ÁLVARO CÉSAR PEREIRA DA SILVA

PRINCÍPIOS *S.O.L.I.D. DE DESIGN APLICADOS NA MELHORIA DE CÓDIGO-FONTE EM SISTEMAS ORIENTADOS A OBJETOS*

LAVRAS - MG
2013

ÁLVARO CÉSAR PEREIRA DA SILVA

PRINCÍPIOS *S.O.L.I.D. DE DESIGN APLICADOS NA MELHORIA DE CÓDIGO-FONTE EM SISTEMAS ORIENTADOS A OBJETOS*

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, área de concentração em Engenharia de *Software*, para obtenção do título de Bacharel em Sistemas de Informação.

Orientador
Prof. Dr. Cristiano Leite Castro

LAVRAS – MG
2013

ÁLVARO CÉSAR PEREIRA DA SILVA

PRINCÍPIOS *S.O.L.I.D. DE DESIGN APLICADOS NA MELHORIA DE CÓDIGO-FONTE EM SISTEMAS ORIENTADOS A OBJETOS*

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, área de concentração em Engenharia de *Software*, para obtenção do título de Bacharel em Sistemas de Informação.

APROVADA em 21 de AGOSTO de 2013

Prof. M. Sc. André Grützmann	UFLA
Prof. Dr. André Luiz Zambalde	UFLA
Prof. Dr. Raphael Winckler de Bettio	UFLA

Prof. Dr. Cristiano Leite Castro
(Orientador)

LAVRAS – MG
2013

AGRADECIMENTOS

Agradecer é sempre uma tarefa difícil e, por mais cuidado que tenhamos ao escolher a quem agradecer e como agradecer, não raro fica a sensação de injustiça, pois são tantos fatores que nos levam às nossas conquistas, que mesmo nossos chamados "inimigos" tem parcela no nosso sucesso. Mas, como não poderia deixar de aproveitar esse momento, vejamos o que o coração traz à tona.

Gostaria de agradecer aos meus familiares, mãe (Margarida), pai (Cláudio), meus avós (Vô Chico e Vó Nair, que mesmo não tendo tido a chance de partilhar desse momento comigo nessa vida, fizeram muito pela minha criação), aos meus tios e tias, em especial Tia Marise e Tio Luís que foram como pais em muitos momentos. Agradecer pelo esforço, paciência, confiança, amor e por todas as broncas e castigos. Vocês me permitiram galgar algo maior nessa vida.

Agradecer ao Prof. Cristiano pela oportunidade e orientação nessa etapa, permitindo que este projeto se concretizasse e me dando a chance de amadurecer minha visão sobre o que realmente a universidade pode significar e o que devemos levar de bom dela para nossas vidas.

Aos meus amigos, "poucos – bem poucos – e que não zombavam; Quando, em noite de febre endoidecido; Minhas pálidas crenças duvidavam." (Alvarez de Azevedo). Companheiros sem os quais esses anos seriam tempo perdido: Júlio Farah, Daniel Zanzini, Diego Bachim, Ramon Gonçalves, Bruno (Manga) Pinheiro, Pedro

(Pépe) Nogueira, e, em especial, Arina Magalhães (Nina), Igor Alves, Bruno Moretti, Breno Guerra... Essa conquista é tão de vocês quanta minha.

A todos os colegas, alunos e professores que, de uma forma ou de outra, conviveram e participaram desse ciclo que se encerra com este trabalho.

Agradecer à Squadra Tecnologia e aos meus companheiros de trabalho, Marzon Castilho, Felipe (Finganfor) Freire, Lessandro Lucas, Jhonatan Rodrigues e a todos os demais.

E como não poderia deixar de ser, agradecer à luz e razão do meu viver, meu filho Victor, e, mais do que nunca, agradecer à sua progenitora, minha companheira mais que amada, Samara, que, nesses anos todos esteve ao meu lado, nos momentos bons e ruins. Essa conquista é nossa!

Obrigado, de coração, a todos vocês!

“É estranho, mas as coisas boas e os dias agradáveis são narrados depressa, e não há muito que ouvir sobre eles, enquanto as coisas desconfortáveis, palpitantes e até mesmo horríveis podem dar uma boa história e levar um bom tempo para contar.”

— *O Hobbit, JRR Tolkien.*

RESUMO

Qualidade é um atribulo essencial nos dias de hoje em qualquer ramo de negócio. No mercado de *software* isso não é diferente. Tratar dos aspectos que definem a qualidade de um sistema passa, invariavelmente, pelo processo de Projeto e Construção de *Software* e, como peça fundamental de qualquer produto de *software*, pelo código-fonte produzido. Existem diversos recursos, na literatura, que embasam o processo de construção de um *software* visando à criação de melhores códigos e, consequentemente, melhores sistemas, dentre os quais estão os "Princípios *S.O.L.I.D.* de Design". Este trabalho se propôs a estudar o impacto que estes princípios podem gerar no processo de desenvolvimento de um sistema orientado à objetos, através da comparação de métricas de *software* propostas pela norma padrão de qualidade do produto de *software* aplicadas sobre dois sistemas, desenvolvidos sobre um mesmo documento de requisitos, tendo sido um desses sistemas, construído sob a orientação desses princípios. Os resultados dessa comparação são apresentados e discutidos, trazendo uma visão prática do impacto que esse tipo de recurso pode trazer sobre o produto de *software* em geral, principalmente sobre os aspectos ligados ao código-fonte e mostrando dados que comprovam tais benefícios.

Palavras-chave: SOLID. Qualidade de código. Qualidade de *software.* Projeto de *Software*. Construção de *Software*. Métricas de código.

ABSTRACT

Quality is an essential attribute for all business lines. With software market is not different. Deal with the aspects that defines the quality of a system goes through the Design and Software Construction process and, as one of the most important components of any software projects, through the source code. There are many resources, on the literature which supports the process of design and construction of software aiming better source codes and, consequently, better systems. One of this resources, are the "S.O.L.I.D. Principles of Class Design". These principles were applied on the development of an object oriented software that was measured and the results was compared to the results of the same metrics taken from another software, which was developed in another moment but based on the same requirements specification, aiming to study the impacts that these principles can generate. The results of this comparison are shown and discussed, giving a practical vision of what kind of impact that kind of resource can put on the software product, mainly above the source code artifact and brings data which confirm this approach.

Key-words: SOLID. Software Quality. Source code quality. Software Design. Software Construction. Source code Metrics.

LISTA DE FIGURAS

Figura 1.1 - Tipos de Pesquisa Científica [Jung2009] 16
Figura 2.1– Fluxo de Qualidade e suas depêndencias. [ISO/IEC 9126].....22
Figura 2.2 – Modelo para Qualidade interna e externa [ISO/IEC 9126].....23
Figura 2.3 – Modelo para Qualidade em uso [ISO/IEC 9126].....24
Figura 2.4 – Princípios *S.O.L.I.D.* de Design [Martin2012].....56
Figura 2.5 – Mais de uma responsabilidade [SRP2009].....58
Figura 2.6 – Responsabilidades Separadas [SRP2009].....59
Figura 2.7 – Modelo cliente/servidor fora do OCP [OCP1996].....62
Figura 2.8 – Modelo cliente/servidor de acordo com o OCP [OCP1996].....62
Figura 2.9 – Exemplo de violação do LSP [LSP1996].....64
Figura 2.10 – Manipulação de exceção do Java [KK2009].....65
Figura 2.11 – Classe *Door* implementada em C++ [ISP2009].....67
Figura 2.12 – Classe *Timer* e cliente implementados em C++ [ISP2009]....68
Figura 2.13 – Solução por delegação com uso de *Adapter* [ISP2009].....69
Figura 2.14 – Diagrama da solução por herança múltipla [ISP2009].....70
Figura 2.15 – Hierarquia tradicional de modularização [DIP2009].....72
Figura 2.16 – Hierarquia modular adequada ao DIP [DIP2009].....74
Figura 3.1 – Relatório de métricas – *CodePro Analityx*.....80
Figura 3.2 – Relatório de métricas *HTML – CodePro Analityx*.....81
Figura 3.3 – Opções de configuração de métricas – *CodePro Analityx*.....82
Figura 3.4 – Relatório de métricas SisBase – *CodePro Analityx*.....83
Figura 3.5 – Relatório de métricas SisDev – *CodePro Analityx*.....83
Figura 3.6 – SRP aplicado ao componente Celular – SisDev.....85
Figura 3.7 – SRP aplicado a nível de método – SisDev.....85
Figura 4.1 – Relatório de métricas sobre SisBase e SisDev.....91
Figura 4.2 – Gráfico – Resultado para *Lines of Code*.....94
Figura 4.3 – Gráfico – Resultado para *Average Number of Methods per Type*.....95
Figura 4.4 – Gráfico – Resultado para *Average Lines of Code pes Metho*....95
Figura 4.5 – Gráfico – Resultado para *Average Block Depth*.....97
Figura 4.6 – Gráfico – Resultado para *Average Cyclomatic Complexity*.....98
Figura 4.7 – Gráfico – Resultado para *Weighted Metod*.....98
Figura 4.8 – Gráfico – Resultado para *Efferent Coupling*.....99

LISTA DE TABELAS

Quadro 2.1 – Métricas Externas de Testabilidade *[ISO- 9126 – 2]*.....30
Quadro 2.2 – Métricas Internas de Testabilidade *[ISSO-9126 – 3]*.....32
Quadro 2.3 – Métricas de Qualidade em Uso para Satisfação*[ISSO-912 – 4*.....33
Quadro 2.4 – Categorias de Manutenção de *Software [ISO/IEC 14764]*.....36
Quadro 2.5 – Índices de complexidade e risco relativo [Elias2008].....45

SUMÁRIO

1 INTRODUÇÃO 13

1.1.1 Tipo de Pesquisa 15

1.2 MOTIVAÇÃO 17

1.3 OBJETIVO 18

1.4 ESTRUTURA DO TRABALHO 18

2 REFERENCIAL TEORICO 21

2.1 QUALIDADE DE SOFTWARE 21

2.1.1 Métricas de Qualidade de Software 27

2.1.1.1 Métricas Externas 29

2.1.1.2 Métricas Internas 31

2.1.1.3 Métricas de Qualidade em Uso 32

2.1.2 Manutenção e Manutenibilidade de Software 34

2.1.2.1 Manutenção de *Software* 34

2.1.2.2 Manutenibilidade de *Software* 36

2.1.2.2.1 Métricas Externas de Manutenibilidade 38

2.1.2.2.2 Métricas Internas de Manutenibilidade 40

2.1.3 Métricas de Código 42

2.1.4 Considerações Finais 45

2.2 PROJETO E CONSTRUÇÃO DE *SOFTWARE* 46

2.2.1 Projeto de Software 46

2.2.1.1 Princípios de Projeto de *Software* 46

2.2.2 Construção de Software 48

2.2.2.1 Fundamentos da Construção de *Software* 49

2.2.2.2 Codificação na Construção de *Software* 50

2.2.2.2.1 Princípios da Orientação a Objetos e Gestão de Dependências 51

2.2.2.2.1.1 Princípios S.O.L.I.D. de Design 55

2.2.2.2.1.1.1 Princípio de Responsabilidade Única (SRP) 56

2.2.2.2.1.1.2 Princípio Aberto/Fechado (OCP) 59

2.2.2.2.1.1.3 Princípio da Substituição de Liskov (LSP) 63

2.2.2.2.1.1.4 Princípio da Segregação de Interfaces (ISP) 66

2.2.2.2.1.1.5 Princípio da Inversão de Dependências (DIP) 71

2.2.3 Considerações Finais 75

***3* METODOLOGIA 76**

3.1 DO ESCOPO DO PROJETO 76

3.2 DO *SOFTWARE* BASE 77

3.3 DAS FERRAMENTAS UTILIZADAS 79

3.3.1 Google CodePro Analityx 79

3.4 DO SISTEMA IMPLEMENTADO 82

3.5 DAS DIFICULDADES ENCONTRADAS 86

3.6 DAS MÉTRICAS ESTUDADAS 87

4 ANALISE E DISCUSSÃO DOS RESULTADOS 90

4.1 DOS RESULTADOS OBTIDOS 90

4.1.1 Análise dos resultados 92

4.1.1.1 Métricas de Tamanho do código 93

4.1.1.2 Métricas de Complexidade 97
4.1.1.3 Métricas de Acoplamento 99
4.2 CONSIDERAÇÕES FINAIS 100
5 CONCLUSÃO 102
5.1 TRABALHOS FUTUROS 106
REFERÊNCIAS BIBLIOGRÁFICAS 107
ANEXO A 111
ANEXO B 116

1 Introdução

À medida que cresce a demanda por sistemas complexos, com grande responsabilidade no contexto das organizações, a qualidade desponta como um fator essencial no desenvolvimento de *software*. Sendo assim, cada vez mais, há uma disposição para se investir em qualidade.

Pensar em agregar qualidade, apesar de algo básico, se torna, também, algo complexo, uma vez que o processo de desenvolvimento é algo extenso, abrangente, e muito detalhado, tendo total influência na qualidade final do produto de *software*.

Os conceitos existentes na Engenharia de *Software* propõem como aspectos de qualidade de *software*, entre outras coisas, um código-fonte legível, de fácil expansão e manutenção, sendo o código-fonte um dos mais importantes artefatos gerados num projeto ([Swebok2004]).

Atualmente, existem inúmeras ferramentas, técnicas e modelos de desenvolvimento de *software* que permeiam a garantia da qualidade tanto no âmbito do processo quanto do produto. Conhecer e entender esses mecanismos se torna algo fundamental para que esse objetivo (garantia da qualidade) seja alcançado.

Com isso, propor um processo de desenvolvimento condizente com as situações em que a equipe de desenvolvimento atuará, bem como desenvolver uma arquitetura de *software* voltada a facilitar o

desenvolvimento qualificado são etapas importantes, e a escolha das melhores ferramentas e práticas pode ser algo crucial para que se obtenha um bom artefato de código-fonte.

Este trabalho se propõe a apresentar, de forma teórica e prática, algumas ferramentas existentes no mercado e na literatura que permeiam o desenvolvimento orientado a atender as considerações propostas em [Swebok2004], referentes ao processo de codificação na Construção de *Software*, resultando em um código-fonte de maior qualidade.

Entre os princípios e ferramentas existentes na literatura que permitem o desenvolvimento de códigos melhores, podem ser citados: a técnica de TDD (*Test Driven Development* – [Beck2002]*),* Testes Unitários ([Aniche2012]) e Integrados, *Design Patterns* ([Freeman2007]), Princípios *S.O.L.I.D.* de design ([Martin2000], esses princípios serão detalhados nos capítulos mais adiante), dentre vários outros. Este trabalho estuda os conceitos e práticas envolvidos nos Princípios *S.O.L.I.D.* de design, na busca da construção de códigos-fonte de maior qualidade.

A necessidade de métricas e medição pode ocorrer junto com uma mudança de comportamento do time, interessado em entregar código-fonte com qualidade e que se enquadre nos padrões esperados pela equipe. A essa mudança de comportamento, está associado o importante papel das ferramentas de apoio, que atuam no processo de automação dos ciclos de codificação e teste [Koscianski2006], oferecendo apoio real aos times de desenvolvimento.

Em termos de ferramentas de apoio, existe, hoje, uma gama de artefatos de análise estática que podem ser integrados às *IDEs* (*Integrated Development Enviroment*), permitindo um controle, em tempo de desenvolvimento, da fidelidade do código produzido aos padrões de *layout* e documentação definidos para o projeto, bem como análises de composição e fluxos de códigos, permitindo a visualização de *deadcodes* e metrificação em diversos aspectos. Dentre essas ferramentas, podem ser citadas: *CheckStyle*, PMD, *FindBug*, *Google CodePro Analytix*, entre outras.

No caso particular deste trabalho, a ferramenta Google CodePro Analytix é explorada, devido à seu caráter robusto que permite tanto a análise estática de código, bem como metrificações dos aspectos associados à qualidade.

1.1.1 Tipo de Pesquisa

Uma pesquisa pode ser classificada de acordo com a estrutura apresentada na Figura 1.1 [Jung2009].

Quanto à natureza, este trabalho classifica-se como **pesquisa aplicada**, pois é realizada sobre um problema com finalidades de aplicação. Quanto aos seus objetivos, este trabalho pode ser caracterizado como **pesquisa exploratória**, pois avalia a melhoria do código-fonte e da característica de qualidade de manutenibilidade na construção de um software OO (orientado a objetos) utilizando princípios e metodologias/técnicas de desenvolvimento para descobrir possíveis impactos, negativos ou positivos, nesse processo. Quanto à

sua abordagem, este trabalho é uma **pesquisa quantitativa**, pois visa à medição e à análise de algumas características do código-fonte do

Figura 1.1 - Tipos de Pesquisa Científica [Jung, 2009]

software produzido sob orientação de técnicas e princípios existentes na literatura. Quanto aos procedimentos, este trabalho pode ser caracterizado como um **estudo de caso**, pois é apresentado um estudo da aplicação de técnicas e da análise de seus impactos (fenômenos) no desenvolvimento de um software. A coleta de dados deste trabalho se dá pela **observação** do código-fonte produzido.

1.2 Motivação

O mercado de desenvolvimento de *software* fica, a cada dia, mais exigente, pois os *softwares* exercem papéis cada vez mais vitais nas atividades das organizações.

Um dos fatores diferenciais e pré-requeridos nesse mercado é, sem dúvida, a qualidade dos produtos e, nesse conceito de qualidade, têm-se mais do que ausência de erros e funcionamento correto. Alguns dos indicadores de qualidade de *software* propostos na literatura, dizem respeito às características de manutenibilidade e confiabilidade do *software*, e, um fator fundamental nesses quesitos é o código-fonte gerado.

Sendo assim, tem-se embutido no processo de desenvolvimento de um *software*, a necessidade de uma metodologia, de uma arquitetura, de um ambiente de desenvolvimento que permeie e facilite a implementação de códigos-fonte simples, legíveis, de fácil manutenção e expansão e, principalmente, com baixos índices de erros reportados, e com o menor impacto possível no tempo de desenvolvimento. Nesse contexto, surgiu a proposta deste estudo cujo foco está na aplicação dos princípios de Projeto e Construção de *Software* e metodologias e/ou técnicas de desenvolvimento em busca de qualidade de código-fonte na construção de sistemas orientados a objetos (OO).

1.3 Objetivo

Pode-se definir como objetivo geral deste estudo, através da aplicação dos Princípios *S.O.L.I.D.* de *Design* no desenvolvimento de um sistema OO a partir de um documento de requisitos pré-definido, identificar os possíveis benefícios que estes princípios podem agregar à qualidade do código-fonte gerado.

Como objetivos específicos podem ser citados:

- Produção de código-fonte guiada pelos Princípios *S.O.L.I.D.* de *Design;*
- Produção de métricas em cima do código-fonte gerado com o uso da ferramenta *Google CodePro Analityx*;
- Avaliação dessas métricas e comparação com os resultados da abordagem clássica para fim de análise qualitativa do código-fonte gerado;
- Conclusão sobre os benefícios da aplicação dos princípios abordados e seus impactos no processo de desenvolvimento de *software*;

.

1.4 Estrutura do trabalho

Este trabalho está organizado em seis capítulos.

O segundo capítulo aborda os conceitos de qualidade de *software* propostos na literatura, apresentando a norma ISO/IEC 9126, que trata da qualidade do produto de *software* e suas subáreas. Também são apresentados os conceitos relacionados às formas de medições e métricas relativas a cada uma das subáreas. Neste capítulo também é abordado, de forma mais extensa, o contexto de qualidade de *software* relacionado à Manutenção e Manutenibilidade de *Software*, bem como algumas das métricas relacionadas. Ao final deste capítulo são apresentadas as métricas selecionadas para medição da qualidade do código-fonte e manutenibilidade produzidas neste trabalho.

O terceiro capítulo é destinado à apresentação dos conceitos da Engenharia de *Software* relacionados às áreas de Projeto e Construção de *Software*, seus princípios e fundamentos, que serão a base metodológica deste trabalho. Neste capítulo também são apresentados os Princípios *S.O.L.I.D.* de *Design*, método alvo de estudo deste trabalho.

O quarto capítulo é dedicado ao detalhamento da metodologia aplicada neste trabalho, destrinchando aspectos do sistema implementado em termos do *software* base e do *software* desenvolvido aqui, apresentando algumas das dificuldades encontradas e elicitando as métricas estudadas.

O quinto capítulo traz a análise dos resultados obtidos bem como a discussão em cima dos mesmos, de forma a ilustrar o impacto gerado sobre cada perspectiva da análise realizada.

Por fim, o sexto capítulo traz as conclusões a que se chegou com o presente estudo, abordando a experiência adquirida e os fatores que mais impactaram nessas conclusões. Também são apresentadas sugestões para possíveis trabalhos futuros relacionados ao tema e à metodologia aqui aplicada.

2 Referencial Teorico

2.1 Qualidade de Software

O conceito de qualidade pode ser entendido como sendo um conjunto de características de uma entidade, que lhe confere a capacidade de satisfazer as necessidades (explicitas e implícitas) dos *stakeholders*. No entanto, qualidade é um conceito relativo, está fortemente relacionada à conformidade com os requisitos, à satisfação do cliente e ao atendimento as suas expectativas [Duarte2000].

Qualidade não é algo obtido espontaneamente. Sua aquisição reflete a atuação, de forma objetiva e eficiente, de alguns fatores e princípios empregados no processo de desenvolvimento de um produto, seja ele de *software* ou não, trazendo à tona a idéia de que qualidade deve ser construída. Assim, a qualidade do produto depende fortemente da qualidade de seu processo de desenvolvimento. "Um bom processo não garante que os produtos produzidos são de boa qualidade, mas é um indicativo de que a organização é capaz de produzir bons produtos. Nesse sentido, a NBR ISO/IEC 9126 - sob o título geral 'Engenharia de software – Qualidade do produto' - enfatiza que um processo de qualidade influencia os atributos de ***qualidade interna*** do produto de *software*, que por sua vez influencia os atributos de ***qualidade externa*** desse mesmo produto, e que, consequentemente, influenciam os atributos de ***qualidade em uso*** percebidos pelo usuário." (Edgard Davidson[1]), como pode ser percebido na Figura 2.1.

[1] Edgard Davidson é profissional especialista em engenharia de software e desenvolvimento de sistemas, professor universitário e coordenador de dois cursos de pós-graduação, ambos

A norma ISO/IEC 9126, supracitada, se utiliza das características internas, externas e de uso para descrever um modelo de qualidade de *software*, sendo que uma parte, referenciada como **Qualidade interna e externa**, especifica seis características principais para a qualidade, as quais são subdivididas em subcaracterísticas, que se manifestam externamente quando o *software* é utilizado como parte de um sistema, e são resultantens de atributos internos do *software*.

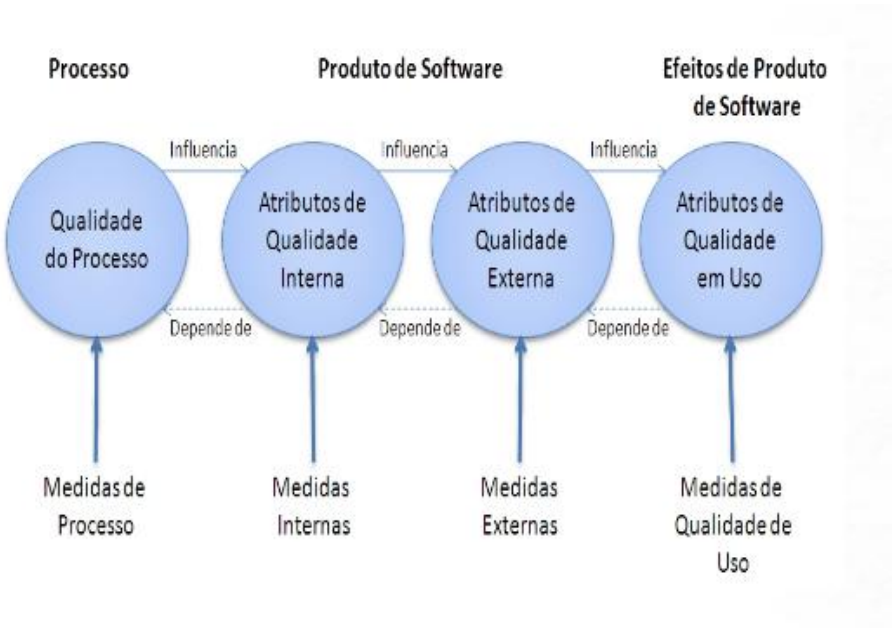


Figura 2.1 – Fluxo de Qualidade e suas depêndencias. [ISO/IEC 9126]

O esquema à seguir (Figura 2.2) ilustra essa subdivisão das caracteristicas de qualidade interna e externa de um produto de software, de acordo com a ISO/IEC 9126:

---

ofertados pela UNA. Mestrando em Engenharia Elétrica com ênfase em Engenharia de Software, Especialista em Engenharia de Software, Graduado em Sistemas de Informação.

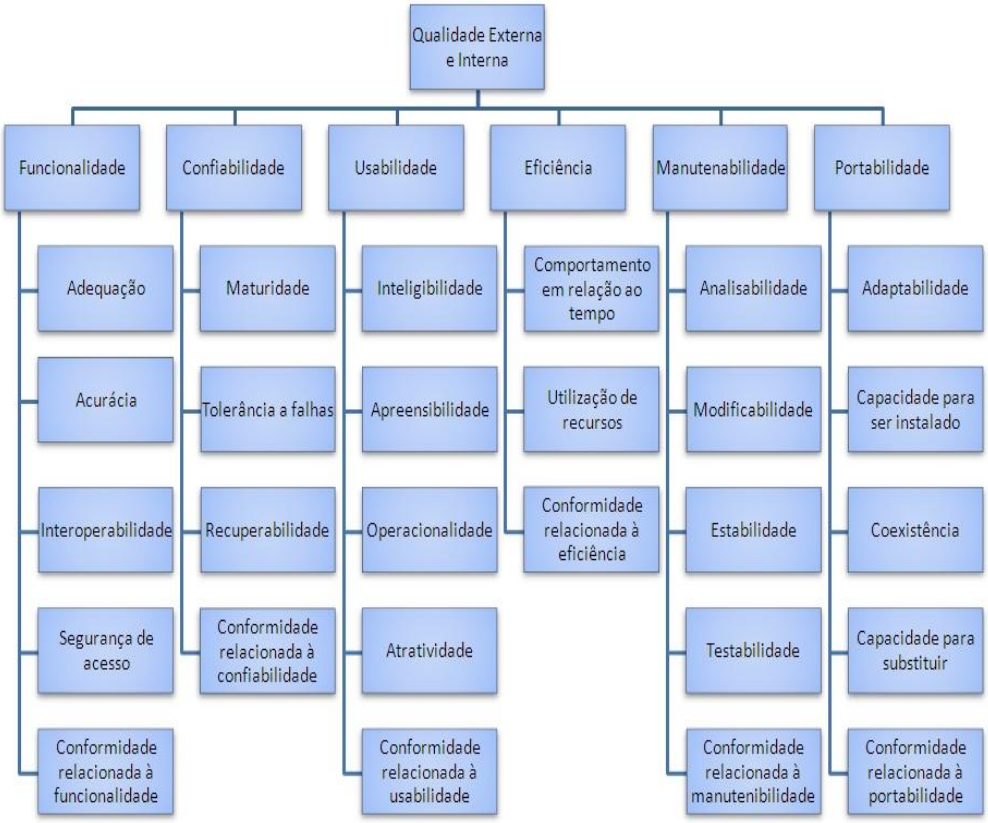


Figura 2.2 – Modelo para Qualidade interna e externa [ISO/IEC 9126]

A segunda parte da norma ISO/IEC 9126 apresenta outras quatro caracteristicas de **Qualidade em Uso**, representada na Figura 2.3 à seguir, porém não se extende além do nível de característica, apresentando apenas, e de forma superficial, a camada superior da sua composição. Segundo definição na ISO/IEC 9126, qualidade em uso é a visão da qualidade do produto de *software* do ponto de vista do usuário, quando este produto é usado em um ambiente e um contexto de uso especificados. Ela mede o quanto usuários podem atingir seus objetivos num determinado ambiente e não as propriedades do *software* em si.

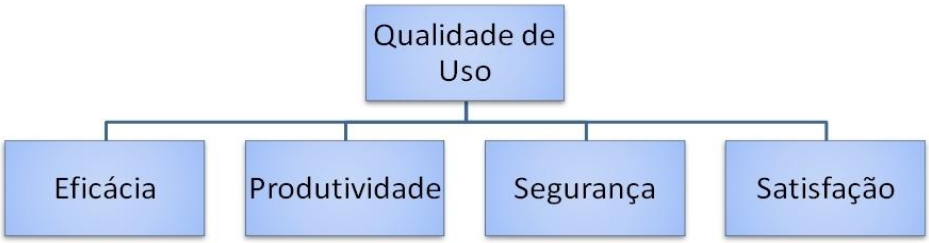


Figura 2.3 – Modelo para Qualidade em uso [ISO/IEC 9126]

As formas de avaliação da qualidade de *software* podem ser por medição direta, indireta ou medição de suas consequencias. O produto de *software* completo pode ser avaliado por métricas externas, sendo que estas métricas descrevem a interação desse produto com o ambiente e são avaliadas através da observação do sistema em operação. A qualidade em uso pode ser medida pelo grau de proximidade no atendimento às metas de qualidade (eficácia, segurança, produtividade e satisfação), quando atuando em um ambiente especifico, operado por um grupo especifico de usuários. Essa avaliação é, geralmente, complementada por medições de aspectos mais específicos, os quais são possíveis também em estégios anteriores do processo de desenvolvimento.

Em estágios inicias do processo, somente recursos e processos podem ser avaliados, porém, à medida que o desenvolvimento anda, e artefatos (especificação, códigos-fonte, etc.) são gerados, estes podem ser medidos por níveis de métricas internas específicas, que podem ser usadas para prever comportamentos e valores para métricas externas ou para o pré-estabelecimento de requisitos essenciais para qualidade externa.

O escopo deste trabalho se limita à avaliação de caracteristicas de qualidade interna/externa do produto de *software*. Nesse sentido, a ISO/IEC 9126 categoriza os atributos de qualidade de *software* em seis categorias, que se subdividem em subcategorias, como mostrado na Figura 2.2. Essas caracteristicas (e respectivas subcaracteristicas) podem ser avaliadas por meio de métricas internas e externas, e estão listadas a seguir (as definições e aspectos apresentados seguem o apresentado diretamente na norma ISO/IEC 9126):

i. **Funcionalidade:** diz respeito à capacidade do produto de software de prover funções que atendam às necessidades explícitas e implícitas, quando o *software* estiver sendo utilizado sob condições especificadas. Esta característica está relacionada com o que software faz para atender às necessidades, enquanto que outras características estão principalmente relacionadas a quando e como ele atende às necessidades.Assume aspectos tais como: *adequação, acurácia, interoperabilidade, segurança de acesso e conformidade.*

ii. **Confiabilidade:** diz respeito à capacidade do produto de *software* de manter um nível de desempenho especificado, quando usado em condições especificadas. Em *software* não ocorre desgaste ou envelhecimento. As limitações em confiabilidade são

decorrentes de defeitos na especificação de requisitos, projeto e implementação. As falhas decorrentes desses defeitos dependem de como o produto de *software* é usado e das opções de programa selecionadas e não do tempo decorrido. Assume as pectos tais como: *maturidade, tolerancia à falhas, recuperabilidade, disponibilidade e conformidade.*

iii. **Usabilidade:** diz respeito à capacidade do produto de *software* de ser compreendido, aprendido, operado e atraente ao usuário, quando usado sob condições especificadas. Como usuários pode-se incluir operadores, usuários finais e usuários indiretos que sejam dependentes ou estejam sob influência do uso do software.Assume aspectos tais como: *intelegibilidade, apreensibilidade, operacionabilidade, atratividade e conformidade.*

iv. **Eficiência:** diz respeito à capacidade do produto de *software* de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas. Recursos podem incluir outros produtos de *software*, configurações de hardware e *software* do sistema e materiais (por exemplo, papel para impressão, disquetes).Assume aspectos tais como: *comportamento em relação à tempo, utilização de recursos e conformidade.*

v. **Manutenibilidade:** diz respeito à capacidade do produto de *software* de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do *software* devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais. Assume aspectos tais como: *analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade.*

vi. **Portabilidade:** diz respeito à capacidade do produto de *software* de ser transferido de um ambiente para outro. O ambiente pode ser organizacional, de hardware ou de *software*. Assume aspectos tais como: *adaptabilidade, capacidade de ser instalado, coexistencia, capacidade para substituir e conformidade.*

Este trabalho assume, como alvo de estudo as características de qualidade agregadas ao artefato de *software* **Código-Fonte**, e seu reflexo e impacto nas características de **Manutenibilidade** do produto de *software*, através da avaliação de métricas inerentes à esse artefato, no processo de construção de *software*, e o aspecto de **Qualidade de software** com foco na qualidade do código-fonte.

2.1.1 Métricas de Qualidade de *Software*

Métricas de *software* podem auxiliar de forma efetiva a administração de um projeto no intuito de controlar determinadas tarefas e/ou rotinas ([Pressmann1995]), garantindo, desta forma,

acompanhamento durante todas as etapas que compõem o ciclo de vida de um projeto de *software* ([Koscianski2006] e [Lanza2006]), baseando-se em padrões e convenções.

A aplicação de métricas no desenvolvimento de *software* vem ganhando uma maior proporção e maturidade entre os times de desenvolvimento, em busca de uma melhor qualidade para o produto final projetado ([Kan2002] e [Pressmann1995]).

Avaliar a qualidade de um produto de *software* é um processo que exige a existência de uma forma se medir as características do produto analisado, e, para se saber o valor de uma dada característica de qualidade, se torna necessário a criação de métricas para quantificá-la e representar seu valor. Essa idéia, proposta por [Duarte2000], sugere que métricas devem ser capazes de fornecer valores matemáticos que verifiquem se o produto de *software* atende aos requisitos de qualidade desejados.

Em uma definição mais clara, uma métrica pode ser entendida como sendo toda particularidade quantificável de um *software* relacionada a uma característica. A aplicação dessas métricas ainda pode variar dependendo do produto analisado, porém as métricas devem ser sempre pertinentes ao paradigma e a tecnologia adotada no desenvolvimento. [ISO/IEC 9126]

[Duarte2000] sugere em seu trabalho uma divisão de tipos de métricas em diretas e indiretas, sendo:

- **Métricas Diretas:** são aquelas realizadas em função de atributos observados e que, através de

uma medição, se aplica uma medida (custo, número de linhas de código, etc.).

- **Métricas Indiretas:** são aquelas descritas em termos de outras características, obtidas através de outras métricas (funcionalidade, complexidade, etc.).

Levando em conta essa divisão inicial, as métricas propostas na ISO/IEC 9126, se enquadram no grupo de métricas indiretas e são classificadas em três categorias que são brevemente explicadas nas subseções seguintes: métricas externas (ISO/IEC 9126 – 2 (2002)), métricas internas (ISO/IEC 9126 – 3 (2002)) e métricas de qualidade em uso (ISO/IEC 9126 – 4 (2002)).

A norma ISO/IEC 9126 (2001) sugere ainda que exista uma relação entre os tipos de qualidades propostos e seus atributos, como mostrado anteriormente na Figura 2.1.

2.1.1.1 Métricas Externas

Métricas externas utilizam medidas de um produto de *software* derivadas de medidas do comportamento do sistema do qual o *software* é parte, através de teste, operação e observação do *software* executável. Métricas externas oferecem a usuários, avaliadores, executores de teste e desenvolvimento os benefícios de poderem avaliar o produto de *software* durante seu teste ou operação. [ISO/IEC 9126]

Para fins de ilustração, o Quadro 2.1 a seguir apresenta uma compilação das métricas apresentadas pela norma ISO/IEC 9126 – 2 como métricas externas de testabilidade, subcategoria do aspecto manutenibilidade de qualidade de *software*.

Nome da Métrica	Proposito da Métrica	Medida e Fórmula	Interpretação	Tipo de Escala	Tipo de Medida
Disponibilidade da função de teste *built-in*	Usuários e mantenedores podem executar testes operacionais sem preparação adicional com facilidade?	X= A / B A = número de casos em que o mantenedor pode usar adequadamente funções built-in B = número de oportunidades de casos de teste	0 <= X <= 1 Quanto mais próximo de 1 melhor	Absoluta	A = quantidade B = quantidade X = quantidade / quantidade
Eficiência de re-teste	Usuários e mantenedores podem executar testes operacionais e determinar se o *software* está pronto para operação ou não?	X = Sum(T) / N T = tempo gasto para o teste determinar se o problema reportado está resolvido N = número de soluções falhas	0 < X Quanto menor, melhor.	Taxa	T = tempo N = quantidade X = tempo / quantidade
Rastreabilidade do teste	Usuários e mantenedores podem executar testes com *checkpoints* após a manutenção facilmente?	X = A / B A = número de casos em que o mantenedor pode pausar e recomeçar a execução do teste no ponto desejado para verificação passo a passo B = numero de pausas na execução do teste	0 <= X <= 1 Quanto mais próximo de 1 melhor	Absoluta	A = quantidade B = quantidade X = quantidade / quantidade

Quadro 2.1 – Métricas Externas de Testabilidade *[ISO/IEC 9126 - 2]*

2.1.1.2 Métricas Internas

Métricas internas podem ser aplicadas a um produto de *software* não executável, tais como documentos ou código-fonte, respectivamente, durante o projeto e a codificação. O proposito básico destas métricas internas é assegurar que a qualidade externa e a qualidade em uso requerida sejam alcançadas. Métricas internas oferecem a usuários, avaliadores, executores de testes e desenvolvedores os benefícios de poderem avaliar a qualidade do produto de *software* e considerar questões relativas à qualidade bem antes do produto de *software* tornar-se executável [ISO/IEC 9126 - 3].

Métricas internas medem atributos internos pela análise das propriedades estáticas dos produtos de *software* intermediários ou preparados para a entrega. As medições de métricas internas utilizam números ou frequências de elementos que compõem o software e que aparecem, por exemplo, em declarações de códigos-fonte, no gráfico de controle e nas representações de fluxo de dados e de transição de estados [ISO/IEC 9126].

Para fins de ilustração, o Quadro 2.2 a seguir apresenta uma compilação das métricas apresentadas pela norma ISO/IEC 9126 – 3 como métricas internas de testabilidade, subcategoria do aspecto manutenibilidade de qualidade de *software*.

Nome da Métrica	Proposito da Métrica	Medida e Fórmula	Interpretação	Tipo de Escala	Tipo de Medida
Completude da função de teste *built-in*	O quão completo é a capacidade de teste da *built-in*?	X = A/B A = número de *built-ins* implementados de acordo com o especificado na revisão B = número de funções *built-in requeridas*	0 <= X <= 1 Quanto mais próximo de 1 melhor	Absoluta	A = quantidade B = quantidade X = quantidade / quantidade
Autonomia da testabilidade	O quão independentemente e o *software* pode ser testado?	X = A/B A = número de dependências em outros sistemas para teste que são simuladas com *stubs* B = número total de dependências de teste em outros sistemas	0 <= X <= 1 Quanto mais próximo de 1 melhor	Absoluta	A = quantidade B = quantidade X = quantidade / quantidade
Observabilidade do progresso de teste	O quão completo são os resultados da *built-in* apresentados durante os testes?	X = A/B A = número de *checkpoints* implementados de acordo com o especificado na revisão B = número de *checkpoints* projetados	0 <= X <= 1 Quanto mais próximo de 1 melhor	Absoluta	A = quantidade B = quantidade X = quantidade / quantidade

Quadro 2.2 – Métricas Internas de Testabilidade *[ISO/IEC 9126 - 3]*

2.1.1.3 Métricas de Qualidade em Uso

Métricas de qualidade em uso medem o quanto um produto atende às necessidades de usuários. A avaliação da qualidade e uso é a

visão da qualidade de um sistema contendo o *software*, sob a perspectiva do usuário. É medida em termos de resultados do uso do *software* e não das propriedades do *software* em si. Qualidade em uso é, para o usuário, o efeito combinado da qualidade externa e interna [ISO/IEC 9126].

Para fins de ilustração, o Quadro 2.3 a seguir apresenta uma compilação das métricas apresentadas pela norma ISO/IEC 9126 – 4 como métricas de satisfação.

Nome da Métrica	Proposito da Métrica	Medida e Fórmula	Interpretação	Tipo de Escala	Tipo de Medida
Escala de Satisfação	Qual o nível de satisfação do usuário?	X = A/B A = questionário com escala psicométrica B=média da população	X > 0 Quanto maior, melhor.	Taxa	A = quantidade X = quantidade
Pesquisa de satisfação	Qual o nível de satisfação do usuário em funções específicas?	X = A A = resultado da pesquisa	Comparação com valores anteriores ou média da população.	Ordinal	A = quantidade X = quantidade
Uso discreto do produto	Qual proporção dos usuários potenciais optou pelo sistema?	X = A/B A = numero de vezes que a função, aplicação ou sistema é usado B = numero de vezes que o usuário teve intenção de usar	0 <= X <= 1 Quanto mais próximo de 1 melhor	Taxa	A = quantidade B = quantidade X = quantidade de/ quantidade

Quadro 2.3 – Métricas de Qualidade em Uso para Satisfação *[ISO/IEC 9126 - 4]*

2.1.2 Manutenção e Manutenibilidade de *Software*

O ciclo de vida do produto de *software* se estende muito além do seu desenvolvimento. Uma vez em operação, defeitos no software são descobertos, ambientes de operação são alterados e surgem novos requisitos, novas necessidades, e o processo de adequação do produto de *software* a esse novo cenário é chamado **Manutenção de Software** ([Swebok2004]).

2.1.2.1 Manutenção de *Software*

De acordo com [Swebok2004] a área de conhecimento Manutenção de *Software* é definida como "o total de atividades necessárias para prover suporte custo benéficas ao software". Essas atividades são realizadas tanto antes da entrega final do produto, como, e principalmente, após a entrega do sistema ao cliente. Entre as atividades do estágio de "pré-entrega" incluem o planejamento das operações "pós-entrega" e determinações logísticas para atividades de transição. No estágio "pós-entrega", as atividades incluem modificação do software, treinamento e atividades de *help-desk.*

A IEEE 1219, padrão para manutenção de *software*, define o processo de Manutenção de *Software* como "modificações de um produto de *software* após a entrega visando a correção de falhas, melhoria de desempenho ou outros atributos, ou adaptação do produto à modificações no ambiente de operação".

A ISO/IEC 14764, padrão internacional para manutenção de *software*, define manutenção de *software* como "modificações do

código e documentações associadas devido a problemas ou necessidade de melhorias. O objetivo é modificar o produto de software existente preservando a integridade", e enfatiza os aspectos de "pré-entrega" da manutenção.

[Swebok2004] apresenta como categorias de Manutenção de *Software*, a mesma divisão apresentada pela ISO/IEC 14764, que inclui quatro categorias, mostradas a seguir:

- **Manutenção Corretiva:** Modificações reativas de um produto de *software*, realizada após a entrega, para corrigir problemas descobertos;
- **Manutenção Adaptativa:** Modificações de um produto de *software*, realizadas após a entrega, para manter um produto *software* "usável" em um ambiente mudado ou em processo de mudanças.
- **Manutenção Perfectiva:** Modificações de um produto de *software* após a entrega para melhorar seu desempenho ou manutenibilidade.
- **Manutenção Preventiva:** Modificações de um produto de *software* após a entrega para detectar e corrigir falhas latentes para evitar que estas se tornem falhas efetivas.

A ISO/IEC 14764 ainda classifica as manutenções adaptativas e perfectivas como melhorias, e agrupa as manutenções corretiva e

preventiva como correções. A ISO/IEC 14764 também caracteriza os tipos de manutenção de acordo com seu caráter de execução, agrupando as manutenções preventivas e perfectivas como proativas e as manutenções corretivas e adaptativas como reativas, como mostrado no Quadro 2.4 a seguir:

	Correção	**Melhoria**
Proativo	Preventiva	Perfectiva
Reativo	Corretiva	Adaptativa

Quadro 2.4 – Categorias de Manutenção de *Software* [ISO/IEC 14764]

2.1.2.2 Manutenibilidade de *Software*

Seguindo a definição apresentada em [Swebok2004], manutenibilidade é a característica que mede o quão facilmente um *software* pode ser mantido, incrementado, adaptado e corrigido para satisfazer requisitos específicos. A ISO/IEC 9126 define manutenibilidade como uma das características de qualidade interna-externa de um *software.*

A característica de manutenibilidade, de acordo com a ISO/IEC 9126, pode ser subdividida em quatro subcategorias de qualidade, sendo estas:

i. **Analisabilidade:** Capacidade do produto de *software* de permitir o diagnóstico de deficiências

ou causas de falhas no *software*, ou a identificação de partes a serem modificadas.

ii. **Modificabilidade:** Capacidade do produto de *software* de permitir que uma modificação especificada seja implementada. Implementação inclui modificações no código, projeto e documentação.

iii. **Estabilidade:** Capacidade do produto de *software* de evitar efeitos inesperados de modificações no *software.*

iv. **Testabilidade:** Capacidade do produto de *software* de permitir que o *software*, quando modificado, seja validade.

Essas subcategorias englobam métricas que permitem a medição e avaliação da componente manutenibilidade da qualidade do *software*, como apresentado na norma ISO/IEC 9126 – 2 (que trata de métricas externas) e ISO/IEC 9126 – 3 (que trata das métricas internas).

As próximas subseções apresentam as métricas propostas pela ISO/IEC 9126 que dizem respeito à característica de manutenibilidade do produto de *software*. À partir dessa abordagem, poderão ser inferidas aquelas que servirão ao propósito deste trabalho, visando a melhorias do código-fonte e consequente melhoria da manutenibilidade.

Por se tratar de um objeto de interesse deste trabalho, descrevem-se, nas subseções seguintes, as métricas de qualidade relacionadas à característica de manutenibilidade, tais como apresentadas na norma ISO/IEC 9126.

2.1.2.2.1 Métricas Externas de Manutenibilidade

A norma ISO/IEC 9126 – 2 apresenta um conjunto de definições e métricas externas para a análise da característica de manutenibilidade de um produto de software, bem como para suas subcategorias. Com base nesse conjunto, tem-se que:

i. **Métricas Externas de Manutenibilidade** devem ser capazes de medir atributos tais como comportamento do mantenedor, usuário ou sistema que inclui o *software*, quando o *software* é mantido ou modificado durante o teste ou manutenção;

ii. **Métricas Externas de Analisabilidade** devem ser capazes de medir atributos como o esforço de mantenedores e usuários ou o gasto de recursos utilizados na tentativa de diagnosticar deficiências ou causas de falhas, ou para identificar partes à serem modificadas. As métricas citadas para analisabilidade são:

capacidade de trilha de auditoria, apoio de função diagnostico, capacidade de analise de falhas, eficiência da analise de falhas e capacidade de monitoramento de *status*;

iii. **Métricas Externas de Modificabilidade** devem ser capazes de medir atributos tais como o esforço de mantenedores ou usuários medindo o comportamento do mantenedor, usuário ou do sistema que contém o software quando se tenta implementar no produto uma modificação especificada. As métricas citadas para analisabilidade são: eficiência do ciclo de mudanças, tempo decorrido na implementação de mudanças, complexidade da modificação, modificabilidade parametrizada e capacidade de controle de mudanças do *software*;

iv. **Métricas Externas de Estabilidade** devem ser capazes de medir atributos relacionados à comportamentos inesperados do sistema que contém o *software* quando é testado ou operado após modificações. As métricas citadaspara estabilidade são: taxa de sucesso de mudanças e localização de impactos de modificações;

v. **Métricas Externas de Testabilidade** devem ser capazes de medir atributos como o esforço de mantenedores e usuários medindo o

comportamento do mantenedor, usuário ou sistema que contem o *software* quando se tenta testar o *software* modificado ou não. As métricas citadas para testabilidade foram apresentadas na Tabela 2.1 apresentada anteriormente;

Uma descrição mais completa sobre as métricas externas de manutenibilidade e suas subcarateristicas pode ser encontrada na ISO/IEC 9126 – 2.

2.1.2.2.2 Métricas Internas de Manutenibilidade

A norma ISO/IEC 9126 – 3 apresenta um conjunto de definições e métricas internas para a análise da característica de manutenibilidade de um produto de *software*, bem como para suas subcategorias. Com base nesse conjunto, tem-se que:

i. **Métricas Internas de Manutenibilidade** são usadas para predizer o nível de esforço requerido para modificar um produto de *software.*

ii. **Métricas Internas de Analisabilidade** indicam um conjunto de atributos para predizer o esforço gasto por mantenedores ou usuários, ou os recursos gastos, na tentativa de diagnosticar

deficiências ou causas de falhas, ou para identificar as partes a serem modificadas em um produto de *software*. As métricas citadas para analisabilidade são: registro de atividades e prontidão das funções de diagnostico;

iii. **Métricas Internas de Modificabilidade** indicam um conjunto de atributos para predizer o esforço gasto por mantenedores ou usuários quando tentando implementar uma modificação especificada no *software.* Uma métrica citada para modificabilidade é: registrabilidade de mudanças;

iv. **Métricas Internas de Estabilidade** indicam um conjunto de atributos para predizer o quão estável o produto de *software* seria após qualquer modificação As métricas citadas para estabilidade são: impactos de mudanças e localização do impacto de mudanças;

v. **Métricas Internas de Testabilidade** indicam um conjunto de atributos para predizer o montante de funções autônomas de ajuda em testes, projetadas e implementadas, presentes no produto de *software.* As métricas citadas para testabilidade foram apresentadas na Tabela 2.2 apresentada anteriormente;

Uma descrição mais completa sobre as métricas internas de manutenibilidade e suas subcaraterísticas pode ser encontrada na ISO/IEC 9126 – 3.

2.1.3 Métricas de Código

Este trabalho visa explorar métricas pertinentes às fases de codificação e teste [Kan2002], pois são estas as encarregadas de manter o *software* em funcionamento [Beck2004]. Neste contexto, o código se torna um instrumento a ser inspecionado e verificado a todo o momento em que ocorrerem alterações, garantindo, através de ciclos de melhorias, o bom funcionamento, desde a menor unidade de código, até a integração de todo o sistema [Duvall2002].

Métricas de código possuem um importante papel no acompanhamento e controle dos processos de codificação e testes, realizados, quase sempre, por equipes com diferentes formas de pensar e atuar no desenvolvimento de *software* ([Sommerville2003] e [Koscianski2006]).

Um dos maiores benefícios desse tipo de métrica é a eliminação de retrabalho, um dos principais problemas em toda a Engenharia de *Software*, que pode ser considerado como o maior dos desperdícios [Poppendieck2008]. Entre as métricas de código disponíveis podem ser citadas:

- **Tamanho:** é uma medida determinística para outras métricas como Complexidade. Entre as principais métricas desse grupo estão: **LOC** (*Line Of Code*) que

determina o número de linhas do código, **KLOC** (*Kilo Line Of Code*) que representa o *LOC* para *softwares* muito grandes, e **SLOC** (*Source Code Line of Code*) que representa o número de linhas lógicas e físicas, na tentativa de eliminar linhas em branco ou comentários realizados [Pressmann1995].

- **Complexidade:** permite mensurar o quanto um determinado módulo do *software* é legível, ou o quão complexo pode se tornar um elevado número de aninhamentos de estruturas lógicas e de repetição dentro de uma funcionalidade [Koscianski2006]. Um bom exemplo da aplicação dessa métrica é a **Complexidade Ciclomática**, onde, quanto maior o índice de complexidade mensurado mais suscetível a falhas o código analisado estará [Lanza2006]. Uma relação entre o índice de complexidade e o risco relativo a cada um desses índices pode ser visto no Quadro 2.5 a seguir. Outras métricas de complexidade são: **Profundidade Média de Bloco** e **Métodos Ponderados** [AnalytiX].

- **Cobertura de Código:** é associada ao desenvolvimento orientado a testes. Essa métrica mensura a abrangência dos testes em relação ao código desenvolvido, tornando o desenvolvimento coeso com o que é solicitado ([Beck2002] e [Beck2004]).

- **Métricas Orientadas a Objetos:** essas métricas visam avaliar alguns fatores como o nível de acoplamento ou

de abstração de um determinado módulo, bem como medir a profundidade da árvore de herança ([Lanza2006], [Pressmann1995] e [Sommerville2003]). Algumas das principais métricas deste segmento, as chamadas métricas CK (Chidamber e Kemerer) são: **WMC** (*Weighted Methods per Class*), que evidencia o índice de métodos ponderados por classe, mensurando a complexidade de cada um separadamente, **DIT** (*Depth of Inheritance Tree*) que mensura a profundidade da árvore de herança, indo da raiz à classe mais distante, **NOC** (*Number of Children*) que mensura as descendências de uma determinada classe, **CBO** (*Coupling Between Object classes*) que relaciona o nível de acoplamento entre objetos, sendo eles meramente acessados, herdados ou por receberem exceções de um nível inferior, **RFC** (*Response For Class*) que representa o número de métodos que podem ser chamados em resposta a uma mensagem recebida, e **LCOM** (*Lack of Cohesion in Methods*) que relaciona a falta de coesão, indicando o quanto os métodos se relacionam, como por exemplo, utilizando atributos compartilhados de uma classe [Koscianski2006].

Complexidade Ciclomática	Avaliação de Riscos
1-10	Um simples programa sem muito risco.
11-20	Programa complexo, risco moderado.
21-50	Programa de alo nível de complexidade.
Maior que 50	Programa instável (risco muito alto).

Quadro 2.5 – Índices de complexidade e risco relativo [Elias2008]

2.1.4 Considerações Finais

Este capítulo apresentou os conceitos relacionados à Qualidade de *Software*, apresentados na literatura, bem como as considerações trazidas pela norma NBR ISO/IEC 9126 (e apêndices). Também foram apresentados os conceitos de Manutenção e Manutenibilidade de *Software*, alvos deste trabalho, bem como a teoria por trás das Métricas de *Software* envolvidas no processo de avaliação da qualidade.

Por fim foram apresentadas algumas métricas ligadas aos aspectos estudados neste trabalho (qualidade de código e manutenibilidade).

2.2 Projeto e Construção de *Software*

2.2.1 Projeto de *Software*

[Swebok2004] define *Projeto* como sendo o processo de definição de arquitetura, componentes, interfaces e outras características de um sistema ou componente, e, também, como sendo o resultado desse processo. Ainda de acordo com [Swebok2004], se visto como um processo, Projeto de *Software* é a atividade do ciclo de vida da engenharia de *software* na qual os requisitos são analisados para produzir uma descrição da estrutura interna do *software* que servira de base para sua construção.

O processo de Projeto de *software* tem papel crucial na qualidade do código-fonte gerado, visto que é nesse momento que são definidos os princípios que serão empregados na construção da solução proposta, e também onde se definem os fluxos existentes e as ferramentas de apoio a serem empregadas.

2.2.1.1 Princípios de Projeto de *Software*

Os princípios de Projeto de *software* são aspectos chaves considerados fundamentais por varias pesquisas e conceitos da área. [Swebok2004] apresenta os princípios de Projeto de *software* como sendo:

- **Abstração:** processo de “esquecimento (ou desconsideração) de informações de forma que coisas diferentes possam ser tratadas com iguais”. Existem dois mecanismos de abstração no contexto de Projeto de *software*, especificação e parametrização. Abstração por especialização conduz a três tipos principais de abstração: procedural, de dados e controle de abstração.

- **Acoplamento e Coesão:** acoplamento é definido como a força do relacionamento entre os módulos, enquanto coesão é definida por como os elementos que compõem os módulos são relacionados.

- **Decomposição e modularização:** decompor e modularizar um *software* grande em módulos menores e o mais independentes possível, geralmente objetivando agregar funcionalidades ou responsabilidades em diferentes componentes.

- **Encapsulamento / informações escondidas:** encapsulamento/ informações escondidas significa agrupar e “empacotar” os elementos e detalhes internos de uma abstração e fazer com que esses detalhes fiquem inacessíveis.

- **Separação de interfaces e implementação:** separar interfaces e implementações envolve a definição de um componente pela especificação de uma interface publica, conhecida como *cliente*, separada dos detalhes sobre como o componente funciona.

- **Suficiência, completude e primitivismo:** alcançar suficiência, completude e primitivismo significa garantir que o

componente captura todas as características essenciais de uma abstração, e nada mais que isso.

2.2.2 Construção de *Software*

De acordo com [Swebok2004], Construção de *Software* se refere à construção detalhada do trabalho, através de uma combinação de codificação, verificação, testes unitários e integrados e, depuração. A área de conhecimento Construção de *Software* está ligada diretamente à diversas outras áreas de conhecimento da Engenharia de *Software*, devido à enorme quantidade de atividades que não só permeiam as atividades posteriores, mas também consolidam tudo o que foi definido e planejado anteriormente no ciclo de vida do projeto de *software*. Sendo assim, pode-se considerar a construção de *software* como o alicerce do projeto de *software*, a fonte maior de possibilidades de sucesso do projeto.

[Swebok2004] sugere ainda que a Construção é uma atividade na qual o *software* deve chegar a um acordo com as restrições caóticas e arbitrárias do mundo real, e segui-las exatamente. Devido a essa proximidade com o mundo real, a construção é mais impulsionada por considerações de ordem prática do que as outras áreas de conhecimento, e a Engenharia de *Software* é mais "artesanal" nesta área.

As atividades de construção de *software* consolidam, em forma de códigos e artefatos, os projetos realizados e aprovados pela atividade de Projeto de *Software*, tornando palpáveis as soluções proposta através da aplicação de técnicas baseadas nos princípios de Projeto de *Software*, supracitados, e nos fundamentos da Construção

de *Software*, apresentados posteriormente, de forma a produzir um sistema que atenda aos requisitos funcionais e não funcionais da demanda e aos critérios de qualidade requeridos e definidos pela equipe atuante no projeto.

2.2.2.1 Fundamentos da Construção de *Software*

Os fundamentos da Construção de *Software* são apresentados a seguir, segundo sua descrição em [Swebok2004]. Os três primeiros conceitos são aplicados tanto para projeto quanto para construção.

- **Minimizar complexidade:** a necessidade de se reduzir a complexidade dos sistemas é aplicada a, essencialmente, todos os aspectos de construção do *software*, e é particularmente crítico para o processo de verificação e testes da Construção de *Software.* Complexidade reduzida é obtida através da ênfase na criação de códigos simples e legíveis através da utilização de padrões e aplicação de técnicas de codificação.

- **Antecipação de mudanças:** a maioria dos *softwares* sofrerá mudanças ao longo do tempo e a antecipação de mudanças impulsiona muitos aspectos da construção de *software.* O *software* e, inevitavelmente, parte da mudança do ambiente externo e essas mudanças afetam os *softwares* de diversas maneiras.

- **Construção para verificação:** construção para verificação significa construir *software* de forma que

falhas possam ser prontamente detectadas pelo engenheiro de *software*, tanto durante os testes independentes quanto durante as atividades operacionais. Técnicas específicas que dão suporte a construção para verificação incluem seguir padrões de programação de revisão de código, testes unitários, organização de código para testes automatizados e restrição na utilização de estruturas de linguagens complexas ou de difícil entendimento, entre outros.

- **Padrões de construção:** padrões que afetam os esforços da construção incluem: Métodos de comunicação (padrões de formatação de documentos e conteúdos), Linguagens de programação (padrões utilizados nas convenções), plataformas e ferramentas empregadas.

2.2.2.2 Codificação na Construção de *Software*

Em se tratando de construção de *software*, há um consenso de que o processo essencial dessa atividade é a Codificação. Do ponto de vista do desenvolvimento de *software*, codificação é a representação da solução detalhada proposta ao problema em termos de uma linguagem de programação, em geral de alto nível, que possam ser automaticamente traduzidas para linguagem de máquina pelo processo de compilação.

[Swebok2004], apresenta como considerações aplicáveis à atividade de codificação da construção de *software*, as seguintes:

- Técnicas para criação de código-fonte entendível, incluindo padrões de nomenclatura e *layout* de código.
- Uso de classes, tipos enumeráveis, variáveis, constantes e outras entidades similares.
- Uso de estruturas de controle.
- Manipulação de condições de erros, tanto de erros planejados como de fluxos de exceção.
- Prevenção de violações de segurança à nível de código, como estouros de memória.
- Utilização de recursos através de mecanismos de exclusão e controle de acesso à recursos seriamente reusáveis, como *threads* e *locks* de bancos de dados.
- Organização de código-fonte (em rotinas, classes, pacotes ou outras estruturas).
- Documentação de código.
- Refinação de código.

2.2.2.2.1 Princípios da Orientação a Objetos e Gestão de Dependências

Em uma época em que a grande maioria dos programadores e envolvidos em construção de *software* faz uso, direta ou

indiretamente, de linguagens orientadas a objetos, soa meio sem nexo discutir a respeito do que seja projeto orientado a objetos, do que trata esse assunto e quais seus benefícios e custos. Porém, o que se percebe na indústria de *software* atualmente é que a maioria dos envolvidos utilizam essas linguagens sem conhecer os seus porquês, e sem conhecer como se pode explorar a grande maioria dos seus benefícios.

[Martin2000] aponta causas para os problemas de *softwares* com origem no código-fonte, geralmente causados por problemas no Projeto da construção do sistema. Para [Martin2000], o projeto das aplicações de *software* inicialmente é algo limpo, elegante e convincente, de uma beleza tão simples e pura que faz com que *design*ers e implementadores fiquem ansiosos para vê-lo funcionando. Algumas dessas aplicações conseguem manter essa pureza de Projeto através do inicio do desenvolvimento e primeira entrega. A partir daí algo ruim acontece e a beleza do Projeto proposto se torna "uma massa podre de código cada vez mais difícil de manter". Nesse ciclo, eventualmente o esforço necessário para fazer as mais simples alterações no sistema se torna tão alto que engenheiros e gerentes clamam por um re-projeto, que raramente tem sucesso e gera um acúmulo ainda maior de problemas, que atrasam as previsões de entregas e superfaturam os esforços de manutenção a tal ponto, que a equipe se desespera por outro re-projeto.

[Martin2000] cita quatro sintomas principais para esse "apodrecimento" dos códigos-fonte gerados:

- **Rigidez:** é a tendência do *software* de se tornar difícil de mudar, mesmo quando se trata de mudanças simples. Cada mudança causa um efeito cascata de outras

mudanças em módulos dependentes. Quando alterações profundas não são permitidas, a rigidez vem à tona. Assim, o que começa como uma deficiência de projeto acaba convergindo em uma política de gerenciamento adversa.

- **Fragilidade:** é a tendência do *software* "quebrar" em muitos lugares toda vez que é mudado. Frequentemente, essas quebras ocorrem em áreas que não possuem um relacionamento conceitual com a área mudada. Com a fragilidade se tornando cada vez pior, a probabilidade de quebra cresce e torna o *software* impossível de manter. Cada correção o torna pior, introduzindo mais problemas.

- **Imobilidade:** é a inabilidade para reuso de *softwares* de outros projetos ou partes do mesmo projeto. Em geral acontece com um engenheiro descobrindo que precisa de um módulo muito similar ao que outro engenheiro escreveu, porém esse módulo depende de muita "bagagem". Após muito trabalho, percebe-se que o esforço e o risco necessários para separar as partes desejáveis das indesejáveis desse módulo são grandes demais para serem tolerados, causando a reescrita do módulo ao invés do seu reaproveitamento.

- **Viscosidade:** ocorre de duas formas: viscosidade de projeto e viscosidade de ambiente. Quando de encontro com uma alteração, engenheiros geralmente encontram mais de uma forma para realizar a mudança. Algumas

> dessas maneiras preservam o projeto, outras não. Quando os métodos que preservam o Projeto são mais difíceis de serem empregados que os cortes, a viscosidade de Projeto é alta. É fácil fazer a coisa errada, mas difícil implementar a coisa certa. Viscosidade de ambiente ocorre quando o ambiente de desenvolvimento é lento e ineficiente. Por exemplo, se o tempo de compilação é muito longo, desenvolvedores serão tentados a fazer mudanças que não exijam grandes recompilações, mesmo que essas mudanças não sejam ótimas do ponto de vista do projeto.

[Martin2000] aponta como causa desse desses sintomas e, consequentemente, desse "apodrecimento", as mudanças nos requisitos e gestão de dependências, quando esta não é feita da maneira correta durante o desenvolvimento do projeto. Para [Martin2000], as mudanças nos requisitos têm ocorrido de forma que o projeto inicial não pode antecipar e, frequentemente, essas mudanças demandam rapidez em sua conclusão, sendo realizadas por desenvolvedores que não estão familiarizados com a filosofia inicial do Projeto, ocasionando violações no projeto inicial.

Entretanto, a Engenharia de *Software* deixa claro que os requisitos frequentemente mudam e que os engenheiros de *software* devem estar preparados para mudanças em seus projetos. Se o projeto falha demais devido às mudanças de requisitos, o Projeto utilizado é que está falho e se torna responsabilidade do engenheiro de *software*

encontrar maneiras de tornar o projeto resistente as mudanças, protegendo-o desse “apodrecimento”.

Segundo [Martin2000], as mudanças que causam o apodrecimento dos Projetos são aquelas que introduzem dependências novas e não planejadas. Todos os quatro sintomas apresentados anteriormente são, direta ou indiretamente, causados por dependências impróprias entre módulos do *software.* É a arquitetura de dependências que está se degradando e, com ela, a habilidade do *software* em ser mantido.

[Martin2000] diz que para evitar a degradação da arquitetura de dependências, as dependências entre os módulos precisam ser gerenciadas. Tal gerenciamento consiste na criação de barreiras através das quais, dependências não se propagarão.

2.2.2.2.1.1 Princípios S.O.L.I.D. de Design

Nesse contexto, surgiu o conjunto de princípios conhecidos como *S.O.L.I.D.*, que propõem uma maneira para desenvolvimento orientado a objetos visando à gestão de dependências e combate aos sintomas do apodrecimento do código-fonte, permeando a construção de melhores códigos do ponto de vista do Projeto e Construção de *Software.*

Os princípios *S.O.L.I.D.* foram apresentados inicialmente por [Martin2000], como princípios de Projeto de classes, sob uma perspectiva da orientação a objetos.

S.O.L.I.D., como mostra a Figura 2.4, é uma concatenação das iniciais dos nomes dos cinco princípios abordados, sendo estes: princípio da responsabilidade única (SRP), princípio Aberto/Fechado (OCP), princípio da substituição de Liskov (LSP), principio de segregação de interfaces (ISP) e princípio de inversão de dependência (DIP).

Figura 2.4 – Principios *S.O.L.I.D* de Design. [Martin2012]

2.2.2.2.1.1.1 Princípio de Responsabilidade Única (SRP)

"*There should never be more then one reason for a class to change.***"**
[Srp2009]

O princípio de responsabilidade única, ou *Single Responsibility Principle* (SRP), de acordo com Martin [Srp2009] tem como base a coesão. Coesão mede a capacidade da classe em executar uma única função. Classes com alto índice de coesão são mais fácies de se entender e, também, mais fáceis de manter. Essa é a força motivacional por trás do SRP. Se uma classe tiver mais de uma razão para mudar, isso significa que as responsabilidades que causam essas mudanças devem ser separadas em outras classes.

Porque é importante separar responsabilidades em diferentes classes? Martin [Srp2009] responde a isso afirmando que "responsabilidade é um eixo para mudanças". Quando os requisitos mudam, essa mudança se manifestará através de alterações na responsabilidade juntamente com a classe. Se uma classe tem mais de uma razão para mudar, então a responsabilidade se torna acoplada. Alterações em uma responsabilidade podem prejudicar ou inibir a habilidade da classe em lidar com outras responsabilidades. Esse tipo de acoplamento leva a projetos frágeis que "quebram" de maneiras inesperadas quando alterados.

Martin [Srp2009] demonstra o ideal do SRP com o exemplo da Figura 2.5, onde a classe *Rectangle* possui dois métodos, um para desenhar o retângulo na tela e outro para calcular a área do retângulo.

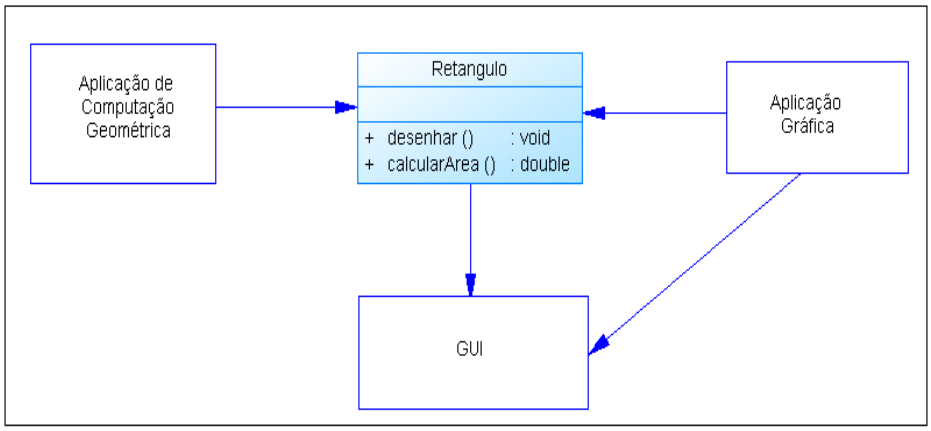


Figura 2.5 – Mais de uma responsabilidade. [Srp2009]

Esse Projeto viola o conceito do SRP. A classe *Rectangle* possui duas responsabilidades, sendo uma, prover um modelo matemático da geométrica do retângulo e a outra, desenhar o retângulo em uma unidade interface gráfica. [Srp2009] alega que caso a classe *Rectangle* seja utilizada em duas aplicações diferentes, sendo uma para computação geométrica, sem qualquer interesse na parte gráfica, e outra puramente gráfica, uma alteração em qualquer dessas aplicações que resulte na necessidade da alteração da classe *Rectangle*, força o desenvolvedor a reconstruir, retestar e reexecutar a outra aplicação de forma a validar os impactos dessa alteração da classe. Não fazer essa validação pode causar quebras diversas que não foram previstas.

Pelo conceito do SRP ([Srp2009]), o ideal seria separar as duas responsabilidades em classes diferentes, como mostrado na Figura 2.6.

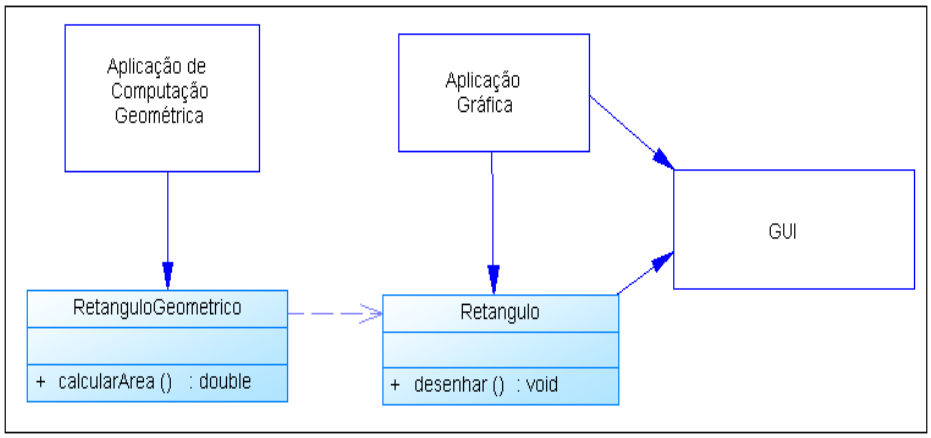


Figura 2.6 – Responsabilidades separadas. [Srp2009]

[Srp2009] conclui afirmando que o SRP é um dos princípios mais simples, porém um dos mais difíceis de praticar corretamente. Agrupar responsabilidades é algo natural do ser humano e, encontrar e separar essas responsabilidades umas das outras é parte da essência do Projeto de *software.*

2.2.2.2.1.1.2 Princípio Aberto/Fechado (OCP)

"Class should be open for extension, but closed for modification."
[Kk2009]

De acordo com [Kk2009], o princípio aberto/fechado, ou *Open-Closed Principle* (OCP), é o mais importante entre todos os da categoria de princípios de classes, já que todos os demais, de certa forma, são derivados do OCP. Este princípio surgiu no trabalho de Bertrand Meyer ([Meyer1988]), que é reconhecido como autoridade no paradigma de orientação a objetos.

[Ocp1996] introduz seu estudo afirmando que quando uma mudança simples em um sistema acarreta um efeito cascata de mudanças para os módulos dependentes, o sistema exibe atributos indesejados que são ligados ao mau projeto. O sistema se torna frágil, rígido, imprevisível e não reutilizável. O OCP ataca esse estado de maneira muito direta, dizendo para o Projeto criar módulos que nunca mudem. Quando os requisitos mudam, o comportamento da classe afetada deve ser alterado pela adição de código, e não pela alteração de código já existente e que já funcione.

De maneira geral, OCP é a habilidade de adicionar novas capacidades aos sistemas, sem a necessidade de modificar um conjunto de classes preexistentes, seguindo um dos objetivos idealizados pela orientação a objetos, que é permitir a adição de novas estruturas de dados ao *software* sem modificar o código base já existente.

Para [Ocp1996], módulos em conformidade com o OCP partem de dois atributos principais:

- **São Abertos para Extensão:** isso significa que o comportamento do módulo pode ser estendido, ou seja, pode-se fazer o módulo adotar comportamentos novos ou diferentes com as mudanças nos requisitos, ou para atender novas necessidades da aplicação.

- **São Fechados para Modificações:** o código-fonte de um dado módulo é inviolável. Ninguém está autorizado a alterar o código desse módulo.

Um questionamento razoável a essa altura é: como manter um módulo fechado e aberto ao mesmo tempo? Por meios naturais, estender o comportamento de um módulo é feito através da alteração desse módulo. Um módulo que não pode ser mudado normalmente tem um comportamento fixo. Então, como esse conflito de atributos pode ser resolvido? [Ocp1996] responde à isso de forma simples, através de *Abstrações.*

A maioria das linguagens orientadas a objetos permite a criação de abstrações que são fixas e ainda assim representam um grupo incalculável de comportamentos possíveis. Essas abstrações são classes bases, e o grupo de incalculáveis possibilidades e representado por todas as possiblidades de classes derivadas dessa abstração. Isso permite que o módulo manipule a abstração, permanecendo fechado para modificação já que depende de uma abstração que é fixa, e podendo ter seu comportamento estendido através da criação de novas classes derivadas da abstração.

[Ocp1996] usa como exemplo, a situação cliente/servidor. A Figura 2.7 apresenta uma representação simples desse modelo, que não segue o OCP. No exemplo, as classes *Client* e *Server* são ambas concretas e não existem garantias de que as funções internas da classe *Server* sejam virtuais. No exemplo, a classe *Client* consome a classe *Server.* Se um objeto *Client* precisa consumir um objeto *Server* diferente, seria necessária uma alteração do *Client* para referenciar esse novo modelo de *Server*.

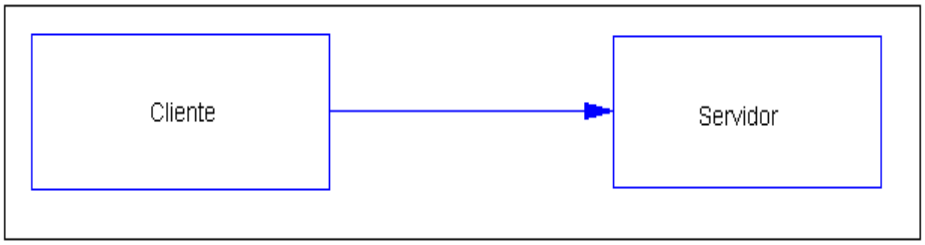


Figura 2.7 – Modelo cliente/servidor fora do OCP. [Ocp1996]

Pelo OCP, como mostra a Figura 2.8, cria-se uma *AbstractServer*, uma classe abstrata, identificada nas figuras daqui em diante por uma elipse, com funções puramente virtuais, que será consumida pela classe *Client*. A classe *Server* passa a ser uma classe derivada da abstração, de forma que o objeto *Client* consome um objeto *Server*. Caso seja necessária a utilização de uma classe *Server* diferente, basta que esse *NewServer* derive da abstração, garantindo que não sejam necessárias mudanças na classe *Client.*

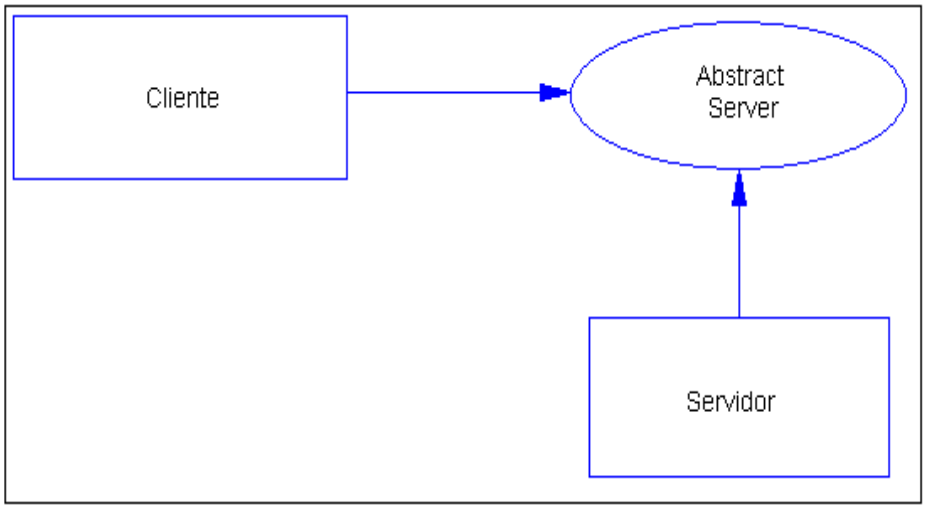


Figura 2.8 – Modelo cliente/servidor de acordo com o OCP. [Ocp1996]

2.2.2.2.1.1.3 Princípio da Substituição de Liskov (LSP)

"*Derived classes must be substitutable for their base classes.****"***
[Martin2009]

Esse princípio foi descrito primeiramente por [Liskov1988], e dizia:

> *"What is wanted here is something like the following substitution property:*
> *If for each object o1, of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."*

[Kk2009] aponta o princípio da substituição de Liskov, ou *Liskov Substitution Principle* (LSP), como sendo uma extensão do OCP, de forma que é difícil diferenciá-los, mas apontando uma sutil diferença entre eles. Para [Kk2009], OCP é centrado no acoplamento de abstrações. LSP, apesar de fortemente ligado no acoplamento de abstrações, também é fortemente ligado a pré-condições e pós-condições, que ligam o LSP ao Projeto por Contrato, de Bertrand Meyer, onde os conceitos de pré-condições e pós-condições foram formalizados.

Uma pré-condição é um contrato que precisa ser satisfeito antes de um método ser invocado. Uma pós-condição, em contra partida, deve ser verdadeira após a conclusão da execução do método. Se uma pré-condição não é conhecida, o método não deve ser invocado, e se uma pós-condição não é conhecida, o método não retorna.

[Lsp1996] apresenta um exemplo simples de violação do LSP, na linguagem C++, referindo-se ao uso do *Run-Time Type Information* (RTTI) para selecionar uma função baseada no tipo de objeto corrente, como mostra a Figura 2.9.

```
void DrawShape(const Shape& s){
    if(typeid(s) == tipeid(Square)){
        DrawSquare(static_cast <Square&>(s));
    } else if(typeid(s) == tipeid(Circle)){
        DrawCircle(static_cast <Circle&>(s));
    }
}
```

Figura 2.9 – Exemplo de violação do LSP. [Lsp1996]

[Lsp1996] classifica este trecho como "claramente mal formado", já que a função precisa conhecer todas as possibilidades derivadas da classe *Shape*, e ainda precisa ser alterada sempre que novas derivações do *Shape* forem criadas.

[Kk2009] diz que a relação de pré-condição e pós-condição possui um significado embutido em um relacionamento de herança não suportado pelo Java, exceto algumas asserções manuais ou comentários não executáveis de forma que as violações do LSP podem ser difíceis de encontrar, mas aponta que para ilustrar como essa relação de pré-condição e pós-condição e o LSP, basta considerar como o mecanismo de tratamento de exceção do Java funciona.

A Figura 2.10, a seguir, ilustra esse comportamento. Considerando que *InvalidAmountException*, Figura 2.10 - *(a)*, é uma exceção definida pela aplicaçao, e que deriva da classe base de

exceção do Java, *Exception*, sendo invocada se o montante a ser depositado for menos que zero.

Pela regra, quando o método *deposit* é sobrescrito em uma subclasse, não se pode invocar uma exceção que esteja em um nível mais alto de abstração que a *InvalidAmountException*, logo, uma declaraçao como a feita em Figura 2.10- (b) não é permitida pois a classe *Exception* é um ancestral da classe *InvalidAmountException*. Mesmo que essa ideia não corresponda diretamente ao conceito de pré-condição e pós-condição, ela pega a essência, já que é possível dizer que qualquer pré-condição estipulada por um método de uma subclasse não pode ser mais forte que as do método da classe base, e que qualquer pós-condição estipulada por um método de uma subclasse não pode ser mais fraca que as do método da classe base.

```
a) public abstract deposito(int valor) throws ValorInvalidoException;
b) public void depositar(int valor) throws Exception;
```

Figura 2.10 – Manipulaçao de Exceçao do Java. [Kk2009]

Para [Lsp1996], a importância do LSP, se torna obvia quando se consideram as conseqüências de violá-lo. Se existe uma função que não segue o LSP, então essa função usa um ponteiro ou referencia para uma classe base, porém precisa conhecer todas as classes derivadas dessa classe base. Essa função viola o OCP porque ela precisara ser modificada sempre que uma nova classe derivada for criada.

2.2.2.2.1.1.4 Princípio da Segregação de Interfaces (ISP)

"Many specific interfaces are better than a single, general interface." [Kk2009]

[Kk2009] diz que qualquer interface deve ser de alta coesão. O Java coloca a interface como sendo um tipo de referência de dados que contém declarações de métodos, mas sem implementações, sendo, em essência, uma classe abstrata com todos os seus métodos abstratos. Entender o papel que uma interface assume no contexto das aplicações é algo de grande importância. De fato, interfaces proporcionam flexibilidade, permitindo que objetos assumam seus tipos, logo, uma interface pode ser interpretada como sendo simplesmente um papel que um objeto assume em algum momento do seu ciclo de vida. Sendo assim, ao definir uma interface, deve-se ter cuidado para que ela não acople múltiplos papéis, garantindo que um objeto, cuja classe implementa esta interface, assumirá uma única responsabilidade.

Para [Isp2009] o Princípio de Segregação de Interfaces, ou *Interface Segregation Principle* (ISP), lida com as desvantagens de interfaces "gordas" ou "poluídas" e, consequentemente, de baixa coesão. O ISP reconhece que existem alguns objetos que requerem interfaces não coesas, no entanto, sugere que os clientes não devem ter conhecimento sobre esses objetos como classes únicas. Em vez disso, clientes devem ter acesso a essas classes através de classes base abstratas que possuam interfaces coesas. Para algumas linguagens, essas classes base abstratas são conhecidas como interfaces, protocolos ou assinaturas.

"Clients should not be forced to depend upon interfaces that they do not use." [Isp2009]

[Isp2009] usa essas palavras para definir o ISP. Quando clientes são forçados a depender de interfaces que eles não usam, então estes clientes estão sujeitos às mudanças dessas interfaces. Isto resulta em um acoplamento inadvertido entre todos os clientes que usam essa interface. É necessário se esquivar desse tipo de acoplamento quando possível, bem como separar interfaces, evitando esse efeito de "poluição".

[Isp2009] usa como exemplo um sistema de segurança onde existem objetos do tipo *Door* que podem estar trancados ou não, e no qual existe um marcador para aberto ou fechado, indicando a situação atual desse objeto. A implementação dessa classe abstrata *Door* é mostrada pela Figura 2.11.

Security Door

```
class Door{
    public:
        virtual void Lock() = 0;
        virtual void Unlock() = 0;
        virtual boolean isDoorOpen() = 0;
};
```

Figura 2.11 – Classe *Door* implementada em C++. [Isp2009]

O exemplo considera também uma segunda classe, *TimedDoor*, que implementa essa interface *Door*, e que precisa soar um alarme quando objeto permanecer aberto por muito tempo. Para isso, este objeto *TimedDoor* comunica-se com um outro objeto chamado *Timer*, que está ilustrado na Figura 2.12. Quando um objeto quer ser

informado sobre o *timeout* de uma porta, ele chama a função *Register* do *Timer*.

Timer

```
class Timer{
    public:
        void Register(int timeout, TimerClient* client);
}
class TimerClient{
    public:
        virtual void TimeOut() = 0;
}
```

Figura 2.12 – Classe *Timer* e cliente implementados em C++. [Isp2009]

Aqui, existe um objeto que tem duas interfaces separadas usadas por dois clientes separados; *Timer*, e os usuários de *Door*. Essas interfaces precisam se implementadas no mesmo objeto desde que essa implementação de ambas as interfaces manipulem os mesmos dados. A questão é: como fazer isso conforme o ISP? Como separar as interfaces quando elas precisam se manter juntas?

Para [Isp2009] a resposta para essas questões encontra-se no fato de que um cliente de um objeto não precisa acessá-lo através de uma interface desse objeto. Este acesso pode e deve, preferencialmente, ser feito através de delegação, ou através de uma classe base desse objeto.

Separação por delegação emprega o *Design* Patter *Adapter* para o problema do *TimedDoor*, criando um objeto adaptador que deriva do *TimerClient* e delaga para o *TimedDoor*, como ilustra a Figura 2.13.

Quando a *TimedDoor* registra uma requisição de *timeout* para o *Timer*, ela cria um adaptador *DoorTimerAdapter* registrado com o

Timer. Quando o *Timer* envia uma mensagem de *timeout* para o adaptador, este delega a mensagem de volta para a *TimedDoor*. Esta solução está em conformidade com o ISP e previne o acoplamento de *Door* com os clientes do *Timer*.

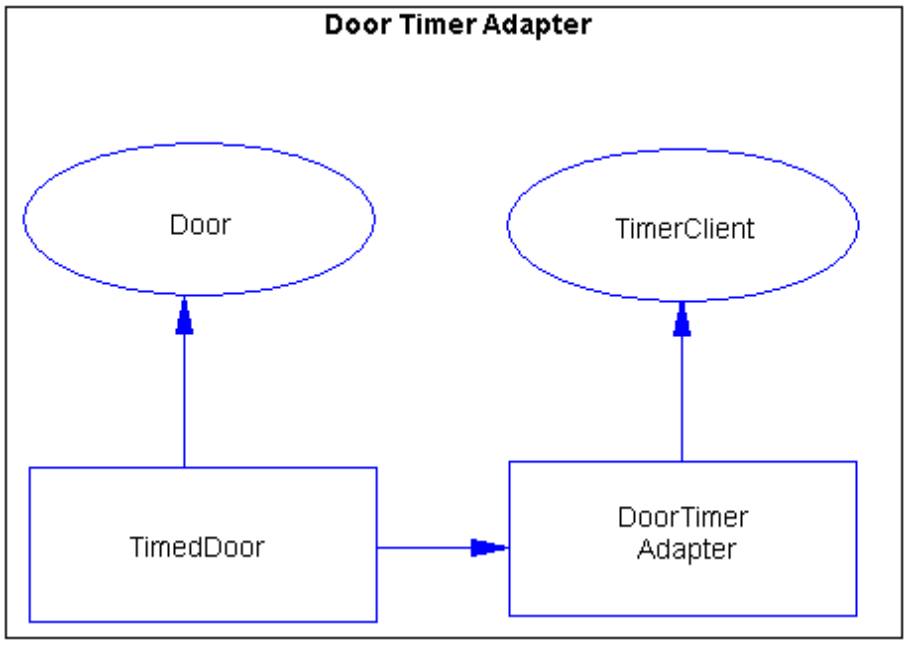


Figura 2.13 – Solução por delegação com uso do *Design* Patter *Adapter*. [Isp2009]

Entretanto essa solução não é elegante, já que ela exige a criação de outro objeto toda vez que ocorre uma requisição de *timeout*. Além disso, a delegação requer um montante, mesmo que mínimo, de memória e tempo de execução. Outra solução proposta por [Isp2009], visando contornar este "porém" da delegação, diz respeito ao uso de herança múltipla para sanar este problema, como mostrado Figura na 2.14.

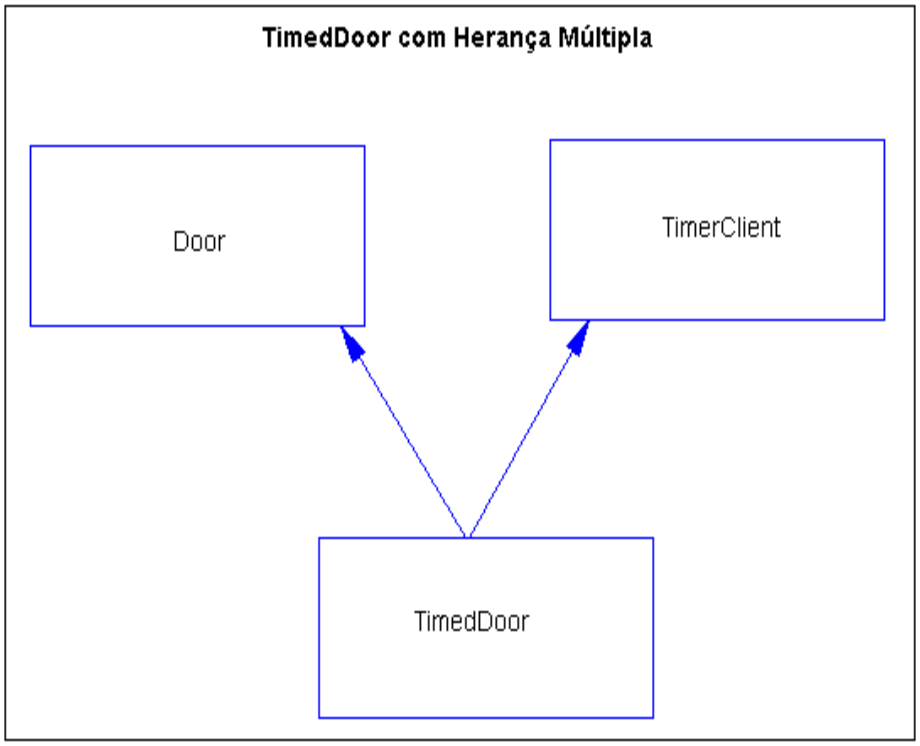


Figura 2.14 – Diagrama da solução por herança múltipla. [Isp2009]

Por esta solução, *TimedDoor* herda de ambas as classes, *Door* e *TimerClient*, e, embora clientes de ambas as classes bases possam fazer uso da *TimedDoor*, nenhum deles se prende realmente à classe *TimedDoor*, ou seja, eles usam o mesmo objeto através de interfaces separadas.

A solução por herança múltipla é preferida por não forçar novos requisitos de máquina. A única situação em que ela é preterida à solução por delegação é quando a interpretação feita pelo adaptador for necessária ou se diferentes interpretações forem necessárias em diferentes momentos.

Mais informações sobre o padrão *Adapter* ou outros *Design Patterns* podem ser encontradas em [Gof1995] e [Freeman2007].

2.2.2.2.1.1.5 Princípio da Inversão de Dependências (DIP)

"Depend on abstractions, not concretions." [Martin2009]

Para [Kk2009], o Princípio de Inversão de Dependência, ou *Dependency Inversion Principle* (DIP), formaliza o conceito de acoplamento abstrato e afirma claramente que se deve acoplar em nível de abstração, não em nível concreto. [Kk2009] define acoplamento abstrato com sendo "a noção de que classes não são acopladas a outras classes concretas, ou que possam ser instanciadas, em vez disso, classes devem ser acopladas a outras classes base ou abstratas". Esse conceito, de acoplamento abstrato, é o meio pelo qual o LSP alcança sua flexibilidade, mecanismo essencial ao DIP, e é o coração do OCP. [Dip2009] aponta o DIP como sento o resultado estrutural do uso rigoroso do OCP e LSP. Para o Java, uma classe abstrata pode ser aquela referenciada pelo modificador *abstract* ou por uma interface.

[Dip2009] justifica a ideia de "inversão" que nomeia o princípio, pelo fato de que modelos mais tradicionais de desenvolvimento, tais como *Design* e Analise Estruturada, tendem a criar estruturas nas quais módulos de alto nível dependem de módulos de baixo nível e onde abstrações dependem de detalhes, de forma que um dos objetivos desses métodos passa a ser a definição de uma hierarquia de subprograma que descreva como módulos de alto nível

fazem chamadas à módulos de nível mais baixo. Nesse sentido, [Dip2009] aponta a estrutura de dependências de um sistema orientado a objetos bem desenhado como sendo “invertida” em relação à estrutura normalmente criada por modelos tradicionais. A Figura 2.15 ilustra a idéia sugerida pelo modelos tradicionais, ilustrando a hierarquia de comunicação entre módulos, onde um módulo de nível mais alto depende de módulos de nível mais baixo.

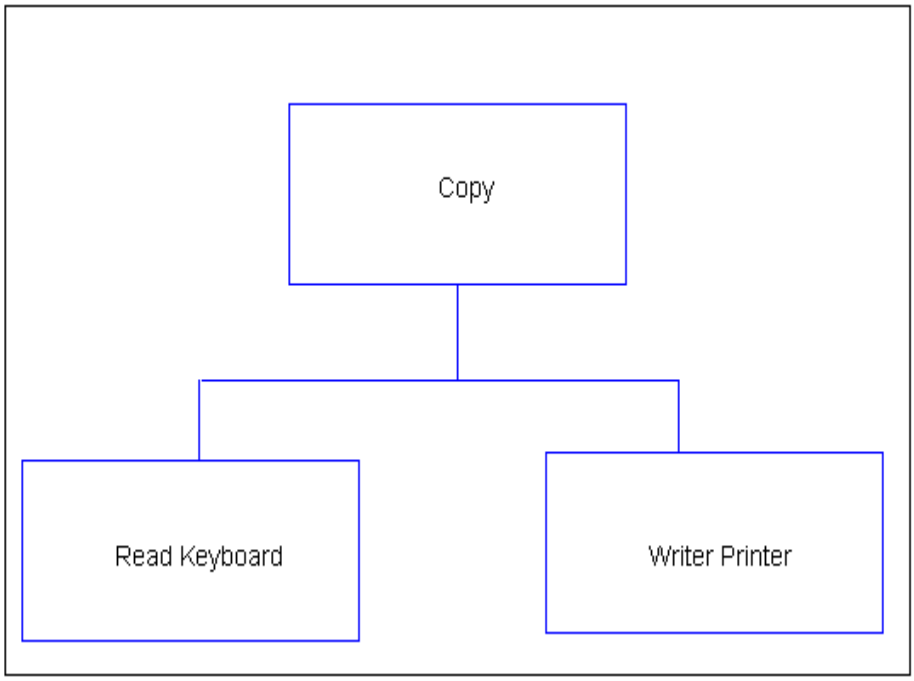


Figura 2.15 – Hierarquia tradicional de modularização. [Dip2009]

O problema apontado para o diagrama da Figura 2.15, de acordo com [Dip2009], é a reusabilidade do módulo de alto nível *Copy*. Devido à dependência com os módulos inferiores *Read Keyboard* e *Write Printer*, *Copy* não é reutilizável em nenhum contexto que não envolva um teclado ou uma impressora, o que é ruim no caso de o módulo *Copy* encapsular a inteligência do sistema ou a política

desejada para reaproveitamento. Modificar o módulo *Copy* é uma saída para que esse problema seja sanado, porém, os impactos de modificações no seu comportamento podem não ser interessantes, e geralmente não são, tornando o sistema rígido e frágil. Esse é um problema comum e seu contexto justifica a proposta do DIP.

[Dip2009] aponta dois precedentes básicos que compõem o conceito vital do DIP:

> *"A: High level modules should not depend upon low level modules. Both should depend upon Abstractions."* [Dip2009]

> *"B: Abstraction should not depend upon details. Details should depend upon abstractions."* [Dip2009]

[Dip2009] considera as implicações dessa relação de dependência entre módulos, afirmando serem os módulos de alto nível aqueles que contem as políticas de decisão e negócios mais importantes da aplicação. Se esses módulos dependem de módulos de menor nível, mudanças em baixo nível acarretam em impactos diretos sobre eles, forçando-os a mudar. Além disso, [Dip2009] considera que são os módulos de alto nível que interessam nas políticas de reusabilidade e quando módulos de alto nível dependem de módulos de baixo nível, tona-se extremamente difícil reutilizá-los em diferentes contextos.

A Figura 2.16 fornece um modelo mais adequado para o problema da classe *Copy* apresentado. Nela, o problema da dependência hierárquica modular é sanado pela utilização de abstrações, representadas pelas elipses, que permitem que o módulo *Copy* se torne reutilizável independentemente do contexto já que *Copy*

passa a depender apenas das abstrações *Reader* e *Writer* e os mecanismos detalhados (*Keyboard Reader* e *Printer Writer*) dependem dessas mesmas abstrações. Por esse modelo, não importam quantos mecanismos de leitura e escrita sejam criados, desde que estes derivem das abstrações *Reader* e *Writer*, *Copy* não é afetado e sua reusabilidade fica garantida.

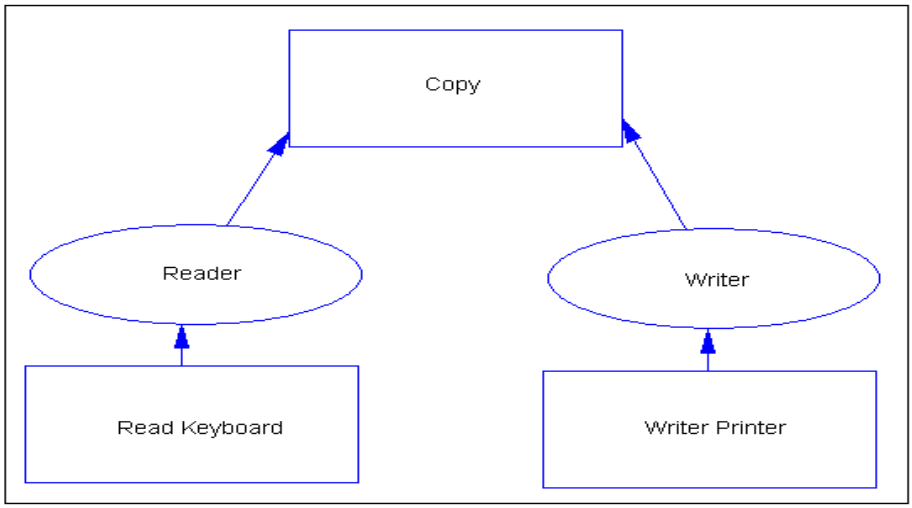


Figura 2.16 – Hierarquia modular adequada ao DIP. [Dip2009]

[Dip2009] diz ainda que o DIP está na raiz de muitos dos aclamados benefícios da orientação a objetos e sua aplicação apropriada se faz necessária na construção de módulos e *framewoks* reutilizáveis. O DIP tem importância crítica na construção de melhores códigos, que sejam resistentes aos contras decorrentes das mudanças e que, uma vez que as abstrações e detalhes são isolados uns dos outros, sejam muito mais fáceis de serem mantidos.

2.2.3 Considerações Finais

Neste capítulo foram apresentadas as áreas de conhecimento da Engenharia de *Software* com maior impacto no artefato código-fonte de um *software*, **Projeto e Construção de *Softwares***. Este conhecimento se faz necessário para que se possam compreender quais são os grandes vilões por trás dos problemas comumente encontrados nos *softwares* legados e desenvolvidos atualmente que impactam diretamente na qualidade dos sistemas vendidos.

Como qualidade passou a ser um termo essencial em qualquer negócio, buscar melhorar os processos continuamente se faz vital para o sucesso de qualquer empreendimento. Especificamente no contexto de *software*, entender o processo, bem como enxergar formas de melhorá-lo, tem sido uma tarefa recorrente em todas as empresas e pode-se dizer que é consenso que desenvolver e entregar um bom *software* passa, vital e criticamente, pelo código-fonte que é construído, e que o código é a alma de um projeto bem sucedido.

Sendo assim, conhecer, compreender e aplicar de forma correta os princípios e fundamentos do Projeto e Construção de *Software* é o primeiro passo para se garantir a qualidade de um sistema.

Este trabalho se concentra na aplicação dos *Princípios S.O.L.I.D. de Design em Orientação a Objetos* na busca da melhoria do código-fonte e consequentes impactos nas características de qualidade agregadas ao *software*, principalmente no que diz respeito aos problemas recorrentes oriundos de um mau projeto e construção do sistema.

3 Metodologia

Este capítulo é dedicado ao detalhamento da metodologia aplicada a este projeto, apresentando as características do *software* base (SisBase), bem como do *software* desenvolvido (SisDev), detalhando as ferramentas utilizadas e apresentação, e breve descrição, das métricas selecionadas para a análise da qualidade do produto obtido.

3.1 Do escopo do projeto

O objetivo do projeto foi, através da aplicação dos "*Princípios S.O.L.I.D.de Design*" no desenvolvimento de um sistema orientado a objetos a partir de um documento de requisitos pré-definido, identificar os possíveis benefícios que estes princípios podem agregar à qualidade do código-fonte gerado, considerando métricas de código para tal.

O processo de análise desses possíveis benefícios foi baseado na medição do *software* produzido, através da ferramenta de apoio *Google CodePro Analityx*, que será descrita em seção subsequente, e utilizou uma série de métricas de qualidade de *software* fornecidas pela ferramenta de medição utilizada e selecionadas de acordo com o alvo da análise, que foi o código-fonte. Essas métricas, bem como suas formas de medição e ranqueamento serão descritas em seções posteriores ainda neste capítulo.

3.2 Do *software* base

O *software* base (SisBase) utilizado para comparação neste projeto foi proposto como parte da disciplina GCC110 – Programação Orientada a Objetos, ministrada pelo Departamento de Ciência da Computação da Universidade Federal de Lavras, à época, sob responsabilidade do Prof. Dr. Cristiano Leite de Castro, orientador deste projeto, que selecionou um dos trabalhos realizados para servir de objeto de comparação para o objetivo deste projeto.

A proposta do sistema base é simular um sistema de uma operadora de telefonia móvel, utilizado para manter dados sobre clientes, planos, promoções e telefones habilitados, bem como realizar operações sobre esses dados, como cadastros para os objetos envolvidos e operações sobre os mesmos (incluir créditos e promoções a um dado celular, realizar ligações levando-se em conta as promoções habilitadas, controle de validade sobre créditos e promoções, etc.), além de manter um histórico de dados que fica disponível para consultas.

Foi definido um conjunto de regras pertinentes a cada objeto envolvido no sistema que são levadas em conta durante as operações disponíveis, como promoções disponíveis apenas para certo tipo de telefone habilitado, modelos de consumo de créditos baseado no tipo de plano e promoções habilitadas, etc., e que definem um conjunto de operações que são exercidas pelo sistema, garantindo um ciclo funcional para as atividades da operadora em questão (o documento de requisitos pode ser conferido na íntegra no Anexo A deste trabalho).

O *software* base foi desenvolvido sob uma perspectiva "clássica" de desenvolvimento, baseada apenas nos conhecimentos do seu desenvolvedor, não apresentando características de aplicação de nenhum recurso ou metodologia que visasse o melhoramento do sistema implementado, exceto pela presença do padrão de projeto *Factory Method* que, segundo [Freeman2007], define uma interface para criar um objeto, mas permite às classes decidir qual classe instanciar, ou seja, o *Factory Method* permite a uma classe deferir a instanciação para subclasses.

O padrão *Factory Method* foi utilizado em casos onde a utilização de herança seria uma alternativa de implementação. Para [GOF1995], o *Factory Method* deve ser utilizado quando:

- Uma classe não pode antecipar a classe de objetos que deve criar;
- Uma Classe quer que suas subclasses especifiquem os objetos que criam;
- Classes delegam responsabilidade para uma dentre várias subclasses auxiliares, e você quer localizar o conhecimento de qual subclasse auxiliar que é a delegada.

Nos demais aspectos, o *software* base não fez uso de nenhum outro recurso aparente que impactasse em aspectos de implementação, como *frameworks* ou ferramentas de apoio.

3.3 Das ferramentas utilizadas

Tanto o *software* base quanto o desenvolvido neste projeto foram implementados em linguagem Java e baseados nos princípios de Orientação a Objetos.

Este projeto se utilizou da ferramenta Eclipse (http://www.eclipse.org/) como *IDE* (*Integrated Development Environment*) e foi desenvolvida sobre a versão sete (7) do Java. Nenhum framework adicional foi utilizado devido à característica simples do sistema base (que também não fez uso de frameworks).

Foi adicionado o plug-in da ferramenta *Google CodePro Analityx* (https://developers.google.com/java-dev-tools/download-codepro?hl=pt-BR). A escolha dessa ferramenta se deu devido à sua fácil integração com a *IDE* escolhida e por sua facilidade de manuseio, fornecendo um conjunto de métricas que atendiam às necessidades do projeto.

3.3.1 Google CodePro Analityx

[AnalityX] define o *Google CodePro Analityx* como sendo uma solução líder na indústria de *software*, utilizada na detecção e correção de problemas ligados à qualidade de *software*, normalmente enfrentados durante o processo de desenvolvimento. Esta solução foi projetada com o intuito de auxiliar os desenvolvedores e organizações preocupadas com questões como revisão e dependência de código,

fornecendo mecanismos de auditoria e mensuração de código, geração de testes, cobertura de código, etc..

Dentre as funcionalidades disponíveis nessa ferramenta, este trabalho destaca a computação de métricas, recurso este essencial para a execução do objetivo analítico do projeto. Este recurso, de acordo com o documentado em [AnalityX], pode fornecer informações vitais sobre o código implementado e é capaz de exportar relatórios em formato *HTML (HyperText Markup Language)* visando facilitar a distribuição dessas informações.

As Figura 3.1 e Figura 3.2, a seguir, apresentam dois formatos de visualização das métricas computadas pela ferramenta, sendo a Figura 3.1 a representação da exibição na *IDE* e a Figura 3.2 a representação do relatório *HTML* suportado.

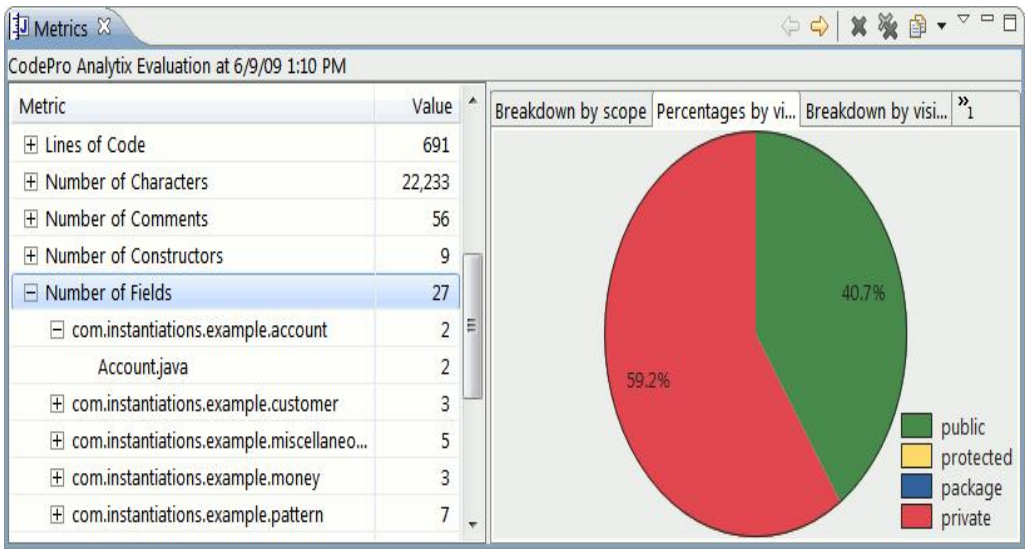
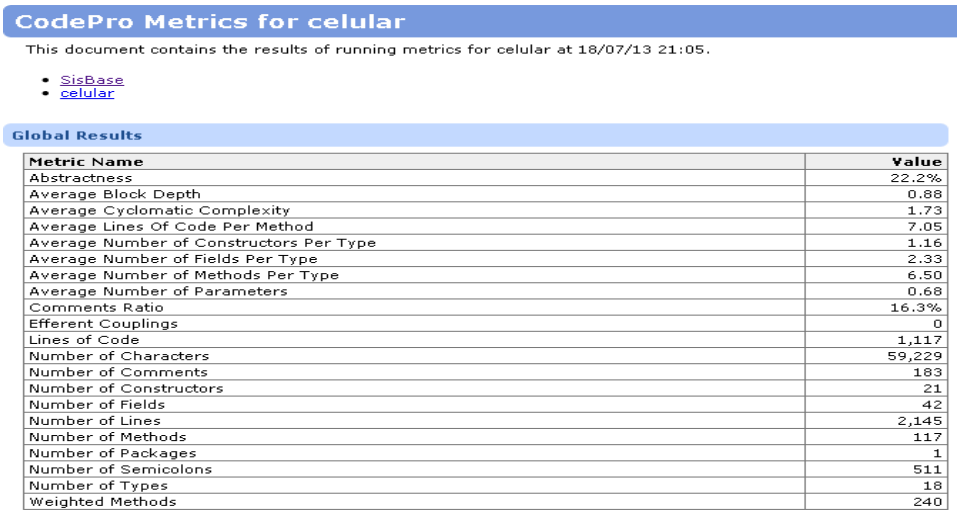


Figura 3.1 – Relatório de métricas – *Google CodePro Analityx*



CodePro Metrics for celular

This document contains the results of running metrics for celular at 18/07/13 21:05.

- SisBase
- celular

Global Results

Metric Name	Value
Abstractness	22.2%
Average Block Depth	0.88
Average Cyclomatic Complexity	1.73
Average Lines Of Code Per Method	7.05
Average Number of Constructors Per Type	1.16
Average Number of Fields Per Type	2.33
Average Number of Methods Per Type	6.50
Average Number of Parameters	0.68
Comments Ratio	16.3%
Efferent Couplings	0
Lines of Code	1,117
Number of Characters	59,229
Number of Comments	183
Number of Constructors	21
Number of Fields	42
Number of Lines	2,145
Number of Methods	117
Number of Packages	1
Number of Semicolons	511
Number of Types	18
Weighted Methods	240

Figura 3.2 – Relatório de métricas *HTML – Google CodePro Analityx*

Na Figura 3.1 temos, à esquerda, uma tabela onde são apresentadas as métricas previstas pelo *Google CodePro Analityx* e seus respectivos valores, calculados à partir de parâmetros nativos da ferramenta. À direita, temos uma representação gráfica da métrica destacada na tabela fornecida (*Number of Fields*, no caso, que calcula o número total de atributos criados no conjunto analisado). São apresentados gráficos para todas as métricas previstas e, vale destacar, o relatório do *plug-in* permite a navegação para níveis mais específicos das métricas em relação ao projeto, sendo possível ter uma dimenção tanto global (medida sobre o pacote como um todo) quanto local (medida sobre cada classe do pacote).

A Figura 3.2 mostra um das seções do relatório *HTML* gerado pelo *plug-in*, que traz algumas tabelas informativas sobre os resultados da metrificação, bem como informações sobre violações na computação das métricas. Este relatório também dá o dimensionamento global e local sobre as métricas e pacotes selecionados.

A configuração dos parâmetros para o cálculo das métricas, bem como o conjunto de métricas a ser utilizado, podem ser alterados através das funções disponíveis na ferramenta (Figura 3.3).

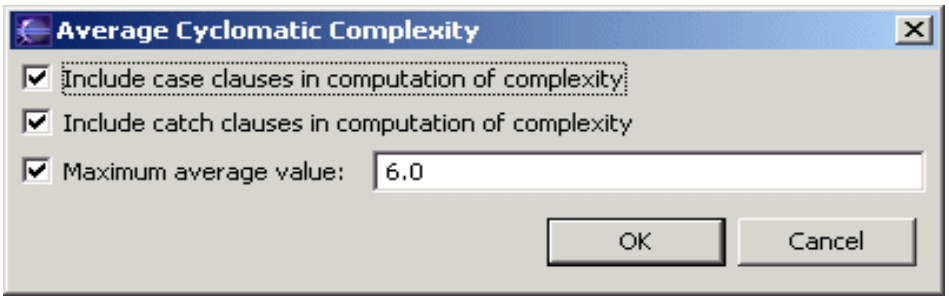


Figura 3.3 – Opções de configuração de métricas [Analityx]

3.4 Do sistema implementado

O SisDev foi desenvolvido sem nenhum contato com o código-fonte do SisBase, tendo apenas o documento de requisitos e o diagrama de classes para auxiliar no mapeamento de recursos e requisitos. Dessa forma, buscou-se garantir um maior nível de independência entre os sistemas estudados.

A Figura 3.4 a seguir, traz os resultados do cálculo das métricas para o SisBase, enquanto a Figura 3.5 traz os resultados para o SisDev (os valores mostrados serão discutidos no Capítulo 5).

Metric	Value
Average Block Depth	0.98
<default>	3.57
celular	0.88
Average Cyclomatic Complexity	2.69
<default>	21.57
celular	1.73
Average Lines Of Code Per Method	11.75
<default>	104.57
celular	7.05
Average Number of Methods Per Type	4.57
<default>	7.00
celular	4.44
Efferent Couplings	18
<default>	1
celular	17
Lines of Code	1,858
<default>	741
celular	1,117
Weighted Methods	391
<default>	151
celular	240

Figura 3.4 – Relatório de métricas SisBase – ***Google CodePro Analityx***

Metric	Value
Average Block Depth	0.88
entidade	0.72
service	1.76
Average Cyclomatic Complexity	1.19
entidade	1.00
service	1.96
Average Lines Of Code Per Method	4.41
entidade	3.49
service	8.00
Average Number of Methods Per Type	1.76
entidade	0.33
service	5.20
Efferent Couplings	17
entidade	12
service	5
Lines of Code	702
entidade	444
service	258
Weighted Methods	152
entidade	101
service	51

Figura 3.5 – Relatório de métricas SisDev – ***Google CodePro Analityx***

A ausência de métricas destacadas em vermelho (como pode ser percebido na Figura 3.4 em relação à Figura 3.5) nos permite ter uma ideia, mesmo que intuitiva, dos ganhos obtidos neste trabalho.

Os *"Princípios S.O.L.I.D.* de *Design"* estudados permitiram ao produto de *software* construído apresentar um comportamento indicativo da melhora esperada, porém, vale destacar que, enquanto princípio, os conceitos abordados têm um caráter abstrato que deve ser levado em conta ao se considerar a eficiência e amplitude da aplicação dos mesmos no desenvolvimento.

Um exemplo da aplicação dos princípios estudados neste trabalho é mostrado na Figura 3.6 a seguir. Nela, temos a clara divisão de responsabilidades (Princípio de Responsabilidade Única - *SRP*) para o componente *Celular* do sistema, onde as responsabilidades estão dividas entre a classe de dados (Entidade), cuja função é limitada ao armazenamento de informações, e a classe de serviço (Service), que é responsável pela manipulação das informações da entidade visando a atuação funcional do componente *Celular* com o sistema.

Uma extensão do conceito envolvido no *SRP* pode ser aplicado a nível de método, tornando o papel funcional de cada operação responsável por apenas uma ação específica e bem definida. Essa extensão pode ser percebida na Figura 3.7.

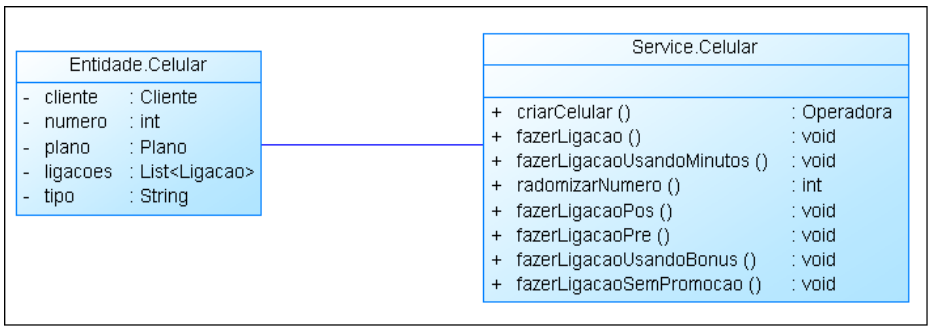


Figura 3.6 – SRP aplicado ao componente *Celular* – SisDev

```
public Operadora criarCelular(String tipo, Plano plano, ArrayList<Promocao> promocoes, Operadora operadora){

    Celular cel = new Celular(tipo, plano, promocoes, randomizarNumero());
    operadora.getCelulares().add(cel);
    return operadora;
}
```

Figura 3.7 – SRP a nível de método – SisDev

Devido à simplicidade do sistema implementado, alguns conceitos envolvidos nos princípios estudados não encontraram espaço para serem aplicados, tais como os conceitos envolvidos em ISP, já que em um sistema desse porte, a utilização de interfaces não é justificada, devido à simplicidade e funções bem definidas para cada tipo construído. Insistir no uso de interfaces apenas para insinuar o emprego deste princípio seria uma violação da seriedade deste trabalho, além do que isso poderia influenciar negativamente a computação das métricas. O emprego de especializações para classes abstratas pode ser visto como uma extensão dos conceitos de ISP, permitindo o aumento da coesão entre os contextos em que tais especializações são aplicadas.

Outros conceitos tiveram suas bases abstraídas para outros níveis, de forma a agregar valor à proposta deste estudo, casos do SRP (tal como descrito anteriormente), do DIP, cuja essência em inverter dependências foi aplicada em nível de atributos, onde, por exemplo, as dependências do tipo Operadora para com os tipos Plano e Promocao, que são classes abstratas. Os conceitos do princípios OCP estão representados também nos modelos de abstração utilizados, bem como no conceito de modularização empregado (que também aborda as bases do SRP), onde foram separadas classes de dados (Entidades) das classes de negócio (Services).

3.5 Das Dificuldades encontradas

Como em todo processo de desenvolvimento, é necessário ter a sensibilidade e percepção de que nem todos os recursos disponíveis são realmente aplicáveis e/ou benéficos, ou seja, é importante ter em mente que ao se pensar em melhorar ou aperfeiçoar artefatos, deve-se levar em consideração os diversos aspectos pertinentes a esta etapa, bem como visualizar as necessidades presentes e futuras que possam vir a justificar a utilização de certos recursos, como padrões de projeto, modelos de arquiteturas e técnicas de implementação. [Aniche2012] já destacava que esse tipo de desatenção pode ocasionar problemas de Projeto de *Software* (como os descritos nas primeiras seções do Capítulo 3) oriundos de um longo processo de projeto (*up-front design*), onde os projetistas tentam antecipar requisitos futuros do sistema e criam “um projeto de classes demasiadamente flexíveis ou desnecessariamente sofisticadas” ([Aniche2012] Cap. 13).

Com os *"Princípios S.O.L.I.D.* de *Design"* não foi diferente. [Kk2009], por exemplo, já destacava que o princípio da Substituição de Liskov (*LSP*) não é suportado "naturalmente" pela linguagem Java por ser baseado em "uma relação entre pré-condições e pós-condições associadas a um relacionamento de herança" não suportado pela linguagem, o que tornaria a detecção das violações características do *LSP* algo muito difícil.

Outra dificuldade que merece destaque é o nível de abstração necessário para se atacar um problema sob orientação desse tipo de princípios, uma vez que a ideia de princípios remete a um nível bem anterior à implementação propriamente dita, estando relacionada aos conceitos sob os quais o desenvolvedor enxerga e ataca o problema proposto. Sendo assim, contornar a concretude da vivência adquirida para aplicação de outra perspectiva de desenvolvimento pode ser algo muito difícil.

Manter a disciplina é um aspecto chave, principalmente quando os resultados não são tão claros e imediatos quando se utiliza uma abordagem técnica, onde existe todo um roteiro a ser seguido e um mapeamento claro da forma pelas quais os resultados se apresentam. Essa talvez tenha sido a maior dificuldade encontrada.

3.6 Das Métricas Estudadas

Levando-se em consideração o disposto na seção **2.3 Métricas de Código** deste trabalho, bem como os recursos fornecidos pela

ferramenta de análise escolhida, este trabalho propõe a avaliar e analisar as métricas relativas ao tamanho do código, à complexidade e à coesão e acoplamento, uma vez que esses quesitos estão diretamente relacionados às definições de código de qualidade ([Swebok2004] e [Janzen2008]) e têm um grande impacto nas características de manutenibilidade do produto de *software* ([Swebok2004] e ISO/IEC 9126-1).

Assim, as métricas listadas a seguir foram computadas para o sistema desenvolvido neste trabalho, bem como para o sistema base, tendo estas sido selecionadas entre aquelas presentes na ferramenta *CodePro Analityx* e induzidas pelo trabalho de [Janzen2008], que, mesmo abordando uma técnica diferente da estudada aqui, buscava conclusões sobre os mesmos aspectos tratados neste projeto. As métricas selecionadas foram:

- **Profundidade média de bloco (*Average Block Depth - Complexity*):** utilizada para identificar métodos e construtores com muitos níveis de blocos aninhados, o que pode dificultar a legibilidade e compreensão do código [AnalityX];

- **Complexidade ciclomática média (*Average Cyclomatic Complexity - Complexity*):** calcula a média entre as complexidades ciclomáticas dos métodos definidos em todo o conjunto de classes examinado, sendo o termo "complexidade ciclomática", a medição do número de caminhos de execução possíveis dentro de cada método [AnalityX];

- **Média de linhas de código por método (*Average Lines of code per method - Basic*):** calcula o número médio de linhas

de código (*LOC*) dos métodos definidos no conjunto analisado [AnalityX];

- **Média de métodos por tipo (*Average number of methods per type - Basic*):** calcula o número médio de métodos definidos em todo o conjunto de classes analisado [AnalityX];

- **Linhas de código (*Lines of code - Basic*):** contador para as linhas de código presentes no conjunto analisado, não se limitando apenas aos métodos definidos, como ocorre na "Média de linhas de código por método" [AnalityX];

- **Acoplamentos eferentes (*Efferent couplings - Dependency*):** refere-se ao número de tipos definidos no conjunto analisado que dependem de outros tipos. Essa medida leva em consideração as dependências entre classes internas ao projeto analisado, bem como referências externas, por meio de *"imports"* [AnalityX];

- **Métodos Ponderados (*Weighted methods - Complexity*):** calcula a soma ponderada de todos os métodos do conjunto analisado, permitindo uma visualização tanto global quanto por classe, caso este conhecido como "Métodos Ponderados por Classe" ou *WMC – Weighted Methods per Class* [Meirelles2008];

4 Analise e Discussão dos Resultados

Este capítulo tem por finalidade trazer uma discussão sobre os resultados e análise das métricas obtidas visando captar os indícios base para a conclusão sobre a proposta deste trabalho. Aqui, são apresentados gráficos comparativos entre os dois sistemas e o raciocínio sobre o impacto que os resultados trouxeram sobre a qualidade do sistema desenvolvido. Também são apresentados argumentos que justificam a interpretação dada a cada resultado.

4.1 Dos Resultados Obtidos

A Figura 4.1, a seguir, traz os resultados da mensuração de ambos os sistemas analisdos. A tabela representada traz, na coluna à esquerda, uma "árvore" com as métricas representadas na raíz e cada sistema como "folhas" dessas métricas. Na coluna à direita, a tabela traz os valores calculados, sendo que o valor à frente da métrica representa a medida global, ou seja, a média da métrica sobre o conjunto analisado. Os valores à frente de cada sistema trazem uma média local, calculada a partir do conjunto de classes de cada sistema.

Os "nós" destacados em vermelho representam as métricas que excederam o valor máximo tolerado pela ferramenta (os valores máximos de cada métrica, assumido nesta análise, foram definidos por padrão na ferramenta). Podemos perceber que o SisDev não apresentou nenhuma métrica valorada acima do máximo permitido, enquanto o SisBase foi responsável pelo "estouro" do limite em quatro das sete métricas estudadas.

Metric	Value
Average Block Depth	0.93
SisBase	0.98
SisDev	0.88
Average Cyclomatic Complexity	1.99
SisBase	2.69
SisDev	1.19
Average Lines Of Code Per Method	8.33
SisBase	11.75
SisDev	4.41
Average Number of Methods Per Type	3.25
SisBase	4.57
SisDev	1.76
Efferent Couplings	35
SisBase	18
SisDev	17
Lines of Code	2,560
SisBase	1,858
SisDev	702
Weighted Methods	543
SisBase	391
SisDev	152

Figura 4.1 – Relatório de métricas sobre SisBase e SisDev

Um detalhe que merece destaque é que o SisBase apresenta uma classe responsável pela interação com o usuário (denominada Interface.java) que foi considerada no processo de metrificação devido ao fato de que, apesar de sua função estar relacionada a um modelo de interação não implementado para o SisDev, ela apresenta um grande número de processos lógicos envolvidos nos requisitos do sistema proposto. Assim, é válido destacar que os resultados dispostos para o SisBase neste capítulo tem grande influência negativa dessa classe. O Anexo B traz uma visão da análise para o SisBase desconsiderando essa classe Interface.java onde é possível perceber a alteração dos valores das métricas. Também é válido destacar que o efeito gerado pela presença ou não dessa classe Interface.java não altera os resultados obtidos (apenas diminui a diferença entre os valores calculados para algumas métricas).

4.1.1 Análise dos resultados

O estudo das métricas computadas para o SisBase e SisDev foi realizado sob três perspectivas fundamentais ao se tratar de qualidade de código fonte: 1- Tamanho do código, 2- Complexidade e 3- Acoplamento. Tendo essas perspectivas bem definidas, as métricas selecionadas foram agrupadas para facilitar a visualização do impacto da metodologia propostas em cada um desses aspectos de forma que cada um desses grupos ficou composto como segue:

1. Tamanho do Código: representado pelas métricas básicas fornecidas pela ferramenta de análise, o *CodePro Analityx*. [Janzen2008] define as métricas referentes ao tamanho do código como sendo as mais simples. As métricas agrupadas aqui são:
    a. Média de linhas de código por Método (*Average Lines Of Code Per Method*);
    b. Média de métodos por tipo (*Average Number of Methods Per Type*);
    c. Linhas de Código (*Lines of Code*);
2. Complexidade: representado pelas métricas de complexidade fornecidas pela ferramenta de análise. Medidas de complexidade incluem a contagem de caminhos independentes através do código (complexidade ciclomática) e o grau de aninhamento presento no código (profundidade de bloco) [Janzen2008]. De certa forma, podemos incluir o

tamanho do código como um fator influente nas métricas de complexidade já que, em geral, classes (e/ou métodos) menores tendem a ser mais simples e mais fáceis de serem compreendidos. As métricas presentes neste agrupamento são:

a. Profundidade média de bloco*(Average Block Depth*);

b. Complexidade Ciclomática média (*Average Cyclomatic Complexity*);

c. Métodos Ponderados (Weighted Methods);

3. Acoplamento: representado pelas métricas de dependência fornecidas pela ferramenta de análise. Acoplamento é um dos princípios de Projeto de *Software* previstos em [Swebok2004] e cuja desatenção acarreta nos sintomas de um código ruim descritos por [Martin2000], ambos alvos do estudo aqui realizado. As métricas selecionadas para este agrupamento são:

a. Acoplamentos Eferentes (*Efferent Couplings*);

As seções a seguir trazem a análise para cada uma dessas perspectivas citadas.

4.1.1.1 Métricas de Tamanho do código

As figuras a seguir (Figura 4.2, Figura 4.3 e Figura 4.4) trazem os gráficos comparativos para as métricas de tamanho de código estudadas. Elas permitem visualizar, separadamente, a melhoria conseguida com a metodologia utilizada, retratando a resultante de um

código mais simplificado, com métodos bem definidos e mais coesos, e classes especializadas de acordo com as responsabilidades definidas.

Tais aspectos são uma influência direta da atuação do princípio SRP, bem como sua extensão para nível de método, o que permitiu a construção de classes e métodos com responsabilidades isoladas e, consequentemente, menores.

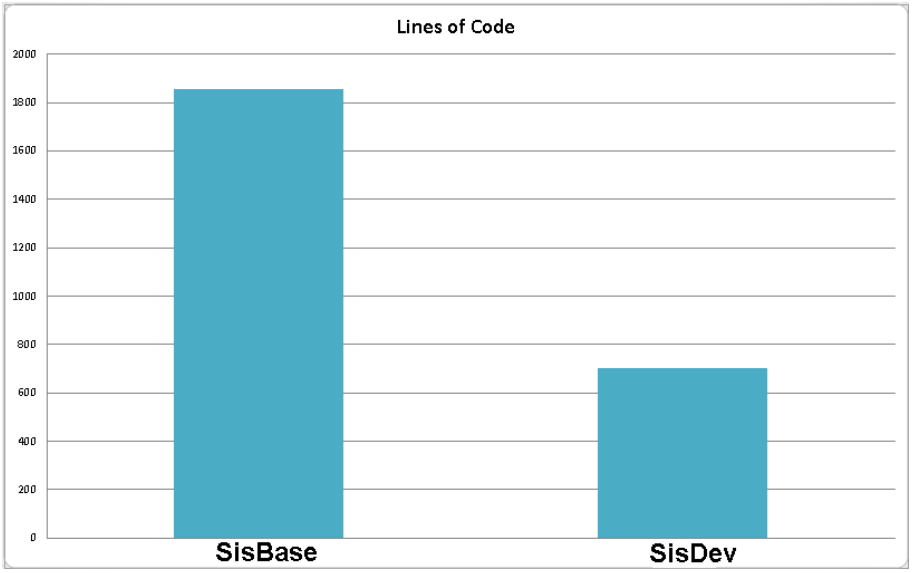


Figura 4.2 – Gráfico - Resultado para *Lines of Code*

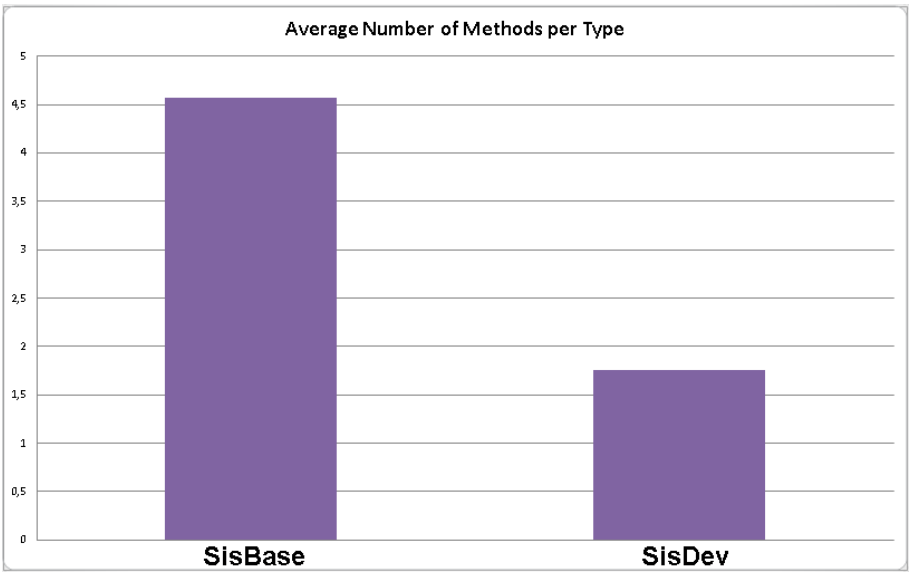


Figura 4.3 – Gráfico - Resultado para *Average Number of Methods per Type*

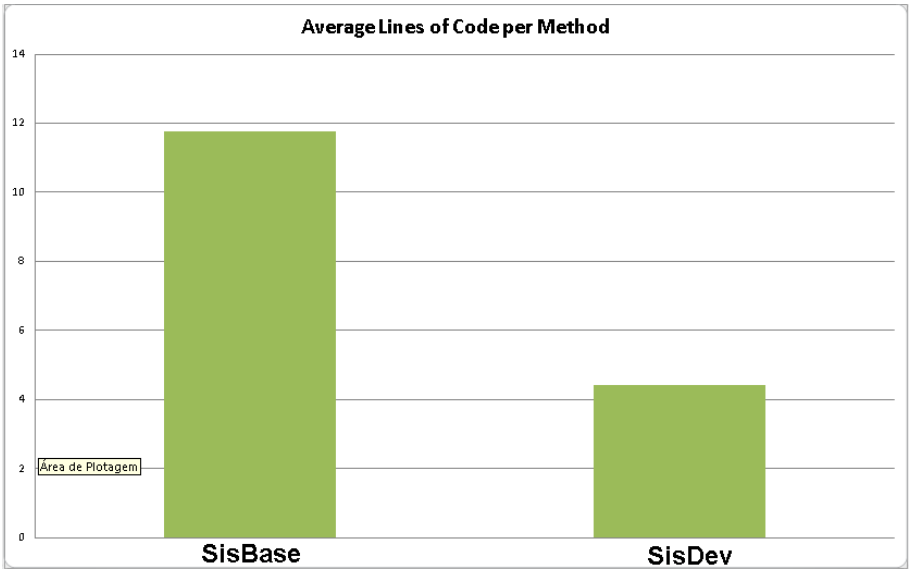


Figura 4.4 – Gráfico - Resultado para *Average Lines of Code per Method*

Essa divisão de responsabilidade somada à modularização aplicada em nível de projeto é responsável pela diferença significativa observada para essas métricas, dando destaque para o Número Médio

de Métodos por Tipo (*Average Number of Methods per Type*), onde, para o SisBase o resultado foi de 4,57, enquanto para o SisDev, esse valor foi de 1,76 (essa métrica foi configurada para desconsiderar os métodos de acesso – *get* e *set*).

Podemos observar o reflexo da aplicação da metodologia na métrica Média de Linhas de Código por Método (*Average Lines of Code per Method*) cuja diferença (11,75 para o SisBase, contra 4,41 para o SisDev) representa a extensão do conceito do SRP para o nível de método, o que refletiu na criação de métodos mais especializados, menores e mais coesos.

Um detalhe interessante aqui é que, contrariando um pouco a tendência inicial, onde a criação de métodos menores para a separação de responsabilidades sugere a criação de mais métodos, a modularização aplicada permitiu reduzir o número de métodos por tipo juntamente com o número de linhas de código para cada um dos métodos.

Os gráficos dão a clara visão do benefício que a aplicação dos princípios estudados agregou ao objetivo deste projeto, que era a questão da produção de código simples. Outro fator que merece destaque acerca do tamanho do código foi a abordagem de modularização (pacotes de Entidade e Service, sendo o primeiro responsável por abrigar as classes de dados, e o segundo por abrigar classes de serviço, responsável pela lógica modular) na aplicação dos princípios, o que permitiu uma separação ainda mais profunda de responsabilidades. Essa modularização é um princípio de Projeto de *Software* que também agrega valor ao código-fonte [Swebok2004].

4.1.1.2 Métricas de Complexidade

As figuras a seguir (Figura 4.5, Figura 4.6 e Figura 4.7) trazem os gráficos para as métricas de complexidade estudadas. Mais uma vez é visível a melhoria conseguida através da aplicação da metodologia proposta. Aqui existe um reflexo das melhorias analisadas na seção anterior, uma vez que a criação de classes e métodos com responsabilidades bem definidas e coesas, geralmente, resultam em um menor grau de complexidade [Janzen2008].

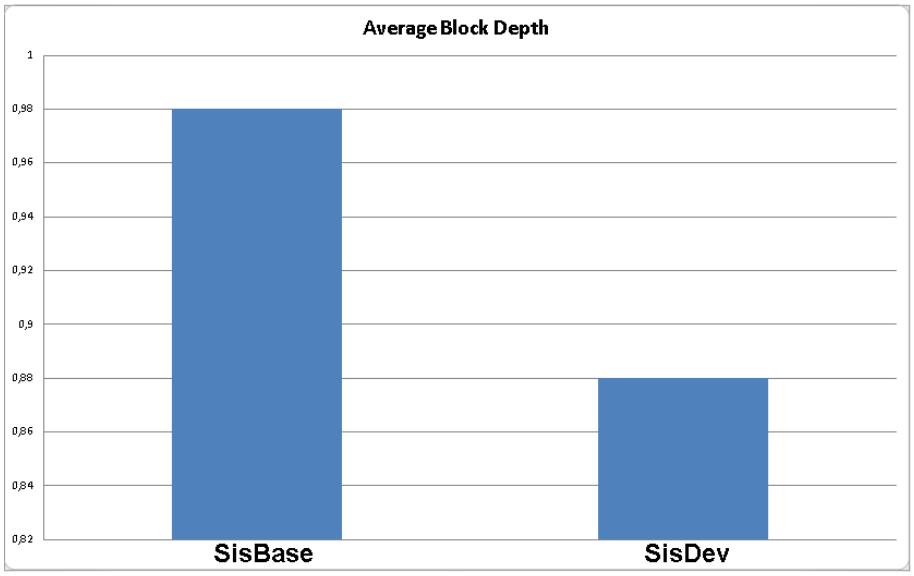


Figura 4.5 – Gráfico - Resultado para *Average Block Depth*

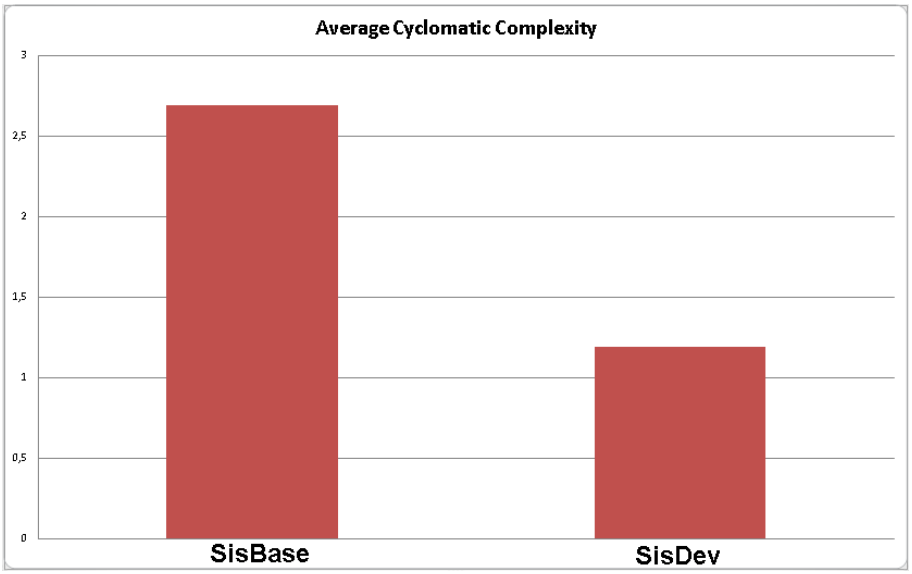


Figura 4.6 – Gráfico - Resultado para *Average Cyclomatic Complexity*

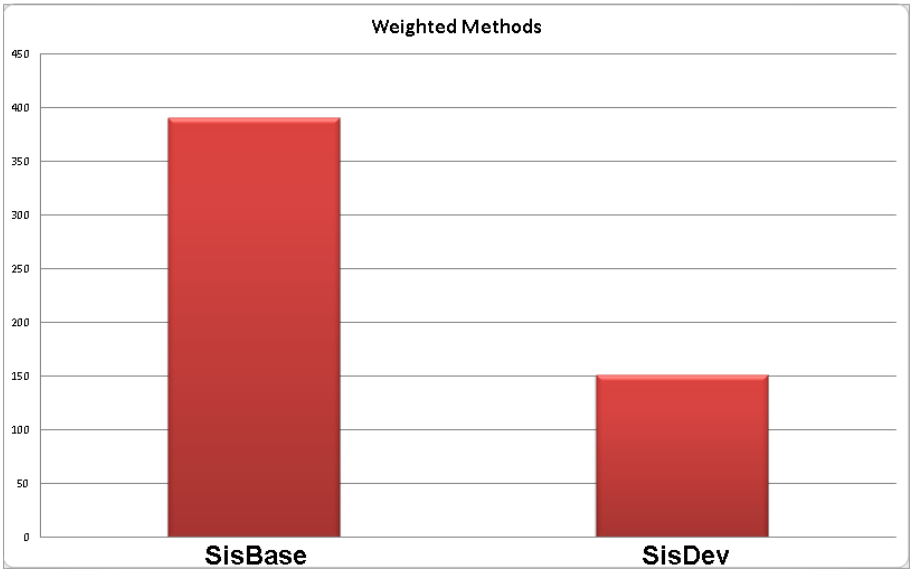


Figura 4.7 – Gráfico - Resultado para *Weighted Methods*

Os gráficos apresentados, mais uma vez, apontam uma melhora considerável do SisDev em relação ao SisBase, mais uma vez

justificável pela característica por trás do ideal de modularização e separação de responsabilidades que, aliados à essência do OCP, que prega a capacidade de extensão modular sem modificação do esqueleto, garantiram a abrangência dos métodos sem acarretar o aumento da complexidades e profundidade de bloco dos mesmos, tendo reduzido, em mais de 50%, os valores para as métricas avaliadas.

4.1.1.3 Métricas de Acoplamento

A Figura 4.8 traz o gráfico para a métrica de Acoplamento estudada, *Efferente Coupling*, onde mais uma vez é possível perceber algum ganho, mesmo que esta reflita a menor diferença, entre as métricas estudadas.

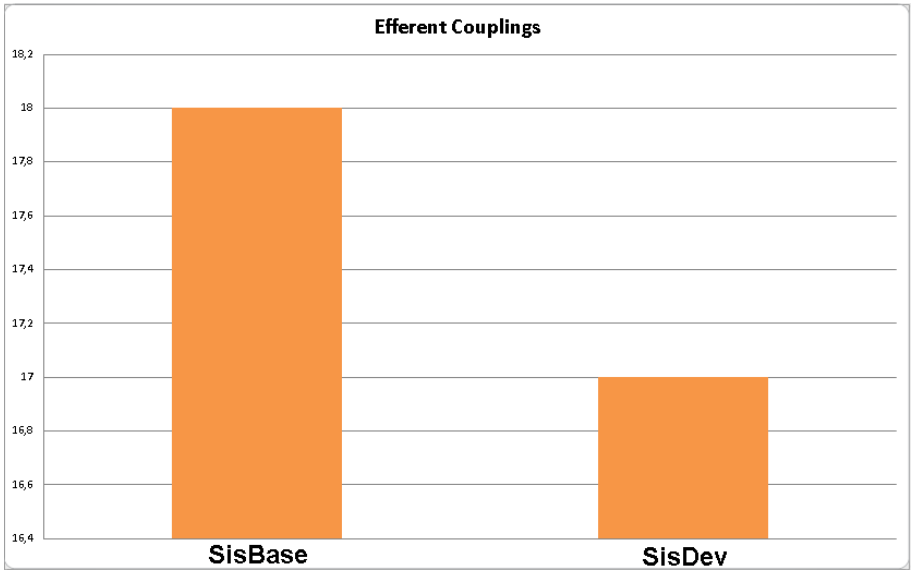


Figura 4.8 – Gráfico - Resultado para *Efferent Couplings*

A diferença entre os valores encontrados para essa métrica (18 para o SisBase e 17 para o SisDev) pode ser justificada pela diferença no número de classes de cada sistema e pelo nível de acoplamento que classes mais especializadas (SRP + OCP + modularização) pode gerar entre seus métodos.

Não foi encontrada uma causa razoável o suficiente para ser apontada como raiz para essa pequena diferença em relação a esta métrica, porém como a diminuição do acoplamento modular é um dos objetivos dos princípios analisados, essa visão de melhora se faz importante para o contexto analisado neste trabalho.

4.2 Considerações Finais

Após a computação das métricas selecionadas e análise dos resultados, ficou claro que o emprego de um conceito de melhoria de código, como os princípios estudados neste trabalho, agrega muito valor na qualidade final do código-fonte produzido.

O processo de análise permitiu a visualização quantitativa dessa margem de melhora obtida quando se tem um modelo que visa a otimização do processo de codificação, teste e correção de *software*, e essa melhora é essencial quando levado em conta o ciclo de vida do produto de *software*, visto que, como mostram os resultados obtidos, há uma redução no grau de complexidade e acoplamento geral do sistema, além da diminuição do tamanho do código e aumento da coesão e, consequentemente, da legibilidade do código produzido. Todos esses fatores, somados, implicam num benefício em termos de

manutenibilidade do sistema, segundo proposto na literatura e nas normas referentes à Qualidade do Produto de *Software.*

Dessa forma, os resultados, apesar de serem estatisticamente inválidos, mostram, com certo grau de clareza, um indicativo satisfatório da eficiência dos "*Princípios S.O.L.I.D. de Design"* em se tratando da melhoria do código-fonte e das características de manutenibilidade de um produto de *software.*

Uma forma de validar estatisticamente esses resultados seria aplicar esta mesma metodologia em um sistema maior ou em um espaço amostral significativo, o que permitiria avaliar com mais precisão o volume de melhora obtida, porém, para os objetivos propostos neste trabalho, esse nível de análise se faz suficiente.

5 Conclusão

Buscar a melhoria da qualidade do produto de *software* é uma tarefa que exige muito mais que apenas vontade por parte das equipes de desenvolvimento. Os processos conhecidos para este fim requerem muita disciplina e conhecimentos de diversos fatores que impactam no produto final e, consequentemente, na qualidade obtida.

O presente trabalho permitiu uma visão mais ampla desse domínio através da revisão de literatura realizada, onde foi possível conhecer as entrelinhas da literatura no que diz respeito aos processos de *software* e ao ideal de qualidade que, mais do que esperado, se tornou algo essencial. A aplicação dos conceitos estudados também permitiu uma abertura de mentalidade, uma vez que os princípios estudados possuíam um forte caráter abstrato, que demandou um nível de compreensão muito além do aspecto técnico, que muitas vezes é levado em conta no mercado tecnológico.

Os "*Princípios S.O.L.I.D. de Design"* propostos por Martin trouxeram uma percepção diferente sobre os problemas que afetam o código-fonte dos sistemas desenvolvidos e consequentemente a qualidade do mesmo, permitindo a visualização de caminhos para evitar esses problemas, mesmo que em uma abordagem profundamente teórica e abstrata, uma vez que, como o próprio nome sugere, "princípios" remetem a um momento muito anterior à implementação em si, estando fortemente ligados aos aspectos conceituais e fundamentais dos processos de Projeto e Construção de *Software.*

O estudo realizado no presente trabalho permitiu observar como tais princípios (ou seus conceitos base) impactam no processo de desenvolvimento de um sistema orientado a objetos no que compete à qualidade do código-fonte produzido e à aspectos de manutenibilidade, tal como previsto pelas normas de qualidade (ISO/IEC) e pelas bases de conhecimento que regem esse momento do ciclo de vida do *Software* (SWEBOK).

Durante a prática metodológica deste trabalho foi possível constatar a dificuldade em se elevar (ou retroceder?) o desenvolvimento de *software* à nível de princípio, já que o recurso proposto neste estudo não é uma metodologia de caráter prático, onde existe um *script* a ser seguido para a obtenção de resultados, como percebemos em técnicas mais concretas como o *Test-Driven Development* (TDD), *Extreme Programming* (XP) e outras metodologias ágeis que tem se disseminado no mercado atual. Como destacado anteriormente, pensar em nível de princípio requer maturidade, disciplina e capacidade de abstração para que se possa visualizar o problema a ser atacado com vias de aplicação dos conceitos que, para o caso do *S.O.L.I.D.*, estavam muito mais ligados às bases conceituais do Projeto de Construção de *Software*, do que à aplicação técnica de uma linguagem ou abordagem de desenvolvimento.

Esta, sem dúvida alguma, foi a grande dificuldade encontrada neste trabalho, pois atacar os conceitos por trás do pensamento lógico envolvido no desenvolvimento de sistemas se torna muito mais difícil quando já se tem uma vivência, uma experiência prática que não tenha sido concebida sob esses conceitos e forma de pensamento. Então, é

importante ressaltar que quando se pretende estudar uma abordagem baseada em conceitos muito teóricos e, consequentemente, muito abstratos, deve-se estar preparado para enfrentar a resistência da própria mente em mudar essa base conceitual pré-existente.

Também é importante destacar que, tendo sido feita a opção por este tipo de estudo, a escolha do escopo do sistema a ser estudado deve ser feita com muito cuidado uma vez que conceitos são criados sobre objetos e situações genéricas, nem sempre sendo possível uma aplicação concreta e correta. Neste sentido também se faz importante ter a sensibilidade de perceber esse tipo de detalhe e ter a capacidade de abstrair além dos problemas, trazendo os conceitos estudados para um nível mais próximo da realidade estudada (como foi feito neste trabalho com os conceitos do SRP, estendido em nível de método, do DIP, estendido em nível de atributos, e do OCP, aplicados com o conceito de modularização).

A sutileza envolvida no estudo e aplicação de princípios como o *S.O.L.I.D.* também é um fator que merece um parágrafo, principalmente porque os resultados da sua aplicação não são tão óbvios quando não se está familiarizado com o ambiente de aplicação dos mesmos, o que pode gerar uma sensação de frustração e pensamento errôneo sobre a eficiência do estudo e sua capacidade de atingir os objetivos desejados. Neste trabalho mesmo, só foi possível uma visualização de benefício ao se computar e analisar as métricas, já na fase final do projeto.

A criação de código realizada neste estudo e a análise sobre esse código possibilitou visualizar indícios muito positivos, através de métricas de *software* reconhecidas e computadas com o auxílio de

uma ferramenta reconhecida no mercado, o *Google CodePro Analityx*, que aponta grandes melhoras nos aspectos de qualidade de código, tais como tamanho, complexidade, acoplamento e coesão, o que permitiu concluir que este tipo de abordagem de desenvolvimento é benéfico para o produto de *software*.

Seguindo a linha apontada pelos resultados da computação das métricas selecionadas, conclui-se que os objetivos propostos para esse trabalho foram alcançados, tendo sido gerado um código-fonte mais enxuto, de menor complexidade e, em decorrência desses fatores, de melhor legibilidade, garantindo assim uma melhor qualificação sobre o aspecto de manutenibilidade, outro alvo do presente estudo.

Apesar dos bons resultados alcançados, vale destacar que este estudo não tem peso estatístico, o que inibe uma conclusão categórica sobre o benefício real da metodologia proposta. Também é importante destacar, neste sentido, que o cenário em que este trabalho foi desenvolvido não pode ser considerado o ideal, uma vez que o *software* base utilizado é um sistema pequeno e de caráter didático (proposto como parte da disciplina de Programação Orientada a Objetos), não fornecendo uma base concreta para uma comparação relevante, em termos científicos. Apesar disso, os resultados percebidos podem ser interpretados como um forte indício da melhoria concreta desejada na concepção deste projeto, de forma que se crê que a tendência apontada pelos resultados se mantenha num cenário de estudo estatístico-cientifico.

5.1 Trabalhos Futuros

Como sugestão para trabalhos futuros relacionados a este projeto, fica a ideia da reprodução da metodologia estudada aqui em um cenário estatístico-científico, onde seria possível coletar dados de análise sobre uma base mais concreta e analisável, como um *software* de maior volume de código e/ou relevância para o mercado, permitindo reforçar ou rebater as conclusões apresentadas aqui.

Outra possibilidade seria a comparação desses resultados com uma abordagem mais prática no cenário de desenvolvimento, estudando a eficiência de alguma outra metodologia em atacar os problemas do apodrecimento de código e degradação da qualidade em termos de manutenibilidade e projeto.

Uma terceira possibilidade para um trabalho futuro seria a alteração do cenário proposto pelo sistema implementado, criando uma demanda corretivo-adaptativa que possibilitaria uma análise com enfoque na perspectiva de uma equipe de manutenção, o que permitiria avaliar o benefício prático agregado por este trabalho no aspecto de manutenibilidade do *software*.

Finalizando, um estudo sobre a viabilidade do emprego desses e outros princípios de Projeto e Construção de *Software* no modelo didático-acadêmico poderia ser interessante para a difusão dos conceitos abordados e aclamados pela literatura em Engenharia de *Software*, permitindo uma diminuição do abismo existente entre a mentalidade clássica e as muitas possibilidades de melhorias para os modelos e processos envolvidos no ciclo de vida do produto de *software.*

REFERÊNCIAS BIBLIOGRÁFICAS

NBR ISO/IEC 9126-1: 2001. DISPONÍVEL EM HTTP://WWW.ISO.ORG/ISO/ISO_CATALOGUE/CATALOGUE_TC/CATALOGUE_DETAIL.HTM?CSNUMBER=22749, ACESSADO EM 10/04/2012.

[Swebok2004] IEEE (2004) SWEBOK: GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE

[Duarte2000] DUARTE, K.C.& FALBO, R.A. (2000) UMA ONTOLOGIA DE QUALIDADE DE SOFTWARE

[GALIN2004] GALIN, D. (2004) SOFTWARE QUALITY ASSURANCE – FROM THEORY TO IMPLEMENTATION

[LINDSTROM2003] LINDSTROM, L. & JEFFRIES, R. (2003) EXTREME PROGRAMMING AND AGILE SOFTWARE DEVELOPMENT METHODOLOGIES

[PESCIO1997] PESCIO, C. (1997) PRINCIPLES VERSUS PATTERNS

[MARTIN2000] MARTIN, R. C. (2000) Design PRINCIPLES AND Design PATTERNS

[KK2009] KNOERNSCHILD, K. (2009) SOLID PRINCIPLES OF CLASS Design – DISPONIVEL EM: http://www.kirkk.com/modularity/2009/12/solid-principles-of-class-design/

[SRP2009] MARTIN, R. C. (2009) SRP – THE SINGLE RESPONSIBILITY PRINCIPLE – DISPONIVEL EM: http://www.objectmentor.com/resources/articles/srp.pdf

[OCP1996] MARTIN, R. C. (1996) THE OPEN-CLOSED PRINCIPLE – DISPONIVEL EM: http://www.objectmentor.com/resources/articles/ocp.pdf

[LSP1996] MARTIN, R. C. (1996) THE LISKON SUBSTITUTION PRINCIPLE – DISPONIVEL EM:
http://www.objectmentor.com/resources/articles/lsp.pdf

[ISP2009] MARTIN, R. C. (2009) THE INTERFACE SEGREGATION PRINCIPLE – DISPONIVEL EM:
http://www.objectmentor.com/resources/articles/isp.pdf

[DIP2009] MARTIN, R. C. (2009) THE DEPENDENCY INVERSION PRINCIPLE – DISPONIVEL EM:
http://www.objectmentor.com/resources/articles/dip.pdf

[MARTIN2009] MARTIN, R. C. (2009) THE PRINCIPLES OF OOD - DISPONIVEL EM:
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[MARTIN2012] MARTIN, R. C. & KRIENS, P. (2012) JAVA APLICATION ARCHITECTURE – MODULARITY PATTERNS WITH EXAMPLES USING OSGI

[FREEMAN2007] FREEMAN, E. & FREEMAN, E. (2007) USE A CABEÇA! PADRÕES DE PROJETO, 2° EDIÇÃO REVISADA – EDITORA ALTA BOOKS

[KITCHENHAM1996] KITCHENHAM, B. & PFLEEGER, S. L. (1996) SOFTWARE QUALITY; THE ELUSIVE TARGET

[BEIKO2007] BEIKO, J. L. (2007) FUNCTIONALITY BASED REFACTORING: IMPROVING SOURCE CODE COMPREHENSION

[KAUFMAN2007] KAUFMAN, M. & DORIN, R. (2007) ADOPTING A LIFECYCLE APPROACH TO SOFTWARE QUALITY MANAGEMENT

[JUNG2009] JUNG, C. F. METODOLOGIA APLICADA A PROJETOS DE PESQUISA: SISTEMAS DE INFORMAÇÃO & CIÊNCIA DA COMPUTAÇÃO. TAQUARA, 2009. 1 CD-ROM.

[AGUAYO2005] AGUAYO, M. T., GUERRA, A. C. & COLOMBO, R. M. T. (2005) PROCESSO DE AVALIAÇÃO DA MANUTENIBILIDADE DE PRODUTOS DE SOFTWARE

[Meyer1988] Meyer, B. (1988) Object Oriented Software Construction, Prentice Hall, p 23

[Liskov1988] Liskov, B. (1988) Data Abstraction and Hierarchy, SIGPLAN Notices, p 23,5

[Gof1995] HELM, R.; VLISSIDES, J.; JOHNSON, R.; GAMMA, E. (1995) Design Pattern – elements of reusable object-oriented software, BookMan;

[Elias2008] Elias, G.S. & Wildt, D. (2008) Métricas para melhoria contínua de código – Um estudo de caso com Java

[Kan2002] KAN, Stephen H (2002). Metrics and Models in Software Quality Engineering. Boston: Addison – Wesley.

[Pressmann1995] PRESSMANN, Roger S. (1995). Engenharia de Software. 3. ed. São Paulo: Makron Books.

[Koscianski2006] KOSCIANSKI, André; SOARES, Michel dos Santos (2006). Qualidade de Software. São Paulo: Novatec

[Lanza2006] LANZA, Michele; Manrinescu, Radu; Ducassi, S. (2006). Object-Oriented Metrics in Pratice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer.

[Duvall2002] DUVALL, Paul M (2002). Continuous Integration. Boston: Addison – Wesley.

[Sommerville2003] SOMMERVILLE, Ian (2003). Engenharia de Software. São Paulo: Addison Wesley.

[Poppendieck2008] POPPENDIECK, Mary (2008). Lean Software Development.

[Janzen2008] Janzen, D. S. & Saiedian, H. (2008) Does Test-Driven Development Really Improve Software Design Quality?, IEEE Software.

[AnalityX] (Em <https://developers.google.com/java-dev-tools/codepro/doc/>. Acessado em 03 de Outubro de 2012

[Beck2004] BECK, Kent (2004). Programação Extrema Explicada.Porto Alegre: Bookman.

[Aniche2012] ANICHE, M. TEST-DRIVEN DEVELOPMENT: TESTE E DESIGN NO MUNDO REAL. [S.l.]: CASA DO CODIGO, 2012. ISBN 9788566250046.

[Meirelles2008] Meirelles, P. R. M. (2008) Levantamento de Métricas de Avaliação de Projetos de *Software* Livre.

[Beck2002] BECK, K. Test Driven Development: By Example. Boston, MA, USA:Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

Anexo A

Universidade Federal de Lavras
Departamento de Ciência da Computação
GCC110 – Programação Orientada a Objetos

Trabalho Prático:

Sistema de Operadora de Celular

Implemente um conjunto de classes para um sistema de uma operadora de telefones celulares. A operadora necessita manter informações sobre seus celulares, seus clientes e suas contas. Uma descrição do funcionamento do sistema é dada abaixo:

§· A operadora mantém as seguintes informações sobre seus clientes: *cpf* (ou *cnpj*), nome e endereço. Ao adquirir um celular, o cliente adere-se a um dos planos da operadora. Cada plano possui um nome e define um valor a ser cobrado por minuto de ligação. Por simplicidade do trabalho, considere que cada plano tem um único valor, independente do horário e do local da ligação;

§· Um cliente pode possuir diversos celulares, mas um celular deve pertencer a
apenas um cliente;

§· Cada celular vendido pela operadora possui um número e está associado a um cliente e a um plano e, possui uma lista de ligações. Os celulares são classificados em *SmartPhones* ou *RegularPhones*, para efeitos de promoções que
a operadora possa oferecer em seus planos.

§· Os planos da operadora são do tipo cartão (pré-pago) ou assinatura (pós-pago). Para um plano do tipo cartão, a operadora deve controlar o valor dos créditos restantes e a

data de validade desses créditos. Para um plano do tipo assinatura, a operadora deve controlar dia (do mês) de vencimento da fatura;

§· Para cada ligação efetuada de um celular, a operadora registra a data/hora da
ligação e a sua duração, em minutos. Uma ligação pode ser do tipo simples (chamada comum) ou ligação de Internet (consumo de pacotes proporcionais para Internet);

§· Cada plano possui uma lista de promoções ofertadas de acordo com o tipo de
plano e o tipo do celular. As promoções podem ser de três tipos: “Internet”, “Minutos” e “Bônus”.

§· Internet: ofertada para celulares do tipo *SmartPhone*, habilitados em ambos os tipos de plano: pós e pré-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, velocidade, franquia, velocidade além da franquia.

§· Minutos: ofertada para celulares de ambos os tipos (*SmartPhone* e *RegularPhone*) habilitados no plano pós-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, validade e quantidade.

§· Bônus: ofertada para celulares de ambos os tipos (*Smartphone* e *Regularphone*) habilitados no plano pré-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, validade, quantidade e limite-diário.

O sistema deve possuir as seguintes opções:

§· Cadastrar clientes e cadastrar planos. O usuário fornece os dados do cliente ou do plano;

§· Cadastrar promoção. O usuário informa o tipo da promoção e os dados referentes a ela.

§· Habilitar (criar um novo) um celular. O usuário escolhe o tipo de celular (*SmartPhone* ou *RegularPhone*), o tipo do plano (cartão ou assinatura), promoções (entre as disponíveis para o modelo de telefone e plano) e fornece o dia de vencimento da fatura (para celular habilitado em plano de assinatura). O número do celular é gerado automaticamente pelo sistema;

§· Adicionar promoção. O usuário informa o número de celular. O sistema exibe as promoções disponíveis (que ainda não façam parte) que podem ser adicionadas àquele aparelho/plano.

§· Adicionar créditos para celular habilitado no plano cartão. O usuário fornece o
número do celular e o valor dos créditos a adicionar. O valor dos créditos é somado aos créditos atuais e a data de validade dos créditos é atualizada para 180 dias após a data corrente do sistema;

§· Registrar ligação. O usuário fornece o número do celular, a data/hora da ligação e a sua duração, em minutos. No caso de celular habilitado no plano de cartão, o sistema deve gerar uma exceção se os créditos não forem suficientes para o tempo da ligação ou se a data de validade dos créditos estiver vencida. Por simplicidade do trabalho, a ligação é registrada após a sua ocorrência. Em um
caso real, deveria haver opção para iniciar a ligação, para adicionar tempo de ligação, verificar tempo de crédito e para finalizar a ligação; tudo ocorrendo em tempo real; O sistema deve também levar em conta as promoções cadastradas para aquele aparelho para fins de contabilização dos minutos cobrados.

§· Ligações do tipo Internet devem ser contabilizadas segundo o saldo do pacote promocional do plano. Uma exceção deve ser lançada caso haja a tentativa de registro de uma ligação do tipo Internet em um aparelho que não contemple esse tipo de ligação em seu plano.

§· Listar os dados referente a Internet. O usuário fornece o número do celular. O sistema verifica a existência de uma promoção do tipo Internet cadastrada para o número fornecido. Caso exista, deve ser exibida a franquia (franquia da promoção – total gasto) e a velocidade atual (velocidade da promoção ou velocidade além da franquia).

§· Listar o valor da conta para celular habilitado no plano de assinatura. O usuário fornece o número do celular. O sistema imprime na tela o valor total das ligações ocorridas após o dia de vencimento do mês anterior e o dia de vencimento do mês corrente, para o celular informado; O sistema deve também levar em conta as promoções cadastradas para este celular para fins de contabilização dos minutos.

§· Listar o valor dos créditos e a data de validade para celular de cartão. O usuário fornece o número do celular. O sistema imprime na tela os dados do celular informado;

§· Listar extrato de ligações. O usuário fornece o número do celular e uma data. O sistema imprime na tela todas as ligações efetuadas do celular a partir da data
informada. Para cada ligação, devem ser impressos a data, a duração da ligação
e o valor cobrado;

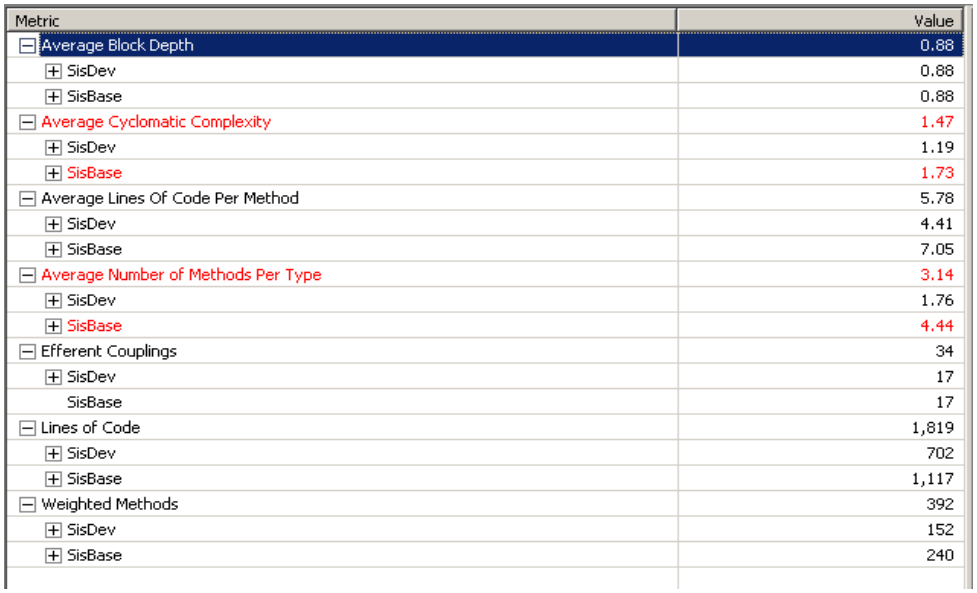
§· Listar clientes, listar planos, listar promoções e listar celulares. O sistema imprime na tela todos os dados de cada cliente ou plano ou celular.

§· Informativo de vencimento de fatura/créditos/promoções. Sempre que um desses itens alcançar a data de vencimento, o sistema deve informar ao usuário os dados do cliente e celular cujos item venceu.

§· Verificar validade das promoções. O usuário informa o número do celular. O
sistema exibe as promoções vinculadas àquele aparelho com as respectivas datas de vencimento.

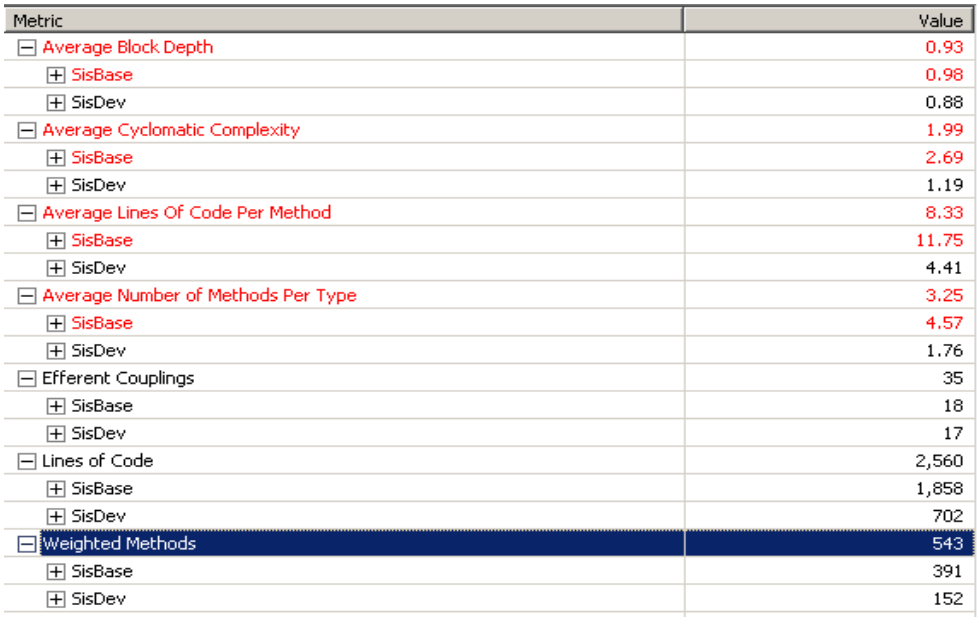
Anexo B

Métricas do SisBase sem classe de interface



Metric	Value
Average Block Depth	0.88
SisDev	0.88
SisBase	0.88
Average Cyclomatic Complexity	1.47
SisDev	1.19
SisBase	1.73
Average Lines Of Code Per Method	5.78
SisDev	4.41
SisBase	7.05
Average Number of Methods Per Type	3.14
SisDev	1.76
SisBase	4.44
Efferent Couplings	34
SisDev	17
SisBase	17
Lines of Code	1,819
SisDev	702
SisBase	1,117
Weighted Methods	392
SisDev	152
SisBase	240

Figura Anexo B – Resultado das métricas selecionadas sobre SisDev e SisBase – sem a classe Interface.java



Metric	Value
Average Block Depth	0.93
SisBase	0.98
SisDev	0.88
Average Cyclomatic Complexity	1.99
SisBase	2.69
SisDev	1.19
Average Lines Of Code Per Method	8.33
SisBase	11.75
SisDev	4.41
Average Number of Methods Per Type	3.25
SisBase	4.57
SisDev	1.76
Efferent Couplings	35
SisBase	18
SisDev	17
Lines of Code	2,560
SisBase	1,858
SisDev	702
Weighted Methods	543
SisBase	391
SisDev	152

Figura Anexo B – Resultado das métricas selecionadas sobre SisDev e SisBase – com a classe Interface.java

É possível notar que a classe desconsiderada nessa análise do SisBase (Interface.java) gera um impacto significativo em todas as métricas, com destaque para as métricas de complexidade. Contudo, os benefícios percebidos com a metodologia aplicada neste trabalho não se alteram, permanecendo os valores para o SisDev ainda melhores ou equivalentes, se comparados aos valores do SisBase.