Orientação a Objeto - Regras Básicas de Design

Objetivos da Seção

- Aprender algumas das regras básicas nas quais o programador deve se apoiar ao projetar software
  - Buscamos princípios de um bom projeto OO
- Acompanhar um exemplo de refatoramento de software, transformando um software de qualidade pobre num de melhor qualidade

Regras Básicas de Design

- O que é Design?
  - É uma das partes mais difíceis da programação
  - Consiste em criar abstrações
- Isto significa três coisas:
  - Quais classes devem ser criadas?
  - Quais responsabilidades (métodos) devem ser assumidas por cada classe?
  - Quais são os relacionamentos entre tais classes e objetos dessas classes?
- Criar boas abstrações é difícil e vem com experiência
- Porém, algumas regras básicas ajudarão a adquirir a experiência mais rapidamente

Responsabilidades

- Responsabilidades são obrigações de um tipo ou de uma classe
- Obrigações de fazer algo
  - Fazer algo a si mesmo
  - Iniciar ações em outros objetos
  - Controlar ou coordenar atividades em outros objetos
- Obrigações de conhecer algo
  - Conhecer dados encapsulados
  - Conhecer objetos relacionados
  - Conhecer coisas que ele pode calcular
- Exemplos
  - Uma Conta bancária tem a responsabilidade de logar as transações (fazer algo)
  - Uma Conta bancária tem a responsabilidade de saber sua data de criação (conhecer algo)

Regra 1: Keep It Simple, Stupid (KISS)

- Lembre de Saint-Exupéry:
  - "Atingimos a perfeição não quando nada pode acrescentar-se a um projeto mas quando nada pode retirar-se"
- Esta é a MegaRegra

Regra 2: Colocar as Responsabilidades com os Dados

- Qual é o princípio mais fundamental para atribuir responsabilidades?
- É este: Atribuir uma responsabilidade ao expert de informação - a classe que possui a informação necessária para preencher a responsabilidade
- Exemplo: Entre as seguintes classes do mundo bancário, (Agencia, Conta, ContaCaixa, ContaSimples, Extrato, ExtratoHTML, Moeda, Movimento, Real, Transacao), quem deve ser responsável pela responsabilidade "Localizar a conta com certo número"?
  - Perguntamos: onde estão guardadas as Contas? (onde estão os dados)
  - Estão na Agencia
  - Portanto, a classe Agencia deve ter a responsabilidade (através do método localizarConta(int número))
- Consequências
  - A encapsulação é mantida, já que objetos usam sua própria informação para cumprir suas responsabilidades

        - Leva a fraco acoplamento entre objetos e sistemas mais robustos e fáceis de manter
    - Leva a alta coesão, já que os objetos fazem tudo que é relacionado à sua própria informação
- Também conhecido como:
    - "Quem sabe, faz"
    - "Expert"
    - "Animação" (objetos são vivos e podem assumir qualquer responsabilidade, mesmo que sejam passivos no mundo real)
        - Exemplo: No mundo bancário real, uma agência é algo passivo e não "localiza contas"
    - "Eu mesmo faço"
    - "Colocar os serviços junto aos atributos que eles manipulam"

Regra 3: Fraco Acoplamento

- O problema:
    - Como minimizar dependências e maximizar o reuso?
    - O acoplamento é uma medida de quão fortemente uma classe está conectada, possui conhecimento ou depende de outra classe
    - Com fraco acoplamento, uma classe não é dependente de muitas outras classes
    - Com uma classe possuindo forte acoplamento, temos os seguintes problemas:
        - Mudanças a uma classe relacionada força mudanças locais à classe
        - A classe é mais difícil de entender isoladamente
        - A classe é mais difícil de ser reusada, já que depende da presença de outras classes
- A solução: Atribuir responsabilidades de forma a minimizar o acoplamento
- Discussão
    - Minimizar acoplamento é um dos princípios de ouro do projeto OO
    - Acoplamento de manifesta de várias formas:
        - X tem um atributo que referencia uma instância de Y
        - X tem um método que referencia uma instância de Y
            - Pode ser parâmetro, variável local, objeto retornado pelo método
        - X é uma subclasse direta ou indireta de Y
        - X implementa a interface Y
    - A herança é um tipo de acoplamento particularmente forte
        - Uma seção futura esmiuça o assunto
    - Não se deve minimizar acoplamento criando alguns poucos objetos monstruosos (God classes)
        - Exemplo: todo o comportamento numa classe e outras classes usadas como depósitos passivos de informação
- Exemplo: Ordenação de registros de alunos por matrícula e nome

```
class Aluno {
     String nome;
     long   matrícula;

     public String getNome() { return nome; }
     public long   getMatrícula() { return matrícula; }

     // etc.
}

ListaOrdenada listaDeAlunos = new ListaOrdenada();
Aluno novoAluno = new Aluno(...);
//etc.
listaDeAlunos.add(novoAluno);
```

- Agora, vamos ver os problemas

```
class ListaOrdenada {
     Object[] elementosOrdenados = new Object[tamanhoAdequado];

     public void add(Aluno x) {
          // código não mostrado aqui
          // ...
          long matrícula1 = x.getMatrícula();
          long matrícula2 = elementosOrdenados[k].getMatrícula();
          if(matrícula1 < matrícula2) {
               // faça algo
          } else {
               // faça outra coisa
          }
     }
```

- O problema da solução anterior é que há forte acoplamento
  - ListaOrdenada sabe muita coisa de Aluno
    - O fato de que a comparação de alunos é feito com a matrícula
    - O fato de que a matrícula é obtida com getMatrícula()
    - O fato de que matrículas são long (representação de dados)
    - Como comparar matrículas (com <)
  - O que ocorre se mudarmos qualquer uma dessas coisas?
- Solução 2: mande uma mensagem para o próprio objeto se comparar com outro

```
class ListaOrdenada {
     Object[] elementosOrdenados = new Object[tamanhoAdequado];

     public void add(Aluno x) {
          // código não mostrado
          // ...
          if(x.compareTo(elementosOrdenados[K]) < 0) {
               // faça algo
          } else {
               // faça outra coisa
          }
     }
```

- Reduzimos o acoplamento escondendo informação atrás de um método
- Problema: ListaOrdenada só funciona com Aluno
- Solução 3: use interfaces para desacoplar mais ainda

```
Interface Comparable {
     public int compareTo(Object outro);
}

class Aluno implements Comparable {
     public int compareTo(Object outro) {
          // compare registro de aluno com outro
          return ...
     }
}

class ListaOrdenada {
     Object[] elementosOrdenados = new Object[tamanhoAdequado];

     public void add(Comparable x) {
          // código não mostrado
          if(x.compareTo(elementosOrdenados[K]) < 0) {
               // faça algo
```

```
        } else {
            // faça outra coisa
        }
    }
```

- Outro exemplo de redução de acoplamento: polimorfismo com interfaces
  - Temos vários tipos de composites (coleções) que não pertencem a uma mesma hierarquia
    - ColeçãoDeAlunos
    - ColeçãoDeProfessores
    - ColeçãoDeDisciplinas
  - Temos um cliente comum dessas coleções
    - Digamos um selecionador de objetos usado numa interface gráfica para abrir uma list box para selecionar objetos com um determinado nome
    - Exemplo:
      - Quero listar todos os alunos com nome "João" e exibí-los numa list box para escolha pelo usuário
      - Idem para listar professores com nome "Alfredo"
      - Idem para listar disciplinas com nome "Programação"
  - Queremos fazer um único cliente para qualquer uma das coleções
  - O exemplo abaixo tem polimorfismo em dois lugares

```
interface SelecionávelPorNome {
    Iterator getIteradorPorNome(String nome);
}

interface Nomeável {
    String getNome();
}

classe ColeçãoDeAlunos implements SelecionávelPorNome {
    // ...
    Iterator getIteradorPorNome(String nome) {
        // ...
    }
}

classe Aluno implements Nomeável {
    // ...
    String getNome() { ... }
}

classe ColeçãoDeProfessores implements SelecionávelPorNome {
    // ...
    Iterator getIteradorPorNome(String nome) {
        // ...
    }
}

classe Professor implements Nomeável {
    // ...
    String getNome() { ... }
}

classe ColeçãoDeDisciplinas implements SelecionávelPorNome {
    // ...
    Iterator getIteradorPorNome(String nome) {
        // ...
    }
}
```

```
classe Disciplina implements Nomeável {
    // ...
    String getNome() { ... }
}

classe ComponenteDeSeleção {
    Iterator it;
    // observe o tipo do parâmetro (uma interface)
    public ComponenteDeSeleção(SelecionávelPorNome coleção, String nome) {
        it = coleção.getIteradorPorNome(nome); // chamada polimórfica
    }
    // ...
    void geraListBox() {
        response.out.println("<select name=\"nome\" size=\"1\">");
        while(it.hasNext()) {
                int i = 1;
                // observe o tipo do objeto
                Nomeável obj = (Nomeável)it.next();
                response.out.println("<option value=\"escolha" + i + "\">" +
                                     obj.getNome() + // chamada polimórfica
                                     "</option>");
        }
        response.out.println("</select>");
    }
}

    // Como usar o código acima num servlet:
    // supõe que as coleções usam o padrão Singleton
    ComponenteDeSeleção cds =
        new ComponenteDeSeleção(ColeçãoDeAlunos.getInstance(), "João");
    cds.geraListBox();

    cds = new ComponenteDeSeleção(ColeçãoDeDisciplinas.getInstance(), "Programa
    cds.geraListBox();
```

Regra 4: Alta Coesão

- O problema:
  - Como gerenciar a complexidade?
  - A coesão mede quão relacionados ou focados estão as responsabilidades da classe
    - Também chamado coesão funcional
  - Uma classe com baixa coesão faz muitas coisas não relacionadas e leva aos seguintes problemas:
    - Difícil de entender
    - Difícil de reusar
    - Difícil de manter
    - "Delicada": constantemente sendo afetada por outras mudanças
  - Uma classe com baixa coesão assumiu responsabilidades que pertencem a outras classes e deveriam ser delegadas
- Solução:
  - Atribuir responsabilidades que mantenham alta coesão
- Exemplo: O que acha da classe que segue?

```
class Angu {
    public static int acharPadrão(String texto, String padrão) {
        // ...
    }
```

```
    public static int média(List números) {
        // ...
    }
    public static outputStream abreArquivo(string nomeArquivo) {
        // ...
    }
}
class Xpto extends Angu { // quer aproveitar código de Angu
    ...
}
```

Um Exemplo de Refatoramento

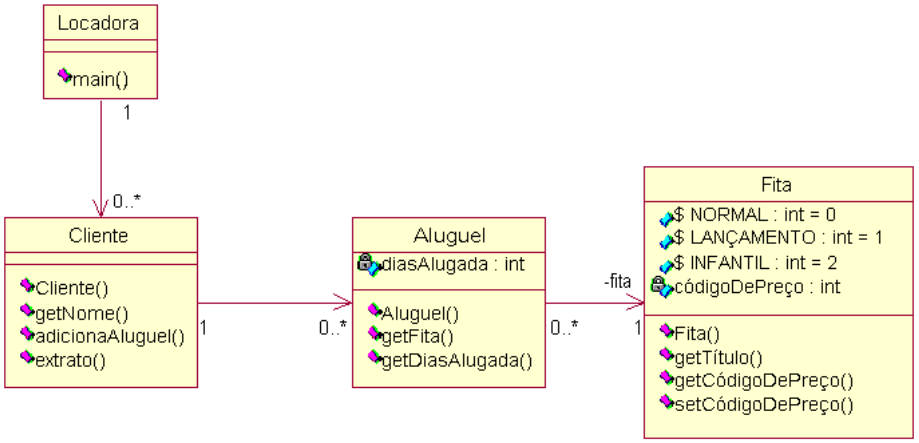
- Como programador, você precisa treinar seu nariz para detectar "mau cheiro" em código
- Veremos um exemplo disso agora
- Melhoraremos o programa através de refatoramento
    - Refatoramento alterar um programa mas sem afetar a funcionalidade que ele oferece
- Nosso exemplo é pequeno devido a restrições de tempo, mas imagine o que ocorreria se um código grande fosse tão mal feito quanto o que veremos agora
- Refatoramento sempre deve ser feito apoiando-se em Testes de Unidade para assegurar-se de que as transformações no código não quebrem código que funciona
    - Não mostraremos os testes de unidade aqui por questão de tempo

O Programa Original

- Referência: Livro de Fowler: "Refactoring"
- O programa é simples: ele calcula e emite um extrato de um cliente numa vídeo-locadora
- Vamos executar o programa para ver uma saída possível:

```
C:\...\locadora-v1>java Locadora
Registro de Alugueis de Juliana
        O Exorcista                     3.5
        Men in Black                    2.0
        Jurassic Park III               9.0
        Planeta dos Macacos             12.0
        Pateta no Planeta dos Macacos   12.0
        O Rei Leao                      42.0
Valor total devido: 80.5
Voce acumulou 8 pontos de alugador frequente
```

- Eis o diagrama UML das classes principais



- A classe Fita é usada apenas para conter os atributos (locadora-v1\Fita.java)

```
public class Fita {
  public static final int NORMAL = 0;
  public static final int LANÇAMENTO = 1;
  public static final int INFANTIL = 2;

  private String título;
  private int códigoDePreço;

  public Fita(String título, int códigoDePreço) {
    this.título = título;
    this.códigoDePreço = códigoDePreço;
  }

  public String getTítulo() {
    return título;
  }

  public int getCódigoDePreço() {
    return códigoDePreço;
  }

  public void setCódigoDePreço(int códigoDePreço) {
    this.códigoDePreço = códigoDePreço;
  }
}
```

- A classe Aluguel representa o aluguel de uma fita por um certo número de dias (locadora-v1\Aluguel.java)

```
public class Aluguel {
  private Fita fita;
  private int diasAlugada;

  public Aluguel(Fita fita, int diasAlugada) {
    this.fita = fita;
    this.diasAlugada = diasAlugada;
  }

  public Fita getFita() {
    return fita;
  }

  public int getDiasAlugada() {
    return diasAlugada;
  }
}
```

- A classe Cliente representa um freguês da locadora de vídeo (locadora-v1\Cliente.java)

```
import java.util.*;

public class Cliente {
  private String nome;
  private Collection fitasAlugadas = new ArrayList();

  public Cliente(String nome) {
    this.nome = nome;
  }
```

```
  public String getNome() {
    return nome;
  }

  public void adicionaAluguel(Aluguel aluguel) {
    fitasAlugadas.add(aluguel);
  }

  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      double valorCorrente = 0.0;
      Aluguel cada = (Aluguel)alugueis.next();

      // determina valores para cada linha
      switch(cada.getFita().getCódigoDePreço()) {
      case Fita.NORMAL:
        valorCorrente += 2;
        if(cada.getDiasAlugada() > 2) {
          valorCorrente += (cada.getDiasAlugada() - 2) * 1.5;
        }
        break;
      case Fita.LANÇAMENTO:
        valorCorrente += cada.getDiasAlugada() * 3;
        break;
      case Fita.INFANTIL:
        valorCorrente += 1.5;
        if(cada.getDiasAlugada() > 3) {
          valorCorrente += (cada.getDiasAlugada() - 3) * 1.5;
        }
        break;
      } //switch
      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" + valorCorrente + f
      valorTotal += valorCorrente;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
                 " pontos de alugador frequente";
    return resultado;
  }
}
```

- Finalmente, a classe Locadora exercita o programa:

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

```
    Cliente c1 = new Cliente("Juliana");

    c1.adicionaAluguel(new Aluguel(new Fita("O Exorcista                    ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Men in Black                   ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Jurassic Park III              ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Planeta dos Macacos            ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Pateta no Planeta dos Macacos ", F
    c1.adicionaAluguel(new Aluguel(new Fita("O Rei Leao                     ", F

    System.out.println(c1.extrato());
  }
}
```

Comentários sobre o Programa Original

- O programa não está bem projetado e não é "orientado a objeto"
- O mau cheiro que indica isso é:
  - O método extrato() é muito grande e faz tudo sozinho
  - Não há responsabilidades assumidas pelas classes Fita e Aluguel
- Mas o que importa isso se o programa funciona?
  - Código ruim é difícil de alterar
  - Se é difícil, então bugs serão introduzidos
  - Exemplo: o que deve ser mudado para ter um extrato em HTML?
    - Nada pode ser reusado!
    - Um novo método inteiro deve ser escrito, sem aproveitar código existente
  - Claro que você pode resolver isso com "cut-and-paste"
    - Mas o que ocorre se as regras de preços mudarem?
    - Vai ter que alterar código em dois lugares
  - Outro exemplo: a classificação em 3 tipos de fitas vai mudar mas os donos da locadora não sabem exatamente o que querem ainda e você pode ter certeza que haverá várias mudanças ao longo do tempo
    - Nosso código está pronto para lidar facilmente com um novo esquema de classificação de fitas? Não.
    - Nosso código está pronto para lidar facilmente com um novo esquema de pontos de alugador frequente? Não.

Refatoramento: Decomposição e Redistribuição do método extrato()

- Antes de continuar, repetimos: Para refatorar, você precisa ter testes automáticos
  - Vamos supor que eles existam (não os veremos por questão de tempo)
- Ataquemos o primeiro problema: o método extrato() é muito grande e "faz tudo sozinho"
  - Vamos decompor este método em pedaços menores
- Vamos pegar um bloco de código com alguma coesão e vamos extrair e colocá-lo num método
  - Qual bloco escolher?
  - A experiência é importante aqui mas também lembre a regra sobre Alta Coesão
  - O switch é o cálculo de valorCorrente para uma fita e parece um pedaço coeso que merece um método à parte
    - Teste de coesão: O trabalho que o método faz pode ser dito numa frase curta?
    - Se puder, é um bom método a ser criado
    - Aqui, podemos dizer que o bloco extraído "calcula o preço de aluguel de uma fita"
    - Parece coeso
- Segue o cósigo antes e depois do refatoramento, com o azul indicando as mudanças
- Antes

```
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
```

```
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      double valorCorrente = 0.0;
      Aluguel cada = (Aluguel)alugueis.next();

      // determina valores para cada linha
      switch(cada.getFita().getCódigoDePreço()) {
      case Fita.NORMAL:
        valorCorrente += 2;
        if(cada.getDiasAlugada() > 2) {
          valorCorrente += (cada.getDiasAlugada() - 2) * 1.5;
        }
        break;
      case Fita.LANÇAMENTO:
        valorCorrente += cada.getDiasAlugada() * 3;
        break;
      case Fita.INFANTIL:
        valorCorrente += 1.5;
        if(cada.getDiasAlugada() > 3) {
          valorCorrente += (cada.getDiasAlugada() - 3) * 1.5;
        }
        break;
      } //switch
      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" + valorCorrente + f
      valorTotal += valorCorrente;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
                 " pontos de alugador frequente";
    return resultado;
  }
```

- Depois

```
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      valorCorrente = valorDeUmAluguel(cada);

      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
```

```
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" + valorCorrente + f
      valorTotal += valorCorrente;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
              " pontos de alugador frequente";
    return resultado;
  }
  private int valorDeUmAluguel(Aluguel cada) {
    int valorCorrente = 0;
    // determina valores para cada linha
    switch(cada.getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorCorrente += 2;
      if(cada.getDiasAlugada() > 2) {
        valorCorrente += (cada.getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorCorrente += cada.getDiasAlugada() * 3;
      break;
    case Fita.INFANTIL:
      valorCorrente += 1.5;
      if(cada.getDiasAlugada() > 3) {
        valorCorrente += (cada.getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorCorrente;
  }
```

- Depois de uma mudança dessas, compilamos e testamos para verificar que não quebramos nada
  - Ao testar, verificamos que vários testes falham!
  - Examinando os testes que falharam e o código, observamos logo que usamos "int" em vez de "double"
  - Daí a importância de sempre ter testes para refatorar
- O método é mudado para a versão seguinte:

```
  private double valorDeUmAluguel(Aluguel cada) {
    double valorCorrente = 0;
    // determina valores para cada linha
    switch(cada.getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorCorrente += 2;
      if(cada.getDiasAlugada() > 2) {
        valorCorrente += (cada.getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorCorrente += cada.getDiasAlugada() * 3;
      break;
```

```
    case Fita.INFANTIL:
      valorCorrente += 1.5;
      if(cada.getDiasAlugada() > 3) {
        valorCorrente += (cada.getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorCorrente;
  }
```

- Separamos o método grande em dois
  - Agora podemos continuar a trabalhar em cada pedaço individualmente
  - Princípio da Divisão-e-Conquista para lidar com a complexidade
- Vamos fazer uma mudança pequena no método valorDeAluguel
  - Algumas variáveis vão mudar de nome

```
  private double valorDeUmAluguel(Aluguel umAluguel) {
    double valorDoAluguel = 0;
    // determina valores para cada linha
    switch(umAluguel.getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(umAluguel.getDiasAlugada() > 2) {
        valorDoAluguel += (umAluguel.getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += umAluguel.getDiasAlugada() * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(umAluguel.getDiasAlugada() > 3) {
        valorDoAluguel += (umAluguel.getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
```

- Vale a pena mudar nomes de variáveis assim?
  - Claro!
  - O código deve comunicar bem seu propósito para outros programadores
  - Nomes de variáveis são um meio básico de comunicação
  - Qualquer idiota pode escrever código que um computador entende. Bons programadores escrevem código que um ser humano pode entender.

Refatoramento: Movendo o Cálculo de Valores

- Examine o código do método valorDeUmAluguel()
  - O método usa informação de um objeto da classe Aluguel mas nada usa do Cliente
  - Pela regra de design "Colocar as Responsabilidades com os Dados", desconfiamos que o método está na classe errada
- Vamos mover o método para a classe Aluguel
- Código antes:

```
class Cliente ...
  private double valorDeUmAluguel(Aluguel umAluguel) {
    double valorDoAluguel = 0;
    // determina valores para cada linha
```

```
    switch(umAluguel.getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(umAluguel.getDiasAlugada() > 2) {
        valorDoAluguel += (umAluguel.getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += umAluguel.getDiasAlugada() * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(umAluguel.getDiasAlugada() > 3) {
        valorDoAluguel += (umAluguel.getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
```

- Código depois

```
class Aluguel ...
  double getValorDoAluguel() { // observe que o parâmetro umAluguel sumiu
    double valorDoAluguel = 0;
    // determina valores para cada linha
    switch(getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(getDiasAlugada() > 2) {
        valorDoAluguel += (getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += getDiasAlugada() * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(getDiasAlugada() > 3) {
        valorDoAluguel += (getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }

class Cliente ...
  private double valorDeUmAluguel(Aluguel umAluguel) {
    return umAluguel.getValorDoAluguel();
  }
```

- No código acima, observe como delegamos em Cliente.valorDeUmAluguel
  - Fizemos isso para fazer pequenas mudanças de cada vez ao refatorar
  - Depois de testar, podemos remover o método valorDeUmAluguel e chamar getValorDoAluguel() diretamente
- Código antes

```
class Cliente ...
    ...
    valorCorrente = valorDeUmAluguel(cada);
    ...
```

- Código depois

```
class Cliente ...
      ...
      valorCorrente = cada.getValorDoAluguel();
      ...
```

- Isso parece muito mais orientado a objeto!
  - A classe correta (Aluguel) assumiu a responsabilidade de calcular o preço do aluguel
- O próximo passo pode ser a remoção de variáveis temporárias desnecessárias
  - Exemplo: valorCorrente em extrato()
  - Variáveis a menos são coisas a menos para dar errado, não ter valor correto, não ser inicializada, etc.
- Código antes

```
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      valorCorrente = valorDeUmAluguel(cada);

      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" + valorCorrente + fimDeLinha;
      valorTotal += valorCorrente;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
                 " pontos de alugador frequente";
    return resultado;
  }
```

- Código depois

```
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
      valorTotal += cada.getValorDoAluguel();
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
```

```
            " pontos de alugador frequente";
    return resultado;
  }
```

- Claro que depois de cada mudança, compile e teste

Refatoramento: Extração de Pontos de Alugador Frequente

- O que fizemos com o valor do aluguel pode ser feito com o cálculo dos pontos de alugador frequente (PAF)
- Quem deve ter a responsabilidade de calcular os PAF?
  - O cálculo depende de informação que Aluguel conhece
  - Deixe portanto o cálculo na classe Aluguel
- Código antes:

```
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      // trata de pontos de alugador frequente
      pontosDeAlugadorFrequente++;
      // adiciona bonus para aluguel de um lançamento por pelo menos 2 dias
      if(cada.getFita().getCódigoDePreço() == Fita.LANÇAMENTO &&
         cada.getDiasAlugada() > 1) {
         pontosDeAlugadorFrequente++;
      }

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
      valorTotal += cada.getValorDoAluguel();
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
              " pontos de alugador frequente";
    return resultado;
  }
```

- Código depois:

```
class Cliente ...
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      pontosDeAlugadorFrequente += cada.getPontosDeAlugadorFrequente();

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
      valorTotal += cada.getValorDoAluguel();
```

```
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
              " pontos de alugador frequente";
    return resultado;
  }

class Aluguel ...
  int getPontosDeAlugadorFrequente() {
    if(getFita().getCódigoDePreço() == Fita.LANÇAMENTO && getDiasAlugada() > 1)
       return 2;
    } else {
       return 1;
    }
  }
```

Refatoramento: Remoção de Variáveis Temporárias

- Mais uma vez, vamos falar de variáveis temporárias
- Embora elas possam ser úteis, elas frequentemente são indicativos de "mau cheiro"
- Examine, por exemplo, a variável valorTotal
  - Ela é usada para calcular o valor total do extrato enquanto estamos no loop
  - Na realidade, o loop está servindo para três coisas:
    - Montar o String do extrato
    - Calcular o valor total
    - Calcular os FAP
  - Porém, esse trabalho talvez seja necessário em outro lugar
    - Por exemplo, posso querer saber o valor total em outro método (extratoHTML()) e terei portanto que repetir o cálculo do preço total neste lugar
  - Faz sentido criarmos um método getValorTotal()?
    - Este método faz algo que podemos resumir em uma frase curta?
    - Sim! Portanto, crie o método
- O mesmo pode ser dito sobre a variável temporária pontosDeAlugadorFrequente
  - Seria melhor criar um método getPontosDeAlugadorFrequente()
- Embora esses dois passos devam ser feitos separadamente com testes a cada passo, vamos logo ver o resultado dos dois passos
- Código antes:

```
class Cliente ...
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    double valorTotal = 0.0;
    int pontosDeAlugadorFrequente = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      pontosDeAlugadorFrequente += cada.getPontosDeAlugadorFrequente();

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
      valorTotal += cada.getValorDoAluguel();
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + valorTotal + fimDeLinha;
    resultado += "Voce acumulou " + pontosDeAlugadorFrequente +
```

```
                " pontos de alugador frequente";
    return resultado;
  }
```

- Código depois:

```
class Cliente ...
  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + getValorTotal() + fimDeLinha;
    resultado += "Voce acumulou " + getPontosTotaisDeAlugadorFrequente() +
                 " pontos de alugador frequente";
    return resultado;
  }

  private double getValorTotal() {
    double valorTotal = 0.0;
    Iterator alugueis = fitasAlugadas.iterator();
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      valorTotal += cada.getValorDoAluguel();
    }
    return valorTotal;
  }

  private int getPontosTotaisDeAlugadorFrequente() {
    int pontos = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      pontos += cada.getPontosDeAlugadorFrequente();
    }
    return pontos;
  }
```

- Pare aí! Acabamos de deixar o código *maior* com a última mudança!
  - Valeu a pena?
  - Sim!
  - Motivos:
    - Criamos dois métodos úteis que poderão ser usados mais na frente
    - Eles poderão até ser tornados públicos se for necessário que entrem na interface da classe
    - Organizamos o código melhor onde cada pedacinho é mais simples de entender
      - Compare o método original extrato() com a última versão
- E quanto ao desempenho?? Temos mais loops do que antes!
  - É possível que haja um problema de desempenho mas só saberemos isso com um perfil de execução
  - Neste caso, numa aplicação de locadora de vídeo onde clientes alugam poucas fitas, eu *garanto* que o desempenho não será afetado

- Os loops adicionais vão adicionar alguns milissegundos ao processamento
- Mas quanto tempo demora para imprimir o extrato em papel?!!?

- Agora podemos ver como é fácil criar um extrato em HTML devido à existência dos dois métodos úteis que criamos

```
class Cliente ...
  public String extratoHTML() {
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "<H1>Registro de Alugueis de <EM>" +
                       getNome() + "</EM></H1><P>\n";
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      // mostra valores para este aluguel
      resultado += cada.getFita().getTítulo() + ": " +
                   cada.getValorDoAluguel() + "<BR>\n";
    } // while
    // adiciona rodapé
    resultado += "<P>Valor total devido: <EM>" + getValorTotal() + "</EM>\n";
    resultado += "<P>Voce acumulou <EM>" + getPontosTotaisDeAlugadorFrequente()
              "</EM> pontos de alugador frequente";
    return resultado;
  }
```

Refatoramento: Responsabilidades onde estão os dados

- O switch está com problemas
  - Ao ver um switch, verifique se o teste está sendo feito em cima dos seus próprios dados ou em cima dos dados de outro objeto
  - Aqui, vemos que o Aluguel faz um switch em cima de dados da Fita!
  - Portanto, faz mais sentido mover o método getValorDoAluguel() para a classe Fita
- Código antes:

```
class Aluguel ...
  double getValorDoAluguel() {
    double valorDoAluguel = 0;
    // determina valores para cada linha
    switch(getFita().getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(getDiasAlugada() > 2) {
        valorDoAluguel += (getDiasAlugada() - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += getDiasAlugada() * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(getDiasAlugada() > 3) {
        valorDoAluguel += (getDiasAlugada() - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
```

- Código depois:

```
class Aluguel ...
  double getValorDoAluguel() {
    return fita.getValorDoAluguel(diasAlugada);
  }
}

class Fita ...
  double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 0;
    switch(getCódigoDePreço()) {
    case NORMAL:
      valorDoAluguel += 2;
      if(diasAlugada > 2) {
        valorDoAluguel += (diasAlugada - 2) * 1.5;
      }
      break;
    case LANÇAMENTO:
      valorDoAluguel += diasAlugada * 3;
      break;
    case INFANTIL:
      valorDoAluguel += 1.5;
      if(diasAlugada > 3) {
        valorDoAluguel += (diasAlugada - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
}
```

- Podemos fazer o mesmo com o cálculo de PAF:
- Código antes:

```
class Aluguel ...
  int getPontosDeAlugadorFrequente() {
    if(getFita().getCódigoDePreço() == Fita.LANÇAMENTO && getDiasAlugada() > 1)
       return 2;
    } else {
       return 1;
    }
  }
```

- Código depois:

```
class Aluguel ...
  int getPontosDeAlugadorFrequente() {
    return fita.getPontosDeAlugadorFrequente(diasAlugada);
  }


class Fita ...
  int getPontosDeAlugadorFrequente(int diasAlugada) {
    if(getCódigoDePreço() == LANÇAMENTO && diasAlugada > 1) {
       return 2;
    } else {
       return 1;
    }
  }
```

Refatoramento: Uso de Polimorfismo

- Queremos um código que permita facilmente adicionar novas classificações de fitas

Refatoramento: Interfaces

- Temos uma classe (Fita) que possui dois métodos que têm comportamento diferente dependendo de algum atributo do objeto
  - Veja o switch de getValorDoAluguel()
  - Veja o teste em getPontosDeAlugadorFrequente()
- Isso é indicativo que o polimorfismo poderia limpar as coisas
- De fato, Fitas diferentes poderiam responder de forma diferente às duas perguntas getValorDoAluguel() e getPontosDeAlugadorFrequente()
- Podemos portanto ter polimorfismo em cima de tipos de Fitas
- Melhor ainda: queremos isolar dois mundos
  - O mundo das coisas que podem ser alugadas (fitas, jogos, DVDs, ...)
  - O mundo que usa tais coisas
- Usaremos uma interface para isolar esses dois mundos
- Vamos primeiro definir uma interface para a situação
  - Chamaremos a interface de Alugavel
  - Agora, serão Fitas, mas depois poderão ser DVDs, jogos, etc.
- Código antes:

```
public class Aluguel {
  private Fita fita;
  private int diasAlugada;

  public Aluguel(Fita fita, int diasAlugada) {
    this.fita = fita;
    this.diasAlugada = diasAlugada;
  }

  public Fita getFita() {
    return fita;
  }
  ...
}
```

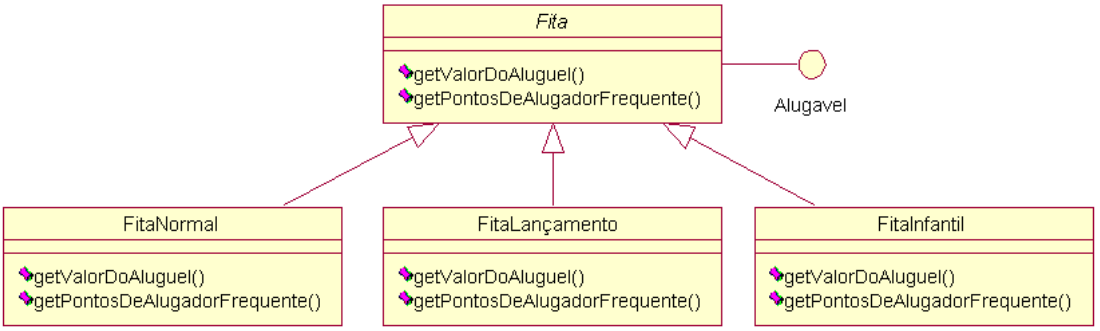
- Código depois:

```
interface Alugavel {
  String getTítulo();
  double getValorDoAluguel(int diasAlugada);
  int getPontosDeAlugadorFrequente(int diasAlugada);
}

class Fita implements Alugavel {
  ...
}

public class Aluguel {
  private Alugavel fita;
  private int diasAlugada;

  public Aluguel(Alugavel fita, int diasAlugada) {
    this.fita = fita;
    this.diasAlugada = diasAlugada;
  }

  public Alugavel getFita() {
    return fita;
```

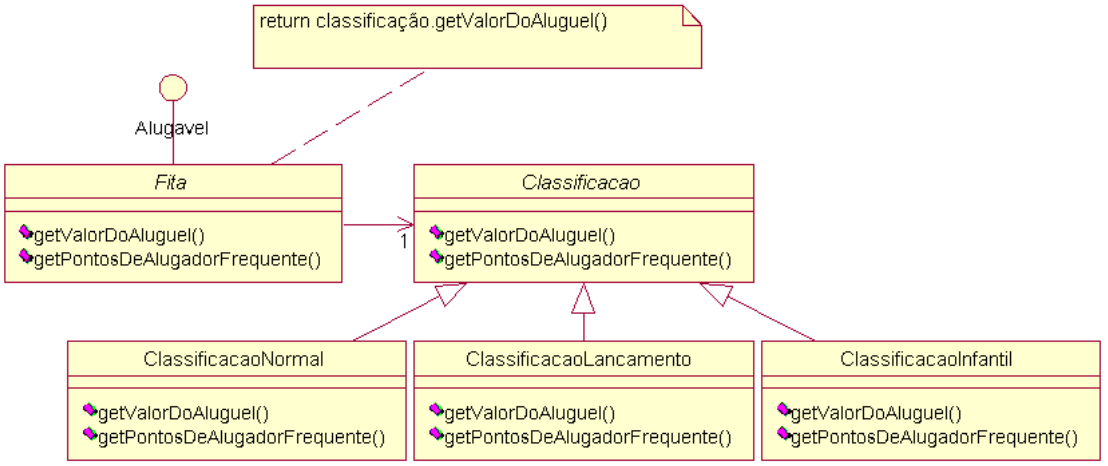
```
    }
    . . .
}
```

Refatoramento: Herança

- Ainda não temos polimorfismo porque apenas uma classe implementa a interface Alugavel
- A primeira solução que vem à mente é fazer como segue:



- Mas isso não funciona porque uma fita pode mudar sua classificação durante sua vida
  - Não podemos mudar a classe de um objeto durante sua vida
  - Isso se chama de "mutação" e não deve ser feito!
- Como lidar com isso?
  - Separe o que é igual daquilo que muda
  - Encapsule cada um em objetos diferentes
- Resultado:



- Observe que cada fita agora será composta de dois objetos:
  - Um para a fita em si
  - Um para a classificação da fita
- Falamos que está havendo composição de objetos
- Para implementar getvalorDoAluguel(), a Fita delega para o objeto de classificação
- Por que tudo isso é melhor?
  - A composição pode ser alterada em tempo de execução
    - Isto é, a Fita recebe um novo objeto composto de Classificacao
  - Isso faz com que a composição seja frequentemente superior à herança
- A herança ainda ocorre, mas não no mundo das fitas, mas no mundo das classificações
- Agora, vamos fazer isso accontecer no código
- São 2 passos:
  - Implementar a composição de objetos de forma a permitir a mudança dinâmica do objeto de classificação
  - Mover o método getValorDoAluguel de Fita para Classificacao

- Substituir os testes (switch/if) com polimorfismo
- Façamos a primeira etapa
  - Queremos que cada fita vire dois objetos: uma fita e uma classificação
  - Por enquanto, o objeto Classificacao é quem vai responder getCódigoDePreço()
    - Está havendo delegação
  - Não quero que a interface externa mude para quem usa a classe Fita
    - Portanto, quem cria o novo objeto é a própria classe Fita para esconder tudo
- Código antes:

```
class Fita ...
  private int códigoDePreço;

  public Fita(String título, int códigoDePreço) {
    this.título = título;
    this.códigoDePreço = códigoDePreço;
  }

  public int getCódigoDePreço() {
    return códigoDePreço;
  }

  public void setCódigoDePreço(int códigoDePreço) {
    this.códigoDePreço = códigoDePreço;
  }
```

- Código depois:

```
class Fita ...
  private Classificacao classificação;

  public Fita(String título, int códigoDePreço) {
    this.título = título;
    setCódigoDePreço(códigoDePreço);
  }

  public int getCódigoDePreço() {
    return classificação.getCódigoDePreço();
  }

  public void setCódigoDePreço(int códigoDePreço) {
    switch(códigoDePreço) {
    case NORMAL:
      classificação = new ClassificacaoNormal();
      break;
    case LANÇAMENTO:
      classificação = new ClassificacaoLancamento();
      break;
    case INFANTIL:
      classificação = new ClassificacaoInfantil();
      break;
    }
  }
}

public abstract class Classificacao {
  public abstract int getCódigoDePreço();
}
```

```
public class ClassificacaoNormal extends Classificacao {
  public abstract int getCódigoDePreço() {
    return Fita.NORMAL;
  }
}

public class ClassificacaoLancamento extends Classificacao {
  public abstract int getCódigoDePreço() {
    return Fita.LANÇAMENTO;
  }
}

public class ClassificacaoInfantil extends Classificacao {
  public abstract int getCódigoDePreço() {
    return Fita.INFANTIL;
  }
}
```

- Observe que, em tempo de execução, podemos mudar a classificação de uma fita
  - Basta chamar setCódigoDePreço(), como antes
- Agora, podemos atacar a segunda etapa:
  - Os objetos de classificação devem implementar getValorDoAluguel() e getPontosDeAlugadorFrequente() que estão em Fita
  - Mais uma vez, teremos delegação>
    - O objeto Fita delega para Classificacao o cálculo de getValorDoAluguel() e getPontosDeAlugadorFrequente()
- Código antes:

```
class Fita ...
  public double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 0;
    switch(getCódigoDePreço()) {
    case NORMAL:
      valorDoAluguel += 2;
      if(diasAlugada > 2) {
        valorDoAluguel += (diasAlugada - 2) * 1.5;
      }
      break;
    case LANÇAMENTO:
      valorDoAluguel += diasAlugada * 3;
      break;
    case INFANTIL:
      valorDoAluguel += 1.5;
      if(diasAlugada > 3) {
        valorDoAluguel += (diasAlugada - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
}
```

- Código depois:

```
class Fita ...
  public double getValorDoAluguel(int diasAlugada) {
    return classificação.getValorDoAluguel(diasAlugada);
  }
```

```
class Classificacao
  public double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 0;
    switch(getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(diasAlugada > 2) {
        valorDoAluguel += (diasAlugada - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += diasAlugada * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(diasAlugada > 3) {
        valorDoAluguel += (diasAlugada - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
}
```

- Estamos prontos para a última etapa: introduzir o polimorfismo
  - Implementamos os métodos adequados nas classes de Classificacao
  - Depois, o método getValorDoAluguel() pode virar abstrato na classe Classificacao
- Código antes:

```
class Classificacao
  public double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 0;
    switch(getCódigoDePreço()) {
    case Fita.NORMAL:
      valorDoAluguel += 2;
      if(diasAlugada > 2) {
        valorDoAluguel += (diasAlugada - 2) * 1.5;
      }
      break;
    case Fita.LANÇAMENTO:
      valorDoAluguel += diasAlugada * 3;
      break;
    case Fita.INFANTIL:
      valorDoAluguel += 1.5;
      if(diasAlugada > 3) {
        valorDoAluguel += (diasAlugada - 3) * 1.5;
      }
      break;
    } //switch
    return valorDoAluguel;
  }
}
```

- Código depois:

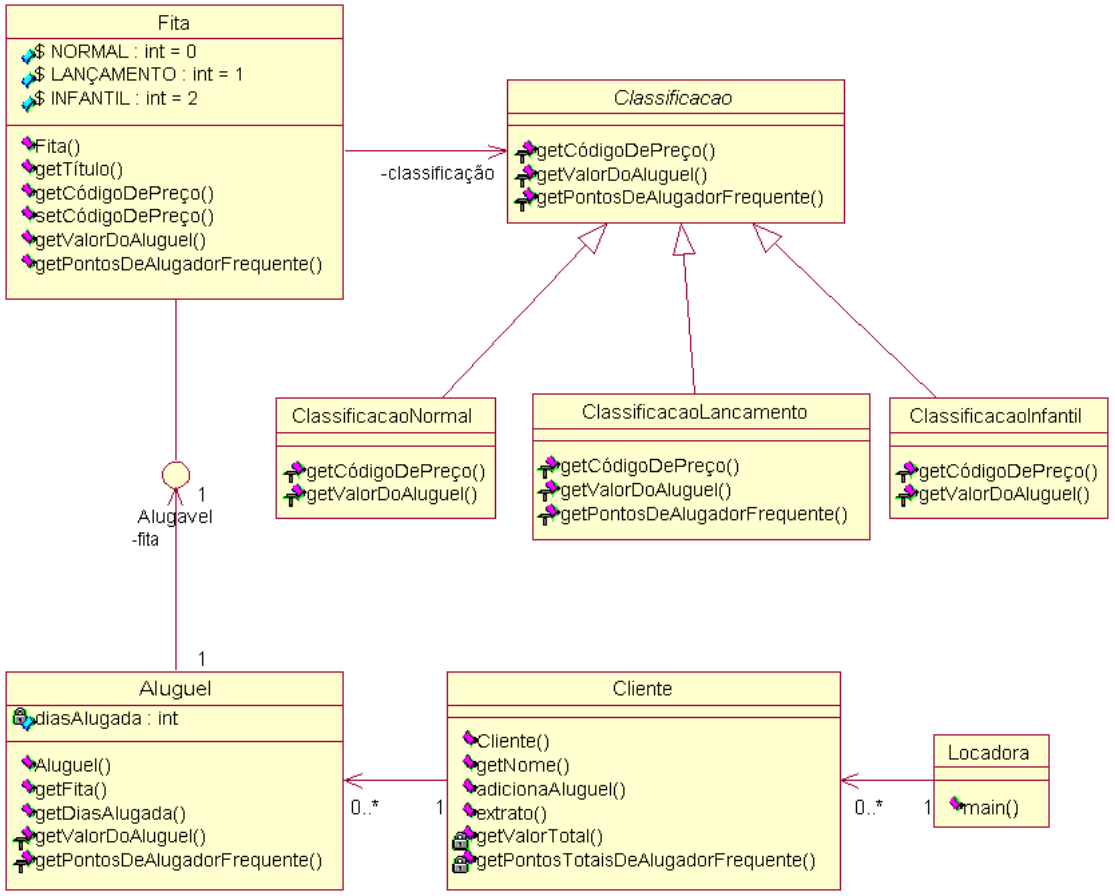
```
abstract class Classificacao
  abstract double getValorDoAluguel(int diasAlugada);
```

```
class ClassificacaoNormal extends Classificacao {
  public double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 2;
    if(diasAlugada > 2) {
      valorDoAluguel += (diasAlugada - 2) * 1.5;
    }
    return valorDoAluguel;
  }
}

class ClassificacaoLancamento extends Classificacao {
  public double getValorDoAluguel(int diasAlugada) {
    return diasAlugada * 3;
  }
}

class ClassificacaoInfantil extends Classificacao {
  public double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 1.5;
    if(diasAlugada > 3) {
      valorDoAluguel += (diasAlugada - 3) * 1.5;
    }
    return valorDoAluguel;
  }
}
```

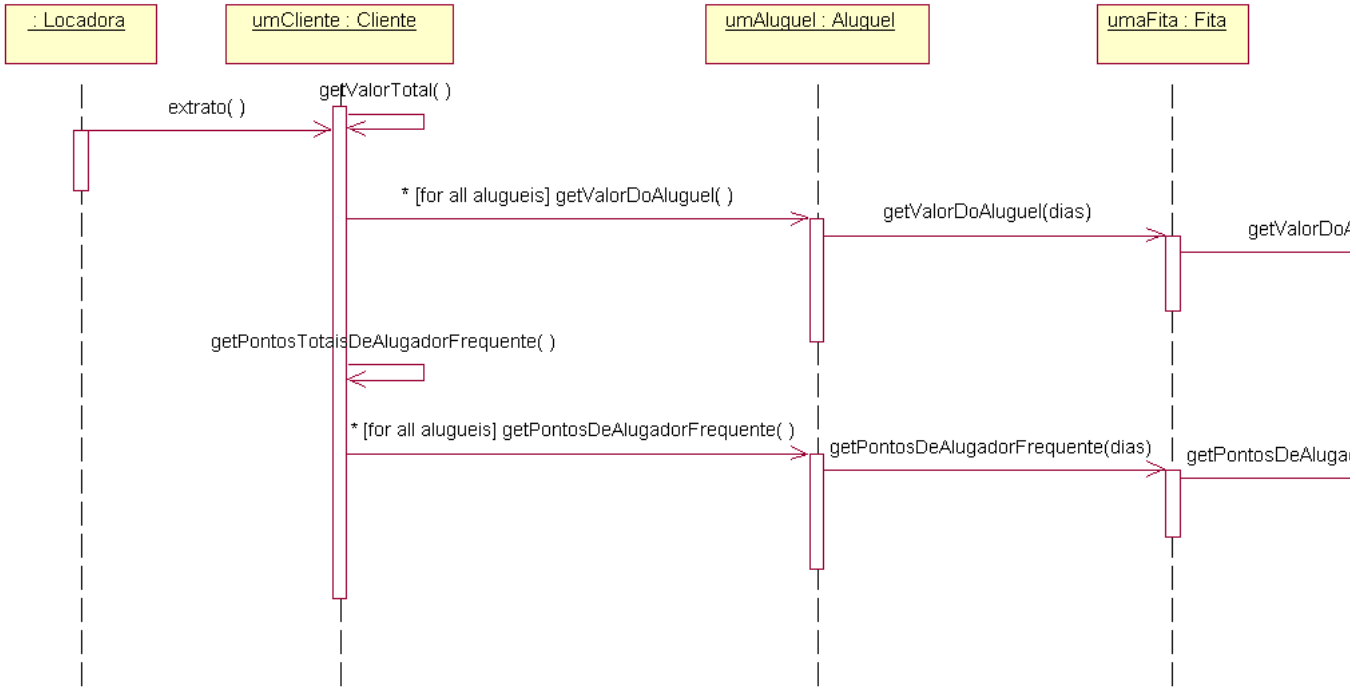
- O mesmo pode ser feito com getPontosDeAlugadorFrequente()
- Valeu a pena tanto esforço para introduzir polimorfismo?
  - Primeiramente, um bom programador já teria colocado polimorfismo desde o início
  - Segundo, valeu a pena sim: o código é muito mais simples de mudar quando houver um novo esquema de preços, por exemplo.

A solução final

- A estrutura final é a seguinte:



- Um diagrama de sequência (em UML) mostra as interações entre objetos



- Agora, podemos ver o código final
  - Observe que a classe Locadora não mudou

```
public class Locadora {
  public static void main(String[] args) {
    Cliente c1 = new Cliente("Juliana");
```

```
    c1.adicionaAluguel(new Aluguel(new Fita("O Exorcista                    ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Men in Black                   ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Jurassic Park III              ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Planeta dos Macacos            ", F
    c1.adicionaAluguel(new Aluguel(new Fita("Pateta no Planeta dos Macacos ", F
    c1.adicionaAluguel(new Aluguel(new Fita("O Rei Leao                     ", F

    System.out.println(c1.extrato());
  }
}
```

```
import java.util.*;

public class Cliente {
  private String nome;
  private Collection fitasAlugadas = new ArrayList();

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

  public String getNome() {
    return nome;
  }

  public void adicionaAluguel(Aluguel aluguel) {
    fitasAlugadas.add(aluguel);
  }

  public String extrato() {
    final String fimDeLinha = System.getProperty("line.separator");
    Iterator alugueis = fitasAlugadas.iterator();
    String resultado = "Registro de Alugueis de " + getNome() + fimDeLinha;
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();

      // mostra valores para este aluguel
      resultado += "\t" + cada.getFita().getTítulo() + "\t" +
                   cada.getValorDoAluguel() + fimDeLinha;
    } // while
    // adiciona rodapé
    resultado += "Valor total devido: " + getValorTotal() + fimDeLinha;
    resultado += "Voce acumulou " + getPontosTotaisDeAlugadorFrequente() +
              " pontos de alugador frequente";
    return resultado;
  }

  private double getValorTotal() {
    double valorTotal = 0.0;
    Iterator alugueis = fitasAlugadas.iterator();
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      valorTotal += cada.getValorDoAluguel();
    }
    return valorTotal;
  }
```

```
  private int getPontosTotaisDeAlugadorFrequente() {
    int pontos = 0;
    Iterator alugueis = fitasAlugadas.iterator();
    while(alugueis.hasNext()) {
      Aluguel cada = (Aluguel)alugueis.next();
      pontos += cada.getPontosDeAlugadorFrequente();
    }
    return pontos;
  }
}
```

```
public class Aluguel {
  private Alugavel fita;
  private int diasAlugada;

  public Aluguel(Alugavel fita, int diasAlugada) {
    this.fita = fita;
    this.diasAlugada = diasAlugada;
  }

  public Alugavel getFita() {
    return fita;
  }

  public int getDiasAlugada() {
    return diasAlugada;
  }

  double getValorDoAluguel() {
    return fita.getValorDoAluguel(diasAlugada);
  }

  int getPontosDeAlugadorFrequente() {
    return fita.getPontosDeAlugadorFrequente(diasAlugada);
  }
}
```

```
interface Alugavel {
  String getTítulo();
  double getValorDoAluguel(int diasAlugada);
  int getPontosDeAlugadorFrequente(int diasAlugada);
}
```

```
public class Fita implements Alugavel {
  public static final int NORMAL = 0;
  public static final int LANÇAMENTO = 1;
  public static final int INFANTIL = 2;

  private String título;
  private Classificacao classificação;

  public Fita(String título, int códigoDePreço) {
    this.título = título;
    setCódigoDePreço(códigoDePreço);
  }
```

```
  public String getTítulo() {
    return título;
  }

  public int getCódigoDePreço() {
    return classificação.getCódigoDePreço();
  }

  public void setCódigoDePreço(int códigoDePreço) {
    switch(códigoDePreço) {
    case NORMAL:
      classificação = new ClassificacaoNormal();
      break;
    case LANÇAMENTO:
      classificação = new ClassificacaoLancamento();
      break;
    case INFANTIL:
      classificação = new ClassificacaoInfantil();
      break;
    }
  }
  public double getValorDoAluguel(int diasAlugada) {
    return classificação.getValorDoAluguel(diasAlugada);
  }
  public int getPontosDeAlugadorFrequente(int diasAlugada) {
    return classificação.getPontosDeAlugadorFrequente(diasAlugada);
  }
}
```

```
public abstract class Classificacao {
  abstract int getCódigoDePreço();
  abstract double getValorDoAluguel(int diasAlugada);

  int getPontosDeAlugadorFrequente(int diasAlugadas) {
    return 1;
  }
}

class ClassificacaoNormal extends Classificacao {
  int getCódigoDePreço() {
    return Fita.NORMAL;
  }

  double getValorDoAluguel(int diasAlugada) {
    double valorDoAluguel = 2;
    if(diasAlugada > 2) {
      valorDoAluguel += (diasAlugada - 2) * 1.5;
    }
    return valorDoAluguel;
  }
}

class ClassificacaoLancamento extends Classificacao {
  int getCódigoDePreço() {
    return Fita.LANÇAMENTO;
  }

  double getValorDoAluguel(int diasAlugada) {
```

```
      return diasAlugada * 3;
    }

    int getPontosDeAlugadorFrequente(int diasAlugadas) {
      return (diasAlugadas > 1) ? 2 : 1;
    }
  }

  class ClassificacaoInfantil extends Classificacao {
    int getCódigoDePreço() {
      return Fita.INFANTIL;
    }

    double getValorDoAluguel(int diasAlugada) {
      double valorDoAluguel = 1.5;
      if(diasAlugada > 3) {
        valorDoAluguel += (diasAlugada - 3) * 1.5;
      }
      return valorDoAluguel;
    }
  }
```

Palavras Finais

- Uma faceta importante do refactoring é o ritmo
  - Testa, mude um pouco, testa, mude um pouco, ...
  - Este ritmo permite fazer refatoramento de forma simples, rápida e segura

oo-9 programa anterior próxima