Código Limpo e seu Mapeamento para Métricas de Código Fonte

Lucianna Thomaz Almeida

João Machini de Miranda

Orientador: Prof. Dr. Fabio Kon
Coorientador: Paulo Roberto Miranda Meirelles

São Paulo, Dezembro, 2010

Resumo

Ao produzir software, os desenvolvedores não podem ficar satisfeitos com um código que simplesmente faz o trabalho que deve ser feito. É preciso considerar que será necessário manter a aplicação, fazer mudanças à medida em que os requisitos se alterarem e que outros terão que usar e aprimorar o código. Diante disso, este trabalho aborda 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. Segundo Robert Martin, um programador deveria sempre fazer com que seu código seja uma composição de instruções e abstrações que possam ser facilmente entendidas, uma vez que gastamos a maior parte do tempo lendo-o para incluir funcionalidades e corrigir falhas. Os seguintes livros são base para os nossos estudos: *Clean Code* de Robert Martin e *Implementation Patterns* de Kent Beck. Ambos possuem diversos aspectos que ajudam a identificar o que é um código limpo como escolhas de bons nomes, ausência de duplicações, organização e simplicidade. Esta monografia contém uma reunião desses conceitos, explicitando sua relevância e aplicação. Além disso, essa monografia apresenta um mapeamento entre um conjunto de métricas de código-fonte e os conceitos acima citados, de forma a facilitar a detecção de trechos de código que poderiam receber melhorias. Para que as métricas possam ser mais facilmente incorporadas no cotidiano dos programadores, apresentamos uma maneira de interpretar os valores das métricas através de cenários problemáticos e suas possíveis melhorias. Por fim, um estudo de caso sobre a ferramenta Analizo permitiu uma validação dos conceitos apresentados ao longo do trabalho, além de contribuir diretamente com melhorias para o código da ferramenta.

Sumário

Capítulo 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 (AgM, 2001). Nesse caso, 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). Um Código Limpo é fundamentado em testes e decisões técnicas que visam a clareza, flexibilidade e simplicidade do código-fonte, o que pode ter um impacto importante no desenvolvimento de software, aumentando a produtividade e diminuindo os custos de manutenção. Por fim, ao longo desta monografia também apresentaremos 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.

1.1 Objetivos

O primeiro objetivo deste trabalho é apresentar um levantamento teórico de conceitos relacionados ao que denominamos código limpo, buscando expor um conjunto de técnicas e boas decisões que possam ser adotadas ao longo do desenvolvimento para auxiliar a criação de um código mais expressivo, simples e flexível. Essa apresentação permite um contato com recomendações e preocupações de desenvolvedores de software reconhecidos internacionalmente ao trabalharem com o paradigma da Orientação a Objetos.

Além disso, também temos como objetivo o mapeamento entre um grupo de métricas de código-fonte e os conceitos acima citados, de forma a facilitar a detecção de trechos de código que poderiam receber melhorias. Para que as métricas possam ser mais facilmente incorporadas no cotidiano dos programadores, também temos como objetivo a apresentação de uma maneira de interpretar os valores das métricas através de cenários problemáticos que ocorrem frequentemente durante o desenvolvimento. Desse modo, queremos apresentar uma maneira de utilizar o poder da automatização das métricas associado a interpretação dos valores calculados, facilitando assim a aproximação dos conceitos relacionados ao código limpo.

1.2 Desenvolvimento do Trabalho

Para definir um código limpo e selecionar o conjunto de conceitos e técnicas desta monografia, trabalhamos sobre duas principais referências: *Implementation Patterns* (Beck, 2007) de Kent Beck e *Clean Code* (Martin, 2008) de Robert C. Martin. Ambos os livros foram escolhidos pelo grande reconhecimento dado a esses autores pela comunidade de desenvolvimento de software. Baseado em suas experiência, Beck e Martin expõem uma série de recomendações para a criação de um bom código orientado a objetos.

Ao longo desse trabalho, escolhemos uma parcela dos conceitos comuns em ambos os livros. Diversos desses aspectos são bastante conhecidos e apresentamos aqui exemplos práticos que explicitam como as técnicas alteram a clareza e legibilidade do código. Posteriormente, trabalhamos sobre uma implementação do algoritmo de Dijkstra para encontrar a árvore geradora de custo mínimos, buscando encontrar melhorias que ilustrassem a aplicação dos tópicos estudados.

Para elaborar o mapeamento das métricas de código-fonte, tivemos como ponto de partida as 22 métricas da ferramenta Analizo (Terceiro *et al.*, 2010), a qual colaboramos para o desenvolvimento por mais de um ano. Além disso, baseamos nossos estudos no livro *Object Oriented Metrics In Practice* (Lanza e Marinescu, 2006) de Michele Lanza e Radu Marinescu, que apresenta uma maneira bastante objetiva de avaliar o design de aplicações orientadas a objetos.

A união da nossa experiência no desenvolvimento da Analizo e o estudo das referências tornou possível a seleção das métricas apresentadas nessa monografia. No entanto, desde o desenvolvimento e estudo sobre a Analizo, nosso grupo de pesquisa enfrentou dificuldades com a compreensão dos valores das métricas, o que nos motivou a focar no seu entendimento. Para tal, trabalhamos sobre pequenos

exemplos de código utilizando o seguinte ciclo: (i) cálculo das métricas; (ii) análise do código para a encontrar uma melhoria que pudesse aproximá-lo de um código limpo e observação dos valores das métricas para encontrar relações com a melhoria; (iii) implementação da melhoria; (iv) recálculo das métricas e análise das alterações.

Uma vez definida a seleção de métricas e com bom entendimento quanto a um código limpo, trabalhamos sobre o código da própria Analizo para implementar melhorias e tornar seu código mais limpo. Dessa forma, validamos os conceitos utilizados ao longo da monografia para proporcionarmos mais uma contribuição a essa ferramenta.

1.3 Organização do Trabalho

Além dessa introdução, este texto está organizado em quatro capítulos. No Capítulo 2, apresentamos o que é um código limpo através da união das concepções de importantes especialistas no desenvolvimento de software. O foco será nos livros *Clean Code* de Robert Martin (Martin, 2008) e *Implementation Patterns* de Kent Beck (Beck, 2007) e serão apresentados tantos conceitos teóricos quanto técnicas conhecidas para a criação de um código expressivo, simples e flexível.

O capítulo seguinte, Capítulo 3, apresenta os conceito das métricas de código-fonte e discute como podemos utilizá-las para extrair informações sobre as características do código. Além disso, definimos um mapeamento objetivo entre o conceito de código limpo e as métricas de código-fonte, com o objetivo de auxiliar os programadores na utilização desses conceitos.

No Capítulo 4, apresentamos o estudo de caso feito sobre a ferramenta Analizo para validar o mapeamento definido e aplicarmos os conceitos discutidos ao longo do trabalho através da implementação de melhorias que pudessem tornar seu código limpo.

Por fim, as conclusões e possibilidades de trabalhos futuros serão expostos no Capítulo 5.

Capítulo 2

Código Limpo

A definição de um bom código não é precisa. Do mesmo modo que enfretamos dificuldades para definir o que é arte, não podemos definir um conjunto de parâmetros lógicos e mensuráveis para delimitar a diferença de qualidade entre códigos-fonte. Podemos considerar aspectos como testabilidade, eficiência, facilidade de modificação, processo pelo o qual foi desenvolvido, entre outros. Diante dessa dificuldade, nos deteremos a trabalhar com o chamado código limpo orientado a objetos no âmbito deste trabalho.

No livro *Clean Code* (Martin, 2008), o autor 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 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.

Expressividade

Dentre as definições encontradas pela pesquisa de Robert Martin acima citada, algumas afirmações se destacam quanto a expressividade. Na visão de Grady Booch (Booch, 2007), um código limpo pode ser lido como um texto em prosa e deixa claro as intenções do autor através de operações e abstrações bem escolhidas. Já para Dave Thomas e Kent Beck, a medida para a expressividade está na facilidade para um desenvolvedor, que não o autor original do trecho de código, entender, modificar e utilizá-lo.

Grande parte dessas ideias receberam influência do livro de Donald Knuth, *Literate Programming* (Knuth, 1992). A intenção era programar em linguagens naturais, como o inglês, e intercalar as expressões com macros e pequenos trechos de código-fonte tradicional. O arquivo poderia ser lido como um texto comum, mas no plano de fundo seriam executados os trechos correspondentes a cada uma das expressões em linguagem natural.

Nesse sentido, um código é expressivo quando pode ser facilmente lido nas diferentes camadas de abstração, deixando detalhes de implementação "escondidos" em níveis mais baixos. Nas palavras de

Ward Cunningham, a solução implementada faz com que a linguagem pareça ter sido feita para o problema sendo resolvido.

Simplicidade

A simplicidade é um dos aspectos mais valorizados por Kent Beck. Na sua visão, devemos programar buscando reduzir a quantidade de informação que o leitor deve compreender para fazer alterações. Dessa maneira, o autor dá grande importância para a minimização do número de abstrações, comentários e estruturas complexas entre as classes que são desnecessários.

Flexibilidade

A flexibilidade reflete a facilidade de estender a aplicação sem fazer grandes alterações na estrutura já implementada. Não queremos ter que mudar diversos métodos e diversas classes para adicionar uma funcionalidade.

O código pretendido deve se adequar e aproveitar bem das vantagens inerentes ao paradigma da orientação a objetos. Tal adequação está bem representada tanto nos princípios *SOLID* (Martin, 2000), quanto nos princípios apresentados em *Implementation Patterns* abaixo.

- Consequências Locais: devemos estruturar o código de forma que as alterações fiquem encapsuladas em partes do programa;
- Minimizar Duplicações: devemos particionar o programa em partes pequenas com o intuito de minimizar as repetições tanto na forma de trechos semelhantes como em hierarquias de classe parelelas e estruturas com informações duplicadas;
- Proximidade da Lógica e Dados: devemos aproximar a lógica e os dados sobre quais trabalha uma vez que ambos serão alterados simultaneamente;
- Simetria: devemos criar abstrações de maneira simétrica para que o leitor, uma vez entendida uma das partes, entenda rapidamente as demais.

2.1 O Desenvolvimento de um Código Limpo

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. O mais crítico dos impedimentos é a insegurança. Se ao fazer uma alteração, ficamos receosos de que algo pode ter quebrado, provavelmente logo deixaremos de fazê-las, progressivamente afastando o código da limpeza desejada. Dessa forma, a presença de testes automatizados são a fundação para o desenvolvimento de um código limpo. Os testes dão a segurança de que as partes estão funcionando corretamente, possibilitando as diversas refatoracões para deixar o código claro.

Além de auxiliar nas alterações, os testes compõem parte da expressividade, simplicidade e flexibilidade de um código. Para que desenvolvedores que não o autor de um trecho de código o compreendam, os testes podem reportar como o elemento deve ser usado. Por fim, a medida para a simplicidade pode ser feita através da suíte de teste: se os testes estão todos corretos, não é necessário adicionar complexidade, apenas melhorias sem adicionar funcionalidade.

Outro tipo de empecilho que podemos encontrar são os comentários e documentações para métodos e classes. Para qualquer alteração feita no código, em tese os comentários e formas de documentação também deveriam ser mudados para ficarem de acordo com o estado atual. Uma contraposição é muito frequente, como aponta Robert Martin: por um lado, se nos policiarmos para que a atualização do código e comentários seja simultânea, muito provavelmente deixaremos de fazer alterações devido ao excesso de trabalho; por outro, se deixarmos de atualizar uma das formas de documentação, estaremos tornando-as dispensáveis, além de potencialmente se tornarem fontes de informações incorretas.

A solução dada é concentrar a atenção no desenvolvimento de um código limpo. 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.

As seções subsequentes apresentam aspectos importantes para termos um código limpo no que diz respeito a detalhes quanto aos nomes, métodos e classes.

2.2 Nomes

Em uma primeira análise sobre um código-fonte, uma das afirmações que pode ser feita é que ele é constituído por um conjunto de palavras reservadas da linguagem e um número imenso de nomes escolhidos pelos desenvolvedores. Por exemplo, um código escrito em Smalltalk praticamente não contém palavras da sintaxe da linguagem, contendo uma grande quantidade de envio de mensagens aos objetos e usos de variáveis, ambos com nomes escolhidos pelos desenvolvedores.

Nomes que Revelam Intenção

De acordo com Robert Martin, o nome de uma variável, método ou classe deveria responder a todas as questões a cerca do elemento sendo nomeado. Deveria contar porque o elemento existe, o que faz e como deve ser usado, de forma que comentários se tornem redundantes. O exemplo abaixo ilustra a dificuldade do leitor em compreender um código com nomes pouco reveladores.

```
def caminhoR(Vertice v):
    Vertice w
    est[v] = 0
    for(w = 0; w < tamanho(); w++)
      if(adj(v,w))
        if(est[w] == -1):
```

```
        imprime(w)
        caminhoR(w)
```

Listing 2.1: *Exemplo de nomes pouco reveladores*

Esse método, pertencente a uma classe Grafo, é difícil de ser compreendido sem um vasto conhecimento da convenção de notações e o algoritmo utilizado. Muitas perguntas poderiam ser feitas como (i) O que faz caminhoR?, (ii) O que é o vértice *v*? E quanto ao *w*? Eles tem algo em comum?, (iii) O que significa *adj(v, w)* ser igual a *true*?.

A seguir segue o mesmo código com alterações somente nos nomes das variáveis e métodos, sem nenhuma alteração na implementação.

```
def imprimeVerticesDoPasseioComOrigemEm(Vertice origem):
  Vertice proximo;
  estado[origem] = JA_VISITADO
  for(proximo = 0; proximo < numeroDeVertices(); proximo++)
    if(saoAdjacentes?(origem, proximo))
      if(estado[proximo] == NAO_VISITADO):
        imprime(proximo)
        imprimeVerticesDoPasseioComOrigemEm(proximo)
```

Listing 2.2: *Melhorias nos nomes podem tornar o código mais claro*

Muitas questões podem ser respondidas depois da leitura desse trecho. O nome da função deixa claro que sua tarefa é passear pelos vértices do grafo começando pelo vértice origem passado como parâmetro. O vetor *est* foi renomeado para estado e as constante 0 e -1 receberam um nome, revelando o significado das operações em que constam. A simples mudança de *tamanho()* para *numeroDeVertices()* provê um contexto mais completo para o entendimento do laço.

O exemplo poderia ser melhorado para ficar mais alinhado com o paradigma da orientação a objetos e com os conceitos que serão apresentados mais adiante. Nesta seção, tratamos a importância da escolha de nomes e como pode auxiliar o entendimento. É possível revelar o significado de elementos dando nomes para as constantes, por exemplo. Nomes como origem e próximo podem nomear objetos da mesma classe para mostrar o contexto e semântica de como estão sendo usados. Métodos podem trazer mais expressividade e fluência à leitura se forem nomeados levando em conta o código cliente e nas circunstâncias em que será chamado.

Diferenças Significativas

Durante a criação de um método ou uma classe, é muito comum que, no mesmo contexto, haja objetos de tipos iguais. Nessa situação, não basta escolher nomes que representem bem o que é o objeto, mas também é preciso criar uma diferença significativa entre eles. O intuito é deixar claro ao leitor do código (ou seja, o programador) o papel de cada uma das partes, tanto para facilitar o uso de métodos deixando claro a ordem dos parâmetros, quanto para que a semântica de uma operação fique mais clara.

Quando um leitor se depara com um método como *copiaElementos(lista1, lista2)*, pode não ficar claro qual é a ordem dos argumentos: os elementos da lista1 são copiados para lista2 ou vice-versa? Com uma simples distinção criada através dos nomes, a leitura não levanta nenhum tipo de dúvidas em *copiaElementos(origem, destino)*.

Nomes Temporários

Uma solução utilizada para melhorar a clareza das operações de um trecho de código é utilizar nomes temporários para revelar como um objeto será utilizado naquele contexto. O exemplo abaixo ilustra bem esse tipo de situação.

```
encontraCaminhoEntre(estacoes.encontra("Jabaquara"), estacoes.linhaAmarela().
    ultima())
```

Listing 2.3: *Exemplo sem o uso de nomes temporários*

Com algum esforço, o leitor identificará que o código está buscando o caminho entre a estação Jabaquara e a última estação da linha Amarela. Porém, a expressividade da versão alterada abaixo faz com que a leitura e compreensão sejam imediatas.

```
estacaoJabaquara = estacoes.encontra("Jabaquara")
ultimaEstacaoLinhaAmarela = estacoes.linhaAmarela().ultima()
encontraCaminhoEntre(estacaoJabaquara, ultimaEstacaoLinhaAmarela)
```

Listing 2.4: *Usando nomes temporários*

Nome Único por Conceito

Outra recomendação dada por Robert Martin (Martin, 2008) se refere a unicidade de nomes por conceito. O objetivo do estilo de programação buscado é facilitar a compreensão do código. Então, nomear um conceito de maneiras diferentes é bastante confuso, uma vez que o leitor não saberá se o contexto trata de ideias distintas. Qual seria a diferença entre *get*, *fetch*, *retrieve*? Ou encontrar, procurar e buscar?

Se apenas uma nomenclatura for utilizada, o leitor não terá questões, além de poder se aproveitar da simetria de operações valorizada por Kent Beck (Beck, 2007). Não é um acaso que os métodos sobre coleções sempre possuem um mesmo conjunto de nomes como o método *add* para adicionar elementos. Desta forma, as dúvidas de qual deles utilizar para cada uma das opções é minimizada, sendo mais fácil a assimilação do conceito.

Nomes dos Métodos

Na próxima seção serão apresentados fundamentos teóricos quanto ao uso de métodos para a documentação do código através de sua expressividade. Antes, para que as técnicas exemplificadas sejam eficazes como pretendido por Robert Martin e Kent Beck, os nomes dos métodos devem ser bem escolhidos de modo que descrevem muito bem a tarefa que realizam.

O uso de nomes descritivos pode muitas vezes resultar em nomes longos e difíceis de serem digitados muitas vezes. Entretanto, a quantidade de ocasiões em que serão lidos é substancialmente maior do que a quantidade em que as escrevemos. Desta maneira, a economia de palavras deve ser descartada em favor de uma boa expressividade (Beck, 2007).

A linguagem Smalltalk é constantemente elogiada quanto a sua expressividade. Grande parte dos elogios estão concentrados no uso de seletores para o envio de mensagens para os objetos. Podemos gerar métodos como *criaJogoDeFutebolEntre:* SaoPaulo *e:* Palmeiras *naData:* diaDoJogo. Os elementos em itálico são o verdadeiro protótipo do método, deixando claro a ordem dos argumentos, seu

papel e o que exatamente está sendo executado. Essas são as características desejadas para os nomes dos métodos formarem em conjunto uma narrativa da execução.

2.3 Métodos

Os métodos são o núcleo fundamental para a criação de um código expressivo. Considerando as ideias levantadas por Grady Booch e Kent Beck, o objetivo é desenvolver um código que seja lido como um texto em prosa e, nessa analogia, a intenção é utilizar os nomes dos métodos para narrar a execução.

De acordo com os princípios já abordados anteriormente, buscamos minimizar as repetições e queremos que as mudanças sejam localizadas, de forma que a lógica esteja perto dos dados que manipula. Nesse sentido, os métodos encapsulam um trecho de código, provendo um escopo fechado para variáveis locais e permitindo chamadas como uma maneira de evitar duplicações.

2.3.1 Fundamentação Teórica

O primeiro fundamento levantado no livro *Clean Code* quanto aos métodos é uma preocupação quanto ao tamanho. Segundo Robert Martin, métodos devem ser pequenos (e se possível menores do que isso). O autor também cita que não existe uma fundamentação científica para tal afirmação, além de não podermos afirmar o que é "pequeno" e definir limitantes quanto ao número de linhas.

A proposta é que cada método seja pequeno o suficiente para facilitar sua leitura e compreensão. Devemos ter em vista a dificuldade de assimilação de grandes porções de informação durante a leitura e o fato de que nem sempre uma instrução é clara. A partir dessas ideias, um método será uma porção curta de código que trabalha com poucas variáveis e tem um nome explicativo que espelha a sua funcionalidade.

Uma maneira bastante interessante de pensar sobre o tamanho dos métodos não é simplesmente contar seu número de linhas, mas compreender bem a tarefa que realiza. Nas palavras de Robert Martin, "Funções deveriam ter uma única tarefa. Deveriam fazê-la bem. E fazê-la somente."

Isso significa que para criar uma lista de números primos usando o algoritmo do *Crivo de Erastótenes*, não queremos ter um único método que contenha todos os detalhes da criação de uma lista de inteiros e a marcação de múltiplos que não tem chance de serem primos. Seria muito difícil entendê-lo e modificar detalhes da implementação. Queremos várias métodos pequenos que fazem cada uma das tarefas necessárias para essa computação, de forma que a leitura seja suficiente e uma documentação externa redundante.

Outra questão teórica importante e enfatizada por Kent Beck são os níveis de abstrações e a simetria de operações dentro de um método. Um incremento de uma variável de instância está fundamentalmente em outro nível de abstração do que a chamada de um método. Não queremos um método com essas duas instruções. Isso possivelmente faz com que o leitor não saiba se uma operação é um detalhe de implementação ou um conceito importante dentro da lógica, além de abrir as portas para mais ocorrências deste mesmo tipo, aumentando a complexidade do código a cada alteração.

Pensando em outros fatores que podem levar a dificuldades para a leitura do nosso código, sabemos que estamos trabalhando em um código limpo quando cada rotina que lemos faz o que esperávamos (Martin, 2008). No contexto de métodos, como seus nomes são a documentação da tarefa que realizam, não queremos que, ao olhar o corpo de um deles, nos deparemos com uma operação que não

imaginávamos. Ao criar um método, temos que considerar que os leitores não terão a mesma linha de pensamento que temos naquele momento e, diante disso, temos que ter uma crescente preocupação com todas as nossas decisões.

O último fundamento teórico quanto aos métodos também diz respeito ao programa como um todo. Queremos ter um fluxo normal bem estabelecido, deixando-o simples e o tratamento de erros separado. Em outras palavras, não queremos que o código fiquei cheio de verificações de erro misturados com a lógica sem erros.

2.3.2 Técnicas

Tendo em vista a fundamentação teórica, a seguir descrevemos algumas técnicas que nos possibilitam criar métodos que se encaixem no estilo de programação buscado.

Composição de Métodos

A Composição de Métodos é a base para a criação de um código limpo. A proposta é compor nossos métodos em chamadas para outros métodos rigorosamente no mesmo nível de abstração.

Usando o exemplo do algoritmo do *Crivo de Erastótenes*, para encontrarmos os primos de 1 a N devemos criar um conjunto de elementos que represente os inteiros de 1 a N, marcar todos números múltiplos de outros e depois coletar todos aqueles que não estão marcados (os primos). Para isso criamos um método da classe *GeradorDePrimos*:

```
def primosAte(N):
  criaInteirosDesmarcadosAte(N)
  marcaMultiplos()
  colocaNaoMarcadosNoResultado()
  retorna resultado
```

Listing 2.5: *Algoritmo do Crivo de Erastótenes decomposto em métodos*

Nesse exemplo, compomos o método *primosAte(N)* a partir de chamadas para outros métodos, cada um com uma tarefa e um nome explicativo. Dessa forma, o leitor pode entender facilmente como funciona o algoritmo de uma maneira geral.

Também ficam bastante claros os níveis de abstrações criados. A função primosAte(n) está em um nível e os métodos que invoca estão todos no mesmo patamar logo abaixo. Queremos criar essa estrutura em cadeia, tornando possível que o leitor só precise entender os níveis que interessem no momento.

A composição de métodos não é uma técnica que desconhecemos, uma vez que a todo momento criamos estruturas dessa maneira sem ter conhecimento da sua efetividade. Entretanto, a facilidade de leitura propiciada faz com que esse seja o padrão mais relevante do livro *Implementation Patterns* segundo o próprio autor (Inf, 2008; Beck, 2007).

Métodos Explicativos (Explaining Methods)

Para auxiliar na expressividade de um trecho de código, podemos criar um método com nome específico do contexto em que será invocado. A operação pode ser bastante simples como um incremento de variável ou a chamada de um método em um objeto guardado em uma variável de instância,

mas o nome dado será de grande valor para a documentação do código. Além disso, a função pode ser reutilizada posteriormente evitando duplicações e encapsulando a operação.

Essa ideia pode ser empregada em diferentes contextos e níveis de abstração. Para Kent Beck, devemos considerar a criação de um método explicativo quando ficamos tentados a comentar uma única linha de código. Abaixo está um exemplo extraído do livro *Implementation Patterns*.

```
flags |= BIT_CARREGAMENTO //Liga bit de carregamento
```

Listing 2.6: *Exemplo de operação pouco clara que recebe comentário*

O comentário pede uma mudança usando um método explicativo.

```
def ligaBitCarregamento():
  flags |= BIT_CARREGAMENTO
```

Listing 2.7: *Exemplo de Método Explicativo que dispensa os comentários*

Em outro exemplo, criamos um método chamado *destaca()* para dar o contexto necessário para uma operação de nível mais baixo.

```
def destaca(palavra):
  palavra.corDoFundo(Cor.amarela)
```

Listing 2.8: *Método destaca() facilita a leitura do código cliente*

Considerando a região de código que chama *destaca()*, as melhorias são diversas com essa alteração. Se a operação fosse simplesmente colocada sem nenhum comentário, provavelmente o leitor se perguntaria por quê a cor de fundo da palavra está sendo alterada naquele contexto. O comentário seria algo como "destacar a palavra no texto", o que se torna obsoleto com o método explicativo.

Por fim, ainda há formas mais implícitas de utilizar métodos explicativos. Quando utilizamos o padrão de projeto *Adapter* (Gamma *et al.*, 1995) para criar uma classe que abstrai uma coleção de elementos, de certa forma estamos criando uma interface cujos métodos façam mais sentido para o domínio de aplicação trabalhado.

Métodos como Condicionais

Um caso específico de método explicativo bastante utilizado é uma chamada para um método que contenha uma expressão booleana como valor de retorno.

```
if(dobra[inicio] != dobra[inicio+1] && inicio < n)
```

Listing 2.9: *Exemplo de uma expressão booleana pouco clara*

Um condicional não muito claro, como o mostrado acima, pode deixar o próprio autor confuso quanto ao seu significado, depois de um certo tempo de sua redação. A criação de uma função que contenha a expressão booleana nos permite escolher um nome apropriado para a circunstância, deixando o condicional claro.

```
def aindaHaDobrasASeremFeitas():
    return dobra[inicio] != dobra[inicio+1] && inicio < n
```

Listing 2.10: *Método que encapsula expressão booleana*

Evitar Estruturas Encadeadas

Tendo em vista a busca por métodos pequenos e com uma única tarefa, um aspecto relevante é a criação de estruturas encadeadas. Se um método tem uma cadeia de *ifs* e *elses*, o leitor terá dificuldades para compreender todos os casos e fluxos possíveis.

Em ambos os livros estudados, os autores enfatizam o uso de chamadas de métodos logo em seguida de um condicional. Novamente a operação que será encapsulada estará dentro de um método com nome expressivo, o que deixa esse método curto e expressivo.

Mais uma vez utilizando o exemplo do *Crivo de Eristótenes*, é necessário marcar todos os números que sejam múltiplos de um primo, representado por um número não marcado.

```
def marcaMultiplos():
  for(candidato = 0; candidato <= limite; candidato++)
    marcaMultiplosSeNaoEstaMarcado(candidato)

def marcaMultiplosSeNaoEstaMarcado(candidato):
  if(naoMarcado(candidato))
    marcaMultiplosDe(candidato)

def marcaMultiplosDe(primo):
  for(multiplo = 2*primo; multiplo < limite; multiplo += i)
    numeros[multiplo].marca()
```

Listing 2.11: *Alternativa para evitar estruturas encadeadas usando funções pequenas*

Cada método encapsula um nível na cadeia de estruturas encadeadas. Ao invés de muitos *fors* e *ifs* encadeados, obtivemos funções pequenas e muito focadas em uma única tarefa que podem ser mais facilmente testadas de forma independente.

Cláusulas Guarda (Guard Clauses)

Outra técnica para evitar o uso de estruturas complexas de condicionais são as cláusulas guarda. Quando criamos uma expressão condicional (if), o leitor naturalmente espera um bloco com a contrapartida (else). A ideia é criar um retorno breve para expressar que uma das partes dessa contraposição é mais relevante. Portanto, a estrutura exemplificada no trecho de código 2.13 é mais recomendada do que no trecho de código 2.12 nos exemplos abaixo, pois revela ao leitor que o fluxo mais relevante é o caso em que ocorre a inicialização, uma vez que, no caso contrário, nenhuma operação é executada.

```
def inicializa():
  if(!jaInicializado)
    //Implementacao do codigo de inicializacao
```

Listing 2.12: *Método sem Cláusula Guarda*

```
def inicializa():
  return if(jaInicializado)
    //Implementacao do codigo de inicializacao
```

Listing 2.13: *Método com Cláusula Guarda*

Objeto Método (Method Object)

Ao dividir os métodos em outros menores com uma única tarefa, podemos criar métodos com grande complexidade e difícil refatoração: muitas variáveis, muito parâmetros necessários e/ou muitas estruturas encadeadas. Diante dessa situação, uma alternativa possível é utilizar um objeto (*Method Object*) que encapsule essa lógica.

O exemplo 2.14 ilustra uma classe *ImpressorDePrimos* cujo trabalho é imprimir o enésimo primo. Sendo assim, calcula tal número através do método *calculaPrimo* que implementa o Crivo de Erastótenes, operação complexa que, apesar de receber poucos parâmetros, possui muitas tarefas.

```
classe ImpressorDePrimos:
  def imprimePrimo(n):
    primo = calculaPrimo(n)
    print "----------------------------------------"
    print "O primo numero " + n + "eh " + primo
    print "----------------------------------------"

  def calculaPrimo(n):
    //implementacao do Crivo de Erastotenes
```

Listing 2.14: *Classe ImpressorDePrimos com método complexo*

Abaixo exemplificada, a classe *CalculadorDoEnesimoPrimo* recebe como parâmetro de seu construtor o valor *n* relativo ao número do primo a ser calculado e o atribui a uma variável de instância. Sua interface é somente composta pelo método *calcula*. Com essa mudança, todos os detalhes da operação complexa foram encapsulados em uma classe que terá testes, aumentando nossa confiança e flexibilidade para refatorações de um método com muitas tarefas.

```
classe ImpressorDePrimos:
  def imprimePrimo(n):
    primo = calculaPrimo(n)
    print "----------------------------------------"
    print "O primo numero " + n + "eh " + primo
    print "----------------------------------------"

  def calculaPrimo(n):
    calculador = CalculadorDoEnesimoPrimo(n)
    return calculador.calcula()


classe CalculadorDoEnesimoPrimo:
  instVar numeroDoPrimoProcurado

  def construtor(n):
    numeroDoPrimoProcurado = n

  def calcula():
    //implementacao do Crivo de Erastotones
```

Listing 2.15: *Classe ImpressorDePrimos delega a tarefa para um novo objeto da classe CalculadorDoEnesimoPrimo*

Tal solução produz uma nova classe que segue o princípio da Proximidade de Lógica e Dados, além de ser coesa e encapsular uma operação complexa. O código cliente é simplificado, uma vez que possui apenas uma delegação para outro objeto.

Minimizar os Argumentos

O número de argumentos se torna bastante importante quando queremos métodos pequenos e com apenas uma tarefa. Se um método recebe muitos argumentos, provavelmente os utiliza para um conjunto de operações e não uma somente. Diante disso, nosso objetivo é sempre minimizá-los por algumas razões. Primeiramente, testes de funções com poucos ou nenhum argumento requerem menores esforços uma vez que as combinações de casos de teste são limitadas. Além disso, teremos um maior acoplamento da classe em que a função está contida com todas as classes dos objetos que recebe, já que é necessário conhecer suas interfaces para utilizá-los. E por fim, o leitor terá que entender todos esses casos de uso.

Com essa motivação queremos evitar parâmetros desnecessários e que confundem o leitor. Os três tópicos subsequentes ilustram abordagens mais específicas quanto ao uso de parâmetros.

- **Evitar *Flags* como Argumentos**

  Se passamos uma *Flag* (booleana ou proveniente de uma enumeração) como argumento para gerar dois tipos de comportamentos de um método, claramente esse método realiza mais do que uma tarefa. O natural seria criar um método para cada uma das tarefas como mostrado no exemplo 2.17 abaixo de uma classe Figura abaixo.

```
def rotaciona(angulo, sentidoHorario):
  if(sentidoHorario == true)
    this.angulo += angulo
  else
    this.angulo -= angulo
```

Listing 2.16: *Método rotaciona() com dois comportamentos através do uso de uma flag como argumento*

```
def rotacionaSentidoHorario(angulo):
  this.angulo += angulo

def rotacionaSentidoAntiHorario(angulo):
  this.angulo -= angulo
```

Listing 2.17: *Dois métodos com uma única tarefa cada*

- **Objeto como Parâmetro**

  Diante de uma lista grande de argumentos, devemos questionar se não existe alguma abstração que une esses elementos. Por exemplo, considere uma classe *FábricaDeMáquinas* cujo construtor é *FábricaDeMáquinas(númeroDeMáquinas, númeroDeFuncionários, horasDeFuncionamento)*. Uma possibilidade seria criar um objeto *EspecificaçãoDeFábrica* que contivesse esses argumentos como variáveis de instância e métodos que fossem úteis para que o código desta *FábricaDeMáquinas* fosse o mais claro possível.

- **Parâmetros como Variável de Instância**

  Na utilização da composição de métodos, frequentemente será preciso passar um argumento para diversos métodos, que possivelmente não o utilizam, somente para satisfazer a necessidade de um deles. Uma solução plausível é transformar esse argumento em uma variável de instância acessível por todos os métodos.

  No entando, ao longo das alterações para minimizar o número de parâmetros dos métodos, não podemos deliberadamente transformar um parâmetro em variável de instância sem o devido cuidado com a abstração da classe. A mudança só faz sentido se a variável de fato pertence ao estado do objeto da classe em questão. Uma variável que é passada em um método e repassado para outro, provavelmente não pertence ao estado do objeto.

  Esse tipo de alteração tem grande importância no design do sistema como um todo. Na seção de Classes esse tópico será abordado e veremos o papel dessa técnica em refatorações de maior escala.

Uso de Exceções

Como foi dito nos fundamentos teóricos desta seção, queremos ter um fluxo normal bem definido e sem interferências do código de tratamento de erros. Nesse contexto, devemos preferir o uso de exceções sobre retornar códigos de erro e valores nulos.

O grande problema com o retorno de códigos de erros e valores nulos está na limpeza do código cliente. Se, ao chamar um método, é necessário utilizar condicionais para verificar qual foi o erro causado ou certificar que não chamaremos um método sobre uma referência nula, o código cliente provavelmente terá que lidar com estruturas encadeadas e muitos casos de teste. Essa decisão viola os princípios de um código expressivo para o leitor uma vez que, durante a leitura, terá que compreender cada um destes erros mesmo que os mesmos não sejam interessantes no momento.

2.4 Classes

Entre os princípios propostos por Kent Beck estão a proximidade da lógica com os dados na qual trabalha e a preocupação quanto às consequências locais. Esses conceitos além de estarem inseridos na justificativa para a adoção do paradigma da orientação a objetos, também estão intimamente relacionados com as classes que compõe o sistema. Queremos que as classes encapsulem dados e operações e tenham uma interface que permita um código cliente com o mínimo de dependências, sempre visando a simplicidade do design e do conteúdo das classes.

Diante da concepção ágil de tomar boas decisões durante o desenvolvimento que proporcionem melhorias imediatas, nessa seção serão discutidas algumas preocupações quanto as classes, buscando facilitar as mudanças no sistema.

2.4.1 Responsabilidades e Coesão

Do mesmo modo que consideramos importante limitar a quantidade de informação que o leitor lida ao ler métodos, queremos que as classes sejam o menores possível. Além de facilitar a leitura e entendimento, programar buscando minimizar o tamanho das classes nos auxilia a criar unidades coesas e a evitar duplicações.

Logicamente, se nossas classes são pequenas, teremos que reunir uma grande quantidade delas, tornando nossos sistemas compostos de muitas classes pequenas. Em uma analogia bastante simples, é mais fácil encontrar um objeto em muitas gavetas pequenas do que em poucas gavetas grandes e repletas.

Princípio da Responsabilidade Única (Single Responsibility Principle)

A questão que surge após a afirmação que as classes deveriam ser pequenas é como definir o que é ser pequena. O número de métodos pode nos dar um bom indicativo quanto a quantidade de operações diferentes que a classe pode executar, mas podemos encontrar classes com poucos métodos e que possuem contextos totalmente diferentes. Por exemplo, uma classe *Controlador* com os métodos *capturaEntrada()*, *criaCorpoHtml()* e *criaCabecalhoHtml()* possui dois tipos de tarefas diferentes: lidar com a criação de HTML e processar a entrada do usuário.

A forma de medir o tamanho de uma classe proposto por Robert Martin está atrelado com a quantidade de responsabilidades que a mesma possui. Podemos pensar em uma responsabilidade como uma razão para mudar. No exemplo acima, para que o sistema seja adaptado para utilizar um novo padrão HTML, é necessário alterar a classe *Controlador*. O mesmo teria que ser feito caso um novo tipo de entrada do usuário fosse concebido. Dessa forma, dizemos que o *Controlador* tem pelo menos duas responsabilidades.

Dentro da ideia de código limpo, queremos que nossas classes sigam o Princípio da Responsabilidade Única (*Single Responsibility Principle*) que, de maneira geral, afirma: as classes deveriam ter uma única responsabilidade, ou seja, ter uma única razão para mudar. Queremos evitar classes como o *Controlador*, de acordo com o princípio SOLID exposto por Robert Martin em *Agile Software Development, Principles, Patterns, and Practices* (Martin, 2002).

É importante destacar que esse princípio não é uma contraposição com as ideias empregadas na concepção da arquitetura através dos Cartões CRC (**C**lasse, **R**esponsabilidade e **C**olaboração (Beck e Cunningham, 1989)) ou do chamado Design Dirigido a Responsabilidades (Rebecca Wirfs-Brock, 2003). Nessas duas abordagens criamos o design da aplicação considerando as classes envolvidas, suas colaborações e responsabilidades. O Princípio da Responsabilidade Única é mais atrelado à implementação, enquanto que nos cartões CRC consideramos principalmente as abstrações. Dessa forma, podemos conceber que um software precisa de uma abstração *Gerente* representada por uma classe, mesmo que na implementação suas responsabilidades sejam obtidas através de delegações para classes pequenas como Objetos Métodos.

Abaixo está um exemplo adaptado do livro *Agile Software Development, Principles, Patterns, and Practices*. A classe Modem faz bastante sentido do ponto de vista abstrato: é responsável pela comunicação entre máquinas estabelecendo a conexão, enviando e recebendo dados e fechando a conexão.

```
classe Modem:
  void ligarParaNumero(numero)
  void envia(dado)
  char recebe()
  void desliga()
```

Listing 2.18: *Classe Modem com duas responsabilidades (omitindo a implementação dos métodos)*

Porém, considerando a associação entre responsabilidade e uma razão para mudar, poderiamos dizer que a classe Modem viola o Princípio da Responsabilidade Única. São duas responsabilidades: tratar da conexão através de *ligarParaNumero* e *desliga* e a segunda é enviar e receber dados usando *enviar* e *receber*.

Uma opção para essa classe seria uma divisão das responsabilidades em duas classes distintas. A nova classe Conexao teria uma única responsabilidade que seria a criação de uma conexão entre dois pontos, enquanto que a classe CanalDeDados ficaria com o envio e recebimento de informação através de uma conexão. Por fim, a antiga classe Modem teria apenas que ligar entre as duas classes, mantendo sua antiga interface para os clientes que já a utilizavam, mas agora delegando seu trabalho para as classes recentemente criadas.

Outro exemplo é apresentado na classe *Email* abaixo. Essa classe é responsável pela criação de um e-mail e a configuração de seus dados assunto, destinatários e conteúdo. Claramente não há nenhuma violação.

```
classe Email:
  adicionaAssunto(String assunto)
  adicionaDestinatarios(Lista destinatarios)
  adicionaConteudo(String conteudo)
```

Listing 2.19: *Classe Email recebendo uma string como conteúdo*

No entanto, vamos supor que se tornou necessária a criação de duas formatações para o conteúdo: uma como HTML e outra como XML. A primeira solução seria a criação do código abaixo.

```
classe Email:
  adicionaAssunto(String assunto)
  adicionaDestinatarios(Lista destinatarios)
  adicionaConteudoComoHTML(String conteudo)
  adicionaConteudoComoXML(String conteudo)
```

Listing 2.20: *Classe Email com duas responsabilidades*

Nesse caso, podemos observar que a classe possui duas responsabilidades, podendo ser alterada tanto para mudanças nos dados que primeiramente guardava, quanto para a criação de novas formatações.

Uma opção de design que preserva o Princípio da Responsabilidade Única é exemplificado no diagrama UML da Figura 2.1. O método *adicionaConteudo* recebe um objeto *Conteudo* e passa a ser somente um *setter* de uma variável de instância de *Email*. Além disso, teremos duas classes *ConteudoHTML* e *ConteudoXML* filhas de *Conteudo*, que só terão como tarefa a formatação de uma string.

Coesão

Voltando ao exemplo da classe *Controlador*, de um ponto de vista técnico, qual crítica poderia ser feita a essa classe? A classe *Controlador* não é coesa. Se tal classe tem a responsabilidade de lidar com a criação de HTML, logicamente terá um conjunto de variáveis de instância e métodos que as utilizam capazes de executar o trabalho em conjunto. O mesmo pode ser dito para o processamento da entrada do usuário. Um leitor que precisa entender apenas um desses contextos se deparará com inúmeros detalhes de implementação tanto da responsabilidade que deseja trabalhar, quanto da que não está interessado.

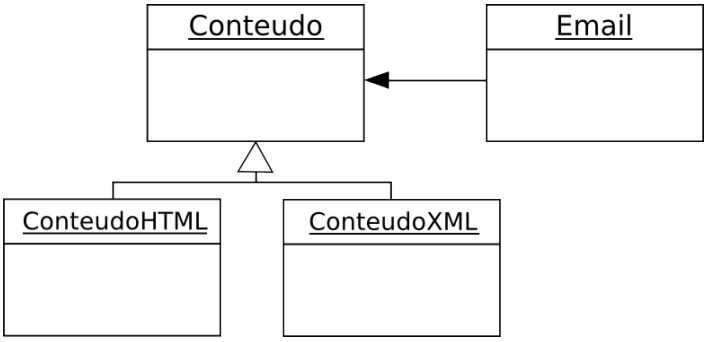


Figura 2.1: *Design que respeita o Princípio da Responsabilidade Única*

Quando a coesão de uma classe é alta, seus métodos e variáveis são codependentes e se unem como uma unidade lógica (Beck, 2007). De maneira mais prática, uma classe é totalmente coesa quando todos os seus métodos usam todas as variáveis de instância.

Dessa forma, a coesão da classe está intimamente relacionada com as responsabilidades que assume. Se uma classe tem muitas responsabilidades, provavelmente terá um conjunto de métodos que pouco se comunicam e usam poucas variáveis em comum, tendo baixa coesão. Quando a coesão é baixa, provavelmente será uma boa ideia criar uma nova classe.

Um bom exemplo de classe extremamente coesa é a implementação de uma estrutura de dados e suas operações. Na implementação simplificada de uma pilha apresentada abaixo, é fácil perceber como os métodos da classe *Pilha* estão intimamente relacionados com suas variáveis de instância.

```
classe Pilha:
  int maxPosicoes
  int topo
  vetor[maxPosicoes] elementos

  def vazia?():
    return topo == 0

  def cheia?():
    return topo == maxPosicoes

  def insere(Elemento elemento):
    levanta_excecao("Pilha cheia") if cheia?
    elementos[topo] = elemento
    topo += 1

  def remove_topo():
    levanta_excecao("Pilha vazia") if vazia?
    topo -= 1
    elementos[topo]
```

Listing 2.21: *Classe coesa que implementa uma pilha*

2.4.2 Acoplamento entre Classes

Outro aspecto muito relevante do ponto de vista da orientação a objetos é o acoplamento entre as classes. Essa medida está atrelada a quanto as classes do sistema dependem uma das outras. As

dependências podem se expressar de diversas formas desde o uso de um método de outra classe até a "perigosa" manipulação e modificação de dados de outra classe.

Tendo em vista a busca por consequências locais e a simplicidade e expressividade do código, não queremos que as classes dependam fortemente umas das outras de forma que o entendimento e a modificação de uma leve a alterações em outras classes. Além disso, a cada dependência gerada, os testes de unidade criados provavelmente precisaram da utilização de artifícios para isolar cada uma das partes sendo mais difíceis de serem compreendidos.

Por outro lado, buscamos um sistema formado por muitas classes coesas e com uma única responsabilidade, fazendo com a comunicação entre os objeto seja inevitável. Diante desse cenário, o objetivo passa a ser a detecção de acoplamentos desnecessários, visando simplificar a comunicação entre os objetos através de suas interfaces.

Como vimos no exemplo da Figura 2.1, uma solução encontrada para o problema foi criar uma depedência entre *Email* com *Conteudo*. O acoplamento criado não deixou as classes mais complexas, mas flexibilizou a criação de novas formatações.

Abaixo estão apresentados alguns conceitos e técnicas relacionados ao acoplamento em um código limpo.

A Lei de Demeter (The Law of Demeter)

A Lei de Demeter diz que um método "M" de uma classe "C" só deveria chamar um método:

- da própria classe;
- de um objeto criado por M;
- de um objeto passado como argumento para M;
- de um objeto guardado em uma variável de instância de C.

O cumprimento dessa heurística visa evitar que uma classe conheça os detalhes de implementação dos objetos que manipula. A invocação de métodos da própria classe não causa nenhum tipo de dependência pela óbvia ausência de algo sobre a qual depender. Os outros tipos de chamadas especificados na Lei causam uma dependência, mas que se limita às interfaces dos objetos manipulados. A classe deve conhecer quais os métodos dos objetos, os argumentos que aceitam e o tipo do valor de retorno, caso houver.

Um exemplo claro de violação são os chamados "trenzinhos de método". Suponha que temos uma classe *JogoDeFutebol* e uma classe cliente "C" executa a seguinte operação:

```
jogoDeFutebol.getTimes().primeiro().getCartoes().count()
```

Listing 2.22: *Um chamado "trenzinhos de método" que exemplifica uma violação da Lei de Demeter*

A chamada para *getTimes()* está de acordo com a Lei de Demeter, pois é uma invocação de um método de um objeto guardado em um variável de instância *jogoDeFutebol* da classe *JogoDeFutebol*. O valor de retorno dessa chamada é uma lista contendo os dois times que não é objeto criado e/ou guardado dentro da classe "C", nem um argumento passado para "M". Dessa forma, a chamada para *primeiro()* sobre essa lista é a primeira das violações à heurística.

Considerando os problemas dessa abordagem e o acoplamento gerado, que partes do sistema a classe "C" conhece? Primeiramente, conhece a classe *JogoDeFutebol*, já que possui um destes objetos guardados em uma variável de instância e chama um método *getTimes()*. Além disso, conhece o valor de retorno desta chamada e sabe que é uma lista composta por objetos da classe *Time* e que, portanto, poderia chamar o método *primeiro()* sobre a lista. Ainda conhece que um Time tem uma coleção de cartões acessada através de *getCartões()*, que aceita a mensagem *count()*.

Esse exemplo mostra bem que tipo de problemas podem ocorrer quando o acoplamento entre as partes do sistema está alta. Claramente, a classe "C" conhece muitas partes do sistema e, a partir desse momento, qualquer mudança nessa estrutura pode quebrar seu código. Quando temos que criar estruturas como a exemplificada, é necessário repensar o design e nos perguntarmos porque a classe "C" precisava ter acesso a todas aquelas informações. Provavelmente, é o caso de fazer mais delegações ao invés de depender tanto dos dados de outros objetos.

Métodos "Invejosos"

Em alguns casos, um método mesmo sem violar a Lei de Demeter, tem um grande acoplamento com outra classe. O exemplo abaixo não viola a Lei de Demeter, já que faz chamadas somente a métodos de um objeto que lhe foi passado como parâmetro.

```
def salarioDoMes(empregado):
  return 0 if empregado.naoTrabalhouEsseMes()
  salarioPorHora = empregado.salarioPorHora()
  bonus = empregado.bonus()
  horasTrabalhadas = empregado.horasTrabalhadas()
  return salarioPorHora * horasTrabalhadas + bonus
```

Listing 2.23: *Método salario() é muito acoplado a classe Empregado sem violar a Lei de Demeter*

O método *salario* possui um grande acoplamento com a classe *Empregado* ao utilizar diversos de seus métodos, de forma que "inveja" os seus dados (Fowler *et al.*, 1999).

Delegação de Tarefa

Para minimizar o acoplamento gerado pelos "trenzinhos de método" e métodos "invejosos", temos que considerar uma mudança nas tarefas realizadas pelas classes envolvidas. Se um método da classe "A" utiliza muitos métodos e dados da classe "B", muito provavelmente sua tarefa pertence a "B" já que queremos aproximar a lógica e seus dados.

Dessa maneira, uma possível solução consiste na alteração do método para que "peça" a classe "B" para que realize uma tarefa através de um novo método. Podemos dizer que o método fez uma "Delegação de Tarefa".

O exemplo abaixo ilustra métodos de uma classe *CalculaReceita*, cujo código poderia se beneficiar com uma Delegação de Tarefa.

```
def calculaTotalDeSalarios():
  total = 0
  for empregado in empregados:
    total += salarioDoMes(empregado)
  return total
```

```
def salarioDoMes(empregado):
  return 0 if empregado.naoTrabalhouEsseMes()
  salarioPorHora = empregado.salarioPorHora()
  bonus = empregado.bonus()
  horasTrabalhadas = empregado.horasTrabalhadas()
  return salarioPorHora * horasTrabalhadas + bonus
```

Listing 2.24: *Método salarioDoMes com problemas de "inveja"*

O método *salarioDoMes* tem problemas de "inveja" sobre a classe *Empregado*, uma vez que faz muitas chamadas e acessos a essa classe ao invés de pedir que realize uma tarefa. O código abaixo exibe uma possível solução.

```
classe CalculaReceita:
  def calculaTotalDeSalarios():
    total = 0
    for empregado in empregados:
      total += empregado.salarioDoMes()
    return total

classe Empregado:
  def salarioDoMes():
    return 0 if naoTrabalhouEsseMes()
    return salarioPorHora * horasTrabalhadas + bonus
```

Listing 2.25: *Redução entre o acoplamento das classes CalculaReceita e Empregado*

Podemos observar algumas mudanças importantes: (i) o método *calculaTotalDeSalarios* ficou muito simples só fazendo um Delegação de Tarefa para a classe *Empregado*; (ii) o código do método *salarioDoMes* também ficou mais simples já que foi transferido para perto dos dados que utiliza; (iii) de forma geral, reduzimos o acoplamento entre as classes pois existe apenas uma chamada de método para fazer a comunicação.

Objeto Centralizador

Os cenários apresentados acima consideram que uma classe depende somente de uma segunda. No entanto, frequentemente criamos métodos que lidam com mais de uma classe, gerando um grande acoplamento entre as envolvidas.

Associada com o objetivo de minimização de dependências e a criação de classes com única responsabilidade, uma maneira de contornar esse cenário é criar uma classe que possa fazer essa relação entre classes. De maneira bastante análoga ao desenvolvimento do Objeto Método (Seção 2.3.2), o intuito é transferir a atividade complexa de centralizar a comunicação entre objetos diferentes para um novo "objeto centralizador".

São dois resultados importantes dessa alteração: (i) isolaremos a complexidade do acoplamento entre classes em uma classe que poderá receber testes e refatorações independentes; (ii) a medida que encapsulamos as dependências entre objetos, obtemos mais classes que podem ser mais facilmente reaproveitadas posteriormente.

Princípio da Inversão de Dependência (Dependency Inversion Principle)

O objetivo da minimização do acoplamento entre nossas classes é facilitar as alterações que serão feitas no sistema. Queremos conseguir estender a infra-estrutura de classes que temos para criar novas funcionalidades.

No exemplo utilizado por Robert Martin em sua coluna para a revista *The C++ Report* (Cpp, 1989-2002) sobre o assunto discutido, temos uma classe *Copiador* que utiliza um *LeitorDeTeclado* para coletar inteiros digitados pelo usuário e um *EscritorDeArquivo* para escrever o dado em um arquivo. Quais alterações precisam ser feitas para adicionar a funcionalidade de ler os inteiros de um arquivo? Será necessário criar mudanças na classe *Copiador* para que possa receber dados coletados de dois tipos diferentes de objetos que lidam com a entrada de dados. Tal alteração só se torna necessária devido a dependência que Copiador tem com classes como *EscritorDeArquivos* e *LeitorDoTeclado*.

A solução ideal para o problema seria que *Copiador* dependesse de abstrações como *Leitor* e *Escritor* que tivesse uma interface definida de forma que subclasses como *LeitorDeTeclado* ou *LeitorDeArquivo* pudessem ser utilizadas polimorficamente. Todos os tipos de *Leitor* teriam que seguir uma abstração ao implementar o método *lerInteiro()* e *Copiador* só precisaria depender dessa interface. Essa mudança seguiria o Princípio da Inversão de Dependência (*Dependency Inversion Principle* (Martin, 1997)) , que diz:

- Classes de alto nível não devem depender de classes de baixo nível. Ambas deveriam depender de abstrações.
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Durante o desenvolvimento, frequentemente essa solução pode não ser clara ou, em um cenário oposto, os programadores tentam prever todas as partes do código que enfrentam esse problema e utilizam a Inversão de Dependência. Entretanto, nenhum dos casos pode ser o ideal para o código. Como afirmado por Kent Beck (Beck, 2007), só devemos investir em flexibilidade em partes do sistema que realmente precisam de flexibilidade. Dessa forma, talvez a melhor solução não precise ser implementada em um primeiro momento, mas quando a necessidade surgir. Provavelmente, nesse instante temos que ter certeza que fizemos uma boa escolha para as próximas alterações.

Outro fato importante a ser considerado quanto a Inversão de Dependência está no isolamento dos testes automatizados. Ao testar classes que dependem de outras classes, queremos isolar a classe sendo testada de forma que os testes sejam de fato unitários. Se a classe em teste depender de abstrações, o trabalho será simplificado. No exemplo da classe *Copiador*, poderíamos criar um *EscritorTeste* e um *LeitorTeste* que tivesse operações convenientes para que os testes de *Copiador* fosse isolados.

2.5 Unindo Conceitos

Nas seções anteriores foram enumeradas diversas características que devem ser consideradas para o desenvolvimento de um código limpo. Através de uma parcela do conjunto apresentado, podemos fazer boas escolhas que trarão melhorias imediatas para o código. Entretanto, o entendimento e aplicações de composições das recomendações podem causar alterações ainda mais poderosas.

Para exemplificar a união dos conceitos apresentados ao longo capítulo, apresentaremos a seguir um conjunto de melhorias para o código de uma classe *Digrafo* que implementa uma lista de adjacência

para representar as arestas de um grafo dirigido, além de calcular a árvore geradora de custo mínimo através do algoritmo de Dijkstra no método *custosAPartirDoVertice*.

Inicialmente, a classe se comunica com duas classes: *Aresta* representa uma ligação de um vértice com outro e o custo desta ligação, enquanto que *FilaDePrioridade* representa uma uma fila com prioridades com as operações *vazia()*, *insere(aresta)* e *verticeDaArestaComCustoMinimo()*.

```
class Digrafo:
  instVar numeroDeVertices
  instVar listaDeAdjacencia

  def construtor(numVertices):
    numeroDeVertices = numVertices
    listaDeAdjacencia = nova Lista(numeroDeVertices)

  def adicionaAresta(origem, destino, custo):
    listaDeAdjacencia[origem] = nova Aresta(destino, custo)

  def arestasDoVertice(vertice):
    return listaDeAdjacencia[vertice]

  def custosAPartirDoVertice(vertice):
    custos = nova Lista(numeroDeVertices)
    fila = nova FilaDePrioridades(numeroDeVertices)

    for i in (1, numeroDeVertices):
      custos[i] = -1

    custos[vertice] = 0
    fila.insere(nova Aresta(0,0))

    while(fila.vazia()):
      verticeDoMomento = fila.verticeDaArestaComCustoMinimo()
      for aresta in (arestasDoVertice(verticeDoMomento)):
        verticeDestino = aresta.verticeDestino()
        custo = aresta.custo()

        if(custos[verticeDestino] == -1):
          custos[verticeDestino] = custos[verticeDoMomento] + custo
          fila.insere(nova Aresta(verticeDestino, custos[verticeDestino]))

        else if(custos[verticeDestino] > custos[verticeDoMomento] + custo):
          custos[verticeDestino] = custos[verticeDoMomento] + custo

    return custos
```

Listing 2.26: *Classe Digrafo com problemas de limpeza*

O problema mais aparente desse código é a complexidade do método *custosAPartirDoVertice*. Um leitor que não conhece o algoritmo teria grandes dificuldades para compreender sua implementação devido a falta de expressividade do método. São muitas linhas de código, diversos detalhes de implementação e estruturas encadeadas complexas para realizar muitas tarefas.

A primeira alteração que pode ser feita é a criação de Métodos como Condicionais (Seção 2.3.2)

e Métodos Explicativos (Seção 2.3.2) para auxiliar a responder questões sobre algumas operações.

Por exemplo, o que significa a expressão booleana *custos[verticeDestino] == -1*? Em uma visão mais detalhada, podemos observar que os custos de todos os vértices foram primeiramente inicializados com a constante "-1", representando que, até momento, o vértice não foi atingido e não sabemos quanto custa um caminho até ele. Uma solução interessante capaz de gerar uma maior clareza e documentação da implementação é a criação dos métodos exibidos no treco de código 2.27.

```
instVar Infinito = -1

def marcaTodosOsVerticesComoNaoAtingidos(custos):
  for i in (1, numeroDeVertices):
    custos[i] = Infinito

def verticeNuncaAtingido(custos, vertice):
  custos[vertice] == Infinito
```

Listing 2.27: *Métodos Explicativos*

Além da criação de métodos com a mesma motivação, poderemos também quebrar as etapas do algoritmo através da Composição de Métodos (Seção 2.3.2). Essa mudança se baseia no desenvolvimento de um conjunto de métodos com nomes significativos para encapsular operações em diferentes níveis de abstração, além de minimizar as estruturas encadeadas. O trecho de código abaixo mostra um método *inicializaCustos* criado para encapsular a inicialização do vetor de custos.

```
def inicializaCustos(vertice):
  custos = nova Lista(numeroDeVertices)
  marcaTodosOsVerticesComNaoAtingidos(custos)
  marcaVerticeInicialComCustoZero(custos, vertice)
  retorn custos
```

Listing 2.28: *Processo de inicialização do vetor de custos fica mais claro com a Composição de Métodos*

Toda inicialização do vetor de custos fica mais clara através desse método. Operações confusas como *custos[vertice] = 0* agora se encontram encapsuladas e com um nome explicativo.

Podemos observar que a contínua aplicação da Composição de Métodos gera um crescimento no número de métodos e no total de linhas da classe, mas cada um dos métodos possui poucas linhas de código e realiza apenas uma tarefa simples e direta. Outra consequência é a necessidade da passagem de argumentos como *custos* e *fila*, uma vez que os métodos de menor nível de abstração são aqueles que efetivamente realizam as operações utilizando-os.

Para solucionar esse aumento no número de argumentos sendo repassados através da classe, podemos promover alguns destes à variáveis de instância (Seção 2.3.2). É fácil notar nos exemplos abaixo como essa alteração limpa o código dos métodos de mais alto nível que não utilizam os argumentos em suas tarefas.

```
instVar custos
instVar fila

def custosAPartirDoVertice(vertice):
  inicializaCustos(vertice)
  inicializaFila()
  atualizaCustosAteAcabarVertices()
```

```
  return custos

def atualizaCustosAPartirDoVerticeComCustoMinimo()
  while(aindaHaVertices()):
    atualizaCustosAPartirDoVerticeComCustoMinimo()

def atualizaCustosAPartirDoVerticeComCustoMinimo():
  verticeDoMomento = fila.verticeDaArestaComCustoMinimo()
  for aresta in (arestasDoVertice(verticeDoMomento)):
    atualizaCustoSeCaminhoMaisBarato(verticeDoMomento, aresta)
```

Listing 2.29: *Métodos de alto nível sem a passagem de argumentos*

Todas as alterações feitas até o momento foram feitas levando em conta a melhoria do código dos métodos que compõe a classe *Digrafo*. Porém, uma grande alteração poderia ser feita quando consideramos a abstração encapsulada por essa classe. Por quais motivos poderia ser alterada? Primeiramente, poderia ser mudada caso fosse necessário implementar o grafo dirigido com uma matriz de adjacência ao invés de uma lista. Também poderiamos querer alterar o algoritmo de Dijkstra para utilizar outra alternativa.

Claramente, a classe *Digrafo* viola o Princípio da Única Responsabilidade. De um ponto de vista mais técnico, podemos observar que as variáveis de instância *custo* e *fila* não são utilizadas pelos métodos *adicionaAresta* e *arestaDoVértice*, mas só são de interesse dos métodos desenvolvidos para a composição do método *custosAPartirDoVertice*.

A solução para essa falta de coesão é a divisão em duas classes. Essa ruptura poderia ser feita de forma a preservar o método *custosAPartirDoVertice* na interface *Digrafo*, implementando o método através da delegação para uma nova classe que encapsula o algoritmo. No entanto, essa decisão causaria uma dependência circular: *Digrafo* dependeria da nova classe *CalculadorCaminhosMinimos* para calcular a árvore geradora de custo mínimo e, por outro lado, *CalculadorCaminhosMinimos* dependeria das arestas e vértices de *Digrafo*.

Uma solução mais interessante é *CalculadorCaminhosMinimos* receber como argumento de seu construtor um objeto *Digrafo*. Dessa maneira, a classe *Digrafo* ficou pequena e coesa com a única responsabilidade de criar a representação do grafo.

```
class Digrafo:
  instVar numeroDeVertices
  instVar listaDeAdjacencia

  def construtor(numVertices):
    numeroDeVertices = numVertices
    listaDeAdjacencia = nova Lista(numeroDeVertices)

  def adicionaAresta(origem, destino, custo):
    listaDeAdjacencia[origem] = nova Aresta(destino, custo)

  def arestasDoVertice(vertice):
    return listaDeAdjacencia[vertice]
```

Listing 2.30: *Classe Digrafo com apenas uma responsabilidade*

O resultado do esforço exemplificado é um conjunto de classes que atingiu grande proximidade dos três valores: expressividade, simplicidade e flexibilidade. O código ficou significantemente mais

expressivo, pois as classes são formadas por muitos métodos com bons nomes de forma que o leitor possa ler cada nível de abstração em uma linguagem bastante próxima da natural.

Além disso, a composição com classes pequenas e extremamente coesas torna mais simples a compreensão de cada uma das partes, uma vez que é necessário trabalhar com pouca informação para entender um método ou uma classe. Se essas características também forem empregadas para as outras classes do sistema, resultam na minimização de duplicações, na aproximação da lógica e dados e na manutenção de consequências locais, provendo maior flexibilidade.

A tabela a seguir resume as técnicas apresentadas ao longo do capítulo.

Técnica	Descrição	Contribuições	Consequências
Composição de Métodos	Compor os métodos em chamadas para outros rigorosamente no mesmo nível de abstração abaixo	• Facilidade de entendimento de métodos menores • Criação de métodos menores com nomes explicativos	- Operações por métodos + Número de parâmetros da classe + Métodos na classe
Métodos Explicativos	Criar um método que encapsule uma operação pouco clara geralmente associada a um comentário	• O código cliente do método novo terá uma operação com nome que melhor se encaixa no contexto	+ Métodos na classe
Métodos como Condicionais	Criar um método que encapsule uma expressão booleana para obter condicionais mais claros	• Facilidade na leitura de condicionais no código cliente • Encapsulamento de uma expressão booleana	+ Métodos na classe
Evitar Estruturas Encadeadas	Utilizar a composição de métodos para minimizar a quantidade de estruturas encadeadas em cada método	• Facilidade para a criação de testes • Cada método terá estruturas mais simples e fáceis de serem compreendidas	- 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	• Estruturas de condicionais mais simples • Leitor não espera por uma contrapartida do condicional(ex: if sem else)	- Estruturas encadeadas na classe
Objeto Método	Criar uma classe que encapsule uma operação complexa simplificando a original (cliente)	• O código cliente terá um método bastante simples • Nova classe poderá ser refatorada sem preocupações com alterações no código cliente • Nova classe poderá ter testes separados	- Operações no método cliente - Responsabilidades da classe cliente + Classes + Acoplamento da classe cliente com a nova classe
Evitar Flags como Argumentos	Ao invés de criar um método que recebe uma flag e tem diversos comportamentos, criar um método para cada comportamento	• Leitor não precisará entender um método com muitos condicionais • Testes de unidade independentes	+ Métodos na classe

Tabela 2.1: *Parte I - Técnicas: descrição, contribuições e consequências*

Técnica	Descrição	Contribuições	Consequências
Objeto como Parâmetro	Localizar parâmetros que formam uma unidade e criar uma classe que os encapsule	• Menor número de parâmetros facilita testes e legibilidade • Criação de uma classe que poderá ser reutilizada em outras partes do sistema	- Parâmetros passados pela classe + Classes + 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	• Não haverá a necessidade de passar longas listas de parâmetros através de todos os métodos	- Parâmetros passados pela classe - 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	• Clareza do fluxo normal sem tratamento de erros através de valores de retornos e condicionais	- Estruturas Encadeadas
Maximizar Coesão	Quebrar uma classe que não segue o Princípio da Responsabilidade Única	• Cada classe terá uma única responsabilidade • Cada classe terá seus testes independentes • Sem interferências na implementação das responsabilidades	+ Classes - Métodos em cada classe - Atributos em cada classe
Delegação de Tarefa	Transferir um método que utiliza dados de uma classe "B" para a "B"	• Redução do acoplamento entre as classes • Proximidade dos métodos e dados sobre os quais trabalham	- Métodos na classe inicial + Métodos na classe que recebe o novo método
Objeto Centralizador	Criar uma classe que encapsule uma operação com alta dependência entre classes	• Simplificação da classe cliente • Redução do acoplamento da classe cliente com as demais • Nova classe poderá receber testes e melhorias independentes	- Operações no método cliente - Responsabilidades da classe cliente + Classes + Acoplamento da classe cliente com a nova classe

Tabela 2.2: *Parte II - Técnicas: descrição, contribuições e consequências*

Capítulo 3

Mapeamento de Código Limpo em Métricas de Código-fonte

Após a discussão teórica realizada na primeira parte dessa monografia, apresentaremos um mapeamento dos conceitos de código limpo para métricas de código-fonte. Nesse mapeamento usamos um conjunto de métricas para automatizar a busca por características do código-fonte. Em seguida, criamos uma interpretação dos valores das métricas, fazendo associações entre eles e as técnicas e conceitos relacionados a limpeza do código. Nosso objetivo é facilitar a detecção de trechos que poderiam sofrer alterações que os tornem mais expressivos, simples e flexíveis.

3.1 Métricas de Código-Fonte

Métricas nos permitem criar mecanismos automatizáveis para detecção de características obtidas através da análise do código-fonte. Elas podem ser usadas como pontos de avaliação da qualidade do software (Meirelles e Kon, 2009). Além disso, no contexto de métodos ágeis, as métricas são importantes parâmetros para *tracking* de projetos (acompanhamento do andamento do projeto), por exemplo, auxiliando na contagem do número de linhas de código já produzidas ou no cálculo da cobertura de teste.

Diante do vasto conjunto existente de métricas, compreender os significados dos valores de todas é uma tarefa custosa, desmotivando o uso de métricas de código-fonte de maneira geral. Essa realidade motivou o desenvolvimento, no Centro de Competência de Software Livre do IME-USP, de ferramentas que automatizam a interpretação dos resultados obtidos. Uma dessas ferramentas é a Kalibro (inicialmente demoninada Crab (Meirelles *et al.*, 2009)), que permite ao usuário configurar intervalos numéricos para possibilitar uma interpretação qualitativa do valor de cada métrica. Assim, podemos usar intervalos como "Bom", "Regular" e "Ruim" ao invés de "0 a 1/3", "1/3 a 2/3" e "2/3 a 1", facilitando o entendimento e a classificação dos aspectos medidos a partir do código.

A dificuldade dessa abordagem consiste em como determinar esses intervalos numéricos. Para definí-los, podemos usar os conceitos apresentados por Robert Martin (Martin, 2008) e outros autores ou elaborar testes empíricos e estatísticos. Entretanto, através dessas abordagens, fixaríamos valores que não levariam em consideração os diferentes domínios de aplicação, linguagens e outros aspectos relevantes para definir a maneira que programamos, como comentado por Kent Beck em *Implementation Patterns* (Beck, 2007).

Partindo dessa realidade, adotamos uma abordagem baseada em cenários para identificar trechos de código com características indesejáveis. Nesses cenários, elaboramos um contexto criado a partir de poucos conceitos de código limpo no qual um pequeno conjunto de métricas é analisado e interpretado através da combinação de seus valores. Por exemplo, conceitualmente buscamos métodos pequenos e com apenas uma tarefa (Seção 2.3). Sendo assim, elaboramos um cenário que contextualiza a interpretação de métricas 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 pesquisa não pretende afirmar 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.

3.2 O Mapeamento

O mapeamento proposto neste trabalho visa facilitar a procura por problemas quanto a limpeza do código. A ideia é criar cenários que relacionem os conceitos de código limpo com métricas de código-fonte. Cada cenário é descrito através dos seguintes componentes:

- **Conceitos de Limpeza Relacionados**: conceitos de limpeza de código que motivaram a criação do cenário.
- **Características**: indicam a presença de falhas em relação as referências apresentadas.
- **Métricas**: mecanismos que permitem analisar o código a procura das características do cenário.
- **Objetivo Durante a Refatoração**: quais devem ser as ideias principais durante a refatoração para eliminar o problema.
- **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 afirmar se existem problemas no código, mas sim indicar partes dele que talvez possam ser melhoradas de acordo com os conceitos discutidos nesse trabalho.

Para incentivar a avaliação do código ao longo de seu desenvolvimento, é muito importante que a interpretação dos valores das métricas seja 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.

Até o momento, 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.

Existem dois tipos de métricas de código-fonte. Algumas avaliam características de métodos e outras de classes. As métricas de classe são normalmente somas ou médias dos valores das métricas

de métodos. Essa divisão de alvo da análise das métricas também existe no contexto dos cenários, que discutiremos adiante.

3.2.1 Conjunto de Métricas

Abaixo apresentamos o conjunto selecionado de métricas.

NC (*Number of Calls*): Número de Chamadas

A métrica NC[1] calcula o número de métodos e atributos internos e externos usados por um método. Membros externos são aqueles que pertencem a classes não relacionadas hierarquicamente com a classe da operação analisada. Os internos são os pertencentes a mesma classe.

O valor dessa métrica pode variar entre zero e a soma do número de atributos e métodos presentes no sistema. Obtemos zero quando o método em análise não acessa nenhum atributo e não faz chamadas a métodos.

NEC (*Number of External Calls*): Número de Chamadas Externas

A métrica NEC calcula o número de métodos e atributos externos acessados por um método. Definimos como externos os membros pertencentes a classes não relacionadas hierarquicamente com a classe do método em análise.

NEC é uma adaptação da métrica ATFD (*Access To Foreing Data* (Marinescu, 2002)) que calcula o número de atributos de classes não relacionadas que são acessados diretamente ou através de chamadas a métodos de acesso.

Seu valor pode variar de zero até o número total de métodos e atributos do sistema. Nesse caso, zero indica que não há comunicação entre o método e qualquer outra classe sem relação com a sua.

Calculamos a média de NEC quando precisamos avaliar o número médio de chamadas de métodos e acessos a atributos externos por método de uma classe.

ECR (*External Calls Rate*): Taxa de Chamadas Externas

A métrica ECR calcula a taxa de chamadas externas realizadas por um método. Para isso, são necessários o número de chamadas externas (NEC) e o total de chamadas (NC) realizadas pelo método. O valor da taxa será o resultado da divisão de NEC por NC.

Essa métrica é uma adaptação da LAA (*Locality of Attribute Accesses* (Lanza e Marinescu, 2006)) que calcula o número de atributos pertencentes a mesma classe que o método em análise, dividido pelo número total de variáveis acessadas (incluindo atributos usados através de métodos de acesso).

Seu valor varia entre zero e 1. Obtemos zero quando o método não possui chamadas externas e 1 quando todas as chamadas realizadas são externas.

NCC (*Number of Classes Called*): Número de Classes Chamadas

A métrica NCC calcula o número de classes das quais métodos são chamados ou atributos são acessados por um método. Nesse calculo não contamos classes que tenham relação hierarquica com a classe do método em análise.

[1] A métrica NC foi elaborada para esse trabalho, pois não a encontramos nas referências utilizadas.

Essa métrica é uma adaptação da FDP (*Foreign Data Providers* (Lanza e Marinescu, 2006)) que calcula o número de classes das quais atributos são acessados. Seu valor varia entre zero e o número total de classes do sistema. Obtemos zero quando o método não utiliza elementos de outras classes.

NRA (*Number of Reachable Attributes*): Número de Atributos Alcançáveis

A métrica NRA[2] calcula o número de atributos alcançáveis a partir de um método. Um atributo pode ser alcançado direta ou indiretamente. O acesso é direto quando o atributo é usado no próprio corpo do método, e é indireto quando existe uma chamada para algum método que usa o atributo direta ou indiretamente.

Para descobrirmos quantos atributos são alcançáveis por um método, podemos criar um grafo com atributos e métodos como vértices e os acessos e chamadas a eles como arestas. Com esse grafo criado, podemos encontrar os vértices atingíveis a partir do vértice que representa o método em análise. Então, contamos quantos deles são atributos. E assim, obtemos o número de atributos alcançáveis pelo método.

O valor dessa métrica varia entre zero e o total de atributos da classe do método. Obtemos zero quando o método não alcança nenhum atributo de sua classe e o valor máximo quando ele usa direta ou indiretamente todos os atributos de sua classe.

MaxNesting (*Maximum Nesting Level*): Nível Máximo de Estruturas Encadeadas

A métrica MaxNesting (Lanza e Marinescu, 2006) calcula o nível máximo de estruturas encadeados presentes no corpo de um método.

Seu valor varia entre zero e a quantidade total de fluxos condicionais do método. Obtemos zero quando o método não possui fluxos condicionais e atingimos o valor máximo quando todas as quebras condicionais presentes no método encontram-se em níveis diferentes da mesma estrutura.

Os comandos *if*, *while* e *for* são alguns exemplos de controladores de fluxo bastante usados em diversas linguagens.

CYCLO *Cyclomatic Complexity*: Complexidade Ciclomática

A métrica CYCLO (McCabe, 1976) calcula o número de caminhos linearmente independentes no método analisado, conhecido como complexidade ciclomática.

Seu valor mínimo é 1 e não existe um limite máximo para o seu resultado. Obtemos 1 quando não há quebras do fluxo principal. Cada estrutura de controle de fluxo presente no corpo do método adiciona 1 no valor total da CYCLO.

Quando os valores de MaxNesting e CYCLO são próximos, sabemos que muitos dos controladores de fluxo presentes no método estão encadeados em uma mesma estrutura. Quando são muito distantes, podemos concluir que os controladores estão espalhados em estruturas diferentes e que elas não são muito profundas.

NP: Número de Parâmetros

A métrica NP calcula o número de parâmetros de um método. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado. Obtemos zero quando o método avaliado não possui

[2] A métrica NRA foi elaborada para esse trabalho, pois não a encontramos nas referências utilizadas.

parâmetros.

Calculamos a média de NP quando precisamos avaliar a média do número de parâmetros por método de uma determinada classe.

NRP: Número de Repasses de Parâmetros

A métrica NRP[3] calcula o número de parâmetros recebidos pelo método em análise e repassados como argumento em chamadas a outras operações pertencentes a sua classe.

Seu valor varia entre zero e o número de parâmetros do método (indicado pela métrica NP). Obtemos zero quando nenhum parâmetro é repassado. O valor máximo é atingido quando todos os parâmetros são usados como argumentos em alguma chamada de operação realizada no corpo do método.

Calculamos a média de NRP quando precisamos avaliar a média do número de repasses de parâmetros por método de uma classe.

LOC (*Lines of Code*): Número de Linhas

A métrica LOC (Lorenz e Kidd, 1994) calcula o número de linhas efetivas de um método. Linhas compostas apenas por comentários ou vazias não são consideradas efetivas.

NOA (*Number of Attributes*): Número de Atributos

A métrica NOA calcula o número de atributos de uma classe. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado. Obtemos zero quando a classe analisada não possui atributos.

NOM (*Number of Methods*): Número de Métodos

A métrica NOM calcula o número de métodos de uma classe. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado. Obtemos zero quando a classe não possui métodos.

SLOC: Soma do Número de linhas

A métrica SLOC calcula a soma do número de linhas efetivas de todos os métodos de uma classe. Ou seja, seu resultado é a soma dos valores do LOC de cada método da classe.

AMLOC: Média do Número de Linhas Por Método

AMLOC calcula a média do número de linhas efetivas por método da classe analisada.

LCOM4 (*Lack of Cohesion of Methods*): Falta de Coesão Entre Métodos

A métrica LCOM4 (Hitz e Montazeri, 1996) calcula a falta de coesão dos métodos da classe analisada. Para isso, é montado um grafo cujos vértices são os atributos e métodos da classe. Cada aresta desse grafo representa um acesso a atributo ou uma chamada de método através de um outro método. Nesse grafo não são inseridos métodos construtores.

O valor de LCOM4 é o número de componentes conexos desse grafo. Seu valor mínimo é 1 e não existe um limite máximo para o seu resultado. Obtemos 1 quando a classe não possui subdivisões, ou

[3] A métrica NRP foi elaborada para esse trabalho, pois não a encontramos nas referências utilizadas.

seja, quando a classe é bastante coesa. Normalmente, desejamos classes que não possuem conjuntos isolados de métodos e atributos, pois o número de subdivisões costuma indicar a quantidade de classes que deveriam existir para que cada uma fosse responsável por apenas uma responsabilidade (Seção 2.4.1).

3.2.2 Os Cenários

Método Grande

Os conceitos tratados no Capítulo de Código Limpo que constituem este cenário, denominado Método Grande, são: Composição de Métodos (Seção 2.3.2), a necessidade de Evitar Estruturas Encadeadas (Seção 2.3.2) e, a possibilidade de usar Cláusulas Guarda (Seção 2.3.2).

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.

Durante uma refatoração deste cenário, nosso objetivo é 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.

Após a refatoração de todas as ocorrências deste cenário na classe em análise, esperamos que a sua 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 complexidade 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.

Possuir muitas funcionalidades e parâmetros são mais duas características deste cenário. Porém, não precisamos nos preocupar com elas diretamente. Suas quantidades serão reduzidas como consequência da refatoração baseada nos outros aspéctos citados.

Para exemplificar este cenário, vamos analisar o método *custosAPartirDoVertice* (trecho de código 3.1). Nessa avaliação, devemos calcular os valores das métricas que indicam a presença das características do problema descrito.

```
def custosAPartirDoVertice(vertice):
  custos = novo Lista(numeroDeVertices)
  fila = nova FilaDePrioridades(numeroDeVertices)

  for i in (1, numeroDeVertices):
    custos[i] = -1

  custos[vertice] = 0
  fila.insere(nova Aresta(0,0))
```

```
  while(fila.vazia()):
    verticeDoMomento = fila.verticeDaArestaComCustoMinimo()
    for aresta in (arestasDoVertice(verticeDoMomento)):
      verticeDestino = aresta.verticeDestino()
      custo = aresta.custo()

      if(custos[verticeDestino] == -1):
        custos[verticeDestino] = custos[verticeDoMomento] + custo
        fila.insere(nova Aresta(verticeDestino, custos[verticeDestino]))

      else if(custos[verticeDestino] > custos[verticeDoMomento] + custo):
        custos[verticeDestino] = custos[verticeDoMomento] + custo

  return custos
```

Listing 3.1: *O método custosAPartirDoVertice é um exemplo de Método Grande*

Quando contamos o número de linhas efetivas desse método, obtemos *LOC = 17*. Ao contabilizar a quantidade de controladores de fluxo, chegamos a *CYCLO = 6*. Por fim, temos que a profundidade máxima das estruturas encadeados, medida por MaxNesting, é igual a 4. Baseado na busca por métodos pequenos, esses valores podem ser indícios de que esse método possui problemas relacionados a limpeza de código por ter características de um método grande.

O trecho de código 3.2 é uma possível refatoração para o método *custosAPartirDoVertice*. Nesse caso, reduzimos a quantidade de tarefas do método, passando-as para outros métodos menores: *inicializaCustosEFila* e *atualizaCustosAteAcabarVertices*. Esses métodos também são pequenos pois quebram suas tarefas e as passam para outras operações que seguirão esse mesmo princípio. Quando calculamos as métricas deste cenário no método refatorado, obtemos *LOC = 3*, *CYCLO = 1* e *MaxNesting = 0*. Comparando com os resultados antigos, podemos observar uma grande redução nos valores após a refatoração. Dados os novos resultados, podemos dizer que o método *custosAPartirDoVertice* não está mais no cenário de método grande.

```
def custosAPartirDoVertice(vertice):
  inicializaCustosEFila(vertice)
  atualizaCustosAteAcabarVertices()
  return custos

def inicializaCustosEFila(vertice):
  verticeOrigem = vertice
  inicializaCustos()
  inicializaFila()

def atualizaCustosAteAcabarVertices()
  while(aindaHaVertices()):
    atualizaCustosAPartirDoVerticeComCustoMinimo()
```

Listing 3.2: *Uma possível refatoração para o método custosAPartirDoVertice deixar de ser um método grande*

Método com Muitos Fluxos Condicionais

Os conceitos de código limpo que constituem este cenário são a Composição de Métodos (Seção 2.3.2), Evitar Estruturas Encadeadas (Seção 2.3.2) e o Uso de Exceções (Seção 2.3.2). 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).

Durante uma refatoração deste cenário, buscamos 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.

Para exemplificar este cenário, usaremos o método *deletaPaginasETodasAsReferencias* (trecho de código 3.3). Quando analisamos esse bloco de código usando as métricas indicadas neste cenário, encontramos os valores $CYCLO = 4$ e $MaxNesting = 3$. Se considerarmos esses valores como indícios de que temos muitos controladores de fluxo e uma longa estrutura desses controladores encadeados, podemos dizer que esse método tem muitos fluxos condicionais.

```
def deletaPaginasETodasAsReferencias(pagina):
  if(deletaPagina(pagina) == E_OK):
    if(registro.deletaReferencia(pagina.nome) == E_OK):
      if(configChaves.deletaChave(pagina.nome.fazChave()) == E_OK):
        logger.log("pagina deletada.")
      else:
        logger.log("confgChave nao foi deletado."")
    else:
      logger.log("deletaReferencia do registro falhou.")
  else:
    logger.log("falha ao deletar a pagina")
    return E_ERROR
```

Listing 3.3: *O método deletaPaginasETodasAsReferencias é um exemplo de método com muitos fluxos condicionais*

Analisando o código a procura de possíveis refatorações, observamos que esse método realiza três operações e a verificação de falha para cada uma delas. Esse tratamento de erros poderia ser feito usando *try catch*. Essa mudança diminuiria o valor de *MaxNesting* e de *CYCLO*, reduzindo o problema encontrado. O resultado dessa refatoração é o método exposto no trecho de código 3.4.

```
def deletaPaginasETodasAsReferencias(pagina):
  try:
    deletaPagina(pagina)
    registro.deletaReferencia(pagina.nome)
    configChaves.deletaChave(pagina.nome.fazChave())
  catch(Exception e):
    logger.log(e.pegaMensagem())
```

Listing 3.4: *Uma refatoração usando try catch para o método deletaPaginasETodasAsReferencias*

Com essa refatoração deixamos o método mais simples e fora deste cenário.

Método com Muitos Parâmetros

Este cenário tem como base a técnica do uso de Objeto com Parâmetro (Seção 2.3.2). A principal característica de um método que está nesse contexto é ter um elevado número de parâmetros (valor alto de NP).

Durante uma refatoração desse cenário, o objetivo é 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.

Para exemplificar este cenário, usaremos o método *calculaQuantidadeDeDias* (trecho de código 3.5). Quando calculamos sua quantidade de parâmetros, obtemos $NP = 6$. Esse valor pode ser considerado alto para a métrica NP, assim é factível dizer que esse método está no contexto de método com muitos parâmetros.

```
def calculaQuantidadeDeDias(diaInicial, mesInicial, anoInicial, diaFinal, mesFinal,
    anoFinal):
  ...
```

Listing 3.5: *O método calculaQuantidadeDeDias é um exemplo de método com muitos parametros*

Ao analisarmos os parâmetros presentes nesse exemplo, podemos notar que eles representam duas datas compostas por *dia, mês e ano*, que indicam o início e o fim de um período. Então, uma refatoração possível para esse código seria criar uma classe *Data* da qual cada objeto representasse uma dessas datas. Assim, reduziríamos o valor de *NP* de seis para dois, eliminando o problema de alto número de parâmetros desse método (trecho de código 3.6).

```
def calculaQuantidadeDeDias(dataInicial, dataFinal):
  ...
```

Listing 3.6: *O método calculaQuantidadeDeDias depois da refatoração para redução do número de parâmetros*

Muita Passagem de Parâmetros Pela Classe

Este é o primeiro cenário que descrevemos relacionado com a avaliação da classe como um todo e não de métodos separados. Os conceitos de código limpo que constituem este cenário são o uso de Objeto como Parâmetro (Seção 2.3.2) e a transformação de parâmetros em variáveis de instância (Seção 2.3.2).

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. O método *acumulaRelatorioDasMetricasGlobais* (trecho de código 3.7) exemplifica a existência de parâmetros que são apenas repassados em chamadas a outras operações. Os três parâmetros desse exemplo (*totaisDasMetricasDeModulo*, *contagensDeModulo* e *listaDeValores*) possuem essa característica.

```
def acumulaRelatorioDasMetricasGlobais(totaisDasMetricasDeModulo,
    contagensDeModulo, listaDeValores):
  resumo = inicializaResumo()
  adicionaTotaisDasMetricasDeModulo(resumo, totaisDasMetricasDeModulo)
  adicionaContagensDeModulo(resumo, contagensDeModulo)
  adicionaValoresEstatisticos(resumo, listaDeValores)
  adicionaFatorTotalDeAcoplamento(resumo, contagensDeModulo)
  return resumo
```

Listing 3.7: *O método custosAPartirDoVertice é um exemplo de método com muito repasse de parâmetros*

Os objetivos durante uma refatoração deste cenário 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.

Os resultados esperados após a refatoração do código são 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 (Seção 2.4.2). 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 (Seção 2.4.2).

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).

Para exemplificar este cenário, usaremos o método *salarioDoMes* da classe *CalculaReceita* (trecho de código 3.8). Quando calculamos a taxa de chamadas externas realizadas por esse método, obtemos $ECR = NEC/NC = 4/4 = 1$. O resultado é o valor máximo para essa métrica, o que indica que o método possui apenas chamadas externas e que, portanto, tem alto acoplamento. Devemos, então, calcular a quantidade de classes diferentes que são chamadas por esse método. O resultado é $NCC = 1$. Esse valor indica que todos os elementos externos usados são da mesma classe. Como a taxa de chamadas externas é alta e todos os elementos externos usados são da mesma classe, podemos dizer que o método *salarioDoMes* está no cenário de método "invejoso".

```
classe CalculaReceita:
  def calculaTotalDeSalarios():
    total = 0
    for empregado in empregados:
      total += salarioDoMes(empregado)
    return total
```

```
  def salarioDoMes(empregado):
    return 0 if empregado.naoTrabalhouEsseMes()
    salarioPorHora = empregado.salarioPorHora()
    bonus = empregado.bonus()
    horasTrabalhadas = empregado.horasTrabalhadas()
    return salarioPorHora * horasTrabalhadas + bonus
```

Listing 3.8: *Método salarioDoMes com problemas de "inveja"*

Para resolver esse problema, podemos passar para a classe empregado o método *salarioDoMes* (trecho de código 3.9). Quando isso acontecer os valores das suas métricas serão $ECR = NEC/NC = 0/4 = 0$ e $NCC = 0$, o que resolve o seu problema de "inveja" dada a redução do acoplamento entre as classes *CalculaReceita* e *Empregado*.

```
classe Empregado:
  def salarioDoMes():
    return 0 if naoTrabalhouEsseMes()
    return salarioPorHora * horasTrabalhadas + bonus

classe CalculaReceita:
  def calculaTotalDeSalarios():
    total = 0
    for empregado in empregados:
      total += empregado.salarioDoMes()
    return total
```

Listing 3.9: *Redução do acoplamento entre as classes CalculaReceita e Empregado*

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 que é calculado através da métrica NEC. Os conceitos de código limpo que constituem este cenário são o de Objeto Centralizador (Seção 2.4.2) e o de Objeto Método (Seção 2.3.2).

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.

Para exemplificar este cenário, usaremos o método *adicionaDadosDaRodada* (trecho de código 3.10). Quando o avaliamos usando as métricas *ECR* e *NCC* obtemos valores elevados, pois nesse código são usados seis métodos externos de quatro classes diferentes. Com esses resultados dizemos que esse método é dispersamente acoplado.

```
classe CalculaReceita:
```

```
def calculaTotalDeSalarios():
  total = 0
  for empregado in empregados:
    total += salarioDoMes(empregado)
  return total

def salarioDoMes(empregado):
  return 0 if empregado.naoTrabalhouEsseMes()
  salarioPorHora = empregado.salarioPorHora()
  bonus = empregado.bonus()
  horasTrabalhadas = empregado.horasTrabalhadas()
  return salarioPorHora * horasTrabalhadas + bonus
```

Listing 3.10: *Método "adicionaDadosDaRodada" dispersamente acoplado*

Classe Pouco Coesa

Os conceitos de código limpo que constituem este cenário são a maximização da Coesão (Seção 2.4.1) e o Princípio da Responsabilidade Única (Seção 2.4.1). Assim como o cenário que indica muita passagem de parâmetros pela classe, esse contexto analisa a classe como um todo.

A principal característica de uma classe pouco coesa é possuir subdivisões em grupos de métodos e atributos que não se relacionam (valor de LCOM4 $>$ 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.

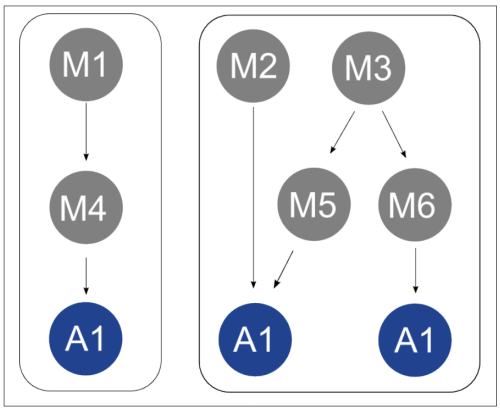


Figura 3.1: *Representação abstrata de uma classe pouco coesa*

Na figura acima temos a representação abstrata de uma classe. Os círculos cinzas são os métodos e os azuis atributos da classe. As ligações representam chamadas e acessos entre esses elementos. Quando contamos os grupos de elementos conexos, obtemos $LCOM4 = 2$, e no cálculo do número médio de atributos atingidos por cada método, obtemos *média de NRA* $= 1.16$. Como o valor de $LCOM4$ é maior do que um e o de NRA é razoavelmente menor do que o número total de atributos ($NOA = 3$), podemos classificar essa classe como pouco coesa.

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).

3.3 Resumo dos Cenários

A tabela abaixo resume os cenários apresentados ao longo deste capítulo.

Cenário	Conceitos de Limpeza	Características	Objetivos Durante a Refatoração	Resultados Esperados
Método Grande	• Composição de Métodos • Evitar Estruturas Encadeadas • Uso de Cláusula Guarda	• Grande número de linhas efetivas (alto LOC) • Muitas estruturas controladoras de fluxo (alto CYCLO) • Muitas estruturas controladoras de fluxo encadeadas (alto MaxNesting)	• Reduzir LOC do método • Reduzir complexidade ciclomática (CYCLO) • Reduzir MaxNesting	• Redução da média de LOC por método da classe • Não há alteração da complexidade ciclomática (CYCLO) da classe • Redução do MaxNesting dos métodos da classe
Método com Muitos Fluxos Condicionais	• Composição de Métodos • Evitar Estruturas Encadeadas • Uso de Exceções	• Muitas estruturas controladoras de fluxo (alto CYCLO) • Muitas estruturas controladoras de fluxo encadeadas (alto MaxNesting)	• Reduzir complexidade ciclomática do método (CYCLO) • Reduzir MaxNesting	• Redução do MaxNesting dos métodos da classe • OBS: - a complexidade ciclomática pode não reduzir, pois, em alguns casos, o uso de try catch mantêm a quantidade de quebra de fluxo. - pode acontecer um aumento no número de métodos da classe quando criamos um método separado para cada quebra de fluxo do código.
Método com Muitos Parâmetros	• Objeto como Parâmetro	• Elevado número de parâmetros (alto NP)	• Minimizar o número de parâmetros (NP)	• Redução do número de parâmetros (NP) • Aumento do número de classes
Muita Passagem de Parâmetros na Classe	• Objeto como Parâmetro • Parâmetros como Variáveis de Instância	• Elevada média de parâmetros repassados pela classe (alta média de NRP) • Elevado número médio de parâmetros por método (alta média de NP)	• Reduzir o número de repasses de parâmetros pela classe (NRP) • Minimizar o número de parâmetros por método (NP)	• Redução da média do número de parâmetros por método (média de NP) • Redução do número de repassagem de parâmetros (soma de NRP) • Possível aumento do número de variáveis de instância • aumento do número de classes

Tabela 3.1: *Parte I - Resumo dos cenários apresentados neste trabalho*

Cenário	Conceitos de Limpeza	Características	Objetivos Durante a Refatoração	Resultados Esperados
Método "Invejoso"	• Delegação de Tarefa	• Elevado uso de métodos e atributos externos (alto ECR) • Usa poucas classes diferentes (baixo NCC)	• Minimizar o número de chamdas externas (NEC)	• Redução do número de chamadas externas (NEC) • Redução da taxa de chamadas externas (ECR) • Redução da média de chamadas externas (média de NEC) da classe
Método Dispersamente Acoplado	• Objeto Centralizador • Objeto Método	• Elevado uso de métodos e atributos externos (alto ECR) • Usa várias classes diferentes (alto NCC)	• Minimizar o número de chamdas externas (NEC)	• Redução do número de chamadas externas (NEC) • Redução do número de chamadas externas (NCC) • Aumento do número de classes • Redução da média de chamadas externas (média de NEC) da classe
Classe Pouco Coesa	• Maximização da Coesão • Princípio da responsabilidade única	• Classe subdividida em grupos de métodos e atributos que não se relacionam (LCOM4 > 1) • Métodos atingem em média poucos atributos da sua classe (média de NRA muito menor do que NOA)	• Reduzir a subdivisão da classe aumentando sua coesão (reduzir LCOM4) • Reduzir a diferença entre média de atributos atingidos por meetodo (média de NRA) e o número de atributos da classe (NOA).	• Aumento do número de classes • Classes novas com as seguintes características: - LCOM4 = 1 - média de NRA próxima de NOA

Tabela 3.2: *Parte II - Resumo dos cenários apresentados neste trabalho*

Capítulo 4

Estudo de Caso

Utilizando as técnicas e conceitos relacionados a um código limpo e seu mapeamento em métricas de código fonte, apresentamos ao longo desse capítulo um estudo de caso sobre o código da ferramenta Analizo (Terceiro *et al.*, 2010). O objetivo é analisar sua versão atual olhando para as métricas e as interpretações anteriormente apresentadas para detectar melhorias possíveis. Ao final, destacaremos as principais mudanças e contribuições para a sua "limpeza".

4.1 Analizo

A *Analizo* (Terceiro *et al.*, 2010) é um conjunto de ferramentas livres para análise e visualização do código-fonte. Atualmente, ela permite a extração e cálculo de 22 de métricas de código-fonte, além da geração de grafos de dependência e o acompanhamento da evolução do software. Seu principal diferencial em relação aos demais aplicativos do mesmo domínio é o suporte a múltipas linguagens, o que a possibilita trabalhar com código Java, C++ e C.

A escolha da Analizo para o estudo de caso se deu devido à familiaridade com as ferramentas, uma vez que grande parte de seu desenvolvimento foi feito durante o nosso projeto de Iniciação Científica, em colaboração com os doutorandos Paulo Meirelles da Universidade de São Paulo e Antônio Terceiro da Universidade Federal da Bahia.

Durante o processo, o esforço foi dirigido na implementação de métricas para o projeto de forma a possibilitar as pesquisas relacionadas a qualidade de projetos de software livre através do cálculo de métricas de código-fonte (Meirelles *et al.*, 2010; Meirelles e Kon, 2009; Morais *et al.*, 2009; Terceiro *et al.*, 2010), bem como tornar as métricas providas pela Analizo compatíveis com as selecionadas no contexto do projeto Qualipso, ao qual a referida iniciação científica foi vinculada.

Por fim, salientamos que a implementação das métricas da Analizo foi anterior aos estudo para este trabalho. Dessa forma, não há uma correspondência direta entres as métricas discutidas no mapeamento com os conceitos de código limpo e as métricas da Analizo.

Tendo em vista que, a maior parte do código na Analizo foi adicionado sem significativa preocupação quanto à limpeza do código, visualizamos a possibidade de aplicar os conceitos estudados sobre código limpo nessa ferramenta. Assim sendo, esse estudo de caso se baseia na possibilidade de promover melhorias de forma a deixar seu código mais limpo, o que constitui em mais uma contribuição a esse software livre e demonstra na prática o potencial das técnicas discutidas neste trabalho.

O foco do estudo é a parte da ferramenta denominada *Analizo Metrics* responsável pelo cálculo das métricas de código-fonte providas pela Analizo. Enfatizamos que, essa ferramenta é inteiramente

escrita na linguagem *Perl*, seguindo o paradigma da Orientação a Objetos. Entretando, como não há aplicativo capaz de calcular as métricas selecionadas nessa linguagem, todos os valores foram obtidos manualmente por nós, seguindo o algoritmo de cada métrica, ao longo das refatorações realizadas.

A arquitetura da *Analizo Metrics* é bastante simples e composta por poucas classes. No momento em que o usuário fornecesse um arquivo sobre o qual deseja obter as métricas, um aplicativo auxiliar é chamada para extrair as características do código-fonte e construir um objeto da classe *Model*. Esse objeto abstrai uma representação simplificada do código e detém informações como a lista de todos os módulos do código avaliado, além de características de suas variáveis e funções [1].

O passo seguinte é feito pela classe *Metrics*, que efetua o cálculo de métricas de dois tipos: (i) métricas de módulo, como o número de linhas de funções e número de variáveis de um dado módulo; (ii) métricas globais que contabilizam os valores do primeiro conjunto para o cálculo de outras métricas como COF (Fator de Acoplamento) e valores estatísticos como a variância e média.

Esse estudo de caso é centrado na melhoria da classe *Metrics*, cujo estado inicial continha: (i) um conjunto de métodos que, dado um módulo, calcula o valor de uma métrica deste módulo; (ii) um método *report_module* que retorna um dicionário que associa o nome de uma métrica e seu valor sobre um dado módulo e, (iii) um método *report* que combina os valores das métricas de cada módulo, calcula as métricas globais e retorna um YAML (*YAML Ain't Markup Language*) contendo todos os valores.

4.2 Estado Inicial

Inicialmente, a característica central da classe *Metrics* 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.

[1] Como a ferramenta também analisa códigos não orientados a objetos foram escolhidos os termos "módulo", "variáveis" e "funções" para generalizar os paradigmas.

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.

```
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);
}
```

Listing 4.1: *Método acc com muitos fluxos condicionais*

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.

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*.

4.3 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;
}
```

Listing 4.2: *Simplicidade do método acc*

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.

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

Listing 4.3: *Método que encapsula detalhes e pôde ser reutilizado*

Em contraposição, dois cenários relacionados foram observados:

- **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);
    }
  }
}
```

Listing 4.4: *Método que acumulou muitos parâmetros*

- **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;
}
```

Listing 4.5: *Método poluído pelo aumento número de parâmetros*

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.

4.4 Terceiro Estado

Como discutido na apresentação da técnica do Uso de Parâmetros como Variável de Instância (Seção 2.3.2), 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 4.6 abaixo, podemos notar a diferença na limpeza de um método beneficiado pela melhoria mencionada e cuja versão anterior é apresentada no trecho de código 4.5.

```
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();
}
```

Listing 4.6: *Método sem longas listas de parâmetros*

Novamente, as alterações tiveram consequências para a classe e um novo cenário pode ser observado:

- **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.

4.5 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.

4.5.1 Resultados finais

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.

Novamente, ressaltamos que esses valores foram calculados sobre uma linguagem com características específicas. Em Perl, os parâmetros de um métodos são coletados dentro do corpo do método, o que mostra que a média de 5 linhas seja de fato 4, uma vez que a primeira linhas sempre é didicada a extração de pelo menos um parâmetro (“‘self”).

Outro aspecto que podemos observar ao final das mudanças é que de fato minimizamos o número de parâmetros. Inicialmente, 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.

Em termos das métricas, o resumo do estudo de caso é a minimização de LOC, CYCLO, MAXNESTING, NP, LCOM4 e da diferença entre a média NRA e NOA.

Do ponto de vista conceitual, também obtivemos diversos resultados. É fácil perceber o aumento na expressividade e simplicidade do código através dos tópicos já levantados. Para compreender as tarefas dos métodos dentre uma classe, basta lê-lo, uma vez que sua implementação geralmente consiste em chamadas para outros métodos cujos nomes espelham bem suas atividades. Caso a ferramenta *Analizo Metrics* fosse implementada em uma linguagem mais apropriada para a Orientação a Objetos, os resultados poderiam ser ainda mais efetivos.

As contribuições para a flexibilidade do sistema são mais sutis. As novas classes responsáveis pelo encapsulamento do cálculo de uma métrica de módulo associada a classe *ModuleMetrics* tem grande influência sobre esse conceito. Para adicionar uma nova métrica, basta criar uma classe com a mesma interface das demais e adicioná-la na lista do método *initialize_ calculators* de *ModuleMetrics*.

Além disso, a maximização da coesão também colabora para a flexibilidade do sistema. À medida que o código vai sendo limpo, as responsabilidades começam a ficar mais claras. Ao quebrar as classes, semelhanças nas abstrações ficam evidentes. Por exemplo, classes como *AverageCycloComplexity* e *AverageMethodLinesOfCode*, poderiam ser refatoradas para implementar o padrão de projeto

Template Method (Gamma *et al.*, 1995) através de uma herança com uma nova classe.

Um dos aspectos pouco comentados ao longo desse estudo de caso foi o acoplamento entre as classes. Como trabalhamos sobre uma classe, poucas considerações foram feitas acerca desse conceito. No entanto, dois tipos interessantes puderam ser constatados.

O método *initialize_calculators* da classe *ModuleMetrics* usa muitos métodos provenientes de diversas classes (invoca o método *new* sobre todas), o que resulta em ECR = 1 (Taxa de Chamadas Externas) e NCC = 15 (Número de Classes Chamadas). Claramente o método se encaixa no cenário de Método Dispersamente Acoplado (Seção 3.2.2).

É interessante notar que essa classe foi um resultado do esforço para criar classes coesas, tendo sido criada para encapsular a responsabilidade de lidar com esse acomplamento. Algum método teria que fazê-lo, então que seja dentro de uma classe e nenhum cliente saiba dessa implementação. Dessa forma, podermos dizer que a classe *ModuleMetrics* é um Objeto Centralizador.

Outro método que pode ser destacado é *add_descriptive_statistics* da classe *GlobalMetrics*, cujo alto valor de ERC e baixo NCC igual a 1 o encaixe no cenário de Método Invejoso (Seção 3.2.2).

```
sub _add_descriptive_statistics {
  my ($self, $metric) = @_;
  my $statistics = Statistics::Descriptive::Full->new();
  $statistics->add_data(@{$self->values_lists->{$metric}});

  $self->metric_report->{$metric . "_average"} = $statistics->mean;
  $self->metric_report->{$metric . "_maximum"} = $statistics->max;
  $self->metric_report->{$metric . "_mininum"} = $statistics->min;
  $self->metric_report->{$metric . "_median"} = $statistics->median;
  $self->metric_report->{$metric . "_mode"} = $statistics->mode;
  $self->metric_report->{$metric . "_sum"} = $statistics->sum;
  $self->metric_report->{$metric . "_variance"} = $statistics->variance;
}
```

Listing 4.7: *Método Invejoso*

De fato, seria interessante que a classe *Descriptive::Statistics* tivesse um método que recebesse uma lista de valores e devolve-se um *hash* com chave sendo um nome de um estimador estatístico e com valor sendo o valor do dado estimador. Dessa forma, *add_descriptive_statistics* não precisaria fazer tantos pedidos para um mesmo objeto.

Esse tipo de cenário é comum quando trabalhamos com classes externas, já que não podemos controlar sua interface. No entanto, poderíamos criar uma classe que fizesse esse trabalho e encapsulasse a dependência.

Capítulo 5

Conclusão

Ao longo desta monografia, 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.

Considerando o desenvolvimento do estudo de caso, podemos notar que o mapeamento expressou de forma bastante fiel aspectos do código nos quais poderíamos fazer melhorias. Entre os resultados interessantes do mapeamento, podemos destacar uma similariedade na sequência da detecção dos cenários e suas respectivas melhorias, fato observado tanto no estudo como nos exemplos expostos nos demais capítulos do trabalho.

Em um primeiro momento, frequentemente detectamos a presença dos cenários Método Grande e Métodos com Muitos Fluxos Condicionais, expressos pelo LOC (Lines of Code), CYCLO (Cyclomatic Complexity) e MaxNesting (Maximum Nesting Level) dos métodos. Seguindo os conceitos de código limpo, diante desse contexto, uma melhoria possível é utilizar a *Composição de Métodos* buscando *Evitar as Estruturas Encadeadas*.

Na medida em que os métodos são decompostos, podemos nos questionar quanto ao limiar entre um método grande e um de acordo com os princípio de código limpo. Em outras palavras, a questão é como um desenvolvedor pode considerar que os resultados da *Composição de Métodos* feita até o dado instante obteve métodos pequenos o suficiente. Nos parece que a resposta de tal questão só pode ser feita através de um bom entendimento dos conceitos relacionados a um código limpo. Ao longo deste trabalho, não definimos um valor para as métricas LOC, CYCLO e MaxNesting capaz de expressar esse limiar, de forma que os métodos estejam todos realizando apenas uma tarefa.

O segundo estado recorrente durante as alterações é a detecção de *Métodos com Muitos Parâmetros* e *Muita Passagem de Parâmetros pela Classe*. Para que as tarefas possam ser divididas pelos métodos criados na *Composição de Métodos*, muitos parâmetros são necessários e frequentemente encontramos métodos que simplesmente repassam os mesmos.Para minimizar tais cenários, podemos promover alguns parâmetros para variáveis de instância.

Nesse contexto, podemos nos questionar quanto a quais parâmetros promover e qual é o limiar das métricas NRP (Número de Repasses de Parâmetros) e NP (Número de Parâmetros) em que podemos tomar a alteração por acabada. Novamente a resposta não está nas métricas, mas nos conceitos. Um parâmetros só pode ser promovido se, de fato, faz parte do estado da classe, conceito que não encontramos uma maneira de automatizar.

Seguindo os passos frequentes ao longo das alterações, a partir do momento em que novas variáveis de instância são adicionadas, frequentemente detectamos a queda na coesão da classe em questão.

Nesse momento podemos identificar mais facilmente as responsabilidades que a classe concentrava de uma maneira mais próxima da implementação. Desta forma, provavelmente existem conjuntos de métodos e variáveis que não se relacionam, causando um valor LCOM4 (Lack of Cohesion of Methods) maior do que 1, ou métodos que trabalham apenas sobre uma parcela das variáveis, o quer resulta em baixa média de NRA (Número de Atributos Alcançáveis) em relação ao número de variáveis de instância (NOA - Número de Atributos).

Sendo assim, geralmente optamos por quebrar a classe com baixa coesão. A medida para essa divisão capaz de gerar classes extremamente coesas são as responsabilidades as quais encapsulam. Sabemos que classes com uma responsabilidade geralmente possuem LCOM4 igual a 1, mas o inverso não pode ser afirmado.

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.

No entanto, notamos a seguinte contraposição: por um lado, as métricas de código-fonte e os cenários nos ajudam a aprender e aplicar os conceitos relacionados a um código limpo; por outro, se não conhecermos os conceitos e as técnicas de um código limpo, não teremos facilidade para interpretar os cenários e os valores das métricas.

Essa limitação torna difícil a automatização completa da detecção dos cenários. Em outras palavras, através deste trabalho não foi possível identificar uma maneira automática de decidir se um código é limpo ou encontrar trechos de código que definitivamente precisam de melhorias.

O que podemos afirmar é que, nos exemplos de código 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 que permitem tais valores são responsáveis pela limpeza do código.

Adicionalmente, sugerimos como trabalho futuro a identificação de outros cenários que ampliem os possíveis relacionamentos entre métricas e código limpo. Também será uma boa pesquisa, verificar os cenários descritos neste trabalho com outras métricas, preferencialmente implementadas em ferramentas de cálculo de métricas de código-fonte tais como a Analizo.

Do ponto de vista prático, outro trabalho futuro seria implementar na Analizo as métricas NRP, MaxNesting e NRA, pois dentre as métricas mapeadas com o conceitos de código limpo, são as únicas ainda não implementadas. Além disso, será uma oportunidade de um outro desenvolvedor da Analizo verificar se o código está mais simples, expressivo e flexível ao ponto de inserir essas novas funcionalidade.

Concluindo, é importante ressaltar que este é um trabalho diferenciado ao compilar e abordar de forma prática os conceitos sobre código limpo do ponto de vista de renomados especialistas em desenvolvedores de software. Adicionamos exemplos reais para que o entendimento fosse mais próximo do ponto de vista de um programador. Também, apresentamos uma forma simples e clara de utilizar métricas de código-fonte, demonstrando em cenários reais as correlações com os conceitos de código limpo que foram amplamente discutidos.

Para exemplificar a união dos conceitos trabalhados, aplicamos o mapeamento entre métricas e os conceitos de código limpo na refatoração de um software livre chamado Analizo. Essa ferramenta recebeu contribuições nossas por mais de um ano e vem sendo constantemente usada em pesquisas envolvendo os valores das métricas de código-fonte.

Portanto, além de um texto com uma nova abordagem que poderá servir de referência para outros estudantes e pesquisadores da área de qualidade e métricas de código-fonte, finalizamos este trabalho com uma contribuição efetiva para uma ferramenta de cálculo de métricas de código-fonte, de tal forma que ficou facilitada a inserção de novos contribuidores uma vez que seu código está mais limpo.

Apêndice A

Artigo Analizo SBES 2010

Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit

Antonio Terceiro[1], Joenio Costa[2], João Miranda[3], Paulo Meirelles[3], Luiz Romário Rios[1], Lucianna Almeida[3], Christina Chavez[1], Fabio Kon[3]

[1] Universidade Federal da Bahia (UFBA)

{terceiro,luizromario,flach}@dcc.ufba.br

[2]Universidade Católica do Salvador (UCSAL)

joenio@perl.org.br

[3]Universidade de São Paulo (USP)

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

Abstract. *This paper describes Analizo, a free, multi-language, extensible source code analysis and visualization toolkit. It supports the extraction and calculation of a fair number of source code metrics, generation of dependency graphs, and software evolution analysis.*

1. Introduction

Software engineers need to analyze and visualize the software they create or maintain in order to better understand it. Software Engineering researchers need to analyze software products in order to draw conclusions in their research activities. However analyzing and visualizing large individual software products or a large number of individual software products is only cost-effective with the assistance of automated tools.

Our research group have been working with empirical studies that require large-scale source code analysis, and consequently we resort to source code analysis tools in order to support some of our tasks. We have defined the following requirements for the tool support we needed:

- Multi-language. The tool should support the analysis of different programming languages (in particular, at least C, C++ and Java), since this can enhance the validity of our studies.
- Free software. The tool should be free software[1], available without restrictions, in order to promote the replicability of our studies by other researchers.
- Extensibility. The tool should provide clear interfaces for adding new types of analyzes, metrics, or output formats, in order to promote the continuous support to our studies as the research progresses.

In this paper, we present Analizo, a toolkit for source code analysis and visualization, developed with the goal of fulfilling these requirements. Section 2 describes related work. Section 3 describes Analizo architecture. Section 4 presents Analizo features. Section 5 presents Analizo use cases. Finally, Section 6 concludes the paper and discusses future work.

[1]In our work, we consider the terms "free software" and "open source software" equivalent.

2. Related work

While evaluating the existing tools to use in our research, we analyzed the following ones: CCCC [Littlefair 2010], Cscope [Steffen et al. 2009], LDX [Hassan et al. 2005], CTAGX [Hassan et al. 2005], and CPPX [Hassan et al. 2005]. Besides the research requirements described, we have included two practical requirements:

- The tool must be actively maintained. This involves having active developers who know the tool architecture and can provide updates and defect corrections.
- The tool must handle source code that cannot be compiled anymore. For example, the code may have syntax errors, the libraries it references may be not available anymore, or the used libraries changed API. This is important in order to be able to analyze legacy source code in software evolution studies.

The requirements evaluation for the tools are presented in Table 1. Since we only looked at tools that were free software, the table does not have a line for that requirement.

Requirement	**CCCC**	**Cscope**	**LDX**	**CTAGX**	**CPPX**
Language support	C++, Java	C	C, C++	C	C, C++
Extensibility	No	No	No	No	No
Maintained	Yes	Yes	No	No	No
Handles non-compiling code	Yes	No	No	No	No

Table 1. Found tools versus posed requirements

As it can be seen in Table 1, none of the existing tools we found fulfills all of our requirements. In special, none of the tools were able to analyze source code in all three needed languages, and none of them had documented extension interfaces that could be used to develop new analysis types or output formats.

3. Architecture

Analizo architecture is presented in Figure 1, using a Layered style [Clements et al. 2002]. Each layer in the diagram uses only the services provided by the layers directly below it.

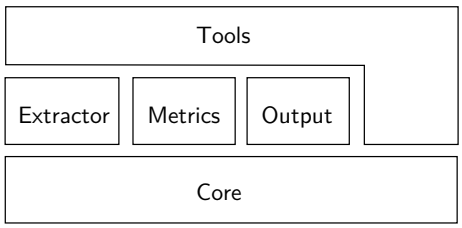


Figure 1. Analizo architecture, using the Layered Style [Clements et al. 2002]

The *Core* layer contains the data structures used to store information concerning the source code being analyzed, such as the list of existing modules[2], elements inside each module (attributes/variables, or methods/functions), dependency information (call,

[2]we used the "module" concept as a general term for the different types of structures used in software development, as classes and C source files

inheritance, etc). This layer implements most of Analizo business logic, and it does not depend on any other layer.

The *Extractors* layer comprises the different source code information extraction strategies built in Analizo. Extractors get information from source code and store them in the *Core* layer data structures. It requires only the creation of a new subclass to add a new type of extractor that interfaces with another external tool or provides its own analysis directly. Currently, there are two extractors. Both are interfaces for external source code parsing tools:

- *Analizo::Extractors::Doxyparse* is an interface for Doxyparse, a source code parser for C, C++ and Java developed by our group [Costa 2009]. Doxyparse is based on Doxygen[3], a multi-language source code documentation system that contains a robust parser.
- *Analizo::Extractors::Sloccount* is an interface for David A. Wheeler's Sloccount[4], a tool that calculates the number of effective lines of code.

The other intermediate layers are *Metrics* and *Output*. The *Metrics* layer processes *Core* data structures in order to calculate metrics. At the moment, Analizo supports a fair set of metrics (listed in Section 4). The *Output* layer is responsible for handling different file formats. Currently, the only output format implemented is the DOT format for dependency graphs, but adding new formats is simply a matter of adding new output handler classes.

The *Tools* layer comprises a set of command-line tools that constitute Analizo interface for both users and higher-level applications. These tools use services provided by the other layers: they instantiate the core data structures, one or more extractors, optionally the metrics processors, an output format module, and orchestrate them in order to provide the desired result. Most of the features described in Section 4 are implemented as Analizo tools.

Those tools are designed to adhere to the UNIX philosophy: they accomplish specialized tasks and generate output that is suitable to be fed as input to other tools, either from Analizo itself or other external tools. Some of the tools are implemented on top of others instead of explicitly manipulating Analizo internals, and some are designed to provide output for external applications such as graph drawing programs or data analysis and visualization applications.

4. Features

4.1. Multi-language source code analysis

Currently, Analizo supports source analysis of code written in C, C++ and Java. However, it can be extended to support other languages since it uses Doxyparse, which is based on Doxygen and thus also supports several different languages.

4.2. Metrics

Analizo reports both project-level metrics, which are calculated for the entire project, and module-level metrics, which are calculated individually for each module. On the

[3] doxygen.org/

[4] dwheeler.com/sloccount/

project-level, Analizo also provides basic descriptive statistics for each of the module-level metrics: sum, mean, median, mode, standard deviation, variance, skewness and kurtosis of the distribution, minimum, and maximum value. The following metrics are supported at the time of writing[5]:

- Project-level metrics: Total Coupling Factor, Total Lines of Code, Total number of methods per abstract class, Total Number of Modules/Classes, Total number of modules/classes with at least one defined attributes, Total number of modules/classes with at least one defined method, Total Number of Methods.
- Module-level metrics: Afferent Connections per Class, Average Cyclomatic Complexity per Method, Average Method LOC, Average Number of Parameters per Method, Coupling Between Objects, Depth of Inheritance Tree, Lack of Cohesion of Methods, Lines of Code, Max Method LOC, Number of Attributes, Number of Children, Number of Methods, Number of Public Attributes, Number of Public Methods, Response For a Class.

4.3. Metrics batch processing

In most quantitative studies on Software Engineering involving the acquisition of source code metrics on a large number of projects, processing each project individually is impractical, error-prone and difficult to repeat. Analizo can process multiple projects in batch and produce one comma-separated values (CSV) metrics data file for each project, as well as a summary CSV data file with project-level metrics for all projects. These data files can be easily imported in statistical tools or in spreadsheet software for further analysis. This can also be used to analyze several releases of the same project, in software evolution studies.

4.4. Metrics history

Sometimes researchers need to process the history of software projects on a more fine-grained scale. Analizo can process a version control repository and provide a CSV data file with the metrics values for each revision in which source code was changed in the project. Git and Subversion repositories are supported directly, and CVS repositories must be converted into Git ones beforehand.

4.5. Dependency Graph output

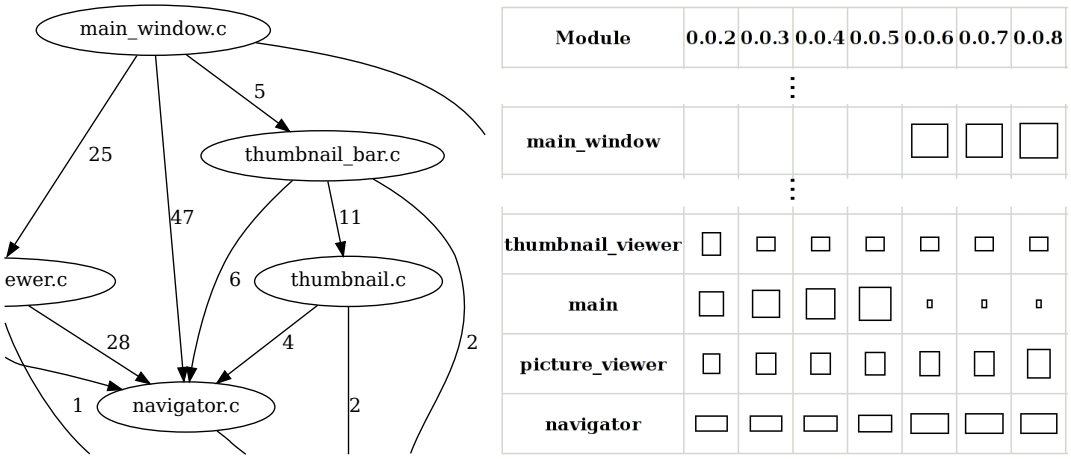
Analizo can output module dependency information extracted from a source code tree in a format suitable for processing with the Graphviz[6] graph drawing tools. Figure 2(a) presents a sample dependency graph obtained by feeding Graphviz' *dot* tool with Analizo graph output.

4.6. Evolution matrix

Another useful Analizo feature is generating evolution matrices [Lanza 2001]. After processing each release of the project (see Section 4.3), the user can request the creation of an evolution matrix from the individual data files. Figure 2(b) shows an excerpt of a sample evolution matrix produced by Analizo.

[5]References to literature on each metric were omitted because of space constraints.

[6]`graphviz.org/`



(a) Sample module dependency graph (b) Sample evolution matrix

Figure 2. Examples of Analizo features.

5. Use cases

Analizo has been validated in the context of research work performed by our group that required tool support with at least one of its features:

- [Amaral 2009] used Analizo module dependency graph output to produce an evolution matrix for a case study on the evolution of the VLC project. Later on, an evolution matrix tool was incorporated in Analizo itself.
- [Costa 2009] did a comparison between different strategies for extracting module dependency information from source code, leading to the development of Doxyparse – the Analizo Doxygen-based extractor.
- [Terceiro and Chavez 2009] used the metrics output on an exploratory study on the evolution of structural complexity in a free software project written in C.
- [Morais et al. 2009] used the Analizo metrics tool as a backend for Kalibro[7], a software metrics evaluation tool. Later on, Kalibro Web Service[8] was developed, providing an integration with Spago4Q[9] – a free platform to measure, analyze and monitor quality of products, processes and services.
- [Terceiro et al. 2010] used the metrics history processing feature to analyze the complete history of changes in 7 web server projects of varying sizes.
- [Meirelles et al. 2010] used Analizo metrics batch feature to process the source code of more than 6000 free software projects from the Sourceforge.net repository.

Most of the work cited above contributed to improvements in Analizo, making it even more appropriate for research involving source code analysis.

6. Final remarks

This paper presented Analizo, a toolkit for source code analysis and visualization that currently supports C, C++ and Java. Analizo has useful features for both researchers

[7] `softwarelivre.org/mezuro/kalibro/`
[8] `ccsl.ime.usp.br/kalibro-service`
[9] `spago4q.org`

working with source code analysis and professionals who want to analyze their source code in order to identify potential problems or possible enhancements.

Future work includes the development of a web-based platform for source code analysis and visualization based on Analizo. This project is current under development.

Analizo is free software, licensed under the GNU General Public License version 3. Its source code, as well as pre-made binary packages, manuals and tutorials can be obtained from `softwarelivre.org/mezuro/analizo`. All tools are self-documented and provide on-line manuals. Analizo is mostly written in Perl, with some of its tools written in Ruby and Shell Script.

This work is supported by CNPQ, FAPESB, the National Institute of Science and Technology for Software Engineering (INES), Qualipso project, and USP FLOSS Competence Center (CCSL-USP).

References

Amaral, V. (2009). Análise de evolucao de projetos de software livre através de matrizes de evolucao. Undergraduation course conclusion project, Universidade Federal da Bahia.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). *Documenting Software Architecture : Views and Beyond*. The SEI series in software engineering. Addison-Wesley, Boston.

Costa, J. (2009). Extração de informações de dependência entre módulos de programas c/c++. Undergraduation course conclusion project, Universidade Católica do Salvador.

Hassan, A. E., Jiang, Z. M., and Holt, R. C. (2005). Source versus object code extraction for recovering software architecture. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*.

Lanza, M. (2001). The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42, New York, NY, USA. ACM.

Littlefair, T. (2010). CCCC - C and C++ Code Counter. Available at `http://cccc.sourceforge.net/`. Last access on June 3rd, 2010.

Meirelles, P., Jr., C. S., Miranda, J., Kon, F., Terceiro, A., and Chavez, C. (2010). A Study of the Relationship between Source Code Metrics and Attractiveness in Free Software Projects. *Submitted*.

Morais, C., Meirelles, P., and Kon, F. (2009). Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte. Undergraduation course conclusion project, Universidade de São Paulo.

Steffen, J., Hans-Bernhard, and Horman, B. N. (2009). *Cscope*. http://cscope.sourceforge.net/.

Terceiro, A. and Chavez, C. (2009). Structural Complexity Evolution in Free Software Projects: A Case Study. In Ali Babar, M., Lundell, B., and van der Linden, F., editors, *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*.

Terceiro, A., Rios, L. R., and Chavez, C. (2010). An Empirical Study on the Structural Complexity introduced by Core and Peripheral Developers in Free Software projects. *Submitted*.

Referências Bibliográficas

AgM(2001) *Manifesto for Agile Software Development.* Disponível em http://agilemanifesto.org. Citado na pág. 3

Cpp(1989-2002) *The C++ Report Magazine.* SIGS Publications Group. Citado na pág. 25

Inf(2008) *Inteview: Kent Beck on Implementation Patterns.* URL http://www.infoq.com/interviews/beck-implementation-patterns. Citado na pág. 13

Beck(1999) Kent Beck. *Extreme Programming Explained.* Addison Wesley. Citado na pág. 7

Beck(2007) Kent Beck. *Implementation Pattens.* Addison Wesley. Citado na pág. 3, 4, 5, 7, 11, 13, 21, 25, 33

Beck e Cunningham(1989) Kent Beck e Ward Cunningham. A laboratory for teaching object-oriented thinking. Em *Object-Oriented Programming, Systems, Languages and Applications Conference Proceedings.* Citado na pág. 19

Booch(2007) Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison Wesley. Citado na pág. 7

Fowler *et al.***(1999)** Martin Fowler, Kent Beck, John Brant, William Opdyke, e Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley. Citado na pág. 23

Gamma *et al.***(1995)** Erick Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley. Citado na pág. 14, 56

Hitz e Montazeri(1996) M. Hitz e B. Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. Em *IEEE Transactions on Software Engineering.* Citado na pág. 37

Knuth(1992) Donald Knuth. *Literate Programming.* Citado na pág. 7

Lanza e Marinescu(2006) Michele Lanza e Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems.* Springer. Citado na pág. 4, 35, 36

Lorenz e Kidd(1994) Mark Lorenz e Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall. Citado na pág. 37

Marinescu(2002) Radu Marinescu. Measurement and quality in object-oriented design. Citado na pág. 35

Martin(2000) Robert C. Martin. Principles and patterns. Citado na pág. 8

Martin(2002) Robert C. Martin. *Agile Software Development: Principles, Patterns and Practices.* Prentice-Hall. Citado na pág. 19

Martin(2008) Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship.* Prentice Hall. Citado na pág. 3, 4, 5, 7, 11, 12, 33

Martin(1997) Robert C. Martin. The dependency inversion principle. Citado na pág. 25

McCabe(1976) T.J. McCabe. A measure of complexity. Em *IEEE Transactions on Software Engineering*. Citado na pág. 36

Meirelles *et al.***(2010)** Paulo Meirelles, Carlos Santos Jr., João Miranda, Fabio Kon, Antonio Terceiro, e Christina Chavez. A Study of the Relationship between Source Code Metrics and Attractiveness in Free Software Projects, 2010. *Submitted*. Citado na pág. 49

Meirelles e Kon(2009) Paulo R. M. Meirelles e Fabio Kon. Mangue: Metrics and Tools for Automatic Quality Evaluation of Source Code. Em *VIII SBQS - Simpósio Brasileiro De Qualidade De Software (VII Workshop De Teses E Dissertacões Em Qualidade De Software (WTDQS))*. Citado na pág. 33, 49

Meirelles *et al.***(2009)** Paulo R. M. Meirelles, Raphael Cóbe, Simone Hanazumi, Paulo Nunes, Geiser Challco, Straus Martins, Eduardo Morais, e Fabio Kon. Crab: Uma Ferramenta de Configuração e Interpretação de Métricas de Software para Avaliação da Qualidade de Código. Em *XXIII SBES - Simpósio Brasileiro de Engenharia de Software (XVI Sessão de Ferramentas)*. Citado na pág. 33

Morais *et al.***(2009)** Carlos Morais, Paulo Meirelles, e Fabio Kon. Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte. Undergraduation course conclusion project, Universidade de São Paulo. URL http://www.ime.usp.br/~cef/mac499-09/monografias/carlos-morais/. Citado na pág. 49

Rebecca Wirfs-Brock(2003) Alan McKean Rebecca Wirfs-Brock. *Object Design: Roles, Responsibilities, and Collaborations*. Addison Wesley. Citado na pág. 19

Terceiro *et al.***(2010)** Antonio Terceiro, Joenio Costa, João Miranda, Paulo Meirelles, Luiz Romário Rios, Lucianna Almeida, Christina Chavez, e Fabio Kon. Analizo: an extensible multi-language source code analysis and visualization toolkit. Citado na pág. 4, 49

Terceiro *et al.***(2010)** Antonio Terceiro, Luiz Romário Rios, e Christina Chavez. An Empirical Study on the Structural Complexity introduced by Core and Peripheral Developers in Free Software projects, 2010. *Submitted*. Citado na pág. 49