR). pp. 92–101 (2014)
5. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: 20th International Symposium on the Foundations of Software Engineering (FSE). pp. 50:1–50:11 (2012)
6. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoring challenges and benefits at Microsoft. IEEE Transactions on Software Engineering 40(7) (July 2014)
7. Murphy, G.C., Kersten, M., Findlater, L.: How are Java software developers using the Eclipse IDE? IEEE Software 23(4), 76–83 (2006)
8. Murphy-Hill, E.R., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering 38(1), 5–18 (2012)
9. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: 27th European Conference on Object-Oriented Programming (ECOOP). pp. 552–576 (2013)
10. Opdyke, W.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
11. Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H.: Mining software evolution to predict refactoring. In: 1st International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 354–363 (2007)
12. Silva, D., Terra, R., Valente, M.T.: Recommending automated extract method refactorings. In: 22nd IEEE International Conference on Program Comprehension (ICPC). pp. 146–156 (2014)
13. Tsantalis, N., Chatzigeorgiou, A.: Identification of Extract Method refactoring opportunities for the decomposition of methods. Journal of Systems and Software 84(10), 1757–1782 (2011)
14. Tsantalis, N., Guana, V., Stroulia, E., Hindle, A.: A multidimensional empirical study on refactoring activity. In: Conference of the Centre for Advanced Studies on Collaborative Research (CASCON). pp. 132–146 (2013)
15. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 54–65 (2005)