Israel Aece

Software Developer

PUBLICADO POR

ISRAEL AECE
POSTADO NO

14/06/2016
PUBLICADO EM

ARQUITETURA, CSD
COMENTÁRIOS

1 COMENTÁRIO

Eventos de Domínio – Disparo e Consumo

i
1 Vote

No artigo anterior (https://israelaece.com/2016/06/14/eventos-de-dominio-geracao/) falamos sobre a criação e utilização de eventos de domínio. O artigo abordou até o momento do disparo do evento propriamente dito, através da classe *DomainEvents*, só que sem mostrar detalhes de sua implementação. Existem diversas formas de se implementar o método de disparo do evento, mas antes de falarmos sobre estas técnicas, precisamos abordar como se constrói os consumidores dos eventos.

Para especificarmos os tratadores, vamos criar uma *interface* que descreverá apenas um método: *Handler*. Só que trata-se de uma *interface* genérica, onde o tipo T deve ser alguma classe que obrigatoriamente implemente a *interface IDomainEvent*, e que para o exemplo que estamos utilizando e evoluindo é a classe *SaldoDaContaAlterado*.

```
public interface IHandler<T>
    where T : IDomainEvent
{
    void Handle(T @event);
}
```

Com a *interface* criada, temos que implementar a mesma em classes que serão consideradas os tratadores dos eventos, substituindo o tipo T por algum evento que nosso domínio define e estamos interessados em sermos notificados quando ele acontecer. Dentro da implementação do método *Handle* ficamos livres para executar tudo o que for necessário para aquele contexto, e que no nosso caso, é colocar "uma lupa" sobre o cliente que está com saldo negativo. Note que como parâmetro do método *Handle* temos (ou deveríamos ter) todas as informações necessárias a respeito do que ocorreu.

```
public class MonitorDeClientes : IHandler<SaldoDaContaAlterado>
{
    public void Handle(SaldoDaContaAlterado @event)
    {
        if (@event.SaldoAtual < @event.SaldoAnterior)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(
                "Monitorando o Cliente {0}. Saldo: {1:N2}",
                @event.NomeDoCliente,
                @event.SaldoAnterior);

            Console.ResetColor();
        }
    }
}
```

Uma vez que a classe concreta está criada e implementada, precisamos acopla-la a execução para que ela seja executada. Agora fazemos o uso da classe *DomainEvents* para acomodar a relação dos eventos de domínio. Aqui temos duas formas de proceder, sendo uma lista de tratadores estáticos ou de tratadores dinâmicos. Os tratadores estáticos permitem à aplicação já identificar todos os tratadores existentes, em outras palavras, podemos utilizar *Reflection* para encontrar todas as classes que implementam a *interface IHandler<T>* e adiciona-las a coleção de tratadores da aplicação, e também via *Reflection*, instanciarmos essas classes que representam os eventos toda vez em que ele for disparado pela domínio.

```
public static class DomainEvents
{
    private static List<Type> handlers = new List<Type>();

    static DomainEvents()
    {
        handlers =
            (
                from t in Assembly.GetExecutingAssembly().GetTypes()
                from i in t.GetInterfaces()
                where
                    i.IsGenericType &&
                    i.GetGenericTypeDefinition() == typeof(IHandler<>)
                select t
            ).ToList();
    }

    public static void Raise<T>(T @event) where T : IDomainEvent
    {
        handlers.ForEach(h =>
        {
            if (typeof(IHandler<T>).IsAssignableFrom(h))
                ((IHandler<T>)Activator.CreateInstance(h)).Handle(@event);
        });
    }
}
```

O uso externo da classe *ContaCorrente* não muda em nada, ou seja, continuamos interagindo com os métodos públicos que ela expõe. Como a varredura em busca por classes que implementam a *interface IHandler<T>* está no construtor estático da classe *DomainEvents*, tão logo quando a aplicação for inicializada os tipos serão identificados e adicionado, e quando o método *Raise* for invocado quando um lançamento de débito ou crédito ocorrer, o nome do cliente e seu saldo serão apresentados na tela em cor vermelha.

```
var cc = new ContaCorrente("Israel Aece");
cc.Lancar(new ContaCorrente.Lancamento("Pagto de Energia", -1000M));
```

A outra opção que temos é a relação dinâmica de tratadores, onde também podemos utilizar *Reflection* para descobrir os tratadores que implementam a *interface IHandler<T>*, porém há a possibilidade de dinamicamente adicionar novos tratadores em tempo de execução de acordo com a necessidade através do método *Register*. O método *Raise* agora já não instancia dinamicamente o tratador, ou seja, isso é responsabilidade do código que o consome, dando a possibilidade de fazer uso da instância antes e depois se desejar, o que pode ser útil durante os testes para saber se o evento foi o não disparado.

```
public class MonitorDeClientes : IHandler<SaldoDaContaAlterado>
{
    public readonly List<string> ClientesMonitorados = new List<string>();

    public void Handle(SaldoDaContaAlterado @event)
    {
        if (@event.SaldoAtual < @event.SaldoAnterior)
            this.ClientesMonitorados.Add(@event.NomeDoCliente);
    }
}
```

Neste modelo, para exemplificar, ao invés de escrever na tela o cliente monitorado, o adicionamos na coleção de clientes, que nada mais é que um campo da classe. E a classe *DomainEvents* também mudará a sua implementação para possibilitar o vínculo dinâmico de eventos, onde temos um dicionário que para cada tipo de evento uma coleção de *delegates* é criada.

```
1  public static class DomainEvents
2  {
3      private static Dictionary<Type, List<Delegate>> handlers =
4          new Dictionary<Type, List<Delegate>>();
5
6      static DomainEvents()
7      {
8          handlers =
9              (
10                 from t in Assembly.GetExecutingAssembly().GetTypes()
11                 where
12                     !t.IsInterface &&
13                     typeof(IDomainEvent).IsAssignableFrom(t)
14                 select t
15             ).ToDictionary(t => t, t => new List<Delegate>());
16     }
17
18     public static void Register<T>(Action<T> handler) where T : IDomainEven
19     {
20         handlers[typeof(T)].Add(handler);
21     }
22
23     public static void Raise<T>(T @event) where T : IDomainEvent
24     {
25         handlers[typeof(T)].ForEach(h => ((Action<T>)h)(@event));
26     }
27 }
```

Por fim, o código que consome também sofrerá uma alteração para exibir o uso monitor antes e depois do evento que foi disparado.

```
1  var monitor = new MonitorDeClientes();
2  DomainEvents.Register<SaldoDaContaAlterado>(monitor.Handle);
3
4  var cc = new ContaCorrente("Israel Aece");
5  cc.Lancar(new ContaCorrente.Lancamento("Pagto de Energia", -1000));
6
7  Console.WriteLine("Qtde: {0}", monitor.ClientesMonitorados.Count);
```

Em ambas as técnicas é possível ter diversos tratadores para um mesmo evento gerado. Isso é comum e muito mais elegante do que em um simples tratador realizar mais tarefas do que ele deveria fazer. Se ele é responsável por monitorar, não deveria ser responsável por notificar o gerente que a conta de seu cliente ficou negativa. Nos tratadores também vale o princípio de responsabilidade única para garantir uma fácil manutenção e legibilidade.

Para finalizar, essas técnicas funcionam bem, mas existem alguns problemas funcionais que podem tornar o sistema propício a falhas. Mas isso será assunto do próximo artigo da série.

- Código (https://israelaece.com/tag/codigo/)
- Domínio (https://israelaece.com/tag/dominio/)

Um comentário sobre “Eventos de Domínio – Disparo e Consumo”

1. Pingback: Eventos de Domínio – Outra Opção de Disparo | Israel Aece