CENTRO UNIVERSITÁRIO FACVEST
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

PRINCÍPIOS SOLID E BOAS PRÁTICAS DE DESENVOLVIMENTO DE SOFTWARE

Área: Engenharia de Software

ÉRICO PADILHA JUNIOR

LAGES (SC), NOVEMBRO DE 2012

CENTRO UNIVERSITÁRIO FACVEST
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

PRINCIPIOS SOLID E BOAS PRÁTICAS DE DESENVOLVIMENTO DE SOFTWARE

Área: Engenharia de Software

Érico Padilha Junior

Projeto apresentado à Banca Examinadora do Trabalho de Conclusão do Curso de Ciência da Computação para análise e aprovação.

Lages (SC), Dezembro de 2012

ÉRICO PADILHA JUNIOR

PRINCIPIOS SOLID E BOAS PRÁTICAS DE DESENVOLVIMENTO DE SOFTWARE

Trabalho de Conclusão de Curso apresentado à banca examinadora e ao Centro Universitário FACVEST como parte dos requisitos para a obtenção de bacharel em Ciência da Computação.

Prof. Msc. Márcio José Sembay

Lages, SC _____/_____/2012. Nota _____ _____

_____

Prof. Coordenador do Curso de Ciência da Computação

LAGES (SC), DEZEMBRO DE 2012
EQUIPE TÉCNICA

Acadêmico
Érico Padilha Junior

Professor Orientador
Prof. Márcio José Sembay, Msc.

Coordenador de TCC
Prof. Márcio José Sembay, Msc.

Coordenador do Curso
Prof. Márcio José Sembay, Msc.

RESUMO

Para uma empresa de software, o tempo de entrega de um produto sempre foi um grande problema. No fim do prazo estipulado, o software tem de estar funcionando para ser entregue ao cliente. O mesmo ocorre durante uma solicitação de nova funcionalidade ou manutenção quando solicitado pelo cliente. Outro ponto importante é o tempo de vida do produto criado. Em busca de melhoria nesses fatores, é um objetivo apresentar um estudo sobre princípios e práticas que possam ser utilizadas pelos desenvolvedores de software. É realizado também um questionário sobre o estudo dessas praticas, e aplicado aos desenvolvedores, em duas empresas regionais, a fim de obter um levantamento sobre a utilização destas. O assunto torna-se importante no momento em que sabemos que o uso destas boas práticas e princípios SOLID pode trazer benefícios para todos os fatores citados anteriormente.

ABSTRACT

For a software company, the delivery time of a product has always been a big problem. At the end of the stipulated period, the software has to be running to be delivered to the customer. The same happens during a request for new functionality or maintenance when requested by the customer. Another important point is the lifetime of the product created. In search of improving these factors, it is a goal to present a study of principles and practices that can be used by software developers. It also conducted a questionnaire on the study of these practices, and applied to developers in three regional businesses in order to obtain a survey on the use of these. The issue becomes important when we know that the use of these best practices and SOLID principles can bring benefits to all the factors mentioned above.

LISTA DE ABREVIATURAS

UML – *Unified Modeling Language* (Linguagem de Modelagem Unificada).

SOLID – ***S****ingle responsibility*, ***O****pen-closed*, ***L****iskov substitution*, ***I****nterface segregation* e ***D****ependency inversion*.

SRP – *Single Responsibility Principle*. (Princípio da Responsabilidade Única)

OCP – *Open-closed Principle*. (Princípio do Aberto / Fechado)

LSP – *Liskov Substitution Principle*. (Princípio da Substituição de Liskov)

ISP – *Interface Segregation Principle*. (Princípio da Segregação de Interface)

DIP – *Dependency Inversion Principle*.

IDE – *Integrated Development Environment* (Ambiente Integrado de Desenvolvimento).

GUI – *Graphical User Interface* (Interface Gráfica do Usuário).

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Levando em consideração o grande crescimento da demanda da área de desenvolvimento de *software* nos dias atuais, as empresas, e principalmente os desenvolvedores, lutam contra o tempo para entregar o produto ao cliente. Para que a empresa possa ter sucesso, é necessário que elas sejam ágeis, e não violem o prazo. É necessário também qualidade e rapidez ao implementar uma nova funcionalidade no produto, ou realizar alguma correção ou manutenção.

Para que tudo isso seja viável, é possível eliminar o mal pela raiz, e melhorar as condições e problemas pelos que os programadores passam. Acontece que os problemas são gerados por eles mesmos, então é necessária uma educação na hora de codificar. Conforme o tempo passa, se o desenvolvedor não segue padrões ou práticas que deixem o código mais organizado para que se aumente a legibilidade, o código começa a ficar bagunçado e confuso.

> "Conforme a confusão aumenta, a produtividade da equipe diminui, assintoticamente aproximando-se de zero. Com a redução da produtividade, a gerência faz a única coisa que ela pode; adiciona mais membros ao projeto na esperança de aumentar a produtividade. Mas esses novos membros não conhecem o projeto do sistema, não sabem a diferença entre uma mudança que altera o propósito do projeto e aquela que o atrapalha." (MARTIN, 2010, p. 4).

Aumentar a quantidade do quadro de funcionários não vai resolver o problema, às vezes pode até piorá-lo. O código confuso de outra pessoa, ou até mesmo da mesma pessoa, pode trazer problemas para a equipe. Martin (2010) afirma que dentro de um ou dois anos, as equipes que trabalharam com maior rapidez e eficiência no início de um projeto podem perceber mais tarde, que estão perderam parte de sua produtividade. Apesar disso, a equipe permanece sobre tremenda pressão para aumentar a produtividade.

Por fim, a equipe precisa se reeducar com as boas práticas de desenvolvimento, para que, possam escrever um código bom, limpo e "maciço". E se o erro já foi cometido, é necessário refatorar o projeto danificado. É através disso que pode se evitar problemas que acontecem ao longo do tempo, em um código ruim. Este pode ser considerado um trabalho de prevenção.

1.1. Justificativa

Em uma empresa de desenvolvimento de *softwares*, existem diversos setores importantes para que o produto final (*software*) possua qualidade. No setor de desenvolvimento, os funcionários (desenvolvedores) recebem certos requisitos do sistema para que o desenvolvam. Explica Martin (2010), que desta maneira, é possível perceber que o produto final, possuirá diferentes tipos de código, aplicados a fim de um único objetivo, pois cada desenvolvedor possui a sua lógica e prática.

Pelo fato do produto ser um *software*, o mesmo, geralmente, precisa receber manutenção. A dificuldade desta, por sua vez, é relativa ao tempo de vida do produto e a maneira que o mesmo foi codificado. Não se sabe ao certo quando ocorreu o início dos Princípios de Design de Desenvolvimento, SOLID.

Generalizando, estes princípios possuem conceitos e conselhos de desenvolvimento. Na maioria das vezes, este conjunto de conceitos é esquecido, ou até mesmo ignorado, pois se tratando de uma empresa que visa lucros, a mesma possui um prazo de entrega do produto. Para não extrapolar o prazo, os desenvolvedores fazem o possível para que o produto final seja entregue e tenha qualidade.

Os princípios, quando aplicados juntos, tendem a facilitar a manutenção do produto, e até mesmo seu desenvolvimento, além de aumentar o seu tempo de vida. “A ferramenta crítica de design para o desenvolvimento de *software* é a mente bem educada em princípios de design. Não é a UML ou qualquer outra tecnologia.” (LARMAN, 2005, p. 272). Uma causa muito comum de um *software* ter um tempo curto de vida é a maneira como é desenvolvido. Obviamente, uma empresa do ramo de programas de computador visa gerar lucros. Além de serviços, uma das principais fontes de lucro desta empresa são os *softwares*, o que significa que quanto maior for o tempo de vida deles, maior será o lucro da mesma.

Foi aplicado um questionário de pesquisa sobre essas boas práticas e princípios em duas empresas, pois estas estão em constante crescimento na região serrana de Santa Catarina. Houve a tentativa também da aplicação em uma terceira empresa, que está entre as maiores na região, mas não houve interesse da parte deles em participar da pesquisa.

Neste sentido, esforços foram realizados para estudar e analisar estes princípios, que se tornam essenciais, como uma boa prática a ser aplicada no setor de desenvolvimento das empresas do ramo.

1.2. Importância

Esse trabalho se torna importante, pois tem como objetivo mostrar a necessidade da aplicação de princípios SOLID de design e desenvolvimento de *software*, para prover facilidade de manutenção à equipe de desenvolvimento e mostrar como está o uso dos mesmos, dentro das empresas nos dias hoje.

> "Podem dizer que a programação deixou de ser uma preocupação e que devemos nos preocupar com modelos e requisitos. Outros até mesmo alegam que o fim do código, ou seja, da programação, está próximo; que logo todo o código será gerado, e não mais escrito. E que não precisarão mais de programadores, pois as pessoas criarão programas a partir de especificações." (MARTIN, 2010, p. 2).

É possível perceber que, a afirmação de Martin, feita em 2010, ainda continua verídica, pois as empresas ainda dependem de programadores e de códigos, para o desenvolvimento de seus produtos. "Os códigos representam os detalhes dos requisitos e em certo nível, e não há como ignorar ou abstrair esses detalhes; eles precisam ser especificados" (MARTIN, 2010, p. 2). O nível das linguagens tende a crescer, e junto, os recursos das IDEs, visando facilitar o trabalho de desenvolvimento, mas mesmo assim não será capaz de substituir os detalhes e as lógicas do desenvolvedor.

Portanto, ao analisar o pensamento de Martin, sendo que teremos um código, apesar de toda a criatividade e intuição do ser humano, poderemos e deveremos aplicar conceitos no código que a fim de melhorar a produtividade junto ao desenvolvimento em equipe e à manutenção do *software*.

1.3. Objetivos

1.3.1. Objetivo Geral

Estudar, analisar e mostrar a importância das boas práticas de desenvolvimento e princípios SOLID para a produção de *software* em equipe.

1.3.2. Objetivos Específicos

Os objetivos específicos deste trabalho constituem no estudo de vários conceitos, que ao serem aplicados a um software, resultam em maior facilidade de manutenção e consequentemente, maior tempo de vida do mesmo. Sendo assim:

- Apresentar referencial bibliográfico sobre boas práticas de desenvolvimento de *software* e princípios SOLID.
- Apresentar uma pesquisa em duas empresas regionais, elaborada através de questionário, sobre o uso de boas práticas e princípios SOLID.

2. METODOLOGIA

A Figura 02 mostra as etapas percorridas no desenvolvimento deste Trabalho de Conclusão de Curso.

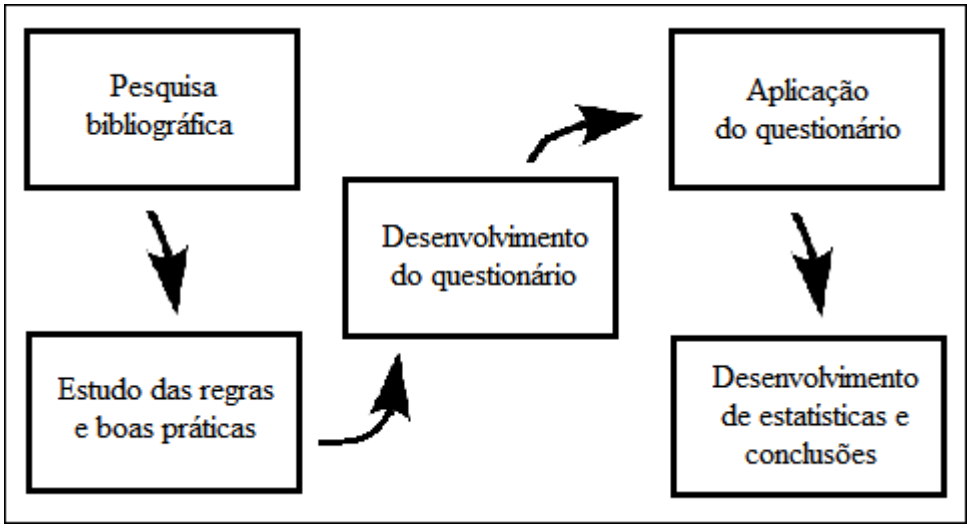


Figura 1 – Etapas do desenvolvimento do TCC.

Fonte: o próprio autor.

2.1. Caracterização da pesquisa

Esta pesquisa é exploratória, pois tem como objetivo aumentar o conhecimento e familiaridade com o objeto de estudo, que são as boas práticas de desenvolvimento e princípios SOLID.

"Muitas vezes o pesquisador não dispõe de conhecimento suficiente para formular adequadamente um problema ou elaborar de forma mais precisa uma hipótese." (HEERDT; LEONEL, 2007, p. 63). Neste caso, é necessário "desencadear um processo de investigação que identifique a natureza do fenômeno e aponte características essências das variáveis que se quer estudar" (KÖCHE, 1997, p. 126).

Para ocorrer esta investigação na pesquisa exploratória é necessário o desenvolvimento de um questionário.

> "O planejamento da pesquisa exploratória é bastante flexível e pode assumir caráter de pesquisa bibliográfica, pesquisa documental, estudos de caso, levantamentos, etc. As técnicas de pesquisas que podem ser utilizadas na pesquisa exploratória são:

> formulários, questionários, entrevistas, fichas para registro de avaliações clínicas, leitura e documentação quando se tratar de pesquisa bibliográfica.” (HEERDT, LEONEL, 2007, p. 63).

Quanto à natureza da pesquisa em relação aos objetivos deste trabalho, a abordagem será quanti qualitativa. Será quantitativa porque o questionário será aplicado a vários funcionários, em duas empresas de desenvolvimento de software, e qualitativa porque o quesito a ser observado nos questionário é o uso das boas práticas e princípios SOLID.

Em relação ao método de pesquisa, será por meio de levantamento, por se tratar da aplicação de um questionário desenvolvido, tomando como base e conceitos, as boas práticas de desenvolvimento e princípios SOLID. Segundo Gil (2002), os levantamentos caracterizam-se pela interrogação direta das pessoas cujo comportamento se deseja conhecerem. Basicamente, procede-se à solicitação de informações a um grupo significativo de pessoas acerca do problema estudado para, em seguida, mediante análise quantitativa, obterem-se as conclusões correspondentes aos dados pesquisados.

2.2. Coleta e Análise de Dados

Para coleta de dados é necessário realizar uma pesquisa por meio de levantamento de dados, no qual estejam especificados os pontos de pesquisa e os critérios para a seleção das perguntas.

Andrade (2010) afirma que todas as etapas da coleta de dados devem ser esquematizadas, a fim de facilitar o desenvolvimento da pesquisa, bem como assegurar uma ordem lógica na execução das atividades.

Os materiais utilizados para realização das perguntas contidas no questionário (anexo C) serão adotados do livro “Código Limpo”, de Robert C. Martin, juntamente com os conceitos dos princípios SOLID, retirado do site “http://objectmentor.com/” e do livro “Princípios, Padrões e Práticas Ágeis em C#”, ambos do mesmo autor citado acima. Foi realizado contato telefônico com os gerentes de projeto das empresas participantes para elaboração indireta na criação do questionário. Após a realização da pesquisa nas empresas, é desenvolvida uma estatística a fim de explorar a quantidade de desenvolvedores que utilizam das boas práticas de desenvolvimento, e dos princípios SOLID.

2.3. Itens abordados dentro do levantamento

O questionário foi elaborado a partir da abordagem de itens dentro das boas práticas de desenvolvimento de *software* e dos princípios SOLID. São os seguintes:

- Conceitos para nomeação:
  - Nomeação de modo geral dentro do código:
    - Uso de nomes que revelem o seu proposito;
    - Não utilizar nomes que causem confusão;
    - Nomes de fácil pronúncia;
    - Evitar o uso de trocadilho;
  - Nomeação de variáveis
    - Nomes com distinções significativas;
      - Variáveis representadas por letras;
    - Uso de nomes passíveis de busca;
  - Nomeação de classes;
  - Nomeação de métodos;
- Conceitos outras
  - Itens dos outros conceitos
- Princípios SOLID
  - Single Responsibility Principle (Princípio da responsabilidade única);
  - Open / Closed Principle (Princípio do Aberto / Fechado);
  - Liskov Substitution Principle (Princípio da Substituição de Liskov);
  - Dependency Inversion Principle (Princípio da Inversão de Dependência);
  - Interface Segregation Principle (Princípio da Segregação de Interfaces);

3. REVISÃO BIBLIOGRÁFICA

3.1. A DESCRIÇÃO DA ESCRITA DE UM CÓDIGO LIMPO

Atualmente nas empresas de desenvolvimento de software, os desenvolvedores claramente tem um inimigo em comum: o tempo. Essa é uma das razões pela qual o principal objetivo, seja entregar o produto final, não se importando com os meios a serem tomados. Mas existem controvérsias neste sentido. “Todos os desenvolvedores (...) sabem que bagunças (...) reduzem o rendimento. Mesmo assim se sentem pressionados a cometer essas bagunças para cumprir os prazos” (MARTIN, 2010, p. 6). Martin (2010) aborda a realidade do ramo de desenvolvimento de software, porém, ele mesmo explica que a segunda parte do dilema está errada, e que não será possível cumprir o prazo de entrega se fizer bagunça no código.

Então podemos concluir que manter o código organizado, mesmo que da pior maneira, é essencial para entrega de um produto dentro do prazo. É preciso saber como manter o código organizado da maneira correta. Foram criadas normas, convenções, conceitos para que o código seja bem escrito, mas apenas isso não basta.

> “Escrever um código limpo exige o uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de uma sensibilidade meticulosa adquirida sobre limpeza. A sensibilidade ao código é o segredo. (...) Ela não só nos permite perceber se o código é bom ou ruim, como também nos mostra a estratégia e disciplina de como transformar um código ruim em um limpo.” (MARTIN, 2010, p. 6).

Outro ponto muito importante para obter um código bem escrito, além da disciplina, é a prática. Para Mohtashim (2008), prática, neste sentido, pode significar um hábito, uma rotina, ou algo que não precisa ser lembrado; complementando que a prática vem apenas do ato de praticar e necessita dedicação e comprometimento. Então a prática não é algo simples, e sim, algo que requer muito esforço e dedicação para se adquirir. Observa-se então que a curva de aprendizagem para a escrita de um código organizado é alta, onde se obedece aos princípios e boas práticas, e onde são seguidos todos os conceitos e convenções.

3.2. A NOMEAÇÃO DENTRO DO CÓDIGO

Dentro de uma empresa de desenvolvimento de software, no decorrer da escrita de um código, o desenvolvedor esta a todo tempo utilizando novos nomes para seus pacotes, classes, funções ou atributos. “Nomeamos, nomeamos e nomeamos. Como fazemos muito isso, é melhor que façamos bem” (MARTIN, 2010, p. 17). Por ser uma empresa, seus produtos são feitos em equipe. Dentro desta parte de nomeação dentro do código, existem conceitos de: nomes de variáveis, nomes de classes, nomes de métodos e funções, e nomes de interfaces e implementações.

Nomear não é uma tarefa fácil. “O mais difícil sobre escolher bons nomes é a necessidade de se possuir boas habilidades de descrição e um histórico cultural compartilhado. Essa é uma questão de aprender, e não é técnica, gerencial ou empresarial.” (MARTIN, 2010, p. 30).

3.2.1. Conceitos de nomeação de aplicação geral

Alguns conceitos de nomeação se aplicam tanto em variáveis e atributos, como em nomes de classes, métodos e funções, tais como: uso de nomes que revelem o propósito, a não utilização de nomes confusos, e nomes de fácil pronúncia.

3.2.1.1. Uso de nomes que revelem o seu propósito

Uma tarefa aparentemente e teoricamente simples, como escolher nomes para usar em seu código, na prática pode ser difícil. “O nome de uma variável, função ou classe deve responder a todas as questões. Ele deve lhe dizer por que existe, o que faz e como é usado. Se um nome requer comentário, então ele não revela o seu proposito” (MARTIN, 2010, p. 18).

Nota-se que o autor cita que, ao nomear uma classe, função ou variável, é preciso que o nome responda a certas perguntas. Estas perguntas servem para facilitar o entendimento dos elementos pelo desenvolvedor que esta lendo o código. A aplicação destes conceitos em uma empresa pode deixar a leitura e a manutenção mais rápidas. Esta é a recompensa por escolher bons nomes.

3.2.1.2.Não utilizar nomes que causem confusão

Um fato comum durante a leitura de um código que você não escreveu, é o entendimento errado de uma variável ou método.

> "Devemos evitar palavras cujos significados podem se desviar daquele que desejamos. (...) Não se refira a um grupo de contas como *listaDeContas*. A palavra lista significa algo específico para programadores. Se o que armazena as contas não for uma lista de verdade, poderá confundir outros. Portanto, *grupoDeContas* ou apenas *contas* seria melhor" (MARTIN, 2010, p. 19).

Nomes citados acima, como no exemplo de Robert Martin, além de possivelmente confundir o desenvolvedor, ao mesmo tempo podem atrasá-lo, o que não é bom para ele e nem para a empresa ao cumprir prazos. Portanto é necessária máxima atenção ao declarar as variáveis ou atributos.

3.2.1.3.Nomes de fácil pronúncia

"Uma parte considerável de seu cérebro é responsável pelo conceito das palavras. E por definição, as palavras são pronunciáveis" (MARTIN, 2010, p. 21) Toda nomeação que fazemos durante o desenvolvimento do código é feita com palavras. Portanto, as mesmas têm de ser pronunciáveis.

O dialogo sobre o projeto no cotidiano de uma empresa de desenvolvimento é indispensável. E ao citar o nome de um método ou atributo a um colega desenvolvedor, quanto mais pronunciável ele for, melhor. Martin (2010) complementa que se não puder pronunciar este nome, não terá como discutir sobre o mesmo com a equipe, dentro da empresa.

3.2.1.4.Evitar o uso de trocadilhos

"Evite usar a mesma palavra para dois propósitos. Usar o mesmo termo para duas ideias diferentes é basicamente um trocadilho. Se você seguir a regra de "uma palavra por conceito" pode acabar ficando com muitas classes que possuam, por exemplo, o método *add*" (MARTIN, 2010, p. 26).

Este conceito para evitar uso de trocadilhos serve para evitar que o método *add*[1], por exemplo, possua dois significados em classes diferentes. Quando o nome a assinatura do método não segue um padrão, então existe o "trocadilho", pois eles têm significados diferentes.

> "Digamos que tenhamos muitas classes nas quais *add* criará um novo valor por meio da adição e concatenação de dois valores existentes. Agora digamos que estejamos criando uma nova classe que possua um método (*add*) que coloque seu único parâmetro em uma coleção. (...) neste caso, a semântica é diferente. Portanto, deveríamos usar um nome como *inserir* ou *adicionar*." (MARTIN, 2010, p. 26).

Isso faz com que a leitura do código seja rápida, e não se torne um estudo demorado, dificultando o trabalho do desenvolvedor.

3.2.2. Conceitos para nomeação de variáveis

Existem alguns conceitos de nomeação que são aplicáveis apenas a variáveis, e não a métodos e classes. São elas: nomes com distinções significativas e uso de nomes passíveis de busca.

3.2.2.1.Nomes com distinções significativas

Este conceito necessita um pouco mais de interpretação para ser compreendida. Distinções acontecem em variáveis representadas por nomes e variáveis representadas por letras. "(...) não é possível usar o mesmo nome para referir-se a duas coisas diferentes em um mesmo escopo. Se os nomes precisam ser diferentes, então também devem ter significados distintos" (MARTIN, 2010, p. 20). O principal objetivo deste conceito é prevenir confusões dentro de um mesmo escopo.

Normalmente o uso de variáveis nomeadas com letras, é feito em laços de repetição que são pequenos. Martin explica que "usar números sequenciais em nomes (a1, a2,..., aN) é o oposto da seleção de nomes expressivos. Eles não geram confusão, simplesmente não oferecem informação alguma ou dica sobre a intenção de seu criador" (MARTIN, 2010, p. 20).

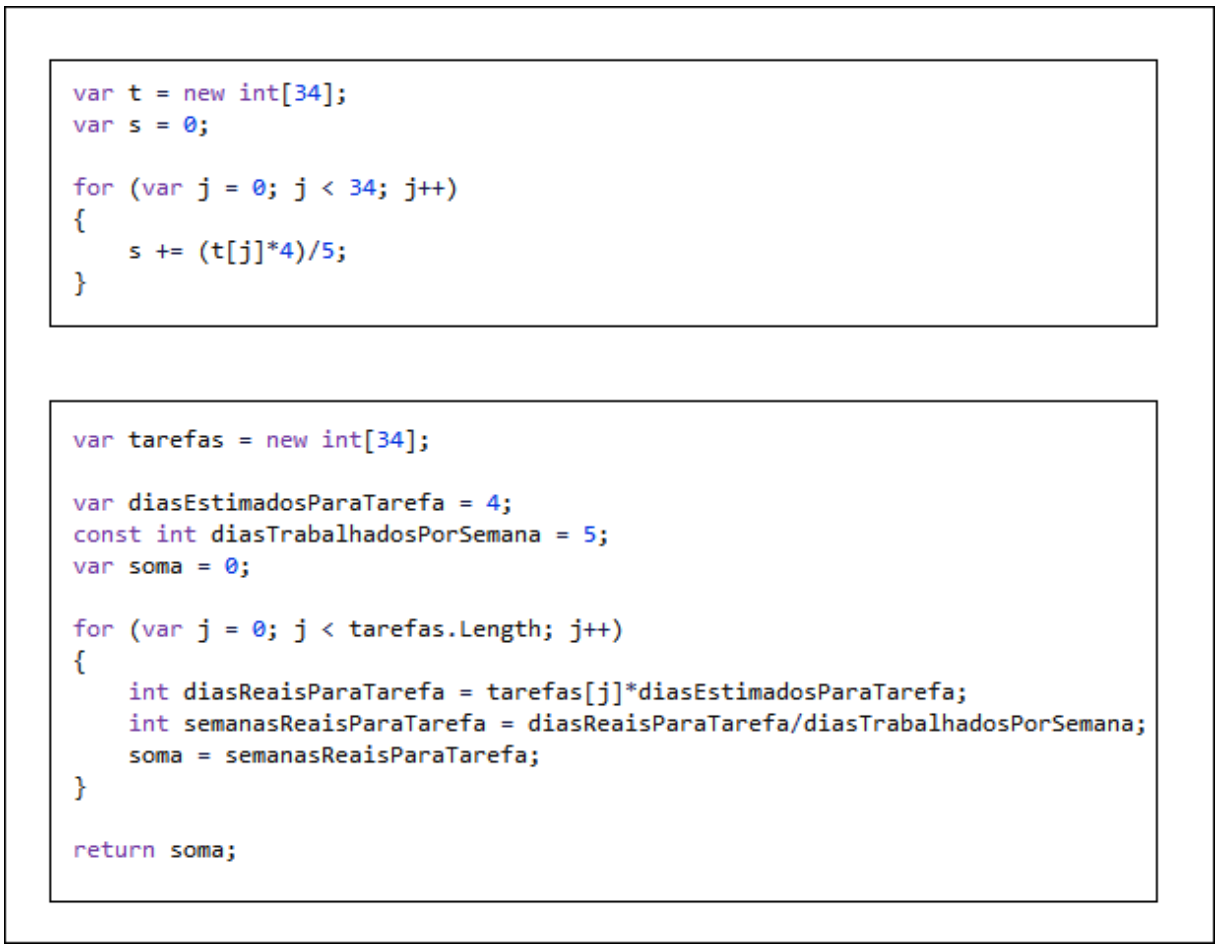
---

[1] Do inglês, "adicionar".

Percebe-se que ao usar números sequenciais em nomes de variáveis, o desenvolvedor quebra o primeiro conceito citado por Martin, a qual diz que cada nome deve revelar o seu propósito. Portanto, dependendo do contexto, é necessário usar palavras. “Faça a distinção dos nomes de uma forma que o leitor compreenda as diferenças” (MARTIN, 2010, p. 21).

3.2.2.2. Uso de nomes passíveis de busca

A função deste conceito é que, ao fazer a nomeação, sejam definidos nomes que permitam e sejam fáceis de buscar. “Nomes de uma só letra ou números possuem um problema em particular por não ser fácil localizá-los ao longo de um texto” (MARTIN, 2010, p. 22). No trecho “números possuem um problema em particular”, o autor se refere quando são utilizadas em constantes, por exemplo, dentro de um laço de repetição “*for*”.



```
var t = new int[34];
var s = 0;

for (var j = 0; j < 34; j++)
{
    s += (t[j]*4)/5;
}
```

```
var tarefas = new int[34];

var diasEstimadosParaTarefa = 4;
const int diasTrabalhadosPorSemana = 5;
var soma = 0;

for (var j = 0; j < tarefas.Length; j++)
{
    int diasReaisParaTarefa = tarefas[j]*diasEstimadosParaTarefa;
    int semanasReaisParaTarefa = diasReaisParaTarefa/diasTrabalhadosPorSemana;
    soma = semanasReaisParaTarefa;
}

return soma;
```

Figura 2 – Exemplo de nomes passíveis de busca

Fonte: Adaptado de Martin (2010).

Percebe-se que no uso incorreto (trecho de código da parte de cima na Figura 02), são utilizados números para definir o limite do laço de repetição, onde “s” representa “soma”, “t” representa “tarefas”. Ainda no uso incorreto (trecho de código acima na Figura 02), são usados números fixos ao invés de variáveis, que é o caso do numero “4” e do numero “5”, que representam, respectivamente, os dias necessários para realizar determinada tarefa e dias de trabalho por semana. Além disso, o trecho de código perde a leiturabilidade, fazendo com que o laço de repetição dependa de outro trecho para ser compreendido. Já no código escrito na parte de baixo na Figura 02, apesar do código escrito ser mais extenso, a leitura e entendimento do mesmo é facilitado, pois as variáveis dentro do escopo são nomeadas, e as constantes ganham um nome de referência. Os dois trechos de código representam o mesmo método, e ao compara-los é possível entender a sua função.

“O tamanho de um nome deve ser proporcional ao tamanho do escopo. Se uma variável ou constante pode ser vista ou usada em vários lugares dentro do código, é imperativo atribuí-la um nome fácil para busca” (MARTIN, 2010, p. 22). É possível que, por exemplo, a variável “s” (no trecho acima na Figura 02) seja utilizada em outras partes do código; assim, fica relativamente difícil encontrá-la. Mas ao definir o nome de referência daquela mesma variável como “soma”, aumenta a facilidade de encontrá-la. O mesmo se aplica as constantes que foram nomeadas.

3.2.3. Conceitos para nomeação de Classes

O conceito para nomeação de classes, juntamente com as que se aplicam de modo geral, é simples. “Classes e objetos devem ter nomes com substantivo(s), como *Cliente*, *PaginaWiki*, *Conta*, *AnaliseEndereco*. Evite palavras como Gerente, Processador, Dados ou Info no nome de uma classe, que também não deve ser um verbo” (MARTIN, 2010, p. 25). Os nomes de classes não podem ser verbo, porque definem um tipo de um objeto. Acredito que o autor descreva a necessidade de evitar certos nomes, pois são nomes que possam gerar confusão dentro do sistema.

3.2.4. Conceitos para nomeação de Métodos

Este conceito é praticamente oposto a de nomeação de classes, pois “os nomes de métodos, devem ter verbos, como *Postar Pagamento*, *Excluir Pagina*, ou *Salvar*.” (MARTIN,

2010, p. 25) Isso se da porque o método se refere a uma ação do objeto representado pela classe, e um verbo é capaz de representar uma ação. Quanto ao nome iniciar ou não com letra maiúscula, é relativo à convenção da linguagem utilizada.

Existe outro quesito para a nomeação de métodos, que é o de selecionar uma palavra por conceito. Este se caracteriza pela padronização de métodos, em diferentes classes, mas que possuem a mesma função.

> "Escolha uma palavra por cada conceito abstrato e fique com ela. Por exemplo, é confuso ter *pegar*, *recuperar* e *obter* como métodos equivalentes de classes diferentes. (...) Infelizmente você geralmente precisa lembrar-se qual empresa, grupo ou pessoa criou a biblioteca ou a classe, de modo a recordar qual termo foi usado. Caso contrário, você perde muito tempo vasculhando pelos cabeçalhos e exemplos de códigos antigos." (MARTIN, 2010, p. 26).

Utilizando o exemplo de Martin, seria viável utilizar a palavra *get*[2]ou apenas *pegar* para todos os métodos que o autor citou.

3.3. OS PRINCÍPIOS SOLID DE DESIGN E DESENVOLVIMENTO

Os princípios SOLID são divididos em cinco: Responsabilidade Única, Aberto / Fechado, Substituição de Liskov, Inversão de Dependência e Segregação de Interface. Para começar este tema, Martin explica que:

> "Os princípios SOLID não são regras. Eles não são leis. Eles não são verdades perfeitas. Eles dão nome a um conceito de modo que você pode falar e raciocinar sobre este conceito. (...) Dado algum código ou design que faz você se sentir mal a respeito, você pode ser capaz de encontrar um princípio que explica esse sentimento ruim e aconselha como se sentir melhor." (MARTIN, 2009, p. 1).

Os princípios SOLID não tem proposito de transformar o mau programador em um bom programador. Ao contrário das boas práticas de desenvolvimento de *software*, as quais você deve praticar o uso frequente para que vire um hábito, os princípios SOLID são conceitos diferentes, e mais complexos. Eles são similares às boas práticas, mas não é correto utiliza-los livremente sem um estudo. Martin (2009) explica que estes princípios devem ser aplicados com julgamento e que se os mesmo forem aplicados de forma mecânica, se tornam tão ruins como se não fossem aplicadas ao todo.

[2] Do inglês, "pegar".

"Esses princípios resultam de décadas de experiência em engenharia de *software*. Eles foram produzidos através da integração das ideias e publicações de uma grande quantidade de desenvolvedores de *software* e pesquisadores. São casos especiais de princípios consagrados da engenharia de *software*." (MARTIN, 2011, p. 122).

Os princípios SOLID são cinco gerenciamentos de dependências para a programação orientada a objetos. Ao trabalhar com um *software* onde o gerenciamento de dependência é tratado de modo equivocado, o código pode sofrer certas consequências:

> "O código pode se tornar rígido, frágil e difícil de reutilizar. O código rígido passa a possuir dificuldade a modificação, ou dificuldade na alteração da funcionalidade que o mesmo já possui e até mesmo ao adicionar novos recursos. Código frágil é suscetível à introdução de novos erros, particularmente aqueles que aparecem em um módulo, quando a alteração foi feita em outra parte do código." (CARR, 2010).

Ao seguir os princípios SOLID, é possível obter como resultado um código mais flexível e robusto, que tende a possuir maior possibilidade de reutilização e facilidade de manutenção.

3.3.1. Princípio da Responsabilidade Única

"O princípio da Responsabilidade Única (SRP – Single Responsibility Principle) foi descrito no trabalho de Tom DeMarco (1979), e Meilir Page-Jones (1982). Eles o chamavam de 'coesão', o que definiam como a afinidade funcional dos elementos de um módulo." (MARTIN, 2011, p.135). Então, para haver coesão, é necessário que os componentes dentro de uma classe tenham afinidade funcional. É a partir desta afinidade funcional – quando os membros têm sua função relacionada com a de outros membros da classe – que teremos a responsabilidade única.

Segundo Martin (2011), o SRP (Princípio da Responsabilidade Única) assume que não deve haver mais do que uma razão para uma classe mudar.

> "Isso significa que você deve projetar suas classes de forma que cada uma tenha uma única finalidade. Isso não significa que cada classe deve ter apenas um método, mas que todos os membros da classe estão relacionados com a função principal da classe. Quando uma classe tem múltiplas responsabilidades, estes devem ser separados em classes novas." (CARR, 2010).

Uma consequência do não uso deste princípio, é que as responsabilidades começam a ficar altamente ligadas umas as outras. Isso pode provocar erros durante a manutenção, aumentando o tempo necessário para a execução da mesma. Martin diz que "este tipo de acoplamento leva a projetos frágeis que estragam de maneiras inesperadas quando alterados". (MARTIN, 2011, p. 136). A melhor maneira de representar a aplicação de um princípio é através de um exemplo. A figura a seguir mostra um exemplo onde uma classe possui mais de uma responsabilidade.

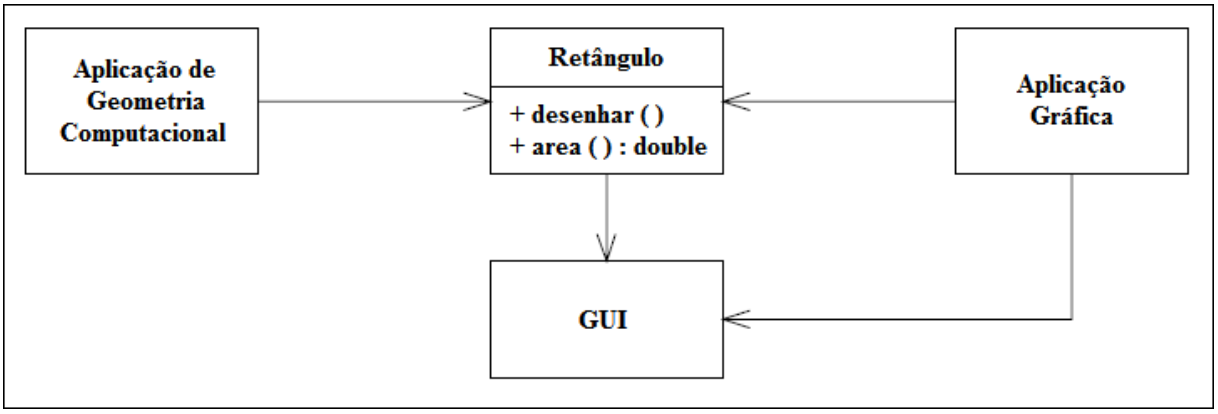


Figura 3 – Mais de uma responsabilidade

Fonte: Adaptada de Martin (2011, p. 136).

A classe *Retângulo*, representada na Figura 03, possui dois métodos: *desenhar*, e *area* que retorna um valor *double*. O primeiro método serve para desenhar o objeto, o outro para calcular a área do mesmo. Conforme a Figura 03, é notável que duas aplicações utilizam da classe retângulo: a *Aplicação Gráfica* e *Aplicação de Geometria Computacional*. O modo que este esquema foi montado viola o Princípio de Responsabilidade única, pois a classe *Retângulo* possui mais de uma responsabilidade. A primeira é fornecer o cálculo de área, a segunda é desenhar a figura na *GUI*. Um fato ocorrente é que os métodos da classe, não possuem afinidade funcional, ou seja, deveriam estar em classes separadas.

> "A violação do SRP causa vários problemas desagradáveis. Primeiro, precisamos incluir *GUI* na *Aplicação de Geometria Computacional*. (...) Segundo, se uma alteração na *Aplicação Gráfica* fizer *Retângulo* mudar por algum motivo, essa mudança poderá nos obrigar a reconstruir, testar novamente e entregar a *Aplicação de Geometria Computacional* outra vez." (MARTIN, 2011, p. 136).

O problema ocorre porque a responsabilidade de desenhar o retângulo ficou altamente acoplada com a responsabilidade de realizar o cálculo da área. Ao aplicar o Princípio da Responsabilidade Única, teremos uma reestruturação do projeto para evitar problemas futuros, como mostrado na Figura 04:

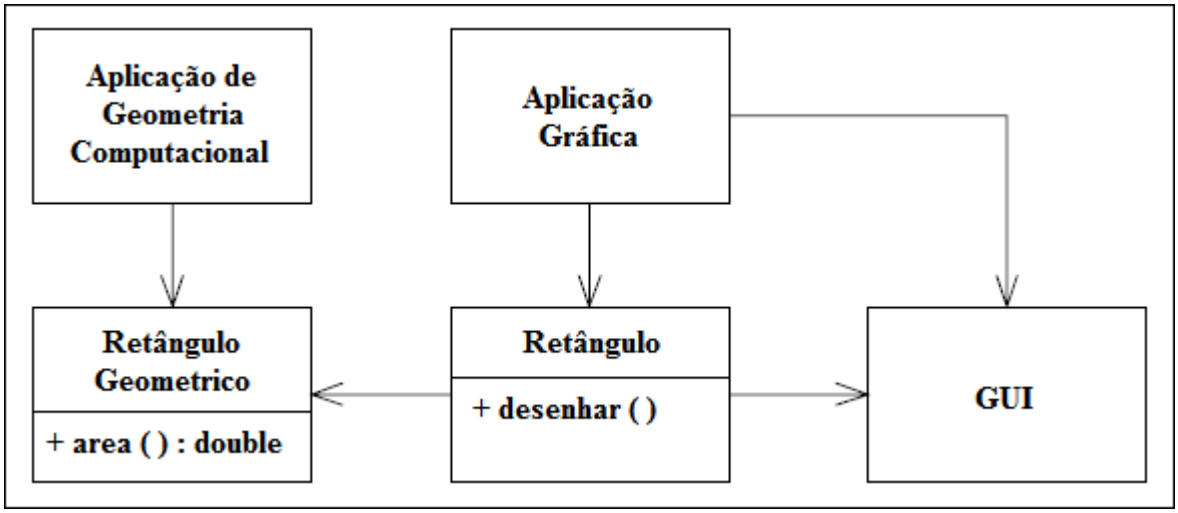


Figura 4 – Responsabilidades separadas

Fonte: Adaptada de Martin (2011, p. 137).

"O Princípio da Responsabilidade Única é simples, porém difícil de acertar. Temos tendência de unir responsabilidades. Encontrar e separar essas responsabilidades corresponde em grande parte do projeto de software em si." (MARTIN, 2011, p. 139). Quando uma classe possui várias responsabilidades, as chances de ela precisar ser alterada aumentam, e consequentemente, conclui Carr (2010) que, cada vez que uma classe é modificada, existe o risco de introduzir erros. E quando se concentra em uma única responsabilidade, esse risco é reduzido e limitado.

3.3.2. Principio do Aberto / Fechado

De acordo com Martin (2011), o princípio do Aberto / Fechado foi criado em 1988, por Bertrand Meyer, que atualmente é professor de Engenharia de Software na universidade ETH Zurich, consagrada uma das maiores e melhores universidades de tecnologia da Europa (fonte[3]). Este princípio diz que "as entidades de *software* (classes, módulos, funções etc.)

[3] Disponível em: [http://se.ethz.ch/~meyer/#Personal] Acesso em: 28 de out, 2012.

devem ser abertas para ampliação, mas fechadas para modificação.” (MARTIN, 2011, p. 141).

A tendência ao aplicar este princípio durante o desenvolvimento, é que, ao realizar uma manutenção não seja necessário alterar o código que já está escrito. “Se o OCP for bem aplicado, mudanças desse tipo são obtidas pela adição de novo código” (MARTIN, 2011, p. 141). Ao executar a manutenção, possuindo este princípio aplicado, a probabilidade de erros em código que já está funcionando, se torna zero, pois não serão modificados.

Após possuir conhecimento da descrição do Princípio do Aberto / Fechado, podem-se estabelecer duas características principais para os módulos que o seguem: eles são abertos para ampliação e são fechados para modificação.

Dizer que eles são abertos para ampliação, significa que “à medida que os requisitos do aplicativo mudam, podemos ampliar o módulo com novos comportamentos que satisfaçam essas alterações.” (MARTIN, 2011, p.142). Sendo assim, a alteração está apenas no que o módulo faz. E são fechados para modificação, pois ao realizar a ampliação do módulo em questão, não existe uma mudança no seu código fonte. “A versão em binário executável do módulo – seja em uma biblioteca que pode ser vinculada, uma DLL ou um arquivo .EXE – permanece intacta.” (MARTIN, 2011, p.142).

Carr (2010) explica que a parte *fechada* da regra estabelece que, uma vez que um módulo foi desenvolvido e testado, o código só deve ser ajustado para corrigir bugs. Já a parte *aberta* da regra diz que você deve ser capaz de estender e incrementar o código existente, a fim de introduzir novas funcionalidades.

Modificar os comportamentos de um módulo sem alterar o código fonte dele soa como algo ilusório. Mas o princípio não pode estar errado, então existe um caminho a ser seguido. Martin diz que este caminho, é através de abstrações fixas.

> “Em C# ou em qualquer outra linguagem de programação orientada a objetos, é possível criar abstrações fixas e que ainda assim representem um grupo ilimitado de comportamentos possíveis. As abstrações são classes base abstratas e o grupo ilimitado de comportamentos possíveis é representado por todas as classes derivadas possíveis.” (MARTIN, 2011, p.142).

Através de um exemplo, nas figuras 05 e 06, é possível explicar este princípio.

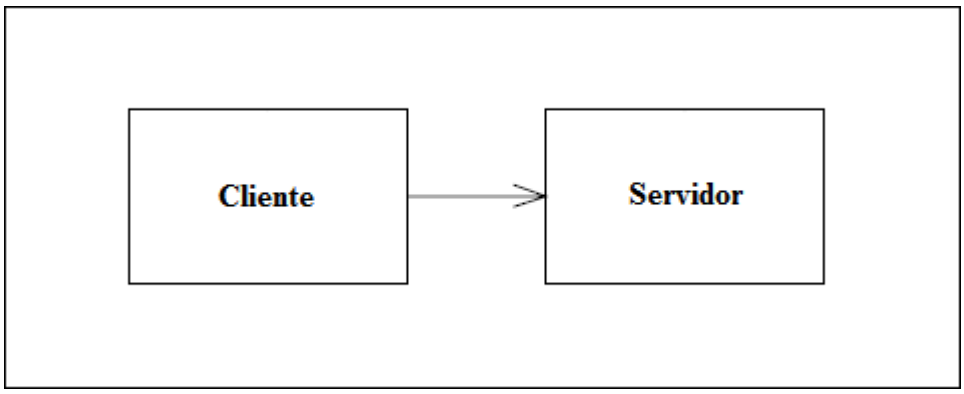


Figura 5 – Cliente não é aberta e fechada.

Fonte: Adaptada de Martin (2011, p.142).

O exemplo simples da Figura 05 mostra uma classe *Cliente* e uma classe *Servidor*, ambas concretas. A classe Cliente utiliza a classe Servidor, da maneira que o projeto é apresentado, não é obedecido Princípio de Aberto / Fechado. “Se quisermos que um objeto *Cliente* use um objeto de servidor diferente, a classe *Cliente* deve ser alterada para citar a nova classe de servidor.” (MARTIN, 2011, p.142). A Figura 06 apresenta uma solução para resolver este problema, com o Princípio do Aberto / Fechado.

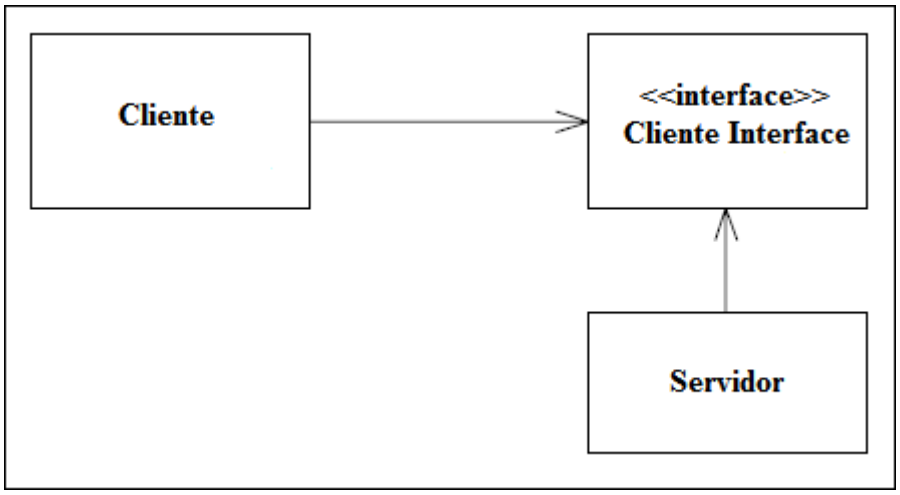


Figura 6 – Cliente é aberta e fechada

Fonte: Adaptada de Martin (2011, p. 143).

Na Figura 06 o princípio é obedecido, e não é mais necessário alterar a classe *Cliente*. Foi criada a interface *Cliente Interface*, que é abstrata, incluindo suas funções membro. A classe *Cliente* implementa essa interface, o que leva aos objetos *Cliente* usarem

objetos da classe derivada *Servidor*. “Se quisermos que objetos *Cliente*, usem uma classe de servidor diferente, uma nova derivada da classe *Cliente Interface* pode ser criada. A classe *Cliente* pode permanecer inalterada.” (MARTIN, 2011, p. 142).

Por fim temos que este princípio pode nos proporcionar os maiores benefícios da orientação a objetos, sendo eles flexibilidade, capacidade de reutilização e facilidade de manutenção. E se o sistema está aberto à adição de funcionalidades, pode-se dizer também que terá maior tempo de vida.

> “A obediência a esse princípio não é obtida simplesmente usando-se uma linguagem de programação orientada a objetos. Também não é recomendável aplicar abstração desenfreada em todas as partes do aplicativo. É necessário dedicação dos desenvolvedores para aplicar abstração somente nas partes do programa que exibem mudanças frequentes. Resistir à abstração precipitada é tão importante quanto a abstração em si”. (MARTIN, 2011, p. 152).

É necessário ter cuidados ao utilizar este princípio, pois assim como o Princípio da Responsabilidade Única, o mau uso pode trazer mais incomodo ao desenvolvedor do que a falta de uso do mesmo.

3.3.3. Princípio da Substituição de Liskov

O Princípio da Substituição de Liskov foi escrito em 1988 por Barbara Liskov, e traz como lema principal que “os subtipos devem ser substituídos pelos seus tipos base.” (MARTIN, 2011, p.153). Ao assumir este comportamento, Martins (2011) afirma que se deve ser capaz de utilizar qualquer classe derivada no lugar da classe pai e obter o comportamento semelhante sem que seja necessária qualquer modificação. O princípio em questão ainda assegura que a classe derivada esteja respeitando o Princípio do Aberto / Fechado, portanto, não deve interferir no comportamento da classe de que herdou.

A Substituição de Liskov segue dois conceitos básicos da orientação a objetos, sendo eles a abstração e o polimorfismo. “Em linguagens estaticamente tipadas, como C#, um dos principais mecanismos que suportam a abstração e polimorfismo é a herança”. (MARTIN, 2011, p. 153). Logo, utilizando a herança é possível criar classe derivadas que herdam comportamentos abstratos da classe mãe.

“A importância desde princípio se torna evidente quando você considera as consequências de sua violação.” (MARTIN, 2011, p. 154). Sendo que temos como principal

ferramenta, para o uso do Princípio da Substituição de Liskov, a herança, é possível explicar a mesma através de exemplos quem possuem herança em si.

Conforme a figura 07 é possível analisar uma violação deste princípio, que consequentemente ainda causa uma violação do Princípio do Aberto / Fechado.

```
public enum ShapeType { circle, square}

public class Shape
{
    private readonly ShapeType _type;

    public Shape(ShapeType type) { _type = type; }

    public void DrawShape(Shape shape)
    {
        if (shape._type == ShapeType.circle)
            (shape as Circle).Draw();
        else if (shape._type == ShapeType.square)
            (shape as Square).Draw();
    }
}
```

```
public class Circle : Shape
{
    public Circle() : base(ShapeType.circle){}

    public void Draw() { /* Implementation */ }
}

public class Square : Shape
{
    public Square() : base(ShapeType.square){}

    public void Draw() { /* Implementation */ }
}
```

Figura 7 – Violação do LSP ocasionando violação do OCP.

Fonte: Adaptado de Martin (2011, p. 154).

"Violar o LSP frequentemente resulta no uso de verificação de tipo em tempo de execução de uma maneira que viola inteiramente o OCP." (MARTIN, 2011, p. 155). Ao violar o LSP no exemplo, a verificação é feita através no trecho de código na classe *Shape*. Em tempo de execução, dentro do escopo de condição *if*, ocorre a verificação do tipo do objeto. Quando isso acontece, é violado também o Princípio de Aberto / Fechado.

> "Claramente a função *DrawShape* (...) viola o OCP. Ela precisa conhecer cada derivada possível da classe *Shape*, e deve ser alterada sempre que novas classes derivadas de *Shape* forem criadas. (...) As classes *Square* e *Circle* derivam de *Shape* e tem funções *Draw()*, mas não sobrescrevem uma função de *Shape*. Como *Circle* e *Square* não podem ser substituídas por *Shape*, *DrawShape* deve inspecionar o objeto *Shape* recebido, determinar o seu tipo e, depois chamar a função *Draw* adequada. (...) Assim, uma violação do LSP é uma violação latente do OCP." (MARTIN, 2011, p. 155)

Analisando e estudando a explicação de Martin sobre a Figura 07, implementei uma das possíveis soluções (conforme Figura 08) utilizando abstração, polimorfismo e herança. Justifico que o autor do exemplo da Figura 07, não exemplificou uma correção.

```
public abstract class Shape
{
    public abstract void DrawShape(Shape shape);
}

public class Circle : Shape
{
    public override void DrawShape(Shape shape)
    {
        /* Implementation */
    }
}
```

```
public class Square : Shape
{
    public override void DrawShape(Shape shape)
    {
        /* Implementation */
    }
}
```

Figura 8 – Corrigindo o exemplo da figura 07

Fonte: o próprio autor.

Na figura 08 é possível observar que com a sobrescrita de método, não é mais necessário verificar o tipo de objeto. No método *DrawShape*, um objeto *Circle* ou *Square* pode agir de diferentes maneiras. Em outra classe que derive de *Shape*, não é necessário alterar a classe base (*Shape*), respeitando assim o OCP.

Por fim temos que o Princípio de Substituição de Liskov é um dos principais fatores concedentes do Princípio do Aberto / Fechado. "A possibilidade de substituição de subtipos permite que um módulo, expresso em termos de um tipo base, seja extensível sem modificação. (...) Assim, o contrato do tipo base precisa ser bem compreendido, se não explicitamente imposto, pelo código." (MARTIN, 2011, p. 169).

3.3.4. Princípio da Segregação de Interface

O Princípio da Segregação de Interfaces possui como lema "Clientes não devem ser forçados a depender de interfaces que eles não irão usar." (MARTIN, 2011, p. 181). Como o próprio nome diz, define-se pela separação das interfaces que podem ser divididas. "Ou seja, as interfaces podem ser divididas em grupos de métodos. Cada grupo atende a um conjunto diferente de clientes. Assim, alguns clientes usam um grupo de métodos e outros clientes usam outros grupos." (MARTIN, 2011, p. 181). Se uma interface pode ser dividida para atender classes diferentes, ela está poluída.

Carr pode complementar o conceito:

> "Muitas vezes, quando você cria uma classe com um grande número de métodos e propriedades , a classe é usada por outros tipos que exigem apenas o acesso a um ou dois membros. As classes se tornam mais acopladas conforme o numero de

> membros é aumentado. Quando você segue o ISP, classes grandes implementam várias interfaces menores agrupam funções de acordo com seu uso. As dependências são vinculadas diminuindo o acoplamento, aumentando robustez, flexibilidade e a possibilidade de reutilização.” (CARR, 2010).

É possível exemplificar esta poluição. Quando temos uma classe C que implemente uma interface I, e não utiliza de todos os seus métodos, então este interface está poluída. Após isso, teremos uma CH que herda desta classe C, para utilizar aquele método de I que não foi utilizado por C. Neste caso, deveria se separar a interface I em duas, e a CH deveria implementar diretamente a nova interface criada.

No anexo A se vê um exemplo de um modelo onde é possível aplicar a ISP. As classes *Transação Deposito*, *Transação Saque* e *Transação Transferência* herdam de *Transação* que é uma classe abstrata. E ao mesmo tempo implementam a interface *UI*. Mas as classes concretas, que são as de específica transação, não utilizam de todos os métodos fornecidos pela interface. O próprio autor explica que:

> “É precisamente essa situação que o ISP nos diz para evitar. Cada uma das transações está usando métodos *UI* que nenhuma outra classe utiliza. Isso gera a possibilidade de que mudanças em uma das classes derivadas de *Transação* obriguem uma mudança correspondente em *UI*, afetando com isso todas as outras derivadas de *Transação* e toda e qualquer outra classe que dependa da interface *UI*.” (MARTIN, 2011, p. 187).

Já no anexo B, foi aplicada a ISP para separar as interfaces. A interface *UI* agora implementa três novas interfaces: *Deposito UI*, *Saque UI* e *Transação UI*. Assim, cada classe concreta implementa apenas a interface que lhe é necessária.

> “Quando uma nova derivada da classe *Transação* for criada, será necessária uma classe base correspondente para a interface *UI* abstrata e, assim, a interface *UI* e todas suas derivadas devem mudar. (...) Essas classes não são amplamente usadas (...), portanto, o impacto da adição de novas classes base de *UI* é minimizado.” (MARTIN, 2011, p. 188).

Interfaces poluídas podem causar confusão se utilizadas aonde todos seus métodos não são necessários. Se uma classe dependente obriga uma interface ser alterada, todos aquelas a implementam devem ser alteradas.

“Assim, os clientes devem depender apenas dos métodos que chamam. Isso pode ser alcançado dividindo-se a interface gorda em muitas interfaces específicas do cliente. (...) Isso

acaba com a dependência dos clientes em relação aos métodos que não chamam, e permite que os clientes sejam independentes uns dos outros.” (MARTIN, 2011, p. 193).

3.3.5. Princípio da Inversão de Dependência

O Princípio da Inversão de Dependência recebe este nome por causa de seus dois conceitos:

a. “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.” (MARTIN, 2011, p. 171).
b. “As abstrações não devem depender de detalhes. Os detalhes devem depender das abstrações.” (MARTIN, 2011, p. 171).
c.

Este princípio existe porque “os métodos de desenvolvimento de software mais tradicionais, como o projeto e análise estruturados, tendem a criar estruturas de software nas quais os módulos de alto nível dependem de módulos de módulos de baixo nível.” (MARTIN, 2011, p. 171). Quando temos módulos de alto nível, são estes que devem influenciar os de baixo nível, e não vice-versa. Os módulos de alto nível são os que contêm as regras de negócio, e devem ser independentes dos de baixo nível, onde estão os detalhes da na codificação. Martin porque é necessário que haja essa independência:

> “São os módulos que definem diretivas de alto nível que queremos reutilizar. (...) Quando módulos de alto nível dependem de módulos de baixo nível, torna-se muito difícil reutiliza-los em diferentes contextos. Contudo, quando os módulos de alto nível são independentes dos módulos de baixo nível, eles podem ser reutilizados com muita simplicidade.” (MARTIN, 2011, p. 172).

É possível analisar o exemplo da Figura 09 para explicar um problema, e uma solução com a Inversão de Dependência. No exemplo, existem a *Camada de Política*, que é uma camada de alto nível onde temos as regras de negócio; após esta, a *Camada de Mecanismo*, que é uma camada de nível inferior; e por fim a *Camada de Utilidade*, a de baixo nível e detalhamento, onde é possível encontrar códigos que podem ser reutilizados como bibliotecas e sub-rotinas.

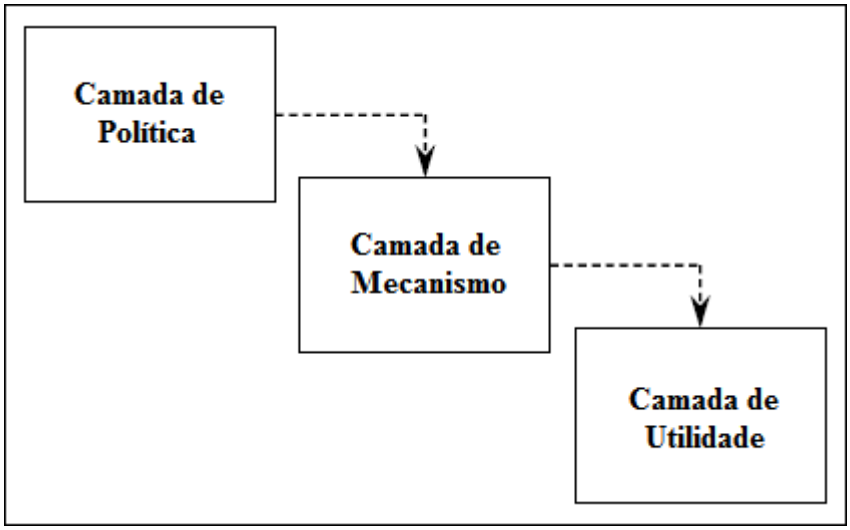


Figura 9 – Esquema de disposição em camadas simplista.

Fonte: Adaptada de Martin (2011, p. 172).

"Nesse diagrama, a camada *Política* de alto nível utiliza uma camada *Mecanismo* de nível inferior, a qual por sua vez utiliza uma camada *Utilidade* de nível detalhado. Embora isso pareça adequado, (...) a camada *Política* é sensível a alterações feitas na camada inferior *Utilidade*." (MARTIN, 2011, p. 172). Isto é correto, pois a dependência é uma característica transitiva. Assim, quando a camada *Política* depende diretamente de algo que depende da camada *Utilidade*, esta depende transitivamente da camada de baixo nível.

Para corrigir isso, são necessárias algumas modificações. A figura 10 mostra um modelo mais adequado.

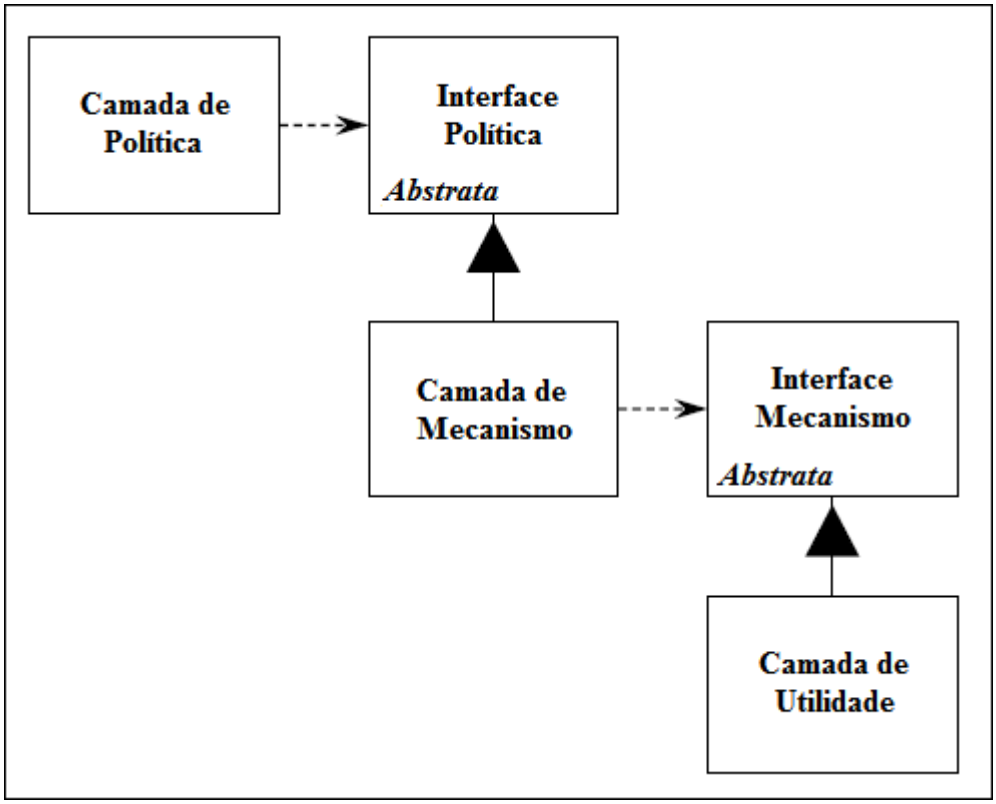


Figura 10 – Camadas invertidas

Fonte: Adaptada de Martin (2011, p. 173).

"Cada camada de nível superior declara uma interface abstrata para os serviços de que precisa. Então as camadas de nível superior são concretizadas a partir de interfaces abstratas. Cada classe de nível superior utiliza a camada de nível mais baixo seguinte, por meio de interface abstrata." (MARTIN, 2011, p. 173). Modificando o exemplo desta maneira, as camadas superiores passam a não depender das inferiores. Ambas as dependências, direta e transitiva, foram eliminadas. Olhando para a figura 10, é possível ver que agora as camadas de baixo nível dependem de interfaces abstratas, respeitando os dois conceitos citados no começo deste capítulo. "Assim, invertendo as dependências, criamos uma estrutura que é simultaneamente mais flexível, durável e móvel." (MARTIN, 2011, p. 173).

Na dependência de abstrações, Martin (2011) diz que uma interpretação do DIP, um tanto mais simplista, apesar de ainda muito poderosa, é a heurística "depender de abstrações". Dito de forma simples, essa heurística recomenda que todos os relacionamentos em um programa devessem terminar em uma classe ou interface abstrata. Em outras palavras, não se deve depender de uma classe concreta, e sim fazer os relacionamentos por uma interface, como apresentado nas figuras 09 e 10. Segundo Martin (2011) nessa heurística temos que:

- Nenhuma variável deve conter uma referência para uma classe concreta;
- Nenhuma classe deve derivar de uma classe concreta; e
- Nenhum método deve sobrescrever um método implementado de qualquer uma de suas classes base;

É visto que nessa forma de análise, é possível que a heurística seja violada ao menos uma vez dentro do projeto, pois em alguma parte do programa é necessário criar instâncias das classes concretas. Mas é possível evitar este fato.

> "Se você puder usar strings para criar classes. A linguagem C# e outras tantas permitem isso. Nessas linguagens, os nomes das classes concretas podem ser passados para o programa como dados de configuração. (...) Por exemplo, a classe que descreve uma string é concreta. (...) Essa classe não é volátil. Isto é, ela não muda com muita frequência. Portanto não causa dano depender diretamente dela." (MARTIN, 2011, p. 174).

Não se pode depender então, de uma classe concreta que seja volátil, que se modifique constantemente e frequentemente. Mas a maioria das classes concretas que é criada dentro de um programa é volátil. "Sua volatilidade pode ser isolada, mantendo-as atrás de uma interface abstrata." (MARTIN, 2011, p. 174).

Mas Martin ainda explica que "essa não é uma solução completa. Existem ocasiões em que a interface de uma classe volátil deve mudar, e essa mudança deve ser propagada para a interface abstrata que representa a classe. Tais alterações forçam o isolamento da interface abstrata." (MARTIN, 2011, p. 174). Isso prova que esta heurística possui brecha, mas ainda sim pode prevenir a regra principal da DIP. Tomando como exemplo a figura 11, temos que a *Interface Mecanismo* somente irá mudar quando o módulo *Mecanismo* sofrer alterações.

"O Princípio da Inversão de Dependência é o mecanismo de baixo nível fundamental por trás de muitos benefícios reivindicados pela tecnologia orientada a objetos. Sua aplicação correta é necessária para a criação de *frameworks*[4] reutilizáveis." (MARTIN, 2011, p. 179). Na ocorrência da aplicação de DIP no desenvolvimento de *software*, as abstrações são isoladas dos detalhes da codificação, resultando em um código muito mais fácil de manter.

[4] Uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.

4. LEVANTAMENTO E RESULTADOS DA PESQUISA

Para o desenvolvimento do questionário de pesquisa, foram levados em consideração os conceitos abordados durante o desenvolvimento deste trabalho, e contato telefônico com gerentes de projeto das empresas A e B. Tendo esses dois apoios, o questionário foi desenvolvido, se tornando um meio de pesquisa com possibilidade de obter resultados concretos.

Conforme levantamento realizado com o anexo C (formulário de pesquisa) foi obtido como resultados, tanto valores esperados como valores inesperados. Abaixo se apresentam os resultados parciais. Os entrevistados foram dez profissionais de duas empresas, sendo eles cinco de cada empresa, cuja experiência varia de um ano até mais de quatro anos. Esta quantidade é considerada boa para resultados, pois a pesquisa se trata de um estudo por amostragem, onde a representação de um todo apresenta margem de erro aceitável.

4.1. Resultados do levantamento sobre Boas Práticas de Desenvolvimento de Software

Tabela 1: Resultados do levantamento de Boas Práticas

Ao nomear variáveis, funções ou classes: estes nomes revelam o propósito de uso e criação das mesmas?	
40%	Sempre
60%	Quase Sempre
0%	Raramente
0%	Nunca
Ao nomear variáveis: você evita que estes nomes possam gerar confusão?	
10%	Sempre
60%	Quase Sempre
20%	Raramente
10%	Nunca
Ao nomear variáveis, funções ou classes: estes nomes podem ser lidos e pronunciados facilmente?	
40%	Sempre
60%	Quase Sempre
0%	Raramente
0%	Nunca
Ao nomear métodos de mesmo nome em classes distintas: você evita que estes métodos possuam a	

mesma assinatura?	
20%	Sempre
50%	Quase Sempre
30%	Raramente
0%	Nunca

Ao nomear variáveis dentro de um escopo: além de seus nomes serem diferentes, os nomes têm os significados distintos?	
40%	Sempre
20%	Quase Sempre
30%	Raramente
10%	Nunca

Ao nomear variáveis dentro de um laço de repetição: você evita que esses nomes possuam uma letra ou uma letra mais um número?	
30%	Sempre
60%	Quase Sempre
10%	Raramente
0%	Nunca

Ao nomear variáveis dentro de escopos, e laços de repetição: esses nomes são passíveis de busca?	
0%	Sempre
70%	Quase Sempre
30%	Raramente
0%	Nunca

Ao nomear classes: com que frequência os nomes utilizados são substantivos?	
10%	Sempre
60%	Quase Sempre
20%	Raramente
10%	Nunca

Ao nomear métodos ou funções: com que frequência os nomes utilizados possuem um verbo?	
20%	Sempre
80%	Quase Sempre
0%	Raramente
0%	Nunca

Fonte: o próprio autor

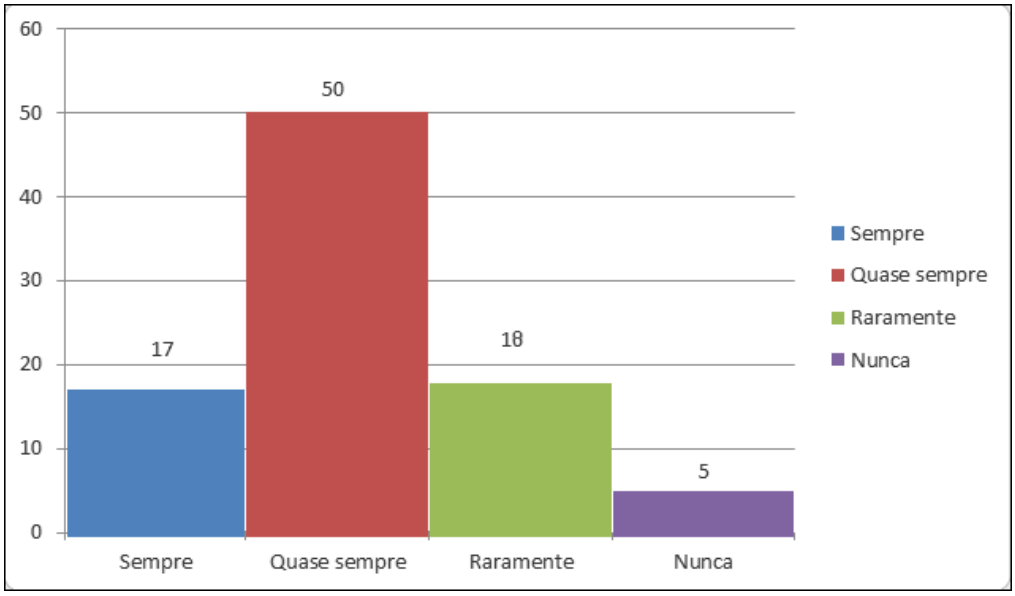


Figura 11 – Gráfico resultado das pesquisas sobre Boas Práticas de Desenvolvimento

Fonte: o próprio autor.

Pela análise da Figura 11, é possível obter a informação de que o uso das boas práticas nas empresas de desenvolvimento de *software*, quanto ao quesito de nomeação dentro do código, está sendo seguido de forma quase completa, pela maioria dos desenvolvedores. As respostas apontadas como “Sempre” e “Quase sempre” eram as respostas positivas quanto ao uso da nomeação de forma correta. Já as apontadas como “Nunca” e “Raramente” eram as respostas negativas.

4.2. Resultados do levantamento sobre o conhecimento e uso dos Princípios SOLID

4.2.1. “Sobre o Princípio da Responsabilidade Única (SRP)”

Tabela 2: Resultados sobre SRP – Formulário sobre Princípios SOLID.

Sobre o Princípio da Responsabilidade Única (SRP):	
40%	Tenho conhecimento e utilizo deste princípio
40%	Não tenho conhecimento.
20%	Tenho conhecimento, mas não pratico o uso.

Fonte: o próprio autor.

Dos entrevistados que responderam “Não tenho conhecimento”, 75% têm mais de 04 anos de experiência profissional. Dos que responderam “Tenho conhecimento e utilizo deste princípio”, 50% tem mais de 04 anos de experiência profissional, enquanto 25% têm de 01 a 02 anos e o restante de 02 a 04 anos.

4.2.2. “Sobre o Princípio do Aberto-Fechado (OCP)”

Tabela 3: Resultados sobre OCP – Formulário sobre Princípios SOLID.

Sobre o Princípio do Aberto-Fechado (OCP):	
30%	Tenho conhecimento e utilizo deste princípio
50%	Não tenho conhecimento.
20%	Tenho conhecimento, mas não pratico o uso.

Fonte: o próprio autor.

Dos entrevistados que responderam “Não tenho conhecimento”, 60% têm mais de 04 anos de experiência profissional. Dos que responderam “Tenho conhecimento e utilizo deste princípio”, 33,4% tem mais de 04 anos de experiência profissional, enquanto 33,3% têm de 01 a 02 anos e o restante de 02 a 04 anos.

4.2.3. “Sobre o Princípio da Substituição de Liskov (LSP)”

Tabela 4: Resultados sobre LSP – Formulário sobre Princípios SOLID.

Sobre o Princípio da Substituição de Liskov (LSP):	
20%	Tenho conhecimento e utilizo deste princípio
80%	Não tenho conhecimento.
0%	Tenho conhecimento, mas não pratico o uso.

Fonte: o próprio autor.

Dos entrevistados que responderam “Não tenho conhecimento”, 50% têm mais de 04 anos de experiência profissional. Dos que responderam “Tenho conhecimento e utilizo deste princípio”, 50% tem mais de 04 anos de experiência profissional, enquanto 25% de 02 a 04 anos.

4.2.4. “Sobre o Princípio da Segregação de Interface (ISP)”

Tabela 5: Resultados sobre ISP – Formulário sobre Princípios SOLID.

Sobre o Princípio da Segregação de Interface (ISP):	
30%	Tenho conhecimento e utilizo deste princípio
50%	Não tenho conhecimento.
20%	Tenho conhecimento, mas não pratico o uso.

Fonte: o próprio autor.

Dos entrevistados que responderam “Não tenho conhecimento”, 80% têm mais de 04 anos de experiência profissional. Dos que responderam “Tenho conhecimento e utilizo deste princípio”, 33,3% tem mais de 04 anos de experiência profissional, enquanto o restante tem de 01 a 02 anos.

4.2.5. “Sobre o Princípio da Inversão de Dependência (DIP)”

Tabela 5: Resultados sobre DIP – Formulário sobre Princípios SOLID.

Sobre o Princípio da Inversão de Dependência (DIP):	
20%	Tenho conhecimento e utilizo deste princípio
50%	Não tenho conhecimento.
30%	Tenho conhecimento, mas não pratico o uso.

Fonte: o próprio autor.

Dos entrevistados que responderam “Não tenho conhecimento”, 60% têm mais de 04 anos de experiência profissional. Dos que responderam “Tenho conhecimento e utilizo deste princípio”, 50% tem mais de 04 anos de experiência profissional, enquanto o restante tem de 02 a 04 anos.

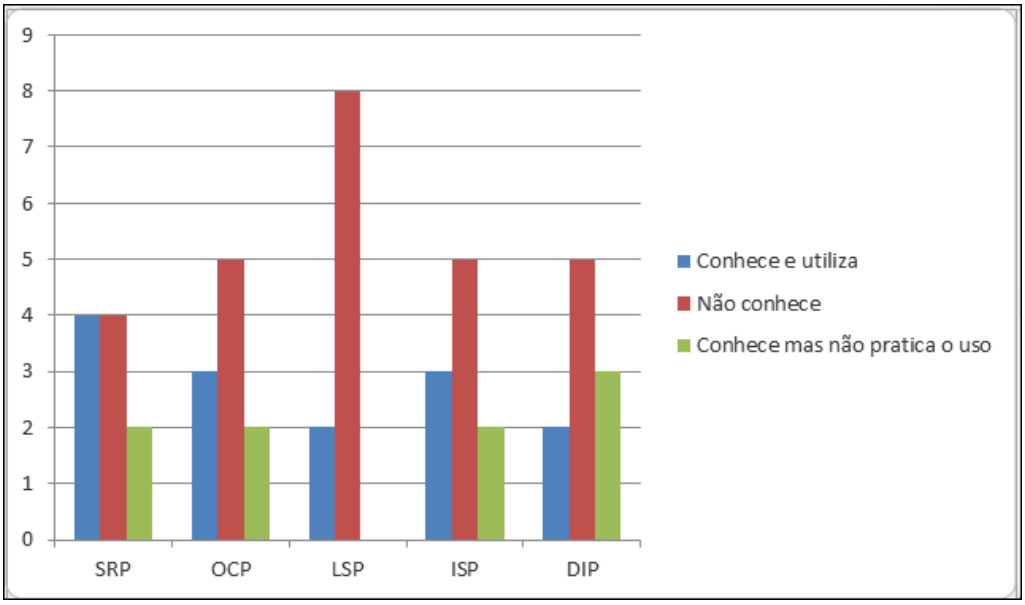


Figura 12 - Resultado do levantamento sobre Princípios SOLID

Fonte: o próprio autor.

Analisando os resultados apresentados na figura 12, e sua associação com o tempo de experiência profissional de cada entrevistado, é possível observar que na maioria das vezes, não existe o conhecimento dos Princípios SOLID. Aqueles que conhecem e utilizam os princípios são pouco mais dos que conhecem, mas não tem prática em utiliza-los.

A observação em cima do resultado, somado com a observação da experiência profissional, nos trás que existem muitos desenvolvedores experientes (mais de 04 anos) que desconhecem os Princípios SOLID. Isso pode ocorrer devido à falta de aperfeiçoamento, ou ao fato de os desenvolvedores experientes serem pouco flexíveis as mudanças e novas técnicas. Era esperado que os desenvolvedores com maior experiência tivessem um maior conhecimento sobre esses princípios, pois é fato que para conhecimento e aplicação correta, os princípios necessitam de prática e estudo.

Por outro lado, existe desenvolvedores com menor tempo de experiência profissional, que já conhecem dos princípios, e também uma parte que os aplica em seu desenvolvimento profissional. Mas o desconhecimento, em parte se aplica também a alguns desenvolvedores mais jovens. É necessário aproveitar enquanto não criou “manias” durante a codificação, para que se possa estudar e aplicar os conceitos dos Princípios SOLID.

5. CONCLUSÃO

Para concluir este estudo, é visto que o desenvolvimento de *software*, por ser uma ciência exata, necessita de regras e conceitos, que tenham como objetivo facilitar a manutenção e o entendimento dos desenvolvedores ao trabalhar com um código escrito por outra pessoa. Este trabalho foi desenvolvido com finalidade de proporcionar um conhecimento maior sobre as boas práticas de desenvolvimento de *software*, e princípios SOLID, além de apresentar por meio de levantamento e amostra, que o uso destes, ainda não é totalmente aplicado.

Como afirma Martin (2009), desenvolver um código bem escrito é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões. Analisado o pensamento de Martin, acrescenta-se que alguns desenvolvedores que possuem uma longa carreira e experiência profissional, ainda desconhecem a existência destes princípios e práticas. E outros profissionais da área, mesmo possuindo um tempo menor de experiência, já se aperfeiçoaram, e utilizam os mesmos. Este acontecimento decorre na maioria das vezes pelo fato de que desenvolvedores mais experientes tenham práticas de desenvolvimento que já são seguidas há anos.

Com este trabalho pode-se aperfeiçoar os conhecimentos sobre os princípios e práticas que podem ajudar o desenvolvedor, e explicar de uma maneira sucinta e breve, como utiliza-los. O estudo nesta área é importante, pois traz diversos benefícios ao desenvolvimento em conjunto, tornando-o mais ágil e eficaz, facilitando também a descoberta de erros no sistema, antes que cheguem ao usuário final.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDRADE, Margarida. **Introdução a Metodologia do Trabalho Científico**. 10. ed. São Paulo: Atlas, 2010.

Bjarne Stroustrup Homepage. Disponível em: [http://www.stroustrup.com/]. Acesso em: 09 de out. 2012.

C# Interface Naming Guidelines. Disponível em: [http://msdn.microsoft.com/en-us/library/8bc1fexb%28v=vs.71%29.aspx]. Acesso em: 29 de set. 2012.

CARR, Richard. **The SOLID Principles**. Disponível em: [http://www.blackwasp.co.uk/SOLID.aspx]. Aceso em: 14 de out. 2012

Dave Thomas Homepage. Disponível em: [http://www.davethomas.net/]. Acesso em: 09 de out. 2012.

Documentação Microsoft VBScript Code Conventions. Disponível em: http://lebeau.home.ml.org. Acesso em: 28 de set. 2012.

GIL, A. C. **Como elaborar projetos de pesquisa**. 4. ed. São Paulo: Atlas, 2002.

HEERDT, Mauri Luiz; LEONEL, Vilson. **Metodologia Científica e da Pesquisa**. 5. ed. Florianópolis: Unisul Virtual, 2007.

HENDERSON, Cal. **Building Scalable Web Sites**. California: O'reilly Media, 2006.

KÖCHE, J. C. **Fundamentos da metodologia científica**: teoria da ciência e prática de pesquisa. 14. ed. Petrópolis: Vozes, 2006.

LARMAN, Craig. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2. ed. New Jersey: Prentice Hall, 2001.

MARTIN, Robert C. **Código limpo**. São Paulo: Alta Books, 2010.

MARTIN, Robert C. **Getting a SOLID Start**. Disponível em: [http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start]. Acesso em 15 de out. 2012

MARTIN, Robert C. **Princípios, Padrões e Práticas Ágeis em C#**. São Paulo: Bookman, 2011.

MARTIN, Robert C. **The Principles of OOD**. Disponível em: [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod]. Acesso em 19 de out. de 2012.

MARTIN, Robert C. **Design Principles and Design Patterns**. Disponível em: [www.objectmentor.com]. Acesso em 19 de out. 2012.

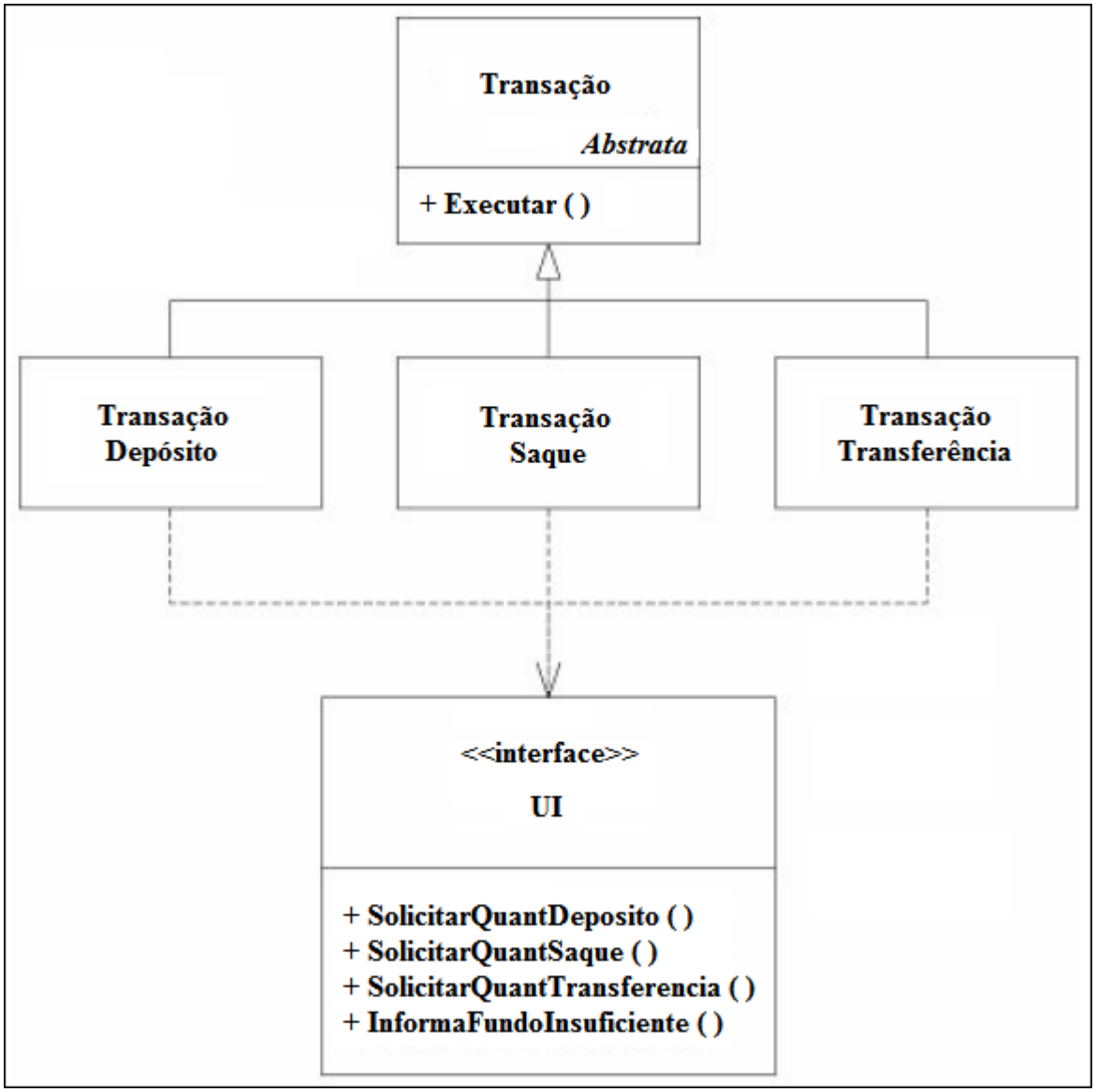
MARTINS, Denis. **Princípios SOLID**. Disponível em: [http://denmartins.wordpress.com/2011/07/19/solid-principles/]. Acesso em 15 de out. 2012.

MOHTASHIM, A. **Software Developer's Best Practices**. Disponível em: [http://www.tutorialspoint.com/developers_best_practices] Acesso em: 09 de out. 2012.

ANEXOS

ANEXO A

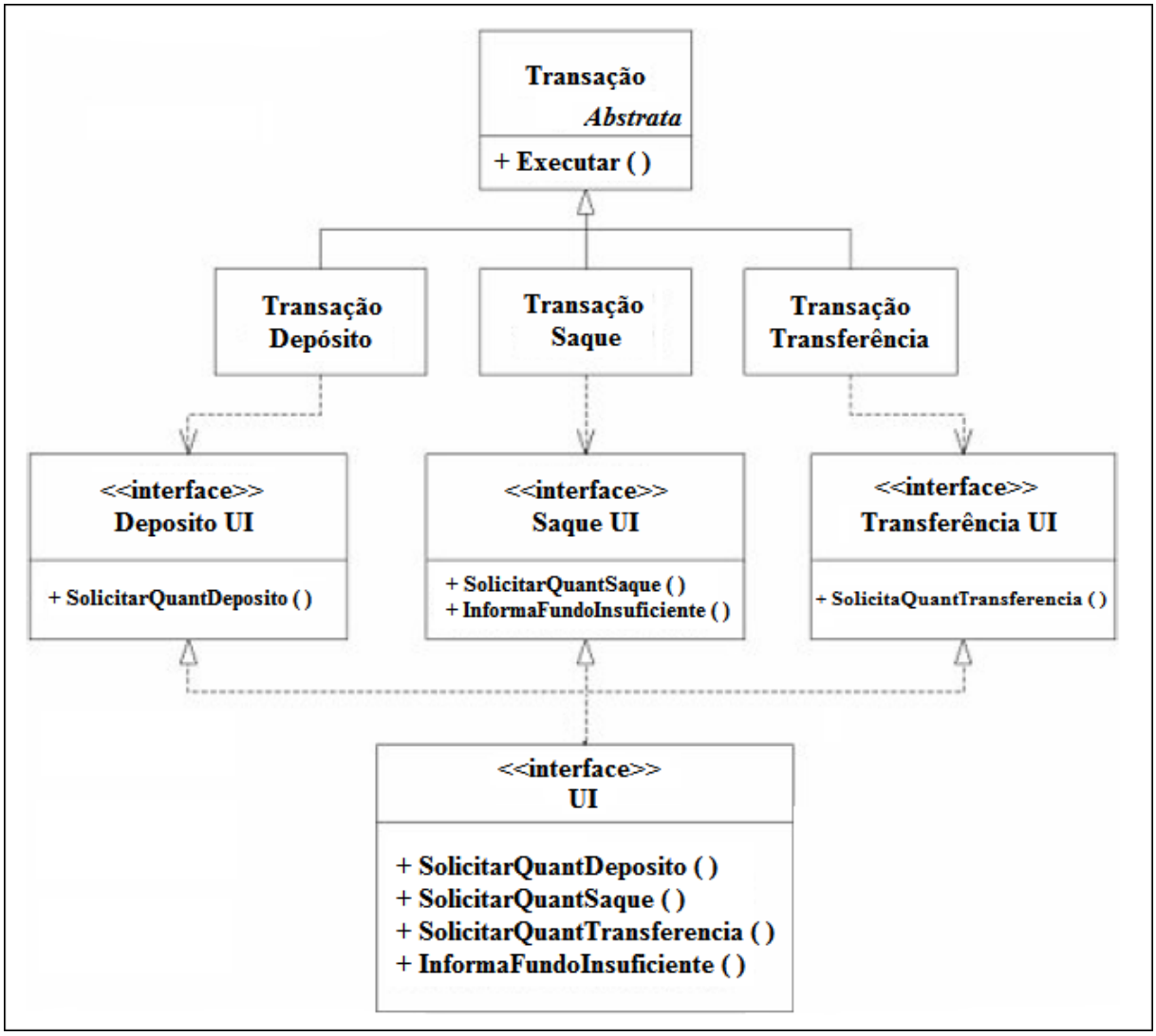
Hierarquia de transações do caixa eletrônico



Fonte: Adaptada de Martin (2011, p. 188).

ANEXO B

Interface UI segregada para caixa eletrônico



Fonte: Adaptada de Martin (2011, p. 189).

ANEXO C

Formulário | Boas práticas de Desenvolvimento e Princípios SOLID

Este formulário foi desenvolvido com base nos conceitos abordados no decorrer desde trabalho e colaboração de gerentes de projeto. Possui como único objetivo, ser fonte de pesquisa e levantamento sobre o uso das boas práticas de desenvolvimento e princípios SOLID. O primeiro nome dos desenvolvedores que responderem não será divulgado sobre hipótese alguma. Serve apenas para controle de respostas.

1. Ao nomear variáveis, funções ou classes: estes nomes revelam o propósito de uso e criação das mesmas?
2. Ao nomear variáveis: você evita que estes nomes possam gerar confusão?
3. Ao nomear variáveis, funções ou classes: estes nomes podem ser lidos e pronunciados facilmente?
4. Ao nomear métodos de mesmo nome em classes distintas: você evita que estes métodos possuam a mesma assinatura?
5. Ao nomear variáveis dentro de um escopo: além de seus nomes serem diferentes, os nomes têm os significados distintos?
6. Ao nomear variáveis dentro de um laço de repetição: você evita que esses nomes possuam uma letra ou uma letra mais um número?
7. Ao nomear variáveis dentro de escopos, e laços de repetição: esses nomes são passíveis de busca?
8. Ao nomear classes: com que frequência os nomes utilizados são substantivos?
9. Ao nomear métodos ou funções: com que frequência os nomes utilizados possuem um verbo?
10. Sobre o Princípio da Responsabilidade Única (SRP)
11. Sobre o Princípio do Aberto-Fechado (OCP)
12. Sobre o Princípio da Substituição de Liskov (LSP)
13. Sobre o Princípio da Segregação de Interface (ISP)
14. Sobre o Princípio da Inversão de Dependência (DIP)