

Gabriel Salomão Morais

O sistema CFC foi desenvolvido na linguagem C# com .NET Framework na versão 4.5 e a ORM Entity Framework, utilizando como banco de dados SQL Server.

O sistema é basicamente de gestão financeira onde o negociante será capaz de cadastrar algum cliente e após isso poderá verificar toda sua situação (se está ativo ou devendo) além da emissão de boletos.

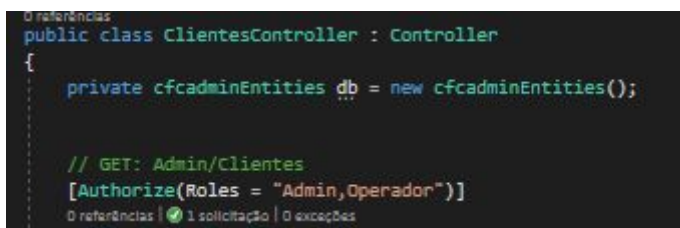
Existem três áreas no sistema

- **Admin** onde oferece uma visão geral e controle sobre o sistema podendo alterar informações do cliente, adição de itens e sinistros.
- **Cliente** onde fica as informações de um cliente específico.
- **Mkt** onde é possível alterar informações do layout da página principal.

Apesar de não ser tão grande existe uma série de melhorias que possam ser feitas para que o sistema continue crescendo com qualidade.

Implementar conceitos do S.O.L.I.D. e Design Patterns pode ser uma boa escolha para facilitar a manutenção do sistema e adição de futuras funcionalidades.

Alguns princípios básico do S.O.L.I.D. pode ser aplicado ao código como no exemplo abaixo.

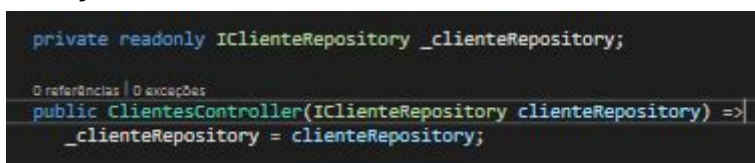


```
0 referências
public class ClientesController : Controller
{
    private cfcadminEntities db = new cfcadminEntities();

    // GET: Admin/Clientes
    [Authorize(Roles = "Admin,Operador")]
    0 referências | 1 solicitação | 0 exceções
```

A primeira opção seria implementar o **S** do Princípio Da Responsabilidade única, todas as Controllers estão dando new no Banco de dados o que seria função de um Design Patterns chamado Repositório, uma classe deve ser o menos acoplado possível, caso haja alguma alteração no contexto afetaria diretamente todas ligadas a ela. A Controller só tem a responsabilidade de responder á solicitações feitas em relação a um site.

Correção



```
private readonly IClienteRepository _clienteRepository;

0 referências | 0 exceções
public ClientesController(IClienteRepository clienteRepository) =>
    _clienteRepository = clienteRepository;
```

Na correção acima utiliza-se outros dois princípios do S.O.L.I.D, o **I** do Princípio Da Segregação da Interface e o **D** do Princípio da Injeção de Dependências, tudo isso é feita para que a controller evitar o acoplamento, todos os métodos e validação devem ser

colocado dentro de interfaces como no exemplo abaixo.

```
0 referências | 0 soluções | 0 exceções
public ActionResult Create(Clientes clientes, string cidade, string estado)
{
    if (cidade == null)
    {
        TempData["erro"] = "Utilize o CEP para Localizar a Cidade!";
        return View(clientes);
    }

    if (estado == null)
    {
        TempData["erro"] = "Utilize o CEP para Localizar o Estado!";
        return View(clientes);
    }
}
```

Aplicando a Interface com a injeção de dependências o código da controller reduziria para

```
if (_clienteRepository.Validacao(cidade, estado))
    return View(clientes);
```

O método do repositório de Cliente

```
0 referências
public class ClienteRepository : RepositoryBase<Clientes>, IClienteRepository
{
    2 referências | 0 exceções
    public bool Validacao(string cidade, string estado)
    {
        if (cidade == null)
            return false;

        if (estado == null)
            return false;

        return true;
    }
}
```

Abstraindo as classes torna o código mais limpo e de fácil manutenção.