Disciplina: Linguagem de Programação II
Professor: Raimundo Osvaldo Vieira

Estudo de Caso 04

Sistema de Folha de Pagamento

Esse estudo de caso foi proposto originalmente pela professora Cecília Rubira (UNICAMP)

Descrição Geral do Domínio da Aplicação

A aplicação que iremos desenvolver se destina a uma empresa comercial que tem dois tipos de funcionários: auxiliares administrativos, que recebem como mensalistas, e gerentes e vendedores, que são comissionados.

A empresa mantém também convênio com uma cooperativa de médicos, através do qual os funcionários podem contratar planos de assistência médica, com diferentes opções de cobertura, cujas mensalidades são debitadas da folha de pagamento do funcionário. Atualmente a cooperativa informa mensalmente, para cada funcionário, os valores a serem debitados ou reembolsados, que são lançados na folha manualmente. Para eliminar esse trabalho a cooperativa se propõe a integrar o seu sistema de cobrança com o sistema da folha, para que se possa efetuar esses lançamentos automaticamente. A empresa pretende firmar outros convênios como esse, o que deve ser previsto na aplicação.

Descrição dos Requisitos Funcionais e das Regras de Negócio

A remuneração de um mensalista é o seu salário base com variações, para mais ou para menos, em função de eventuais horas extras, atrasos ou faltas, da seguinte forma.

1. para efeitos de cálculo, considera-se o valor da hora igual a 1/176 do salário base;
2. as horas extras são pagas em dobro, ou seja: cada hora extra corresponde a 2/176 do salário base;
3. atrasos ou faltas são descontados com base no valor da hora, considerando-se um dia de falta igual a 8 horas;
4. um funcionário não pode ter mais de 4 horas extras ou mais de 2 de atraso, num mesmo dia.

Um funcionário comissionado recebe, além do salário base, comissões pelas vendas efetuadas, não havendo controle de ponto para o mesmo.

A aplicação deverá permitir o registro diário dos eventos mencionados (horas extras, atrasos, altas e créditos de comissões), além de eventuais reajustes salariais e desligamentos de funcionários, e a emissão do demonstrativo de pagamento dos funcionários, com todos os descontos legais (imposto de renda na fonte, INSS, contribuição sindical, etc.).

Análise e Projeto do Sistema

A partir da descrição acima podemos identificar os seguintes objetos que fazem parte da aplicação:

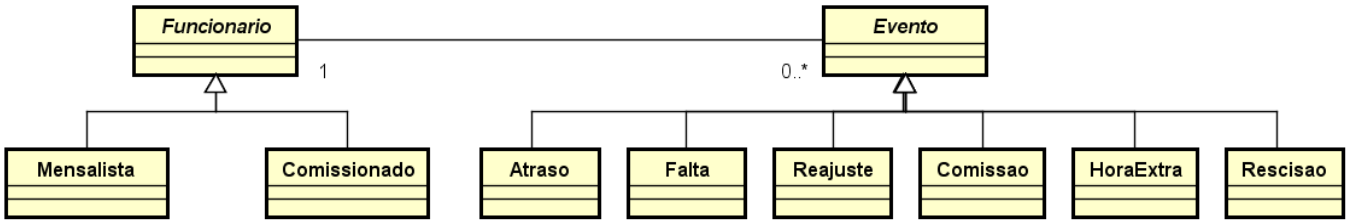
1. funcionários, que se dividem em auxiliares administrativos, gerentes ou vendedores;
2. mensalistas, que são os auxiliares administrativos;
3. comissionados, que são os gerentes e vendedores;

4. eventos que afetam a remuneração dos funcionários, tais como: horas extras, faltas, atrasos, créditos de comissões, reajustes salariais e desligamentos;
5. convênios, tais como o convênio médico;
6. contratos de adesão dos funcionários aos convênios, que preveem lançamentos diretamente na folha de pagamento;
7. demonstrativos de pagamento;
8. descontos legais, como imposto de renda na fonte, INSS e contribuição sindical.

Um funcionário tem que ser mensalista ou comissionado. As diferenças de comportamento entre esses dois tipos de funcionários não permite definir uma relação de super tipo / subtipo entre os mesmos, embora compartilhem de diversas propriedades comuns: ambos são admitidos, reagem a alguns eventos comuns, como reajustes salariais e desligamentos e recebem uma remuneração mensal. Iremos, portanto, definir uma classe abstrata `Funcionario` que será a raiz de uma hierarquia com duas subclasses: `FuncionarioMensalista` e `FuncionarioComissionado`.

Os eventos podem ser tratados de maneira semelhante. Podemos definir uma classe abstrata `Evento` que será a raiz de uma hierarquia com subclasses específicas para cada tipo de evento: `EventoHoraExtra`, `EventoAtraso`, `EventoFalta`, `EventoComissao`, `EventoReajuste` e `EventoRescisão`.

A hierarquia de classes está descrita no diagrama abaixo:

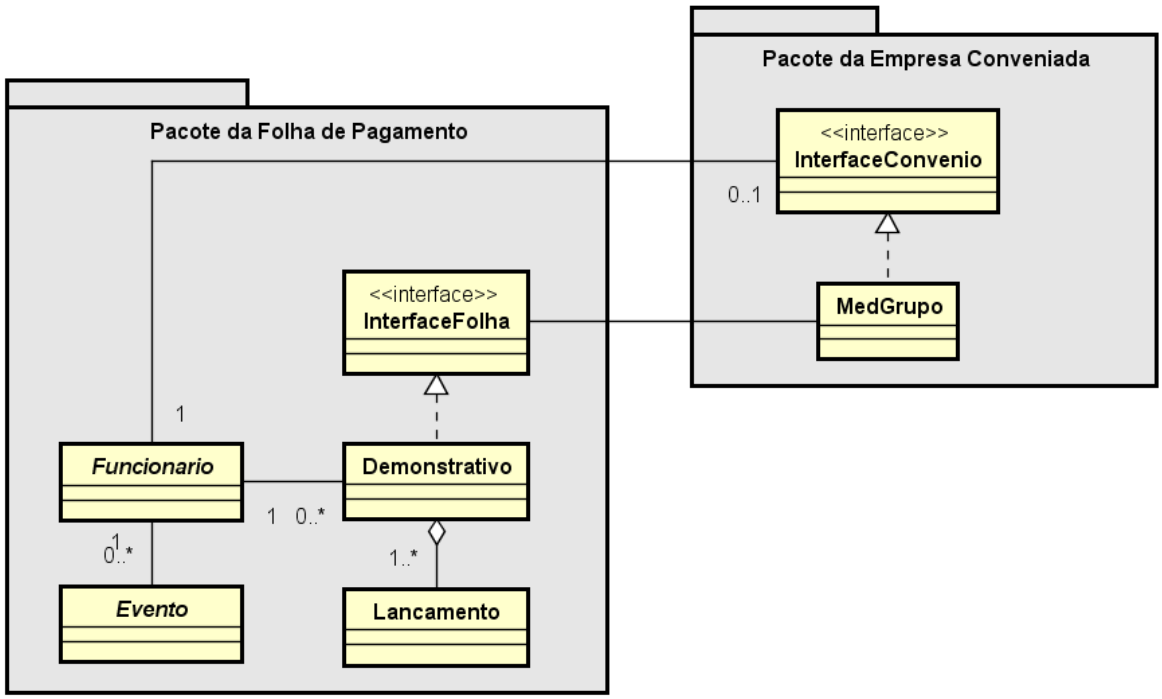


Os convênios e seus contratos são administrados pelas empresas contratadas. O que se deseja é uma forma de integração com os sistemas dessas empresas que permita a inclusão de lançamentos na folha de uma forma segura, sem comprometer o funcionamento do sistema da folha e mantendo o sigilo das informações mantidas pelo mesmo. É desejável também que convênios futuros possam se integrar da mesma forma, sem que seja preciso alterar o sistema da folha.

Para isso iremos definir duas interfaces: uma para o sistema da folha (`InterfaceFolha`) e outra para os sistemas das administradoras dos convênios (`InterfaceConvenio`). Nessas interfaces iremos definir apenas aquelas as operações essenciais para a integração entre os sistemas, de forma a proteger cada um dos sistemas contra acessos indevidos por parte do outro sistema. Do lado do sistema da folha, a `InterfaceFolha` pode ser implementada pelos objetos que manipulam os demonstrativos, já que são o alvo dessa integração. Iremos, portanto, definir uma classe Demonstrativo que irá implementar a interface `InterfaceFolha` e que irá conter os lançamentos da folha (classe `Lancamento`). Do lado do sistema da empresa que administra os contratos, é suficiente que exista alguma classe implementando a interface `InterfaceConvenio`. Nesse exemplo a classe `MedGrupo` irá simular uma classe do sistema do convênio médico, cuja implementação real ficaria a cargo da cooperativa que mantém o convênio.

A seguir, é mostrado o diagrama de classes com as interfaces e classes envolvidas na integração entre os sistemas. Para não sobrecarregar o diagrama as subclasses de `Usuario` e `Eventos` não estão representadas.

Estão representados dois pacotes (*packages*) independentes: o da **folha de pagamento** e o **da administradora do convênio**. Em cada um desses pacotes há uma interface pública que permite às classes de outro pacote o acesso a um conjunto limitado de serviços fornecidos pelas suas classes. Note que, para simplificar o nosso estudo de caso, estamos considerando que um funcionário pode estar associado a um único convênio, ou a nenhum.



Os descontos legais podem ser tratados de forma parecida com os convênios, já que se comportam essencialmente da mesma forma: são administrados de maneira independente, segundo critérios estabelecidos pelo governo. As principais diferenças são: contratos compulsórios e suas operações podem requerer informações que não ficam acessíveis para os convênios em geral. O primeiro ponto é resolvido com a inclusão automática desses "contratos" no momento do registro do funcionário, por exemplo. O segundo ponto pode ser resolvido incorporando as classes responsáveis por esses "convênios" ao mesmo pacote (*package*) da folha e deixando fora dele as classes responsáveis pelos convênios comuns. Podemos, então, criar métodos que permitam o acesso àquelas informações privilegiadas, com visibilidade apenas para as classes do pacote da folha.

Além das classes acima descritas, iremos definir uma classe de exceções (`FolhaException`) que será utilizada para sinalizar qualquer tipo de ocorrência, nas operações da folha, que signifiquem um desvio do comportamento normal do sistema, como por exemplo o registro de um evento para um funcionário que já esteja desligado da empresa.

Definição das Classes

- **Classe `Evento` e suas subclasses**

  Um evento qualquer possui no mínimo dois atributos: o tipo do evento e a data em que ocorreu. Na maior parte dos casos um evento possui também um valor associado, como o número de horas extras ou o valor da comissão. Um evento pode ter ainda atributos específicos de sua classe, como no caso dos eventos da classe `EventoRescisao` que possuem atributos como o motivo da rescisão e se houve aviso prévio.

  Na classe abstrata `Evento`, que engloba todos os eventos possíveis, iremos definir apenas dois atributos de atributo: um para a data do evento (`dtEvento`) e outro para seu valor (`valorEvento`), quando houver. O tipo do evento pode ser identificado pela classe concreta do objeto que o representa. Atributos específicos de uma subclasse serão definidos na própria subclasse.

  Um objeto da classe `Evento` serve, essencialmente, para registrar a ocorrência de um evento e permitir que objetos de outras classes recuperem os atributos desse evento. Para isso são necessárias apenas operações que registarem um evento e leiam os seus atributos, conforme descrito a seguir.

  O registro de um evento pode ser realizado pelo construtor da classe, que deverá garantir a consistência dos seus atributos, tal como o limite máximo de 4 horas para um evento `HoraExtra`.

  Para obtenção dos atributos do evento definiremos as operações `getDtEvento()`, `getValorEvento()` e `getTipoEvento()`. Iremos definir ainda dois métodos auxiliares visando

apenas simplificar a utilização da classe evento pelas outras classes: o método `equals`() que irá comparar os atributos de dois eventos para verificar se são duplicados (mesma data e mesmo tipo), e o método `isTipo()` que verifica se o evento é de um tipo especificado.

Finalmente, assim como para todas as classes, iremos definir o método `toString()` que deve fornecer um `String` com uma representação do evento, que possa ser impressa ou exibida no terminal.

Definimos, assim, a seguinte interface pública para a classe `Evento`

```
public Evento(Date dt, float val);
public Date getDtEvento();
public float getValorEvento();
public String getTipoEvento();
public boolean isTipo(String st);
public boolean equals(Evento e);
public String toString();
```

Com exceção do método construtor, que é exclusivo da classe onde é definido, os demais métodos serão todos herdadas pelas subclasses de Evento. Para cada uma dessas subclasses deverá, portanto, ser definido apenas o seu construtor e outros métodos específicos da classe, se necessário.

- **Classe `Funcionario` e suas subclasses**

Nesse nosso estudo de caso iremos nos restringir aos atributos de um funcionário que são essenciais para o cálculo da folha, a seguir relacionados.

`nomeFunc` e `dtAdmissao`: nome do funcionário e data de admissão;

`salarioBase` e `dtSalario`: salário base atual e respectiva data, que será igual à data de admissão, se ainda não teve reajuste de salário, ou à data do último reajuste salarial;

`dtFechamento`: data de encerramento do período anterior, que será igual à data de admissão, se recém admitido, ou à data de emissão do último demonstrativo de pagamento;

`dtRescisao`: data do afastamento da empresa, se funcionário já não está na empresa, ou 'null' se funcionário permanece ativo;

`eventos`: lista de eventos ocorridos com o funcionário, desde o seu ingresso na empresa;

`demonstrativos`: lista de demonstrativos de pagamento já emitidos para o funcionário;

`convenio` e `idContrato`: convênio contratado pelo funcionário e sua identificação nesse convênio, podendo ser ambos nulos caso o funcionário não tenha contratado nenhum convênio.

Todos esses atributos são comuns a qualquer tipo de funcionário e serão definidos na classe `Funcionario`.

As operações previstas para um funcionário são: registro da admissão do funcionário, inclusão em convênio, registro de eventos diversos, como horas extras, comissões e demissão, e cálculo e emissão do demonstrativo de pagamento. Não iremos considerar operações relacionadas com férias, 13º salário e inúmeras outras que deveriam ser previstas numa situação real.

O registro da admissão de um funcionário pode ser deixado a cargo do método construtor da classe. As operações de inclusão em convênio e registro de eventos se limitam a armazenar as informações correspondentes, que serão utilizadas no cálculo do demonstrativo de pagamento. A operação de cálculo e emissão do demonstrativo processa todos os eventos ocorridos após o encerramento do período anterior e gera um novo demonstrativo, encerrando um novo período.

Definimos, assim, a seguinte interface pública para a classe `Funcionario`:

```
public Funcionario (String nome, Date dt, float sal);
public void registraConvenio(InterfaceConvenio c, String id);
public void registraEvento (Evento e);
```

```
public void geraDemonstrativo();
```

- **As Interfaces de Integração**

  Como definido no projeto das classes, existirão duas interfaces para integração dos sistemas: a `InterfaceConvenio`, que define as operações a serem realizadas pelo sistema da empresa que mantém o convênio, e a `InterfaceFolha`, que define as operações a serem realizadas pelo sistema da folha de pagamento, conforme o esquema descrito a seguir.

  Ao gerar um novo demonstrativo de pagamento para um funcionário que esteja associado a um convênio o sistema da folha irá requisitar ao sistema do convênio a geração dos lançamentos a serem incluídos naquele demonstrativo. Para isso a `InterfaceConvenio` deverá prever uma operação `processaContrato()` que receberá como parâmetros a identificação do funcionário no convênio e o demonstrativo onde serão incluídos os lançamentos. Para gerar esses lançamentos o sistema do convênio deverá poder obter o período a que se refere o demonstrativo e ser capaz de incluir no demonstrativo novos lançamentos, tanto a débito como a crédito. Para isso, a `InterfaceFolha` irá oferecer as operações `getDtInicial()`, `getDtFinal()`, `incluiDebito()` e `incluiCredito()`.

  Com isso definimos essas duas interfaces, conforme a seguir.

```
public interface InterfaceConvenio {
    void processaContrato(String id, InterfaceFolha f)
                                throws Exception;
}
```

  Note que as operações da `InterfaceConvenio` estão definidas com a cláusula `throws Exception`. Queremos, com isso, obrigar que os métodos que utilizem essas operações incluam um tratamento para qualquer exceção que possa ocorrer ao executa-las. De outra forma, eventuais falhas no sistema do convênio poderiam causar a interrupção do processamento da folha.

```
import java.util.Date;
public interface InterfaceFolha {
   Date getDtInicial() throws Exception;
   Date getDtFinal() throws Exception;
   void incluiDebito(String hist, float val) throws Exception;
   void incluiCredito(String hist, float val) throws Exception;
}
```

  No sentido inverso, as operações da interface da folha também preveem a possibilidade de ocorrer algum tipo de exceção, o que deve ser previsto no sistema do convênio.

- **As Classes `Demonstrativo` e `Lancamento`**

  Os atributos básicos de um demonstrativo são: as datas de início e término do período a que e refere e um conjunto de lançamentos. Um lançamento, por sua vez, possui um histórico e um valor, que pode ser positivo ou negativo.

  De acordo com o projeto das classes, a classe `Demonstrativo` implementa a `InterfaceFolha`. Isso quer dizer que todas as operações previstas na `InterfaceFolha` devem fazer parte da interface pública da classe `Demonstrativo`. Além dessas operações, e do construtor da classe, iremos definir também: uma operação `imprime()`, para imprimir o demonstrativo, e a operação `toString()`.

  Com isso definimos a seguinte interface pública para a classe `Demonstrativo`:

```
public Demonstrativo(Funcionario f, Date inicio, Date fim);
public Date getDtInicial();
public Date getDtFinal();
public void incluiDebito(String hist, float val)throws Exception;
public void incluiCredito(String hist, float val)throws Exception;
```

```
public void imprime();
public String toString();
```

Os objetos da classe Lançamento apenas armazenam os atributos dos lançamentos incluídos num demonstrativo. As únicas operações previstas são, portanto, o construtor da classe, as operações para se recuperar os atributos e a operação toString(), conforme a interface seguinte:

```
public Lancamento (String hist, float val);
public String getHistorico();
public float getValor();
public String toString();
```

Implementação do Sistema

A seguir iremos apresentar o código completo de cada uma das classes da aplicação seguido de comentários relativos aos aspectos particulares de sua implementação, quando houver.

A classe `FolhaException`

```
public class FolhaException extends Exception {
	private String descricao;
	private Funcionario funcionario;
	private Evento evento;
	private Exception exception;

	public FolhaException(String st) {
		this.descricao = st;
	}

	public FolhaException(String st, Funcionario f) {
		this.descricao = st;
		this.funcionario = f;
	}

	public FolhaException(String st, Funcionario f, Evento e) {
		this.descricao = st;
		this.funcionario = f;
		this.evento = e;
	}

	public FolhaException(String st, Funcionario f, Exception ex) {
		this.descricao = st;
		this.funcionario = f;
		this.exception = ex;
	}

	public String toString() {
		return ("FuncionarioException: " + this.descricao +
			(this.funcionario == null ? " " : "\n " + this.funcionario) +
			(this.evento == null ? " " : "\n " + this.evento) +
			(this.exception == null ? " " : "\n " + this.exception));
	}
}
```

A descrição da exceção é armazenada no atributo `descricao`, do tipo `String`. Opcionalmente podem ser armazenadas também referências para um funcionário, um evento ou uma outra exceção. Para simplificar o lançamento das exceções foram definidos quatro métodos construtores, para diferentes

combinações desses atributos: apenas o histórico, histórico + funcionário, histórico + funcionário + evento e, por último, histórico + exceção. Esse é um exemplo de polimorfismo de construtor.

A classe `Evento`

```
public abstract class Evento {
	private Date dtEvento;
	private float valorEvento;
	private int dia;
	private int mes;
	private int ano;

	public Evento(Date dt, float val) {
		this.dtEvento = dt;
		Calendar cal = new GregorianCalendar();
		cal.setTime(dtEvento);
		this.dia = cal.get(Calendar.DATE);
		this.mes = cal.get(Calendar.MONTH) + 1;
		this.ano = cal.get(Calendar.YEAR) + 1900;
		this.valorEvento = val;
	}

	public Date getDtEvento() {
		return this.dtEvento;
	}

	public float getValorEvento() {
		return this.valorEvento;
	}

	public String getTipoEvento() {
		return (this.getClass()).getName();
	}

	public boolean equals(Evento e) {
		return this.getTipoEvento().equals(e.getTipoEvento())
			&& this.dia == e.dia && this.mes == e.mes && this.ano == e.ano;
	}

	public String toString() {
		return getTipoEvento() + " em " + this.dia + "/" +
			this.mes + "/" + this.ano +
			" valor=" + valorEvento;
	}
}
```

O método `getTipoEvento()` retorna o nome da classe a que pertence o evento. Para isso são utilizados os métodos `getClass()` e `getName()`, descritos a seguir.

O método `getClass()` é definido na classe `Object`, que é a super classe comum a todas as classes Java. Esse método retorna um descritor da classe do objeto, como no exemplo seguinte.

```
Evento e=new EventoReajuste(...);
Class c=e.getClass();
```

A seqüência acima atribui ao atributo `c` uma referência para o descritor da classe `EventoReajuste`. Um descritor de classe é, por sua vez, um objeto da classe `Class`, cujos métodos permitem a um programa Java obter, em tempo de execução, propriedades da classe de um objeto qualquer[6]. O método

`getName()` é um deles, que retorna um `String` contendo o nome da classe, como no exemplo seguinte.

```
Evento e=new EventoReajuste(...);
Class c=e.getClass();
String st=c.getName();
System.out.println("O nome da classe e' " + st);
```

Nessa sequência o atributo `st` recebe o nome da classe `c`. Como `c` contém o descritor da classe `EventoReajuste` obtemos o seguinte resultado:

```
O nome da classe e' EventoReajuste
```

Uma alternativa a essa forma de implementação do método `getTipoEvento()` seria defini-lo na classe `Evento` como `abstract` e redefini-lo em cada uma de suas subclasses para que retorne uma constante igual ao nome da classe, conforme abaixo:

```
public class Evento {

	// ...
	public abstract String getTipoEvento();
	// ...

}
public class EventoReajuste extends Evento {

	// ...
	public String getTipoEvento() {
		return "EventoReajuste";
	}
	// ...
}
```

A classe `EventoHoraExtra`

```
import excecoes.FolhaException;
import java.util.Date;

public class EventoHoraExtra extends Evento {
	private double qtdHoras;

	public EventoHoraExtra(Date dt, double qtd) throws FolhaException {
		super(dt, qtd);
		if (qtd > 4)
			throw new FolhaException("HoraExtra com qtde. horas > 4.");
	}
}
```

Para garantir que não existirão eventos de hora extra com mais de 4 horas, foi incluído no construtor da classe o teste dessa condição, lançando uma exceção caso ocorra aquela condição. Com isso a tentativa de criação, através de um comando `new`, de um evento fora das especificações provoca uma exceção e o objeto correspondente não é criado. Com isso faz-se necessário incluir a cláusula `throws FolhaException` na definição do construtor da classe.

Como um método construtor não retorna nenhum valor, o lançamento de uma exceção é o único recurso que permite sinalizar qualquer tipo de erro durante a criação de um objeto

A classe `EventoAtraso`

```
import java.util.Date;
import excecoes.FolhaException;
```

```
class EventoAtraso extends Evento {
      public EventoAtraso(Date dt, double qtd) throws FolhaException {
            super(dt, qtd);
            if (qtd > 2)
                  throw new FolhaException("Atraso com qtde. horas > 2");
      }
}
```

A classe **EventoFalta**

```
import java.util.Date;

class EventoFalta extends Evento {
      public EventoFalta(Date dt) {
            super(dt, 0);
      }

      public String toString() {
            return super.getTipoEvento() + " em " + super.getDtEvento();
      }
}
```

A classe **EventoComissao**

```
import java.util.Date;
import excecoes.FolhaException;

class EventoComissao extends Evento {
      public EventoComissao(Date dt, double val) throws FolhaException {
            super(dt, val);
            if (val <= 0)
                  throw new FolhaException("Comissao com valor <= 0.");
      }
}
```

A classe **EventoReajuste**

```
import java.util.Date;
import excecoes.FolhaException;

class EventoReajuste extends Evento {
      public EventoReajuste(Date dt, double val) throws FolhaException {
            super(dt, val);
            if (val < 300)
               throw new FolhaException("Reajuste com novo salario < 300.");
      }
}
```

A classe **EventoRescisao**

```
import java.util.Date;
import excecoes.FolhaException;

public class EventoRescisao extends Evento {

      private int motivo;
      public static final int SE_DEMITIU = 1;
      public static final int FOI_DEMITIDO = 2;
      public static final int APOSENTADO = 3;
      private boolean avisoPrevio;
```

```
	public static boolean CUMPRIU_AVISO = true;
	public static boolean AVISO_INDENIZADO = false;

	public EventoRescisao(Date dt, int tp, boolean av)
                                                throws FolhaException{
		super(dt, 0);
		if (motivo != SE_DEMITIU && motivo != FOI_DEMITIDO &&
		    motivo != APOSENTADO){
		    throw new FolhaException("Rescisao com codigo de motivo
                                        invalido");
		}
		this.motivo = tp;
		this.avisoPrevio = av;
	}

	public int getMotivo() {
		return this.motivo;
	}

	public boolean getAvisoPrevio() {
		return this.avisoPrevio;
	}

	public String toString() {
		return super.getTipoEvento() + " em " + super.getDtEvento() +
			" motivo=" + this.motivo +
			" aviso previo= " + this.avisoPrevio;
	}
}
```

A classe `Funcionario`

```
import java.util.Date;
import java.util.ArrayList;
import java.util.Iterator;
import convenio.InterfaceConvenio;
import excecoes.FolhaException;
import folha.Demonstrativo;
import folha.eventos.Evento;

public abstract class Funcionario {
	protected String nomeFunc;
	protected double salarioBase;
	protected Date dtAdmissao;
	protected Date dtSalario;
	protected Date dtFechamento;
	protected Date dtRescisao;
	protected ArrayList<Evento> eventos;
	protected ArrayList<Demonstrativo> demonstrativos;
	protected InterfaceConvenio convenio;
	protected String idContrato;

	public Funcionario(String nome, Date dt, float sal) {
		this.nomeFunc = nome;
		this.dtAdmissao = dt;
		this.dtFechamento = dt;
		this.dtSalario = dt;
		this.salarioBase = sal;
		this.eventos = new ArrayList<Evento>();
		this.demonstrativos = new ArrayList<Demonstrativo>();
	}
```

```
public void registraEvento(Evento e) throws FolhaException {
      Date hoje = new Date();
      if (dtRescisao != null){
            throw new FolhaException("Evento para funcionario ja'
                                    desligado.", this, e);
      }
      else if (!(e.getDtEvento()).after(dtFechamento)){
            throw new FolhaException("Evento com data anterior aa do
                                     fechamento.", this, e);
      }
      else if ((e.getDtEvento()).after(hoje)){
            throw new FolhaException("Evento com data futura", this, e);
      }
      else if (eventoDuplicado(e)){
            throw new FolhaException("Evento duplicado para o
                                    funcionario.", this, e);
      }
      this.eventos.add(e);
}

private boolean eventoDuplicado(Evento e) {
      Iterator<Evento> lista = this.eventos.iterator();
      while (lista.hasNext()) {
            if (e.equals((Evento) lista.next()))
                  return true;
      }
      return false;
}

public void registraConvenio(InterfaceConvenio c, String id) {
      convenio = c;
      idContrato = id;
}

public void geraDemonstrativo() throws FolhaException {
      Evento e;
      Demonstrativo d;
      Date hoje = new Date();

      Iterator<Evento> lista = eventos.iterator();
      while (lista.hasNext()) {
            e = (Evento) lista.next();
            if ((e.getDtEvento()).after(dtFechamento)) {
                  if (!eventoComum(e))
                        try {
                              processaEvento(e);
                        } catch (FolhaException exc) {
                              System.out.println(exc);
                        }
            }
      }

      d = new Demonstrativo(this, dtFechamento, hoje);
      try {
            d.incluiCredito("Salario base", this.salarioBase);
      } catch (Exception ex) {
            throw new FolhaException("Erro ao lancar salario base",
                                    this, ex);
      }
      geraLancamentos(d);
```

```
            if (idContrato != null){
                  try {
                        convenio.processaContrato(idContrato, d);
                  } catch (Exception ex) {
                        System.out.println("Erro ao lancar convenio - contrato"
                                          + idContrato);
                  }
            }
            this.demonstrativos.add(d);
            this.dtFechamento = hoje;
            d.imprime();
      }

      private boolean eventoComum(Evento e) {
            if (e.getTipoEvento().equals("EventoReajuste")) {
                  dtSalario = e.getDtEvento();
                  salarioBase = e.getValorEvento();
                  return true;
            } else if (e.getTipoEvento().equals("EventoRescisao")) {
                  dtRescisao = e.getDtEvento();
                  return true;
            } else
                  return false;
      }

      public String toString() {
            return ("Funcionario: " + this.nomeFunc);
      }

      abstract void processaEvento(Evento evento) throws FolhaException;

      abstract void geraLancamentos(Demonstrativo d) throws FolhaException;
}
```

O método `registraEvento()` inclui o evento na lista de eventos do funcionário, após verificar se o funcionário não está desligado da empresa, se a data do evento é válida e se já não há outro evento do mesmo tipo e data.

O método `registraConvenio()` apenas guarda a referência do convênio e a identificação do contrato do funcionário no mesmo.

O método `geraDemonstrativo()` percorre a lista de eventos do funcionário selecionando os eventos posteriores ao encerramento do demonstrativo anterior. Para cada um desses eventos é executado o método `eventoComum()`, que processa os eventos que são comuns tanto aos funcionários mensalistas como comissionados. Para os eventos específicos de uma classe de funcionários (`eventoComum()==false`) é executado o método `processaEvento()`.

A chamada do método `processaEvento()` está contida num bloco `try/catch` onde exceções do tipo `FolhaException` são tratadas apenas com a impressão dos dados da exceção, de forma que a ocorrência de algum erro no processamento de um evento não provoque uma interrupção em todo o processo.

O método `processaEvento()` é definido na classe `Funcionario` como `abstract`, com implementações diferentes em `FuncionarioMensalista` e `FuncionarioComissionado`, de acordo com os tipos de eventos válidos em cada caso. A chamada de `processaEvento()`, portanto, só é resolvida em tempo de execução, dependendo da classe do funcionário que está sendo processado. É um exemplo de acoplamento dinâmico.

A classe `FuncionarioMensalista`

```
import java.util.Date;
import excecoes.FolhaException;
```

```
public class FuncionarioMensalista extends Funcionario {
	private double horasExtras;
	private double faltas;
	private double horasAtrasos;

	public FuncionarioMensalista(String nome, Date dt, float sal) {
		super(nome, dt, sal);
	}

	public void processaEvento(Evento e) throws FolhaException {
		if (e.getTipoEvento().equals("EventoHoraExtra")){
			this.horasExtras += e.getValorEvento();
		}
		else if (e.getTipoEvento().equals("EventoFalta")){
			this.faltas++;
		}
		else if (e.getTipoEvento().equals("EventoAtraso")){
			this.horasAtrasos += e.getValorEvento();
		}
		else{
			throw new FolhaException("Evento invalido para mensalista.",
					this, e);
		}
	}

	public void geraLancamentos(Demonstrativo d) throws FolhaException {
		double valor;
		double salarioHora;
		try {
			salarioHora = super.salarioBase / 176;
			if (this.horasExtras > 0) {
				valor = this.horasExtras * 2 * salarioHora;
				d.incluiCredito("Horas Extras (" + this.horasExtras +
					" hs)", valor);
			}
			if (this.faltas > 0) {
				valor = this.faltas * 8 * salarioHora;
				d.incluiDebito("Faltas (" + this.faltas + " dia)",
					valor);
			}
			if (this.horasAtrasos > 0) {
				valor = this.horasAtrasos * salarioHora;
				d.incluiDebito("Atrasos (" + this.horasAtrasos +
					" hs)", valor);
			}
		} catch (Exception ex) {
			throw new FolhaException("Erro ao gerar lancamentos do
					ponto.", this, ex);
		}
	}
}
```

A classe `FuncionarioComissionado`

```
import java.util.Date;
import excecoes.FolhaException;

public class FuncionarioComissionado extends Funcionario {

	private double comissoes;
```

```
	public FuncionarioComissionado(String nome, Date dt, float sal) {
		super(nome, dt, sal);
	}

	public void processaEvento(Evento e) throws FolhaException {
		if (e.getTipoEvento().equals("EventoComissao"))
			comissoes += e.getValorEvento();
		else
			throw new FolhaException("Evento invalido para
					comissionado.", this, e);
	}

	public void geraLancamentos(Demonstrativo d) throws FolhaException {
		try {
			if (comissoes > 0)
				d.incluiCredito("Comissoes", comissoes);
		} catch (Exception ex) {
			throw new FolhaException("Erro ao gerar lancamento de
					comissoes.", this, ex);
		}
	}
}
```

A classe `Demonstrativo`

```
import java.util.Date;
import java.util.ArrayList;
import java.util.Iterator;
import excecoes.FolhaException;
import folha.InterfaceFolha;
import folha.Lancamento;

public class Demonstrativo implements InterfaceFolha {
	private Funcionario funcionario;
	private Date dtInicial;
	private Date dtFinal;
	private ArrayList<Lancamento> lancamentos;

	public Demonstrativo(Funcionario f, Date inicio, Date fim) {
		this.funcionario = f;
		this.dtInicial = inicio;
		this.dtFinal = fim;
		this.lancamentos = new ArrayList<>();
	}

	public Date getDtInicial() {
		return this.dtInicial;
	}

	public Date getDtFinal() {
		return this.dtFinal;
	}

	public void incluiDebito(String hist, double val) throws Exception {
		this.lancamentos.add(new Lancamento(hist, -val));
	}

	public void incluiCredito(String hist, double val) throws Exception {
		this.lancamentos.add(new Lancamento(hist, val));
	}

	public String toString() {
```

```
            return ("Demonstrativo de Pagamento:" + "\n Periodo de " +
                    this.dtInicial + " a " + this.dtFinal + "\n " +
                        this.funcionario);
      }

      public void imprime() {
            Lancamento l;
            float total = 0;
            System.out.println(this);
            Iterator<Lancamento> lista = this.lancamentos.iterator();
            while (lista.hasNext()) {
                  l = (Lancamento) lista.next();
                  System.out.println(l);
                  total += l.getValor();
            }

            System.out.println("Total a pagar: " + total);
      }
}
```

A classe `Lancamento`

```
public class Lancamento {
      private String historico;
      private double valor;

      public Lancamento(String st, double val) {
            this.historico = st;
            this.valor = val;
      }

      public String getHistorico() {
            return this.historico;
      }

      public double getValor() {
            return this.valor;
      }

      public String toString() {
            return (this.historico + "\t" + this.valor);
      }
}
```

A classe `MedGrupo`

```
public class MedGrupo implements InterfaceConvenio {
      public void processaContrato(String id, InterfaceFolha f)
                                                            throws Exception {
            if (id.equals("1"))
                  f.incluiDebito("Convenio MedGrupo", 30);
            else
                  throw new Exception();
      }
}
```

Tarefas Propostas

Tarefa 01	Implemente o conjunto de classes apresentado. Realize as alterações que achar conveniente.
Tarefa 02	Elabore um programa que use as classes e que seja capaz de realizar o gerenciamento da folha de pagamentos. Crie uma classe para ser o “repositório de dados”. Crie uma interface com o usuário e uma classe controle.