Israel Aece

Software Developer

PUBLICADO POR

ISRAEL AECE
POSTADO NO

15/06/2016
PUBLICADO EM

ARQUITETURA, CSD
COMENTÁRIOS

DEIXE UM COMENTÁRIO

Eventos de Domínio – Outra Opção de Disparo

i
Rate This

Nos artigos anteriores falamos sobre a geração (https://israelaece.com/2016/06/14/eventos-de-dominio-geracao/) e consumo (https://israelaece.com/2016/06/14/eventos-de-dominio-disparo-e-consumo/) de eventos de domínio. Entre os assuntos abordados, discutimos os tratadores, que nada mais são que classes que são executadas reagindo ao evento que foi disparado. Ainda falando sobre os tratadores, abordamos a forma de descobrir os tratadores que fazem parte da aplicação (estática ou dinâmica) bem como a possibilidade de incluir novos tratadores em tempo de execução.

Para recapitular, temos a classe *DomainEvents*, qual utilizamos para disparar os eventos. Nos exemplos anteriores, esta classe estava sendo utilizada no interior das entidades, que quando era detectado a necessidade de disparo de algum evento, recorria ao método *Raise*, informando o tipo do evento e suas respectivas informações.

```
public void Lancar(Lancamento lancamento)
{
    var saldoAnterior = this.Saldo;

    this.lancamentos.Add(lancamento);
    this.Saldo += lancamento.Valor;

    DomainEvents.Raise(
        new SaldoDaContaAlterado(this.NomeDoCliente, saldoAnterior, this.Sa
}
```

O problema desta técnica é que a entidade além de criar o evento, também está sendo responsável por disparar ele, e se algum problema acontecer depois do disparo de evento que notifica a alteração do saldo, não é fácil desfazer o que já foi realizado pelo(s) tratador(es). Existem situações em que não dá para assegurar que depois do evento disparado as informações serão corretamente persistidas sem que algum erro ocorra. Considere o exemplo de código a seguir:

```
var repositorio = new RepositorioDeContas();

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

repositorio.Atualizar(cc);
```

Conforme vimos anteriormente, o método *Lancar* dispara o evento e o tratador adiciona o cliente para monitoramento. Imagine agora que ao invocar o método *Atualizar* do repositório, algum exceção ocorra. A complexidade para ir até o monitor e desfazer a inserção do cliente seria muito custosa e de difícil implementação. Isso poderia piorar ainda mais se estivermos trabalhando entre contextos distintos, que podem estar fisicamente separados.

Felizmente temos uma alternativa para melhorar a implementação e o disparo dos eventos, combinando isso com o repositório da entidade. Ao invés das entidades gerarem e dispararem os eventos, criamos internamente uma coleção destes eventos para que ela vá armazenando todos os acontecimentos, e ao atualizar na base de dados, percorremos todos os eventos, disparando cada um deles. Para uma melhor reutilização de código, criamos uma classe base para todas as entidades, ou melhor, para os *aggregate roots*.

```
public abstract class Entidade
{
    private readonly IList<IDomainEvent> eventos =
        new List<IDomainEvent>();

    protected void AdicionarEvento(IDomainEvent evento)
    {
        this.eventos.Add(evento);
    }

    public void RemoverEventos()
    {
        this.eventos.Clear();
    }

    public IEnumerable<IDomainEvent> Eventos
    {
        get
        {
            return this.eventos;
        }
    }
}
```

Internamente esta classe armazenará a coleção de eventos, representado por instâncias de classes que implementam a *interface IDomainEvent*. A implementação do método *Lancar* tem uma suave mudança, e passa a recorrer ao método *AdicionarEvento* (que é *protected*) para adicionar o evento que indica a alteração do saldo.

```
public void Lancar(Lancamento lancamento)
{
    var saldoAnterior = this.Saldo;

    this.lancamentos.Add(lancamento);
    this.Saldo += lancamento.Valor;

    this.AdicionarEvento(
        new SaldoDaContaAlterado(this.NomeDoCliente, saldoAnterior, this.Sa
}
```

Isso por si só não funciona. Conforme falamos acima, temos que mudar o repositório para que ele identifique a existência de eventos e dos dispare. Mas aqui vale observar que ele somente deverá fazer isso depois que a atualização na base de dados (INSERT, UPDATE ou DELETE) seja realizada com sucesso. Os tratadores agora podem realizar suas atividades sem a preocupação de que aquilo poderia, em algum momento, ser desfeito.

Para manter a simplicidade, implementei o disparo dos eventos diretamente no repositório de contas, mas é possível refatorar o repositório a fim de criar uma base para todos os repositórios da aplicação, reutilizando o disparo de eventos para todas as entidades, já que o processo será o exatamente o mesmo. O método *DispararEventos* deve ser chamado sempre que a adição, atualização ou exclusão for realizada, e assim iteramos pela coleção de eventos (exposta pela classe abstrata *Entidade*) e invocamos o método estático *Dispatch* da classe *DomainEvents*. Por fim, depois dos eventos disparados, removemos os mesmos da entidade, já que se alguma coisa nova acontecer a partir dali, estes já estão concluídos.

```
public class RepositorioDeContas : IRepositorio<ContaCorrente>
{
    public void Atualizar(ContaCorrente entidade)
    {
        //Atualizar Base de Dados

        DispararEventos(entidade);
    }

    private static void DispararEventos(Entidade entidade)
    {
        foreach (var evento in entidade.Eventos)
            DomainEvents.Dispatch(evento);

        entidade.RemoverEventos();
    }
}
```

O método *Dispatch* tem funcionalidade semelhante ao *Raise*, mas soa melhor neste cenário, já que aqui ele tem a função de delegar o disparo dos eventos criados pelas entidades para os tratadores. Ao contrário do método *Raise*, que é genérico, o método *Dispatch* lida diretamente com instâncias da *interface IDomainEvent* ao invés dos eventos concretos. Por fim, ele analise se o tipo do evento que o tratador implementa é igual ao evento disparado, e o executa.

```
public static void Dispatch(IDomainEvent @event)
{
    foreach (var handler in handlers)
        if (handler.GetInterfaces()
                   .Any(h => h.IsGenericType && h.GenericTypeArguments[0] ==
            ((dynamic)Activator.CreateInstance(handler)).Handle((dynamic)@ev
}
```

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