Uma breve introdução ao S.O.L.I.D.

Os princípios **SOLID** para programação e design orientados a objeto são de autoria de *Robert C. Martin* (*mais conhecido como Uncle Bob*) e datam do início de 2000. A palavra **SOLID** é um acróstico onde cada letra significa a sigla de um princípio, são eles: **SRP, OCP, LSP, ISP e DIP**.

Os princípios **SOLID** devem ser aplicados no desenvolvimento de software de forma que o software produzido tenha as seguintes características:

- **Seja fácil de manter, adaptar e se ajustar às constantes mudanças exigidas pelos clientes;**
- **Seja fácil de entender e testar;**
- **Seja construído de forma a estar preparado para ser facilmente alterado com o menor esforço possível;**
- **Seja possível de ser reaproveitado;**
- **Exista em produção o maior tempo possível;**
- **Que atenda realmente as necessidades dos clientes para o qual foi criado;**

SRP	The Single Responsibility Principle Principio da Responsabilidade Única	Uma classe deve ter um, e somente um, motivo para mudar. A class should have one, and only one, reason to change.
OCP	The Open Closed Principle Princípio Aberto-Fechado	Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo. You should be able to extend a classes behavior, without modifying it.
LSP	The Liskov Substitution Principle Princípio da Substituição de Liskov	As classes derivadas devem poder substituir suas classes bases. Derived classes must be substitutable for their base classes.
ISP	The Interface Segregation Principle Princípio da Segregação da Interface	Muitas interfaces específicas são melhores do que uma interface geral Make fine grained interfaces that are client specific.
DIP	The Dependency Inversion Principle Princípio da inversão da dependência	Dependa de uma abstração e não de uma implementação. Depend on abstractions, not on concretions.

Princípio de Aberto / Fechado

*"Componentes de software (classes, módulos, funções, etc.) devem ser **abertas** para extensão mas **fechadas** para modificação."*

Ou seja: **quando eu precisar estender o comportamento de um código, eu crio código novo ao invés de alterar o código existente**.

E como isso é possível? Como adicionar comportamentos novos sem alterar o código existente? A resposta é: Abstração!

Princípios do padrão OCP:

Extensibilidade

É uma das chaves da orientação a objetos, quando um novo comportamento ou funcionalidade precisar ser adicionado é esperado que as existentes sejam estendidas e e não alteradas, assim o código original permanece intacto e confiável enquanto as novas são implementadas através de extensibilidade.
Criar código extensível é uma responsabilidade do desenvolvedor maduro, utilizar design duradouro para um software de boa qualidade e manutenibilidade.

Abstração

Quando aprendemos sobre orientação a objetos com certeza ouvimos sobre abstração, é ela que permite que este princípio funcione. Se um software possui abstrações bem definidas logo ele estará aberto para extensão.

Exemplificando:

Imagine que precisamos implementar um serviço de calculo de pagamento baseado em 3 tipos de modalidades:

A vista: Valor a pagar consiste do valor exato da fatura com opcional de desconto.
ParceladoSemJuros: Valor a pagar consiste do valor da fatura dividido pelo numero de parcelas.
ParceladoComJuros: Valor a pagar consiste do valor da fatura dividido pelo numero de parcelas com taxa de juros aplicada.

Solução 01: Ferindo o padrão OCP

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OCP.ModoErrado
{
    public enum TipoPagamento { aVista, ParceladoSemJuros, ParceladoComJuros }

    public abstract class ControlePagamento
    {
        public TipoPagamento TipoPgto { get; set; }

        public double? CalcularPagamento(double valor, double? desconto,
                                         int? parcelas, double? juros)
        {
            switch(TipoPgto)
            {
                case TipoPagamento.aVista:
                    return valor - desconto;

                case TipoPagamento.ParceladoSemJuros:
                    return valor / parcelas;

                case TipoPagamento.ParceladoComJuros:
                    return (valor / parcelas) * juros;

                default:
                    return null;
            }
        }
    }
}
```

Problemas:

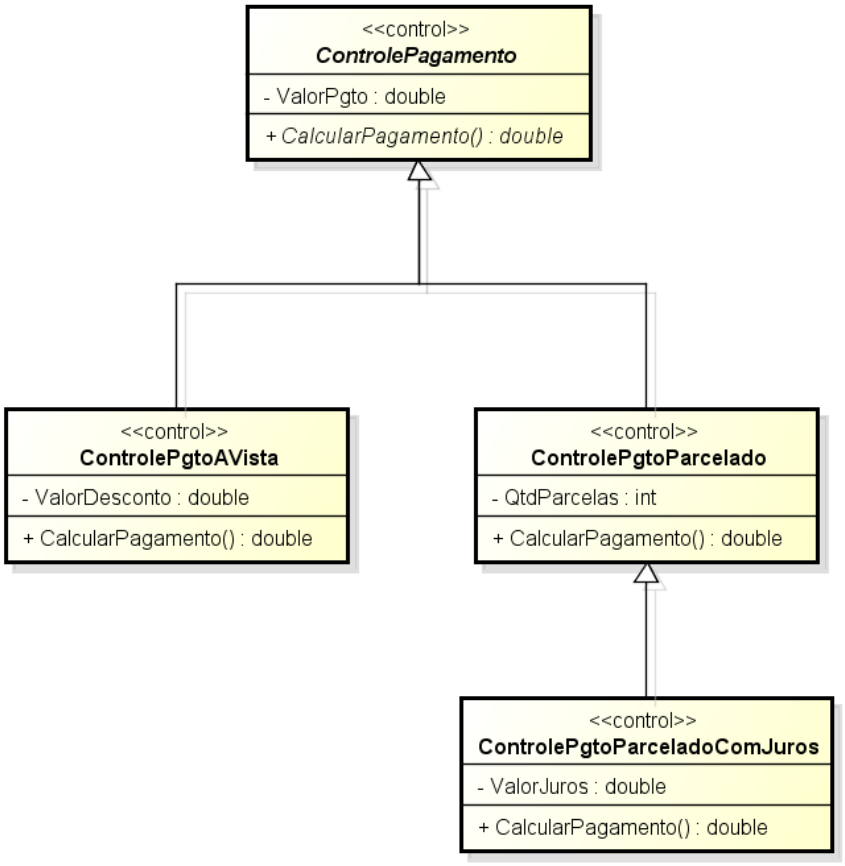
Observe que no código acima, temos muitos problemas de modelagem.
O principal é que, caso seja feita uma modificação no cálculo do pagamento (como incluir uma nova modalidade ou regra de negocio, seja a inclusão de alguma fórmula adicional ou qualquer outro fator) pode causar comportamentos inesperados para quem usa esta classe, até mesmo um bug.
Além de o próprio método CalcularPagamento ter sua escalabilidade e manutenção comprometida conforme novos requisitos forem incrementados.

Qual é o problema de um IF a mais?
Se modificarmos a classe colocando mais um IF ouo CASE de teste, além de ter que substituirmos esta classe na publicação da nova versão, corremos o risco de introduzir alguns bugs em uma classe que já estava funcionando.

Mas por que na prática eu deveria fechar para modificações uma classe?
A razão é simples: porque assim eu posso desenvolver meu software como se fosse em camadas. Estando uma camada muito bem escrita e bem definida, eu tenho certeza de que todas as classes derivadas também funcionarão bem.

As classes derivadas na prática poderiam apenas usar os métodos fechados e acrescentar novos comportamentos ao sistema conforme novas necessidades fossem surgindo.

Solução 02: Refatorando para o padrão aberto e fechado



Note que possuímos agora uma abstração bem definida, onde todas as extensões implementam suas próprias regras de negócio sem necessidade de modificar uma funcionalidade devido mudança ou inclusão de outra.

Codificando:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OCP.ModoCorreto
{
    public abstract class ControlePagamento
    {
        public double Valor { get; set; }

        public abstract double CalcularPagamento();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OCP.ModoCorreto
{
    public class ControlePgtoAVista : ControlePagamento
    {
        public double Desconto { get; set; }

        public override double CalcularPagamento()
        {
            return Valor - Desconto;
        }
    }
}
```

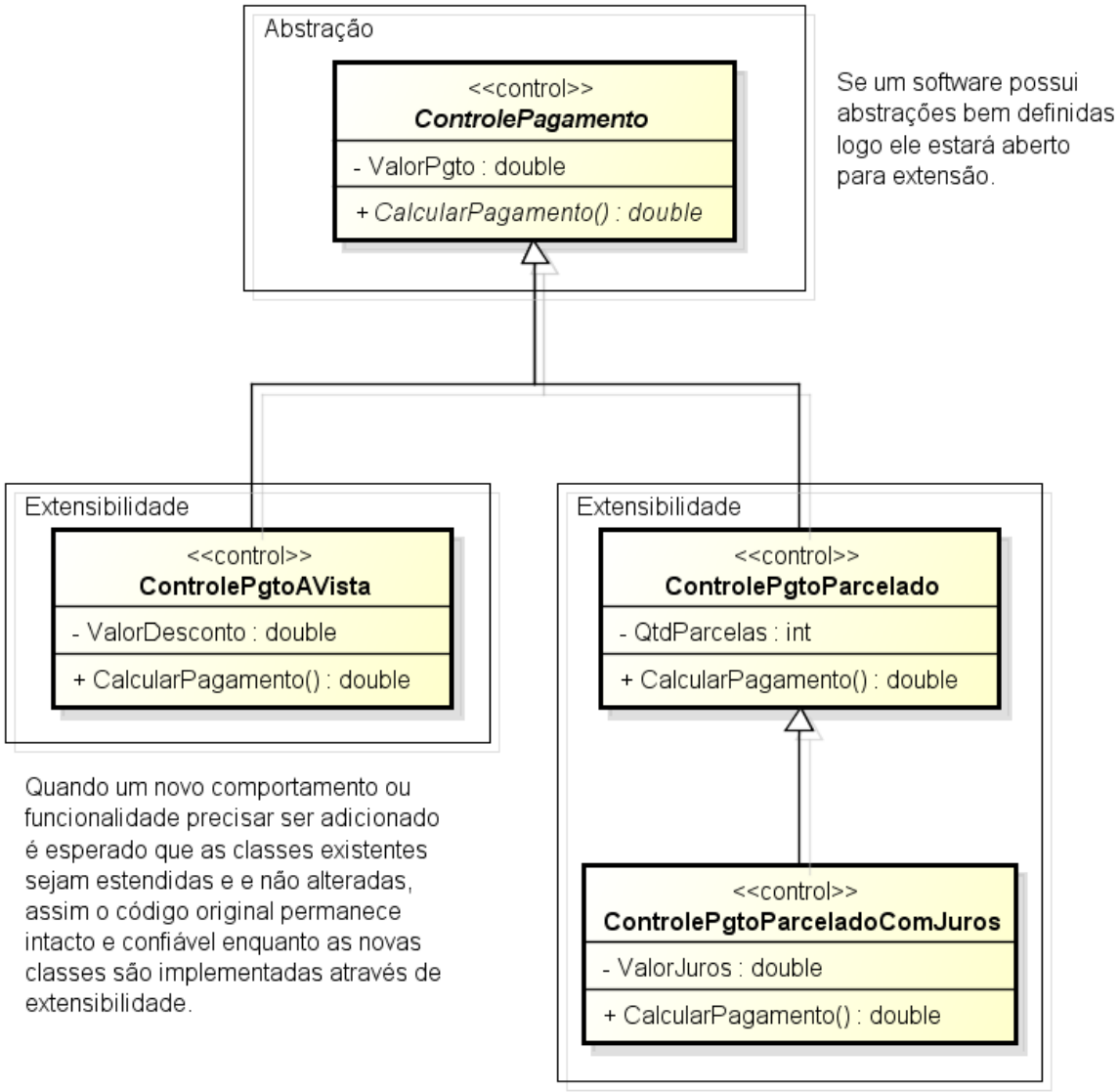
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OCP.ModoCorreto
{
    public class ControlePgtoParcelado : ControlePagamento
    {
        public int Parcelas { get; set; }

        public override double CalcularPagamento()
        {
            return Valor / Parcelas;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OCP.ModoCorreto
{
    public class ControlePgtoParceladoComJuros : ControlePgtoParcelado
    {
        public double Juros { get; set; }

        public override double CalcularPagamento()
        {
            return base.CalcularPagamento() * Juros;
        }
    }
}
```

"Quando eu precisar estender o comportamento
de um código, eu crio código novo
ao invés de alterar o código existente."

O Princípio do Aberto/Fechado nos atenta para a aplicação de abstrações e polimorfismo, de forma consciente, garantindo que tenhamos um software mais flexível e, portanto, mais fácil de ser mantido.

Entendendo o padrão OCP:



Este princípio nos atenta para um melhor design, tornando o software mais extensível e facilitando sua evolução sem afetar a qualidade do que já está desenvolvido.

Fontes de estudo:

- http://code.tutsplus.com/tutorials/solid-part-2-the-openclosed-principle--net-36600
- https://robsoncastilho.com.br/2013/02/23/principios-solid-principio-do-abertofechado-ocp/
- http://blog.thedigitalgroup.com/rakeshg/2015/05/06/solid-architecture-principle-using-c-with-simple-c-example/