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.

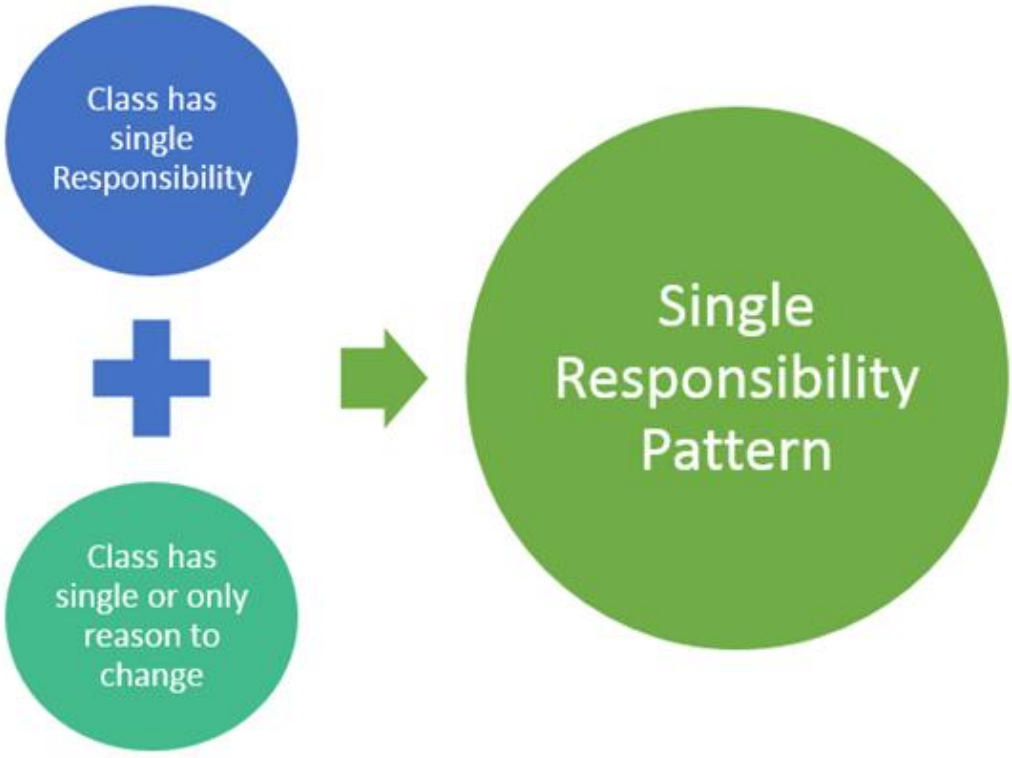
Princípio da Responsabilidade Única

"Uma classe só deveria ter um único motivo para mudar."

Uma classe deve fazer apenas uma coisa, deve fazê-la bem e deve fazer somente ela. Se uma classe tem mais de um motivo para ser alterada, ela não segue este princípio. Se ao se referir a uma determinada classe, você diz, por exemplo: "minha classe tem as informações do cliente e salva o mesmo no banco de dados" perceba que o **"e"** na frase, indica mais de uma responsabilidade, ferindo assim o SRP(single responsability principle).

Aplicar o princípio da responsabilidade única é importante para uma arquitetura madura e sustentável. Quando começamos a dar valor a princípios como este, vemos que estamos amadurecendo e fazendo melhor o que gostamos de fazer: software.

"Cada responsabilidade deve ser codificada em uma classe separada, pois cada responsabilidade é eixo para mudança."



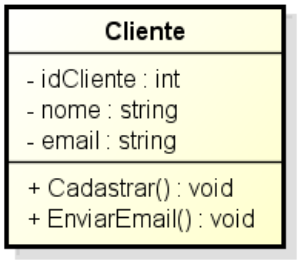
Determinar a única responsabilidade de uma classe ou módulo é muito mais complexo que, simplesmente, verificar em uma lista pré-determianda. Por exemplo, uma dica para encontrarmos as razões para mudanças é verificar a audiência da nossa classe. Os usuários da aplicação ou sistema que desenvolvemos é que requisitarão mudanças para ela. Aqueles que usam que pedirão mudanças. Eis alguns módulos e suas possíveis audiências.

- **Módulo de Persistência** - Pode incluir os DBAs e arquitetos de software;
- **Módulo de Relatório** - Pode incluir os contadores, secretários e administradores;
- **Módulo de Computação de Pagamento para um Sistema de Pagamento** - Pode incluir os advogados, administradores e contadores;
- **Módulo de Busca de Livros para um Sistema de Administração de Biblioteca** - Pode incluir os bibliotecários e/ou os próprios clientes.

Exemplificando:

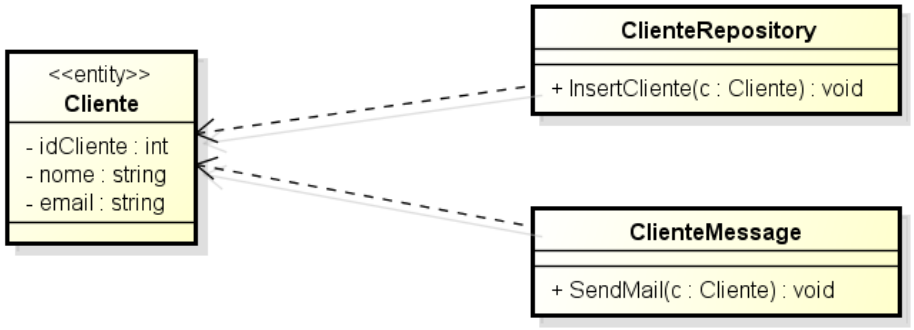
Imagine que precisamos em um software executar uma rotina de gravação em banco de dados de um Cliente e logo em seguida enviar um email de confirmalção para o mesmo:

Solução 01: Ferindo o padrão SRP



O problema desta solução é justamente a não separação de responsabilidades, pois a classe Cliente além de especificar os atributos da própria entidade também define as rotinas de gravação na base de dados e envio de email, tudo na mesma classe.

Solução 02: Refatorando o modelo anterior e separando as responsabilidades para cada classe.



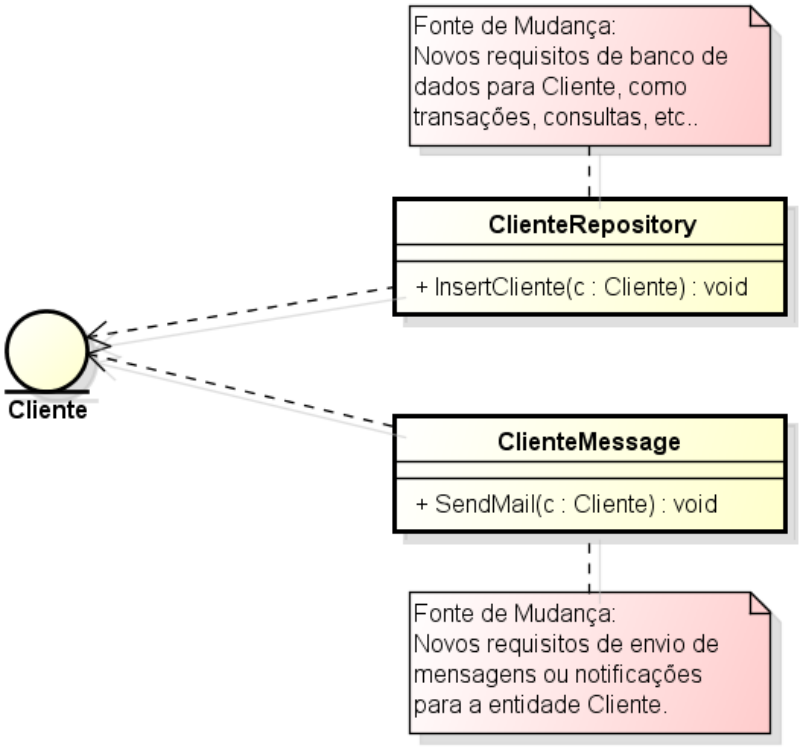
Neste segundo modelo, podemos notar a separação das responsabilidades, pois isolamos a **entidade Cliente** como um **POCO** e definimos 2 classes que utilizam os dados da entidade para realizar as operações:

ClienteRepository: Tem a responsabilidade de fazer operações na base de dados voltados para Cliente, como Insert, Delete, Update, Find, etc...

ClienteMessage: Tem a responsabilidade de implementar o serviço de envio de email para o cliente.

O ator de uma responsabilidade
*é a única **fonte de mudança***
para aquela responsabilidade.
(Robert C. Martin)

Seguindo esse raciocínio, os atores tornam-se a fonte da mudança para a família de funções que os servem. De acordo com que suas necessidades mudam, aquela família de funções também deve mudar para adaptar-se às suas necessidades.



Codificando:

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

namespace SRP.Entities
{
    public class Cliente
    {
        public int IdCliente { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
    }
}
```

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

namespace SRP.Repositories
{
    public class ClienteRepository
    {
        public void InsertCliente(Cliente c)
        {
            //TO DO..
        }
    }
}
```

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

namespace SRP.Messages
{
    public class ClienteMessage
    {
        public void SendMail(Cliente c)
        {
            //TO DO..
        }
    }
}
```

Quando pensamos sobre o sistema que precisamos codificar, podemos analisar diversos aspectos diferentes. Por exemplo, vários requisitos que afetam uma mesma classe podem ser um foco de mudança. Esses focos de mudanças são dicas para a responsabilidade única. Há uma alta probabilidade que grupos de requisitos que estão afetando o mesmo grupo de funções serão responsáveis por outras mudanças ou que precisarão ser separados em um grupo próprio.

O valor primário de um aplicativo é a facilidade de alteração, o secundário é a funcionalidade, no sentido de satisfazer tantos requerimentos quanto possível, atendendo as necessidades dos usuários. Entretanto, para alcançar um valor secundário bastante positivo, o valor primário é obrigatório. Para mantermos nosso valor primário alto, devemos ter um projeto que seja fácil de alterar, estender e adicionar novas funcionalidades, além de garantir que a SRP seja respeitada.

Então, quando formos projetar nossos aplicativos, deveríamos:

1. Buscar e definir atores;
2. Identificar a responsabilidade própria a esses atores;
3. Agrupar nossas funções em classes, para que cada uma tenha uma única responsabilidade.

Fontes de Estudo:

- http://code.tutsplus.com/pt/tutorials/solid-part-1-the-single-responsibility-principle--net-36074
- http://blog.thedigitalgroup.com/rakeshg/2015/05/06/solid-architecture-principle-using-c-with-simple-c-example/
- http://www.devmedia.com.br/arquitetura-o-principio-da-responsabilidade-unica/18700