Fluent Interface

Acadêmicos: Daniel Passos, Gabriel Conti e Maurício Farias
Disciplina: Engenharia de Software
Professor: Marcello Thiry

Definição

Interface fluente (ou em inglês “fluent interface”) é um termo criado pelos especialistas de software Martin Fowler e Eric Evans, sendo utilizado para descrever um padrão para a construção de classes que favorece a obtenção de um código menos extenso e mais legível.

Problema

- Códigos pouco legíveis.
- Distante da linguagem e forma de expressão humana.
- O raciocínio humano tende a ser organizado de forma estruturada o que diverge de abordagens orientadas a objeto (apesar de serem perfeitas pela abstração).

Solução

- Declaração de métodos de forma pouco usual mas com instanciamento extremamente legível e semelhante a linguagem humana, se bem organizado.
- A instância devolvida como resultado da execução de um método, corresponde exatamente à referência que serviu de base para o seu acionamento.
- Encadeamento.

Aplicação

- APIs com orientação a objeto que possuam alto encadeamento de métodos.
- Usa encadeamento de método para implementar o método em cascata (em linguagens que não suportam o cascateamento nativamente).

Exemplo normal

Classe

```
public class Pedido {

        List<Item> itens;
        Cliente cliente;

        void adicionarItem(Item item) {
                itens.add(item);
        }

        void setCliente(Cliente cliente) {
                this.cliente = cliente;
        }

        void fechar() {
                // ...
        }

}
```

Instanciamento

```
Pedido p = new Pedido();
p.setCliente(new Cliente("José"));
p.adicionarItem(new Item("Motocicleta", 1));
p.adicionarItem(new Item("Capacete", 2));
p.fechar();
```

O acesso a diversas operações de um mesmo objeto sucessivamente é algo comum em desenvolvimento de software, o que pode levar à obtenção de extensos blocos de código que impedem uma rápida leitura e entendimento do cenário que se está considerando.

Com Fluent Interface

Classe

```
public class Pedido {

        List<Item> itens;
        Cliente cliente;

        Pedido com(int quantidade, String nome) {
                itens.add(new Item(nome, quantidade));
                return this;
        }

        Pedido para(String nome) {
                cliente = new Cliente(nome);
                return this;
        }

        void fechar() {
                // ...
        }

}
```

Instanciamento

```
new Pedido()
        .para("José")
        .com(1, "Motocicleta")
        .com(2, "Capacete")
        .fechar();
```

```
public class Data {

	public static Date converteTextoParaData(String dataStr) {
		try {
			return new SimpleDateFormat("dd/MM/yyyy").parse(da
taStr);
		} catch (ParseException e) {
			return null;
		}
	}

	public static String converteDataParaTexto(Date data) {
		return new SimpleDateFormat("dd/MM/yyyy").format(data);
	}

	public static Date avancarDiasCorridos(Date dataInicial, int dias)
{
		Calendar c = Calendar.getInstance();
		c.setTime(dataInicial);
		c.add(Calendar.DATE, dias);
		return c.getTime();
	}

}
```

```
String inputDateStr = "28/02/2013";
Date inputDate = Data.converteTextoParaData(inputDateStr);
Date resultDate = Data.avancarDiasCorridos(inputDate, 30);
String resultDateStr = Data.converteDataParaTexto(resultDate);
```

```
public class Data {

	private Date data;
	public Data(String dataStr) {
		try {
			data = new SimpleDateFormat("dd/MM/yyyy").parse(da
taStr);
		} catch (ParseException e) {
			throw new IllegalArgumentException(e);
		}
	}

	public String toString() {
		return new SimpleDateFormat("dd/MM/yyyy").format(data);
	}

	public Data avancarDiasCorridos(int dias) {
		Calendar c = Calendar.getInstance();
		c.setTime(data);
		c.add(Calendar.DATE, dias);
		data = c.getTime();
		return this;
	}

}
```

```
String resultDateStr = new Data("28/02/2013").avancarDiasCorridos(30).toSt
ring();
```

Consequências

- Código muito mais limpo, com maior clareza, legibilidade e intuitivo.
- Código menos extenso, devido ao encadeamento.
- Deve ser usado da maneira correta, e na sequência correta.

Vantagens

- Alta escalabilidade.
- Mais controle sobre como o objeto é criado.
- Melhor legibilidade.

Desvantagens

- O uso impróprio deste padrão pode tornar o código ilegível.
- Não implica na obrigatoriedade de alguns parâmetros.
- Não define ordem para os parâmetros.

Referências

https://medium.com/@sawomirkowalski/design-patterns-builder-fluent-interface-and-classic-builder-d16ad3e98f6c

https://java-design-patterns.com/patterns/fluentinterface/

https://www.martinfowler.com/bliki/FluentInterface.html

https://www.ibm.com/developerworks/library/j-eaed14/

https://ocramius.github.io/blog/fluent-interfaces-are-evil/

http://luizricardo.org/2013/08/construindo-objetos-de-forma-inteligente-builder-pattern-e-fluent-interfaces/