Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Período 2005.1

Um modelo para avaliação da manutenibilidade de código-fonte orientado a objeto

TRABALHO DE GRADUAÇÃO EM
ENGENHARIA DE SOFTWARE

Aluno: Thiago Bispo Arrais de Souza (tbas@cin.ufpe.br)
Orientador: Paulo Henrique Monteiro Borba (phmb@cin.ufpe.br)

Resumo

Uma forma de minimizar os custos com manutenção *após* o desenvolvimento é controlar a manutenibilidade *durante* o desenvolvimento, através da utilização de métricas integradas ao processo de desenvolvimento.

Este trabalho apresenta o relatório de um experimento realizado sobre quatro sistemas de software distintos com objetivo de medir a sua facilidade de compreensão. São apontadas também as primeiras conclusões sobre o experimento, constituindo o início de um modelo de avaliação de manutenibilidade de código-fonte.

Abstract

Using metrics during the development of a software system is a way to control the maintenance costs after the project closure.

This work presents a report from a experiment over four distinct software systems that hoped to measure their ease of comprehesion. Some conclusions are pointed based on that experiment, composing the beggining of the work for a source code maintanability evaluation model.

Agradecimentos

Este trabalho representa o ápice de um período de trabalho e dedicação ao estudo e à pesquisa. Representa, principalmente, a conclusão de uma fase importante da vida, pautada pelo constante aprendizado e respeito.

Agradeço à meus pais, Maria Edileuza e Jânio Arrais de Souza por terem me dado tantas oportunidades de aprendizado, Por muitas vezes terem me apresentado novos desafios e por sempre me apoiarem a concluí-los.

Agradeço à toda minha família pelo apoio incondicional não só durante minha graduação, mas por toda a vida até agora.

Agradeço a todos os professores do Centro de Informática da Universidade Federal de Pernambuco, por terem me guiado tão magistralmente durante meu progresso na graduação. Agradeço em especial a meu orientador, professor Paulo Borba, e aos professores André Santos, Hermano Perrelli e Geber Ramalho.

Agradeço aos colegas da graduação, muitos dos quais são agora, além de colegas de trabalho, meus amigos, pelos momentos de descontração, pela ajuda nos estudos e infindáveis projetos.e até pelos desentendimentos, sempre muito saudáveis e construtivos.

A todos estes, deixo meu agradecimento por terem ajudado a construir o que sou hoje.

Conteúdo

Índice de figuras

Índice de quadros

Índice de tabelas

Introdução

"A improvisação é a ordem natural da guerra"

– Samuel Marshall

Manutenção de software é, pela definição do IEEE, 'o processo de modificar um sistema de software ou componente após a entrega para corrigir falhas, melhorar a performance ou outros atributos, ou adaptar (o sistema) a mudanças de ambiente'[12]. Os custos de manutenção de software são parte significativa dos projetos de software e têm se tornado uma preocupação constante na indústria. Em 1975, Brooks indicou que os custos com manutenção de um programa de largo uso são de 40 por cento ou mais que o custo do seu desenvolvimento[3]. Estudos mais recentes indicam que atualmente este valor varia entre 50 e 70 por cento[23] e que a manutenção de sistemas já existentes tem mais importância para a gerência que o desenvolvimento de novos[13].

Uma forma de minimizar os custos com manutenção *após* o desenvolvimento é controlar a manutenibilidade *durante* o desenvolvimento. No entanto, é largamente aceita a idéia de que 'não se pode controlar o que não se mede'[7]. E a forma usual de medir características específicas de um projeto de software é a utilização de métricas integradas ao processo de desenvolvimento. Elas suprem a gerência, os arquitetos e programadores com informação sobre o estado atual do sistema e a performance do processo, permitindo o controle desejado. Portanto, uma abordagem plausível é utilizar métricas para avaliar a manutenibilidade do software de forma que se possa acompanhar e tomar medidas de modo a minimizar a deterioração natural[19] da qualidade do software durante o desenvolvimento.

Alguns modelos já foram propostos para medição da manutenibilidade de software [6][18] e um bom número leva em consideração fatores concernentes a todas as disciplinas envolvidas na engenharia de software como qualidade da especificação de requisitos, rastreabilidade entre os artefatos de desenvolvimento e complexidade do

sistema. Este último é um dos fatores de maior importância, visto que a maior parte do tempo de manutenção é gasto com a compreensão do sistema[10].

Nossa hipótese é que a complexidade do sistema pode ser bem medida a partir de seu código fonte. A partir dessa hipótese e do exposto acima, podemos concluir que, por ser a complexidade do código fonte um bom fator de aproximação para a complexidade do sistema, a análise e medição do código fonte é fator de grande importância para a determinação da manutenibilidade.

Outro fato que vem a reforçar nossa abordagem é que o código fonte é um dos únicos artefatos presentes em todos os produtos de software, sejam eles desenvolvidos seguindo um processo organizado e bem definido ou com uma abordagem totalmente ad-hoc.

Utilizando métricas de código para aferir atributos de software

"Meça o que é mensurável, e torne mensurável o que não é"

– Galileo Galilei

Uma métrica é uma simplificação numérica de um objeto sobre um certo aspecto, de modo a permitir a comparação com outros objetos. A simplicidade das métricas é ao mesmo tempo sua maior força e sua maior fraqueza. Por um lado, permitem a comparação de objetos arbitrariamente distintos. Por outro, são apenas uma projeção dos atributos do objeto medido, podendo levar olhos desavisados a conclusões precipitadas, pois ignoram aspectos possivelmente importantes.

A utilização de métricas para prever atributos desconhecidos de um objeto a partir de outros já conhecidos exige uma habilidade e senso crítico aguçados. Este capítulo enumera, utilizando exemplos, alguns aspectos significativos para o sucesso de uma análise baseada em métricas.

Os trechos de código dos Quadros 1, 2, 3 e 4 serão utilizados no restante do capítulo. Os dois primeiros apresentam a definição de uma classe chamada Conta em Java e Ruby, respectivamente. Os dois seguintes mostram a implementação do conhecido algoritmo QuickSort[11] de ordenação em Ruby e Haskell, respectivamente.

```
class Conta {
    private int saldo;
    public int getSaldo() {
        return saldo;
    }
    protected void setSaldo(int s) {
        saldo = s;
    }
    public boolean transferir(int valor, Conta creditada) {
        if (getSaldo() >= valor) {
            setSaldo(getSaldo() - valor);
            creditada.setSaldo( creditada.getSaldo() + valor);
            return true;
        }
        return false;
    }
}
```

Quadro 1 – Classe Conta em Java

```
class Conta
    attr_accessor :saldo
    protected :saldo=
    public
        def transferir(valor, creditada)
            if (saldo >= valor)
                self.saldo -= valor
                creditada.saldo += valor
                true
            end
        end
End
```

Quadro 2 – Classe Conta em Ruby

```
def qsort(xs)
    if (xs.length <= 1)
        xs
    else
        pivot = xs.delete_at(0)
        left  = qsort(xs.select { |x| x < pivot })
        right = qsort(xs.select { |x| x > pivot })
        left + [pivot] + right
    end
End
```

Quadro 3 – Algoritmo QuickSort em Ruby

```
qsort [] = []
qsort [x] = [x]
qsort (x:xs) = qsort([y | y <- xs, y < x]) ++ [x] ++ qsort([z| z <- xs, z > x])
```

Quadro 4 – Algoritmo QuickSort em Haskell

A escolha das métricas que serão utilizadas na comparação entre os sistemas deve ser bem estudada e definida. Elas devem ser *comparáveis*, serem aplicadas a um *conjunto de dados significativo* e possuírem um *modelo de interpretação* adequado às necessidades. Estas três características são detalhadas no decorrer deste capítulo, juntamente com algumas considerações sobre como alcançá-las.

Comparabilidade

Os trechos de código mostrados no Quadro 1 e no Quadro 2 são semanticamente equivalentes, porém foram escritos utilizando linguagens diferentes. A definição da classe em Ruby precisa de 4 linhas a menos (23% menos linhas de código) que em Java,

porém não poderemos comparar diretamente a facilidade de compreensão para estes dois trechos de código com base somente nesta métrica.

Os valores da métrica de tamanho de código não são comparáveis por que os resultados desta métrica dependem, entre outros fatores, da linguagem utilizada para implementação, um aspecto que desejamos ignorar para esta análise. Se quisermos utilizar uma métrica para a comparar a complexidade dos dois trechos de código, poderemos usar a complexidade ciclomática[18], uma métrica razoavelmente independente de linguagem.

Para que os resultados obtidos sejam comparáveis, a métrica utilizada deve ser precisa e completamente definida e os objetos devem ser comparados em um ambiente controlado.

Conjunto de dados

Os quadros 1 a 4 mostram trechos de código equivalentes escritos em linguagens de programação diferentes. Se tentarmos generalizar os resultados obtidos para estes trechos para prever o tamanho de uma base de código escrita em uma linguagem em relação à outra, estaremos correndo o risco de cometer uma incoerência estatística. O conjunto de dados que analisamos simplesmente não é significativo o suficiente para nos permitir tais conclusões. Não podemos, a partir deles propor que um programa em Ruby terá aproximadamente 23% menos linhas que o equivalente em Java, nem que qualquer programa escrito em Haskell será 70% do que se fosse escrito em Ruby.

Podemos citar como segundo exemplo a seleção de código para avaliar a repetição de código em um sistema no qual foi utilizada geração automática de código. A utilização de um gerador de código está intimamente ligada à necessidade inerente ao sistema de possuir código repetido em certas partes específicas, portanto a repetição de código nelas não representa desvio dos padrões. Independentemente das métricas que utilizarmos para detectar esta repetição, deveremos excluir do conjunto de dados estas partes específicas para evitar distorcer nossos resultados.

Conjuntos de dados devem ser suficientemente dispersos para permitir um índice de cobertura razoável para o contexto em questão.

Interpretação

Para que, na seção anterior, fôssemos capazes de rejeitar a métrica de número de linhas de código como forma de comparar duas linguagens de programação, precisamos de um modelo de interpretação informal para a métrica. Sentenças como 'o conjunto de dados que analisamos simplesmente não é significativo o suficiente' fazem parte deste modelo, que intuitivamente nos guiou aos resultados.

A função do modelo de interpretação é justificar a utilização da métrica em um certo *contexto*, de modo a esclarecer sua relevância e aplicabilidade. Um bom exemplo de modelo de interpretação é dado por Robert Martin[16] para sua métrica de instabilidade. Apesar de a métrica não ser trivial, o modelo de interpretação é consistente e justifica sua aplicabilidade.

O modelo pode ser visto como uma forma de relacionar as métricas com os princípios e práticas do bom *design*[15]. Uma abordagem inicial ingênua seria definir intervalos padrão para cada métrica. Esta abordagem é capaz de fornecer *rules-of-thumb* iniciais, mas é limitada por três aspectos principais:

i. *Não permite comparação entre sistemas*, por ser uma abordagem que gera valores discretos (i.e. conforme ou não conforme) ao invés de valores em um intervalo contínuo.
ii. *Desconsidera a natureza do sistema avaliado*, por trabalhar com valores padrão pré-fixados. A faixa de valores aceitáveis deve variar de acordo com o domínio.
iii. *Ignora a relação entre as métricas*, ao considerar as faixas de valores padrão por métrica. Uma abordagem mais robusta seria capaz de produzir métricas a partir da composição de outras métricas, como esquematizado na **Erro! A origem da referência não foi encontrada.**.

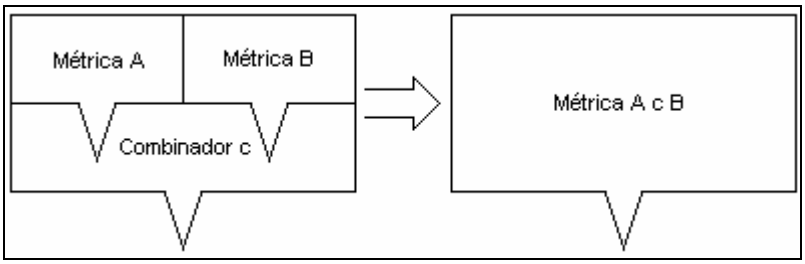


Figura 1 - Combinando métricas

A definição de padrões não é uma tarefa trivial, pois não há consenso sobre quais são as melhores práticas em design de software. Uma prática adequada em um domínio de aplicação pode ser inadequada em outros. Porém, os números coletados para um sistema mostram-se relevantes apenas quando comparadas a números de outros sistemas ou a padrões[22], o que nos sugere que o modelo de avaliação não seja baseado puramente em "valores padrão" para as métricas, mas que seja capaz de permitir a comparação entre sistemas.

O processo de manutenção

"Conhece-te a ti mesmo."

- Sócrates

As atividades de manutenção de software são atividades de modificação de um sistema pré-existente. Elas diferem fundamentalmente na origem da necessidade de mudança. Assim, as atividades de manutenção podem ser funcionalmente dividida em três categorias[1]:

i. Manutenção corretiva: Visa consertar defeitos de um sistema para que fique em conformidade com os requisitos sobre os quais foi desenvolvido. Estes defeitos dizem respeito a quaisquer problemas com o hardware, software ou documentação.
ii. Manutenção adaptativa: Adapta o sistema a mudanças nas necessidades do usuário ou do ambiente. As atividades de manutenção adaptativa têm origem em uma solicitação de melhoramento. Estão incluídas aí atividades como adição de funcionalidades, mudanças no formato dos dados de entrada e adaptação a novas regras de negócio do usuário.
iii. Manutenção perfectiva: Modifica o sistema de modo a aumentar a qualidade do software ou de sua documentação, sem modificar sua funcionalidade. Nesta categoria estão incluídos os esforços para aumentar a legibilidade do software, para melhorar sua performance, para incrementar a reusabilidade, entre outros.

Podemos, então, resumir sistematicamente as categorias de manutenção na Tabela 1.

Categoria de manutenção	Origem da necessidade de mudança
Corretiva	Requisitos do sistema
Adaptativa	Usuário ou ambiente
Perfectiva	Equipe de manutenção

Tabela 1 - Origem das atividades de manutenção

Arthur[1] identifica em sua obra as atividades necessárias a cada um dos tipos de manutenção após a categorização de uma solicitação de mudança. Abaixo estão resumidas estas atividades:

Manutenção corretiva:

i. Elaborar hipóteses sobre a causa do defeito;
ii. Testar as hipóteses para encontrar a causa exata do defeito; e
iii. Reparar o defeito encontrado.

Manutenção adaptativa:

i. Identificar partes da arquitetura envolvidas;
ii. Elaborar alternativas;
iii. Avaliar as alternativas; e
iv. Implementar a alternativa selecionada.

Manutenção perfectiva:

i. Identificar aspectos de qualidade candidatos a melhoria; e
ii. Tratar os candidatos identificados como defeitos e utilizar a técnica para manutenção corretiva.

Habilidades de manutenção

A partir desta relação de atividades, podemos identificar três habilidades do mantenedor que influenciam sua produtividade:

i. Compreensão do código: Influencia na produtividade das atividades Elaborar hipóteses sobre a causa do defeito e Identificar partes da arquitetura envolvidas.
ii. Elaboração de soluções: Influencia as atividades *Reparar o defeito encontrado* e *Elaborar alternativas.*
iii. Modificação do código: Influencia as atividades *Reparar o defeito encontrado*, *Avaliar as alternativas* e *Implementar a alternativa selecionada*.

A diferença entre as duas últimas é que a primeira foca apenas em desenvolver soluções aplicáveis ao software em manutenção, sem, no entanto, considerar as conseqüências práticas disto. Estes detalhes serão desvendados somente em um estágio posterior, onde a habilidade de maior peso será a de modificação do código.

Podemos ilustrar esta diferença com um exemplo prático:

Em um sistema de controle de folha de pagamento os funcionários são agrupados em cargos e os salários são definidos pelo cargo. A totalização do valor a ser pago aos funcionários e como imposto é feita somando-se os valores a serem pagos a cada funcionário. A este sub-total deve ser aplicado um fator multiplicativo simples para cálculo do valor de imposto.

A implementação da funcionalidade de totalização foi feita a partir das classes `Funcionario`, `CadastroFuncionarios` e `TabelaCargoSalario`. A classe `Funcionario` possui alguns dados, entre eles o número que indica o cargo. A classe `CadastroFuncionarios` dá acesso a todos os funcionários cadastrados no sistema e é responsável pela totalização dos valores da folha de pagamento. Isto é realizado através de três métodos: `calcularValorTotalFuncionarios`, `calcularValorTotalImposto` e `calcularValorTotal`. O primeiro método consulta o cargo de cada um dos funcionários, obtém da classe `TabelaCargoSalario` o valor do salário para aquele cargo e soma os valores para todos os funcionários. O método para cálculo do valor de imposto faz uso do primeiro método, multiplicando o valor pelo fator multiplicativo de imposto. O método `calcularValorTotal` calcula o valor do custo total simplesmente somando os valores retornados pelos métodos anteriores.

O mantenedor do sistema recebeu uma solicitação de modificação para que o sistema possibilite a atribuição de um adicional não-taxado por funcionário.

Uma das soluções sugeridas pela equipe de análise e projeto foi inserir um atributo adicional na classe `Funcionario` e outro método para cálculo do valor total não-taxado na classe `CadastroFuncionarios`. Adicionalmente, não deve ser modificada a assinatura nem a semântica de nenhum método já existente, pois isto exigiria altos

custos para modificação da interface gráfica.

Esta solução pôde ser facilmente elaborada, mas exige modificações em pelo menos três locais diferentes: criação do novo atributo na classe `Funcionario`, alteração do método `calcularValorTotalFuncionarios` para levar em consideração o novo atributo e alteração do método de cálculo de valor de imposto para não levar em consideração este valor. Além disso, para evitar duplicação de código, deve ser criado outro método, que tenha a mesma semântica do `calcularValorTotalFuncionarios` antigo e seja preferencialmente utilizado nos outros dois métodos de totalização.

O principal problema com este sistema é o alto grau de acoplamento entre os métodos. A modificação do método para cálculo do valor total devido aos funcionários exigiu a modificação do método de totalização do imposto, mesmo que a interface do método não tenha mudado.

O alto grau de acoplamento não diminuiu significativamente a capacidade de elaboração de soluções do mantenedor, mas o código é dependente da implementação, prejudicando a capacidade de modificá-lo.

Quadro 5 - Diferentes habilidades de manutenção

O desenvolvimento das três habilidades depende tanto das capacidades do mantenedor quanto da qualidade do código-fonte. O modelo desenvolvido neste trabalho baseia-se na influência do código-fonte sobre estas três habilidades.

Estudo de caso

"Uma vida não questionada não merece ser vivida"

– Platão

O estudo de caso que apresentaremos tem como objetivo aferir a facilidade de compreensão de quatro sistemas, uma das habilidades necessárias à manutenção. A partir dos resultados aqui coletados, será possível identificar a relação de algumas métricas de código com a facilidade de compreensão, como faremos no capítulo seguinte. Compreendemos que poderiam ser utilizadas métricas adicionais às selecionadas aqui e que algumas delas poderiam ser substituídas por outras mais apropriadas. O objetivo deste estudo de caso é tão somente instanciar experimentalmente o modelo de avaliação para avaliar sua aplicabilidade.

O método que adotamos baseia-se na produção de testes e refatoração de parte do código. Nossa hipótese inicial é que para gerar casos de teste satisfatórios é necessário primeiramente compreender o código. Nas quatro seções seguintes, discutimos e detalhamos este método.

Desenvolvimento de casos de teste e refactoring são técnicas conhecidas para desenvolvimento e melhoria de código [2][9]. Neste trabalho, utilizamos estas técnicas como forma de compreender o código. Como é gerado código fonte como resultado, elas nos permitem obter métricas concretas sobre a facilidade de compreensão do código, como o número de casos de teste gerado por unidade de tempo ou quantidade de código reestruturado. Estas métricas são identificadas na seção "Resultados observados" deste capítulo.

Metodologia de testes

A metologia de desenvolvimento de testes que adotamos é apoiada no trabalho de Beck[2], onde é definida uma técnica chamada de Test Driven Development (abreviada daqui em diante como TDD). Beck defende que esta técnica 'lhe dá a chance de aprender todas as lições que o código tem para lhe ensinar'. Nossa metodologia certamente não é TDD (ao passo que não visa o desenvolvimento de software), porém apóia-se nesta idéia de que os testes têm a ensinar ao programador. Na verdade, podemos dizer que ao escrever os testes, o programador está registrando o conhecimento que ele adquiriu sobre o sistema em uma linguagem executável por computador. Desta forma, ele pode checar constantemente se suas interpretações sobre o comportamento do código estão corretas. Esta checagem constante e automatizada é uma das bases do desenvolvimento ágil[17].

Tomando esta técnica como base, desenvolvemos um processo de desenvolvimento de testes focado na compreensão do código fonte. Este processo está representado esquematicamente pela Figura 2.

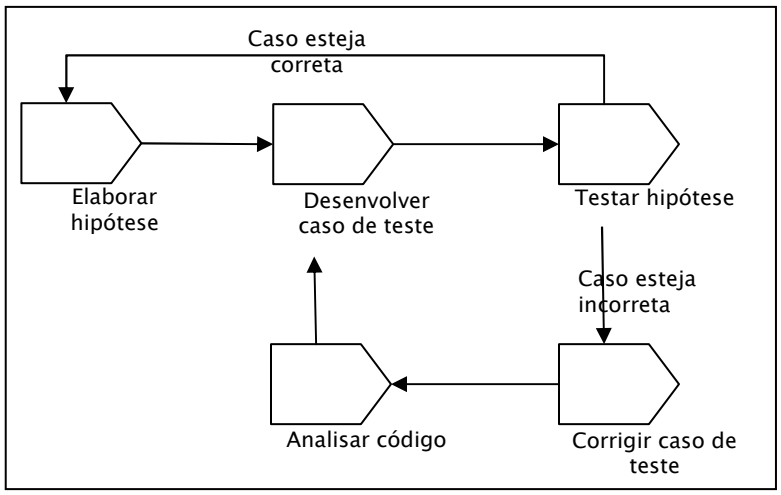


Figura 2 - Metodologia de desenvolvimento de testes

Primeiramente, deve-se elaborar uma hipótese sobre um trecho de código fonte. Esta hipótese deve identificar o comportamento esperado do sistema, em termos de entradas e saídas. O próximo passo é escrever a hipótese em forma de caso de teste

executável. Após isso, testa-se a hipótese através da execução do caso de teste. Se o sistema responder com as saídas esperadas, a hipótese estava correta: o caso de teste deve ser registrado e deve-se elaborar uma nova hipótese para reiniciar o processo. Caso contrário, a hipótese estava equivocada. Neste caso, devemos executar o código do sistema de modo a identificar qual é a saída esperada para a entrada fornecida e corrigir o caso de teste que desenvolvemos para que reflita a saída correta. Após esta correção do caso de teste, analisamos o código para determinar a causa da falha do teste.

Aplicação do processo de desenvolvimento de testes

Para demonstrar o processo de desenvolvimento de testes, iremos aplicá-lo ao trecho de código identificado no Quadro 6.

```
public class C {
    public static int m(int x) {
        if (x > 0) {
            int f = x > 0 ? x : 1;
            while(x > 1) {
                f *= --x;
            }
            return f;
        } else {
            int y = x;
            int r = x;
            ++x;
            while(x < 0) {
                r *= y;
                ++x;
            }
            return r;
        }
    }
}
```

Quadro 6 – Código a ser testado

Como primeira hipótese, podemos considerar que para uma entrada x = 0, a condição do operador ternário da linha 4 será avaliada como falsa, atribuindo um valor 1 à variável f. Na linha seguinte, a condição também será falsa, portanto o código interno ao *while* não será executado, retornando o valor f = 1.

Ao traduzir esta hipótese em código de teste, obteremos algo parecido com o código do Quadro 7.

```
public class CTest {
    public void testInput0() {
        assertEquals(1, C.m(0));
    }
}
```

Quadro 7 – Teste da primeira hipótese

Porém, ao executarmos este caso de teste, percebemos que nos enganamos na elaboração da hipótese. A resposta fornecida pelo sistema é 0, e não 1. Portanto, precisamos corrigir nosso caso de teste:

```
public class CTest {
    public void testInput0() {
        assertEquals(0, C.m(0));
    }
}
```

Quadro 8 – Teste corrigido

Com esta correção, podemos aprender algo que havíamos ignorado sobre o método m. Ao analisar com mais cuidado o código, percebemos que a saída difere do esperado por causa do desvio de execução da linha 3. O valor de entrada zero desvia a execução para o bloco else.

Para nossa próxima hipótese, utilizaremos como entrada o valor x = 1. Neste caso, o código executado será realmente o do primeiro bloco. No entanto, o laço *while* continua sem ser executado, pois a condição de entrada inicial não foi ainda coberta. Nossa saída esperada, portanto, é 1. Acrescentaremos um caso de teste para esta combinação de entrada e saída:

```
public class CTest {
    ...
    public void testInput1() {
        assertEquals(1, C.m(1));
    }
}
```

Quadro 9 – Caso de teste para a hipótese m(1) = 1

Ao executarmos este novo teste, não há falha. O processo então continua a partir do início, para identificar mais casos de teste. Deste pequeno exemplo, podemos ver que, com o desenvolvimento destes dois casos de testes, pudemos identificar três características do trecho de código estudado:

i. O método executa de modos totalmente diferentes para valores negativos e positivos
ii. A lógica para o valor zero é a mesma para os valores negativos.
iii. No primeiro bloco de código, o valor de x de entrada será sempre maior ou igual a 1

Nosso nível de domínio do sistema ainda não nos permite descrever qual a lógica para cada um dos casos, porém já compreendemos melhor o código do que antes de iniciarmos o desenvolvimento dos testes. Se continuarmos desenvolvendo novos testes de modo a testar áreas ainda não testadas da aplicação iremos compreender melhor o código.

Metodologia de refactoring

Ao aplicarmos a técnica de refactoring somos capazes de construir um modelo mental do código. Refactoring é definido como "o processo de modificar um sistema de software de modo que o comportamento externo do código não seja alterado mas que a estrutura interna seja melhorada."[9]. Neste trabalho utilizamos uma definição mais relaxada:

"Refactoring é o processo de modificar a estrutura interna de um sistema de software sem alterar seu comportamento externo".

Quadro 10 - Definição de refactoring

A diferença entre as duas definições vem do nosso interesse apenas em entender o código, e não necessariamente melhorá-lo. Estamos livres para fazer modificações no código que piorem sua estrutura, se isto nos ajudar a compreendê-lo.

A definição de refactoring exige que não alteremos o comportamento externo do software. Para garantirmos que o processo de refactoring realmente não está alterando

este comportamento precisaremos de um mecanismo que permita a verificação desta propriedade, Nossa metodologia de refactoring garante isso através do desenvolvimento de testes anterior à realização do refactoring.

O processo de realização do refactoring em si, é uma aplicação experimental de refactorings unitários, selecionados com base na experiência do mantenedor, de modo a levá-lo a compreender melhor o código. O processo completo pode ser resumido na Figura 3.

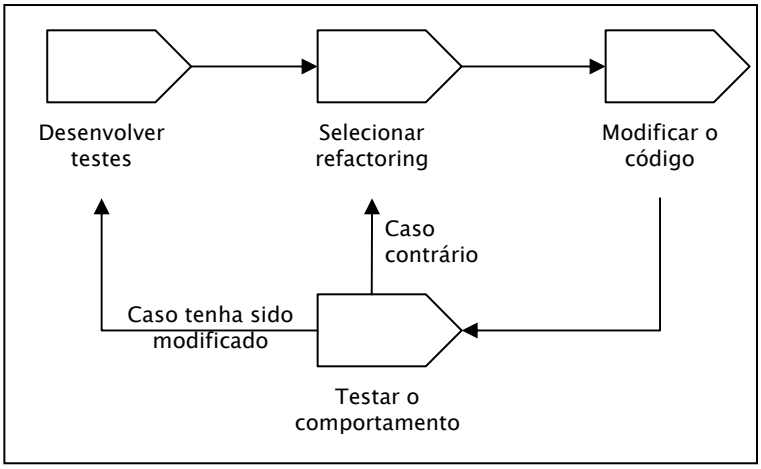


Figura 3 - Metodologia de refactoring

Observamos que a metodologia para refactoring está intimamente ligada à de geração de testes, já que utilizamos os testes para garantir a manutenção do comportamento do código. Estendendo esta idéia, podemos dizer que o refactoring só poderá ser feito quando o trecho de código a ser alterado tiver sido completamente compreendido o que, em nossa abordagem, significa testado. Decidimos então, no nosso experimento, integrar ambas as técnicas de modo a obter resultados mais expressivos do que com as técnicas utilizadas separadamente.

Aplicação da metodologia de refactoring

Voltaremos ao trecho de código exposto no Quadro 6 considerando que, além dos testes desenvolvidos na seção *Aplicação da metodologida de testes*, possuímos um conjunto de testes capaz de cobrir satisfatoriamente nossa base de código. Como forma de entendermos melhor o código apresentado, aplicaremos um pequeno refactoring a ele.

O expressão do lado direito da atribuição da linha 4, inclui o que em Java é chamado de operador ternário. A semântica deste operador na linguagem Java é a mesma de uma construção if em uma linguagem funcional: caso a avaliação do primeiro argumento do operador denote um valor booleano verdadeiro, a expressão toma o valor do segundo argumento, caso contrário o valor tomado é o do terceiro argumento. O Quadro 11 expõe a regra de redução para o operador ternário segundo a semântica padrão da linguagem.

```
true ? a : b → a
false ? a : b → b
```

Quadro 11 - Semântica do operador ternário de Java

Considerando esta regra de avaliação, podemos mudar esta linha de código para uma construção if...else sem modificar o comportamento do código.

```
int f = x > 0 ? x : 1;   ⟺   if (x > 0)
                                 f = x;
                             else
                                 f = 1;
```

Quadro 12 - Refatoração de operador ternário

O próximo passo é executar os testes, o que nos mostra que nosso refactoring realmente não modificou o comportamento do sistema, pois todos os testes continuam rodando com sucesso. Porém, após a realização do refactoring, podemos notar que as

linhas 3 e 4 do código são iguais. Ambas realizam o teste para x > 0, mas a linha 4 só é executada se a condição testada na linha 3 for verdadeira. Isto mostra que o teste repetido na linha 4 é desnecessário, pois a execução sempre será desviada para a linha 7. Podemos então substituir toda a construção if...else por uma chamada incondicional.

```
public static int m(int x) {
    if (x > 0) {
        if (x > 0)
            f = x;
        else
            f = 1;
        while(x > 1) {
            f *= --x;
        }
    return f;
    } else {
        int y = x;
        int r = x;
        ++x;
        while(x < 0) {
            r *= y;
            ++x;
        }
        return r;
    }
}
```

⟺

```
public static int m(int x) {
    if (x > 0) {
        int f = 1;
        while(x > 1) {
            f *= --x;
        }
    return f;
    } else {
        int y = x;
        int r = x;
        ++x;
        while(x < 0) {
            r *= y;
            ++x;
        }
        return r;
    }
}
```

Quadro 13 – Eliminação da estrutura condicional

Continuando o processo, poderemos chegar ao código do Quadro 14, semanticamente equivalente ao código do Quadro 6 e mais compreensível.

```
public class C {
	public static int m(int x) {
		if (x > 0) {
			return fat(x);
		} else if (x == 0) {
			return 0;
		} else {
			return (int) Math.pow(x, Math.abs(x));
		}
	}

	private static int fat(int x) {
		if (x == 0)
			return 1;
		else
			return x * fat(x - 1);
	}
}
```

Quadro 14 – Código final

Ao aplicar o processo de refactoring ao trecho de código selecionado, aprendemos alguns aspectos importantes:

i. Após um refactoring, nos pareceu muito mais claro o fato do operador ternário ser desnecessário.
ii. O processo de refactoring é capaz de modificar o código de modo que fique mais fácil para o mantenedor entendê-lo, como pudemos observar com a eliminação do condicional desnecessário.
iii. Não precisávamos saber anteriormente a semântica do operador ternário na linguagem de implementação. A partir da combinação de testes e refactoring poderíamos inferir o comportamento.

O exemplo sobre o qual trabalhamos é extremamente simples, mas suficientemente compreensível para dar ao leitor um melhor entendimento do conceito de refactoring e testes aplicados ao contexto de manutenção, e mais especificamente à habilidade de compreensão de código necessária à manutenção.

Sistemas estudados

Como estudo de caso analisamos quatro sistemas de domínios diversos. Essa diversidade nos permitiu levantar um conjunto de métricas virtualmente aplicável a vários domínios. Todos os sistemas analisados são escritos predominantemente na linguagem de programação Java e alguns dados relevantes pode ser encontrados na Tabela 2 abaixo:

Sistema	Versão analisada	Tamanho[1] (KNCSS)[2]	Descrição
JUnit	3.8.1	2,7	Framework largamente utilizado para implementação e execução

[1] Na avaliação de tamanho foi considerado apenas o código-fonte Java. Código fonte em outras linguagens (como XML, C ou JSP) não foi computado.

[2] NCSS (Non-commented source statements): Medida de tamanho de código-fonte na qual não são computados os comentários ou linhas em branco.

			de testes unitários em Java
Tomcat	5.5.9	108,7	Servidor de aplicação web de larga utilização
JaTS	Stable Release	20,6	Sistema de geração e transformação de código Java
APES 2	2.5.2	17,7	Ferramenta para modelagem de processos

Tabela 2 - Resumo dos sistemas

Para cada um dos sistemas, foi selecionado um conjunto de código capaz de representar a arquitetura do sistema como um todo. Para este conjunto selecionado, foram gerados casos de teste e parte do código foi refatorado. O tempo decorrido neste processo foi medido e controlado, de modo que os sistemas fossem estudados por um tempo aproximadamente igual e os resultados do experimento estão disponíveis no Apêndice B.

Observações experimentais

O processo de aprendizado de um sistema tem como objetivo aumentar o nível de domínio (ou compreensão) de um indivíduo ou grupo de indivíduos sobre uma massa de código fonte. No início do processo, há uma fase de exploração para familiarização com o código na qual são identificados os componentes principais do sistema. Esta fase é marcada por muita experimentação e o nível de domínio do sistema cresce lentamente.

Após a fase de exploração, há uma nova fase onde o nível de domínio cresce mais rapidamente. Nela, os principais componentes já foram identificados e a concentração de esforço de aprendizado neles permite uma compreensão mais rápida do sistema. Durante esta fase, tende-se a explorar o sistema 'de dentro para fora', no sentido de que o aprendizado é iniciado nos componentes centrais e segue para os mais externos (ou auxiliares).

A última fase corresponde ao nível de domínio desejado para o sistema. Nesta fase, as funcionalidades mais importantes do sistema já foram verificadas e há muito pouco de novo para aprender. Portanto, o nível de domínio volta a crescer muito lentamente, quase estagnando.

Para fins didáticos, o resultado deste processo pode ser resumido em um gráfico relacionando o nível de domínio ao tempo. A este gráfico chamaremos de curva de aprendizado (Figura 4).

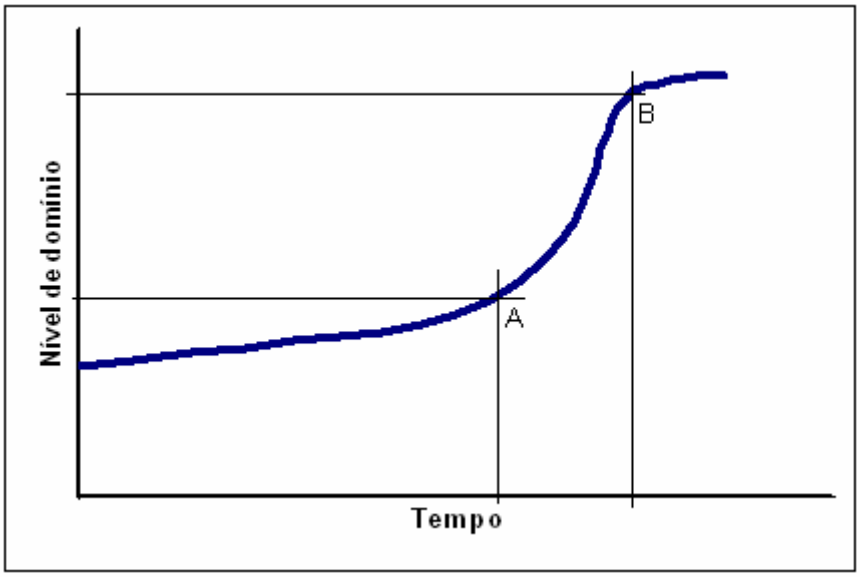


Figura 4 – Curva de aprendizado

Resultados observados

A seguir, identificamos várias métricas que podem ser utilizadas para avaliar a facilidade de compreensão de um trecho de código. A partir destas métricas é possível aproximar a curva de aprendizado para um sistema específico.

Para cada métrica são identificados os seguintes itens:

i. Nome da métrica: Um nome curto capaz de identificar a métrica
ii. Definição: Uma definição sucinta e não-ambígua da métrica
iii. Interpretação: A interpretação da métrica no contexto deste trabalho
iv. Resultados: Os resultados colhidos para a métrica nos sistemas estudados

Métrica: Tempo para o primeiro teste

Definição: Tempo decorrido entre o início do processo de desenvolvimento de testes e a escrita do primeiro caso de teste, que deve envolver as classes que o mantenedor identificou serem mais significativas para a aplicação.

Interpretação: Pode-se observar que a produtividade de escrita de casos de teste aumenta com o passar do tempo. A produtividade de aprendizado é baixa no início da análise de um novo sistema devido à falta de familiaridade do programador com o código a ser testado. Esta falta de familiaridade é diretamente traduzida no tempo necessário para escrita do primeiro caso de teste e pode ser relacionada com o trecho inicial da curva de aprendizado.

Resultados: A Tabela 3 relaciona os tempos necessários para desenvolvimento do primeiro teste para os sistemas estudados.

Sistema	Tempo para o primeiro teste (min)
Apes 2	74
JaTS[1]	--
JUnit	68
Tomcat	60

Tabela 3 - Tempos para o primeiro teste

Métrica: Cobertura média por linha teste

Definição: Relação entre o tamanho do código de teste e o número de instruções testadas

Interpretação: Sistemas altamente acoplados tendem a apresentar um valor alto para esta métrica, devido a um crescimento aproximadamente exponencial da quantidade de chamadas necessárias para a realização de uma função. A falta de

[1] Não foi possível determinar o tempo para o sistema JaTS

modularidade detectada por esta métrica dificulta a utilização de técnicas como *Mock Objects*[14] para isolar os componentes sob teste e, conseqüentemente, a manutenção deles isoladamente.

Resultados: A Tabela 4 relaciona os resultados obtidos para os sistemas estudados.

Sistema	Cobertura média por linha de teste
Apes 2	3.54
JaTS	72.67
JUnit	4,59
Tomcat	38.57

Tabela 4 – Cobertura média por linha de teste

Métrica: Velocidade média de desenvolvimento de testes

Definição: Relação entre o tamanho de código de teste e o tempo utilizado no processo de desenvolvimento de testes

Interpretação: Considerando a hipótese de que a geração de casos de teste está condicionada à compreensão do código testado, a velocidade de desenvolvimento de testes está relacionada à velocidade de aprendizado já que o processo de desenvolvimento de testes adotado evita a escrita de testes redundantes.

Resultados: A Tabela 5 relaciona os resultados obtidos para velocidade de desenvolvimento de testes para os quatro sistemas estudados.

Sistema	Velocidade média de desenvolvimento de testes (NCSS/h)
Apes 2	11,9
JaTS	22,1

JUnit	20,7
Tomcat	11,4

Tabela 5 – Velocidade média de desenvolvimento de testes

Métrica: Velocidade de desenvolvimento de testes após exploração

Definição: Relação entre o tamanho de código de teste e o tempo utilizado no processo de desenvolvimento de testes, desconsiderando o tempo para o primeiro teste.

Interpretação: Esta métrica desconsidera o tempo de exploração, correspondente ao tempo para o primeiro teste. Assim, podemos ter uma aproximação da velocidade média de aprendizado na fase de maior produtividade (na Figura 4, o trecho entre A e B).

Resultados: A Tabela 6 relaciona os valores para a velocidade de desenvolvimento de testes após exploração dos quatro sistemas.

Sistema	Velocidade de desenvolvimento de testes após exploração (NCSS/h)
Apes 2	13,5
JaTS[1]	--
JUnit	23,3
Tomcat	12,7

Tabela 6 – Velocidade de desenvolvimento de testes após exploração

Métrica: Tamanho médio do caso de teste

Definição: Tamanho total do código de teste dividido pelo número de métodos de teste

[1] Não foi possível determinar esta velocidade para o sistema JaTS pela impossibilidade de determinar o tempo para o primeiro teste

Interpretação: A execução de um caso de teste consiste de, basicamente, três passos:

i. Preparar o ambiente para execução;
ii. Executar uma funcionalidade; e
iii. Checar os resultados.

A experiência mostra que a porção de código responsável pela execução da funcionalidade tende a ser pequena em relação ao total de código, sendo os grandes responsáveis pela extensão dos casos de teste a preparação e a checagem. Um código de preparação extenso indica alta dependência implícita de outras partes da aplicação (falta de modularidade), pois mostra que é trabalhoso colocar o sistema em um estado válido para teste. Código de checagem extenso, por outro lado indica alta concentração de responsabilidade nos componentes testados.

Resultados: A Tabela 7 relaciona os resultados observados da métrica tamanho médio do caso de teste para os quatro sistemas estudados.

Sistema	Tamanho médio do caso de teste (NCSS)
Apes 2	33,14
JaTS	18,5
JUnit	23,21
Tomcat	21

Tabela 7 – Tamanho médio do caso de teste

Métrica: Fração de código auxiliar

Definição: Fração do código utilizada para implementar funcionalidades auxiliares à implementação dos casos de teste

Interpretação: O código auxiliar, para os casos de testes implementados durante este experimento, implementa, em sua maioria, código de preparação do ambiente. Assumindo que o código de preparação não deve ser longo, por denotar a utilização de objetos excessivamente grandes[2], esta métrica está inversamente relacionada com a modularidade do sistema.

Resultados: Os resultados de fração de código auxiliar para os quatro sistemas estudados estão relacionados na Tabela 8 abaixo.

Sistema	Fração de código auxiliar
Apes 2	47%
JaTS	43%
JUnit	33%
Tomcat	66%

Tabela 8 – Fração de código auxiliar

Métricas para compreensibilidade

"As coisas deveriam ser tornadas tão

simples quanto possível, porém não mais simples."

– Albert Einstein

As conclusões e resultados apresentados no capítulo anterior nos permitem comparar os quatro sistemas estudados quanto à facilidade de compreensão. Porém o processo para obtenção destas métricas é dispendioso e extremamente dependente da experiência e disciplina do mantenedor. No mundo real, desejamos obter resultados rápidos para que tenhamos uma base de comparação inicial. Estes resultados deverão ser idealmente obtidos de forma automática a um custo mínimo.

Embora esteja fora do escopo deste trabalho construir um sistema para obter estes resultados automaticamente, relacionamos algumas métricas do código fonte (as quais chamamos de *métricas de entrada*) dos sistemas com os resultados do experimento. Estas relações podem no futuro ser compostas em um modelo de avaliação de modo que seja possível ter um "chute inicial" para a facilidade de compreensão dos sistemas a partir do código-fonte do sistema. A Figura 5 esquematiza a utilização deste modelo de avaliação.

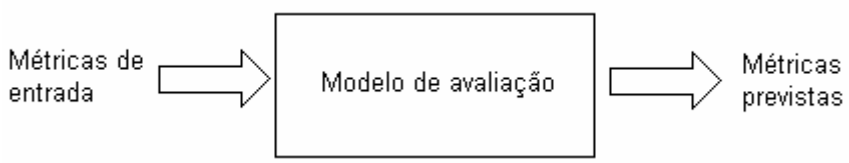


Figura 5 – Esquema do modelo de avaliação

Neste capítulo, primeiramente, definiremos um conjunto de métricas que podem ser facilmente coletadas automaticamente a partir do código fonte do sistema. A estas métricas chamaremos de *métricas de entrada*. Depois aproximaremos relações

matemáticas entre estas métricas de entrada e as métricas definidas no capítulo anterior, chamadas de *métricas de saída*, com base nos resultados do experimento.

Métricas de entrada

Para cada métrica identificada, são relacionados o nome e a definição da métrica utilizada neste trabalho.

Métrica: Falta de coesão entre métodos[4]

Definição: A falta de coesão entre métodos (LCOM) é uma medida de coesão de uma classe. Se m(A) é o número de métodos que acessam um atributo A, LCOM é a média de m(A) para todos os atributos da classe subtraída do número de métodos e dividida por (1-m) (Figura 6).

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

Figura 6 - Falta de coesão entre métodos

Onde:

a é o número de atributos definidos na classe C

m é o número de métodos definidos na classe C

A_j é o j-ésimo atributos da classe C

$\mu(A_j)$ é o número de métodos que referenciam *Aj*.

Métrica: Complexidade ciclomática[4]

Definição: O valor da complexidade ciclomática *V(G)* de um grafo *G* com *n* vértices, *e* arestas, e *p* componentes conectados é

$v(G) = e - n + p$.

Dado um método, associamos a ele um grafo que tem um único ponto de entrada e um ou mais pontos de saída. Cada vértice no grafo corresponde a um bloco de código seqüencial e as arestas correspondem às instruções de desvio do método. Consideramos instruções de desvio qualquer construção do método que envolva alguma decisão. Por exemplo, o valor de complexidade ciclomática do método *m* (Quadro 15), cujo grafo está ilustrado na Figura 7, é $V(G) = 8 - 7 + 2 = 3$.

```
def m
    while(b1)
        b1 = x > 3
    end
    if (b2)
        a = 3
    else
        a = x
    return a
End
```

Quadro 15 – Método *m*

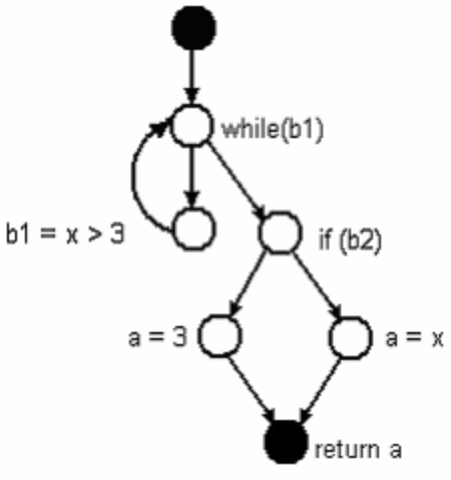


Figura 7 – Grafo do método *m*

Métrica: Acoplamento aferente e eferente[16]

Definição: As métricas de acoplamento são uma medição da interdependência entre os vários componentes de software. O acoplamento aferente diz respeito aos componentes dependentes do componente avaliado, e o acoplamento eferente diz respeito àqueles dos quais ele depende. Neste trabalho, os componentes que tiveram o

acoplamento analisado foram os pacotes Java, portanto o acoplamento aferente de um pacote é o número de pacotes para os quais ele provê serviços (é referenciado) e o acoplamento eferente é o número de pacotes que ele referencia.

Métrica: Profundidade de bloco[17]

Definição: Número máximo de blocos aninhados dentro de um método. Assim como a complexidade ciclomática, é uma medida da complexidade de um método.

Métrica: Número de descendentes[17]

Definição: Número de descendentes (filhos ou descendentes indiretos) de uma classe. Uma classe é considerada filha de outra se houver relacionamento de herança entre elas. A classe que herda (mais especializada) é a filha.

Métrica: Profundidade na árvore hierárquica[17]

Definição: Número de ancestrais de uma classe. Em uma linguagem que permita herança múltipla, é considerado o maior caminho de uma classe até seu ancestral mais primitivo.

Relacionamento entre as métricas de entrada e saída

As métricas coletadas para os sistemas estão resumidas na planilha de métricas (Apêndice A). A partir dos resultados do experimento, geramos uma representação gráfica para cada uma das métricas de saída e entrada. Estes gráficos estão relacionados abaixo, na Figura 8 e na Figura 9, respectivamente.

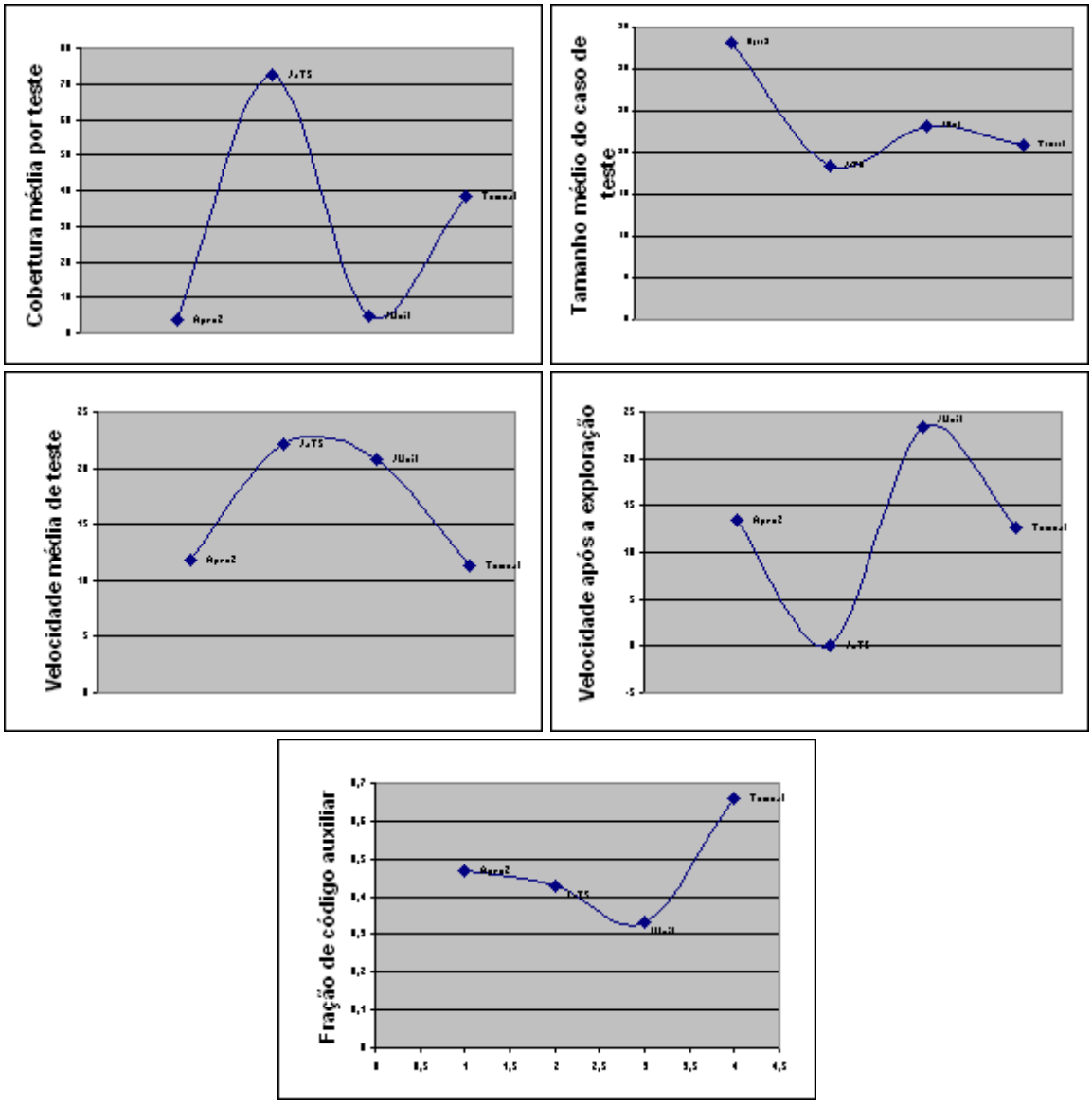
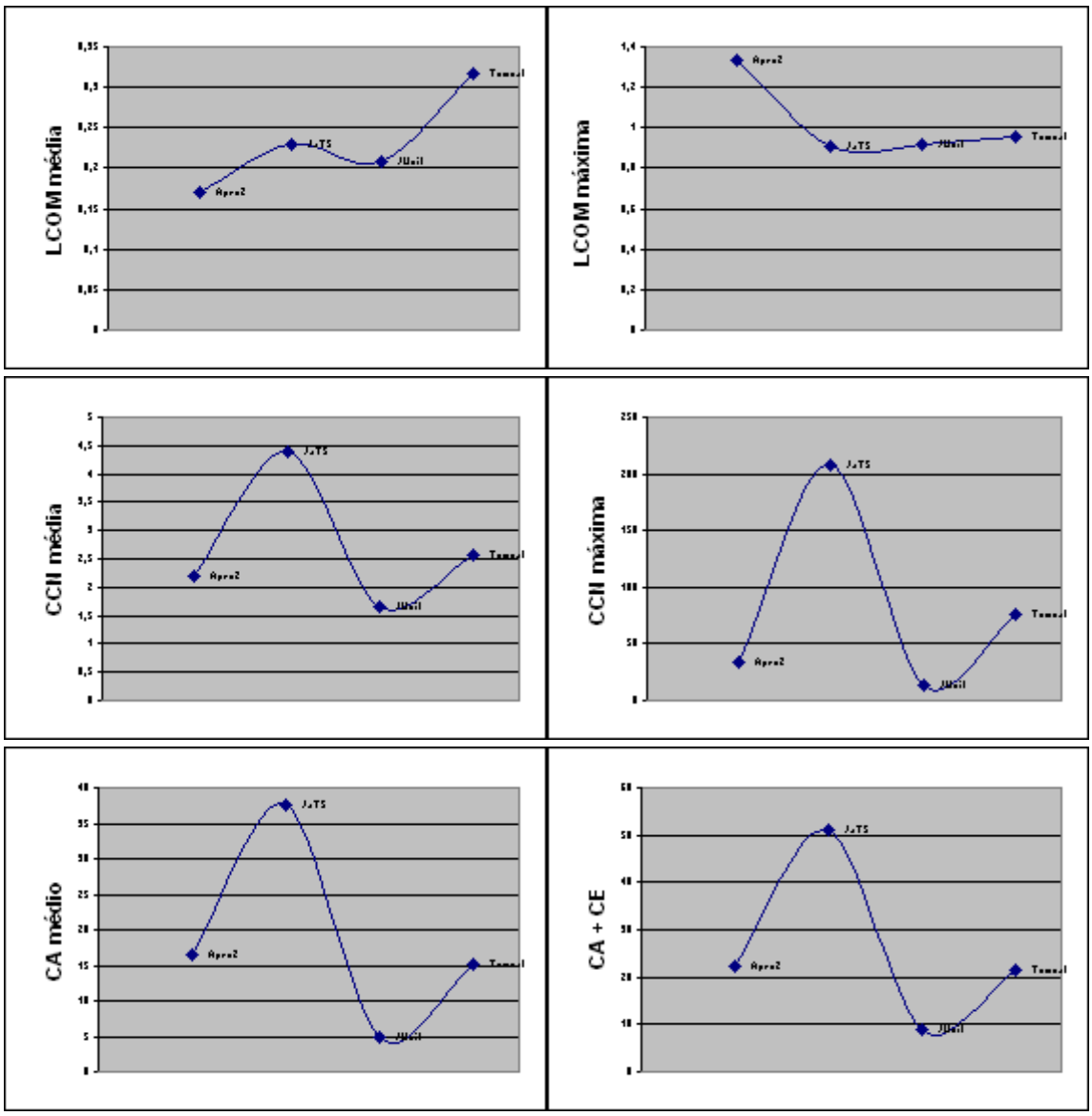


Figura 8 – Métricas de saída


LCOM média
LCOM máxima
CCH média
CCH máxima
CA médio
CA + CE

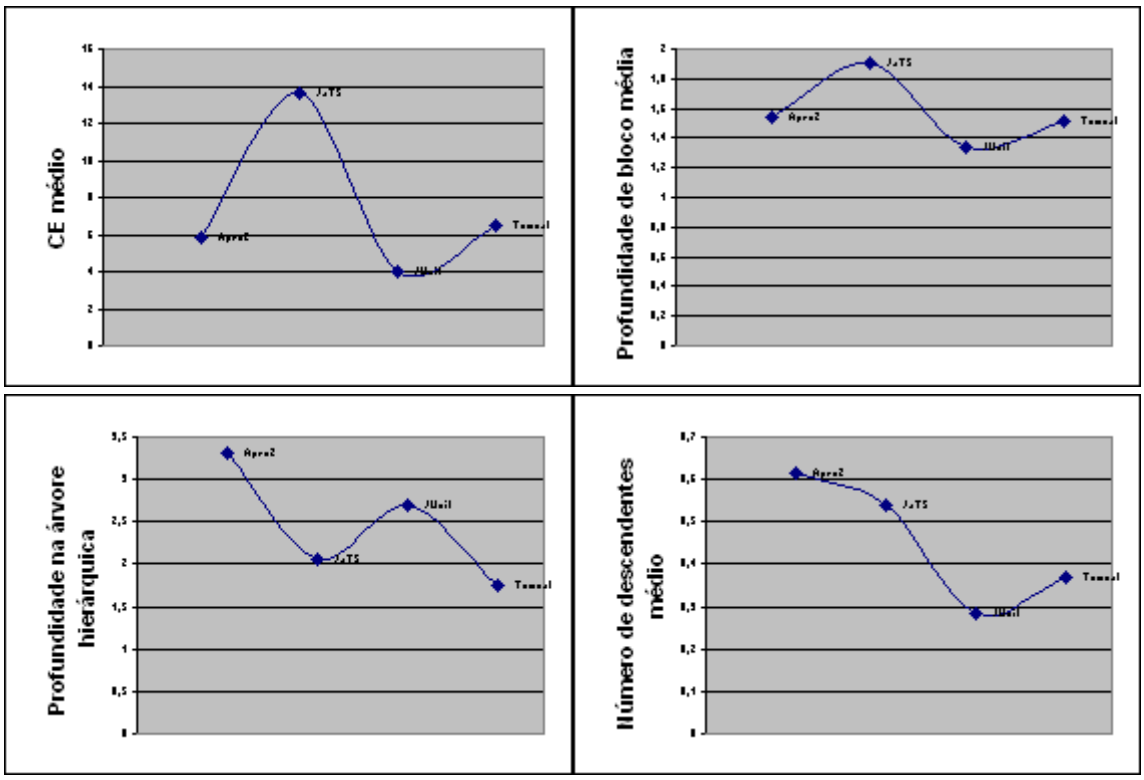


Figura 9 – Métricas de entrada

A partir dos gráficos acima, podemos identificar alguns possíveis relacionamentos entre as métricas de entrada e saída. Os relacionamentos encontrados estão representados de forma gráfica na Figura 10 e na seção seguinte, são feitas algumas considerações sobre eles.

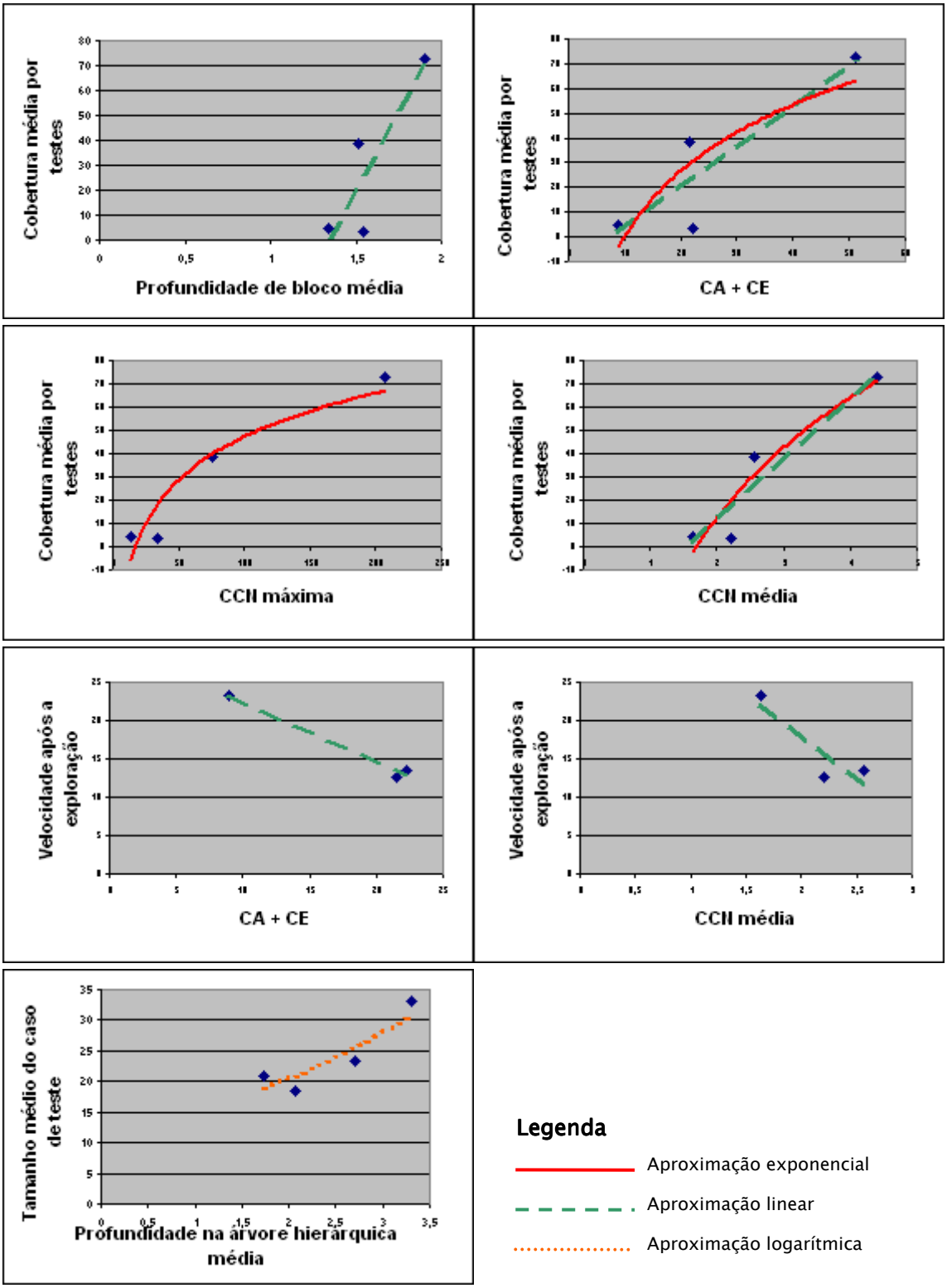


Figura 10 – Relacionamentos entre as métricas

Considerações

A partir deste experimento, algumas relações interessantes puderam ser encontradas e algumas hipóteses inicialmente confirmadas. Esta seção discute alguns aspectos interessantes observados através do experimento.

O primeiro é a afirmação da complexidade ciclomática como medida de compreensibilidade de código. Como pudemos observar, ela é inversamente proporcional à velocidade de desenvolvimento de testes após a exploração. Este resultado vem a confirmar que a métrica, apesar de ter sido desenvolvida visando avaliação de sistemas estruturados, é aplicável a sistemas orientados a objeto.

Porém, ainda mais aplicável a sistemas orientados a objeto, são as métricas de acoplamento. Podemos confirmar isso pela melhor aproximação linear observada no gráfico *CA + CE x Velocidade após a exploração* em relação ao *CCN x Velocidade após a exploração* (Figura 10). Este resultado vem a concordar com os expostos por Rajaraman e Lyu[21].

Outro resultado interessante foi o relacionamento da métrica profundidade na árvore hierárquica com o tamanho dos casos de teste. O gráfico nos sugere que hierarquias muito profundas tendem a se tornar confusas, refletindo diretamente no tamanho dos casos de teste.

Conclusões e trabalho futuro

"Prefiro os sonhos do futuro à história do passado."

– Thomas Jefferson

Os resultados que obtivemos confirmam experimentalmente que a complexidade ciclomática por método e o acoplamento estão diretamente relacionados com a dificuldade de manutenção. Alguns conceitos de orientação a objeto, como herança e polimorfismo, podem ajudar a diminuir esta complexidade e, conseqüentemente, a dificuldade de manutenção. Em especial, podem ser usados sistematicamente para eliminar complexidade na forma de cadeias de instruções condicionais.

A pesquisa também apontou alguns trabalhos para o futuro. Das três habilidades que identificamos para o mantenedor, trabalhamos apenas a compreensibilidade. Seria interessante incrementar o modelo aqui iniciado através de um trabalho similar para as outras duas habilidades. Nesta linha, podemos apontar de antemão a utilização de refactoring como técnica para medir a modificabilidade do código.

Neste trabalho, realizamos refactoring de código porém não correlacionamos nenhuma métrica de refactoring com as métricas do código fonte original. A experiência com a utilização de testes para avaliação de facilidade de compreensão sugere que as métricas de refactoring sejam utilizadas para avaliar a facilidade de modificação.

Além disso, reconhecemos que o conjunto de dados aqui analisado não é suficientemente grande. Uma linha de pesquisa possível para o futuro é realizar o experimento aqui descrito para mais sistemas. Isto permitiria, por exemplo, validar e calibrar as aproximações feitas a partir dos gráficos no capítulo "Métricas para compreensibilidade".

Glossário

Design: Organização dos componentes de software para formar sistemas ou componentes.

Mantenedor: Indivíduo ou organização responsável pela manutenção de um sistema de software.

Mock Objects: Técnica de teste na qual parte do código de produção é substituída por implementações triviais que simulam o comportamento do código real. O termo pode também ser utilizado para classes de objetos criadas como esta técnica.

TDD: Sigla de Test-Driven Development. Técnica para desenvolvimento de software em que o código de teste é escrito antes do código do sistema.

Refactoring: Reestruturação de código, sem alteração do comportamento externamente observável do software.

Referências

"Se enxerguei mais além, foi por estar sobre os ombros de gigantes"

– Isaac Newton

[1] Arthur, L. J. (1988) *Software evolution: The software maintenance challenge*, Willey, New York, USA

[2] Beck, K. (2002) *Test Driven Development: By Example*, Addison-Wesley

[3] Brooks, F. (1982) *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading Mass

[4] Chidamber, S. R. Kemerer, C. K. (1994) *A Metrics Suite for Object Oriented Design*, IEEE Transactions. on Software Engineering., Vol.20, No.6, June 1994.

[5] Cockburn, A. (2001) *Agile Software Development*, Addison-Wesley

[6] Coleman, D. Ash, D. Lowther, B. and Oman, P. (1994) *Using metrics to evaluate software system maintainability*, IEEE Computer, Vol. 27 No. 8, IEEE Press

[7] DeMarco, T. (1982) *Controlling Software Projects*, Yourdon Press, NY

[8] Fenton, N. Pfleeger, S. L. (1996) *Software Metrics: A Rigorous and Practical Approach*, International Thompson Computer Press, London

[9] Fowler, M. (1999) *Refactoring: Improving the design of existing code*, Addison-Wesley

[10] Gibson, V. R. and Senn, J. A. (1989) *System Structure and Software Maintenance Performance*, Communications of the ACM, ACM Press, New York, NY, USA

[11] Hoare, C. A. R. (1961) *Algorithm 64: Quicksort*. Communications of the ACM, Vol. 4 Issue 7, ACM Press, New York, NY, USA

[12] IEEE Std. 610.12-1990 (1993) *Glossary of Softvare Engineering Terminology*, Software Engineering Standards Collection, IEEE CS Press, Los Alamitos, Calif., Order No. 104846T

[13] Lientz, B. P. Swanson, E. B. and Tompkins, G. E. (1978) *Characteristics of application software maintenance*, Communications of the ACM, Vol. 21 Issue 6, ACM Press, New York, NY, USA

[14] Mackinnon, T. Freeman, S. and Craig, P. (2000) *Endo-Testing: Unit Testing with Mock Objects*, eXtreme Programming and Flexible Processes in Software Engineering, Cagliari, Sardinia, Itália

[15] Marinescu, R. (2002) *Measurement and Quality in Object-Oriented Design*, Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara

[16] Martin, R. C. (1994) *OO Design Quality Metrics: An Analysis of Dependencies*, Proceedings of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94

[17] Martin, R. C. (2002) *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall

[18] McCabe, T. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, 2(4):308-320

[19] Oman, P. and Hagemeister, J. (1992) *Metrics for assessing a software system's maintainability*, Proceedings of Conference on Software Maintenance, IEEE Press

[20] Parnas D. L. (1994) *Software Aging*, Proceedings of The 16th International Conference on Software Engineering, IEEE Press

[21] Rajaraman, C. e Lyu, M. (1992) *Reliability and Maintainability Related Software Coupling Metrics in C++ Programs*, Proceedings of International Symposium on Software Reliability Engineering, IEEE Press

[22] Sommerville, I. (1995) *Software Engineering*, Sixth Edition, Addison-Wesley

[23] Swanson, E. B. (1976) *The dimensions of maintenance*, Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, Los Alamitos, CA, USA

[24] Zitouni, M. and Abran, A. (1996) *A model to evaluate and improve the quality of software maintenance process*, Proceedings of the Sixth International Conference of Software Quality, Ottawa