Código Limpo e seu Mapeamento em Métricas de Código-Fonte

Alunos: Lucianna Thomaz Almeida
João Machini de Miranda
Orientador: Fabio Kon
Coorientador: Paulo Meirelles

IME-USP
CCSL CENTRO DE COMPETÊNCIA EM SOFTWARE LIVRE
FLOSS Competence Center
Qualipso Quality Plataform for Open Source Software

Objetivos

- Compreender o que é um Código Limpo através de um conjunto de boas decisões que possam ser adotadas ao longo do desenvolvimento para auxiliar a criação de um código mais expressivo, simples e flexível.
- Identificar um mapeamento entre um conjunto de Métricas de Código-Fonte e os principais conceitos relacionados a um Código Limpo.
- Encontrar uma maneira de interpretar os valores das métricas, de forma a facilitar a detecção de trechos de código que poderiam receber melhorias.

Código Limpo

Através de uma pesquisa sobre duas referências principais - "*Implementation Patterns*" de Kent Beck e "*Clean Code*" de Robert Martin - definimos um Código Limpo como aquele que mais se aproxima dos seguintes valores:

- Expressividade
  Facilidade em que um desenvolvedor, que não o autor original do trecho de código, o entende, modifica e utiliza.
- Simplicidade
  Relacionada a quantidade de informações que o leitor deve compreender para fazer alterações em um trecho de código.
- Flexibilidade
  Reflete a facilidade com que estendemos o código da aplicação sem grandes alterações na estrutura já implementada.

Mapeamento

- Fizemos um mapeamento dos conceitos de código limpo em métricas de código-fonte. Essas métricas são mecanismos que nos permitem encontrar características específicas do código.
- Interpretamos os valores das métricas, os associando com as técnicas e conceitos relacionados à limpeza do código.
- O objetivo é facilitar a detecção de trechos que poderiam sofrer alterações que os tornem mais expressivos, simples e flexíveis.

Código Limpo

- Composição de Métodos
  Compor os métodos em chamadas para outros no nível de abstração abaixo e com nomes explicativos.
  - Os métodos ao lado são bastante curtos e responsáveis por apenas uma atividade descrita pelo seu nome.
- Métodos Explicativos
  Criar um método que encapsule uma operação pouco clara geralmente associada a um comentário.
  - Teremos um novo método com um nome que documenta uma operação como no caso de *inserePrimeiraArestaNula*.
- Métodos como Condicionais
  Criar um método que encapsule uma expressão booleana para obter condicionais mais claros.
  - Teremos um novo método com um nome que documenta uma resultado booleano como *verticeNuncaAtingido*, o que deixa o condicional do cliente mais expressivo.
- Evitar Estruturas Encadeadas
  Utilizar a Composição de Métodos para minimizar a quantidade de estruturas encadeadas em cada método.
  - Cada método terá uma estrutura de condicionais simples, o que aumenta a legibilidade e facilita a detecção de erros.
- Parâmetros como Variável de Instância
  Localizar parâmetros muito utilizados pelos métodos de uma classe e transformá-los em variáveis de instância.
  - Não haverá a necessidade de passar longas listas de parâmetros através da classe.
- Maximizar a Coesão
  Localizar métodos e atributos que formam uma abstração dentro de uma classe e quebrar a classe em duas, cada uma com uma responsabilidade segundo o Princípio da Responsabilidade Única.
  - Classes com grande proximidade entre lógica e dados, sem intereferências entre as responsabilidades.

Exemplo de Código

```
class Digrafo:
  instanceVariables numeroDeVertices, listaDeAdjacencia

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

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

  def arestasDoVertice(vertice):
    return listaDeAdjacencia[vertice]

class CalculadorDeCaminhosMinimos:
  instanceVariables custo, fila, verticeOrigem
  instanceVariables digrafo, numeroDeVertices
  constant Infinito = -1

  def construtor(umDigrafo):
    numeroDeVertices = umDigrafo.numeroDeVertices
    digrafo = o_digrafo

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

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

  def inicializaCustos():
    custos = nova Lista(numeroDeVertices)
    setaTodosOsVerticesComoNaoAtingidos()
    setaVerticeOrigemComoAtingidoSemCusto()

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

def setaVerticeOrigemComoAtingidoSemCusto():
  custos[verticeOrigem] = 0

def inicializaFila():
  fila = nova FilaDePrioridades(numeroDeVertices)
  inserePrimeiraArestaNula()

def inserePrimeiraArestaNula():
  fila.insere(nova Aresta(0,0))

def aindaHaVertices(fila):
  return fila.vazia()

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

def atualizaCustoSeCaminhoMaisBarato(verticeMinimo, aresta):
  verticeDestino = aresta.verticeDestino()
  if(verticeNuncaAtingido(verticeDestino)):
    atualizaCustoEInsereNaFila(verticeDestino, aresta)
  if(caminhoComNovaArestaMaisBarato(verticeDestino, aresta)):
    atualizaCustoComNovaAresta(verticeDestino, aresta)

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

def atualizaCustoEInsereNaFila(verticeDestino, aresta):
  atualizaCustoComNovaAresta(verticeDestino, aresta)
  fila.insere(nova Aresta(verticeDestino, custos[verticeDestino]))

def caminhoComNovaArestaMaisBarato(verticeDestino, aresta):
  novoCusto = custoComNovaAresta(aresta)
  return custos[verticeDestino] > novoCusto

def atualizaCustoComNovaAresta(verticeDestino, aresta):
  novoCusto = custoComNovaAresta(aresta)
  custos[verticeDestino] = custos[verticeMinimo] + novoCusto

def custoComNovaAresta(aresta):
  return custos[aresta.verticeDestino()] + aresta.custo()
```

Métricas de Código-Fonte

- Evitando Métodos Grandes
  - ↑ Grande Número de Linhas Efetivas (alto LOC)
  - ↑ Muitas Estruturas Encadeadas (alto MaxNesting)
  - ↑ Muitos Fluxos Condicionais (alta CYCLO)
  - Como esperado, os métodos desse exemplo ficaram com poucas linhas de código, o que leva a uma baixa média do número de linhas por método. Além disso, encontramos poucos fluxos condicionais em cada método e quase não existem estruturas encadeadas.
- Evitando Métodos com Muitos Parâmetros
  - ↑ Grande Número de Parâmetros (alto NP)
  - Todos os métodos desse código ficaram com poucos parâmetros. O método *adicionaAresta* é o que possui o maior número de parâmetros, sendo que ele possui apenas 3 deles.
- Evitando Passagem de Parâmetros pela Classe
  - ↑ Muitos Parâmetros Repassados pela Classe (alto NRP)
  - ↑ Grande Número de Parâmetros (alto NP)
  - São pequenas a média de parâmetros recebidos e repassados dentro de um mesmo método e a quantidade de parâmetros de cada método. Podemos perceber que variáveis acessadas por vários métodos são usadas como atributos e conjuntos de informações foram agrupados em objetos.
- Evitando Classes Pouco Coesas
  - ↑ Muitas Subdivisões de Grupos de Métodos e Atributos que Não se Relacionam (LCOM4 > 1)
  - ↑ Métodos usam em média poucos atributos da classe (média de NRA <<< NOA)
  - As classes desse exemplo são bastante coesas. Usamos dois indicadores para chegar a essa conclusão. O primeiro é que essas classes não possuem subdivisões em grupos de métodos e atributos que não se relacionam. O segundo é que os métodos delas usam muito os seus atributos. Como essas classes possuem alta coesão, sabemos que elas não precisam ser quebradas em mais classes.

Principais Referências:
BECK, Kent. *Implementation Patterns.* Addison Wesley, 2007;
MARTIN, Robert C. Clean Code, Prentice Hall, 2008;
LANZA, Michele; MARINESCU, Radu. *Object Oriented Metrics in Practice.* Springer, 2006.