CENTRO UNIVERSITÁRIO DE DESENVOLVIMENTO DO CENTRO OESTE – UNIDESC
SISTEMAS DE INFORMAÇÃO

APLICAÇÃO DE SOLID, ARQUITETURA DE SOFTWARE E PADRÕES DE PROJETO

Ana Júlia Leite

Luziânia-GO
Junho/2024

UNIDESC
Centro Universitário de Desenvolvimento do Centro-Oeste

ANA JÚLIA LEITE

Trabalho apresentado ao Centro Universitário

De Desenvolvimento do Centro-Oeste –

UNIDESC, no curso de Sistemas de Informação,

na disciplina Programação Orientada a Objetos 2

Orientador: **Thallys Braz**

SUMÁRIO

1- INTRODUÇÃO

Este trabalho segue um roteiro de três etapas e a primeira etapa é para apresentar os conceitos de SOLID, fundamentos de *software*, os tipos comuns desta arquitetura, arquitetura em camadas, arquitetura orientada a eventos, microempresas e etc.

A segunda etapa do trabalho apresenta os conceitos dos padrões de projeto do *MVC*, *Singleton*, *Factory*, *Proxy* ou *Facade*.

A terceira parte deste trabalho é fazendo a integração com SOLID e Arquitetura de *software*, explicando como os princípios do *SOLID* são aplicados na proposta de arquitetura e implementação do padrão. Será apresentado a forma como esses princípios ajudam a resolver problemas de *design* específico para este trabalho, explicando as vantagens e as desvantagens de se utilizar este software que foi criado por mim.

Este trabalho será apresentado apenas em forma conceitual, logo não haverá nenhum tipo de parte prática, não havendo nenhum tipo de código, apenas algumas ilustrações para facilitar o entendimento sobre o assunto.

2- 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 esse 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 a 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 de boas práticas de desenvolvimento de *Software*, as quais você deve praticar o uso frequente que vire um hábito, os princípios do *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 mesmos forem aplicados de forma mecânica, se tornam tão ruins como se não fossem aplicadas ao todo.

"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 consagrada de engenharia de *software*." (MARTIN, 2011 p. 122).

Os princípios *SOLID* são cinco gerenciamento 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 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).

A 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.

2.1- 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', que definiam como finalidade funcional dos elementos de um módulo." (MARTIM, 2011, p. 135). Então, para haver coesão, é necessário que os componentes dentro de uma classe tenham finalidade funcional. É a partir desta afinidade funcional – que os membros têm sua função relacionada com a de outros membros da classe – que teremos a responsabilidade única.

Segundo Martim (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 haver 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 de 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. Martim diz que "este tipo de acoplamento leva a projetos frágeis que entregam de maneiras inesperadas quando alterados." (MARTIM, 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.

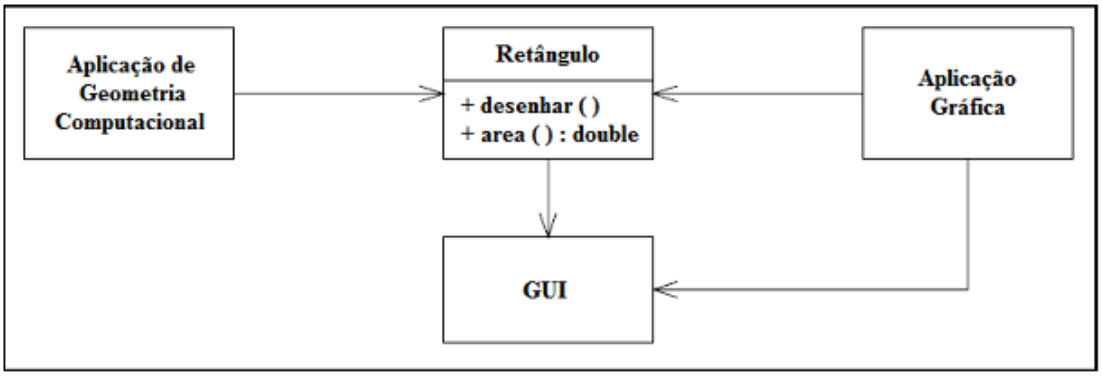

Figura 1 – Mais de uma responsabilidade

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

A classe *Retângulo,* representada na Figura 1, 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 1, é notável que duas aplicações utilizam da classe retângulo: *a aplicação gráfica de geometria computacional*. O modo que este esquema foi montado viola o Princípio da 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 da *GUI*. Um fato ocorrente é que os métodos da classe, não possuem finalidade 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." (MARTIM, 2011, p. 136).

O problema ocorre porque a responsabilidade de desenhar o retângulo ficou altamente acolado com a responsabilidade de realizar o cálculo na área. Ao aplicar o Princípio da Responsabilidade Única, temos uma reconstrução do projeto para evitar problemas futuros, como mostramos na Figura 2:

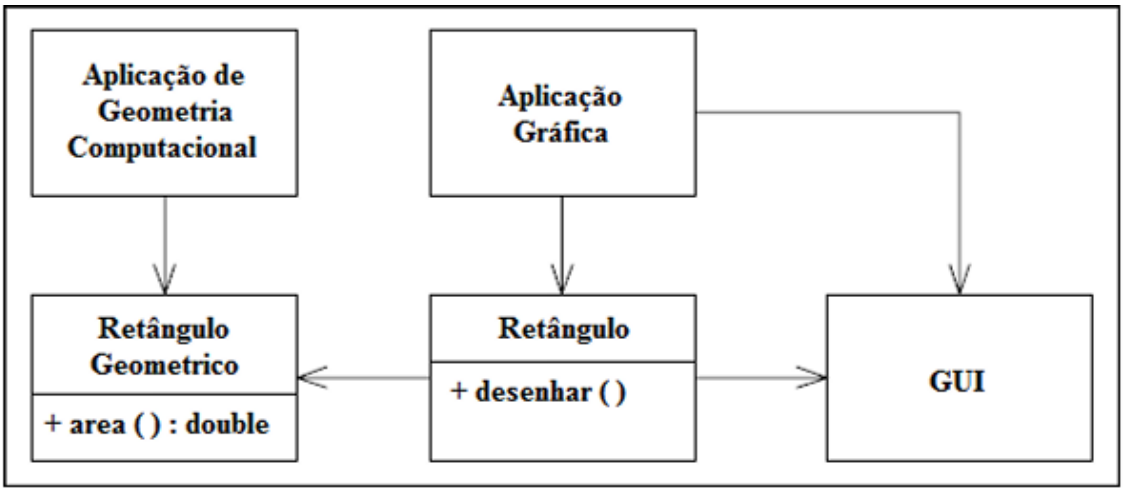

Figura 2 – Responsabilidades separadas

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." (MARTIM, 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, este risco é reduzido e limitado.

2.2- Princípio 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. Este princípio diz que “as entidades de software (classes, módulos, funções, etc.) devem ser abertas para ampliação, mas fechada para modificação.” (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 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 fechadas para modificação, pois ao realizar a ampliação do módulo em questão, não existe uma mudança no seu código fonte. “A versão em binária executável do módulo – seja em uma biblioteca que possa 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 dela 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 orientado a objetos, é possível criar abstrações fixas e que ainda sim representam um grupo limitado 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, na Figura 3 e 4, é possível explicar este princípio.

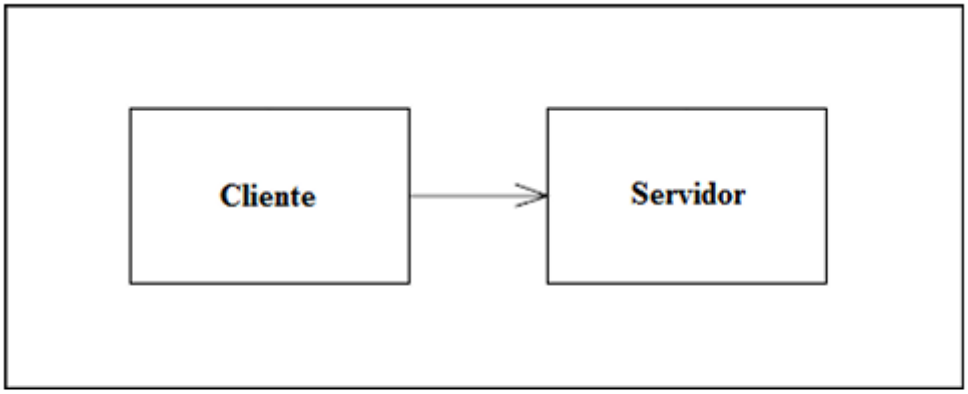

Figura 3 – Cliente não é aberta e fechada.

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

O exemplo simples da Figura 3 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 do 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 4 apresenta uma solução para resolver este problema, com o Princípio Aberto/Fechado.

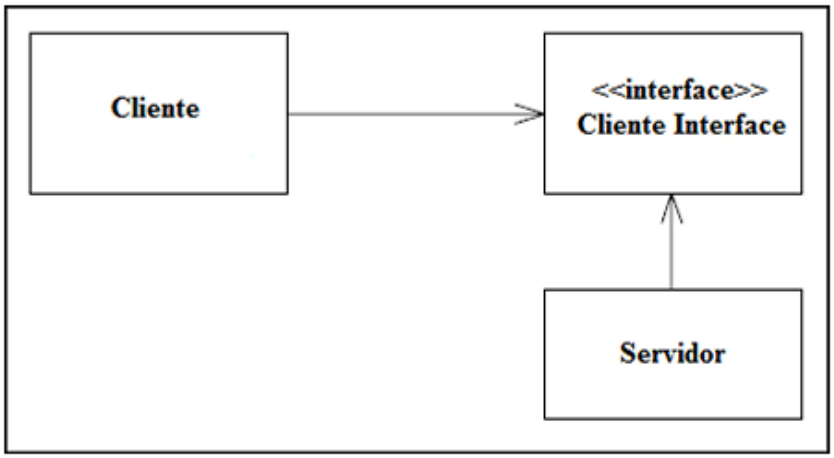

Figura 4 – Cliente é aberta e fechada

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

Na Figura 4 o princípio é obedecido, 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 desenfreadas 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 participada é tão importante quanto a abstração em si.” (MARTIN, 2011, p. 152).

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

2.3- PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

O Princípio da Substituição de Liskov foi escrito em 1988 por Bárbara Liskov, e traz como lema principal que “os subtipos devem ser substituídos pelos seus tipos de base.” (MARTIN, 2011, p. 153). Ao assumir este comportamento, Martin (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 rejeitando o Princípio 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 deste 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 5 é possível analisar uma violação deste princípio, que consequentemente ainda causa uma violação de 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 5 – 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 o princípio do 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* foram criadas, (...) As classes *Square* e *Circle* derivam de *Shape* e tem função *Draw*(), mas não subscrevem uma função de *Shape*. Como *Circle* e *Square* não devem 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 5, implementei uma das possíveis soluções (conforme a Figura 6) utilizando abstração, polimorfismo e herança. Justifico que o autor do exemplo da Figura 5, 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 6 – Corrigindo o exemplo da figura 5

Na Figura 6 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 deriva de *Shape,* não é necessário alterar a classe base (*Shape*), respeitando assim o *OCP*.

Por fim temo que o Princípio da Substituição de Liskov é um dos principais fatores concedentes do Princípio do Aberto/Fecado. “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).

2.4- 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, defina-se pela separação das interfaces que podem ser divididas em grupos de métodos. Cada grupo atende a um conjunto diferentes 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 número de membros é aumentado. Quando você segue o *ISP*, classes grandes implementam várias interfaces menores agrupam funções de acordo com o 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 esta interface está poluída. Após isso, teremos uma classe CH que herda essa classe C, para utilizar aquele método de I que não foi utilizado por C. Neste caso, deveria se separar da 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 Depósito*, *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 explicada a *ISP* para evitar as interfaces. A interface *UI* agora implementa três novas faces: *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 foi criada, será necessária uma classe base correspondente para a interface UI abstrata e, assim, a interface UI e, todas as 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 onde todos os seus métodos não são necessários. Se uma classe dependente obriga uma interface ser alterada, todas 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).

2.5- PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIA

O Princípio da Inversão da 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 de 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 baixo nível.” (MARTIN, 2011, p. 171). Quando temos módulos de alto nível, são estes que devem influenciar os de baixo nível, e devem ser independentes dos de baixo nível, onde estão os detalhes na codificação. 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 7 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 a regra de negócio; após esta, a *Camada de Mecanismo*, que é uma camada de nível inferior; e pôr 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.

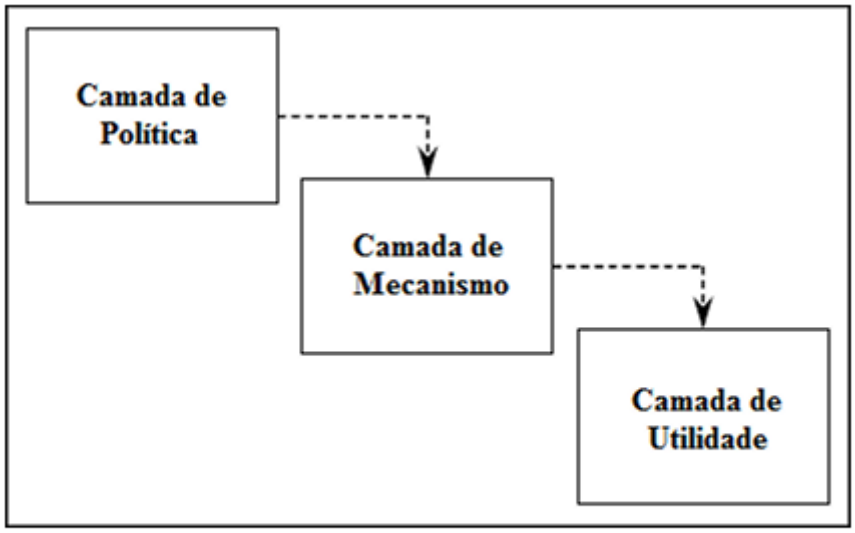

Figura 7 – Esquema de disposição em camadas simplista.

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

"Neste diagrama, a camada *Política* de alto nível utilizada como 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 7 mostra um modelo mais adequado.

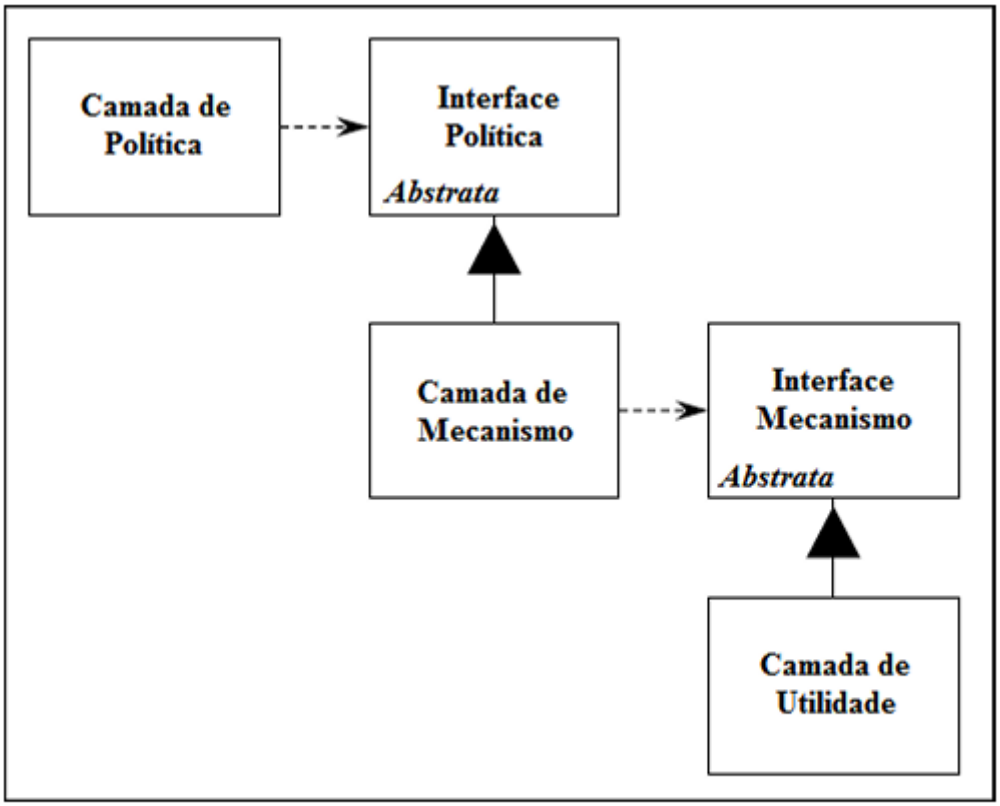

Figura 8 – Camadas invertidas

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 interfaces. Ambas as dependências, direta e transitiva, foram eliminadas. Olhando para a Figura 8, 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 apresentados nas Figuras 7 e 8. 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 subscrever um método implementado de qualquer uma das classes base;

É visto dessa 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 esse fato.

> "Se não puder usar *strings* para criar classes. A linguagem *C#* e outras tantas prometem isso. Nessas linguagens, os nomes das classes concretas podem ser passados para o programa como dados de configurações (...) por exemplo, a classe que descrevem uma *string* é concreta (...) essa classe não é volátil. Isto é, ela não muda com muita frequência. Portanto, não causa danos 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).

Martin ainda explica que "essa não é uma solução completa. Existam 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 foram o isolamento da interface abstrata". (MARTIN, 2011, p. 174). Isso prova que essa heurística possui brecha, mas ainda sim pode prevenir a regra principal da DIP. Tomando como exemplo a Figura 8, temos que a

Interface Mecanismo somente irá mudar quando o módulo *Mecanismo* sofrer alterações.

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

3- Conceito de *MVC*

O padrão *MVC* divide a aplicação em três camadas independentes a fim de melhorar a flexibilidade e o reuso do código.

O *Model*, ou modelo, é a camada lógica que contém a funcionalidade e os dados da aplicação, ocorrendo a manipulação desses dados. Ela é o núcleo da aplicação, pois é onde fica as regras de negócio.

A *View*, ou visualização, é a camada de interação do usuário. Ela exibe a interface gráfica do usuário e a utiliza conforme interações de usuários e modificações no modelo de dados. Este padrão facilita a integração do modelo de dados com diferentes implementações de visualização.

O *Controller*, ou controlador, é a camada que define o modelo que a interface do usuário irá reagir de acordo com suas interfaces. É ela que faz a “ponte” entre a *view* e o *model* conforme as requisições do usuário.

O *MVC* desacopla as *views* e os *models* estabelecendo um protocolo de *subscribe*/*notify* entre eles, portanto sempre que os dados de um modelo mudarem todas as *views* que dependerem deles serão notificadas. Dessa forma, as *views* devem refletir a qualquer mudança de estado por parte do *model*. A Figura 9 ilustra um diagrama da relação entre *model* e *views.*

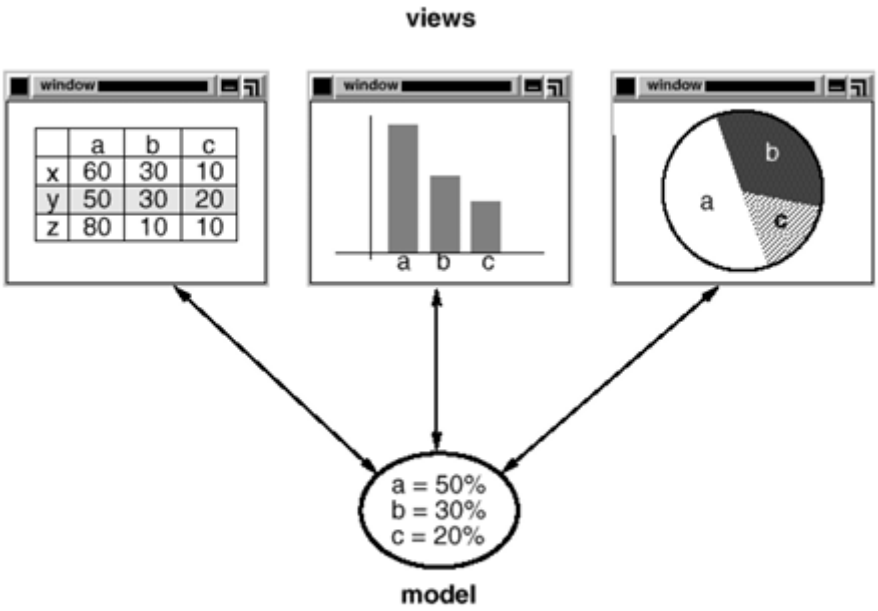

Figura 9: Diagrama de relação entre *model* e *views*.

A relação entre seus componentes é representada na Figura 10.

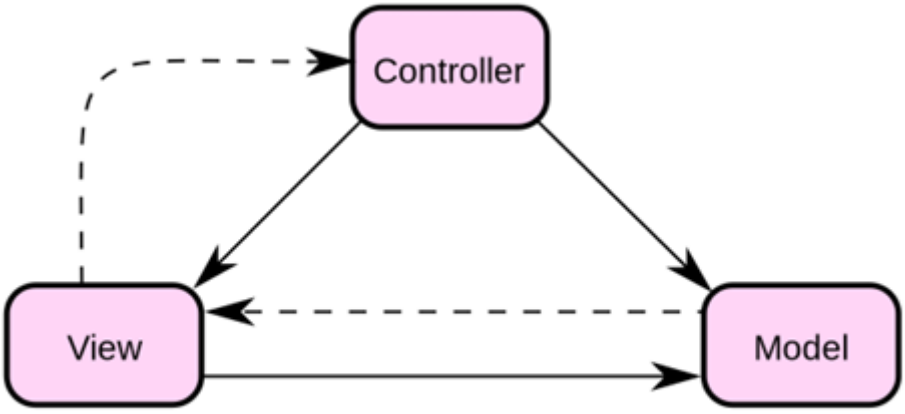

Figura 10: Diagrama de relação dos componentes do MVC.

3.1- SINGLETON

O padrão *Singleton* tem como objetivo “assegurar que uma classe tenha somente uma instância, e provar um ponto de acesso global a ela.”

É importante para algumas classes que se tenha apenas uma instância dela. A solução encontrada por esse padrão é que a classe seja responsável pala sua própria instanciação, fornecendo uma forma de acessá-la. A estrutura desse padrão é ilustrada na Figura 11.

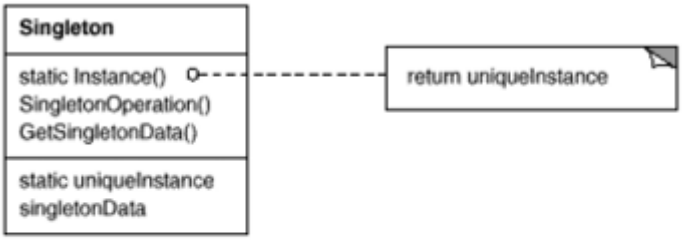

Figura 11: Estrutura do padrão *Singleton*.

Portanto, o uso deste padrão nos permite ter controle de acesso à instância única graças ao encapsulamento por agente da própria classe, criar e configurar facilmente uma subclasse a partir da classe *Singleton*, criar mais de uma instância mudando somente a operação de instanciação.

Para que esse padrão possa ser implementado é preciso assegurar que a instância seja única através da implementação de um método privado responsável pela sua criação. Esse método tem acesso a um atributo privado que armazena a instância dessa classe e assegura que ela tenha sido criada e inicializada.

3.2- *ABSTRACT FACTORY*

O padrão *Abstract Factory* tem como objetivo “fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.” Ou seja, cada produto pode ser implementado de diferentes formas, podendo ter aparências e comportamentos distintos de acordo com a sua interface.

Através de classes abstratas que definem a interface de criação de cada objeto, as classes clientes obtém instâncias dos objetos, porém sem saber que classes concretas elas estão usando.

Caso esse padrão não seja usado, cada classe cliente deveria instanciar a classe específica do produto, tornando dessa forma qualquer alteração posterior a ele mais difícil, pois seria preciso alterar cada cliente que instancie esse produto. A fim de resolver este problema, esse padrão propõe que seja criada uma classe abstrata, chamada *Factory,* responsável por criar a instância do produto especificado pelo cliente. A estrutura desse padrão é ilustrada na Figura 12.

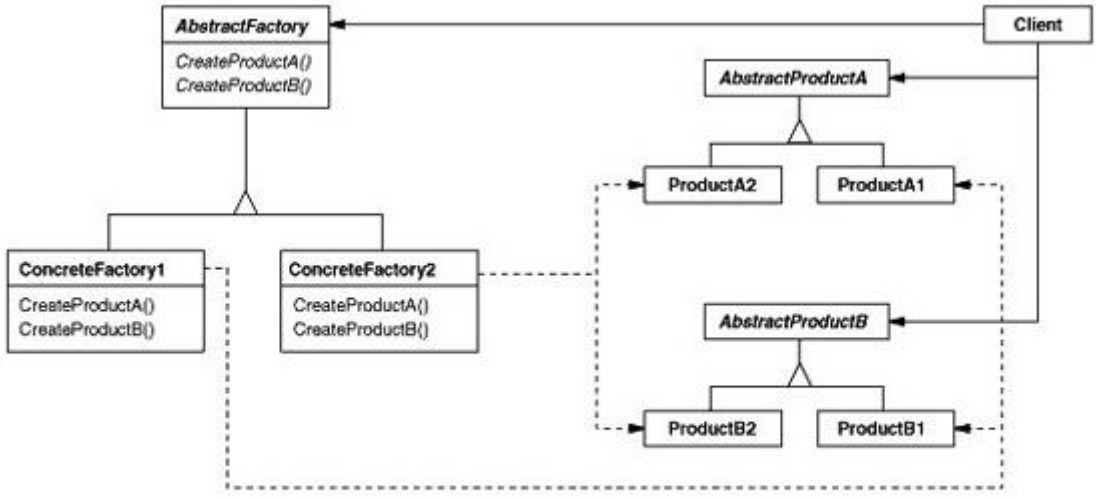

Figura 12: Estrutura do padrão *Abstract Factory*.

Pela sua estrutura é possível que as classes concretas obedecem à sua interface padrão e que a classe cliente utiliza a *Factory* para instanciá-las. Dessa forma, caso sea criada uma nova família de objetos, por

exemplo, basta criar suas classes concretas (produtos) e uma *Factory* responsável por suas instanciações, não afetando os clientes que utilizam os produtos.

Portanto, o uso desse padrão nos permite isolar as classes concretas, encapsulando a responsabilidade e o processo de criação dos produtos. Além disso, é possível realizar alterações nas famílias de produtos mais facilmente já que a classe concreta da *Factory* aparece somente uma vez na aplicação.

3.3- *FACTORY METHOD*

O padrão *Factory Method* tem como objetivo “definir uma interface para criação de um objeto, porém deixar que as subclasses decidam qual classe instanciar.” Ou seja, a instanciação é a atribuição das classes.

Através de classes abstratas, são declarados métodos abstratos de criação de objetos. As subclasses subscrevem esses métodos, definindo a implementação padrão e retornando uma instância do objeto. Esses métodos abstratos são chamados de *Factory Method*, pois eles são responsáveis por “fabricar” um objeto.

Uma grande desvantagem desse padrão é que as classes clientes precisam criar classes da classe que contém o *Factory Method* somente para criar um objeto em particular, fazendo com que o cliente lide com outro ponto de evolução. A estrutura desse padrão é ilustrada na Figura 13.

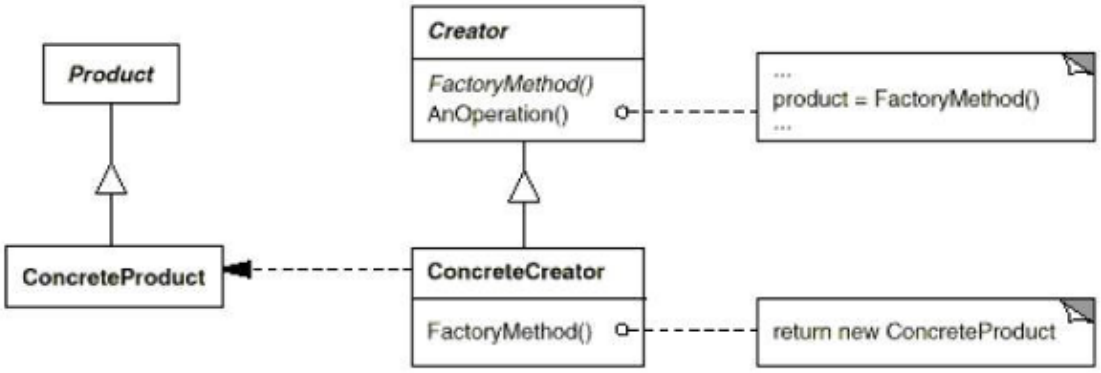

Figura 13 : Estrutura do padrão *Factory Method*.

Pela sua estrutura é possível perceber que a classe *ConcreteCreator* sobrescreve o *Factory Method* de *Creator*, retornando uma instância de *ConcreteProduct*.

Portanto, o uso desse padrão elimina a necessidade de vincular classes específicas ao código, fazendo com que ela lide apenas com a interface do produto.

3.4- *PROXY*

O padrão *Proxy* tem como objetivo “fornecer um substituto ou espaço reservado para outro objeto afim de controlar seu acesso.” Dessa forma, é possível realizarmos a criação e inicialização do objeto somente quando realmente precisamos dele.

Em alguns casos a criação de um objeto pode ser custosa para o sistema e esse objeto pode não ser necessário em um dado momento. A solução para esse problema é a criação de um novo projeto, chamado *proxy*, que é uma imagem do objeto original. Ele é responsável para instanciar o objeto original assim que o cliente o solicitar. A estrutura desse padrão é ilustrada na Figura 14 e um diagrama dessa estrutura é ilustrado na Figura 15.

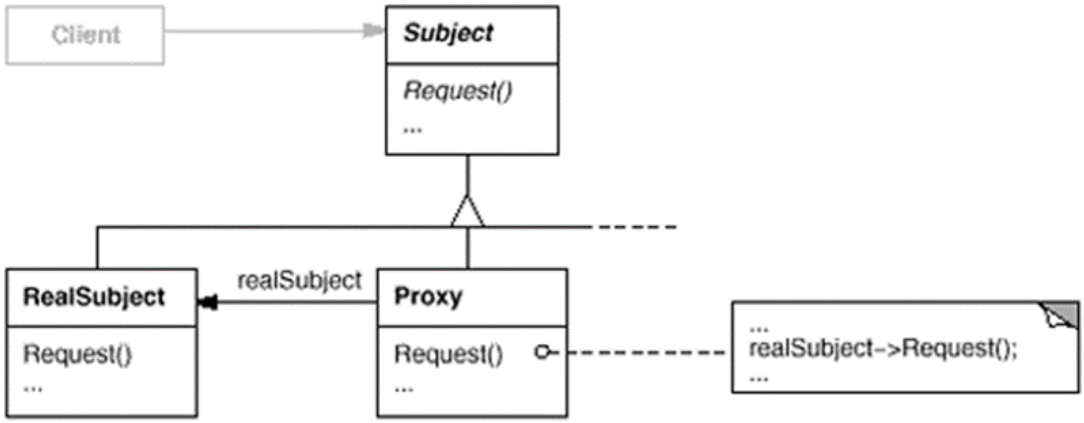

Figura 14 : Estrutura do padrão *Proxy*.

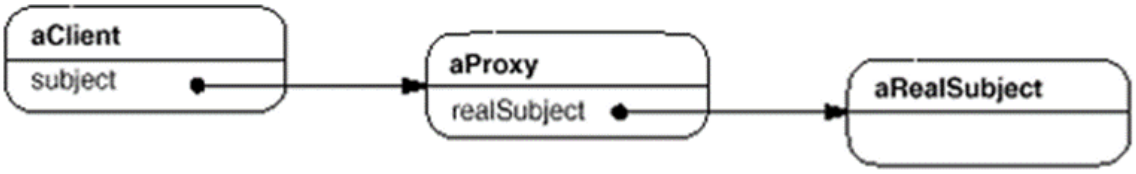

Figura 15 : Diagrama da estrutura *proxy* em tempo de execução.

Com consequência, o uso desse padrão realiza otimizações no sistema com a criação de objeto de acordo com a sua demanda e também omite o fato de um objeto residir em um espaço de endereço diferente.

3.5- *FACADE*

O padrão *Facade* tem como objetivo “fornecer uma interface unificada para um conjunto de interfaces em um subsistema.” Dessa forma, ele define uma interface de alto nível, facilitando sua utilização pelos clientes.

A divisão de um sistema em subsistemas ajuda a reduzir a sua complexidade e o uso desse padrão minimiza as dependências entre eles, fornecendo aos clientes um único ponto de acesso ao sistema através de uma interface simplificada denominada *Facade*.

A classe *Facade*, portanto, fornece apenas as funcionalidades necessárias ao cliente, sendo essas funcionalidades executadas pelas classes do subsistema. A estrutura desse padrão é ilustrada na Figura 16.

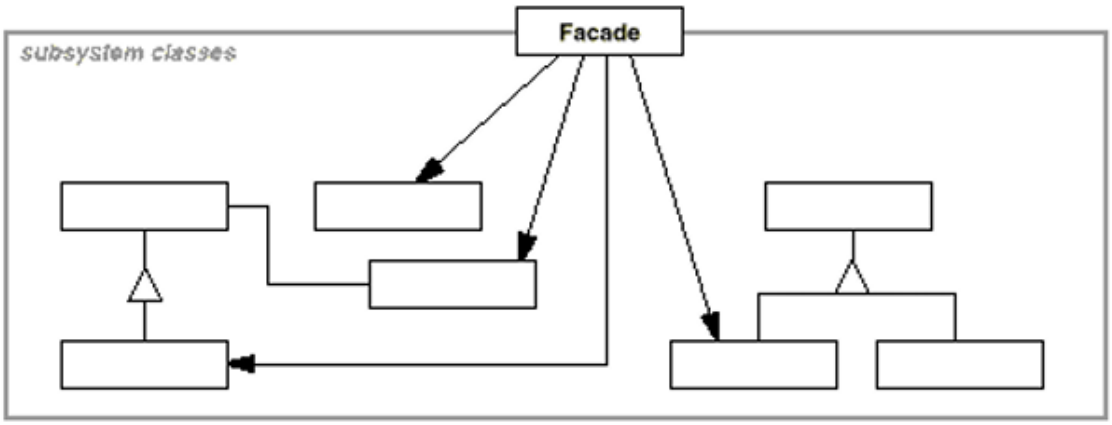

Figura 16 : Estrutura do padrão *Facade*.

Como consequência, o uso desse padrão permite isolar os clientes dos componentes de um subsistema e também alterar os componentes de um subsistema sem afetar o cliente, pois promove um acoplamento fraco entre eles.

4- ARQUITETURA DE SOFTWARE

Para o desenvolvimento deste *software* eu escolhi um sistema simples onde registro a data de aniversário do cliente e no dia certo o meu sistema dispara um e-mail para os funcionários com uma mensagem padrão, para que a empresa envie uma mensagem de felicitação aos clientes aniversariantes.

Para dar início, eu escolho o padrão de projeto *Facade*, pois ele é usado para fornecer uma interface simplificada para um subsistema complexo. Ele oculta a complexidade do sistema e fornece uma interface unificada para o cliente interagir com o sistema.

A classe “Cliente” pode ser vista como uma fachada para a funcionalidade de adicionar um cliente ao banco de dados e enviar um e-mail de felicitação, explicando melhor, a classe “Cliente” fornece uma interface simplificada para o cliente (usuário) interagir com essas funcionalidades complexas.

Adentrando aos Princípios do *Solid*, eu escolho o princípio da Responsabilidade única, pois declaro que a classe “Cliente” deve ser especializada em um único assunto e possuir uma responsabilidade dentro do *software*, ou seja, a classe deve ter uma única tarefa ou ação para executar.

Na classe “Cliente”, ela é responsável para armazenar informações e dentro dela contém os métodos.

Considerando o Princípio da Responsabilidade única um dos princípios mais importantes, ele acaba sendo a base para outros princípios e padrões porque aborda temas como acoplamento e coesão, características que todo código orientado a objetos deveria ter.

A Figura 17 representa o diagrama de classes ilustrando tudo o que foi dito acima sobre a classe “Cliente”.

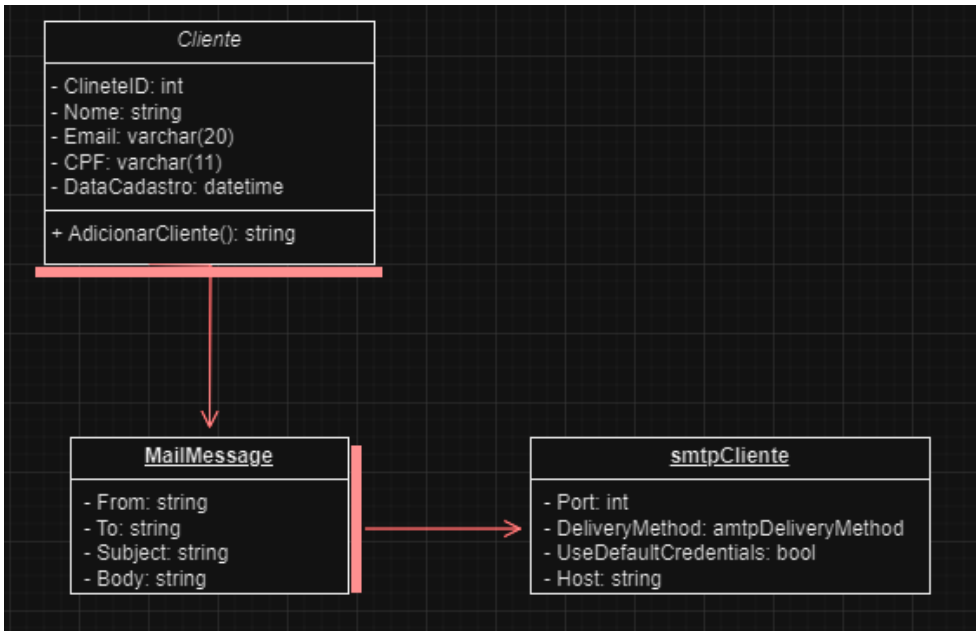

Figura 17: Diagrama de classe

9- CONCLUSÃO

Este trabalho foi muito enriquecedor para conhecermos mais o assunto dos Princípios de SOLID, Padrão de Projeto e Arquitetura de *Software*, com a ajuda de toda a ilustração colocada no trabalho ajudou e ficou de fácil entendimento para todos.

Foi um trabalho fácil de ser concluído, pois achei conteúdos fantásticos que vieram para agregar, encontrados no *Google* Acadêmico, também obtive ajuda do material do professor Thallys Braz, que veio para auxiliar e tirar algumas dúvidas.

Concluo que o trabalho foi muito bem feito, de muito conhecimento e informações, que pra mim era novidade. E com o roteiro do professor ajudou bastante para a organização e conclusão deste trabalho.

.

10- REFERÊNCIA

DE, PRINCÍPIOS SOLID E. BOAS PRÁTICAS, and ÉRICO PADILHA JUNIOR. "CENTRO UNIVERSITÁRIO FACVEST CURSO DE CIÊNCIA DA COMPUTAÇÃO TRABALHO DE CONCLUSÃO DE CURSO."

Medeiros, Ryan Salles. "Um estudo de padrões de projeto e arquiteturas no desenvolvimento de softwares baseado em orientação a objetos."