Um Estudo de Caso do Mapeamento dos Conceitos de Código Limpo para Métricas de Código-fonte

João Machini, Lucianna Almeida, Paulo Meirelles

[1]Centro de Competência em Software Livre – Universidade de São Paulo

{joaomm,lucianna,paulormm}@ime.usp.br

Resumo. *Neste trabalho elencamos as ideias e conceitos elaborados por especialistas no desenvolvimento de software orientado a objetos, buscando um maior entendimento de boas soluções, práticas e cuidados quanto ao código-fonte. Isso resultou em um mapeamento entre um conjunto de métricas de código-fonte e os conceitos de código limpo como escolhas de bons nomes, ausência de duplicações, organização e simplicidade de forma a facilitar a detecção de trechos de código que poderiam receber melhorias. Também, apresenta-se uma maneira de interpretar os valores das métricas através de cenários problemáticos e suas possíveis melhorias.*

1. Introdução

A indústria de software busca continuamente por melhorias nos métodos de desenvolvimento. Do ponto de vista prático, os processos tradicionais investem tempo especificando requisitos e planejando a arquitetura do software para que o trabalho possa ser dividido entre os programadores, cuja função é transformar o design anteriormente concebido em código. Diante de uma perspectiva dos métodos ágeis, é dado muita importância à entrega constante de valor ao cliente (*http://agilemanifesto.org*). Nesse caso, por exemplo, se há um agente financiador ou uma comunidade de software livre como cliente, o foco está na entrega de funcionalidades que possam ser rapidamente colocadas no ambiente produção e receber *feedback* constante. Em ambos os casos, não há como ignorar o fato de que o código-fonte da aplicação será desenvolvido gradativamente e diferentes programadores farão alterações e extensões continuamente. Nesse contexto, novas funcionalidades são adicionadas e falhas sanadas.

Uma situação comum durante o desenvolvimento, é um programador lidar com um trecho que ainda não teve contato para fazer alterações. Nessa tarefa, por exemplo, o programador (i) lida com métodos extensos e classes com muitas linhas de código que se comunicam com diversos objetos; (ii) encontra comentários desatualizados relacionados a uma variável com nome abreviado e sem significado, e (iii) se depara com métodos que recebem valores booleanos e possuem estruturas encadeadas complexas com muitas operações duplicadas. Depois de um longo período de leitura, o desenvolvedor encontra o trecho de lógica em que deve fazer a mudança. A alteração frequêntemente consiste na modificação de poucas linhas e o desenvolvedor considera seu trabalho como terminado, sem ter feito melhorias no código confuso com o qual teve dificuldades.

O quadro exemplificado ilustra uma circustância com aspectos a serem observados. Primeiramente, há uma diferença significativa na proporção entre linhas lidas e as inseridas. Para cada uma das poucas linhas que escreveu, o desenvolvedor teve que compreender diversas outras [Beck 2007]. Além disso, não houve melhorias na clareza ou

flexibilidade do código, fazendo com que os responsáveis pela próxima alteração, seja o próprio desenvolvedor ou o seguinte, tenham que enfrentar as mesmas dificuldades.

Tendo essa situação em vista, neste trabalho apresentamos um estilo de programação baseado no paradigma da Orientação a Objetos que busca o que denominamos de "código limpo", concebido e aplicado por renomados desenvolvedores de software como Robert C. Martin [Martin 2008] e Kent Beck [Beck 2007]. Também, propomos uma maneira de utilizar métricas de código-fonte para auxiliar no desenvolvimento de um código limpo. Beneficiando-se da automação da coleta de métricas e de uma maneira objetiva de interpretar seus valores, os desenvolvedores poderão acompanhar a limpeza de seus códigos e detectar cenários problemáticos.

2. Código Limpo

No livro *Clean Code* [Martin 2008], o autor, Robert Martin, entrevistou grandes especialistas em desenvolvimento de software como Ward Cunningham (colaborador na criação do *Fit*, do *Wiki* e da *Programação Extrema* [Beck 1999]) e Dave Thomas (fundador da OTI - *Object Technology International* - e muito envolvido no Projeto Eclipse) questionando-os quanto a uma definição para código limpo. Cada um dos entrevistados elaborou respostas diferentes, destacando características subjetivas, como elegância, facilidade de alteração e simplicidade, e outras puramente técnicas, incluindo a falta de duplicações, presença de testes de unidade e de aceitação e a minimização do número de entidades.

Em certo sentido, um código limpo está inserido em um estilo de programação que busca a proximidade a três valores: expressividade, simplicidade e flexibilidade. Tais termos também são utilizados por Kent Beck no livro *Implementation Patterns* [Beck 2007] que estão em conformidade com a unidade de pensamento que permeia as respostas dos especialistas. Um importante aspecto quanto o desenvolvimento de um código limpo é o reconhecimento de que ele não será obtido em uma primeira tentativa. Robert Martin ressalta que as versões iniciais de um método, classe e outras estruturas nunca são exatamente uma boa solução. É necessário tempo e preocupação com cada elemento desde o nome escolhido para uma variável até uma hierarquia de classes.

Diante dessa realidade, queremos garantir que não teremos empecilhos para as refatorações. Um caminho para isso é concentrar a atenção no desenvolvimento de um código limpo. Por exemplo, se as variáveis estiverem devidamente nomeadas, não precisaremos criar um comentário para explicá-las. Se os métodos estiverem bem nomeados e possuírem uma única tarefa, não será necessário documentar o que são os parâmetros e o valor de retorno. E por fim, se os testes estiverem bem executados, teremos uma documentação executável do código que garante a corretude.

Robert Martin enfatiza que o desejado é não engessar o desenvolvimento e melhoria do código com dificuldades para os programadores, mas compreender quais elementos do software precisam de documentação e contruí-la adequadamente. Dessa forma, como um dos resultados deste trabalho é o levantamento dos aspectos importantes para termos um código limpo no que diz respeito aos detalhes quanto aos métodos e classes:

– Composição de Métodos. Compor os métodos em chamadas para outros rigorosamente no mesmo nível de abstração abaixo. Isso resulta em (i) facilidade de entendimento de métodos menores; (ii) criação de métodos menores com nomes explicativos. Como

consequências temos (i) menos operações por métodos; (ii) maior número de parâmetros da classe; (iii) mais métodos na classe, mas com menos responsabilidades.

– Métodos Explicativos. Criar um método que encapsule uma operação pouco clara geralmente associada a um comentário. Isso leva ao "código cliente" do método novo ter uma operação com nome que melhor se encaixa no contexto. A consequência disso são mais métodos na classe, mas com menos responsabilidades.

– Métodos como Condicionais. Criar um método que encapsule uma expressão booleana para obter condicionais mais claros. O que resulta (i) na facilidade na leitura de condicionais no código cliente; (ii) no encapsulamento de uma expressão booleana. Mais uma vez, a consequência é mais métodos na classe, mas com menos responsabilidades cada um.

– Evitar Estruturas Encadeadas. Utilizar a composição de métodos para minimizar a quantidade de estruturas encadeadas em cada método. Isso resulta em (i) facilidade para a criação de testes; (ii) cada método terá estruturas mais simples e fáceis de serem compreendidas. Em suma, como consequências teremos menos estruturas encadeadas por método (o número total da classe continuará inalterado).

– Cláusulas Guarda. Criar um retorno logo no inicio de um método ao invés da criação de estruturas encadeadas com if sem else. Isso proporciona (i) estruturas de condicionais mais simples; (ii) o leitor do código não espera por uma contrapartida do condicional (ex: if sem else). Assim, como consequência, temos menos estruturas encadeadas na classe.

– Objeto Método. Criar uma classe que encapsule uma operação complexa simplificando a original ("cliente"). Dessa forma, (i) o "código cliente" terá um método bastante simples; (ii) a nova classe gerada poderá ser refatorada sem preocupações com alterações no "código cliente"; (iii) a nova classe poderá ter testes separados. As consequências são (i) menos operações no método cliente; (ii) menos responsabilidades da classe cliente; (iii) mais classes, com menos resposbilidades cada; (iv) acoplamento da classe cliente com a nova classe.

– Evitar Flags como Argumentos. Ao invés de criar um método que recebe uma *flag* e ter diversos comportamentos, criar um método para cada comportamento. Isso leva (i) ao leitor do código não precisará entender um método com muitos condicionais; (ii) ter testes de unidade independentes. A consequências é ter mais métodos na classe, mas com menos responsabilidades cada um.

– Objeto como Parâmetro. Localizar parâmetros que formam uma unidade e criar uma classe que os encapsule. Isso gera (i) um menor número de parâmetros, o que facilita os testes e a legibilidade; (ii) a criação de uma classe que poderá ser reutilizada em outras partes do sistema. Como consequências disso: (i) menos parâmetros passados pela classe; (ii) mais classes, com menos responsabilidades cada; (iii) mais acoplamento da classe cliente com a nova classe.

– Parâmetros como Variável de Instância. Localizar parâmetro muito utilizado pelos métodos de uma classe e transformá-lo em variável de instância. Assim, não haverá a necessidade de passar longas listas de parâmetros através de todos os métodos. Como consequências temos (i) menos parâmetros passados pela classe; (ii) menor Possível diminuição na coesão.

– Uso de Exceções. Criar um fluxo normal separado do fluxo de tratamento de erros utilizando exceções ao invés de valores de retornos e condicionais. Dessa forma, proporciona clareza do fluxo normal sem tratamento de erros através de valores de retornos e condicionais e, como consequências, menos estruturas encadeadas.

– Maximizar Coesão. Quebrar uma classe que não segue o Princípio da Responsabilidade Única. Isso significa que (i) cada classe terá uma única responsabilidade; (ii) cada classe terá seus testes independentes; (iii) sem interferências na implementação das responsabilidades. As consequências são: (i) mais classes, com menos responsabilidades; (ii) menos métodos em cada classe; (iii) menos atributos em cada classe.

– Delegação de Tarefa. Transferir um método que utiliza dados de uma classe "B" para a "B" (ou seja, mesma classe). Isso leva a (i) redução do acoplamento entre as classes; (ii) proximidade dos métodos e dados sobre os quais trabalham. Dessa forma, como consequências, temos (i) menos métodos na classe inicial; (ii) mais métodos na classe que recebe o novo método.

– Objeto Centralizador. Criar uma classe que encapsule uma operação com alta dependência entre classes. Isso proporciona (i) simplificação da classe cliente; (ii) Redução do acoplamento da classe cliente com as demais; (iii) Nova classe poderá receber testes e melhorias independentes. Consequências: menos Operações no método cliente; menos Responsabilidades da classe cliente; mais Classes; mais Acoplamento da classe cliente com a nova classe.

3. Dos conceitos de código limpo para as métricas de código-fonte

Métricas de código-fonte nos permitem criar mecanismos automatizáveis para detecção de características obtidas através da análise do código-fonte. Entretanto, diante do vasto conjunto existente de métricas, usá-las de forma isolado e compreender os significados dos valores de todas é uma tarefa custosa [Fenton and Neil 1999], o que pode levar ao pouco uso de métricas de código-fonte de maneira geral. Dada a dificuldade em determinar parâmetros numéricos para as métricas de código-fonte. Neste trabalho, propomos uma abordagem baseada em cenários para identificar trechos de código com características indesejáveis. Nesses cenários ("problemáticos"), elaboramos contextos criados a partir de poucos conceitos de código limpo no qual um pequeno conjunto de métricas é analisado e interpretado através da combinação de seus resultados. Por exemplo, conceitualmente buscamos métodos pequenos e com apenas uma tarefa. Sendo assim, estes cenários também contextualizam a possível interpretação de métricas de código-fonte, com por exemplo, relativas ao número de linhas e quantidade de estruturas de controle de forma que seus valores indiquem a presença de métodos grandes.

O mapeamento desenvolvido nesta trabalho não pretende afirmar categoricamente se um código é limpo ou não. O objetivo é facilitar melhorias de implementação através da aproximação dos valores das métricas com os esperados nos contextos de interpretação. O mapeamento proposto neste trabalho visa facilitar a procura por problemas quanto a limpeza do código. Cada cenário é descrito através dos seguintes componentes: (i. **Conceitos de limpeza relacionados**) conceitos de limpeza de código que motivaram a criação do cenário; (ii. **Características**) indicam a presença de falhas em relação as referências apresentadas; (iii. **Métricas**) mecanismos que permitem analisar o código a procura das características do cenário; (iv. **Objetivo durante a refatoração**) quais devem ser as

ideias principais durante a refatoração para eliminar o problema; (v. **Resultados esperados**) quais características devem ser encontradas no código quando terminar a refatoração para eliminação do cenário.

Quando algum cenário é detectado, sugerimos que o usuário analise o trecho indicado para verificar se é possível refatorar o código e melhorar sua limpeza. Esse mapeamento não visa apenas afirmar se existem "problemas" no código, mas sim indicar partes dele que talvez possam ser melhoradas de acordo com os conceitos que definimos como código limpo. Para facilitar a avaliação do código ao longo de seu desenvolvimento, a interpretação dos valores das métricas deve simples. Esperamos que a preocupação com a "limpeza do código" não emerja apenas quando ele estiver pronto, pois nessa fase é provável que ele tenha vários problemas e que seja muito complicado alterá-lo. Por esse motivo, montamos alguns cenários e escolhemos um conjunto pequeno de métricas que nos permite detectar as características procuradas.

Em suma, os conceitos de código limpo que apresentamos foram mapeados para o seguinte conjunto de métricas, baseado em [Marinescu 2002] e [Lanza and Marinescu 2006]: Número de Chamadas (NC); Número de Chamadas Externas (NEC); Taxa de Chamadas Externas (ECR); Número de Classes Chamadas (NCC); Número de Atributos Alcançáveis (NRA); Nível Máximo de Estruturas Encadeadas (MaxNesting); Complexidade Ciclomática (Cyclo); Número de Parâmetros (NP); Número de Repasses de Parâmetros (NRP); Número de Linhas (LOC); Número de Atributos (NOA); Número de Métodos (NOM); Soma do Número de linhas (SLOC); Média do Número de Linhas Por Método (AMLOC); Falta de Coesão Entre Métodos (LCOM4). Até o momento, como limitação deste trabalho, não conseguimos mapear todos os conceitos de código limpo usando métricas. Por exemplo, não consguimos encontrar problemas relacionados a nomes pouco expressivos usando apenas métricas. Nesse caso, não é suficiente analisarmos somente a estrutura do código: precisamos entender o contexto em que cada nome é usado, e para isso, também é necessária uma análise semântica.

3.1. Cenários para aplicação das técnicas de código limpo

– Método Grande. Neste cenário, os conceitos de código limpo tratados são: Composição de Métodos, a necessidade de Evitar Estruturas Encadeadas e a possibilidade de usar Cláusulas Guarda. Nesse contexto, métodos grandes tem como principal característica (i) um elevado número de linhas, ou seja, um alto valor de LOC. É comum também possuir (ii) um elevado número de quebras condicionais de fluxo, indicado por um valor alto da CYCLO e (iii) profundas estruturas com condicionais encadeados, indicadas por um valor alto do MaxNesting. Uma refatoração sugere diminuir o número de linhas (LOC), diminuir a complexidade ciclomática (CYCLO) e eliminar as estruturas encadeadas (MaxNesting) no método em análise. Conseguimos atingir esses objetivos decompondo o método grande em métodos menores. Normalmente, começamos essa decomposição passando os blocos de código dos desvios de fluxo para outros lugares, pois esses são conjuntos de operações evidentemente isoladas do resto do método. Com isso, esperamos que a média de linhas por método (média de LOC) e as profundidades máximas de estruturas encadeadas de cada método (MaxNesting) abaixem. Provavelmente, a soma da complexidade ciclomática da classe (CYCLO) não sofrerá alterações. Isso acontece porque normalmente espalhamos essas quebras de fluxo em outros métodos da mesma classe, mas não as eliminamos da sua lógica. Teremos uma redução da comple-

xidade ciclomática quando parte do código for eliminado ou trasferido para métodos de outras classes. Além disso, pode acontecer um aumento no número de métodos da classe, um aumento do número de métodos de outras classes ou até a criação de novas classes.

– Método com Muitos Fluxos Condicionais. Os conceitos de código limpo que constituem este cenário são a Composição de Métodos, Evitar Estruturas Encadeadas e o Uso de Exceções. Esse cenário ocorre quando temos métodos que são complexos, mas não necessariamente grandes. Suas principais características são: (i) muitas quebras condicionais de fluxo (valor alto de CYCLO) e (ii) longas estruturas com condicionais encadeados (valor alto de MaxNesting). Em métricas, queremos minimizar a complexidade ciclomática (CYCLO) e a profundidade máxima de estruturas encadeadas (MaxNesting) do método, pois queremos deixar o código mais simples e direto. Podemos nos basear na decomposição de métodos para atingirmos nossos objetivos. Depois das modificações, esperamos que a profundidade máxima de estruturas encadeadas tenha diminuído (redução do MaxNesting). Quando a refatoração for baseada no uso de exceções, como *try catch* é provável que a complexidade ciclomática do método não se altere, pois essas estruturas também são contadas como desvios de fluxo no calculo da CYCLO. Pode acontecer também um aumento no número de métodos da classe, porque, em alguns casos, o conteúdo do bloco de cada desvio condicional é deslocado para novos métodos.

– Método com Muitos Parâmetros. Este cenário tem como base a técnica do uso de Objeto com Parâmetro. A principal característica de um método que está nesse contexto é ter um elevado número de parâmetros (valor alto de NP). Objetivo é refatorar para minimizar o número de parâmetros recebidos (diminuir o NP). Os resultados esperados após a refatoração são a redução do número de parâmetros (NP) e o aumento do número de classe. Esse aumento ocorre quando criamos classes para agrupar os parâmetros que pertencem a um mesmo contexto em um único objeto.

– Muita Passagem de Parâmetros Pela Classe. Os conceitos de código limpo que constituem este cenário são o uso de Objeto como Parâmetro e a transformação de parâmetros em variáveis de instância. Podemos separar esse cenário em dois casos diferentes. O primeiro trata do repasse de muitos parâmetros. Sua característica é ter uma elevada média de parâmetros repassados (alta média de NRP). Parâmetros repassados são aqueles recebidos por um método que os repassam em chamadas a operações. O segundo caso se preocupa com a elevada média do número de parâmetros por método da classe analisada (alta média de NP). Então, é comum que os métodos de classes com muita passagem de parâmetro tenham muitos ou repassem muitos parâmetros, ou recebam variáveis que são usadas apenas em chamadas a outros métodos. Em termos de métricas, os objetivos são diminuir o número de repasses de parâmetros pela classe (diminuir o NRP) e reduzir o número de parâmetros dos métodos (reduzir o NP). Fazemos isso transformando os parâmetros muito repassados ou usados em vários métodos em atributos e criando objetos para guardar conjuntos de parâmetros relacionados. Com isso, temos a redução da média do número de parâmetros por método (redução da média de NP), redução do número de repasses de parâmetros (redução do NRP) e o aumento do número de variáveis de instância. Além disso, pode ocorrer um aumento no número de classes caso aconteça a junção de variáveis em um único objeto.

– Método "Invejoso". Método "invejoso" é aquele que usa um elevado número de métodos e atributos de poucas classes não relacionadas hierarquicamente com a sua

própria. Mapeando essas características em métricas de código-fonte, temos neste cenário alta ECR, que indica a taxa de chamadas a métodos e atributos externos, e baixa NCC, que representa o número de classes não relacionadas chamadas pelo método. A preocupação durante uma refatoração neste contexto deve ser a minimização do número de chamadas externas, calculado através do NEC. O conceito de código limpo que constitui este cenário é a Delegação de Tarefa. Após a refatoração, esperamos uma redução do número de chamadas externas (diminuição do NEC), da taxa de chamadas externas (diminuição do ECR) e da média do número de chamadas externas da classe (diminuição da média de NEC).

– Método Dispersamente Acoplado. Um método dispersamente acoplado utiliza um elevado número de atributos e métodos de várias classes não relacionadas hierarquicamente com a sua. Assim como no contexto de método "invejoso", neste cenário temos um valor alto de ECR, indicando que o uso de métodos e atributos externos é maior do que de internos. Porém, a diferença entre eles é o número de classes não relacionadas chamadas pelo método, sendo o valor do NCC alto neste cenário e baixo para métodos "invejosos". Durante uma refatoração, devemos focar na redução do número de chamadas externas (NEC). Os conceitos de código limpo que constituem este cenário são o de Objeto Centralizador e o de Objeto Método. Os resultados esperados após uma refatoração são a redução do número de chamadas externas e de sua média na classe do método analisado (NEC e média de NEC) e a diminuição do número de classes não relacionadas acessadas (NCC). Também pode ocorrer um aumento no número de classes. Isso acontecerá quando uma classe for criada para centralizar o diálogo entre as classes não relacionadas, sendo sacrificada para que as outras mantenham-se limpas e reutilizáveis em outros projetos.

– Classe Pouco Coesa. Os objetivos são a maximização da Coesão e atingir o Princípio da Responsabilidade Única. A principal característica de uma classe pouco coesa é possuir subdivisões em grupos de métodos e atributos que não se relacionam (valor de LCOM4 maior que 1), ou possuir métodos que alcançam em média poucos atributos da propria classe (média de NRA muito menor do que o valor de NOA). Nesse contexto, dizemos que um método alcança um atributo se ele usa o atributo no seu próprio corpo ou de forma indireta, ou seja, através da chamada de algum método que usa o atributo direta ou indiretamente. Os objetivos durante uma refatoração deste cenário são aumentar a coesão (diminuir a LCOM4) e diminuir a diferença entre média do número de atributos alcançáveis (NRA) e o número de atributos da classe analisada (NOA). Portanto, uma boa refatoração neste contexto seria separar as subdivisões já existentes na classe analisada em classes mais coesas. Como resultado da refatoração deste cenário, esperamos encontrar um aumento do número de classes. Além disso, desejamos que as classes existentes não tenham subdivisões (LCOM4 = 1) e que a média do número de atributos alcançáveis por cada método seja próxima da quantidade de atributos da classe (média de NRA próxima de NOA).

4. Estudo de Caso

Utilizando as técnicas e conceitos relacionados a um código limpo e seu mapeamento em métricas de código fonte, analisamos a ferramenta Analizo [Terceiro et al. 2010], escrita em Perl e com um arquitetura orientada a objetos. Olhamos para suas métricas e as interpretando para detectar e fazer as melhorias possíveis (refatorações).

4.1. Estado Inicial

Inicialmente, a característica central da classe *Metrics*, a principal da Analizo, era a complexidade de seus métodos. Eram 25 métodos que acumulavam uma alta média de LOC (aproximadamente 10 linhas de código por método), além de uma alta média de CYCLO (3,5 por método), o que mostra que os métodos continham muitos laços condicionais.

Apesar de acumular muitas responsabilidades e ter baixa coesão, seu LCOM4 tinha um valor baixo causado pelo método *list_of_metrics* que não utilizava variáveis de instância e não chamava outros métodos dentro da classe. O valor de LCOM4 igual a 2, só não era muito maior devido à presença do método *report_module*, responsável pela chamada de todos os métodos relativos ao cálculo de uma métrica de módulo. Dois cenários poderiam ser observados com clareza:

–Método Grande. O método *report* continha muitas tarefas: (i) coletar as métricas de cada um dos módulos através de diversas chamadas para o método *report_module*; (ii) combinar os valores em diversas estruturas de dados; (iii) chamar os métodos para cálculo de métricas globais; (iv) calcular os valores estatísticos e (v) montar a com todas as informações coletadas. Seu corpo continha um grande número de linhas (LOC = 73) e complexidade ciclomática (CYCLO = 11). É interessante notar que possuia um número baixo de estruturas encadeadas (MAXNESTING = 1), mostrando que seu problema central não era a complexidade de suas tarefas, mas sua quantidade. Esse método é um exemplo claro de método em que os detalhes de implementação ficam expostos, o que faz com que o leitor tenha que assimilar muitas informações para fazer uma única alteração.

– Métodos com Muitos Fluxos Condicionais. Métodos para cálculo da métricas como ACC (*Afferent Conexions per Class*) (mostrado abaixo no trecho de código 4.1), LCOM4 (*Lack of Cohesion of Methods*) e CBO (*Coupling Between Objects*) possuiam um código relativamente pequeno, apesar de acumularem muitas estruturas encadeadas. Esse fato pode ser constatado ao observamos que LOC médio dos três exemplos se aproximava de 8, enquanto que o MAXNESTING chegava a 5 e complexidade ciclomática 6, o que indica que praticamente todas as suas linhas tinham algum condicional. O resultado desse cenário eram métodos bastante difíceis de serem compreendidos. Um esforço significativo teria que ser feito pelo leitor para que pudesse afirmar com segurança como era realizado o cálculo da métrica representado por um desses métodos. Um fato interessante de ser comentado é que os métodos da classe *Metrics* recebiam poucos parâmetros. Os métodos responsáveis pelo cálculo de métricas de módulo só recebiam o módulo como parâmetro, enquanto que os demais praticamente não recebiam.

```
sub acc {
  my ($self, $module) = @_;

  my @seen_modules = ();
  for my $caller_member (keys(%{$self->model->calls})){
    my $caller_module = $self->model->members->{$caller_member};
    for my $called_member (keys(%{$self->model->calls->{
        $caller_member}})) {
      my $called_module = $self->model->members->{$called_member};
      if($caller_module ne $called_module && $called_module eq
          $module){
        if(! grep { $_ eq $caller_module } @seen_modules){
```

```
            push @seen_modules, $caller_module;
          }
        }
      }
    }
    return scalar @seen_modules + $self->_recursive_noc($module);
}
```

– O que foi feito? Diante deste contexto, o primeiro passo foi utilizar a *Composição de Métodos* para diminuir o tamanho dos métodos e *Evitar Estruturas Encadeadas*.

4.2. Segundo Estado

A tentativa de melhoria do código dos métodos da classe *Metrics* trouxeram diversas alterações e resultados que melhoraram sua expressividade. Os métodos que inicialmente tinham muitas linhas de código foram reduzidos de forma que a média de LOC da classe abaixasse de 10 para 5,62, ou seja, uma redução de mais de 40%. Além disso, o grande número de condicionais foi significamente reduzido, passando de uma CYCLO média de 3,5 para um valor próximo de 1,51, ou seja 57% menos que o inicial.

Outra contribuição importante foi a queda nos valores de MAXNESTING dos métodos que antes se encaixavam no cenário "Métodos com Muitos Fluxos Condionais". A média de MAXNESTING chegou a aproximadamente 0,53, também um redução de quase 50%.. Essas alterações se deram devido ao grande aumento no número de métodos da classe: os 25 métodos foram decompostos em um total de 53 novos. Considerando os valores das métricas, podemos afirmar que os métodos do segundo estado do estudo de caso ficaram pequenos e com aproximadamente apenas um fluxo condicional.

Pensando nas consequências para a limpeza do código, todos os métodos passaram a ser compostos basicamente por chamadas para outros métodos com nomes explicativos o suficiente para que a leitura de sua implementação se tornasse facilmente compreendida. O exempo abaixo apresenta o código do método *acc* que ilustra bem essa interpretação.

```
sub afferent_connections_per_class {
  my ($self, $module) = @_;

  my $number_of_caller_modules = $self->
      _number_of_modules_that_call_module($module);
  my $number_of_modules_on_inheritance_tree = $self->
      _recursive_number_of_children($module);

  return $number_of_caller_modules +
      $number_of_modules_on_inheritance_tree;
}
```

Com um código com essa simplicidade e expressividade, o leitor pode facilmente afirmar que para calcular a quantidade de conexões aferentes de um dado módulo, basta somar o número de módulos que o chamam e o número de módulos em sua árvore de herança. Além disso, a decomposição dos métodos também trouxe melhorias quanto à flexibilidade do código. A medida que os métodos se tornam menores, mais fácil fica a detecção de trechos semelhantes. O encapsulamento de operações nessas circustâncias torna possível o reuso, além de deixar os métodos clientes mais simétricos e sem muitos

detalhes de implementação. Em contraposição, dois cenários relacionados foram observados:

```
sub _module_parent_of_other {
  my ($self, $module, $other_module) = @_;
  return grep {$_ eq $module} $self->model->inheritance(
      $other_module);
}
```

– Métodos com Muitos Parâmetros. Com o grande aumento no número de métodos e as sucessivas chamadas que fazem entre si, podemos notar um significativo aumento no número de parâmetros de alguns métodos. O método *add_edges_to_used_functions_and_variables* é um exemplo que acumulou 4 parâmetros. É importante destacar que para implementar a Orientação a Objetos em Perl é sempre necessário passar uma referência para o "self". Isso faz como que o número de parâmetros sempre seja uma unidade maior quando comparado com as demais linguagens. Para o cálculo das métricas esse parâmetro nunca foi contabilizado.

```
sub _add_edges_to_used_functions_and_variables {
  my ($self, $graph, $function, @functions, @variables) = @_;

  for my $used (keys(%{$self->model->calls->{$function}})) {
    if (_used_inside_the_module($used, @functions, @variables)) {
      $graph->add_edge($function, $used);
    }
  }
}
```

– Muitos Repasses de Parâmetros pela Classe. Associado ao cenário acima, podemos observar que o repasse de parâmetros se mostrou ao longo de toda a classe *Metrics*, o que deixa o corpo dos métodos poluído com variáveis que não utilizam e somente repassam. O valor médio de parâmetros passou de 0,64 para 1,38 e o número total de repasses foi aumentou de apenas 3 para 21.

```
sub _collect_global_metrics_report {
  my ($self, $module_metrics_totals, $module_counts, $values_lists)
      = @_;
  my $summary = $self->_initialize_summary();
  $self->_add_module_metrics_totals($summary, $module_metrics_totals
      );
  $self->_add_module_counts($summary, $module_counts);
  $self->_add_statistical_values($summary, $values_lists);
  $self->_add_total_coupling_factor($summary, $module_counts);
  return $summary;
}
```

– O que foi feito? O foco das próximas mudanças foi o *Uso de Parâmetros como Variável de Instância* para diminuir o número de parâmetros sendo repassado.

4.3. Terceiro Estado

Como discutido na apresentação da técnica do Uso de Parâmetros como Variável de Instância, não podemos transformar qualquer parâmetro em um elemento do estado do objeto. Sendo assim, nas alterações dessa etapa não foram feitas mudanças quanto aos

parâmetros dos métodos responsáveis pelo cálculo de métricas de módulo, uma vez que não faziam sentido para toda a classe.

No entanto, mesmo através de alterações em somente uma porção dos métodos, as mudanças nos valores das métricas ficaram visíveis. A média de parâmetros passou de 1,38 para 0,83, valor mais próximo ao inicial (0,65). No código 4.3 abaixo, podemos notar a diferença na limpeza de um método beneficiado pela melhoria mencionada e cuja versão anterior é apresentada no trecho de código 4.2. Novamente, as alterações tiveram consequências para a classe e um novo cenário pode ser observado:

```
sub _collect_global_metrics_report {
  my $self = shift;
  $self->_add_module_metrics_totals();
  $self->_add_module_counts();
  $self->_add_statistical_values();
  $self->_add_total_coupling_factor();
}
```

– Classe Pouco Coesa. Ao longo da decomposição dos métodos e a promoção de parâmetros a variáveis de instância, gradativamente podemos notar com mais clareza as responsabilidades da classe no nível da implementação. Em outras palavras, na medida que em alteramos o código, as responsabilidades dentre uma classe começam a se confirmar pela maneira em que as tarefas são implementadas. Desde o início deste estudo de caso, mencionamos que a classe *Metrics* tinha, por exemplo, a responsabilidade de calcular valores de métricas de módulo e valores estatísticos dos mesmos. Nesse estado, podemos notar como essas responsabilidades são implementadas e o limiar entre elas, sobre o qual trabalharemos para quebrar a classe em duas. Essa observação conceitual se mostra através das métricas. A média de atributos atingíveis pelos métodos passou a ser 1,1, valor significativamente menor do que o número de atributos da classe (NOA = 6). Essa diferença mostra que existem conjuntos de métodos somente interessados em alguns dos atributos. Observando o código, podemos notar que os métodos calculadores de métricas de módulo só utilizam o objeto *Model*, enquanto que os métodos relativos a coleta dos valores globais utilizam todos os outros atributos.

– O que foi feito? Nessa etapa, muitas alterações foram feitas. A intenção foi quebrar a classe *Metrics* em várias classes com uma responsabilidade e, para cada uma delas, aplicar as mudanças apresentadas nos outros estados.

4.4. Estado Final

Na transição para o estado final, a principal alteração foi a quebra da classe *Metrics* em muitas outras. O primeiro passo foi a criação de uma classe para uma das métricas de módulo. Essas classes recebem o objeto *Model* como parâmetro de seu construtor e possuem um método *calculate* que devolve o calor da métrica de um dado módulo.

Uma vez com essas classes isoladas, ficou mais simples detectar os parâmetros que poderiam ser promovidos a variável de instância. Por exemplo, a nova classe *LackOfCohesionOfMethods*, responsável pelo cálculo de LCOM4, passou a ter o atribute *graph*, uma vez que o objeto do classe *Graph* faz parte de seu estado e ciclo de vida.

O passo seguinte consistiu na identicação das demais responsabilidades da classe *Metrics* e a criação de uma classe para cada uma delas. A classe *ModuleMetrics* ficou responsável por instanciar e invocar os objetos citados acima, enquanto que *GlobalMetrics*

encapsulou a coleta das métricas globais e valores estatísticos. Essas alterações fizeram com que a classe *Metrics* inicial ficasse apenas com a responsabilidade de combinar todos os valores calculados pelas demais classes.

Ao final de todas as alterações, podemos notar que, de maneira geral, o código se aproximou consideravelmente de um código expressivo, simples e flexível. O tamanho dos métodos em termos do número de linhas e complexidade foram grandes contribuições para essa realidade. A média de LOC passou de aproximadamente 10 para 5 e a média de CYCLO e MAXNESTING passaram, respectivamente, de 3 para 1 e 1,2 para 0,3. Também, a média de número de parâmetros estava em torno de 0,65. Na medida em que os métodos foram decompostos e alguns parâmetros se tornaram atributos, o valor médio gradativamente se aproximou do valor 0,3. Essa redução ocorreu principalmente devido a criação de classes menores e mais coesas com métodos que trabalham continuamente com os seus atributos.

5. Consideração finais

Apresentamos um conjunto de conceitos relacionados a um código limpo e seu mapeamento em métricas de código-fonte com o objetivo de prover aos desenvolvedores uma maneira de pensarem em melhorias para os seus códigos. Em suma, compilamos e abordamos de forma prática os conceitos sobre código limpo do ponto de vista de renomados especialistas em desenvolvedores de software.

Do ponto de vista do mapamento dos conceitos em métricas de código-fonte, o uso de cenários é capaz de detectar trechos de código que poderiam receber melhorias. O que podemos concluir ao final desse estudos é que, nos exemplos de código, estudo de caso e conceitos da bibliografia sobre os quais trabalhamos, para cada método buscamos minimizar os valores das métricas LOC, NP, NRP, CYCLO e MaxNesting, além de criar classes que minimizem LCOM4 e a diferença entre a média de NRA e NOA. As alterações nos valores dessas métricas, ou seja, o controle e monitoramento delas, apontaram para um "código mais limpo", de acordo com os conceitos apresentados.

Por fim, mais detalhes dos conceitos de código limpo, bem como o relato completo do estudo de caso, pode ser obtido em um relatório técnido do IME-USP (*http://ccsl.ime.usp.br/codigolimpo*).

Referências

Beck, K. (1999). *Extreme Programming Explained*. Addison Wesley.

Beck, K. (2007). *Implementation Pattens*. Addison Wesley.

Fenton, N. E. and Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149 – 157.

Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Systems*. Springer.

Marinescu, R. (2002). Measurement and quality in object-oriented design.

Martin, R. C. (2008). *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L. R., Almeida, L., Chavez, C., and Kon, F. (2010). Analizo: an extensible multi-language source code analysis and visualization toolkit.