Casa do Código

MUNDOJ

MundoJ

Orientação a Objetos

mini LIVRO

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

Sumário

Versão: 20.4.31

Capítulo 1

CRIANDO SOFTWARE MAIS PRÓXIMO AO CLIENTE COM DOMAIN DRIVEN DESIGN

— Sérgio Lopes

Domain-Driven Design caiu na boca do povo. Quem navega por fóruns e listas de discussão sobre Java, já percebeu o interesse que DDD tem despertado nas pessoas (aqui na Caelum mesmo temos altas discussões sobre o assunto). E, junto com toda essa atenção, também aparecem muitas dúvidas e ideias erradas.

O objetivo deste artigo é ser uma introdução à Domain-Driven Design (DDD), mostrar suas principais ideias e provocar discussões em torno deste tema tão polêmico.

1.1 DOMAIN E UBIQUITOUS LANGUAGE

O ponto fundamental do DDD é o primeiro D, o Domain. Tudo gira em torno desse tal de Domínio. O domínio é, em poucas palavras, o problema que queremos resolver como programa que estamos desenvolvendo. Alguém (um cliente) tem um problema na área de atuação dele (geralmente nada a ver com informática) e contrata uma equipe de programação para ajudá-lo (nós!).

Segundo o DDD, é impossível resolver esse problema satisfatoriamente sem entender direito o que acontece no domínio do cliente. Não basta os desenvolvedores saberem mais ou menos: é necessário entrar fundo no domínio do cliente.

Mas, é claro que nosso objetivo não é nos tornarmos especialistas completos na área do cliente, mas apenas compreendê-la. A palavra-chave para isso acontecer é *conversa*. Conversa constante e profunda entre os especialistas de domínio e os desenvolvedores.

Aqueles que conhecem o domínio em detalhes devem conversar com aqueles que conhecem programação em detalhes. Juntos, tentarão chegar a uma língua comum em que todos consigam se entender e que será usada em todas as conversas. É o que o DDD chama de *Ubiquitous Language* (UL): uma língua baseada nos termos do domínio, não totalmente aprofundada neste, mas suficiente para descrever o problema satisfatoriamente.

1.2 CONSTRUÇÃO DO DOMAIN MODEL

Durante a conversa constante, todos juntos chegarão a um consenso sobre o Domínio. Os especialistas de domínio, eventualmente, criarão simplificações para facilitar a conversa; e os desenvolvedores podem introduzir conceitos técnicos simples.

Com isso, todos criam um modelo do domínio, o Domain Model. Para o DDD, é uma abstração do problema real, desenvolvida em parceria pelos especialistas do domínio e desenvolvedores.

Segundo o DDD, é esse modelo que os desenvolvedores vão implementar em código. Literalmente. Item por item, como foi acordado por todos. Será desenvolvido um código limpo, com

palavras do domínio, que representa, na programação, o domínio em discussão.

Usando DDD, seu programa orientado a objetos deve expressar a riqueza do Domain Model. Qualquer mudança no modelo (e, acredite, isso é muito comum) deve ser refletida imediatamente no código. Se algo do modelo torna-se inviável de se implementar tecnicamente, não se fazum "ajustezinho" no código; o modelo deve ser mudado para ser mais fácil de se implementar. Ou seja, no DDD, sempre seu código será expressão do modelo, que, por sua vez, é baseado totalmente no domínio.

1.3 IMPLEMENTANDO O DOMAIN MODEL

Escrever código elegante é um dos maiores desafios que os programadores enfrentam. Simplesmente escrever por escrever, qualquer ferramentazinha que gere código consegue fazer. Mas escrever bons códigos, legíveis, flexíveis e ricos é o real desafio.

O DDD define uma série de design patterns para facilitar a implementação do modelo em código. Como qualquer design pattern, é uma idEia de como codificar certos problemas comuns de forma elegante. Mas, perceba que o objetivo de um design pattern é ajudar o programador! E o principal para o DDD, como vimos, é o Domain. Os patterns são, portanto, apenas ferramentas que facilitama implementação do Domain Model no código. Mas, com absoluta certeza, esse não é o ponto principal do DDD.

Observo com freqUência infindáveis discussões em torno de patterns do DDD, nas quais surgem receitas mágicas que aparentemente podem ser aplicadas em qualquer projeto e, pronto, temos DDD. Esse é um dos maiores erros que se pode cometer.

Não existe receita pronta, não existe certo ou errado ao escrever

suas classes. Se quer usar DDD, lembre-se do principal: o Domain. Você pode criar um Model riquíssimo e um código muito bem escrito; mas se ele não for expressão do Domain, se não for a partir da língua comum, você não está usando DDD.

Repare que não sou contrário aos patterns, muito pelo contrário. São extremamente úteis para o programador, abrem a cabeça para soluções que normalmente nos atormentam e criam uma padronização nas ferramentas do dia a dia do desenvolvedor. Critico aqui quem encara o DDD como um conjunto de patterns. Não é, e está longe disso.

1.4 DDD (QUASE) NA PRÁTICA

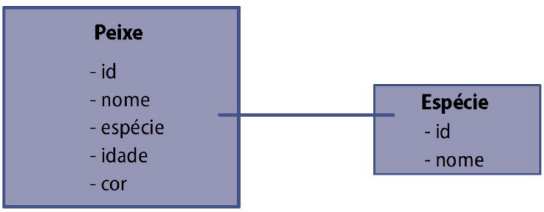
Domain-Driven Design é então sobre o Domain. Sobre todos conversarem a respeito do domínio. Sobre a criação de uma língua comum entre desenvolvedores e especialistas de domínio. Então, para usar DDD, temos de conversar muito! Calma, caro leitor, não o farei conversar com o livro. Mas podemos tentar simular algo.

A conversa

Imagine que somos a equipe de desenvolvedores contratada por alguém interessado em montar uma loja de peixes. E vamos todos lá conversar:

- **Desenvolvedor:** Boa tarde, eu sou programador Java certificado SC#@$.
- **Cliente:** Boa tarde, mas não era bem isso que eu precisava... queria alguém para desenvolver um sistema para mim.
- **Desenvolvedor:** Ops, desculpe. Eu sou um desenvolvedor. Essas coisas que eu falei são detalhes que não interessam mesmo.

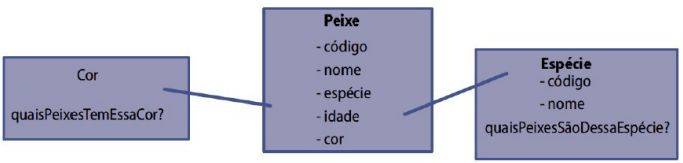
- **Cliente:** Ótimo! Bom, meu nome é Sr. Sell Fish, e quero abrir uma loja que vende peixes que tenho aqui no meu lago.
- **Desenvolvedor:** Certo... Loja de Peixes... tipo um peixe-espada?
- **Cliente:** Não. Peixe-espada é um triquiurídeo, de água salgada. Eu vendo peixes do meu lago, água doce. Tenho lambaris, carpas, tambaquis, tilápias e outros ciclídeos.
- **Desenvolvedor:** ????
- **Cliente:** Deixe-me simplificar: tenho vários peixes, cada um de uma espécie diferente.
- **Desenvolvedor:** Ah, sim. Vários peixes, várias espécies, cada peixe é de uma espécie... Há outras informações importantes sobre cada peixe?
- **Cliente:** Com certeza! Aqui nossos peixes têm um nome de batismo, a idade dele e possuem cores diferentes também.
- **Desenvolvedor:** Humm... veja meu desenho:



- **Cliente:** Interessante. O traço significa que o Peixe tem uma espécie, certo? Mas que raios é "id"?
- **Desenvolvedor:** É um código único que preciso para cada coisa no sistema, coisas da computação. Vou chamar de "código" para facilitar nosso entendimento.

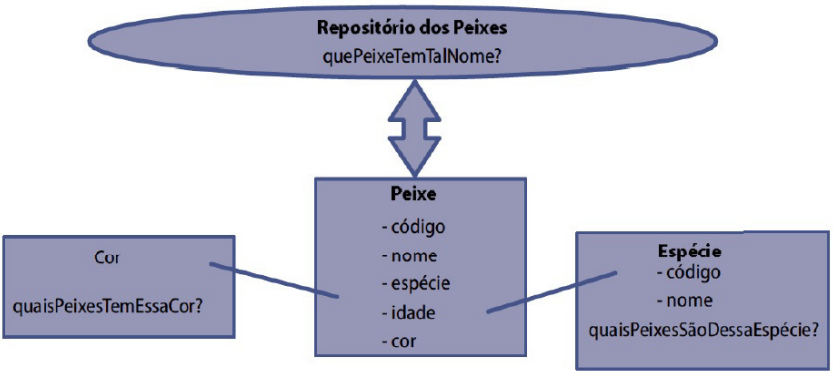
- **Cliente:** Ótimo! Vou precisar cadastrar todos os meus peixes e suas espécies. Quero fazer uma loja diferente cheia de recursos para o cliente encontrar o peixe certo para ele.
- **Desenvolvedor:** Então o cliente terá algumas buscas avançadas... quais seriam importantes?
- **Cliente:** Quero saber quais peixes são de uma determinada espécie ou de uma determinada cor, buscar peixes pelo nome e talvez outras coisas.

- **Desenvolvedor:** Vejamos:



- **Cliente:** Muito bom! Acho que estou entendendo... Mas como vou buscar pelo nome mesmo?

- **Desenvolvedor:** Ah sim... Bom, na verdade você precisa é, dado todos os peixes disponíveis, saber quem tem determinado nome. Eu preciso então procurar esses peixes em algum lugar... posso chamar esse lugar de Repositório de Peixes?
- **Cliente:** Repositório? Pode sim... Um repositório então é onde estão todos os peixes, certo? E, por lá, eu consigo saber qual Peixe tem tal nome?
- **Desenvolvedor:** Isso!



Analisando a conversa

Obviamente a conversa que você leu agora soou bastante artificial. Mas uma conversa real como cliente não deve fugir muito disso. Deve ser simples e girar em torno do domínio. Algumas lições que quero destacar. Evite colocar na conversa termos técnicos desnecessários. E faça o especialista do domínio evitar colocar detalhes muito específicos do domínio na conversa.

Mas tome muito cuidado para não abstrair demais a conversa e acabar fugindo do principal, o domínio. Se você vai precisar entender que ciclídeos são peixes de água doce, aprenda isso. Mas não permita que sejam criadas simplificações ao extremo que façam com que a conversa fuja do domínio real.

Mesma coisa para termos técnicos entrarem na conversa. Se você, desenvolvedor, considera que é preciso um repositório no sistema, faça o especialista de negócio concordar com você e introduza o conceito a ele.

A conversa deve ter uma língua comum, a Ubiquitous Language. E essa língua tem de ser o ponto de encontro dos desenvolvedores com os especialistas de domínio. Se alguém não concorda com algum termo ou acha que determinado termo precisa

fazer parte da UL, isso deve ser conversado.

Tudo isso porque, depois, seu código deve ser exatamente aquilo que foi conversado. A programação deve ser expressão da UL. Jamais programe códigos com conceitos que não fazem parte da UL.

1.5 IMPLEMENTAR

O objetivo aqui não é entrar em detalhes de implementação, mas poderíamos esboçar algum código com base no Domain Model. E, para isso, podemos usar alguns design patterns propostos por Eric Evans, autor do livro *Domain Driven Design*, principal referência sobre o assunto.

Teríamos as classes `Peixe` e `Especie` que seguem o padrão *Entity*. Resumidamente, são objetos com identidade e com ciclo de vida que queremos controlar. A classe `Cor` segue o padrão *Value Object*. É um objeto geralmente imutável, no qual o que realmente interessa é o valor dele (não há identidade, dois objetos com mesmo valor são iguais).

Quanto às entidades do sistema, frequentemente estamos interessados em controlar seu ciclo de vida bem de perto. Criamos vários objetos que precisam ser armazenados em algum lugar seguro, e toda hora precisamos recuperar esses objetos buscando-os das mais variadas formas. O padrão *Repository* representa, no Domain Model, o lugar no qual jogamos objetos e depois os pegamos de volta (mais sobre *Repository* a seguir).

Há muitos outros padrões (*Service*, *Aggregate* etc.), muitas formas de integrá-los e detalhes de implementação que fogem do escopo deste capítulo. Para maior aprofundamento, recomendo o estudo das referências apresentadas no final dele.

1.6 REPOSITÓRIOS, DAOS, LAYERS E MUITA CONFUSÃO

Todo mundo que começa a aprender Orientação a Objetos decentemente logo aprende o que é encapsulamento. E todo mundo que começa a aprender a usar bancos de dados como, por exemplo, Java, aprende o famoso padrão **DAO (*Data Access Object*).

Contam que o DAO é uma boa prática de programação, pois serve para encapsular as particularidades do acesso a banco de dados (ou algo do gênero), e isolar essa complexidade do restante do programa. A ideia é deixar a parte feia da coisa encapsulada no DAO e todo o resto do programa usa objetos bonitinhos sem nem ter ideia do que acontece no banco.

Agora, chega alguém dizendo que existe um tal de repositório que representa o lugar onde nossos objetos são colocados para que depois possamos recuperá-los. E, infelizmente, na cabeça de muita gente, conclui-se que Repositório e DAO são a mesma coisa. **Não são**, embora eu concorde que isso tudo possa gerar muita confusão.

O DDD (e outras pessoas também) propõe que desenvolvamos nosso software divido em camadas (layers) para favorecer encapsulamento, flexibilidade e muitas outras coisas boas. Evans, em particular, propõe a divisão em 4 camadas principais: *Presentation Layer*, *Application Layer*, *Domain Layer* e *Infrastructure Layer*.

A história toda de DDD está, obviamente, focada exclusivamente na **Domain Layer**. É onde você vai implementar o Domain Model, escrever o código que representa o Domain e tudo aquilo que discutimos. Mas, nós sabemos que, alguma hora, vamos precisar de coisas fora do Domain (um HTML para mostrar a mensagem para o usuário, um SQL pra fazer uma busca, um execute em uma Action do Struts etc.). Por isso existem as outras camadas.

Escrever SQL e código de manipulação de banco de dados não faz parte da Domain Layer (a menos que seu domínio seja bancos de dados). Tudo isso faz parte da camada de infraestrutura, lugar no qual seu DAO deve estar. Perceba que, usando DDD ou não, é boa prática ter o DAO; mas o DDD mesmo não diz nada sobre isso, já que ele está fora do Domain.

Repositório é um conceito do Domain Layer, tem de fazer parte do Domínio. Perceba a diferença: DAO surge do problema de encapsular coisas feias de infraestrutura; Repositório surge da necessidade do cliente de obter objetos do domínio. O cliente não tem ideia de banco de dados, não sabe o que é SQL.

O nome repositório não deve ser algo interno ao código, mas **deve fazer parte da Ubiquitous Language**, deve aparecer nas conversas e no Domain Model. Ou seja, repositório deve ser um conceito que o especialista de domínio também entende e, por que está no Model, é ele que vai para o código.

Não há problema em trazer palavriado técnico para a Ubiquitous Language, desde que o príncipio da UL seja mantida: todos entendem o conceito. E, se eventualmente no contexto do domínio sendo tratado, outro nome faça mais sentido que repositório, esse nome deve ser usado (mesmo que nós, técnicos, saibamos que, no fundo, aquilo é um *Repository*). Mas, claro, o nome DAO não faz parte da UL.

Entendida a diferença entre DAO e Repository, fica a dúvida: como implementar um Repositório? Existe sim alguma relação entre Repositórios e DAOs, visto que geralmente as buscas do cliente são feitas no banco de dados. Mas as coisas, como vimos, estão em layers diferentes. Há várias implementações possíveis (e nenhuma regra mágica fixa), todas boas segundo o DDD desde que sigamos os conceitos discutidos.

Só para exemplificar: eu gosto de representar o Repositório na camada do domínio por meio de uma interface Java. Para casos mais simples, implemento a interface direto no DAO na camada de infraestrutura; em casos mais complexos, meu repositório pode ser uma classe concreta que delega coisas para o DAO da infraestrutura. Aqui na Caelum mesmo, usamos abordagens diferentes dependendo da situação.

Mas lembre-se: isso não é regra! Desenvolva pensando no domínio e temos DDD.

1.7 CONSIDERAÇÕES FINAIS

Domain-Driven Design tenta trazer mais qualidade para o desenvolvimento de software, tanto para o processo de desenvolvimento quanto para o produto final. E, para isso, parte do princípio de que o software deve seguir a risca ideias do domínio para o qual ele foi desenvolvido.

1.8 REFERÊNCIAS

EVANS, Eric. *Domain-Driven Design:* Tackling Complexity in the Heart of Software. Editora Addison-Wesley, 2004.

AVRAM, Abel; MARINESCU, Floyd. *Domain Driven Design Quickly*. Info.Q, 2006. Disponível em: http://www.infoq.com/minibooks/domain-driven-design-quickly.

Capítulo 2

PROJETANDO E CODIFICANDO UMA DSL INTERNA

"Os conceitos de Domain-Specific Language (DSL) estão ficando cada vez mais próximos dos desenvolvedores, embora atualmente existam pouquíssimas boas referências. Uma ótima referência é o novo livro de Martin Fowler, no qual o autor trata o assunto dividindo-o em dois ramos, DSL interna e externa. Este capítulo vai explorar a criação de uma DSL interna utilizando a linguagem Java."
— por Leandro Ribeiro Moreira

As empresas desejam que as mudanças nos negócios sejam cada vez mais hábeis, logo os sistemas de informação devem acompanhar essa tendência na mesma intensidade. Criar software de qualidade exige bons profissionais. Atualmente, alguns "gurus" da área estão notando que devem manter o foco do desenvolvimento nos componentes relacionados ao domínio, para assim obter sistemas flexíveis.

O domínio como o centro da aplicação é uma das práticas aconselhadas do modo DDD de se criar sistemas. Visto a atual tendência de se valorizar o domínio ao máximo, o assunto DSL (que já é antigo) volta a tona com novos ares. Normalmente, quando fazemos uma primeira leitura sobre DSL, pensamos que é algo ainda intangível, porém nem chegamos a perceber que o mundo dos

softwares está repleto de DSLs, como, por exemplo, Hamcrest, Jmock, Hibernate criteria e Joda time.

Os sistemas operacionais baseados no UNIX têm uma gama muito grande de DSLs, como, por exemplo, alguns formatos de arquivos de configuração. Uma DSL é uma linguagem voltada para um domínio específico diferentemente das linguagens de programação tradicionais, como Java, C#, Ruby e outras que são de uso geral. Os pesquisadores classificam a DSL em dois tipos: interna e externa.

Uma DSL interna pode ser definida como um estilo de se codificar APIs que sejam mais ricas em expressividade para um domínio específico, que possam até mesmo gerar expressões legíveis a alguém que não saiba nada sobre Java, mas que tenha experiência no domínio em questão. Criar uma linguagem comum que seja compreensível tanto ao expert do domínio quanto aos desenvolvedores, ou seja, uma ubiquitous language. Isso pode ajudar muito o desenvolvimento de uma DSL.

Por ser um assunto novo a muitos desenvolvedores, há confusão dos conceitos, e alguns acreditam até que DSL interna e externa são sinônimos. Os próprios pesquisadores do assunto às vezes entram em contradição. Por exemplo, para alguns, uma DSL interna e uma interface fluente são sinônimos; já para outros, há algumas diferenças.

Há vários padrões para se escrever uma DSL interna. Normalmente, o padrão mais usado na construção de DSL internas é o *method chainning*. Ele oferece meios para que os objetos façam chamadas em sequência a própria instância do objeto. É importante notar que, devido ao fato das DSLs internas serem escritas sobre uma linguagem de programação (a qual se denomina *linguagem hospedeira*), elas sofrem as mesmas limitações da linguagem em que foram escritas.

Interface fluente

No ano de 2005, Martin Fowler e Eric Evans estavam em um Workshop conversando a respeito de um estilo de interface. Então, os dois resolveram nomear este de interface fluente. Mais tarde, por volta do ano de 2007, Fowler afirmou que, da perspectiva de API, vê uma interface fluente como sinônimo de DSL interna.

2.1 EXEMPLO

Demonstrarei a criação de uma DSL interna na prática. Para acompanhá-la, basta ter instalado o JDK 6. O uso de um IDE pode auxiliar na codificação. Para facilitar a criação de uma DSL interna, é mais fácil escrever primeiro a API, para depois verificar a viabilidade de implementação.

O exemplo trabalhado tratará do domínio de vídeo locadoras. Pelo fato de que escrever algo complexo gastaria muito tempo, o exemplo é bem reduzido e abstrai maiores dificuldades.

O processo de locação de vídeos é simples: o cliente vai à locadora, escolhe os vídeos, se dirige ao balcão para concretizar a locação, o caixa solicita a identificação da pessoa, registra vídeo a vídeo, pergunta se o pagamento será realizado na entrega ou no momento da locação, o caixa informa em qual data deverão ser entregues os vídeos, e entrega o comprovante de locação para o cliente.

Para começar, crie a classe que usará a DSL interna. Essa classe se chamará `Testadora` e terá, inicialmente, o método `main()`, conforme listagem a seguir.

Listagem 2.1 - Pré-implementação da classe consumidora:

```
package br.com.mundojava.DSLinterna;

public class Testadora{
    public static void main(String[] args){
    }
}
```

Partindo da descrição de como funciona o processo de locação, implemente uma pequena parte que represente o momento em que o cliente se identifica. Insira-a dentro do método `main` da listagem anterior.

```
Cliente paulo = new Cliente("19784567892","Paulo");
Locacao locacao = Locacao.para(paulo);
```

Note que o método `para()`, da classe `Locacao`, implementa o padrão **FactoryMethod** com um nome não muito comum, mas de alta legibilidade. Seguindo a descrição, o caixa registra vídeo a vídeo, e pergunta ao cliente para quando será o pagamento. Este trecho pode ser descrito da seguinte forma:

```
locacao.adicionar(tropaDeElite, osSimpsons, vanillaSky)
    .paraDevolver(daquiA(DOIS_DIAS)).aPagar();
```

Na simples instrução do código está sendo feito a solicitação do registro de uma locação de três filmes para serem devolvidos em dois dias e com pagamento agendado para o momento da entrega. Perceba que, se o código fosse mostrado a um expert da locadora, provavelmente ele entenderia o seu intuito.

Depois de registrada a locação, um cupom é impresso e entregue ao cliente. O código que poderia sintetizar esse momento seria simplesmente:

```
GerenciadorLocadora.imprimir(locacao);
```

O código parcial, usado na classe consumidora dessa DSL, é apresentado na listagem a seguir.

Listagem 2.2 - Implementação parcial da classe consumidora:

```
package br.com.mundojava.DSLinterna;

public class Testadora{
    public static void main(String[] args){
        Cliente paulo = new Cliente(“19784567892”,”Paulo”);
        Locacao locacao = Locacao.para(paulo);
        locacao.adicionar(tropaDeElite, osSimpsons, vanillaSky)
            .paraDevolver(daquiA(DOIS_DIAS))
            .aPagar();
        GerenciadorLocadora.imprimir(locacao);
        }
}
```

Até o momento, só foi escrita a API, sem nenhum código que a implemente. Pode ser de grande ajuda tentar codificar uma DSL interna a partir da perspectiva de seu uso, que reflita o mais próximo possível do domínio.

Agora, para implementar o código, crie uma classe chamada `Cliente`, que possua um construtor e dois atributos: `nome` e `cpf`. O código-fonte pode ser visualizado na listagem a seguir.

Listagem 2.3 - Implementação da classe cliente:

```
package br.com.mundojava.DSLinterna;

public class Cliente{

    private String cpf;
    private String nome;

    public Cliente(String cpf, String nome){
        this.cpf = cpf;
        this.nome = nome;
    }
    //getters omitidos

}
```

O próximo passo é implementar a classe `Locacao`, a classe que tem um método fábrica nomeado `para()`, e recebe um objeto `Cliente` como parâmetro e devolve uma instância de si. O método

`adicionar()` se aproveita da funcionalidade `var args` do Java e recebe um ou vários vídeos.

O método `paraDevolverEm()` recebe uma data que representa a data de devolução do vídeo. Há mais dois métodos, um `jaPago()` e outro `aPagar()`, que expressam a situação do pagamento da locação. A figura a seguir mostra o diagrama de classe e a listagem a seguir demonstra o código.

Figura 2.1: Diagrama de classes

Listagem 2.4 - Implementação parcial da classe `Locacao` :

```
package br.com.mundojava.DSLinterna;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Locacao{

    private Cliente cliente;
    private List<Video> relacaoVideo = new ArrayList<Video>();
    private Date dataDevolucao;
    private boolean pago;

    private Locacao(Cliente cliente){
        this.cliente = cliente;
    }

    public static Locacao para(Cliente cliente){
        return new Locacao(cliente);
    }
```

```
    public Locacao adicionar(Video... videos){
        for (Video vd : videos){
            relacaoVideo.add(vd);
    }
    return this;
    }

    public Locacao paraDevolver (Date data){
        dataDevolucao = data;
        return this;
    }

    public Locacao aPagar(){
        pago = false;
        return this;
    }

    public Locacao jaPago(){
        pago = true;
        return this;
    }

    //getters omitidos
}
```

Na implementação da classe `Locacao`, é notável a necessidade da criação de outro componente, a classe `Video`. Para oferecer maior riqueza à DSL interna, a classe `Video` será projetada mais próxima da definição dos experts em vídeo. Segue a descrição: um vídeo tem um título original, título traduzido, um ou mais gêneros, atores que compõem o elenco e um pequeno resumo.

Como anteriormente, inicia-se escrevendo a API, para posteriormente implementá-la.

```
Video theSimpsonsMovie = new Video(“Os Simpsons”);
theSimpsonsMovie.nomeOriginal(“The Simpsons Movie”)
    .com(“Homer, Marge, Lisa, Bart ...”)
    .doGenero(“animação/comédia”)
    .pequenoResumo(“A família mais querida do mundo agora " +
        "em um longa-metragem...”);
```

Se fosse usado o modo tradicional de se codificar, a API ficaria um pouco diferente.

```
Video theSimpsonsMovie = new Video(“Os Simpsons”);
theSimpsonsMovie.setNomeOriginal(“The Simpsons Movie”);
theSimpsonsMovie.setAtores(“Homer, Marge, Lisa, Bart ...”);
theSimpsonsMovie.setGeneros(“animação/comédia”);
theSimpsonsMovie.setResumo(“A família mais querida do mundo " +
    "agora em longa-metragem...”);
```

O código da classe `Video` é apresentado na listagem a seguir.

Listagem 2.5 - Implementação da classe `Video` :

```
package br.com.mundojava.DSLinterna;

public class Video{
    private String nome;
    private String nomeOriginal;
    private String genero;
    private String elenco;
    private String sinopse;

    public Video(String nome){
        this.nome = nome;
    }

    public Video nomeOriginal(String nome){
        this.nomeOriginal = nome;
        return this;
    }

    public Video doGenero(String genero){
        this.genero = genero;
        return this;
    }

    public Video com(String elenco){
        this.elenco = elenco;
        return this;
    }

    public Video pequenoResumo(String resumo){
        this.sinopse = resumo;
        return this;
    }

    //getters omitidos
}
```

Voltando à proposta inicial, insira a criação dos filmes no

código.

```
public static void main (String[] args){
    Cliente paulo = new Cliente(“19784567892”,”Paulo”);
    Locacao locacao = Locacao.para(paulo);

Video tropaDeElite = new Video(“Tropa de Elite”);
tropaDeElite.nomeOriginal(“Tropa de Elite”)
    .doGenero(“ação/crime/drama”)
    .com(“Wagner Moura, Caio Junqueira...”)
    .pequenoResumo(“Filme retrata o trabalho do B.O.P.E. "+
        "em ação...”);

        //a criação dos outros filmes omitida

    locacao.adicionar(tropaDeElite, osSimpsons, vanillaSky)
    .paraDevolver(daquiA(DOIS_DIAS))
    .aPagar();
    GerenciadorLocador.imprimir(locacao);
}
```

Note que ainda há problemas não resolvidos: a constante `DOIS_DIAS`, o método `daquiA()` e a classe `GerenciadorLocador` ainda inexistem na solução atual. Para solucionar o problema da constante, basta criar a classe `DataUtil` (apresentada na listagem a seguir) e incluir um `static import` para essa classe em `Testadora`. Isso solucionará o primeiro problema.

```
import static br.com.mundojava.DSLinterna.DataUtil.*;
```

Listagem 2.6 - Implementação da classe `DataUtil`:

```
package br.com.mundojava.DSLinterna;

import java.util.Date;

public class DataUtil{
    public static final int UM_DIA = 1;
    public static final int DOIS_DIAS = 2;
    public static final int TRES_DIAS = 3;
    public static final int QUATRO_DIAS = 4;

    public static Date daquiA(int dataMs){
        dataMs = new Date().getDate() + dataMs;
```

```
        Date data = new Date();
        data.setDate(dataMs);
        return data;
    }
}
```

O segundo problema requer a criação de uma classe chamada `GerenciadorLocador`. Esta deverá conter um método estático chamado `imprimir()`, como pode ser visto na listagem a seguir.

Listagem 2.7 - Implementação da classe `GerenciadorLocador`:

```
package br.com.mundojava.DSLinterna;

public class GerenciadorLocador {
    public static void imprimir(Locacao suaLocacao) {
        System.out.println("Nome: " +
            suaLocacao.getCliente().getNome());

        for (Video mv : suaLocacao. getRelacaoVideo()){
            System.out.println(mv.getNome() + " - " +
                mv.getGenero());
            }

        System.out.println("Total: R$ " + suaLocacao.getTotal());
        System.out.println("Devolver em " +
            suaLocacao.getDataDevolucao());
        System.out.println("Assinatura:________");
        }
}
```

Para um melhor resultado, foi implementado um método `getTotal()` na classe `Locacao`.

```
public String getTotal() {
    if (!pago) {
        int valor = this.relacaoVideo.size() * 2;
        return valor + ",00";
    } else {
        return "0,00";
    }
}
```

O teste final pode ser feito executando a classe `Testadora`.

Algumas melhorias poderiam ser feitas utilizando a classe `Calendar` em vez de `Date`, ou elaborando uma solução mais elegante para classe `DataUtil`, por exemplo.

2.2 BOAS OPORTUNIDADES PARA O USO

Aqui demonstrei alguns conceitos que uma DSL interna deve ter e como implementá-los. Será um exemplo relativamente simples, mas com alto valor didático. Normalmente, a criação de uma DSL interna traz mais benefícios para quem está desenvolvendo uma biblioteca ou framework, reduzindo a curva de aprendizado de seus usuários.

Outra situação seria o desenvolvimento de um grande sistema, no qual várias equipes são responsáveis por subprojetos diferentes. Comumente uma determinada equipe precisará usar algum componente ou serviço que outra equipe está desenvolvendo. Nesse caso, uma DSL interna pode ser uma solução interessante.

Por exemplo, a equipe A precisa saber se uma pessoa já esteve ou não de licença médica. No entanto, a equipe A está engajada em uma parte do sistema que é mais relacionada ao domínio financeiro. Se a equipe B (responsável pelo desenvolvimento dos sistemas de saúde) oferecesse uma DSL interna, a equipe A teria menos dificuldades para solução do problema.

```
boolean tirouLicenca = APIGestaoMedica.
    funcionario(pedro).jaTirouLicenca();
```

Em outra ocasião, a equipe B precisa saber a data do pagamento de um boleto que pertence a um funcionário. Bastaria a equipe A fornecer uma DSL interna.

```
Date dataPagamento = APIFinanceiro.
    funcionario(pedro).pagou(boleto).quando();
```

Os exemplos foram somente para demonstrar o quanto equipes

diferentes em um grande projeto podem se beneficiar do uso de DSL internas. Se você é usuário de algum framework, pense o quanto seria bom se o time de desenvolvedores oferecesse algumas DSL internas. Possivelmente, os usuários teriam um aprendizado mais rápido e intuitivo.

2.3 DESVANTAGENS

Como ainda há poucas pessoas pesquisando profundamente sobre DSLs, sabe-se pouco sobre os contras do uso de DSL internas. Uma desvantagem notável é que seu código orientado a objetos deve sofrer algumas modificações "estranhas", como por exemplo, ter métodos que retornem instância de si mesmo.

Outro ponto negativo, quanto mais expressividade a API fornecer, maior será a dificuldade de se implementar e manter o código. O que é bom para os clientes da API pode ser uma tormenta para os desenvolvedores dela. Nem sempre o retorno da própria instância (`this`) resolverá todos os problemas, às vezes é necessário criar objetos intermediários, aumentando mais a dificuldade da escrita da API.

2.4 PERSISTÊNCIA

Criar uma DSL interna em aplicações nas quais o estado dos objetos necessita ser persistido requer a criação de um mecanismo para que tal tarefa seja feita. Há vários métodos para realizar a persistência em uma DSL interna. Então, serão apresentados dois que se destacam pela simplicidade e facilidade de compreensão.

O primeiro modo é criar um método `persistir()` , o qual deve ser chamado após a conclusão de solicitação de serviços a DSL interna.

```
locacao.adicionar(tropaDeElite)
    .paraDevolver(daquiA(DOIS_DIAS))
    .aPagar()
    .persistir();
```

Esta primeira abordagem deixa a DSL menos relacionada ao domínio, o que vai ao contrário do propósito inicial das DSLs internas. Outro método é encadear as mensagens aos objetos de forma que facilite a persistência na DSL sem que o cliente perceba.

```
GerenciadorLocacao.imprimir(locacao.adicionar(tropaDeElite)
    .paraDevolver(daquiA(DOIS_DIAS))
    .aPagar());
```

No método `imprimir`, que recebe uma locação, você pode persistir o objeto sem que isso fique tão claro para o cliente da DSL interna.

```
public static void imprimir(Locaca locacao){
    //códigos para prover persistência e imprimir comprovante...
    repositorioLocacao.adiciona(locacao);
}
```

2.5 CONSIDERAÇÕES FINAIS

Enfim, criar uma DSL interna é somente criar meios para que a API seja mais fácil de entender e usar em um contexto restrito a um domínio específico. Também é importante citar que a DSL não precisa ser legível a qualquer um, e sim entendível a um expert do domínio.

Por se tratar de uma área ainda em estudo e evolução, todo e qualquer cuidado são poucos. Ler é o melhor conselho. Obtenha informações de diversas fontes, facilitando ter uma posição crítica sobre o andamento desses novos termos que poderão tomar conta do vocabulário, já extenso, dos desenvolvedores.

Existem divergências de opiniões e, possivelmente, sempre haverá entre os pesquisadores do assunto e os desenvolvedores em

geral. Mas é essa discussão que faz com que o conhecimento seja compartilhado, analisado, criticado e adotado ou descartado.

2.6 REFERÊNCIAS

EVANS, Eric. *Domain-Driven Design:* Tacking Complexity in the Heart of Software. Addison-Wesley Longman Publishing Co., 2003.

FOWLER, Martin Fowler. *Refactoring:* Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., 1999.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns:* Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., 1995.

- DSL Patterns: https://martinfowler.com/dslCatalog/
- Guilherme Chapiewski: http://gc.blog.br
- Leandro Moreira: http://archsofty.blogspot.com.br/ e https://leandromoreira.com.br

Capítulo 3

MELHORIA CONTÍNUA DO CÓDIGO COM REFATORAÇÃO

"Muitas pessoas acreditam que a modelagem de uma aplicação é uma atividade que deve vir antes da sua implementação. Com essa ideia, a modelagem e a estrutura interna da aplicação não acompanham sua evolução e acabam se tornando rapidamente obsoletas. O objetivo deste artigo é apresentar a técnica da refatoração, que permite que a atividade de modelagem aconteça de forma contínua e a qualidade do código seja sempre aprimorada." — por Eduardo Guerra

Em uma abordagem clássica de engenharia de software, o projeto das classes é realizado antes da implementação. A aplicação de vários padrões de projeto é feita de forma a procurar flexibilizar a estrutura o máximo possível. Depois do projeto feito, os programadores possuem a função de apenas implementar os métodos previstos no diagrama de classe, baseados nos diagramas de sequência ou colaboração.

Se for preciso alterar algo que tenha impacto na forma como o sistema foi modelado, os diagramas são alterados e o programador que se vire para adaptar o código que já está pronto. Isto quando o projetista já não "saiu de cena" e o programador precisa dar aquele "jeitinho" para inserir aquele novo requisito.

Será que esta é a melhor forma de se fazer as coisas? Será que alguém já viu um projeto com a modelagem tão pesada que parece estar tentando matar uma mosca com uma bazuca? Que jogue a primeira pedra quem nunca olhou um código, muitas vezes de sua própria autoria, e pensou: "Isto não está cheirando muito bem...". Sem falar quando os programadores decidem por conta própria que a modelagem não está boa e, a partir de um ponto, ignoram todo o trabalho que foi feito, começando a mudar várias coisas de forma descontrolada, muitas vezes até violando a arquitetura do sistema.

Se você se identificou com algum desses problemas, ficarei muito feliz em ter a oportunidade de lhe apresentar a técnica da refatoração. Usar a refatoração não significa que você precisa jogar fora todo o conhecimento de seus livros sobre UML e padrões de projeto. A refatoração é uma ferramenta poderosa que pode ser usada em conjunto com outras técnicas para aumentar cada vez mais a qualidade do código.

Com isso, ele fica mais claro e limpo, fazendo com que melhorias no código ou na modelagem possam ser feitas de forma mais segura e controlada. O objetivo deste capítulo é mostrar, na teoria e na prática, a técnica da refatoração e como ela pode ser utilizada na melhoria contínua de um projeto de software.

3.1 O QUE É REFATORAÇÃO?

Segundo Martin Fowler, o autor do livro *Refatoração: Aperfeiçoando o Projeto de Código Existente*, a refatoração é uma técnica de modificação de um software de forma a não alterar o seu comportamento externo e melhorar sua estrutura interna, sendo uma forma disciplinada de se limpar o código que minimiza as chances de introdução de bugs.

Desta definição, um dos conceitos importantes que pode se

retirar é a não alteração do comportamento do código. No momento em que se realiza uma refatoração, não se tem a intenção de se adicionar uma nova funcionalidade, ou alterar uma já existente. A única motivação de uma refatoração deve ser deixar o código mais simples de ser modificado.

O objetivo de uma refatoração é tornar o código mais claro e limpo, seja com uma mudança simples, como alterar o nome de uma variável, ou complexa, como a mudança de uma estrutura de classes.

Por que se deve refatorar?

A resposta é simples: um código que é constantemente refatorado é mais fácil de ser alterado, ou seja, um código mais fácil de ler, sem duplicação, sem lógicas condicionais mirabolantes etc. Com um código fácil de ser alterado, os desenvolvedores podem se preocupar em criar um código enxuto para os requisitos que eles possuem hoje.

Caso no futuro seja necessário alterar os requisitos, com este código constantemente refatorado é mais fácil de implementar as modificações. Desta forma, não existe mais a necessidade de ficar inventando soluções que procuram resolver problemas que ainda não existem e que, caso venham a existir, muito provavelmente não serão da forma que haviam sido previstas inicialmente.

É fácil escrever um código que o computador entenda, difícil é escrever um código que as outras pessoas entendam.

Dentre as vantagens de se utilizar a refatoração, podemos citar:

- Melhora continuamente a estrutura da aplicação, fazendo com que a modelagem, e não só o código, acompanhe as mudanças de requisito;

- Deixa o código-fonte mais legível, tornando mais fácil a sua compreensão;
- Ajuda a encontrar bugs no código. Uma forma de se encontrar um bug em um código confuso é ir refatorando e simplificando-o até encontrar o problema;
- Aumenta a velocidade de desenvolvimento, visto que é bem mais fácil de se acrescentar funcionalidades em um código claro e limpo;
- Ajuda a preparar o código para receber uma nova funcionalidade, de forma a não ser necessário complicar o código enquanto não for preciso.

3.2 A REFATORAÇÃO NO DIA A DIA

A modelagem conhecida como *upfront* é aquela em que todo o projeto é realizado antes da codificação, sendo deixado para os implementadores apenas a tarefa de codificar os métodos. Esta abordagem, como foi citado na introdução, possui seus problemas, como a dificuldade de adaptação a mudanças e o risco de criar complexidade desnecessária.

Nesta abordagem, é gasto bastante tempo na tentativa de se prever todos os problemas e cenários, sendo que quase sempre existirão questões que só serão identificadas no momento da implementação.

Por outro lado, se apenas a refatoração fosse usada como ferramenta de modelagem, a implementação ocorreria da primeira forma que viesse a cabeça, e depois o código seria reestruturado até a modelagem chegar no ponto em que se deseja. Esta abordagem também tem os seus problemas, pois existe dificuldade de iniciar a implementação sem se pensar muito no problema e sem saber direito por onde começar. Existiria também um retrabalho

excessivo, visto que o número de refatorações para ajustar o código seria grande.

A solução para este dilema é aproveitar o melhor das duas abordagens. Deve ser criada uma modelagem, antes de se começar a desenvolver, que servirá como uma abordagem inicial para tentar resolver o problema. O diferencial neste caso é que esta modelagem não precisa tentar prever todas as situações que podem ocorrer e, por este motivo, não precisa ser gasto muito tempo nesta fase.

A partir desta modelagem criada, é iniciada a implementação e, à medida que outras questões vão surgindo, a refatoração serve como um ajuste fino à modelagem criada inicialmente. Também não é preciso tentar contemplar na modelagem inicial flexibilidades para possíveis requisitos futuros, pois, com o código limpo e simples, é possível refatorar a modelagem já existente para a adição deste novo requisito.

Uma famosa frase diz: *"Projete para o futuro, codifique para o momento". Porém, nesta nova abordagem com a refatoração, deve-se criar o código mais claro possível para permitir a adição de novas funcionalidades e evitar prever requisitos futuros, sendo que desta forma deve-se* "Projetar para o momento e codificar para o futuro".*

Quando refatorar?

A refatoração costuma ser aplicada antes ou depois de se implementar uma funcionalidade, porém existe uma regra que sempre deve ser seguida: nunca refatorar durante a inclusão de uma funcionalidade. Se a refatoração é uma técnica de se melhorar o código sem a alteração do comportamento, não faz sentido alterar o comportamento enquanto se refatora.

É comum ver uma refatoração feita antes da adição de uma funcionalidade como uma forma de se preparar o terreno para sua

introdução. Por exemplo, se quando uma classe vai ser adicionada, nota-se que vários comportamentos necessários são iguais ao de uma classe já existente, esta é uma boa hora para a extração de uma superclasse ou para a quebra da classe existente em duas classes, sendo uma para a qual será delegado o comportamento em comum.

A refatoração feita depois da adição da funcionalidade serve para dar aqueles ajustes finais na modelagem do código. No exemplo dado, caso a existência desta classe com comportamento em comum não fosse notada antes do desenvolvimento, a refatoração poderia ser feita depois sem nenhum problema.

A refatoração também pode ser uma ferramenta valiosa quando está se fazendo uma revisão de código, ou mesmo tentando entender algum trecho. Ela também é especialmente útil quando se procura erros em um código alheio.

Como experiência pessoal, posso citar uma situação na qual o código de uma classe estava muito confuso e, para entender o que estava acontecendo, optou-se pela refatoração daquele código. No fim das contas, o código da classe ficou cerca de um quarto do que era inicialmente, sem alteração no comportamento. Desta forma, o erro que estava sendo procurado foi mais facilmente encontrado e, como efeito colateral, o código da classe ficou muito mais enxuto. Certamente um dos maiores prazeres de quem refatora é deletar código duplicado!

Refatorando com segurança

Um sábio ditado popular diz que, em time que está ganhando, não se mexe. Porém, enquanto se refatora, está se mexendo em código que já está funcionando, ou seja, o time está ganhando e, ao se alterar esse código, corre-se o risco da inserção de um bug não intencional. Mesmo com a utilização de ferramentas que fazem as refatorações de forma automatizada, o comportamento do código

ainda pode ser alterado (veja mais na seção *Cuidado na hora de aplicar refatorações* mostra um exemplo).

Em um grande projeto corporativo, não se pode correr o risco de inserir um bug em um sistema que já está em produção. Dessa forma, é necessário algo que garanta que o comportamento do código não foi alterado durante uma refatoração. É neste ponto que se faz presente a importância de se ter uma suíte de testes de unidade. Com todos os testes passando depois de uma alteração, o desenvolvedor tem mais segurança de que o comportamento não foi alterado.

Não está no escopo deste artigo falar sobre testes de unidade, apesar de eles terem um papel fundamental na prática da refatoração. A seção *Refatoração e Extreme Programming* falará um pouco sobre a refatoração no contexto do desenvolvimento orientado a testes. No exemplo que será apresentado, será mostrada uma classe de testes de unidade da classe que será refatorada, a fim de garantir que seu comportamento não será alterado.

"Mau cheiro" no código

"Mau cheiro" é o termo usado quando um determinado código possui algum indício de que está precisando ser refatorado. Um código possuir um "mau cheiro" não significa necessariamente que ele deve ser refatorado, mas que deve ser investigado se ele precisa ou não de refatoração.

A detecção de "mau cheiro" pode ser algo bem intuitivo! Quando houver alguma dificuldade para entender um código ou se identificar algo que não parece muito legal, provavelmente aquele código precisa ser refatorado.

Um dos "maus cheiros" que certamente precisa de refatoração é a existência de código duplicado. Nesse caso, é preciso refatoração

mesmo! Outros "maus cheiros" dificilmente não indicam algum problema, como a existência de uma classe muito longa, um método muito longo, ou mesmo uma lista de parâmetros muito longa. Elementos muito longos são de difícil análise e manutenção, e quase sempre existe uma forma melhor de se obter o mesmo comportamento.

Outros "maus cheiros" não são tão óbvios assim. Quando uma classe começa a acessar muitos atributos de outra classe, temos um caso clássico de "inveja de funcionalidade". Isto nos leva a seguinte pergunta: será que a responsabilidade deste trecho de código não deveria em outra classe?

Existe também o caso clássico das variáveis que gostam de andar sempre juntas, como em parâmetros de métodos ou em variáveis de instância. Talvez esta "amizade" entre estas variáveis signifique que elas deveriam formar uma nova classe.

Com certeza, um dos tipos de "maus cheiro" mais polêmico é a existência de comentários no código. Eu sei que durante a sua vida inteira lhe falaram que isto era uma boa prática, porém muitas vezes os comentários servem como "desodorantes" para um código difícil de entender. Muitas vezes, é muito mais fácil descrever um código "macarrônico" em um comentário, do que fazer com que ele fique claro para quem estiver lendo. O comentário em si não é mau, mas a presença dele pode indicar que um código de difícil entendimento precisa ser melhorado.

O objetivo nesta seção foi mostrar apenas alguns dos muitos tipos de "maus cheiros" existentes, para que se possa ter ideia do tipo de coisa que pode indicar a necessidade de uma refatoração. No livro do Martin Fowler sobre refatoração, citado anteriormente, existem diversos outros tipos de "mau cheiro" documentados. Como foi dito, nem sempre um "mau cheiro" indica um problema real no código, porém eles são dicas de que alguma coisa pode estar

errada.

Refatoração x Desempenho

Quando Fowler definiu refatoração, ele deixou bem claro que o objetivo da refatoração era deixar o código mais limpo e claro. O ato de refatorar, muitas vezes, não tem a intenção de melhorar desempenho, sendo que inclusive alguns códigos depois de serem refatorados apresentam um desempenho inferior em relação ao código original.

Por exemplo, se é criado um método para substituir uma variável local que recebe um cálculo feito com variáveis de instância, se o método for invocado mais de uma vez, o cálculo será feito mais de uma vez e, com certeza, o desempenho será inferior.

Mas será que é aceitável ter uma perda de desempenho por um código mais claro? Isso vai depender muito da perda e da melhoria na clareza no código. Não existe uma regra que valerá para todos os casos.

Nessa questão, o que vai pesar bastante são os requisitos, sendo que em alguns casos estas perdas no desempenho serão perfeitamente aceitáveis e, em outros casos, não. De qualquer forma, ter um código claro e limpo facilita bastante na otimização do desempenho.

Refatore o código de forma a deixá-lo o mais claro e limpo possível. Depois, se o desempenho não for satisfatório, será muito mais fácil fazer os ajustes necessários.

Quando é difícil refatorar

Existem alguns casos em que a refatoração se torna algo difícil de ser feito e, em sua grande maioria, devido a um alto grau de

acoplamento entre partes da aplicação. Quando a refatoração a ser feita envolve banco de dados, principalmente quando este é usado em mais de uma aplicação, a situação se complica. O acoplamento da aplicação com a estrutura do banco acaba sendo bem alto, e esta dependência dificulta a refatoração.

A adição de uma camada de persistência que proteja a aplicação dos detalhes do banco, utilizando, por exemplo, JPA, ajuda bastante, mas não resolve completamente o problema. O livro *Refactoring Databases: Evolutionary Database Design*, de Scott Ambler e Pramodkumar Sadalage, fala com mais detalhes sobre a refatoração de banco de dados, que é muito mais trabalhosa e envolve muito mais passos do que uma refatoração em código-fonte.

Outro problema é quando a refatoração esbarra em uma interface com outro sistema. Muitas vezes não se tem acesso à aplicação que está usando a sua classe, e a mudança da assinatura de um método, por exemplo, faria com que esta outra aplicação não funcionasse com a nova versão.

Quando se trabalha com este tipo de software, como um componente disponibilizado a outras aplicações, a mudança deve ser gradual, primeiro criando o método novo e deixando o método antigo como `deprecated`, e só depois removendo a versão antiga. Neste caso, outra solução seria utilizar uma classe `Adapter` com métodos com a assinatura antiga que invocam os novos métodos da classe com a nova implementação.

Existem trabalhos acadêmicos recentes que buscam soluções para essas questões. O trabalho *Annotations for Seamless Aspect-Based Software Evolution* (ver *Referências*) propõe o uso de anotações em código refatorado, para que a versão antiga do código seja gerada e sincronizada com a versão nova.

3.3 REFATORAÇÃO E EXTREME PROGRAMMING (XP)

Apesar de a refatoração ser uma técnica que pode ser aplicada independente da metodologia que está sendo usada, uma grande parcela de sua popularização perante a comunidade de desenvolvimento foi graças a sua utilização na metodologia Extreme Programming (XP). A refatoração é uma das principais práticas dessa metodologia, e também é parte fundamental da técnica de desenvolvimento conhecida como desenvolvimento dirigido por testes (TDD). Nesta seção, será falado como a refatoração se encaixa na XP e como ela interage com as outras práticas.

Na XP, existe a prática do código coletivo, em que cada artefato de código não possui um dono e pode ser trabalhado por qualquer um da equipe. Isto permite que, sempre que for detectada a necessidade de se refatorar, sendo em um código do próprio desenvolvedor ou criado por outro, a alteração possa ser feita sem problemas, pois o código pertence a todos. A existência da integração contínua faz com que, se uma refatoração feita entra em conflito com o trabalho de outro desenvolvedor, isto seja detectado em um curto espaço de tempo.

A programação em pares e a prática de um ritmo sustentável ajudam o desenvolvedor a ter mais coragem e a cometer menos erros. O uso de padrões de código e a prática de uma modelagem simples deixam o código mais fácil de entender e, consequentemente, mais fácil de refatorar.

Os testes feitos antes do código com certeza são a técnica que melhor apoiam a prática da refatoração. Na XP, a modelagem e a codificação são feitas por meio de uma técnica chamada desenvolvimento dirigido por testes (TDD). Nela, quando se desenvolve uma classe, primeiro se escreve um teste que vai refletir

o comportamento daquela classe perante um determinado cenário de uso.

Com o teste pronto, é feito o mais simples possível para que ele seja executado com sucesso. Estando tudo funcionando, o código é refatorado para a eliminação de duplicação. Com esses pequenos ciclos, o código vai sendo desenvolvido iterativamente, e cada passo é dado com segurança.

Com o uso do desenvolvimento dirigido por testes, o código vai sendo sempre refatorado e melhorado. A suíte de testes que é resultante da utilização desta técnica serve para dar uma grande segurança caso seja necessária a execução de uma grande refatoração.

Desta forma, o uso da refatoração na XP é apoiado por outras práticas, criando uma sinergia no desenvolvimento. Uma parcela do mérito da famosa curva que diz que na XP o custo de mudança no tempo a partir de um determinado ponto se torna constante, pode ser atribuída à prática da refatoração. Isto é verdade porque o ganho em qualidade obtido com uma refatoração contínua do código faz com que a alteração de um requisito seja muito mais fácil de ser inserida.

3.4 EXEMPLO DE REFATORAÇÃO

Nesse momento, acredito que todos devem estar cansados de teoria e ansiosos para ver um código ser refatorado. Quem nunca refatorou um código não sabe como é bom o gostinho de ver tudo ficando mais simples, e isto sem mudar o comportamento da classe!

Para mostrar este exemplo, será utilizado o IDE Eclipse Europa, porém outros IDEs poderiam ser usados sem problema. No Eclipse, as funcionalidades de refatoração podem ser encontradas abrindo o

menu `Refactor`, conforme mostrado na figura seguinte, ou clicando com o botão direito do mouse sobre o código e entrando no submenu `Refactor`.

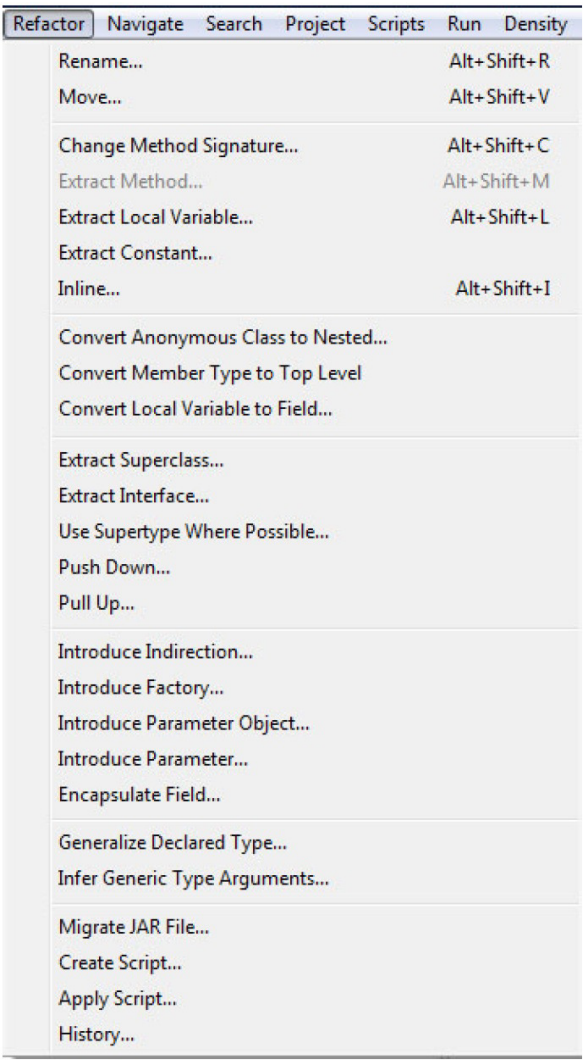


Figura 3.1: Menu de refatoração da ferramenta Eclipse

Contexto do exemplo

O exemplo que será mostrado não tem a intenção de ser completo e com muitas funcionalidades. Seu objetivo é ser didático e ilustrar como um "mau cheiro" é detectado em um código, e como a refatoração procede passo a passo. Para a realização de uma refatoração com segurança, teremos uma classe de teste de unidade que garantirá que o comportamento de nossas classes de exemplo não seja alterado.

A funcionalidade provida por estas classes é muito simples. A classe `ProcessadorDeItens`, apresentada na próxima listagem, possui um método `getItens()` que recebe um `InputStream` e lê em cada linha os dados de um item separados por `;` (ponto e vírgula). Esse método coloca estes dados dos itens em um objeto da classe `Item`, apresentada na listagem seguinte, e retorna uma lista de objetos do tipo `Item`.

Esta classe também possui o método `getValorTotalPorTipo()`, que recebe um `InputStream` e uma String que representa o tipo de item e retorna a soma dos valores totais de cada item (a quantidade vezes o valor unitário) daquele tipo de item. A classe `Item` não possui muita funcionalidade e simplesmente armazena os valores das características de um item.

O fato de esta classe não ter muita funcionalidade já deixa uma "pulga atrás da orelha": será que outra classe está fazendo coisas que eram responsabilidade desta classe?

Listagem 3.1 - Classe `ProcessadorDeItens`:

```
public class ProcessadorDeItens {

    public List<Item> getItens(InputStream input)
        throws IOException {
        List<Item> listItens = new ArrayList<Item>();
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(input));
        String line = null;
        while ((line = reader.readLine()) != null) {
```

```
            Item item = new Item();
            String[] elementos = line.split(";");
            item.setCodigo(elementos[0]);
            item.setTipo(elementos[1]);
            item.setQuantidade(Integer.parseInt(elementos[2]));
            item.setValorUnitario(
                Double.parseDouble(elementos[3]));
            listItens.add(item);
        }
        return listItens;
    }

    public double getValorTotalPorTipo(InputStream input,
            String tipo) throws IOException {
        double valorTotal = 0;
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(input));
        String line = null;
        while ((line = reader.readLine()) != null) {
            String[] elementos = line.split(";");
            if (tipo.equals(elementos[1]))
                valorTotal += Double.parseDouble(elementos[3])
                    * Integer.parseInt(elementos[2]);
        }
        return valorTotal;
    }
}
```

Listagem 3.2 - Classe `Item` :

```
public class Item {

    private String codigo;
    private String tipo;
    private int quantidade;
    private double valorUnitario;

    public String getCodigo() {
        return codigo;
    }
    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }
    public int getQuantidade() {
        return quantidade;
    }
    public void setQuantidade(int quantidade) {
        this.quantidade = quantidade;
```

```
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public double getValorUnitario() {
        return valorUnitario;
    }
    public void setValorUnitario(double valorUnitario) {
        this.valorUnitario = valorUnitario;
    }
}
```

Na próxima listagem está o código do teste de unidade que vai garantir que as modificações no código que forem feitas não afetarão o comportamento externo da classe.

Listagem 3.3 - Teste de unidade para garantir o funcionamento do código:

```
import static org.junit.Assert.assertEquals;

public class TestProcessadorDeItens {

    private final static String dadosDeTeste =
        "3123123;Eletrodomestico;1;1000.00\n"+
        "7567123;Eletrodomestico;3;500.00\n"+
        "3978876;Eletroeletronico;2;700.00\n"+
        "1236543;Movel;7;350.00\n"+
        "9324423;Movel;4;600.00\n";

    @Test
    public void testGetItens() throws IOException {
        ProcessadorDeItens proc = new ProcessadorDeItens();
        List<Item> itens = proc.getItens(
            new ByteArrayInputStream(dadosDeTeste.getBytes()));

        assertEquals("Recuperacao de todos os itens",
                    itens.size(),5);
        assertEquals("Recuperacao de Codigo",
                    "3123123",itens.get(0).getCodigo());
        assertEquals("Recuperacao de Tipo",
                    "Eletrodomestico",itens.get(1).getTipo());
        assertEquals("Recuperacao de Quantidade",
                    2,itens.get(2).getQuantidade());
```

```
        assertEquals("Recuperacao de Valor Unitario",
                    350.00,itens.get(3).getValorUnitario());
    }

    @Test
    public void testGetValorTotalporTipo() throws IOException{
        ProcessadorDeItens proc = new ProcessadorDeItens();
        double totalEletromesticos = proc.getValorTotalporTipo(
            new ByteArrayInputStream(
            dadosDeTeste.getBytes()),"Eletrodomestico");
        double totalMovel = proc.getValorTotalporTipo(
            new ByteArrayInputStream(dadosDeTeste.getBytes()),
                            "Movel");

    assertEquals("Compara o total eletrodomesticos",
                totalEletromesticos,2500.0);
    assertEquals("Compara o total movel",
                totalMovel,4850.0);
  }
}
```

Acertando as responsabilidades das classes

A primeira coisa que me incomoda neste código estão nas linhas dentro do `while` do método `getItens()`. Um trecho de código que chama quatro métodos de uma classe em quatro linhas de código parece-me estar com inveja da funcionalidade desta classe.

Parece fazer sentido que seja responsabilidade da classe `Item` saber como extrair os seus dados separados por `;` de uma String. Desta forma, a primeira refatoração a ser feita é a extração deste trecho de código para um método da classe `Item`.

Infelizmente, esta refatoração não dá para ser feita diretamente e será preciso fazer alguns passos antes de atingir o estado desejado. O primeiro passo é marcar as cinco linhas de código referentes a essa funcionalidade, ir ao menu `Refactor` e escolher a opção `Extract Method`.

Será exibida a janela mostrada na figura a seguir. Note que os parâmetros necessários já vêm configurados, basta dar o nome ao

método, `carregarItemDaString()` e pressionar o botão `Ok`. Caso deseje ver como a classe vai ficar antes de aplicar efetivamente a refatoração, clique no botão `Preview >`. Neste caso, uma janela mostrará todas as modificações que serão feitas no código, conforme figura a seguir.

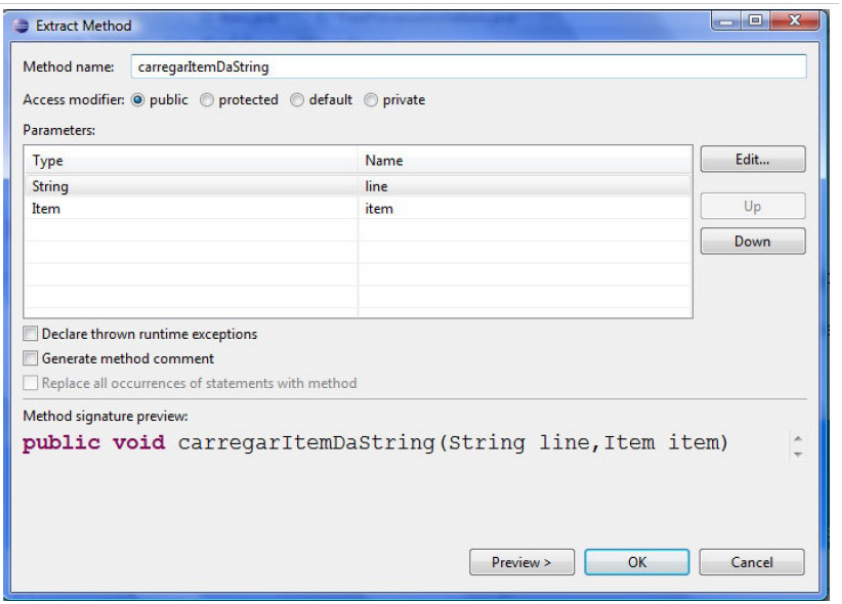


Figura 3.2: Janela pop-up de extração de método

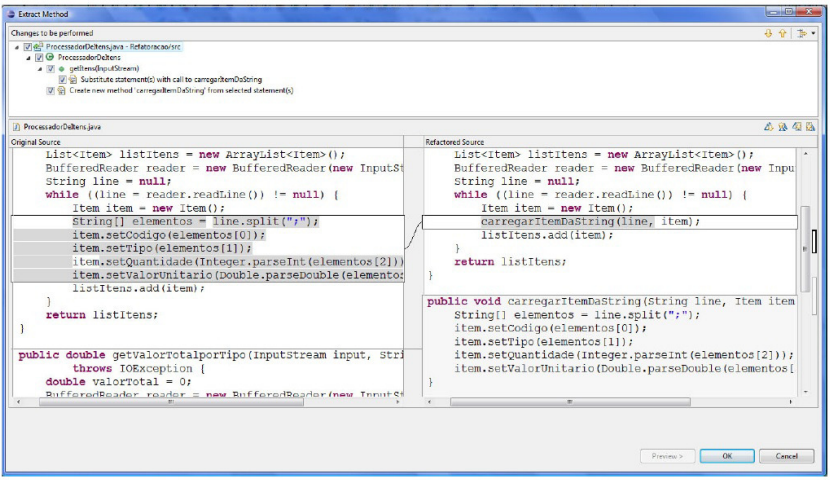


Figura 3.3: Preview das alterações que serão realizadas com a extração do método

Ao concluir a refatoração, será gerado um novo método e trecho de onde o método foi extraído. O código original é substituído pela chamada a este novo método. A seguir, temos o novo método gerado:

```
private void carregarItemDaString(String line, Item item) {
    String[] elementos = line.split(";");
    item.setCodigo(elementos[0]);
    item.setTipo(elementos[1]);
    item.setQuantidade(Integer.parseInt(elementos[2]));
    item.setValorUnitario(Double.parseDouble(elementos[3]));
}
```

Este é um bom momento para a execução da classe de teste! Felizmente, os testes continuam executando com sucesso e pode-se seguir para o próximo passo, que é mover o método extraído para a classe `Item`. Para isso, coloca-se o cursor sobre a assinatura do método e escolhemos no menu `Refactor` a refatoração `Move`.

A janela na figura seguinte é exibida e felizmente já foi identificada que a classe `Item` é uma candidata para receber o método. O nome do método será modificado para

`carregarDaString()` e podemos confirmar a refatoração pressionando `ok`.

Se for apontado um problema de visibilidade do novo método na classe `Item`, depois de confirmar a refatoração, altere o modificador de acesso do método `carregarDaString()` para `public`.

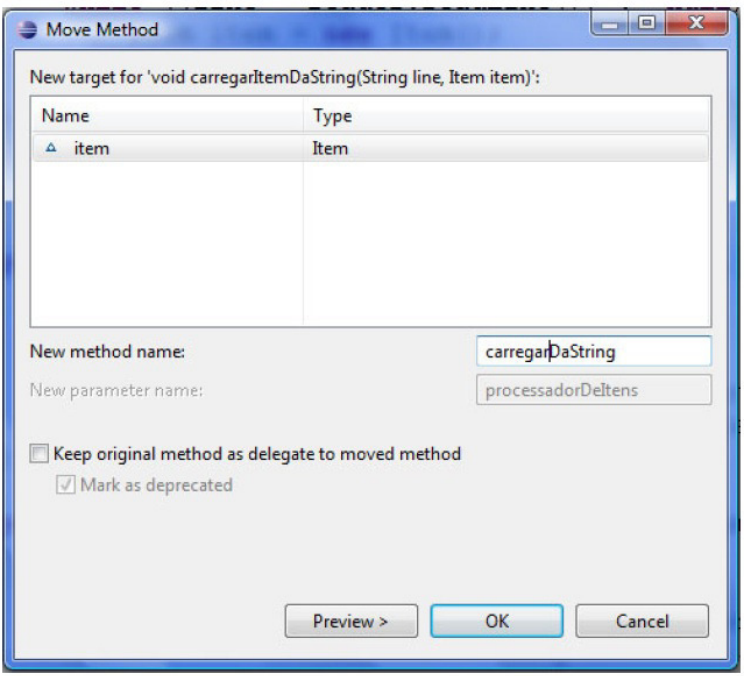


Figura 3.4: Janela pop-up da refatoração Move Method

Caso na sua versão do Eclipse o método `carregarItemDaString()` da classe `ProcessadorDeItens` não for eliminado, basta clicar sobre ele e no menu `Refactor`, e escolher a refatoração `Inline`. Com esta refatoração em todos os lugares em que este método for utilizado, será substituída a chamada do método pela sua implementação (no caso, a chamada ao novo método da classe `Item`).

Com a refatoração terminada, é uma boa hora para se rodar os testes de unidade para verificar se tudo foi feito corretamente. Se tudo correu como o esperado, o novo código das classes `Item` e `ProcessadorDeItens` será os que estão, respectivamente, nas duas próximas listagens.

Listagem 3.4 - Classe `ProcessadorDeItens` depois da primeira refatoração:

```
public class ProcessadorDeItens {

    public List<Item> getItens(InputStream input)
                throws IOException{
        List<Item> listItens = new ArrayList<Item>();
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(input));
        String line = null;
        while((line = reader.readLine()) != null){
            Item item = new Item();
            item.carregarDaString(line);
            listItens.add(item);
        }
            return listItens;
    }
    public double getValorTotalporTipo(InputStream input,
                String tipo) throws IOException{
        double valorTotal = 0;
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(input));
        String line = null;
        while((line = reader.readLine()) != null){
            String[] elementos = line.split(";");
            if(tipo.equals(elementos[1]))
                valorTotal += Double.parseDouble(elementos[3])
                    *Integer.parseInt(elementos[2]);
            }
            return valorTotal;
        }
    }
}
```

Listagem 3.5 - Classe `Item` depois da primeira refatoração:

```
public class Item {
```

```
    private String codigo;
    private String tipo;
    private int quantidade;
    private double valorUnitario;

    //métodos getters e setters omitidos

    public void carregarDaString(String line) {
        String[] elementos = line.split(";");
        setCodigo(elementos[0]);
        setTipo(elementos[1]);
        setQuantidade(Integer.parseInt(elementos[2]));
        setValorUnitario(Double.parseDouble(elementos[3]));
    }
}
```

Deixando o código mais claro

No método `getValorTotalporTipo()` da classe `ProcessadorDeItens`, os valores retirados de cada String que representa um item são usados diretamente, de forma que fica difícil entender o significado de um cálculo como o da linha a seguir:

```
valorTotal += Double.parseDouble(elementos[3])
    *Integer.parseInt(elementos[2]);
```

Uma pessoa que ler esse código pela primeira vez não vai rapidamente compreender que o cálculo está multiplicando o valor unitário de cada item pela quantidade. Uma forma de contornar o problema seria colocar um comentário para explicar o que está acontecendo. Mas para quê fazer isso se o código pode falar por si próprio?

Dessa forma, para deixar o código mais claro, será utilizado o método que acabamos de extrair e mover para a classe `Item`. Antes de fazermos os cálculos, o método `carregarDaString()` extrairá os dados da String e serão usados os métodos de acesso da classe `Item` para a realização dos cálculos. O novo código da função está representado a seguir:

```
public double getValorTotalporTipo(InputStream input,
```

```
            String tipo) throws IOException{
    double valorTotal = 0;
    BufferedReader reader =
                new BufferedReader(new InputStreamReader(input));
    String line = null;
    while((line = reader.readLine()) != null){
        Item item = new Item();
        item.carregarDaString(line);
        if(tipo.equals(item.getTipo()))
            valorTotal += item.getValorUnitario()
                *item.getQuantidade();
        }
        return valorTotal;
}
```

Apesar do número de linhas de código ter aumentado, o código agora é bem mais claro. Nem será necessário comentários para comunicar o que está acontecendo.

Porém, antes de avançar para o próximo passo da refatoração, existe algo que ainda não "cheira bem" nesse código: o fato de o método estar calculando o valor total do item parece ser um tanto invejoso. Assim, faz muito mais sentido esse cálculo ser responsabilidade da própria classe `Item`. Desta forma, o seguinte método foi criado na classe `Item`:

```
public double getValorTotal(){
    return getValorUnitario()*getQuantidade();
}
```

Além de substituir o cálculo no método `getValorTotalPorTipo()`, para evitar confusões, também será alterado o nome da variável local `valorTotal` para `valorTotalPorTipo`. Colocando o cursor em cima da variável e selecionando no menu `Refactor` e a opção `Rename`, o IDE permite no próprio código a alteração do nome. E

m versões anteriores do Eclipse, uma janela era aberta com um campo para se colocar o novo nome da variável. Depois de confirmada a refatoração, todas as referências àquela variável serão

alteradas.

Por mais que pareça não fazer diferença, nomes explicativos para variáveis, métodos e classes são essenciais para a clareza do código. Experimente utilizar apenas nome de variáveis monossílabas para ver como isso aumenta a complexidade do código absurdamente. Se um nome não parece bom, não pense duas vezes, troque-o imediatamente!

Depois de feita a refatoração, é mais uma boa hora para serem executados os testes e para verificar se o comportamento não foi alterado. A seguir, pode ser visto como ficou o código do novo método `getValorTotalPorTipo()`:

```
public double getValorTotalPorTipo(InputStream input,
            String tipo) throws IOException{
    double valorTotalporTipo = 0;
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(input));
    String line = null;
    while((line = reader.readLine()) != null){
        Item item = new Item();
        item.carregarDaString(line);
        if(tipo.equals(item.getTipo()))
            valorTotalporTipo += item.getValorTotal();
        }
    }
    return valorTotalporTipo;
}
```

Eliminando duplicação de código

O último "mau cheiro" que será eliminado desse código é a duplicação de código existente entre os dois métodos da classe `ProcessadorDeItens`. Observando bem os dois métodos, é possível facilmente observar que a funcionalidade é bem parecida. Ambos lêem da classe `InputStream` as Strings relativas aos itens e decodificam em objetos do tipo `Item`.

A diferença é que um método adiciona os objetos em uma lista,

e o outro os utiliza para calcular o somatório dos valores totais de itens que possuem um determinado tipo. O que poderia ser feito para eliminar esta duplicação de código é, no método `getValorTotalporTipo()`, usar o método `getItens()` para recuperar as listas de objetos do tipo `Item` para que os cálculos possam ser feitos depois.

Infelizmente, refatorações como esta não existe IDE que faça, pois são muito específicas do domínio. Dessa forma, será preciso realizá-la de forma manual mesmo. Felizmente, existe uma suíte de testes de unidade para nos mostrar que o comportamento verificado pelos testes não foi alterado. O código final das classes `ProcessadorDeItens` e `Item` podem ser vistos nas duas próximas listagens.

Listagem 3.6 - Classe `ProcessadorDeItens` depois de todas as refatorações:

```
public class ProcessadorDeItens {

    public List<Item> getItens(InputStream input)
                throws IOException{
        List<Item> listItens = new ArrayList<Item>();
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(input));
        String line = null;
        while((line = reader.readLine()) != null){
            Item item = new Item();
            item.carregarDaString(line);
            listItens.add(item);
        }
        return listItens;
    }

    public double getValorTotalporTipo(InputStream input,
                String tipo) throws IOException{
        double valorTotalporTipo = 0;
        List<Item> listItens = getItens(input);
        for(Item item : listItens)
            if(tipo.equals(item.getTipo()))
                valorTotalporTipo += item.getValorTotal();
            return valorTotalporTipo;
```

```
        }
}
```

Listagem 3.7 - Classe `Item` depois de todas as refatorações:

```
public class Item {

    private String codigo;
    private String tipo;
    private int quantidade;
    private double valorUnitario;

    //métodos getters e setters omitidos

    public void carregarDaString(String line) {
        String[] elementos = line.split(";");
        setCodigo(elementos[0]);
        setTipo(elementos[1]);
        setQuantidade(Integer.parseInt(elementos[2]));
        setValorUnitario(Double.parseDouble(elementos[3]));
    }
    public double getValorTotal(){
        return getValorUnitario()*getQuantidade();
    }
}
```

Considerações finais

Ao ser observado o código inicial e o final das classes, é possível perceber que, depois da refatoração, realmente houve uma grande melhoria em termos de clareza do código. Em muitos dos exemplos de refatoração, existem coisas horrendas no código inicial!

Neste exemplo, procurei colocar um código de certa forma razoável para que fosse visto que mesmo um código simples e aparentemente sem problemas pode ser melhorado de forma a aumentar sua clareza e manutenibilidade. Se for alterado, por exemplo, o formato de como os dados vêm para serem processados, as mudanças no código seriam pontuais e não espalhadas por diversas funções.

Do ponto de vista de desempenho, o novo código acaba sendo

inferior ao código original no método em que o valor total por tipo de item é calculado. Isto ocorre, pois, em vez de haver apenas um laço para o cálculo, é utilizado um para a construção de uma lista de itens e outro para o somatório dos valores.

Gostaria de deixar como exercício para os leitores a refatoração da classe `ProcessadorDeItens` para transformá-la em uma classe que representasse um conjunto de itens e, em seguida, refatorar o código de forma a se obter um desempenho compatível com o do método original.

3.5 CUIDADOS PARA APLICAR AS REFATORAÇÕES

Para efetuar qualquer refatoração, mesmo com a utilização de uma ferramenta, é recomendável que se tenha cuidado e o apoio de uma suíte de testes de unidade, para garantir que o comportamento da aplicação não foi alterado. Esta seção vai mostrar um exemplo de uma extração de método feita com ferramenta que não mantém o comportamento original do código.

Esse exemplo refatorará o método apresentado na próxima listagem. Ele recebe um array com linhas e um documento, e insere dentro, nas páginas deste documento, tabelas contendo estas linhas. As classes `Linha` e `Documento` não serão apresentadas, pois o importante é a lógica do código, e não o comportamento das classes envolvidas no exemplo.

Listagem 3.8 - Código do método `geraTabelas()`:

```
public void geraTabelas(Linha[] linhas, Documento documento) {
    Tabela tabela = null;
    tabela = new Tabela();
    tabela.inserirCabecalho();
    tabela.inicializaColunas();
    for(Linha linha : linhas){
        tabela.inserirLinha(linha);
```

```
        if(tabela.isTamanhoPagina(documento)){
            documento.inserePagina(tabela);
            tabela = new Tabela();
            tabela.inserirCabecalho();
            tabela.inicializaColunas();
        }
    }
}
```

Ao observar o código, um desenvolvedor vê o código duplicado (em destaque na listagem) e decide extrair o método usando as funcionalidades de seu IDE. O código fica conforme mostrado na listagem a seguir.

Listagem 3.9 - Código do método `geraTabelas()` refatorado:

```
public void geraTabelas(Linhas[] linhas, Documento documento) {
    Tabela tabela = null;
    inicializaTabela(tabela);
    for(Linha linha : linhas){
        tabela.inserirLinha(linha);
        if(tabela.isTamanhoPagina(documento)){
            documento.inserePagina(tabela);
            inicializaTabela(tabela);
        }
    }
}
private void inicializaTabela(Tabela tabela){
    tabela = new Tabela();
    tabela.inserirCabecalho();
    tabela.inicializaColunas();
}
```

Ao executar o código, o desenvolvedor se dá conta de que o código que estava funcionando, de repente, parou de funcionar. Ao observar o código com mais cuidado, ele observa que, ao passar a variável `tabela` como parâmetro, é passada uma referência para o objeto que está no método `geraTabelas()`.

Essa referência, a princípio, aponta para o mesmo objeto do método que o chamou, como mostrado na figura a seguir. No momento em que se chama o construtor, a variável `tabela` do

método `inicializaTabela()` passa a apontar para outra instância, mas a variável do método `geraTabelas()` continua a mesma, conforme a figura *A variável que apontava para o objeto agora aponta para outro*.

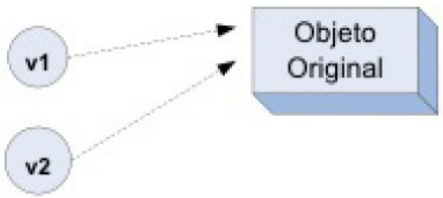


Figura 3.5: As duas variáveis apontam para o mesmo objeto

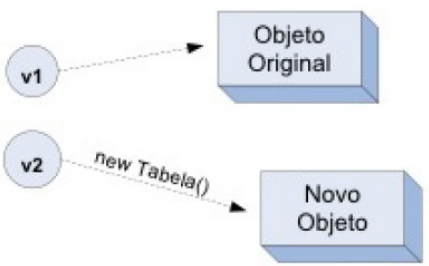


Figura 3.6: A variável que apontava para o objeto agora aponta para outro

Para a refatoração do exemplo funcionar corretamente, deveríamos fazer com que o novo método retornasse uma referência para o objeto criado e, no método principal, que ela fosse atribuída à variável que se deseja. A listagem a seguir apresenta o código refatorado correto.

Listagem 3.10 - Código do método `geraTabelas()` refatorado de forma correta:

```
public void geraTabelas(Linhas[] linhas, Documento documento) {
    Tabela tabela = null;
    tabela = inicializaTabela();
```

```
    for(Linha linha : linhas){
        tabela.inserirLinha(linha);
        if(tabela.isTamanhoPagina(documento)){
            documento.inserePagina(tabela);
            tabela = inicializaTabela();
        }
    }
}
private Tabela inicializaTabela(){
    Tabela tabela = new Tabela();
    tabela.inserirCabecalho();
    tabela.inicializaColunas();
    return tabela;
}
```

Os erros que os IDEs cometem em refatorações são mais comuns do que se imagina. O artigo *Sound and Extensible Renaming for Java*, apresentado no OOPSLA de 2008 (vide *Referências*), mostra diversos erros que os IDEs cometem nas refatorações `Rename` e propõe uma solução para esses tipos de refatoração.

Um dos exemplos mostrados no artigo está apresentado nas duas próximas listagens. Considere que, no IDE Eclipse, renomeia-se, na primeira listagem, a variável `y` para `x`. O resultado é um código com comportamento diferente do original, apresentado na listagem seguinte. A variável `y` é renomeada para `x` na instrução `System.out.println(y);`, o que faz com que na verdade seja utilizada outra variável de instância da `inner class` que também se chama `x`.

Listagem 3.11 - Código exemplo de uma classe complexa:

```
class A {
    public static void main(String[] args) {
        final int y = 23;
        new Thread() {
            int x = 42;
            void run() {
                System.out.println(y);
            }
        }.start();
```

```
    }
}
```

Listagem 3.12 - Código exemplo refatorado erradamente:

```
class A {
    public static void main(String[] args) {
        final int x = 23;
        new Thread() {
            int x = 42;
            void run() {
                System.out.println(x);
            }
        }.start();
    }
}
```

A lição que deve ser tirada desses exemplos é que, por mais que as ferramentas facilitem a nossa vida com funcionalidades de refatoração de código, a refatoração deve ser feita a partir de uma análise crítica do código a ser refatorado. Não é porque uma refatoração é automatizada em um IDE que sua aplicação sempre vai manter o comportamento original do código.

De preferência, tenha sempre uma suíte de testes que verifique o comportamento de suas classes antes de sair refatorando todo o código.

3.6 CONSIDERAÇÕES FINAIS

A refatoração é uma prática fundamental para um desenvolvimento iterativo, em que, a cada iteração, o código deve ser modificado e novas funcionalidades são constantemente adicionadas. O uso de padrões de projeto é de grande importância para a modelagem macro de uma aplicação, porém a refatoração vai muito mais fundo.

A refatoração, além de melhorar a modelagem da aplicação com a eliminação de código duplicado, mexe em detalhes menores, como

em lógicas condicionais complicadas e até mesmo na nomenclatura dos elementos do código. O aumento na qualidade final do código é significativo!

A aplicação das refatorações deve ser feita com bastante cuidado, mesmo com a utilização de IDEs que automatizem parte do processo. A existência de testes de unidade dá ao processo de refatoração controle e segurança.

Assim como para aplicar padrões no projeto de um software, aprender as diferentes formas de se refatorar um código exige bastante estudo. Várias refatorações acabam tendo como resultado a aplicação de um padrão (ver livro *Refactoring to Patterns* nas *Referências*). Dessa forma, aprender a refatorar um código é aprender a trabalhar com sua modelagem de forma contínua, realizando mudanças passo a passo controladamante, sempre visando a melhora da qualidade e da clareza do código-fonte.

3.7 REFERÊNCIAS

AMBLER, Scott W.; SADALAGE, Pramodkumar J. *Refactoring Databases:* Evolutionary Database Design. Addison-Wesley Professional, 2006.

FOWLER, Martin. *Refatoração:* Aperfeiçoando o Projeto de Código Existente. Bookman, 2004.

KERIEVSKY, Joshua. *Refactoring to Patterns.* Addison-Wesley, 2004.

SCHÄFER, Max; EKMAN, Torbjörn; MOOR, Oege de. Sound and Extensible Renaming for Java. *OOPSLA - Object-oriented Programming, Systems, Languages, and Applications,* 2008.

PREVITALI, Susanne Cech; GROSS, Thomas R. Annotations

for Seamless Aspect-Based Software Evolution. *RAM-SE'08 - 5thECOOPWorkshop on Reflection, AOP and Meta-Data for Software Evolution,* 2008.

Capítulo 4

OS PRINCÍPIOS DA MODELAGEM ÁGIL

"O que precisamos modelar? Quanto precisamos modelar? Para que modelar? Os valores, princípios e práticas da Modelagem Ágil (Agile Modeling), um método defendido por Scott Ambler, podem auxiliar sua equipe nessa importante tarefa do desenvolvimento de software. Artefatos "bonitos" feitos por ferramentas caras nem sempre são os melhores para melhorar a comunicação ou melhorar a qualidade do seu código. Modelar o software em grupo e com a participação dos usuários, utilizando, por exemplo, um rascunho, é uma das maneiras interessantes de se conseguir qualidade no seu design." — por Rodrigo Yoshima

Uma das dúvidas mais comuns nas equipes de desenvolvimento é: "Como conseguir efetivamente capturar as necessidades dos usuários e comunicá-las sem ruído para dentro da minha equipe de desenvolvimento?". Para responder a primeira parte dessa pergunta é fácil: é impossível capturar as reais necessidades dos usuários.

Uma das questões mais defendidas neste capítulo é o desenvolvimento iterativo, em que entregamos software funcionando constantemente para observar se as necessidades foram atendidas. Requisitos não são "verdade" até que se prove juntamente com o usuário através de software funcionando.

Já com relação à comunicação da equipe, podemos aplicar

práticas e artefatos para que as pessoas envolvidas no projeto colaborem para que a solução atenda ao negócio e, ao mesmo tempo, boas práticas de programação sejam respeitadas.

Ao iniciar esta leitura, vocês podem pensar que se trata de assunto de "Analistas", porém, não é interessante se ater a papéis. Uma equipe de desenvolvimento de software é mais eficiente se todos são capazes de fazer todas as atividades. Considere cada membro da sua equipe como "analista programador-tester". Observo que separar a equipe entre analistas, programadores e testers é uma má interpretação do mercado com relação ao RUP.

Na sua equipe, algumas pessoas terão qualidades diferenciadas, mas geralmente elas não são relacionadas com as "etapas" da construção do software, e sim com outros fatores menos deterministas. Você pode ter uma pessoa com excelente desenvoltura para conversar com os usuários, outra pessoa consegue codificar com uma rapidez acima do normal e com qualidade! Outra pode ter um conhecimento abrangente para dar soluções a problemas complexos.

As pessoas possuem habilidades diferenciadas, porém, com relação às etapas de desenvolvimento, é preferível que todas tenham capacidade de exercê-las. Isso favorece uma maior produtividade.

Durante o capítulo, vamos apresentar práticas ágeis para melhorar a comunicação, a produtividade e gerar a documentação na medida certa. Tudo isso é fundamentado nos três critérios de sucesso da sua aplicação:

1. A solução atende ao negócio (qualidade externa).
2. O software é fácil de manter e evoluir (qualidade interna).
3. Menor custo e prazo possível (qualidade do projeto).

Tudo o que você faz no seu processo de desenvolvimento deve

ter propósitos claros e substanciais para obter esses critérios. Muitas vezes, determinadas práticas são soluções que procuram por um problema. Outras são declaradamente contra os critérios anteriores. Avalie seu processo de desenvolvimento ou a maneira como a sua equipe se comporta, e veja se o seu dia a dia é focado em obter os critérios apresentados.

4.1 PARA O DESENVOLVIMENTO DE SOFTWARE, NÃO EXISTE SEPARAÇÃO ENTRE DESIGN E CONSTRUÇÃO

É comum as pessoas envolvidas nas tarefas gerenciais de um projeto de software acharem que a Engenharia de Sistemas é próxima da Engenharia Civil ou Engenharia Mecânica. Nas engenharias tradicionais, existe separação entre design e construção, na Engenharia de Software não existe esta separação.

Quando um edifício é construído, o gerenciamento de projeto tradicional faz sentido, pois para esse tipo de produto (o prédio), é necessário ter uma planta detalhada para poder iniciar a construção. Nesse projeto, quem é responsável pela atividade de design é um arquiteto ou engenheiro que possui qualificações intelectuais distintas de quem efetuará o projeto (a construção).

O trabalho de construção será desempenhado pelo pedreiro, que não necessita ter o conhecimento total da engenharia para fazer o seu trabalho. Essa divisão acontece por conta das características totalmente diferentes entre os dois trabalhos: um é altamente intelectual, o outro é mais braçal. Um engenheiro pode ter um custo alto. O pedreiro tem um custo muito menor. Um engenheiro pode gerar subsídios para o trabalho de até 100 pedreiros.

Dado a natureza do projeto civil, é lógico que exista essa divisão de trabalho entre design e construção. Nesse cenário, não faz

sentido as pessoas do projeto serem "engenheiro-pedreiro", como pode ser na engenharia de software.

Quando falamos em software, temos basicamente quatro atividades principais:

1. Compreender o que o usuário quer.
2. Definir como os elementos de software resolverão as necessidades do usuário.
3. Escrever esses elementos de software e integrá-los.
4. Testar os elementos e homologá-los com o usuário.

Antes de tudo, ressalto que essas atividades ocorrem milhares de vezes dentro de um projeto e nem sempre na ordem que o quadro sugere (não faço uma apologia ao desenvolvimento em cascata!). Porém, olhando a lista apresentada, a diferença que temos com relação à engenharia civil é que não existem diferenças bruscas no nível intelectual necessário para desempenhar essas atividades.

A atividade de escrever os elementos de software não é intelectualmente inferior à atividade de compreender o que o usuário quer, ou definir a colaboração entre elementos de software. São atividades diferentes, mas não tão diferentes como o trabalho do engenheiro e do pedreiro. O erro que muitas empresas cometem é achar que analistas de sistemas são como engenheiros civis, e programadores são como pedreiros. Isso é um completo absurdo!

Isso também é uma grande injustiça com os programadores. Programar é a atividade mais intensa, a que mais requer atenção, a mais sujeita a estresse e a que mais emerge riscos no desenvolvimento de software. Processadores de texto e ferramentas UML aceitam qualquer besteira que um analista escreve ou modela.

Já o compilador é implacável: ou você codifica direito, ou a aplicação simplesmente não funcionará. Ou você integra

corretamente os componentes, ou eles simplesmente não vão funcionar juntos. Os erros de programação são os mais visíveis do projeto, tanto que até tem nome: **bug**.

Olhando por esse lado, a postura lógica seria as empresas investirem mais, e remunerarem melhor, os programadores. Entretanto, é comum analistas serem mais bem remunerados do que programadores, mesmo que essa divisão dos papéis não faça sentido algum.

4.2 DESENVOLVIMENTO DE SOFTWARE: HUMANAS OU EXATAS?

Quando pegamos um projeto e focamos nos três critérios de sucesso citados na introdução, temos dificuldade em saber quais qualidades nós e nossa equipe devem ter para balancear essas três variáveis. Basicamente, o que queremos com a modelagem do sistema é satisfazer ou "balancear" as necessidades dos usuários, garantindo a qualidade interna do software e respeitando as restrições de custo e prazo do projeto.

Determinados sistemas são críticos em funcionalidades aos usuários. Capturar essas necessidades é uma atividade altamente social que requer muita comunicação clara e uma grande proximidade com os *stakeholders*. Esse tipo de sistema requer uma maior atenção na qualidade externa, e a concentração maior da modelagem é virada para o lado dos usuários.

Já outros sistemas prezam mais pela qualidade interna. Quando temos de lidar com muita complexidade técnica, que geralmente ocorre em sistemas que lidam com milhares de gigabytes, milhares de usuários, milhares de transações e processamento distribuído, um pequeno desvio de disciplina pode levar o sistema ao caos. Nesse caso, a modelagem é voltada para o aspecto técnico, em que é

obrigatório conhecer a estrutura interna dos componentes, as dependências, algoritmos complexos etc.

Agora, tenha em mente que os critérios de sucesso precisam ser equilibrados. Todo projeto possui restrições de custo e prazo. Não temos todo tempo e dinheiro do mundo para fazer o software. Sua missão como um bom "modelador" é controlar as expectativas do cliente e sempre avaliar quais problemas realmente são **problemas**.

Fazer a solução atender ao negócio é o ponto de parada. Saber fazer o usuário separar aquilo que é desejável daquilo que é imprescindível também é uma habilidade importante para a equipe do projeto.

Uma das coisas que as metodologias ágeis trouxeram à tona é a importante participação dos usuários ou clientes para auxiliar na modelagem do sistema, seja elaborando um mapa mental, ou rascunhando uma tela, ou testando um release que acabou "de sair do forno". Os usuários, ou clientes ou stakeholders possuem um papel importantíssimo para gerar as demandas corretas e avaliar a qualidade dos trabalhos executados. Isso requer uma comunicação rica entre o cliente e todos os envolvidos no projeto.

4.3 A COMUNICAÇÃO RICA DO "PAPEL SOBRE A MESA"

Alistair Cockburn é um dos autores do Manifesto Ágil de 2001, e também é um dos nomes que mais defende uma comunicação aberta, intensa e honesta dentro dos projetos de desenvolvimento de software. Cockburn foi o primeiro autor a defender que até a disposição física da sala onde os membros da equipe trabalham pode acarretar em problemas para os projetos.

Em um de seus artigos, Cockburn apresentou um resultado de

seu estudo sobre a eficiência de artefatos e meios de comunicação.

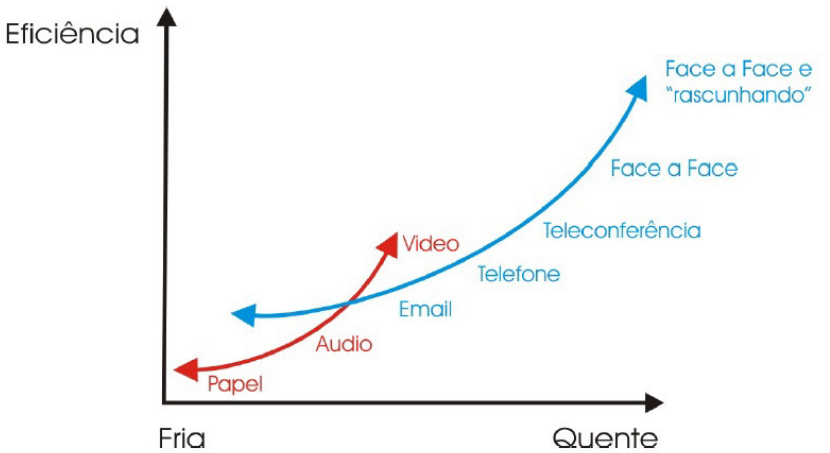


Figura 4.1: Valor do canal de comunicação

O gráfico da figura nos mostra a "Eficiência" versus a "Temperatura" de meios de documentação. A linha vermelha mostra as mídias de documentação de um projeto. Como podemos ver, documentos em papel são os menos eficientes e os mais "frios" para armazenar e demonstrar ideias.

Ainda na linha vermelha, conteúdo em vídeos (imagem e voz) pode ser a opção mais eficiente e amigável para o registro de informações do projeto. Atualmente, tenho usado apresentações com imagem e som (screencasts) para demonstrar arquiteturas de aplicações. Veja um exemplo nas referências e note como um screencast pode poupar a escrita de um documento de arquitetura de 10 a 20 páginas, e ainda melhorar o entendimento.

Ainda sobre o gráfico da figura, a linha azul mostra as mídias de comunicação, os meios que duas pessoas, ou um grupo de pessoas, utilizam para evoluir ideias para o bom desenvolvimento do projeto. Desenvolvimento de software é extremamente dependente de

comunicação. Essa comunicação é crítica tanto no relacionamento do cliente com a equipe como também a comunicação interna da própria equipe.

A figura nos diz que o pior meio de comunicação e documentação são artefatos sendo trocados via e-mail e, infelizmente, muitos projetos usam somente esse meio pobre de comunicação. Esta é a maneira menos eficiente e mais fria de trocar ideias.

Já um meio rico de trocar ideias é a comunicação face a face, preferencialmente utilizando uma maneira compartilhada de demonstrar graficamente as ideias, como um papel de rascunho ou um quadro branco. A ideia é bem simples: junte as pessoas em uma sala e faça-as colaborarem entre si, trocando ideias e sentimentos com o auxílio de uma mídia gráfica compartilhada (um quadro branco, um flip-chart ou papéis de rascunho com lápis são excelentes).

A eficiência dessa técnica é comprovada em Workshops de requisitos como cliente, ou em discussões arquiteturais dentro da equipe. O resultado dessas reuniões são rascunhos que podem ser feios na forma, mas são perfeitos para o propósito de modelagem do sistema.

4.4 RASCUNHOS: UMA FORMA DE MODELAGEM EFICAZ

Sempre que falamos em rascunhos, as pessoas, principalmente na nossa área, ficam com aquela impressão de coisas mal feitas ou informalidade em excesso. Apesar de não apoiar muito o uso de definições de dicionários, nesse caso, creio que seja interessante definir o que é rascunho.

> **RASCUNHO:** s. m., minuta; esboço; delineamento; trabalho prévio de redação em que se fazem emendas ou rasuras, antes de ser passado a limpo.

Essa definição nos deixa claro que rascunho não precisa ser necessariamente mal feito e nem informal. A palavra que mais gostei dessa definição é **delineamento**, é experimentar no sentido de testar uma maneira mais simples de algo que pode ser refinado em tempo oportuno.

Por alguma razão, profissionais de informática possuem um vício por "coisas" digitais. Nós queremos tudo em arquivos digitais que estão em algum lugar do HD.

Entretanto, a indústria sobreviveu várias décadas sem o computador, e várias áreas de atuação vivem em paz sem o uso das máquinas. Na área de desenvolvimento de software, nós podemos utilizar mídias que não são digitais. Sempre que penso em rascunhos, lembro da obra do brasileiro Oscar Niemeyer (http://www.niemeyer.org.br/), um dos maiores nomes da Arquitetura Moderna. Oscar Niemeyer obteve muito sucesso rascunhando projetos de maneira genial.

Figura 4.2: Rascunhos do Congresso Nacional em Brasília por Oscar Niemeyer

Olhando a figura, vemos que o rascunho é uma ferramenta interessante para observar o futuro. Talvez a obra do Congresso Nacional tenha demorado anos, porém, os rascunhos que traduziam a ideia original do cliente ficaram prontos em poucos instantes. O interessante é que a obra final ficou muito parecida com o que se queria de fato.

No desenvolvimento de software, nós podemos aplicar técnicas parecidas, mas em escala menor, aplicando iteratividade (lembre-se de que são engenharias diferentes!). Fazer o cliente nos auxiliar na modelagem é uma das maneiras de fornecer uma visão daquilo que será o software no futuro, mesmo que usando artefatos leves, como um rascunho em uma página A4.

4.5 RASCUNHOS PARA OBTER QUALIDADE EXTERNA NO SOFTWARE

A figura seguinte tem um rascunho desenhado junto com o cliente e com a equipe técnica do ERP (outro stakeholder) que foi criado na reunião de concepção ágil do projeto FertFórtil.

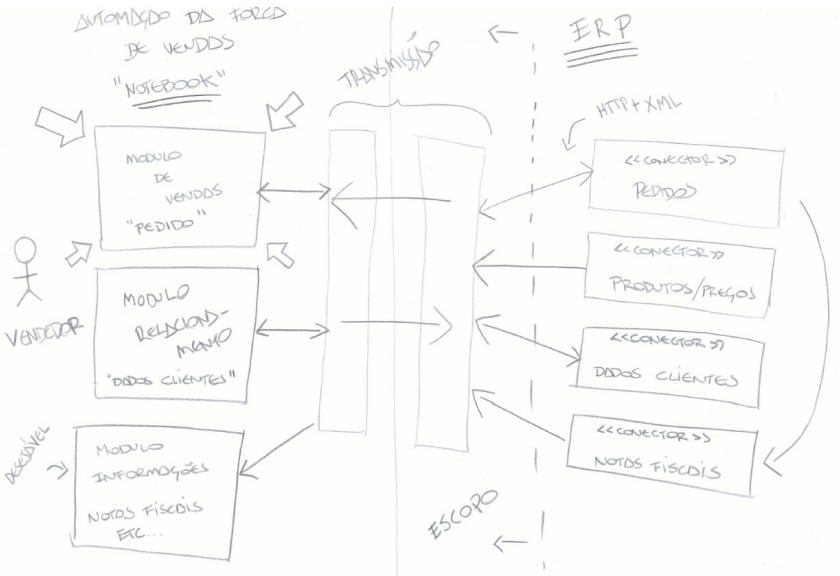


Figura 4.3: Rascunhos iniciais do projeto FertFórtil: integração e separação de módulos

Esse projeto-exemplo foi apresentado no artigo *Entregue Software Funcionando! Gerenciamento de Projeto Ágil* (MundoJava, número 26). Só para situar o projeto, veja a lista de funcionalidades na figura a seguir:

ID	Funcionalidade	Iteração	Pontos
1	Entrada do Pedido (simples)	1	8
2	Interface de Transmissão de Dados Vendedor<->ERP	2	2
3	Integração do Pedido (Vendedor -> ERP)	2	5
4	Entrada do Pedido: Cálculo de Preços e Impostos	3	5
5	Integração de Dados de Preços (ERP -> Vendedor)	3	2
6	Integração de Dados de Impostos (ERP -> Vendedor)	3	2
6	Entrada do Pedido: Vendas Parceladas	4	3
7	Entrada do Pedido: Limite de Crédito	4	3
8	Consulta Informações do Cliente	5	3
9	Integração de Clientes (ERP -> Vendedor)	5	5
10	Consulta Informações de Produtos	6	5
11	Integração de Produtos (ERP -> Vendedor)	6	1
12	Consulta Notas Fiscais e Integração (ERP -> Vendedor)	7	5

Figura 4.4: Product Backlog inicial da aplicação SFA FertFórtil

A aplicação SFA FertFórtil basicamente é a emissão de pedidos pelos vendedores através de um cliente remoto desconectado (notebook). Atualmente, os vendedores passam o pedido via fax para a central. Com o novo sistema, eles usarão um notebook para capturar e transmitir o pedido.

Esse pedido será integrado no ERP da FertFórtil. "Cliente remoto desconectado" significa que a transação não é online, isto é, o vendedor pode capturar pedidos dos clientes o dia todo e transmitir todos eles em lote quando chega em casa, utilizando a internet. Os pedidos são acumulados localmente no notebook de alguma forma.

Essa lista de funcionalidades da figura é o fruto da concepção do software. Esse trabalho de concepção é um esforço inicial em que tentamos pegar "a ideia" geral do problema que o software pretende resolver. É o primeiro contato que você tem com o cliente ou com os futuros usuários.

A concepção é uma atividade altamente social, em que habilidades, como a comunicação, a curiosidade, o interesse e o foco, entre outras capacidades interpessoais e não técnicas, são postas à prova. Mas mais importante para este capítulo é que, nas reuniões iniciais de concepção, nós modelamos muito. Nessas reuniões, surgem os modelos mais importantes para a garantia da aderência da aplicação às necessidades principais do cliente.

Na figura *Rascunhos iniciais do projeto FertFórtil*, temos um resumo geral da aplicação. Como já dissemos, todas as pessoas interessadas nesse diagrama contribuíram para a elaboração dele.

O pessoal do ERP mencionou que a integração com eles ocorre através de conectores HTTP em XML (Web services). Essa é uma informação importante para avaliarmos a integração, e isso consta de uma maneira bem simples no diagrama nesse momento.

Detalhes sobre essa integração serão levantados mais adiante (mais especificamente na iteração 2, vide figura *Product Backlog inicial da aplicação SFA FertFórtil*).

Outras informações importantes levantadas nesse diagrama são: a divisão de módulos da aplicação, a descoberta da necessidade de um módulo de interface de transmissão e o delineamento do escopo do sistema. Sobre o escopo, a linha pontilhada vertical deixa claro que nossa função é só preparar para que os dados entrem consistentes no ERP, porém, a partir disso, toda responsabilidade passa a ser do ERP.

4.6 APROFUNDAR EM DETALHES ANTECIPADAMENTE É RUIM!

Quando falamos em modelagem, seja de projetos de engenharia, de uma embalagem, de uma casa ou de software, é importante ter em mente qual é o objetivo do modelo que trabalhamos no momento. Uma das grandes dúvidas das equipes de projeto é quão longe devemos nos aprofundar em detalhes.

Para responder a essa dúvida, vamos analisar mais uma vez a figura *Rascunhos iniciais do projeto FertFórtil*. Mas, dessa vez, tente responder às perguntas a seguir.

1. Qual é o momento do projeto?
2. Qual é a dúvida que tenho?
3. Quem poderia modelar isso junto comigo para obter as respostas?
4. Qual é o modelo certo?

Respondendo às perguntas focando nessa figura: estamos no primeiro momento como cliente e querendo saber o que é o projeto, descobrindo os objetivos, o escopo, as funcionalidades, os riscos.

Logicamente, quem nos ajudará a obter essas respostas é o cliente. E para compreender as implicações da integração com o ERP (um dos grandes riscos), os representantes desse recurso também estão presentes.

Tendo essas pessoas na sala, o ponto principal dessa reunião é ter uma compreensão de **alto nível** do sistema, e não todos os detalhes. Não é conveniente nessa reunião discutir as minúcias da integração com o ERP. Detalhes serão adicionados iterativamente e de maneira incremental durante todo o projeto.

Essa figura é um modelo de alto nível. Modelos são manifestações de decisões ou percepções. Nesse caso, é a manifestação da percepção da integração, dos módulos e do escopo inicial do projeto.

Um dos valores da modelagem ágil é "modelar comum propósito" e, muitas vezes, modelar com um propósito significa buscar os critérios de sucesso do projeto apresentados no começo do capítulo. Você modela para que o software atenda ao usuário (como fizemos na figura), ou para que o software tenha qualidade interna, ou para reduzir custo ou prazo do projeto.

É altamente questionável quando você modela só para cumprir "etapas" do processo, ou modela porque seu processo de desenvolvimento exige o modelo. Como já falei no artigo *UML não é Documentação* (MundoJava, número 19): você modela o que precisa ser modelado.

Figura 4.5: Protótipo da tela de pedidos no notebook

Uma das questões relevantes é: "Qual é o modelo certo?". É importante ressaltar que temos uma quantidade infinita de maneiras de podemos modelar. Uma prova disso é o protótipo da tela de pedidos da figura anterior.

Este protótipo foi elaborado na mesma reunião com o cliente, pois emissão de pedidos faz parte da primeira iteração do projeto (figura 4.4). Um esboço feito de forma rápida e conjunta com as partes interessadas é um meio muito eficiente de capturar requisitos e descobrir como atingir o objetivo de fazer um software com qualidade externa, que atenda às expectativas dos usuários.

4.7 NÃO VOLTE PARA CASA PARA ESCREVER DOCUMENTOS DE REQUISITOS!

A vantagem em se fazer com que os clientes e usuários participem da modelagem do sistema usando artefatos e ferramentas simples, como rascunhos em papel ou no quadro branco, é promover a colaboração e ganhar agilidade na captura de requisitos. Porém, algumas pessoas ou empresas sofrem de excesso de cerimônia, de forma a não reconhecer artefatos em papel como "válidos" no processo de desenvolvimento. Isso leva a uma burocratização desnecessária na captura de requisitos.

Muitas pessoas levantam requisitos em artefatos informais para depois transformá-los em casos de uso, especificações suplementares, protótipos, documentos de regras de negócio e outros quando "voltam para casa". Isto é, preenchem esses templates depois das reuniões, formalizando o levantamento.

Como houve uma transformação de artefatos, é típica uma revisão com o cliente para obter uma "aprovação". O problema é que essa aprovação nunca ocorre! Ao ver os artefatos transformados, o cliente "se lembra" de mais requisitos que são mais uma vez capturados em artefatos informais, e o ciclo se repete muitas vezes. Com isso, levam-se semanas para "aprovar os requisitos" e um tempo precioso é perdido.

A ideia de trabalhar com rascunhos é rapidamente transformá-

los em "software funcionando" (em um espaço de uma semana), de forma que da próxima vez que você encontrar o cliente, você entregará software a ser homologado, e não documentos a serem aprovados.

Não perca tempo aprovando documentos de requisitos depois que foram levantados, pois eles nunca serão aprovados, e mesmo que sejam, o cliente poderá mudar de ideia quando ver realmente o que ele quer de fato (o sistema!). Lembre-se: documentos de requisitos são abstrações! Isso também nos traz a definição:

> **Software funcionando** é o melhor artefato para levantamento de requisitos.

Como já defendi várias vezes, desenvolvimento iterativo é entrega constante de software pronto para os stakeholders. Dessa forma, os usuários olham o software e solicitam alterações no software e não em documentos. Software é a única coisa concreta que realmente validamos com os usuários.

Muitos artefatos podem ser usados para documentar requisitos. Os rascunhos em papel, um arquivo MP3 como conteúdo da reunião de levantamento, protótipos visuais, casos de uso, documentos em texto etc. Documentos de requisitos têm como objetivo registrar o que os usuários esperam da aplicação independente da solução técnica. Isso faz com que esses documentos sejam simples e rápidos de serem elaborados.

Você pode usar vários deles para registrar os anseios dos usuários (a prática "Utilize o artefato correto" do Agile Modeling). Mas mais importante é que esses artefatos sejam elaborados na reunião de levantamento, e não "em casa".

Se você quer modelar uma visão geral do sistema, use um modelo como o da figura 4.3. Se você quer modelar uma tela, use um rascunho como da figura 4.5. Se você quer documentar uma interação com o usuário, escreva um caso de uso ali mesmo na reunião. Saia da reunião com requisitos levantados e documentados (para falar a verdade, não existe separação entre esses dois). Depois disso, é fazer o software para cumprir o planejamento da iteração.

4.8 UTILIZANDO A MODELAGEM PARA OBTER QUALIDADE INTERNA

Até o presente momento, utilizamos a modelagem ágil para melhorar e documentar decisões focadas na qualidade externa da aplicação. Usamos uma modelagem conjunta com o cliente. A modelagem ágil também pode ser usada dentro da sua equipe de desenvolvimento.

A comunicação entre os membros da equipe segue o mesmo padrão do gráfico apresentado na figura 4.1. A comunicação face a face e com um rascunho (mídia compartilhada) é a maneira mais indicada para discutir e disseminar ideias dentro do time.

Uma das práticas que tenho aplicado em muitos projetos são reuniões para discussão de arquitetura. Quando temos uma equipe desenvolvendo software, as pessoas têm a tendência de focar nos seus próprios problemas sem discutir em grupo as questões. Para isso, marcar uma reunião especificamente para discutir essas pendências é saudável para o projeto.

Sempre que falamos no lado técnico do projeto. focamos em boas práticas de programação OO. Com isso, um sistema com boa qualidade interna possui basicamente as duas características de um bom design:

1. Alta coesão;
2. Baixo acoplamento.

É nessas duas “medidas” que você deve focar no design dos seus objetos. Nessas reuniões de arquitetura, juntar a equipe e a dinâmica não é muito diferente da reunião com os clientes. Usamos muito papel A4 e muitos desenhos no quadro branco.

Nessas reuniões, surgem modelos como os retratados nas figuras a seguir. Nesses encontros, nós “falamos” UML. UML é uma linguagem, ela serve para comunicação! Nesse cenário é que vemos o valor em ter uma notação padrão: o que um membro da equipe “fala” em um rascunho todos compreendem.

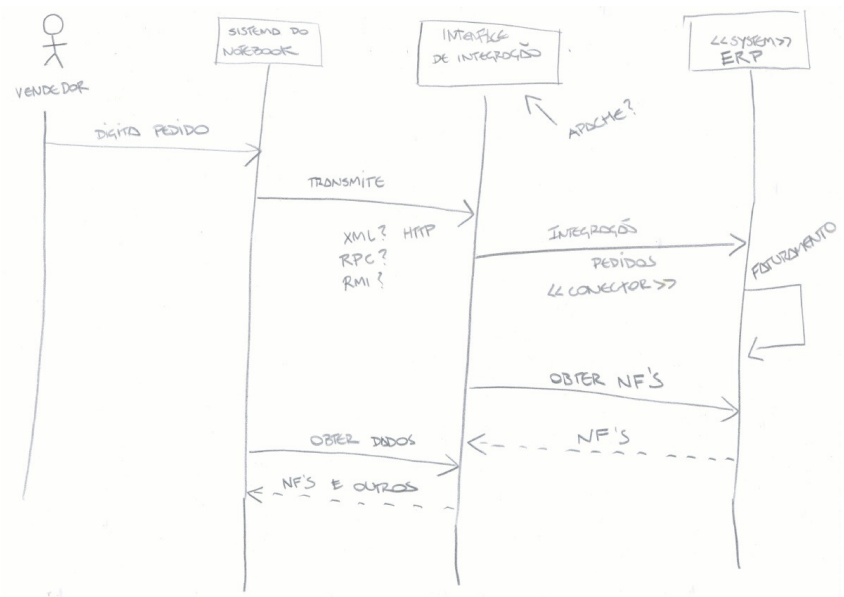


Figura 4.6: Rascunho de diagrama de sequência

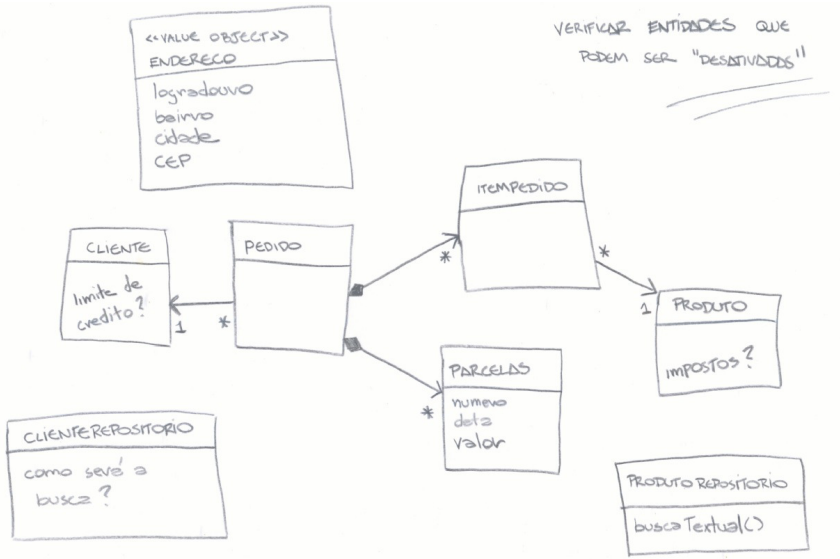


Figura 4.7: Domain model da aplicação

Ferramentas UML também são úteis (sim, eu também uso ferramentas UML!). Determinados modelos são importantes e devem ser mantidos como documentação permanente do projeto.

Muitos deles constam em um documento de arquitetura. Porém, modelos são naturalmente difíceis de manter. Se seu modelo está muito "profundo", a tendência é que fique mais defasado como código.

A modelagem ágil possui um princípio chamado "Travel Light". Não sei como traduzir isso de maneira efetiva, mas a ideia é "não viaje para muito longe com seu modelo". O princípio nos diz que manter modelos é difícil, e pode se tornar mais difícil se você tiver muitos modelos e muitos detalhes.

Talvez três ou quatro modelos que fornecem uma visão geral da análise, da arquitetura e do design são suficientes para melhorar a comunicação da equipe. É errado pensar que, quanto mais

documentação, melhor. A documentação precisa ser enxuta. Outros modelos podem ser simplesmente descartados depois que cumpriram seu papel, transformando-se em código.

4.9 CONCEITOS SEMPRE PROVADOS COM CÓDIGO

Uma das discordâncias que muitas pessoas têm contra metodologias ágeis é o foco no código. Porém, é no código que a alta coesão e baixo acoplamento oferecem benefícios.

Um dos princípios da Modelagem Ágil é: "Prove com Código". Isto é, os modelos são somente uma abstração daquilo que você está construindo de fato. Mas será que o modelo está correto? Será que ele é implementável? Para saber essa resposta, é necessário "provar o modelo com código".

É importante ressaltar que a atividade de modelagem visa obter uma qualidade do código e não do modelo. Nós modelamos para o código. Uma boa modelagem tem como objetivo obter um bom código, e não um bom modelo. A qualidade sempre deve focar o código e os importantes conceitos de alta coesão e baixo acoplamento. A "beleza" do modelo não é necessariamente qualidade.

Outro ponto é que você também pode modelar diretamente no código. Modelos em UML podem te ajudar bastante em uma conversa da reunião de arquitetura, ou para ter a visão geral de um módulo.

Você pode também aplicar Domain-Driven Design com UML, conforme demonstrado na figura 4.7. Por outro lado, você também pode modelar diretamente no código usando a prática da programação em pares do Extreme Programming, e também através

de TDD (Test-Driven Development).

Como já citei aqui, não existe separação entre design e construção na Engenharia de Software. No software, não precisamos necessariamente da "planta" para fazer o código. Olhando para as perguntas apresentadas, o código pode ser considerado um modelo. Modelagem é uma atividade de criação, e não de preenchimento de template de artefato.

4.10 CONCLUSÃO

Este capítulo demonstrou que artefatos simples podem ser usados para aplicar boas práticas de modelagem. Modelagem é uma atividade em grupo, na maioria das vezes, e tudo que fazemos deve ser focado em melhorar a qualidade interna, a qualidade externa ou a produtividade.

Modelar simplesmente para "ter o modelo" deve ser evitado! Muitas outras práticas e valores são importantes na *Agile Modeling*, e recomendo fortemente o estudo das referências listadas.

Mas o mais importante de tudo é focar nas atividades do desenvolvimento e não nos artefatos criados. Quando você levanta requisitos, a atividade de conversar com o usuário, obter informações e enriquecer o conhecimento são mais importantes do que os casos de uso, protótipos, modelos que surgirem dessa atividade.

O mesmo vale para uma modelagem de design. O importante é a criatividade para solucionar as dependências dos objetos, a separação dos conceitos, as mensagens trocadas. O modelo de design é só uma manifestação dessas decisões. A tomada dessas decisões é mais importante que a manifestação delas.

Esse alerta é importante, pois muitas pessoas confundem a

atividade “levantamento de requisitos” com “preenchimento dos templates de artefatos de requisitos”. E, pior ainda, muitas vezes esse preenchimento de artefatos é feito simplesmente para cumprir tarefas inúteis de um processo de desenvolvimento pesado, burocrático e que rouba a criatividade das pessoas.

> As atividades dentro do desenvolvimento de software são mais importantes do que os artefatos gerados.

Tome cuidado com processos de desenvolvimento que parecem organizados. Muitos deles não são organizados de fato. O que o cliente quer é o software, e não os documentos que abstraem o que se espera do software.

De fato, muitos dos meus artefatos são rascunhos de Domain Models, rascunhos de casos de uso, rascunhos de protótipos de tela, rascunhos de diagrama de atividades. Tudo em papel A4, desenhados junto com o cliente (e com isso, automaticamente aprovados por ele, pelo menos até ele ver o software). Com esses rascunhos na mão, volto para casa e transformo isso em software funcionando, seguindo boas práticas de engenharia.

4.11 REFERÊNCIAS

AMBLER, Scott. *Agile Modeling:* Effective Practices for Modeling and Documentation. Disponível em: www.agilemodeling.com.

EVANS, Eric. *Domain-Driven Design:* Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004.

FOWLER, Martin. *The New Methodology*. Dezembro 2005.

Disponível em: http://www.martinfowler.com/articles/newMethodology.html.

IBM. *Test C2140-839:* Rational Unified Process v7.0. 2007. Disponível em: http://www-03.ibm.com/certify/tests/ovrC2140-839.shtml.