Analysis Pattern: Regra de Lançamento (Posting Rule)

O que é

- Uma Regra de Lançamento determina que lançamentos devem ser feitos em reposta a um evento
- Queremos modelar como eventos são transformados em lançamentos

Comece simples ...

- Exemplo simples:
  - Lê o Evento que indica que Luciano trabalhou 10 horas na quarta-feira, verifique sua remuneração por hora, multiplique por 10, e crie um lançamento para que a grana entre no seu pagamento mensal
- Podemos raciocinar em termos de um processador que lê eventos e cria lançamentos:

- O problema é que há muitas variações neste cenário:
  - Cada pessoa pode ser paga de uma forma diferente. Pode haver dezenas de formas de remuneração (acordos) para os empregados (CLT, estatutário, sindicalizado, temporário, trainee, bolsista, ...)
  - Cada acordo pode indicar quantas horas você trabalha antes de receber hora extra, como as horas extras são remuneradas, se trabalha fim de semana, se sábado de manhã é hora extra ou não, etc., etc.

Introduz o acordo ...

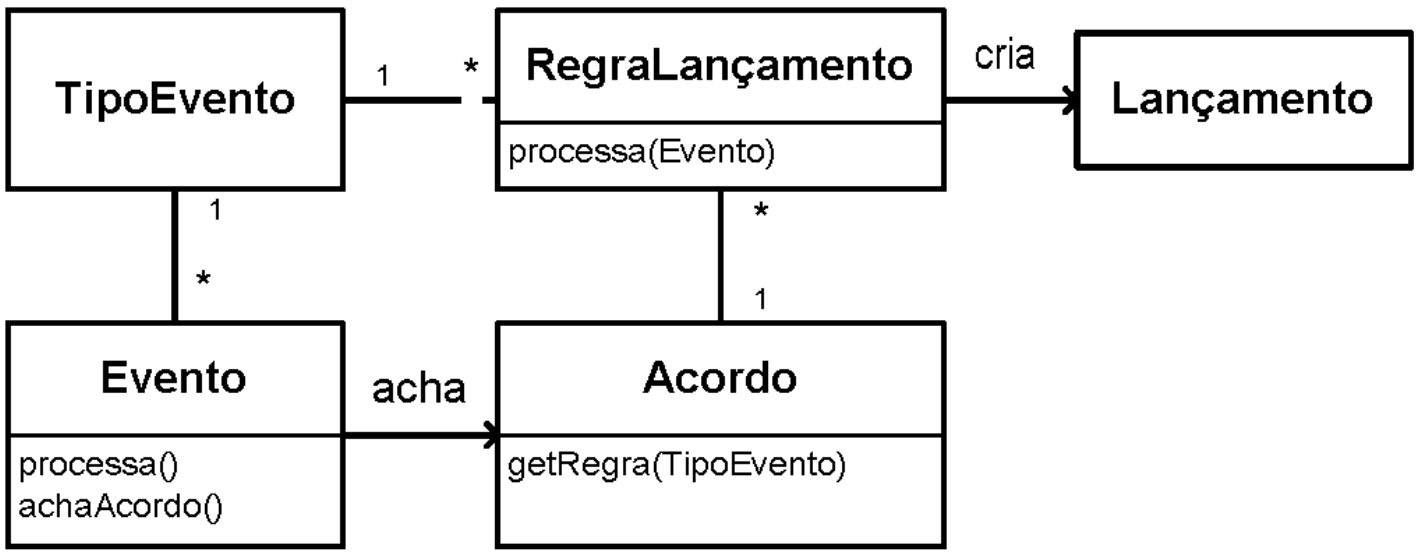
- Podemos modelar os acordos explicitamente como objetos:
  - Um evento acha o acordo que se aplica a ele e o usa para realizar o processamento
  - Observe que o evento é "ativo" agora
    - Ele tem a responsabilidade de achar um acordo e chamar seu método principal

- Vantagem de separar os acordos:
  - Todas as políticas de um acordo são separadas de outros acordos
  - Isso funciona bem se todas as políticas forem diferentes
- Desvantagem
  - Se houver políticas comuns (ex. hora extra é remunerada com 50% adicionais), elas deverão ser alteradas separadamente em cada acordo

    - Em contrapartida, isso evita mudar uma política e inadvertidamente alterar acordos nos quais não se queria mexer
- Por enquanto, vamos supor que temos acordos completamente separados e não compartilham políticas
- Acordos podem ser bichos muito complexos porque há muitos tipos de eventos no sistema
    - Exemplo: Para uma empresa de distribuição de eletricidade, uma pessoa pode usar eletricidade, fazer uma chamada de assistência, pedir desligamento, pedir religamento, alterar o tipo de serviço, etc.
    - Cada tipo de evento deve ser tratado de forma particular

Introduz Regras de Lançamentos ...

- Podemos modelar isso atavés de reificação do tipo de evento para separar o processamento para cada tipo de evento numa Regra de Lançamento
    - Desta forma, cada Regra de Lançamento recebe um tipo único de evento e gera lançamentos apenas para isso
    - O evento acha o acordo, e extrai do acordo a Regra de Lançamento para o tipo de evento, chama a Regra que processa o evento e cria os lançamentos



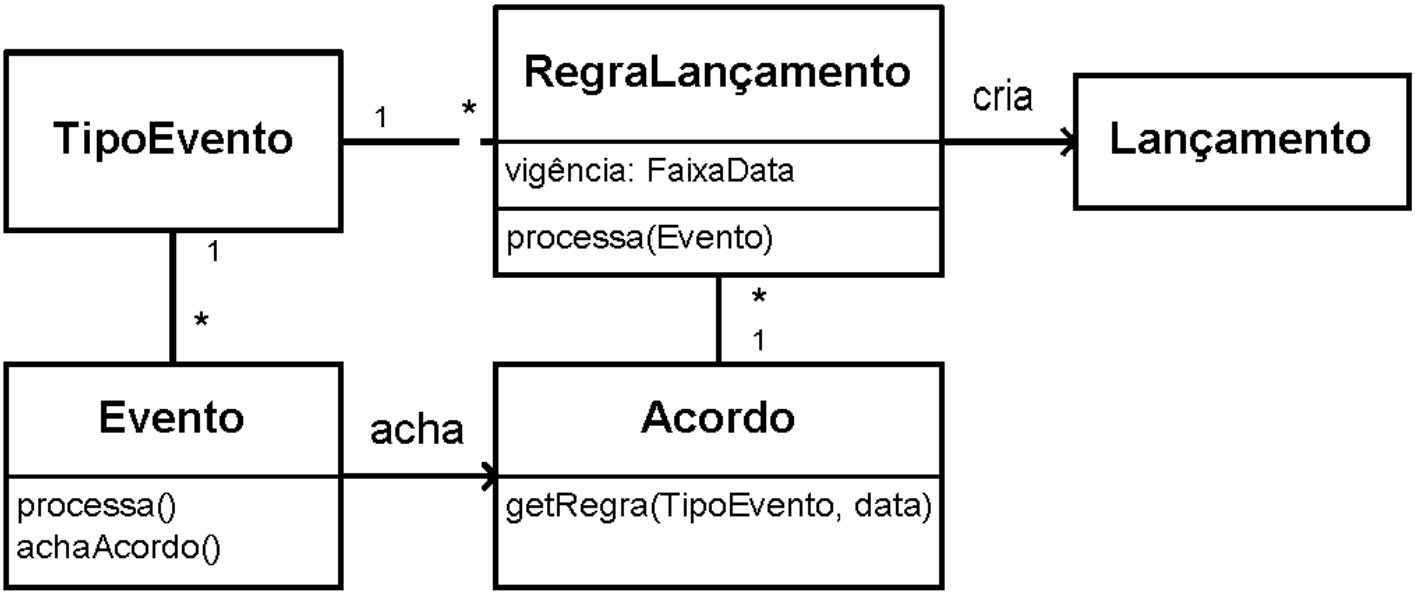
Fatorando situações duplicadas ...

- Com regras de lançamento sendo objetos diferentes dentro de acordos, podemos lidar com o compartilhamento de regras entre acordos
    - Basta alterar a cardinalidade entre Acordo e RegraLançamento para "* *"
    - Teremos que ter mecanismos para verificar em que acordos cada regra pertence para verificar como alterações às regras podem afetar múltiplos acordos

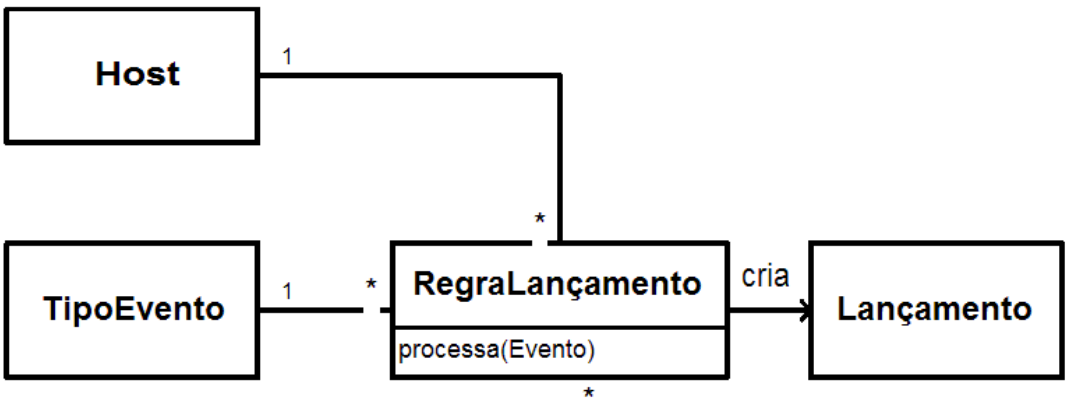
Lidando com tempo ...

- A maior causa de variações na regras diz respeito ao tempo
    - As regras em si podem mudar com tempo!
    - Pode ser uma taxa que muda ou o algoritmo inteiro de cálculo dos valores dos lançamentos
    - Por exemplo, a partir de uma certa data, horas extras podem iniciar depois de 7 horas de trabalho diário em vez de 8
- A coisa importante a sacar é que *não podemos simplesmente alterar a regra*!
    - Podemos saber da existência de um evento com atraso e ter que calcular horas extras de acordo com uma regra de um mês passado
- A forma de resolver isso é de uma o padrão Effectivity Period (Período de Vigência)
    - Este padrão registra uma faixa de datas durante a qual um objeto está em vigor

    - Cada regra recebe um Período de Vigência para tratar do comportamento dependente de tempo
- O acordo deve poder escolher a regra de acordo com TipoEvento e data



- No padrão geral, o que chamamos Acordo acima seria o "Host"
    - Veja o padrão geral de Regra de Lançamento abaixo

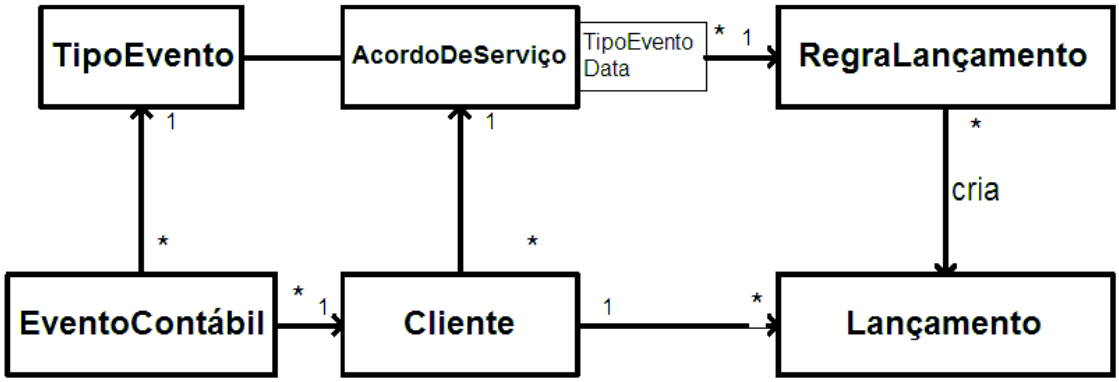


Quando deve ser usado

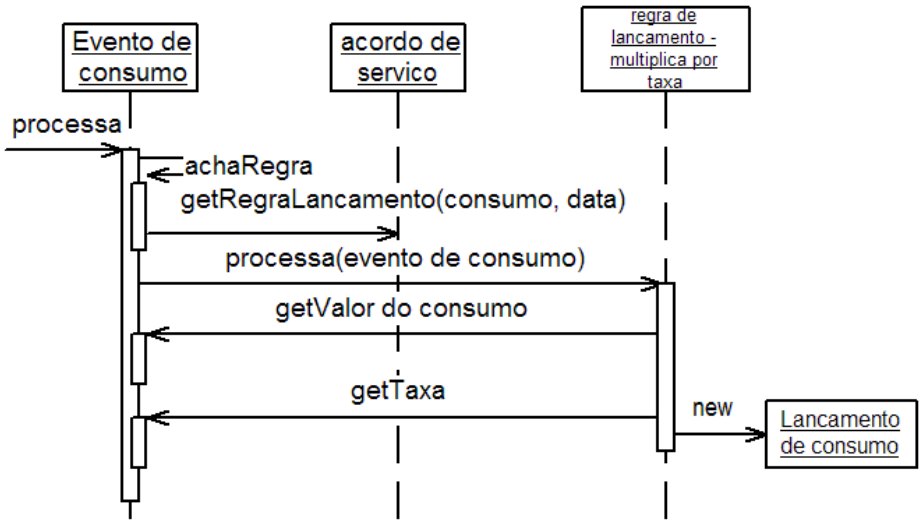
- Forma flexível de organizar lógica que reage a eventos
- Flexibilidade importante quando há fatores voláteis que afetam a lógica com freqüência
    - Se houver apenas uns 2 casos e eles não mudam com tempo, pode-se simplificar o design e usar métodos com if-then-else
    - Quando as situações complicam, é melhor "reificar as mudanças"

Código exemplo

- Vamos considerar uma empresa de distribuição de eletricidade
- Muitos eventos existem (chamada de serviço, instalação de novo equipamento, pagamento de multa, etc.)
    - Alguns podem mudar com a legislação
    - Outros podem mudar com a assinatura de determinados níveis de serviço, etc.
- Vamos nos concentrar no consumo de energia
- Aqui está um modelo básico para o faturamento do cliente



- Neste exemplo, o AcordoDeServiço é o Host da Regra de Lançamento
- Antes de criarmos o código, veja o diagrama de seqüência do que criaremos abaixo:



- No código, iniciamos com o EventoContábil e o TipoEvento

```
class EventoContabil {
    private TipoEvento tipo;
    private Calendar quandoOcorreu;
    private Calendar quandoObservado;
    private Cliente cliente;
    private Set lancamentosResultantes = new HashSet();

    EventoContabil (TipoEvento tipo, Calendar quandoOcorreu,
                    Calendar quandoObservado, Cliente cliente) {
        this.tipo = tipo;
        this.quandoOcorreu = quandoOcorreu;
        this.quandoObservado = quandoObservado;
        this.cliente = cliente;
    }

    Cliente getCliente() {
        return cliente;
    }

    TipoEvento getTipoEvento(){
        return tipo;
    }

    Calendar getQuandoObservado(){
        return quandoObservado;
    }

    Calendar getQuandoOcorreu(){
        return quandoOcorreu;
    }
```

```
    void addLancamentoResultante(Lancamento lancamento){
        lancamentosResultantes.add(lancamento);
    }

    RegraLancamento achaRegra() { /* discutido depois */}

    void processa() { /* discutido depois */}
}

class TipoEvento implements ObjetoNomeavel {
    public static TipoEvento CONSUMO = new TipoEvento("consumo");
    public static TipoEvento CHAMADA = new TipoEvento("chamada serviço");

    public TipoEvento(String nome){
        super(nome);
    }
}
```

- Agora, Lancamento e TipoLancamento

```
class Lancamento {
    private Calendar data;
    private TipoLancamento tipo;
    private Money valor;

    public Lancamento (Money valor, Calendar data, TipoLancamento tipo) {
        this.valor = valor;
        this.data = data;
        this.tipo = tipo;
    }

    public Money getValor() {
        return valor;
    }

    public Calendar getData(){
        return data;
    }

    public TipoLancamento getTipo(){
        return tipo;
    }
}

class TipoLancamento implements ObjetoNomeavel {
    static TipoLancamento CONSUMO_BASICO =
                                new TipoLancamento("Consumo Básico");
    static TipoLancamento SERVICO =
                                new TipoLancamento("Taxa de serviço");

    public TipoLancamento(String nome){
        super(nome);
    }
}
```

- O Cliente é bastante simples:

```
class Cliente implements ObjetoNomeavel {
    private AcordoServico acordoServico;
    private List lancamentos = new ArrayList();

    Cliente (String nome){
        super(nome);
    }

    public void addLancamento (Lancamento lancamento){
        lancamentos.add(lancamento);
    }

    public List getLancamentos(){
        return Collections.unmodifiableList(lancamentos);
    }

    public AcordoServico getAcordoServico(){
```

```
        return acordoServico;
    }

    public void setAcordoServico(AcordoServico acordo){
        acordoServico = acordo;
    }
}
```

- O AcordoServico é o Host para as Regras de Lançamento
- Também mantém a taxa para este acordo
- Assim, acordos diferentes podem ter taxas diferentes e combinações diferentes de Regras de Lançamentos

```
class AcordoServico {
    private double taxa;
    private Map regrasLancamento = new HashMap();

    void addRegraLancamento (TipoEvento tipoEvento,
                             RegraLancamento regra,
                             Calendar vigencia){
        if (regrasLancamento.get(tipoEvento) == null) {
            regrasLancamento.put(tipoEvento, new TemporalCollection());
        }
        temporalCollection(tipoEvento).put(vigencia, regra);
    }

    RegraLancamento getRegraLancamento(TipoEvento tipoEvento, Calendar quando){
        return (RegraLancamento)temporalCollection(tipoEvento).get(quando);
    }

    private TemporalCollection temporalCollection(TipoEvento tipoEvento){
        TemporalCollection resultado =(TemporalCollection)regrasLancamento.get(tipoEvento);
        assert(resultado != null);
        return resultado;
    }

    public double getTaxa(){
        return taxa;
    }

    public void setTaxa(double novaTaxa){
        taxa = novaTaxa;
    }
}
```

- A parte menos óbvia acima é que a coleção de regras de lançamentos é implementada usando o padrão TemporalProperty em que uma coleção retorna um valor dada uma data
- A RegraLancamento é abstrata, porque o cálculo do valor será diferente nas subclasses:

```
abstract class RegraLancamento {
    protected TipoLancamento tipo;

    protected RegraLancamento (TipoLancamento tipo){
        this.tipo = tipo;
    }

    private void facaLancamento(EventoContabil evento, Money valor){
        Lancamento novoLancamento = new Lancamento (valor, evento.getQuandoObservado(), tipo);
        evento.getCliente().addLancamento(novoLancamento);
        evento.addLancamentoResultante(novoLancamento);
    }

    public void processa (EventoContabil evento){
        facaLancamento(evento, calculaValor(evento));
    }

    abstract protected Money calculaValor(EventoContabil evento);
}
```

- Agora, vamos usar o que fizemos

Como usar as classes para lidar com o consumo

- Um cliente usou uma certa quantidade de energia elétrica
- A taxa cobrada é fixa, por consumo
- Temos que guardar a energia consumida no evento e temos que processar o evento com uma regra apropriada
    - Usaremos uma subclasse de Evento

```
public class Consumo extends EventoContabil {
    private Quantidade valor;

    public Consumo(Quantidade valor, Calendar quandoOcorreu,
                   Calendar quandoObservado, Cliente cliente){
        super(TipoEvento.CONSUMO, quandoOcorreu, quandoObservado, cliente);
        this.valor = valor;
    }

    public Quantidade getValor(){
        return valor;
    }

    double getTaxa(){
        return getCliente().getAcordoServico().getTaxa();
    }
}
```

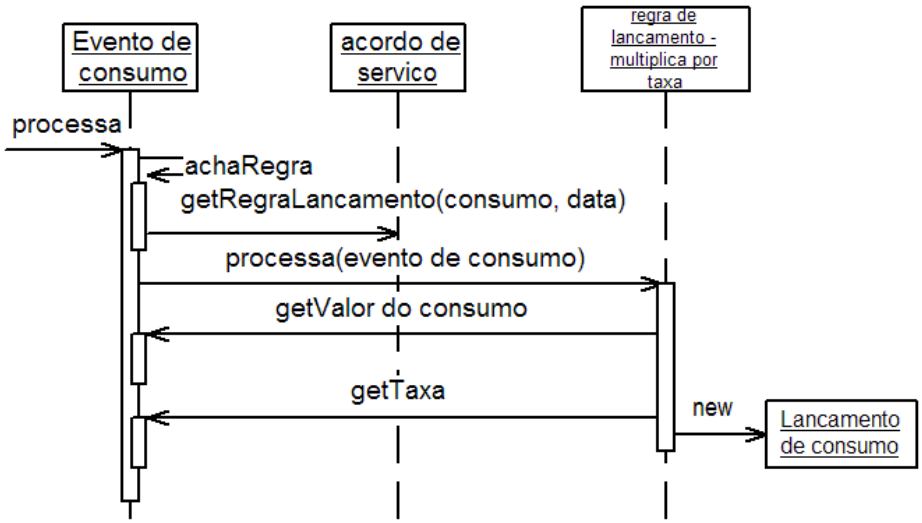
- Agora, precisamos de uma subclasse para a Regra de Lançamento

```
class RegraMultiplicaPorTaxa extends RegraLancamento {
    public RegraMultiplicaPorTaxa (TipoLancamento tipo){
        super(tipo);
    }

    protected Money calculaValor(EventoContabil evento){
        Consumo eventoDeConsumo =(Consumo)evento;
        return Money.reais(eventoDeConsumo.getValor().getValor() *
                           eventoDeConsumo.getTaxa());
    }
}
```

- Como ligar tudo isso junto para que funcione?
- O processo inicia com o evento ao qual se pede que se processe
    - "Processar" significa achar a regra certa e pedir à regra para realizar o processamento

```
class EventoContabil {
    public void processa(){
        achaRegra().processa(this);
    }

    RegraLancamento achaRegra(){
        RegraLancamento regra =
            cliente.getAcordoServico().getRegraLancamento(
                                                this.getTipoEvento(),
                                                this.quandoOcorreu);
        assert(regra != null);
        return regra;
    }
...
```

- Veja novamente o diagrama de seqüência abaixo:



- Para configurar o sistema, criamos os objetos principais e criamos os relacionamentos apropriados:

```
public void configuraClienteNormal (){
    cam = new Cliente("Cafe A Margot");
    AcordoServico padrao = new AcordoServico();
    padrao.setTaxa(10);
    padrao.addRegraLancamento(
            TipoEvento.CONSUMO,
            new RegraMultiplicaPorTaxa(TipoLancamento.CONSUMO_BASICO),
            criaCalendar(2003,1,1));
    cam.setAcordoServico(padrao);
}
...
```

- Agora, podemos criar um evento e processá-lo:

```
public void testConsumo(){
    Consumo evento = new Consumo(
            Unit.KWH.valor(50),
            criaCalendar(2003,10,25),
            criaCalendar(2003,10,25),
            cam);
    evento.processa();
    Lancamento lancamentoResultante = getLancamento(cam, 0);
    assertEquals (Money.reais(500), lancamentoResultante.getValor());
}
```

- Embora tudo isso pareça um pouco complicado, é extremamente flexível, pois mudanças de regras são tratadas com pouco código adicional e com facilidade
  - Isso é OO!
  - Melhor se acostumar!
- Vamos tentar fazer mudanças ao que já fizemos para verificar a facilidade de manutenção ...

Um segundo tipo de evento

- Vamos ver como tratar uma chamada de serviço, cujo custo é normalmente um valor fixo de base
- O acordo com o cliente pode mudar este valor fixo
  - No nosso exemplo, será metade da taxa de base mais 10 reais
- Precisamos de:
  - Uma subclasse de Evento
  - Um novo TipoEvento
  - Uma nova RegraLancamento
- Iniciamos com o Evento

```
class EventoMonetario extends EventoContabil {
```

```
    Money valor;

    EventoMonetario(Money valor, TipoEvento tipo, Calendar quandoOcorreu,
                    Calendar quandoObservado, Cliente cliente) {
        super(tipo, quandoOcorreu, quandoObservado, cliente);
        this.valor = valor;
    }

    public Money getValor(){
        return valor;
    }
}
```

- Observe acima que esta classe pode servir para vários eventos monetários, já que passamos o tipo do evento no construtor
- Poderíamos criar este evento como segue:

```
public void testServico(){
    EventoContabil evento = new EventoMonetario(
                                    Money.reais(40),
                                    TipoEvento.CHAMADA,
                                    criaCalendar(2003,10,25),
                                    criaCalendar(2003,10,25),
                                    cam);
    evento.processa();
    Lancamento lancamentoResultante =(Lancamento)cam.getLancamentos().get(0);
    assertEquals (Money.reais(30), lancamentoResultante.getValor());
}
```

- De forma semelhante, a Regra de Lançamento é genérica e aplica uma fórmula simples a um evento monetário:

```
class RegraFormulaSimples extends RegraLancamento {
    private double multiplicador;
    private Money valorFixo;

    RegraFormulaSimples (double multiplicador, Money valorFixo, TipoLancamento tipo){
        super (tipo);
        this.multiplicador = multiplicador;
        this.valorFixo = valorFixo;
    }
    protected Money calculaValor(EventoContabil evento){
        Money valorDoEvento =((EventoMonetario)evento).getValor();
        return (Money)valorDoEvento.multiply(multiplicador).add(valorFixo);
    }
}
```

- Adicionamos esta regra ao acordo de serviço assim:

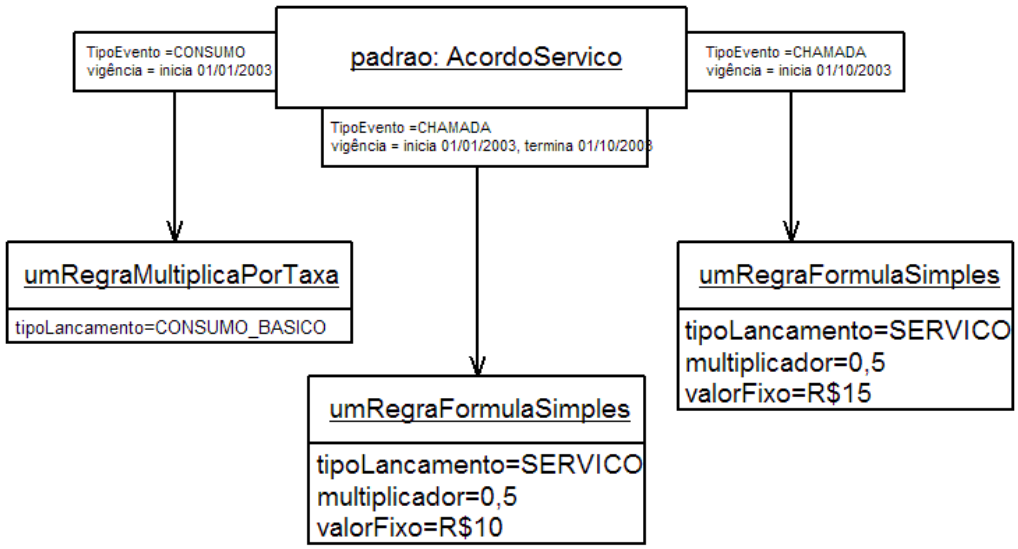
```
public void configuraClienteNormal (){
    cam = new Cliente("Cafe A Margot");
    AcordoServico padrao = new AcordoServico();
    padrao.setTaxa(10);
    padrao.addRegraLancamento(
            TipoEvento.CONSUMO,
            new RegraMultiplicaPorTaxa(TipoLancamento.CONSUMO_BASICO),
            criaCalendar(2003,1,1));
    padrao.addRegraLancamento(
            TipoEvento.CHAMADA,
            new RegraFormulaSimples(0.5,
                                    Money.reais(10),
                                    TipoLancamento.SERVICO),
            criaCalendar(2003,1,1));
    cam.setAcordoServico(padrao);
}
...
```

Lidando com uma mudança de regra

- Se as regras mudarem com tempo, podemos adicionar uma nova regra usando a natureza temporal das regras
- Por exemplo, podemos incluir uma alteração no custo da chamada de serviço:

```
public void configuraClienteNormal (){
    cam = new Cliente("Cafe A Margot");
    AcordoServico padrao = new AcordoServico();
    padrao.setTaxa(10);
    padrao.addRegraLancamento(
            TipoEvento.CONSUMO,
            new RegraMultiplicaPorTaxa(TipoLancamento.CONSUMO_BASICO),
            criaCalendar(2003,1,1));
    padrao.addRegraLancamento(
            TipoEvento.CHAMADA,
            new RegraFormulaSimples(0.5,
                                    Money.reais(10),
                                    TipoLancamento.SERVICO),
            criaCalendar(2003,1,1));
    padrao.addRegraLancamento(
            TipoEvento.CHAMADA,
            new RegraFormulaSimples(0.5,
                                    Money.reais(15),
                                    TipoLancamento.SERVICO),
            criaCalendar(2003,10,1));
    cam.setAcordoServico(padrao);
}
...
```

- Em UML, temos a seguintes situação (observe o uso de Qualificadores)



- Se fizermos uma chamada de serviço depois de 01/10/2003, usaremos a nova fórmula, como mostra a teste a seguir:

```
public void testServicoDepoisDaMudanca(){
    EventoContabil evento = new EventoMonetario(
                                Money.reais(40),
                                TipoEvento.CHAMADA,
                                criaCalendar(2003,10,5),
                                criaCalendar(2003,10,15),
                                cam);
    Evento.processa();
    Lancamento lancamentoResultante = (Lancamento)cam.getLancamentos().get(0);
    assertEquals (Money.reais(35), lancamentoResultante.getValor());
}
```

Um segundo acordo ...

- Da mesma forma que é simples adicionar regras de lançamento, é simples desenvolver novos acordo
- Vamos supor que o governo exija acordos especiais para pessoas de baixa renda
    - Se o consumo for menor do que 50 KWh, então, a taxa de consumo é menor
- Aqui está a regra de lançamento:

```
class RegraBaixaRenda extends RegraLancamento {
    double taxa;
    Quantidade limiteDeConsumo;

    RegraBaixaRenda (TipoLancamento tipo, double taxa, Quantidade limiteDeConsumo) {
        super(tipo);
        this.taxa = taxa;
        this.limiteDeConsumo = limiteDeConsumo;
    }

    protected Money calculaValor(EventoContabil evento){
        Consumo eventoDeConsumo = (Consumo)evento;
        Quantidade consumoAtual = eventoDeConsumo.getValor();
        return consumoAtual.isGreaterThan(limiteDeConsumo) ?
               Money.reais(consumoAtual.getValor() * eventoDeConsumo.getTaxa()) :
               Money.reais(consumoAtual.getValor() * this.taxa);
    }
}
```

- Aqui está um cliente que usa essa regra

```
 private void configuraClienteBaixaRenda() {
        zé = new Cliente("José Severino da Silva");
        AcordoServico baixaRenda = new AcordoServico();
        baixaRenda.setTaxa(10);
        baixaRenda.addRegraLancamento(
                TipoEvento.CONSUMO,
                new RegraBaixaRenda(
                        TipoLancamento.CONSUMO_BASICO,
                        5,
                        new Quantidade(50, Unit.KWH)),
                criaCalendar(2003, 1, 1));
        baixaRenda.addRegraLancamento(
                TipoEvento.CHAMADA,
                new RegraFormulaSimples(
           0,
           Money.reais(10),
           TipoLancamento.SERVICO),
                criaCalendar(2003, 10, 1));
        zé.setAcordoServico(baixaRenda);
}
```

- A fatura de Zé vai depender de quanto ela consome:

```
public void testConsumoBaixaRenda(){
    Consumo evento = new Consumo(
                    Unit.KWH.valor(50),
                    criaCalendar(2003,10,1),
                    criaCalendar(2003,10,1),
                    zé);
    evento.processa();
    Consumo evento2 = new Consumo(
                    Unit.KWH.valor(51),
                    criaCalendar(2003,10,2),
                    criaCalendar(2003,10,2),
                    zé);
    evento2.processa();
    Lancamento lancamentoResultante1 = (Lancamento)zé.getLancamentos().get(0);
    assertEquals (Money.reais(250), lancamentoResultante1.getvalor());
    Lancamento lancamentoResultante2 = (Lancamento)zé.getLancamentos().get(1);
    assertEquals (Money.reais(510), lancamentoResultante2.getvalor());
}
```

programa