Módulo 4

Engenharia de Software

Lição 5

Implementação

Versão 1.0 - Jul/2007

Autor
Ma. Rowena C. Solamo

Equipe
Jaqueline Antonio
Naveen Asrani
Doris Chen
Oliver de Guzman
Rommel Feria
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson
Daniel Villafuerte

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Anderson Moreira Paiva
Anna Carolina Ferreira da Rocha
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Daniel Noto Paiva
Denis Mitsuo Nakasaki
Fábio Bombonato
Fabrício Ribeiro Brigagão
Francisco das Chagas
Frederico Dubiel
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Kleberth Bezerra G. dos Santos
Kefreen Ryenz Batista Lacerda
Leonardo Ribas Segala
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Rancoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Classes são escritas de acordo com o que está especificado no projeto. A tradução de um projeto pode ser uma tarefa complicada. Primeiro, o projetista do sistema pode não apontar as características da plataforma e do ambiente de programação. Segundo, os códigos devem ser escritos de tal forma que sejam compreensíveis não só pelo programador que os fez, mas também pelos outros (outros programadores e testadores). Terceiro, o programador deve se aproveitar das características da organização do projeto, das estruturas de dados e das construções específicas da linguagem de programação enquanto cria um código de fácil manutenção e reutilizável. Este capítulo não pretende ensinar cada linguagem de programação, visto que isto é reservado para um curso de programação. Este capítulo oferece algumas dicas de programação que devem ser adotadas na criação de códigos. Ele também mostra algumas boas práticas de engenharia de software que todo programador deve sempre ter em mente.

Ao final desta lição, o estudante será capaz de:

- Conhecer dicas de programação
- Conhecer práticas de engenharia de software para escrever códigos
- Oferecer exemplos de implementação do projeto, particularmente, pacotes, classes de bancos de dados e classes persistentes, e interfaces gráficas com o usuário

2. Padrões e Procedimentos de Programação

O desenvolvimento de software envolve equipes. Uma variedade de tarefas, executadas por diferentes pessoas, são necessárias para se gerar um produto de qualidade. Algumas vezes, a escrita de um código envolve diversas pessoas. Então, torna-se necessário um bom senso de cooperação e coordenação. É importante que os outros sejam capazes de entender o código que foi escrito, o por quê foi escrito e como se integra ao software que está sendo desenvolvido. Foi por esta razão que padrões e procedimentos de programação foram definidos e podem ser usados consistentemente.

Padrões e procedimentos de programação são importantes por diversas razões. Primeiro, eles ajudam o programador a organizar o raciocínio e a evitar a ocorrência de erros. A maneira de se documentar o código-fonte, tornando-o claro e de fácil compreensão, ajuda o programador a escrever e a manter estes códigos. Também ajuda na localização de falhas e dá dicas na hora de se fazer alterações, pois fica mais fácil localizar a seção do código onde a alteração deve ser aplicada. Da mesma forma, auxilia na tradução do código do projeto para o código-fonte, mantendo a correspondência entre os componentes de projeto com os componentes de implementação. Segundo, pode dar assistência aos outros participantes da equipe, tais como os testadores, os integradores e os responsáveis pela manutenção do software. Permite que a comunicação e a coordenação seja clara entre as equipes de desenvolvimento do software.

As pessoas envolvidas no desenvolvimento do software devem definir os padrões específicos de implementação a serem adotados. Estes podem ser:

1. A plataforma onde o software será desenvolvido e usado. Isto inclui os requisitos mínimos e recomendados para o hardware e o software. Para o software, incluir as versões. As atualizações (upgrades) geralmente não são recomendadas. Entretanto, se as atualizações forem necessárias, o impacto deve ser citado e analisado.
2. Os padrões na documentação do código-fonte. Isto será discutido nas dicas de documentação da programação.
3. Os padrões a serem utilizados para a correspondência dos códigos do projeto com os códigos-fonte. O modelo de projeto não tem valor nenhum se a modularidade do projeto não se reflete na implementação. Características do projeto, tais como baixo acoplamento, alta coesão e interfaces bem definidas, também devem ser características da classe. O objetivo principal do software pode permanecer o mesmo durante todo o seu ciclo de vida, mas sua natureza e características podem mudar com o tempo, à medida que os requisitos do usuário são modificados e evoluções são identificadas. Inicialmente, as mudanças são refletidas no projeto. Entretanto, estas também serão repassadas para os componentes de nível mais baixo. A correspondência do projeto com o código-fonte nos permite localizar o código-fonte que precisa ser modificado.

3. Dicas de Programação

Programação é uma habilidade criativa. O programador tem a flexibilidade de implementar o código. O componente do projeto é utilizado como uma dica da função e objetivo do componente. Dicas específicas de linguagem não são discutidas aqui. Dicas Padrões de Programação Java, visite o endereço http://java.sun.com/docs/codeconv/index.html. Esta seção discutirá várias dicas que se aplicam à programação de modo geral.

3.1. Usando Pseudo-códigos

O projeto geralmente oferece um framework para cada componente. É um esboço do que deve ser feito no componente. O programador adiciona sua criatividade e experiência para elaborar as linhas de código que implementam o projeto. Ele tem a flexibilidade de escolher uma construção particular da linguagem de programação para usar, como usá-la, como o dado será representado, e assim por diante.

Pseudo-códigos podem ser usados para adaptar o projeto à linguagem de programação escolhida. Um exemplo é o inglês estruturado que descreve o fluxo de código. Adotando-se construções e representações de dados, sem se envolver imediatamente com as especificidades de um comando ou bloco de comandos, o programador pode experimentar e decidir qual é melhor implementação. Códigos podem ser reorganizados ou reconstruídos com o mínimo de reescrita.

3.2. Dicas para as Estruturas de Controle

A estrutura de controle é definida pela arquitetura do software. No software orientado a objetos, é baseada em mensagens sendo enviadas entre os objetos de classes, no estado do sistema e nas mudanças nas variáveis. É importante que a estrutura da classe reflita a estrutura de controle do projeto. A modularidade é uma característica de projeto que deve ser traduzida como uma característica da classe. Construindo a classe em blocos modulares (métodos), o programador pode esconder detalhes da implementação em diferentes níveis, tornando todo o sistema fácil de entender, testar e manter. **Acoplamento** e **coesão** são outras características de projeto que também devem ser traduzidas em características da classe.

3.3. Dicas de Documentação

A Documentação da classe é um conjunto de descrições escritas que explicam ao leitor o que estas fazem e como fazem. Dois tipos de documentações podem ser criadas:

Documentação Interna

É um documento descritivo diretamente escrito no código-fonte. É direcionado a alguém que terá que ler o código-fonte. Uma informação resumida é oferecida para descrever suas estruturas de dados, algoritmos e fluxos de controle. Geralmente, esta informação é colocada no início do código. Esta seção do código-fonte é conhecida como ***bloco cabeçalho de comentário***. Ele age como uma introdução ao código-fonte e identifica os seguintes elementos:

- Nome do Componente
- Autor do Componente
- Data em que o Componente foi criado ou teve sua última modificação
- Lugar onde o Componente se aplica no sistema em geral
- Detalhes da estrutura de dados, do algoritmo e do fluxo de controle do componente

Opcionalmente, pode-se adicionar um histórico das revisões que foram feitas no componente. O elemento de histórico consiste de:

- Quem modificou o componente?
- Quando o componente foi modificado?
- Qual foi a modificação?

As documentações internas são criadas para as pessoas que irão ler o código.

Dicas para se escrever códigos

1. Use nomes de variáveis e métodos que sejam significativos.
2. Use formatação para melhorar a legibilidade dos códigos, tais como recuo e divisão em blocos.
3. Em alguns blocos de comandos, coloque comentários adicionais para esclarecer os leitores.
4. Tenha métodos separados para entrada (input) e saída (output).
5. Evite o uso de GOTO's. Evite escrever códigos que saltam de repente de um lugar para outro.
6. A escrita de código também é interativa, por exemplo, geralmente alguém começa com um rascunho. Se o fluxo de controle está complexo e difícil de entender, pode-se reestruturar.

Documentação Externa

Todos os outros documentos que não são parte do código-fonte mas estão relacionados ao código-fonte são conhecido como ***documentos externos***. Este tipo de documentação é para aqueles que não precisam necessariamente ler os códigos das classes. Descrevem como os componentes interagem uns com os outros, incluindo classes de objetos e suas heranças hierárquicas. Para um sistema orientado a objeto, identifica as pré-condições e pós-condições do código-fonte.

4. Implementando Pacotes

Os pacotes oferecem um mecanismo de reuso do software. Um dos objetivos dos programadores é criar componentes de software reutilizáveis de tal forma que códigos não tenham que ser escritos repetidamente. A Linguagem de Programação Java oferece um mecanismo para a definição de pacotes. Na realidade, são diretórios que são utilizados para organizar as classes e interfaces. O Java oferece uma convenção para que os nomes de pacotes e classes sejam únicos. Com centenas de milhares de programadores Java espalhados pelo mundo, o nome que alguém pode utilizar para a sua classe pode coincidir com o nome de uma classe desenvolvida por outros programadores. Os nomes dos pacotes devem ser formados por caracteres ASCII, todos em minúsculas. Devem seguir a Convenção de Nomes de Domínio da Internet conforme especificado no formato X.500.

A seguir estão os passos da definição de um pacote em Java.

1. Defina uma classe public. Se a classe não for pública, somente poderá ser usada por outras classe no mesmo pacote. Considere o código da classe persistente *Athlete* mostrada no código abaixo.

```
package abl.athlete.pc;
public class Athlete{
  private int athleteID;
  private String lastName;
  private String firstName;
  ... // restante dos atributos
  public void setAthleteID(int id) { //métodos modificadores (set)
    athleteID = id;
  }
  ... // outros métodos modificadores
  public int getAthleteID() { // métodos acessores (get)
    return athleteID;
  }
  ... // outros métodos acessores
}
```

2. Escolha um nome para o pacote. Adicione a palavra-chave *package* no arquivo do código-fonte para uma definição de classe reutilizável. Neste exemplo, o nome do pacote é *abl.athlete.pc* que é o nome do pacote para classes persistentes e listas de classes. No exemplo é:

```
package abl.athlete.pc;
```

Colocando a palavra-chave package no início do arquivo fonte indica que a classe definida no arquivo faz parte do pacote especificado.

3. Compile a classe para que seja colocada na devida estrutura de diretório de pacotes. A classe compilada torna-se disponível para o compilador e o interpretador. Quando um arquivo Java contendo um pacote é compilado, o arquivo resultante **.class* é colocado no diretório especificado pelo comando package. No exemplo, o arquivo *athlete.class* é colocado no diretório **pc** dentro do diretório **athlete** que está dentro do diretório **abl**. Se estes diretórios não existirem, o compilador os criará.
4. Para reutilizar a classe, simplesmente importe o pacote. O código mostrado no código abaixo é um exemplo de importação da classe *Athlete* que é utilizada pela classe *DBAthlete*.

```
import abl.athlete.pc.*; // Todas as classes públicas deste pacote
                         // estão disponíveis para uso em DBAthlete
public class DBAthlete {
  private Athlete ath; // Define uma referência para um objeto Athlete
  ... // restante do código
}
```

5. Implementando Controladores

Implementar controladores é como escrever classes nos módulos anteriores. Uma boa prática de programação é a utilização de classes abstratas e interfaces. Esta seção serve como uma revisão de classes abstratas e interfaces. A utilização de classes abstratas e interfaces aumenta consideravelmente a capacidade do software de ser reutilizável e gerenciável. Também permite ao software ter capacidades 'plug-and-play'. Esta seção serve como uma revisão de classes abstratas e interfaces em Java[1].

5.1. Classes Abstratas

Suponha que queiramos criar uma superclasse que tenha certos métodos com alguma implementação, e outros métodos que serão implementados pelas suas subclasses.

Por exemplo, queremos criar uma superclasse chamada LivingThing (SerVivo). Esta classe terá alguns métodos como breath (respirar), eat (comer), sleep (dormir) e walk (andar). Entretanto, há alguns métodos nesta superclasse que não poderemos generalizar o seu comportamento. Tomemos, por exemplo, o método walk. Todos os seres vivos não se locomovem do mesmo jeito. Veja os seres humanos, por exemplo. Os humanos andam em duas pernas, enquanto que outros seres vivos, como os cachorros, andam em quatro patas. Entretanto, há várias características que os seres vivos têm em comum, e este é o motivo para querermos criar uma superclasse genérica para isso.

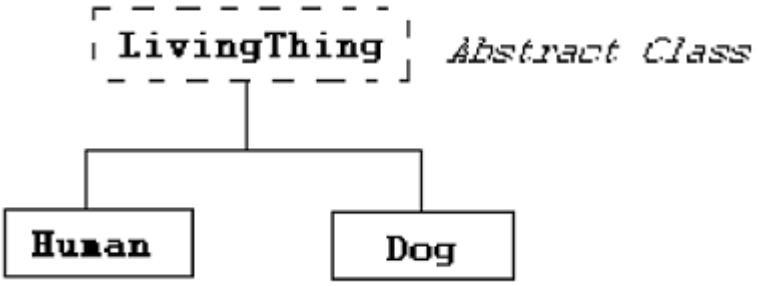

Figura 1: Classe Abstrata

A fim de implementarmos isto, podemos criar uma superclasse que tenha alguns métodos com implementação e outros sem. Este tipo de classe é denominada classe abstrata.

Uma **classe abstrata** é uma classe que não pode ser instanciada. Geralmente aparece no topo de uma hierarquia de classes na programação orientada a objeto, definindo os vários tipos de ações possíveis com os objetos de todas as subclasses.

Os métodos da classe abstrata que não têm implementação são chamados **métodos abstratos**. Para criar um método abstrato, simplesmente escreva a declaração do método sem conteúdo nenhum e use a palavra-chave abstract. Por exemplo:

```
public abstract void someMethod();
```

Criaremos um exemplo de classe abstrata.

```
public abstract class LivingThing {
  public void breath(){
    System.out.println("Living Thing breathing...");
  }
  public void eat(){
    System.out.println("Living Thing eating...");
  }
  /**
   * método abstrato walk
   * Este método será implementado pelas subclasses de LivingThing
   */
  public abstract void walk();
```

1 O uso do texto do curso JEDI Introdução à Programação teve autorização prévia dos seus autores.

```
}
```

Quando uma classe estende a classe abstrata *LivingThing*, será obrigada a implementar o método abstrato *walk*(), caso contrário está subclasse deverá ser uma classe abstrata também e, portanto, não poderá ser instanciada. Por exemplo:

```
public class Human extends LivingThing {
  public void walk(){
    System.out.println("Human walks...");
  }
}
```

Se a classe *Human* não implementasse o método *walk*(), ocorreria a seguinte mensagem de erro:

```
Human.java:1: Human is not abstract and does not override abstract method
walk() in LivingThing
public class Human extends LivingThing
 ^
1 error
```

Dicas de Codificação:

Use classes abstratas para definir vários tipos de comportamentos no topo das hierarquia das classes na programação orientada a objetos, e use as suas subclasses para implementar os detalhes da classe abstrata.

5.2. Interfaces

Uma **interface** é um tipo de bloco especial que contém apenas a assinatura de métodos (e provavelmente constantes). As interfaces definem as assinaturas de um conjunto de métodos, sem o corpo.

As interfaces definem uma maneira pública e padrão para especificar o comportamento das classes. Elas permitem que as classes, independentemente de sua localização na hierarquia de classes, implementem comportamentos comuns. As interfaces implementam o polimorfismo também, pois uma classe pode chamar um método de uma interface e a versão apropriada deste método será executada dependendo do tipo do objeto passado na chamada ao método da interface.

5.3. Por que usamos Interfaces?

Devemos usar interfaces quando *queremos que classes não relacionadas implementem métodos parecidos*. Através das interfaces podemos observar similaridades entre as classes não relacionadas sem que tenhamos que forçar um relacionamento artificial na classe.

Vamos tomar como exemplo a classe **Line** que contém métodos que calcula o tamanho da linha e compara o objeto **Line** com objetos da mesma classe. Suponha que tenhamos outra classe **MyInteger** que contém métodos que compara um objeto **MyInteger** com objetos da mesma classe. Como podemos ver aqui, ambas as classes têm métodos parecidos que os comparam com outros outros objetos do mesmo tipo, entretanto eles não estão relacionados entre si. Para que possamos forçar uma maneira de se ter certeza de que estas duas classes implementam alguns métodos com assinaturas parecidas, podemos usar uma interface para isso. Podemos criar uma classe interface, por exemplo, a interface **Relation** que terá algumas declarações de métodos de comparação. A nossa interface **Relation** pode ser declarada como:

```
public interface Relation {
  public boolean isGreater( Object a, Object b);
  public boolean isLess( Object a, Object b);
  public boolean isEqual( Object a, Object b);
}
```

Uma outra razão para utilizar uma interface na programação de um objeto é revelar a interface de

programação de um objeto, sem revelar a sua classe. Como poderemos ver mais tarde na seção *Interface versus Classes*, podemos utilizar uma interface como um tipo de dado.

Finalmente, precisamos utilizar interfaces para modelar heranças múltiplas o que permite que uma classe tenha mais de uma superclasse. A herança múltipla não está presente no Java, mas está presente em outras linguagens orientadas a objeto como o C++.

5.4. Interface versus Classes Abstratas

A seguir termos as principais diferenças entre uma interface e uma classe abstrata: os métodos de uma interface não têm corpo, uma interface somente pode definir constantes e uma interface não tem nenhum relacionamento direto de herança com qualquer classe particular, elas são definidas independentemente.

5.5. Interface versus Classe

Uma característica comum entre uma interface e uma classe é que ambas são tipos. Isto significa que uma interface pode ser utilizada em lugares onde uma classe pode ser utilizada. Por exemplo, dada uma classe *Person* e uma interface *PersonInterface*, as seguintes declarações são válidas:

```
PersonInterface pi = new Person();
Person pc = new Person();
```

Entretanto, não se pode instanciar uma interface. Um exemplo disso é:

```
PersonInterface pi = new PersonInterface(); //ERRO DE COMPILAÇÃO!!!
```

Outra característica comum é que ambas, interface e classe, podem definir métodos. Entretanto, uma interface não possui código de implementação, o que ocorre na classe.

5.6. Criando Interfaces

Para criar uma interface escrevemos:

```
public interface [InterfaceName] {
  // alguns métodos sem corpo
}
```

Como um exemplo, criaremos uma interface que defina relacionamentos entre dois objetos conforme a "ordem natural" dos objetos.

```
public interface Relation {
  public boolean isGreater(Object a, Object b);
  public boolean isLess(Object a, Object b);
  public boolean isEqual(Object a, Object b);
}
```

Para utilizar a interface, usamos a palavra-chave **implements**. Por exemplo:

```
/**
 * Esta classe define um seguimento de linha
 */
public class Line implements Relation {
  private double x1;
  private double x2;
  private double y1;
  private double y2;

  public Line(double x1, double x2, double y1, double y2) {
    this.x1 = x1;
    this.x2 = x2;
    this.y1 = y1;
    this.y2 = y2;
  }
  public double getLength() {
    double length = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)* (y2-y1));
```

```
    return length;
  }
  public boolean isGreater(Object a, Object b) {
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen > bLen);
  }
  public boolean isLess(Object a, Object b) {
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen < bLen);
  }
  public boolean isEqual(Object a, Object b) {
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen == bLen);
  }
}
```

Quando sua classe tenta implementar uma interface, sempre tenha certeza de que todos ps métodos da interface foram implementados ou, caso contrário, será mostrado o seguinte erro:

```
Line.java:4: Line is not abstract and does not override abstract method
isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
 ^
1 error
```

Dicas de Codificação:

Utilize interfaces para criar algumas definições padrões de métodos em várias classes diferentes. Uma vez que um conjunto padrão de definições de métodos esteja criado, é possível escrever um único método para manipular todas as classes que implementam a interface.

5.7. Relacionamento de uma Interface com uma Classe

Como vimos na seção anterior, uma classe pode implementar uma interface se ela tiver código de implementação para todos os métodos definidos na interface.

Outro aspecto para observar sobre o relacionamento de interfaces com classe é que uma classe só pode ESTENDER UMA superclasse, mas pode IMPLEMENTAR VÁRIAS interfaces. Um exemplo de uma classe que implementa várias interfaces é:

```
public class Person implements PersonInterface, LivingThing,
  WhateverInterface {
  //algum código aqui
}
```

Outro exemplo de uma classe que estende uma superclasse e implementa uma interface é:

```
public class ComputerScienceStudent extends Student
  implements PersonInterface, LivingThing {
  // algum código aqui
}
```

Note que uma interface não faz parte do patrimônio hierárquico da classe. Classes não relacionadas podem implementar a mesma interface.

5.8. Herança entre Interfaces

Interfaces não são parte da hierarquia da classe. Entretanto, interfaces podem ter relacionamentos de herança entre si. Por exemplo, suponha que tenhamos duas interfaces ***StudentInterface*** e ***PersonInterface***. Se *StudentInterface* estende *PersonInterface*, ele vai herdar todas as declarações de métodos em *PersonInterface*.

```
public interface PersonInterface {
  ...
}
public interface StudentInterface extends PersonInterface {
  ...
}
```

Uma **interface** formaliza o polimorfismo. Ela define o polimorfismo de maneira declarativa, sem relação com a implementação. Esta é a chave para a capacidade "plug-and-play" de uma arquitetura. É um tipo especial de bloco contendo apenas assinaturas de métodos (e possivelmente constantes). As interfaces definem as assinaturas de um conjunto de métodos sem o corpo.

6. Implementando Conectividade com Banco de Dados usando Java (JDBC)

A maioria das aplicações utilizam um sistema de banco de dados relacional como repositório de dados. Uma linguagem denominada *Structured Query Language (SQL)* é utilizada quase que universalmente nestes sistemas. Ela é utilizada para fazer perguntas, isto é, requisitar informações que satisfaçam um determinado critério. Java permite que o programador escreva código que utilize os comandos SQL para acessar as informações em um sistema de banco de dados relacional. A engenharia do banco de dados que implementa os dados da aplicação não faz parte do escopo deste curso e normalmente é discutida em um curso de banco de dados. Esta seção discute como Java utiliza o JDBC para se conectar e requisitar serviços a um servidor de banco de dados. No exemplo, mostraremos um trecho de código que recupera registros de atletas do banco de dados. O código assume que *Athlete* e *PCLAthlete* já estejam implementados.

Os trechos de código a seguir são uma implementação de *DBAthlete*. Identifique em *PCLAthlete* onde são carregados os registros de atletas que satisfazem um critério baseado na entrada do usuário. O comando **import java.sql** importa o pacote que contém as classes e interfaces que manipulam bancos de dados relacionais em Java.

O construtor da classe tenta se conectar ao banco de dados. Se conseguir, comandos SQL podem ser enviados ao banco de dados. As linhas seguintes especificam a URL (*Uniform Resource Locator* – Localizador de Recurso Uniforme) do banco de dados que ajuda a classe a localizar o banco de dados, nome do usuário e senha. A URL especifica o protocolo de comunicação (**JDBC**), o subprotocolo (**ODBC**) e o nome do banco de dados (*ABLDatabase*). O nome do usuário e a senha também são especificados para logar no banco de dados; no exemplo, o nome do usuário é "postgre" e a senha é "postgres1".

```
String url = "jdbc:odbc:ABLDatabase";
String username = "postgre";
String password = "postgres1";
```

A definição de classe do driver do banco de dados deve ser carregada antes que se possa conectar ao banco de dados. A seguinte linha carrega o driver. Neste exemplo, o driver de banco de dados padrão ponte JDBC-ODBC é utilizado para permitir que qualquer classe Java acesse qualquer fonte de dados ODBC. Para mais informações do sobre o driver JDBC e os bancos de dados suportados, visite o seguinte site: http://java.sun.com/products/jdbc.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Para estabelecer uma conexão, um objeto de conexão é criado. Ele gerencia a conexão entre o banco de dados e a classe Java. Ele também fornece suporte para a execução de comandos SQL. No exemplo, o seguinte comando fornece esta conexão.

```
connection = DriverManager.getConnection(url, username, password);
```

O método inicial que é invocado para começar a recuperação de registros de atletas do banco de dados é o método *getAthleteRecord*(). O código deste método é mostrado abaixo. O objeto *selectStmt* é utilizado para definir o comando SELECT que especifica quais registros serão recuperados do banco de dados. O método privado *prepareSelectStmt*() gera o comando *select* com o critério apropriado. O objeto *statement* é utilizado para enviar comandos SQL ao banco de dados e é instanciado utilizando-se o método *createStatement*() do objeto *connection*. O método *executeQuery*() do objeto *statement* é utilizado para executar o comando SELECT conforme especificado no objeto *selectStmt*. O resultado da pergunta é colocado e referenciado em **rs** que é um objeto *ResultSet*.

O objeto *PCLAthlete* é carregado pela execução do método *populatePCLAthlete*() que utiliza como entrada o *ResultSet*. Para cada registro no conjunto de resultados, um objeto *athlete* é instanciado e tem seus atribuídos preenchidos com os dados do atleta. O método *getPCLAthlete*() retorna uma referência para a lista de atletas gerada.

```
import java.sql.*;
import java.util.*;

public class DBAthlete {
  private Connection connection;
  private PCLAthlete athleteList;

  public DBTeam() {
    // Para conectar ao Banco de Dados...
    String url = "jdbc:odbc:ABLDatabase";
    String username = "postgre";
    String password = "postgres1";
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      connection = DriverManager.getConnection(url, username, password);
    } catch (ClassNotFoundException e) {
      ...
    } catch (SQLException e) {
      ...
    }
  }
  // para executar o comando SELECT
  public void getAthleteRecord(String searchCriteria) throws SQLException {
    String selectStmt = prepareSelectStmt(searchCriteria);
    Statement statement = connection.createStatement();
    ResultSet rs = statement.executeQuery(selectStmt);
    populatePCLAthlete(rs);
    statement.close();
  }
  // para gerar o comando SELECT com o critério de pesquisa
  private String prepareSelectStmt(String searchCriteria)
     throws SQLException {
    String query = "SELECT * FROM athlete, guardian "
    query = query + "WHERE athlete.guardianID = guardian.guardianID";
    query = query + " AND " + searchCriteria + ";" ;
    return query;
  }
  // para preencher a lista persistente da classe.
  private void populatePCLAthlete(ResultSet rs) throws SQLException {
    ....
    athleteList = new PCLAthlete();
    ....
    // dentro de um bloco try e catch
    ResultSetMetaData rsmd = rs.getMetaData();
    do {
      athleteList.add(getNextMember(rs, rsmd);
    } while (rs.next())
  }
  // para gerar um objeto atleta a ser adicionado a PCLAthlete
  private Athlete getNextMember(ResultSet rs, ResultSetMetaData rsmd)
     throws SQLException {
    Athlete a = new Athlete();
    // Construtor para converter string em integer
    a.setAthleteID(rs.getString(1));
    a.setLastName(rs.getString(2));
    a.setFirstName(rs.getString(3));
    ... // preencher os atributos para o objeto atleta a
    return a;
  }
  // retorna uma referência para a lista de atletas (PCLAthlete)
  public PCLAthlete getPCLAthlete() throws SQLException {
    return athleteList;
  }
}
```

7. Implementando a Interface Gráfica do Usuário

Mesmo sem conhecer sobre as APIs de interface gráfica do usuário (GUI), ainda é possível criar muitas de classes diferentes. Entretanto, os projetos tendem a ser estranhos e pouco atraentes para os usuários. Ter uma boa GUI afeta a utilização da sua aplicação. Isto resulta em facilidade no uso para os usuários de seu projeto. Java oferece ferramentas como o *Abstract Windowing Toolkit* (*AWT*) e o *Swing* para desenvolver aplicações com *GUIs* interativos. A seção seguinte fornece uma discussão de *Abstract Windowing Toolkit* e *Swing*[2].

A biblioteca *Java Fundation Classe* (JFC), é uma parte importante do Java SDK, se refere a uma coleção de APIs que simplificam o desenvolvimento de aplicações Java com GUI. Originalmente consiste de cinco APIs, incluindo o AWT e o Swing. As outras três APIs são Java2D, Accessibility, e Drag and Drop. Todas estas APIs ajudam os desenvolvedores no projeto e implementação de aplicações visualmente incrementadas.

Ambos AWT e Swing fornecem componentes GUI que podem ser usados na criação de aplicações e applets Java. Conheceremos mais sobre applets em outras lições. Ao contrário de alguns componentes AWT que utilizam código nativo, o Swing é totalmente escrito utilizando a linguagem de programação Java. Como resultado, o Swing fornece uma implementação independente de plataforma garantindo que as aplicações distribuídas em diferentes plataformas terão a mesma aparência. O AWT, entretanto, faz com que a aparência de uma aplicação executada em duas máquinas diferentes sejam passíveis de comparação. A API Swing é construída sobre um número de APIs que implementam várias partes do AWT. Como resultado, os componentes AWT ainda podem ser usados com os componentes Swing.

7.1. Componentes GUI AWT

Classes Window Fundamentais

No desenvolvimento de aplicações GUI, os componentes GUI tais como botões e campos de texto são colocados em contêineres. Esta é a lista das classes contêineres que são encontradas no pacote AWT.

Classe	***Descrição***
Component	Uma classe abstrata para objetos que podem ser mostrados na tela e interagir com o usuário. É a raiz de todas as classes AWT.
contêiner	Uma subclasse abstrata da classe *Component*. Um componente que pode conter outros componentes.
Panel	Estende a classe *contêiner*. Um quadro ou janela sem a barra de título, a barra de menu e as bordas. Superclasse da classe *Applet*.
Window	Estende a classe *contêiner*. Um janela de alto nível, o que significa que não pode estar contida dentro de qualquer outro objeto. Não tem bordas ou barra de menu.

Frame	Estende a classe *Window*. Uma janela com título, barra de menu, borda e cantos redimensionáveis. Tem quatro construtores, dois dos quais têm as seguintes assinaturas:
	`Frame()`
	`Frame(String title)`

Tabela 1: Classes Contêineres AWT

Para definir o tamanho da janela, o método *override* setSize é usado.

```
void setSize(int width, int height)
```

Redimensiona o componente para a largura e altura fornecidos como parâmetros.

```
void setSize(Dimension d)
```

Redimensiona este componente para *d.width* e *d.height* baseado na *Dimension d* especificado.

2 O uso do texto do curso JEDI Introdução à Programação II teve autorização prévia dos seus autores.

Por padrão, uma janela não está visível a menos que seja definida sua visibilidade como *true*. Aqui está a sintaxe para o método *setVisible*.

```
void setVisible(boolean b)
```

No projeto de aplicações GUI, objetos *Frame* geralmente são usados. Aqui está um exemplo da criação de tal aplicação.

```
import java.awt.*;

public class SampleFrame extends Frame {
  public static void main(String args[]) {
    SampleFrame sf = new SampleFrame();
    sf.setSize(100, 100); // Tente remover esta linha
    sf.setVisible(true);  // Tente remover esta linha
  }
}
```

Observe que o botão fechar do quadro não funciona ainda pois nenhum mecanismo de tratamento de evento foi adicionado ao programa ainda. Conheceremos sobre tratamento de eventos em futuras lições.

Gráficos

Diversos métodos gráficos podem ser encontrados na classe *Graphics*. Aqui está uma lista com alguns destes métodos.

`drawLine()`	`drawPolyline()`	`setColor()`
`fillRect()`	`drawPolygon()`	`getFont()`
`drawRect()`	`fillPolygon()`	`setFont()`
`clearRect()`	`getColor()`	`drawString()`

Tabela 2: Alguns métodos da classe Graphics

Relacionada a esta classe, temos a classe *Color*, que tem três construtores.

Formato do Construtor	***Descrição***
`Color(int r, int g, int b)`	Os valores inteiros são de 0 a 255.
`Color(float r, float g, float b)`	Os valores float são de 0.0 a 1.0.
`Color(int rgbValue)`	O intervalo de valor vai de 0 a 2^{24}-1 (preto a branco). Vermelho (red): bits 16-23 Verde (green): bits 8-15 Azul (blue): bits 0-7

Tabela 3: Construtores de Color

Aqui está um exemplo de programa que utiliza alguns dos métodos da classe *Graphics.*

```
import java.awt.*;

public class GraphicPanel extends Panel {
  public GraphicPanel() {
    setBackground(Color.black); //constante na classe Color
  }
  public void paint(Graphics g) {
    g.setColor(new Color(0,255,0)); //verde
    g.setFont(new Font("Helvetica",Font.PLAIN,16));
    g.drawString("Hello GUI World!", 30, 100);
    g.setColor(new Color(1.0f,0,0)); //vermelho
    g.fillRect(30, 100, 150, 10);
  }
  public static void main(String args[]) {
    Frame f = new Frame("Testing Graphics Panel");
    GraphicPanel gp = new GraphicPanel();
```

```
    f.add(gp);
    f.setSize(600, 300);
    f.setVisible(true);
  }
}
```

Para que um painel se torne visível, ele deve ser colocado dentro de uma janela visível, como um *Frame*.

Mais Componentes AWT

Aqui está uma lista dos controles AWT. Controles são componentes tais como botões ou campos de texto que permitem ao usuário interagir com a aplicação GUI. São subclasses da classe *Component*.

Label	Button	Choice
TextField	Checkbox	List
TextArea	CheckboxGroup	Scrollbar

Tabela 4: Componentes AWT

O seguinte programa cria uma moldura contendo controles dentro dela.

```
import java.awt.*;

public class FrameWControls extends Frame {
  public static void main(String args[]) {
    FrameWControls fwc = new FrameWControls();
    fwc.setLayout(new FlowLayout());
    fwc.setSize(600, 600);
    fwc.add(new Button("Test Me!"));
    fwc.add(new Label("Labe"));
    fwc.add(new TextField());
    CheckboxGroup cbg = new CheckboxGroup();
    fwc.add(new Checkbox("chk1", cbg, true));
    fwc.add(new Checkbox("chk2", cbg, false));
    fwc.add(new Checkbox("chk3", cbg, false));
    List list = new List(3, false);
    list.add("MTV");
    list.add("V");
    fwc.add(list);
    Choice chooser = new Choice();
    chooser.add("Avril");
    chooser.add("Monica");
    chooser.add("Britney");
    fwc.add(chooser);
    fwc.add(new Scrollbar());
    fwc.setVisible(true);
  }
}
```

7.2. Gerenciadores de Layout

A posição e o tamanho dos componentes dentro de cada contêiner é determinado pelo gerenciador de *layout*. Os gerenciadores de layout determinam o layout dos componentes no contêiner. Estes são alguns gerenciadores de layout incluídos no Java.

1. FlowLayout
2. BorderLayout
3. GridLayout
4. GridBagLayout
5. CardLayout

O gerenciador de layout pode ser definido utilizando o método *setLayout* da classe *contêiner*. O método tem a seguinte assinatura.

```
void setLayout(LayoutManager mgr)
```

Caso se opte pela não utilização de qualquer gerenciador de layout, é possível passar o valor *null* como parâmetro para este método. Entretanto, todos os componentes devem ser posicionados manualmente utilizando o método *setBounds* da classe *Component*.

```
public void setBounds(int x, int y, int width, int height)
```

O método que controla a posição é baseado nos parâmetros *x* e *y*, e o tamanho do componente nos parâmetros *width* e *height*. Isto pode se tornar difícil e tedioso caso se tenham muitos objetos *Component* dentro do objeto *contêiner*. Este método deverá ser chamado para cada componente.

Nesta seção, veremos os três primeiros gerenciadores de *layout*.

Gerenciador FlowLayout

O *FlowLayout* é o gerenciador padrão para a classe *Panel* e suas subclasses, incluindo a classe Applet. Ele posiciona os componentes no sentido esquerda pra direita e de cima para baixo, começando do canto esquerdo superior. Imagine-se digitando em um editor de textos. É assim que o gerenciador *FlowLayout* funciona. Ele tem três construtores que estão listados abaixo.

Construtores do FlowLayout
`FlowLayout()`
Cria um novo objeto FlowLayout com alinhamento central e 5 unidades de espaçamento horizontal e vertical aplicadas aos componentes por padrão.
`FlowLayout(int align)`
Cria um novo objeto FlowLayout com o alinhamento especificado e 5 unidades de espaçamento horizontal e vertical aplicadas aos componentes por padrão.
`FlowLayout(int align, int hgap, int vgap)`
Cria um novo objeto FlowLayout com o primeiro parâmetro como alinhamento e o valor de *hgap* em unidades de espaçamento horizontal e o valor de v*gap* em unidades de espaçamento vertical aplicadas aos componentes.

Tabela 5: Construtores do FlowLayout

O espaçamento se refere ao espaço entre os componentes e é medido em pixels. O parâmetro de alinhamento deve ser um dos seguintes:

1. FlowLayout.LEFT
2. FlowLayout.CENTER
3. FlowLayout.RIGHT

Qual é a saída esperada para a seguinte classe?

```
import java.awt.*;

public class FlowLayoutDemo extends Frame {
  public static void main(String args[]) {
    FlowLayoutDemo fld = new FlowLayoutDemo();
    fld.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 10));
    fld.add(new Button("ONE"));
    fld.add(new Button("TWO"));
    fld.add(new Button("THREE"));
    fld.setSize(100, 100);
    fld.setVisible(true);
  }
}
```

Temos na Figura 2 um exemplo de saída executado na plataforma Windows.

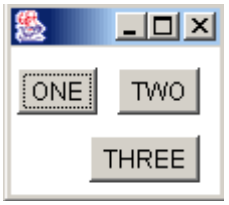

Figura 2: Saída do código exemplo no Windows

Gerenciador BorderLayout

O BorderLayout divide o Contêiner em cinco partes - norte, sul, leste, oeste e centro. Cada componente é adicionado a uma região específica. As regiões norte e sul esticam horizontalmente enquanto as regiões leste e oeste ajustam-se verticalmente. A região central, por outro lado, ajusta-se horizontal e verticalmente. Este é o layout padrão dos objetos *Window*, incluindo os objects das subclasse de *Window* do tipo *Frame* e *Dialog*.

Construtores de BorderLayout
`BorderLayout()`
Cria um novo objeto BorderLayout sem espaços entre os diferentes componentes.
`BorderLayout(int hgap, int vgap)`
Cria um novo objeto BorderLayout com espaçamento horizontal *hgap* unidades e espaçamento vertical com o valor de *vgap* em unidades aplicados entre os diferentes componentes.

Tabela 6: Construtores de BorderLayout

Assim como o gerenciador *FlowLayout*, os parâmetros *hgap* e *vgap* também se referem ao espaçamento entre os componentes dentro do contêiner.

Para adicionar um componente em uma região específica, use o método *add* e passe dois parâmetros: o componente a ser adicionado e a região onde o componente deverá ser posicionado. Observe que apenas um componente pode ser colocado em uma região. Quando se adiciona mais de um componente a uma região em particular, será mostrado apenas o último componente adicionado. A seguinte lista fornece as regiões válidas e campos pré-definidos na classe *BorderLayout.*

1. BorderLayout.NORTH
2. BorderLayout.SOUTH
3. BorderLayout.EAST
4. BorderLayout.WEST
5. BorderLayout.CENTER

Aqui está um exemplo de uma classe que demonstra como o *BorderLayout* funciona.

```
import java.awt.*;

public class BorderLayoutDemo extends Frame {
  public static void main(String args[]) {
    BorderLayoutDemo bld = new BorderLayoutDemo();
    bld.setLayout(new BorderLayout(10, 10));   // pode ser removido
    bld.add(new Button("NORTH"), BorderLayout.NORTH);
    bld.add(new Button("SOUTH"), BorderLayout.SOUTH);
    bld.add(new Button("EAST"), BorderLayout.EAST);
    bld.add(new Button("WEST"), BorderLayout.WEST);
    bld.add(new Button("CENTER"), BorderLayout.CENTER);
    bld.setSize(200, 200);
    bld.setVisible(true);
  }
}
```

Aqui está uma amostra da saída desta classe. A segunda figura mostra o efeito do redimensionamento da janela.

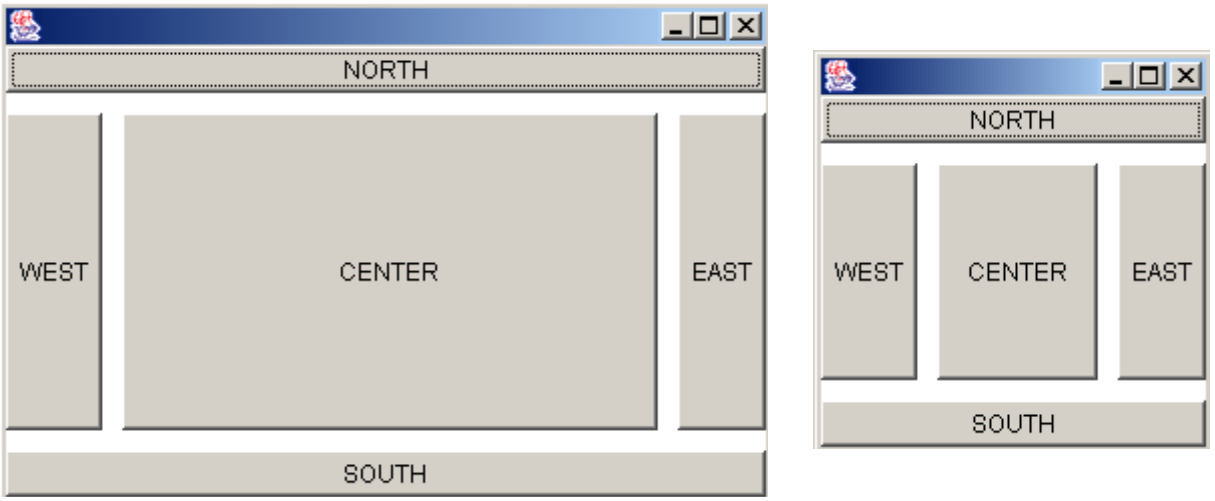

Figura 3: Saída do código exemplo

Gerenciador GridLayout

Com o gerenciador *GridLayout*, os componentes também são posicionados da esquerda para a direita e de cima para baixo assim como no gerenciador *FlowLayout*. Além disso, o gerenciador *GridLayout* divide o contêiner em um determinado número de linhas e colunas. Todas estas regiões terão o mesmo tamanho. Ele sempre ignora o tamanho definido para o componente.

A seguir estão os construtores disponíveis para a classe *GridLayout*.

Construtores de GridLayout
`GridLayout()`
Cria um novo objeto GridLayout com uma única linha e uma única coluna, por padrão.
`GridLayout(int rows, int cols)`
Cria um novo objeto BorderLayout com o número de linhas e colunas especificado.
`GridLayout(int rows, int cols, int hgap, int vgap)`
Cria um novo objeto BorderLayout com o número de linhas e colunas especificado. O espaçamento no valor de *hgap* em unidades horizontais e *vgap* unidades verticais é aplicado aos componentes.

Tabela 7: Construtores de GridLayout

Tente esta classe.

```
import java.awt.*;

public class GridLayoutDemo extends Frame {
  public static void main(String args[]) {
    GridLayoutDemo gld = new GridLayoutDemo();
    gld.setLayout(new GridLayout(2, 3, 4, 4));
    gld.add(new Button("ONE"));
    gld.add(new Button("TWO"));
    gld.add(new Button("THREE"));
    gld.add(new Button("FOUR"));
    gld.add(new Button("FIVE"));
    gld.setSize(200, 200);
    gld.setVisible(true);
  }
}
```

Esta é a saída da classe. Observe o efeito do redimensionamento da janela.

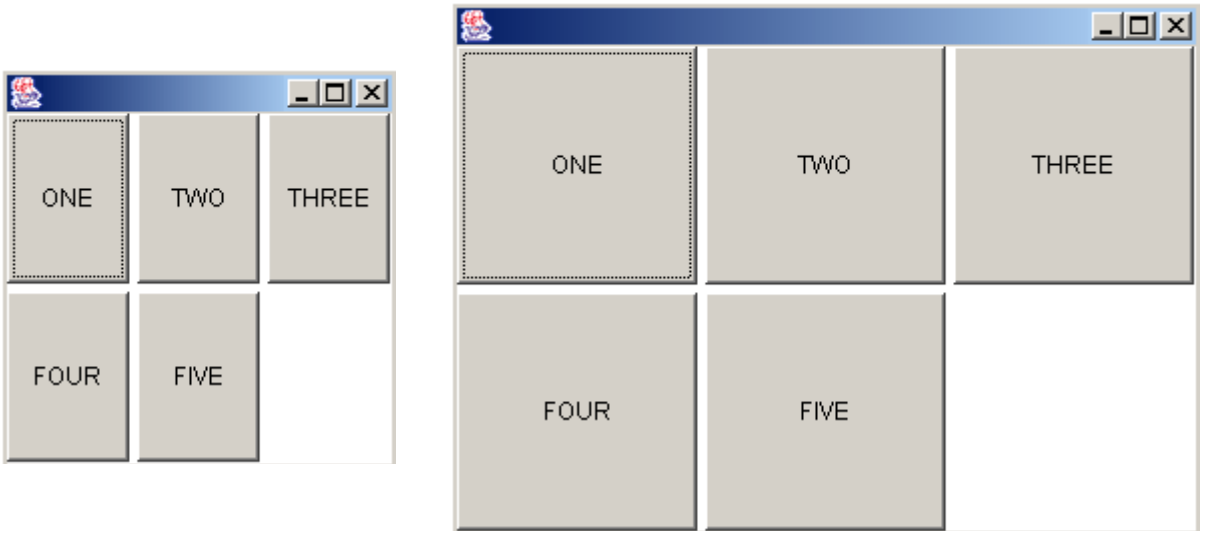

Figura 4: Saída do código exemplo

Layouts Complexos e Painéis

Pode-se criar layouts mais complexos, combinando os diferentes gerenciadores de layout com a utilização de painéis. Lembre-se que um *Panel* é um *contêiner* e um *Component* ao mesmo tempo. É possível inserir *Component*s no *Panel* e depois adicionar o *Panel* em uma região específica de um *contêiner*. Observe esta técnica utilizada no exemplo seguinte.

```
import java.awt.*;

public class ComplexLayout extends Frame {
  public static void main(String args[]) {
    ComplexLayout cl = new ComplexLayout();
    Panel panelNorth = new Panel();
    Panel panelCenter = new Panel();
    Panel panelSouth = new Panel();
    // Painel na posição Norte
    // Panels usam FlowLayout por padrão
    panelNorth.add(new Button("ONE"));
    panelNorth.add(new Button("TWO"));
    panelNorth.add(new Button("THREE"));
    // Painel na posição Centro
    panelCenter.setLayout(new GridLayout(4,4));
    panelCenter.add(new TextField("1st"));
    panelCenter.add(new TextField("2nd"));
    panelCenter.add(new TextField("3rd"));
    panelCenter.add(new TextField("4th"));
    // Painel na posição Sul
    panelSouth.setLayout(new BorderLayout());
    panelSouth.add(new Checkbox("Choose me!"),
    BorderLayout.CENTER);
    panelSouth.add(new Checkbox("I'm here!"),
    BorderLayout.EAST);
    panelSouth.add(new Checkbox("Pick me!"),
    BorderLayout.WEST);
    // Adicionando os painéis para o Frame
    // Frames usam BorderLayout por padrão
    cl.add(panelNorth, BorderLayout.NORTH);
    cl.add(panelCenter, BorderLayout.CENTER);
    cl.add(panelSouth, BorderLayout.SOUTH);
    cl.setSize(300,300);
    cl.setVisible(true);
  }
}
```

Aqui está a saída da classe.

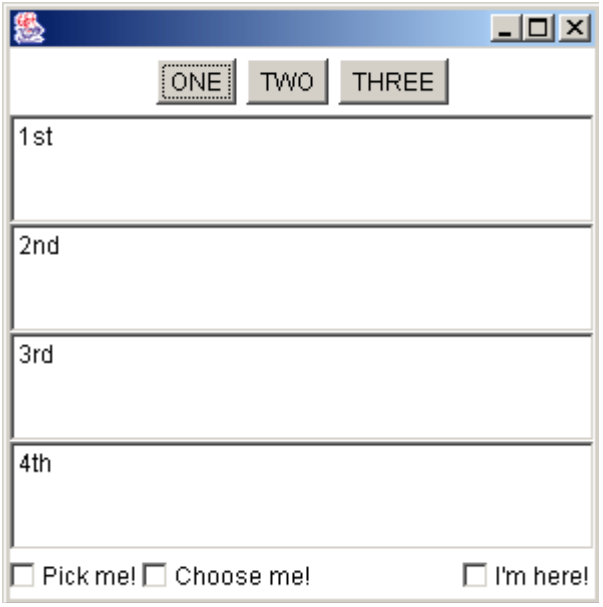

Figura 5: Saída do código exemplo

7.3. Componentes GUI Swing

Assim como o pacote AWT, o pacote Swing provê classes para a criação de aplicativos GUI. O pacote é encontrado em *javax.swing*. A diferença principal entre este dois é que os componentes *Swing* são totalmente escritos usando Java enquanto que o último não é. Como resultado, classes *GUI* escritas usando classes do pacote *Swing* possuem a mesma aparência e comportamento mesmo quanto executadas em sistemas operacionais diferentes. Mais ainda, *Swing* provê componentes mais interessantes para escolha de cores e o painel de opções.

O nome dos componentes *GUI Swing* são semelhantes aos nomes dos componentes *GUI AWT*. Uma diferença óbvia é na convenção de nomeação dos componentes. Basicamente, os nomes dos componentes Swing são simplesmente os nomes dos componentes AWT mas com o prefixo adicional J. Por exemplo, um componente no AWT é a classe *Button*. O componente correspondente para isto no pacote Swing é a classe *JButton*. Abaixo está a lista de alguns dos componentes GUI Swing.

Componente	***Descrição***
`JComponent`	A classe raiz para todos os componentes *Swing*, excluindo os contêineres *top-level*
`JButton`	Um botão. Corresponde à classe *Button* no pacote AWT
`JCheckBox`	Um item que pode ser selecionado ou não pelo usuário. Corresponde à classe *Checkbox* no pacote AWT
`JFileChooser`	Permite ao usuário selecionar um arquivo. Corresponde à classe *FileChooser* no pacote AWT
`JTextField`	Permite editar uma linha única de texto. Corresponde à classe *TextField* no pacote AWT
`JFrame`	Estende e corresponde à classe *Frame* no pacote AWT mas os dois são levemente incompatíveis em termos da adição de componentes para este contêiner. É necessário se obter o conteúdo do painel corrente antes de se adicionar um componente
`JPanel`	Estende a classe *JComponent*. Uma classe de contêiner simples mas não top-level. Corresponde à classe *Panel* no pacote AWT
`JApplet`	Estende e corresponde à classe *Applet* no pacote AWT. Também é incompatível com a classe *Applet* em termos da adição de componentes para este contêiner
`JOptionPane`	Estende a classe *JComponent*. Provê uma maneira fácil de mostrar caixas de diálogo
`JDialog`	Estende e corresponde à classe *Dialog* no pacote AWT. Geralmente usado para informar algo ao usuário ou solicitar uma entrada do usuário.
`JColorChooser`	Estende a classe *JComponent*. Permite ao usuário selecionar uma cor.

Tabela 8: Alguns componentes Swing

Para a lista completa dos componentes *Swing*, favor consultar a documentação API.

Definindo Contêineres Top-Level

Conforme mencionado, os contêineres top-level como o *JFrame* e o *JApplet* no Swing são incompatíveis com os no AWT. Isto ocorre em termos da adição de componentes ao contêiner. Em vez de adicionar diretamente um componente no contêiner como nos contêineres AWT, primeiro devemos obter o *panel* de conteúdo do contêiner. Para fazer isto, teremos que utilizar o método *getContentPane* do contêiner.

Um Exemplo JFrame

```
import javax.swing.*;
import java.awt.*;

public class SwingDemo {
  private JFrame frame;
  private JPanel panel;
  private JTextField textField;
  private JButton button;
  private contêiner contentPane;
  public void launchFrame() {
    // Inicialização
    frame = new JFrame("My First Swing Application");
    panel = new JPanel();
    textField = new JTextField("Default text");
    button = new JButton("Click me!");
    contentPane = frame.getContentPane();
    // Adicione componentes ao painel- usa FlowLayout por padrão
    panel.add(textField);
    panel.add(button);
    // Adicione componentes ao contentPane- usa BorderLayout por padrão
    contentPane.add(panel, BorderLayout.CENTER);
    frame.pack();
    // faz com que o tamanho do quadro seja baseado nos componentes
    frame.setVisible(true);
  }
  public static void main(String args[]) {
    SwingDemo sd = new SwingDemo();
    sd.launchFrame();
  }
}
```

Observe que o pacote *java.awt* ainda é importado pois os gerenciadores de layout sendo usados são definidos neste pacote. Além disso, o fato de se dar um título ao quadro e o empacotamento dos componentes dentro do quadro também é aplicável aos quadros AWT.

Dicas de Codificação:

Observe o estilo de codificação aplicado neste exemplo, ao contrário dos exemplos para o AWT. Os componentes são declarados como campos, um método launchFrame é definido, e a inicialização e adição dos componentes são todos feitos no método launchFrame. Simplesmente não estendemos mais a classe Frame. A vantagem de se utilizar este estilo se tornará aparente quando chegarmos a trabalhar com tratamento de eventos.

Aqui está um exemplo de saída.

Figura 6: Saída do código exemplo

Um Exemplo JOptionPane

```
import javax.swing.*;

public class JOptionPaneDemo {
  private JOptionPane optionPane;

  public void launchFrame() {
    optionPane = new JOptionPane();
    String name = optionPane.showInputDialog("Hi, what's your name?");
    optionPane.showMessageDialog(null,
      "Nice to meet you, " + name + ".", "Greeting...",
      optionPane.PLAIN_MESSAGE);
    System.exit(0);
  }
  public static void main(String args[]) {
    new JOptionPaneDemo().launchFrame();
  }
}
```

Observe como é simples solicitar dados do usuário. Este é o resultado da execução desta classe.

Figura 7: Saída do código exemplo

7.4. Usando AWT e Swing no Sun Java™ Studio Enterprise 8

1. Pressione um duplo clique no ícone do *Java Studio Enterprise 8* em seu computador

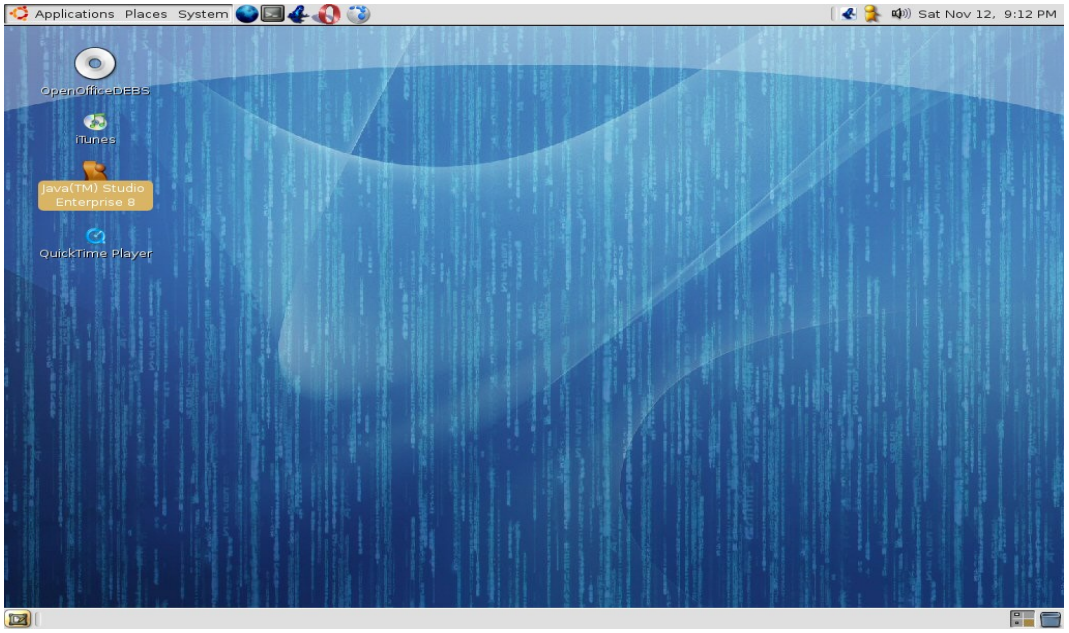

Figura 8: Iniciando o Java Studio Enterprise 8

2. Iniciar um novo projeto para a criação da interface com o usuário. Para isso, selecionar **File** -> **New Project**. A caixa de diálogo *New Project* mostrada na Figura 9 aparecerá.

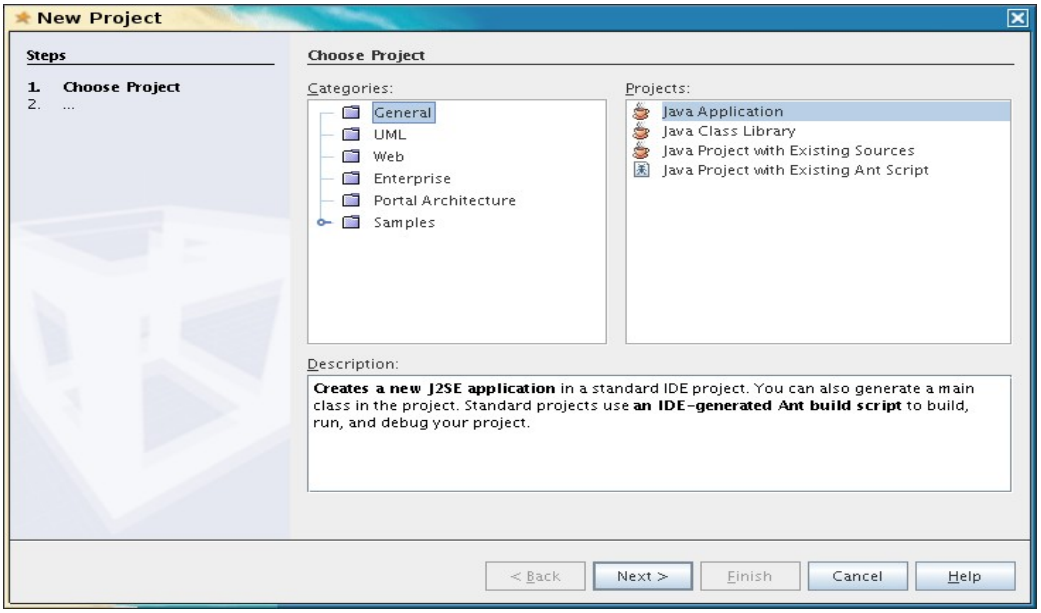

Figura 9: Selecionando o Tipo do Projeto

3. Desta caixa de diálogo, selecionar **General** -> **Java Application**. Pressionar o botão *Next* e a caixa de diálogo **New Java Application** aparecerá.

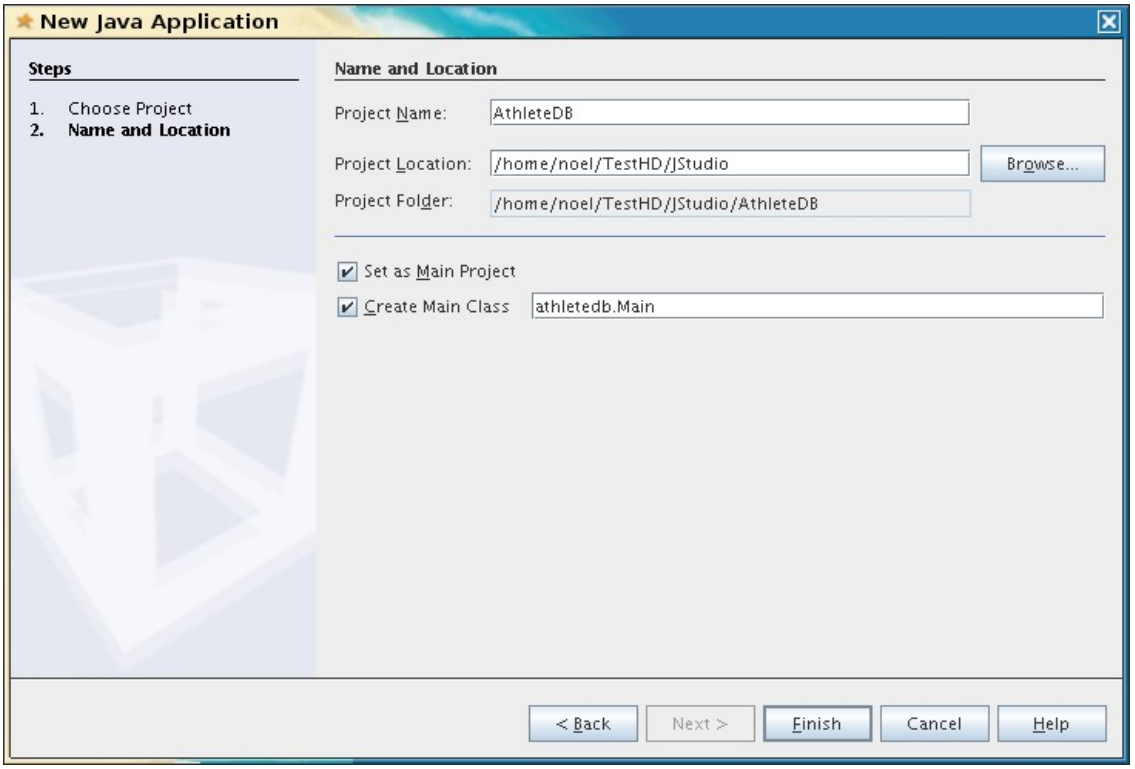

Figura 10: Caixa de Diálogo New Java Application

4. Na caixa de diálogo **New Java Application**, informa o nome do projeto e o diretório que será usado para manter todos arquivos relacionados ao projeto. Figura 10 mostra um exemplo. O nome do projeto é **AthleteDB** e o diretório é **/home/noel/TestHD/JStudio**.
5. O Java Studio Enterprise apresentará a inicialização padrão mostrada na Figura 11.

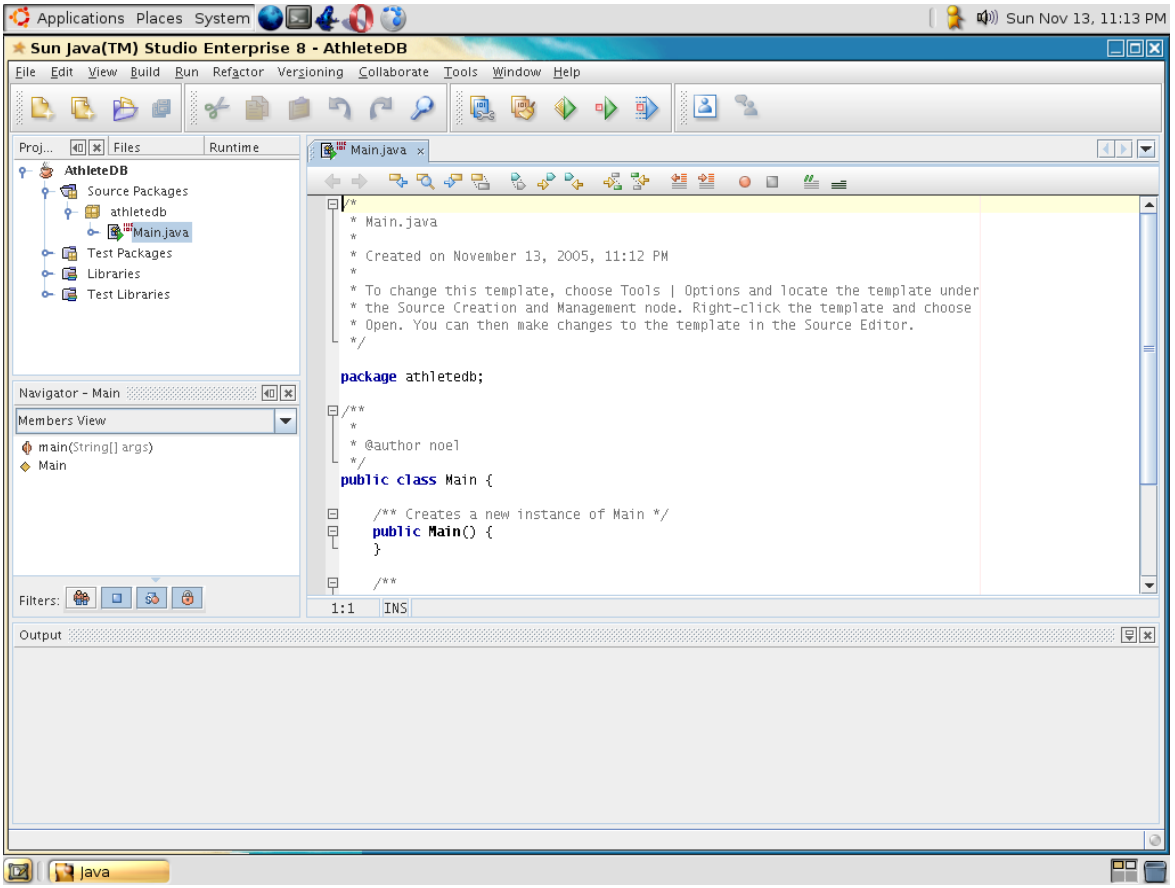

Figura 11: Tela Padrão de Inicialização

6. Para adicionar um quadro, selecionar **File** -> **New File** -> **Java GUI Forms** -> **JFrame Form**. Figura 12 e Figura 13 mostram como fazer isto.

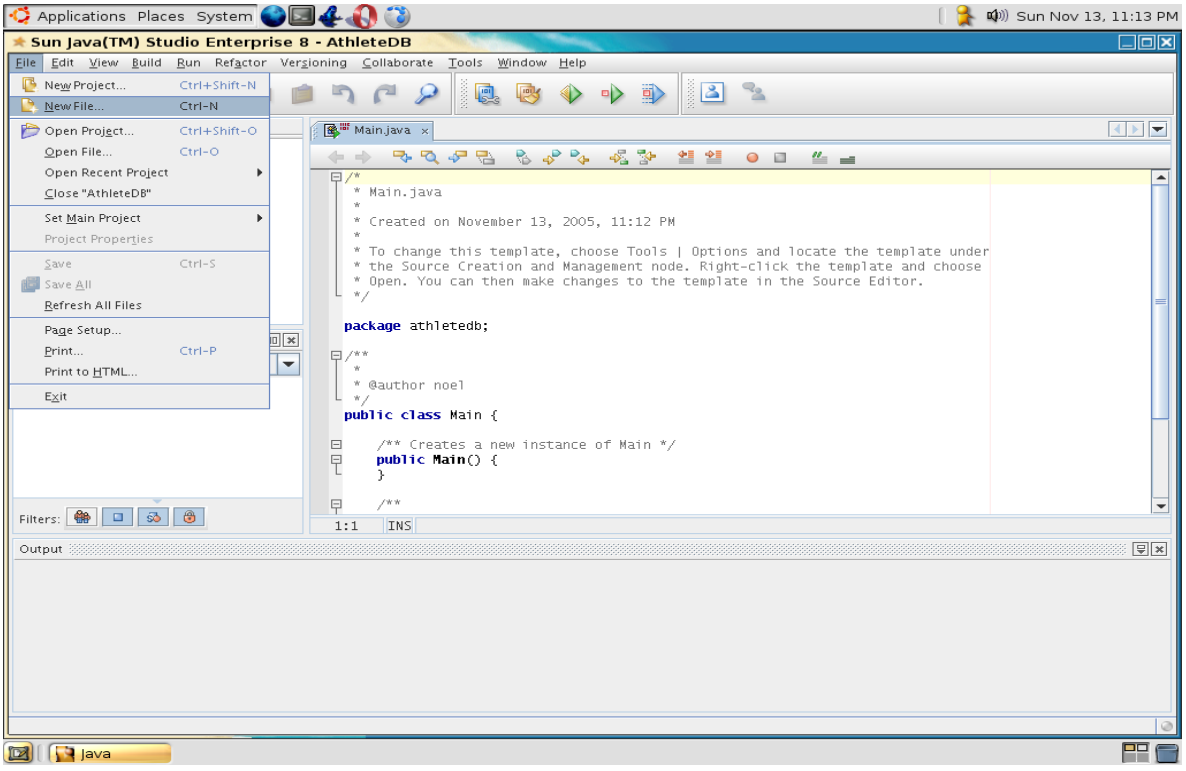

Figura 12: Selecionando a opção New File

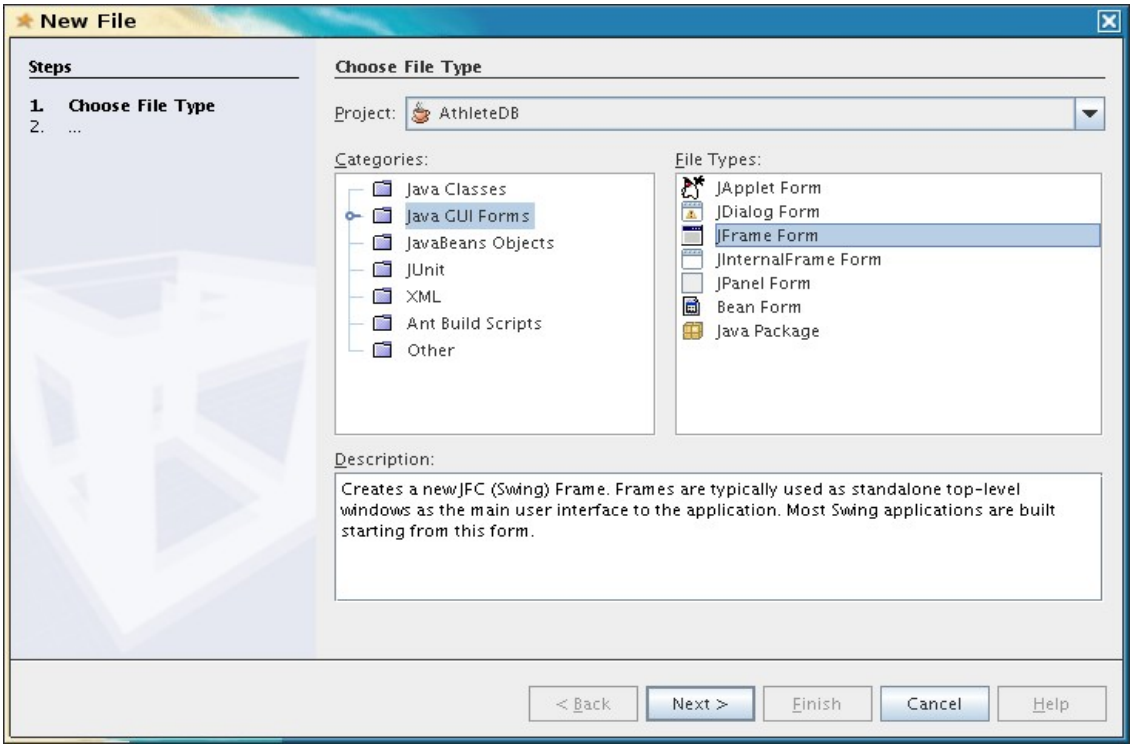

Figura 13: Selecionando Java GUI Forms e JFrame Form

7. Pressionar o botão *Next*. A caixa de diálogo *New JFrame Form* conforme mostrado Figura 14 irá aparecer. Informe o nome da classe. Neste caso, o nome da tela é **AForm.** Então, pressione o botão *Finish*.

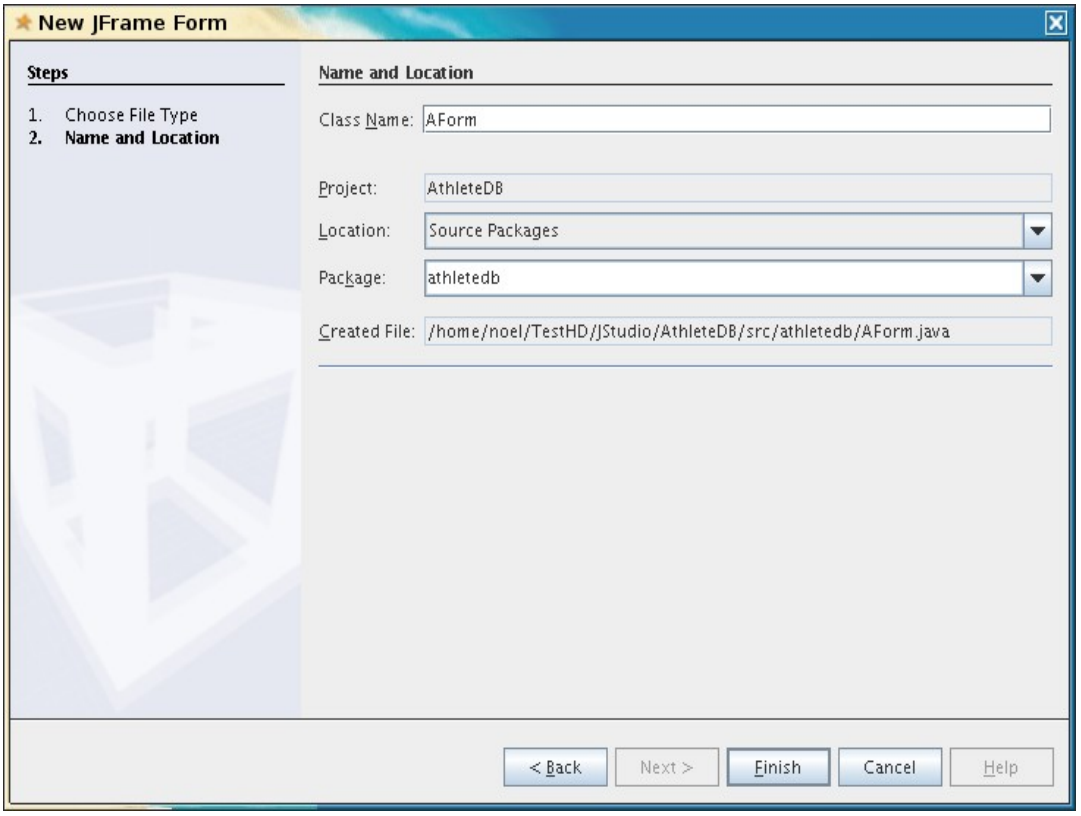

Figura 14: Caixa de Diálogo New JFrame Form

8. O *Design View* inicia-se e é mostrado na Figura 15.

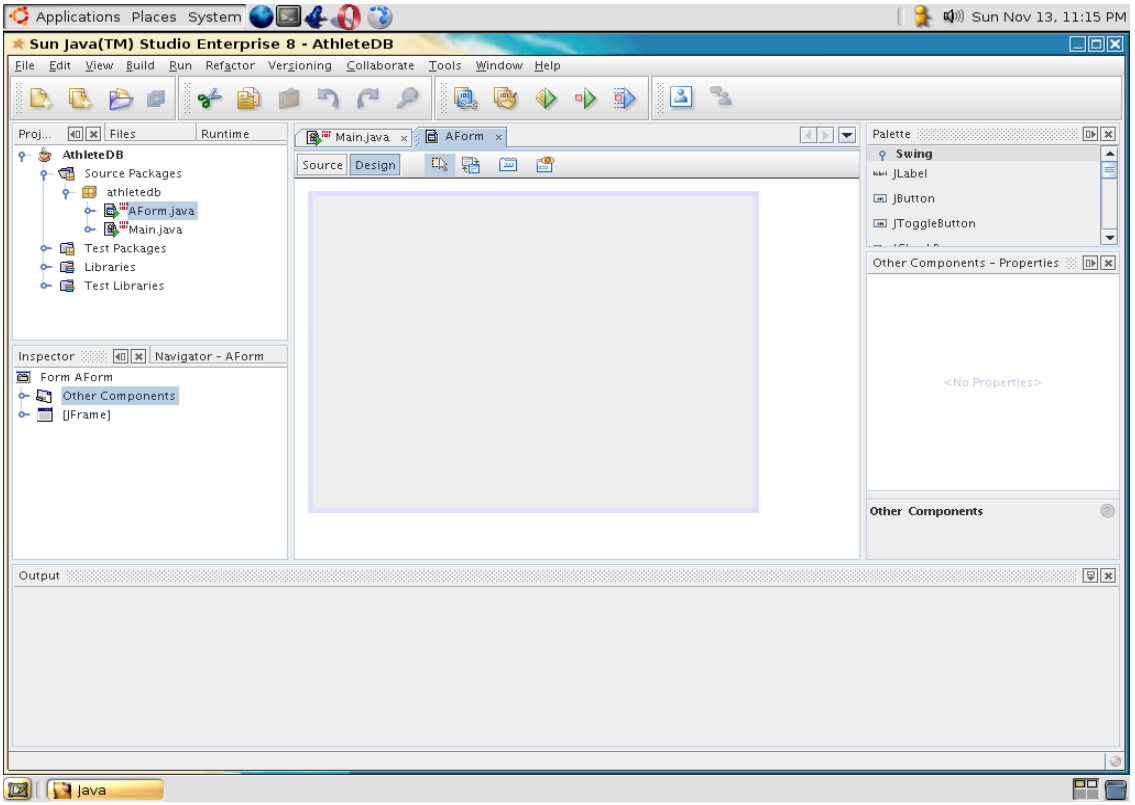

Figura 15: Inicialização do Design View

9. O *Palette Manager* está localizado na janela superior direita e abaixo está o *Component Properties*. O *Palette Manager* contém objetos dos pacotes *Swing*, *AWT*, *Layouts*, e *Beans*.

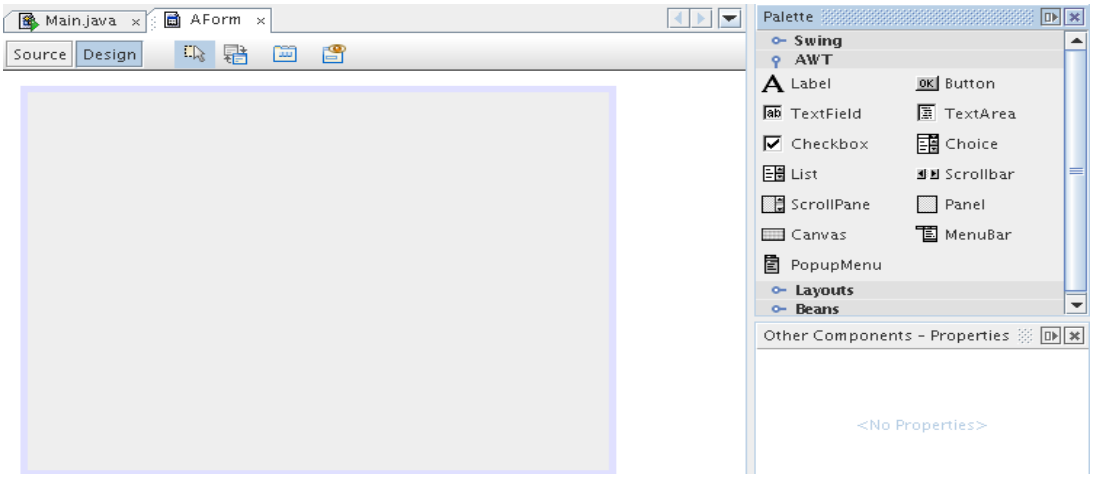

Figura 16: Palette Manager

10. Para utilizar o *Palette Manager* utilizando o estilo *DragNDrop* para a janela de *Design*, certifique-se que o *Selection Mode* está habilitado. É o primeiro ícone ao lado do *Design Tab*. Figura 17 mostra o *Design Tab*. O botão de seleção deve estar destacado; é o botão com a seta e a caixa.

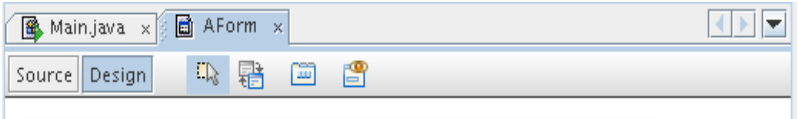

Figura 17 Selection Model Habilitado

11. O *Component Inspector* consiste de um visão geral dos objetos *GUIs* adicionados ao Frame. Figura 18 mostra o *Component Inspector*. É parte da janela rotulada *Inspector*. Uma Visão de

Árvore dos componentes é usada para mostrar a lista dos componentes.

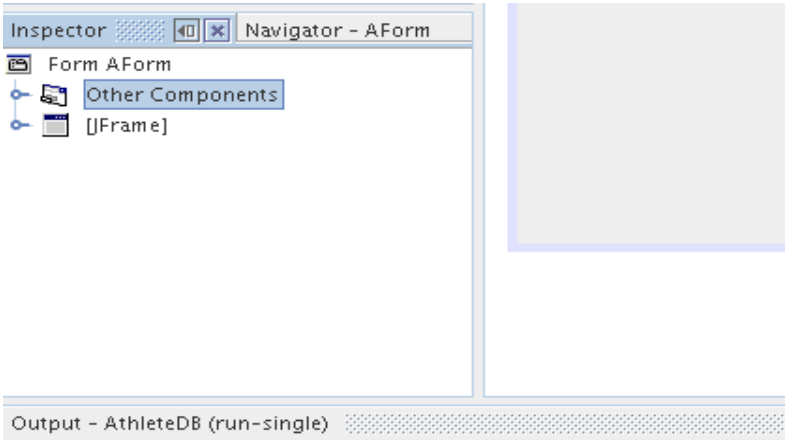

Figura 18: Component Inspector

12. Para construir ou salvar o projeto, selecionar **Build** -> **Build Main Project**. Isto é mostrado em Figura 19.

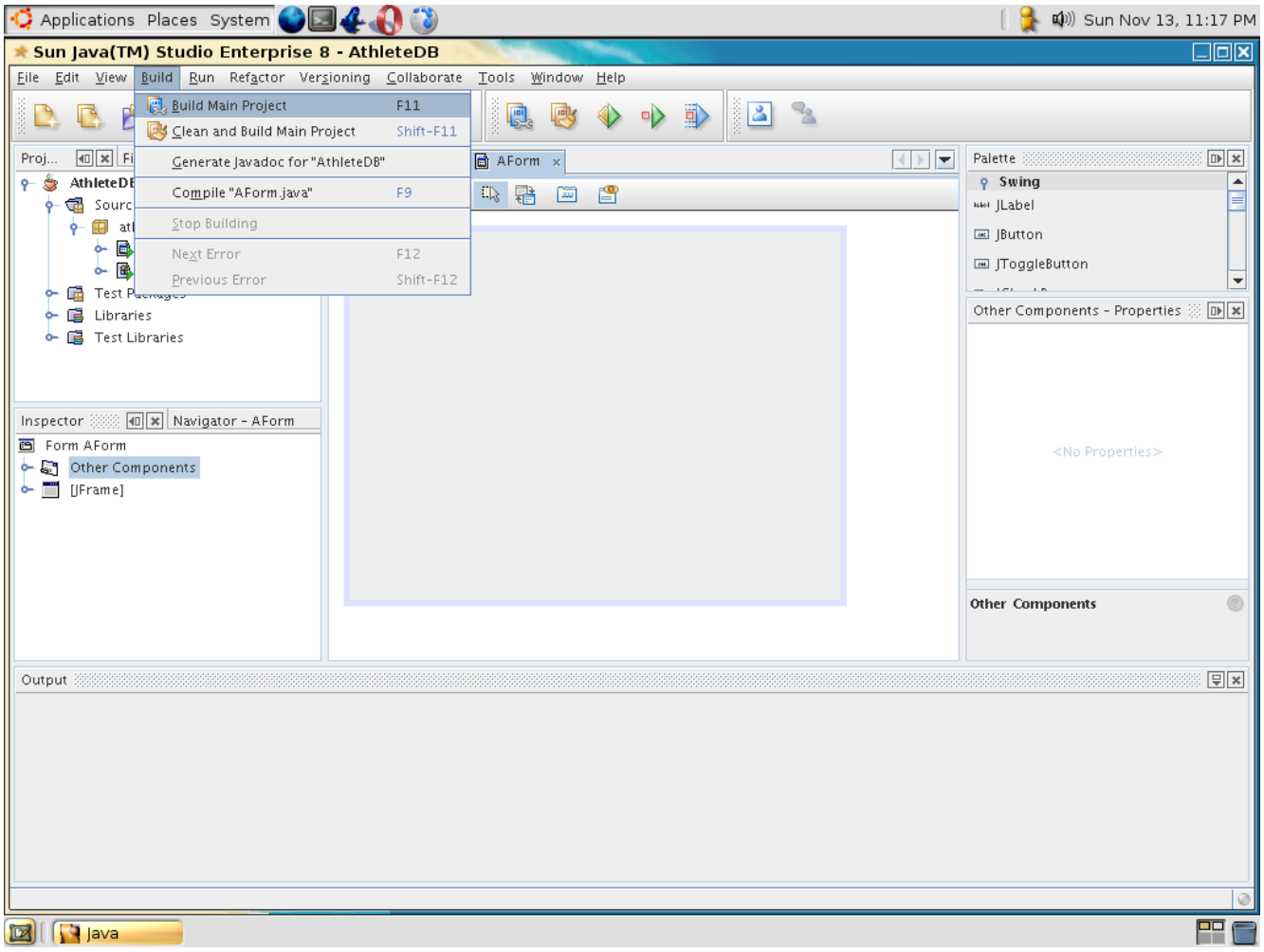

Figura 19: Opção Build

13. O Build Output é gerado no Output Tab na parte de baixo do programa. Isto é mostrado na Figura 20.

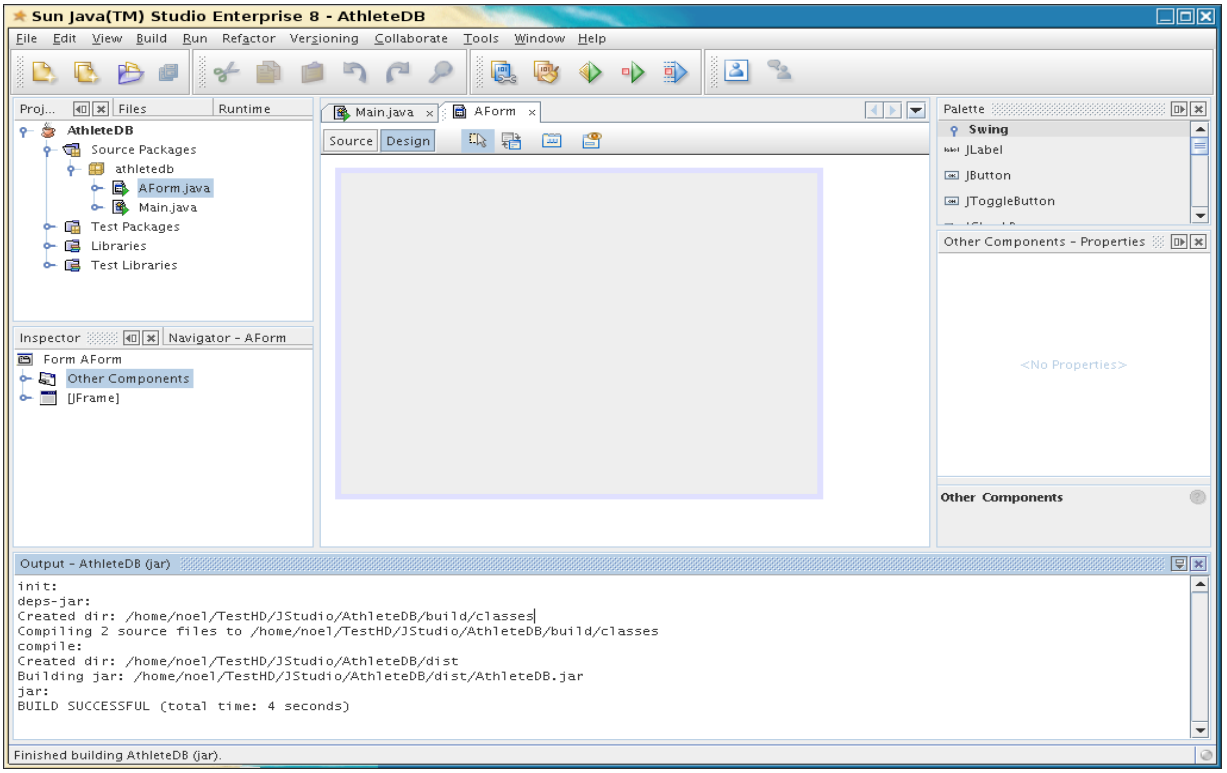

Figura 20: Build Output

14. Para executar o projeto específico, clique **Run** -> **Run File** -> **Run "AForm.java"**. Isto é mostrado na Figura 21.

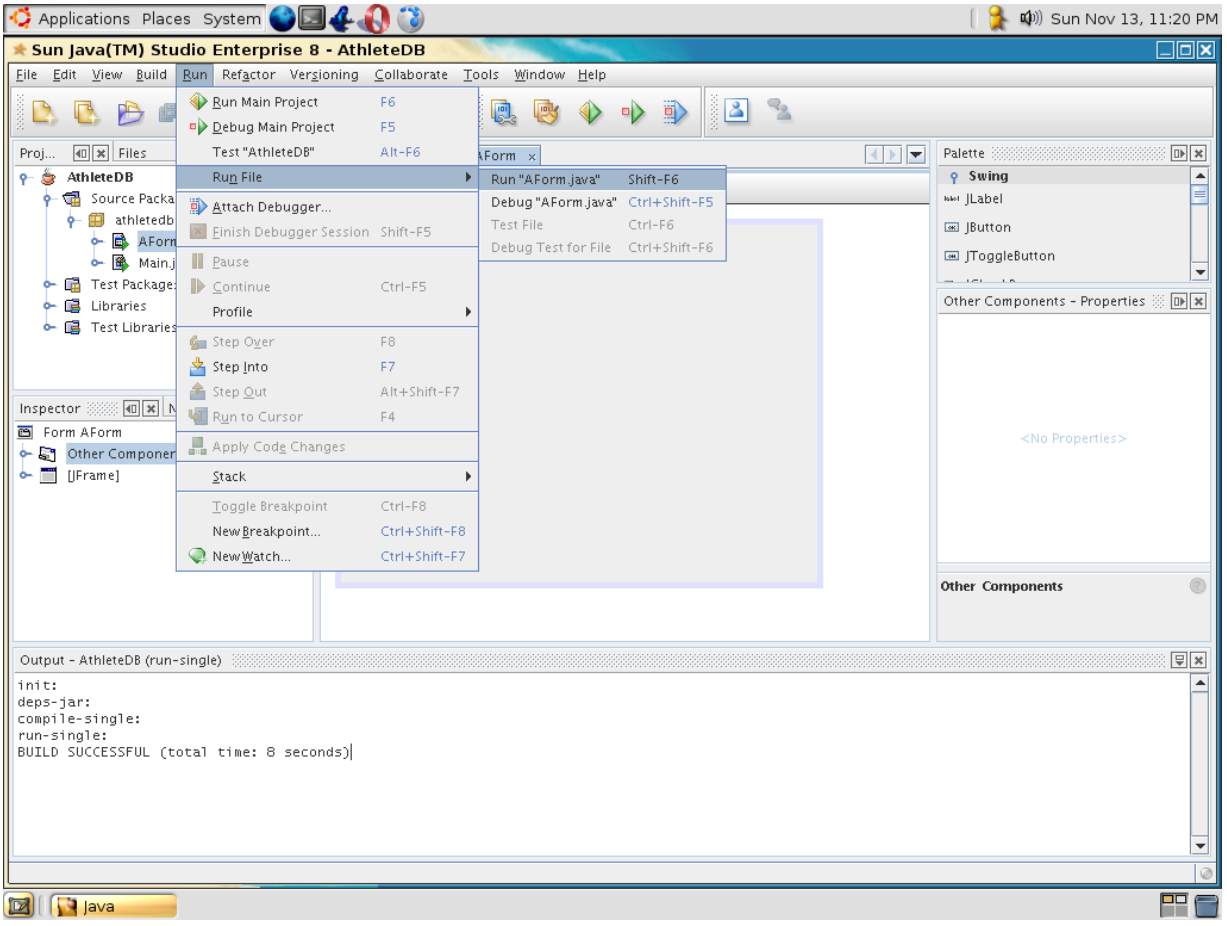

Figura 21: Opção Run

Esboçando o Formulário Athlete da Aplicação Exemplo

1. Clique com o botão direito em *JFrame* no *Component Inspector***.** Vai ser aberto um menu contendo o seu layout assim como outras propriedades. O *GridLayout* estará selecionado por padrão. Veja a Figura 22 para ver como isto é feito.

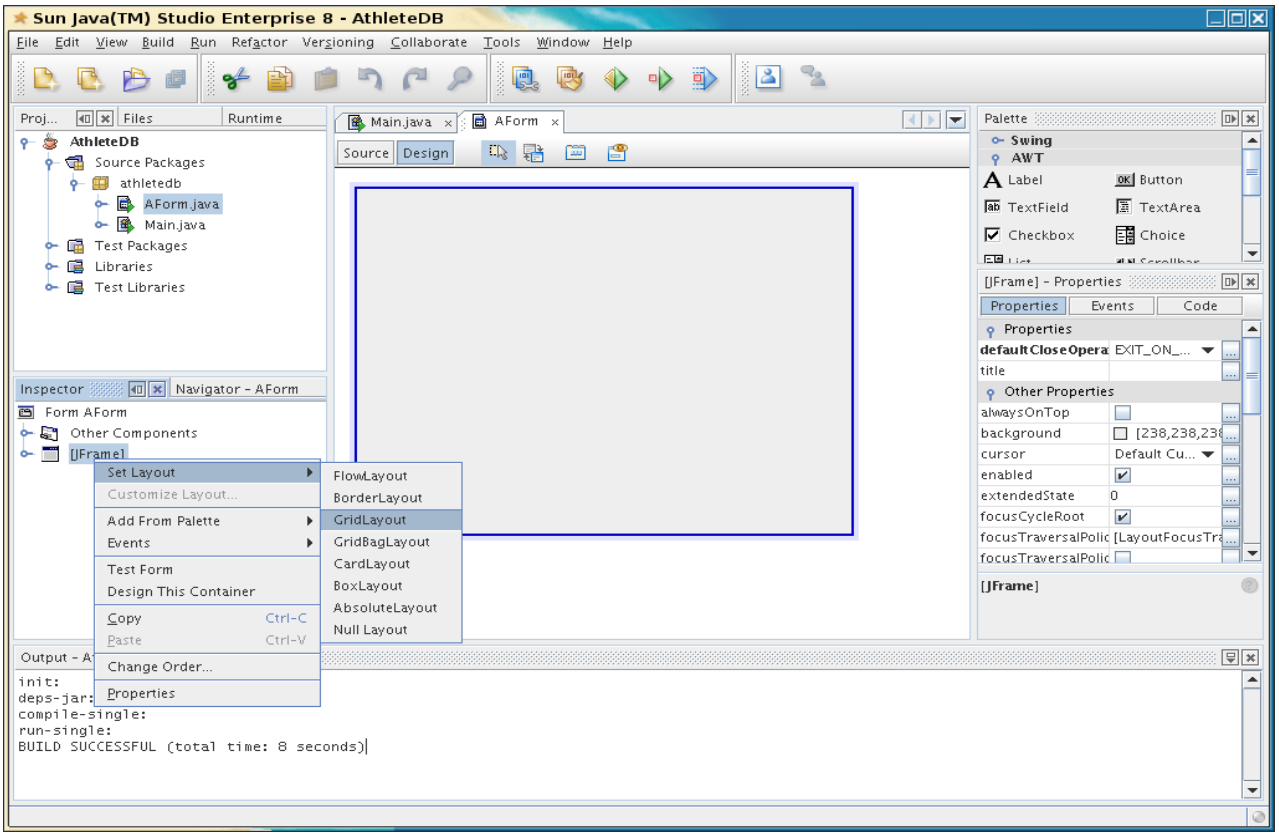

Figura 22: Definindo o Layout

2. Dê uma olhada no *Properties Tab* abaixo do Palette Manager, as propriedades para o *GridLayout* estão ilustrados e também podem ser modificadas. Figura 23 mostra uma instância do *Properties Tab*.

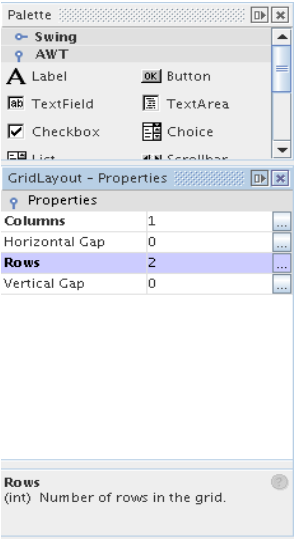

Figura 23: Properties Tab

3. Para adicionar um Panel, clique com o botão direito no nome do quadro na janela do Component Inspector. Neste exemplo, é o **JFrame1**. Selecionar **Add From Palette** -> **AWT** -> **Panel**. Figura 24 mostra isto.

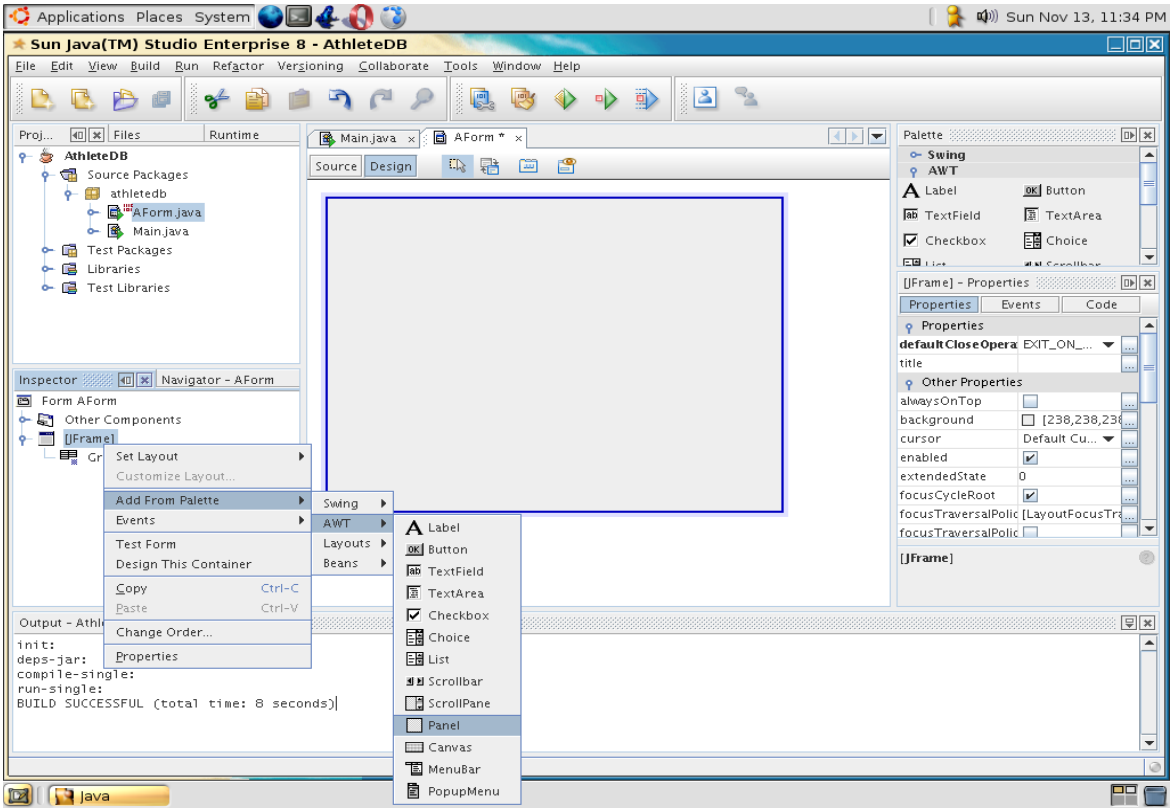

Figura 24: Adicionando um Panel

4. Para renomear o painel, clique com o botão direito no nome do painel na janela do *Component Inspector*. Selecionar *Rename*. Informar o novo nome do painel. Figura 25 mostra isto.

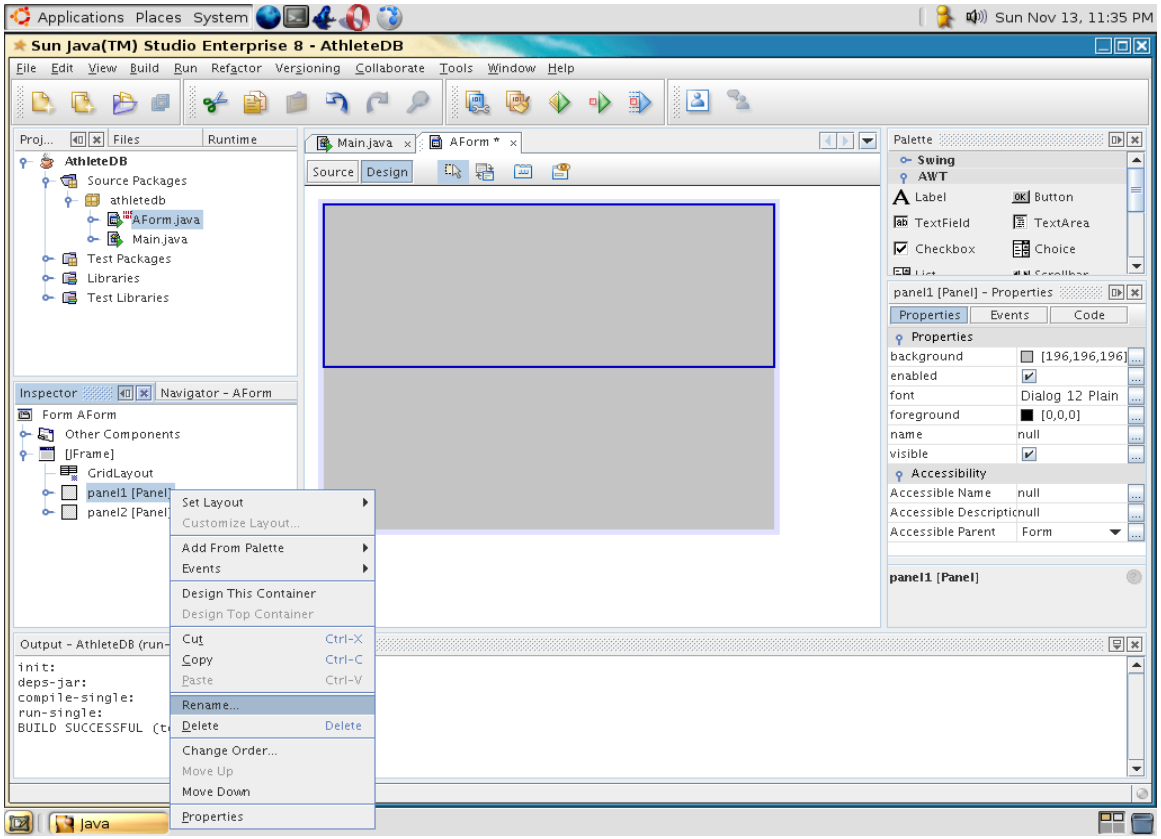

Figura 25 Renomeando um Panel

5. Adicione *Labels* utilizando o estilo *DragNDrop*. Clique no **ícone Selection Mode** e no **ícone Label.** Então, clique e arraste-o à janela *Design*. Note a mensagem no rodapé do programa

mostrando **Add Label to p1**. Figura 26 mostra isto.

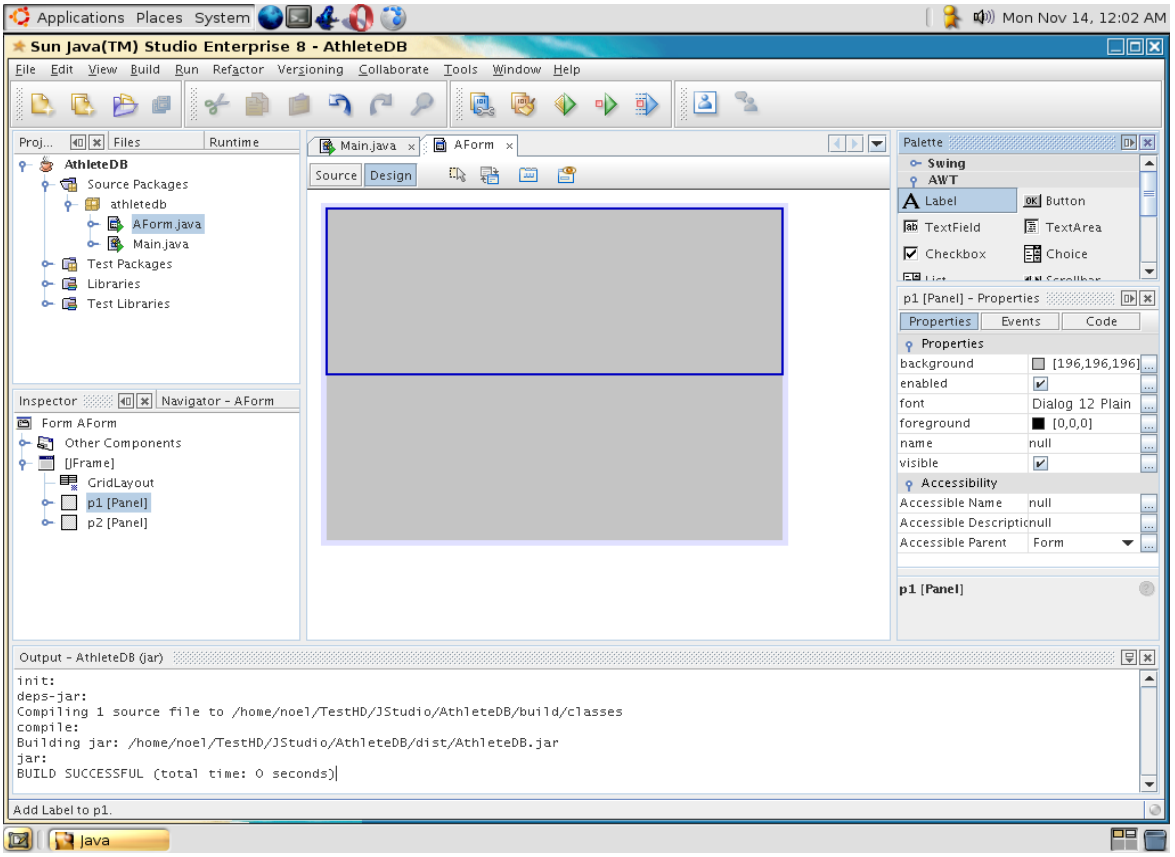

Figura 26: Adicionando um Label

6. O **Component Inspector** mostra agora que o Label **label1** foi adicionado ao Panel **p1** e o **Properties Tab** agora mostra os Properties, Events, e Code para aquele label. Figura 27 mostra isto.

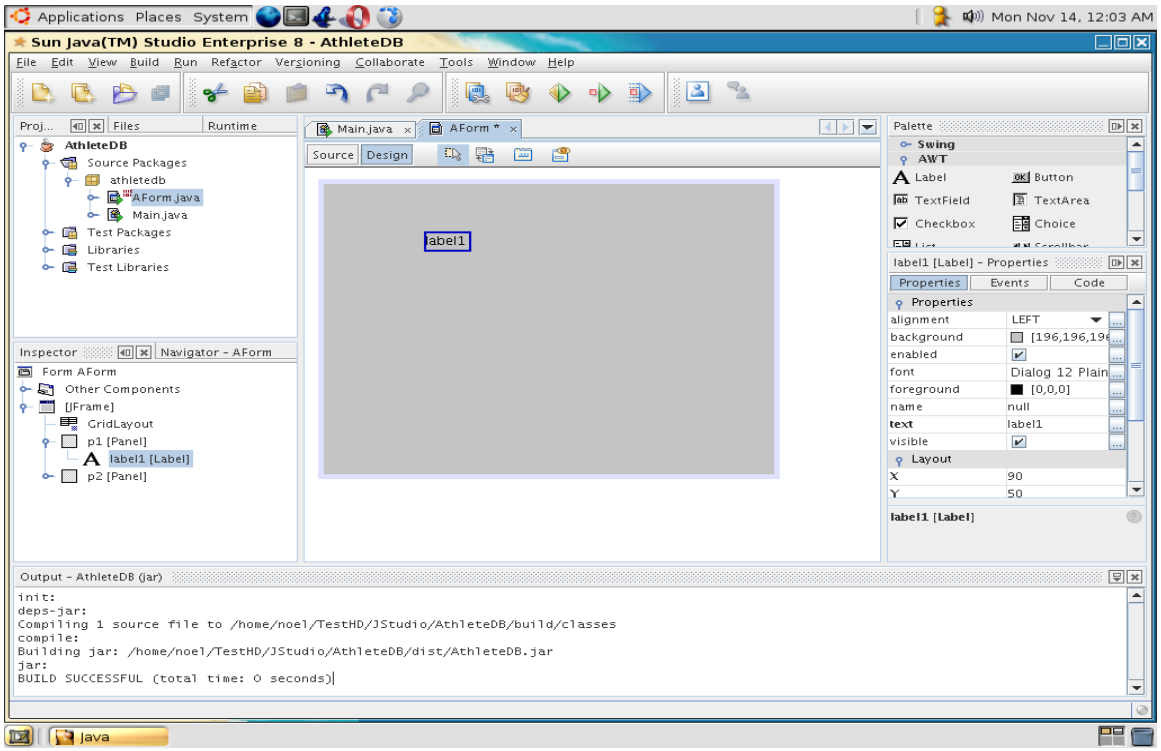

Figura 27 Exemplo de Label1 e Propriedades

7. A adição de GUIs ao Design também pode ser feito clicando-se com o botão direito o quadro ou painel no **Component Inspector**. Isto é mostrado na Figura 28.

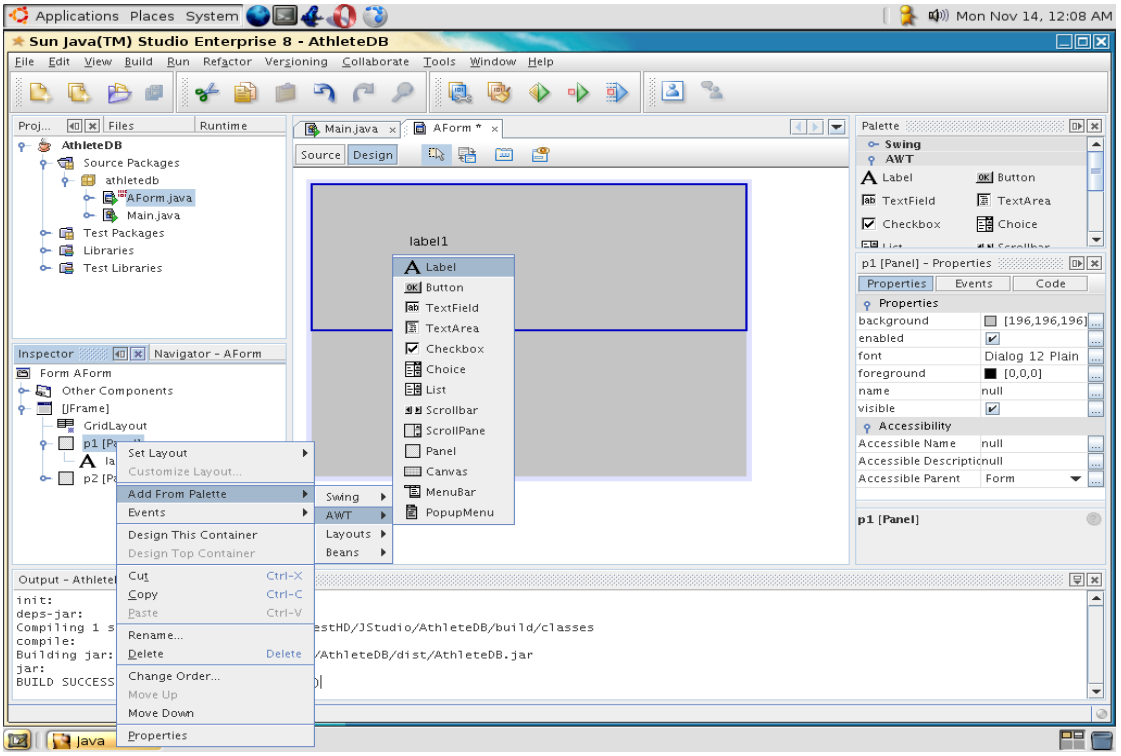

Figura 28: Clicando com o botão direito em Frame ou Panel

8. Continue adicionando e modificando as propriedades dos sucessivos objetos *Labels*, *TextFields*, *TextArea*, *Buttons* e os *Checkboxes* ao Panel **p1** e faça o mesmo no Panel **p2**. É possível também realizar um *DragNDrop* entre os quadros. Um exemplo de **AForm.java** é mostrado na Figura 29 com componentes de janela adicionais.

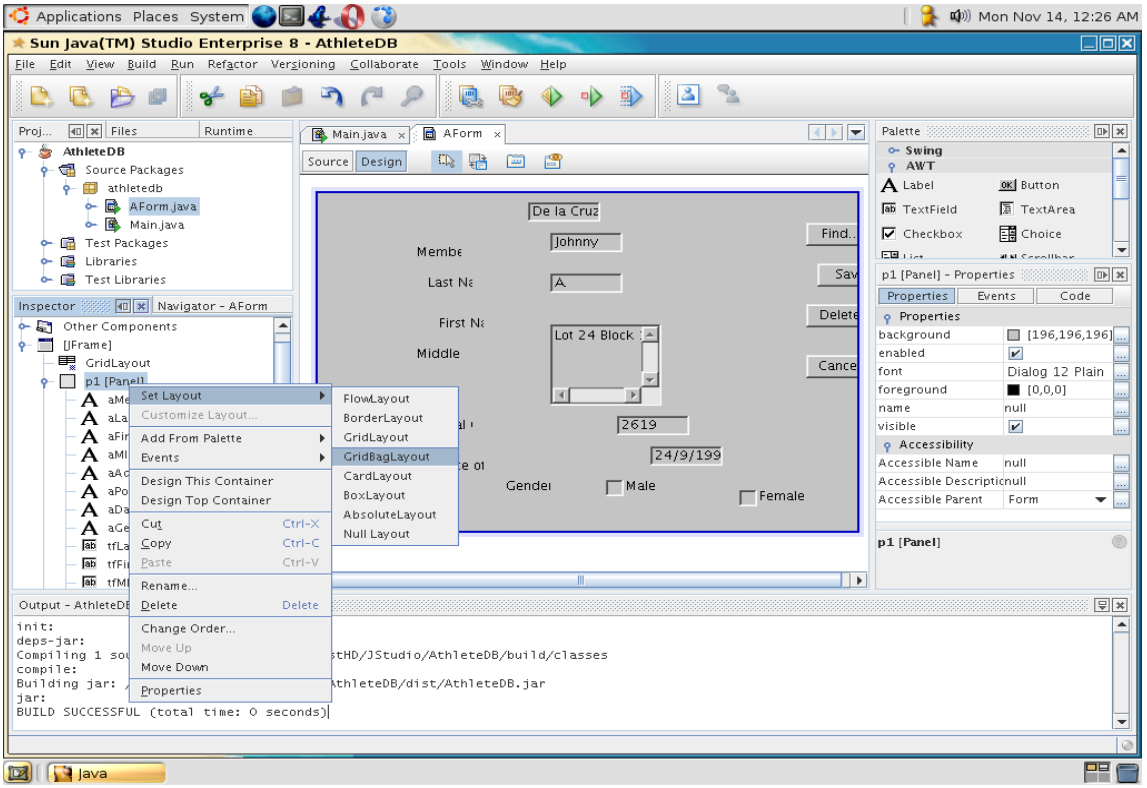

Figura 29 Exemplo AForm.java

9. Para testar o formulário, clique o **ícone Test Form** no Design Panel para ver o layout corrente. Figura 30 e Figura 31 mostram o que acontece.

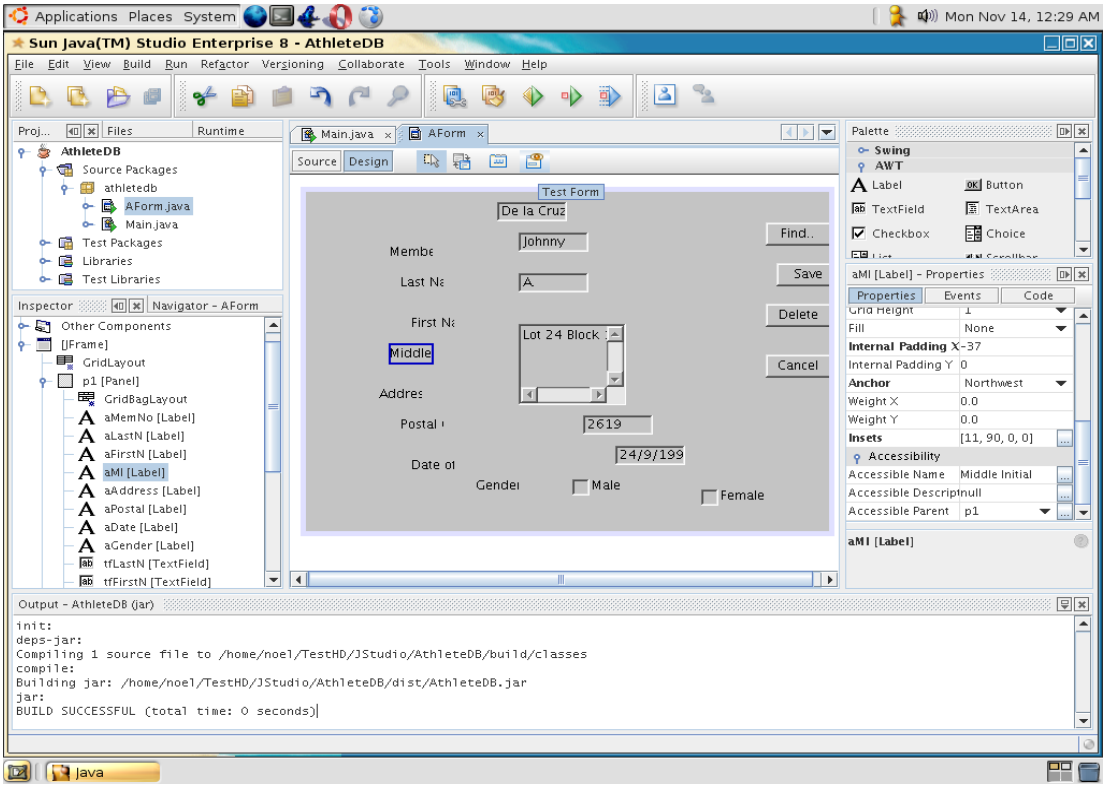

Figura 30: Clicando no Ícone Test Form

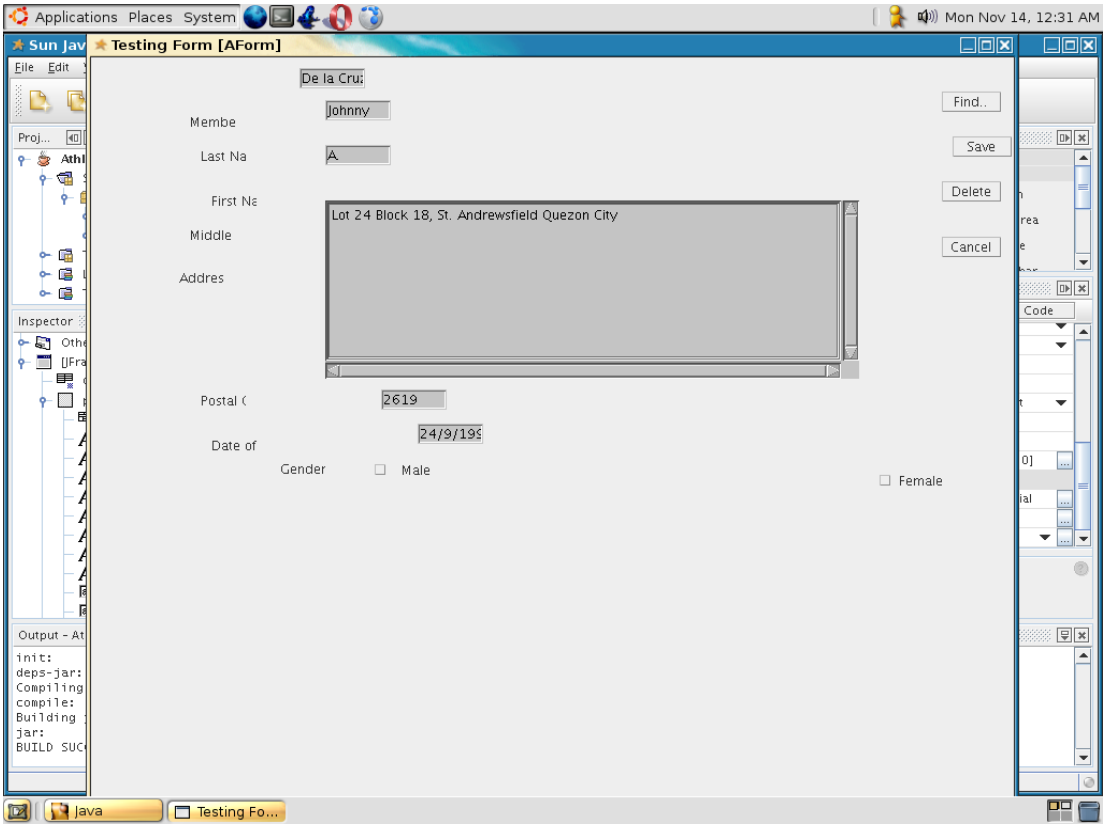

Figura 31: Executando o Test Form

10. Clique com o botão direito no **GridBagLayout** no *Component Inspector*. Selecione *Customize* para arrumar os *layouts* utilizando o estilo *DragNDrop* também. A janela *Customizer* será

mostrada. Figura 32 mostra um exemplo.

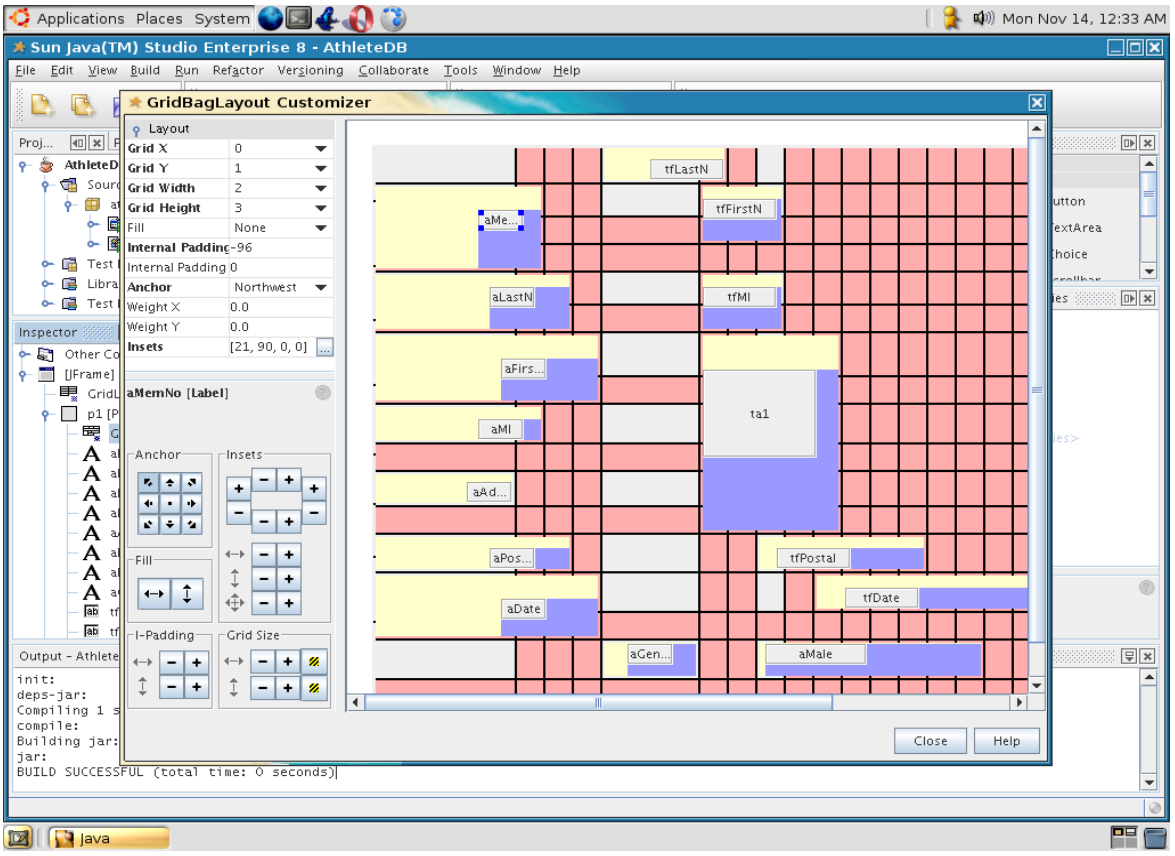

Figura 32: GridBagLayout Customizer

11. Realize um *DragNDrop* nos componentes para às suas posições corretas conforme a disposição mostrada na Figura 33.

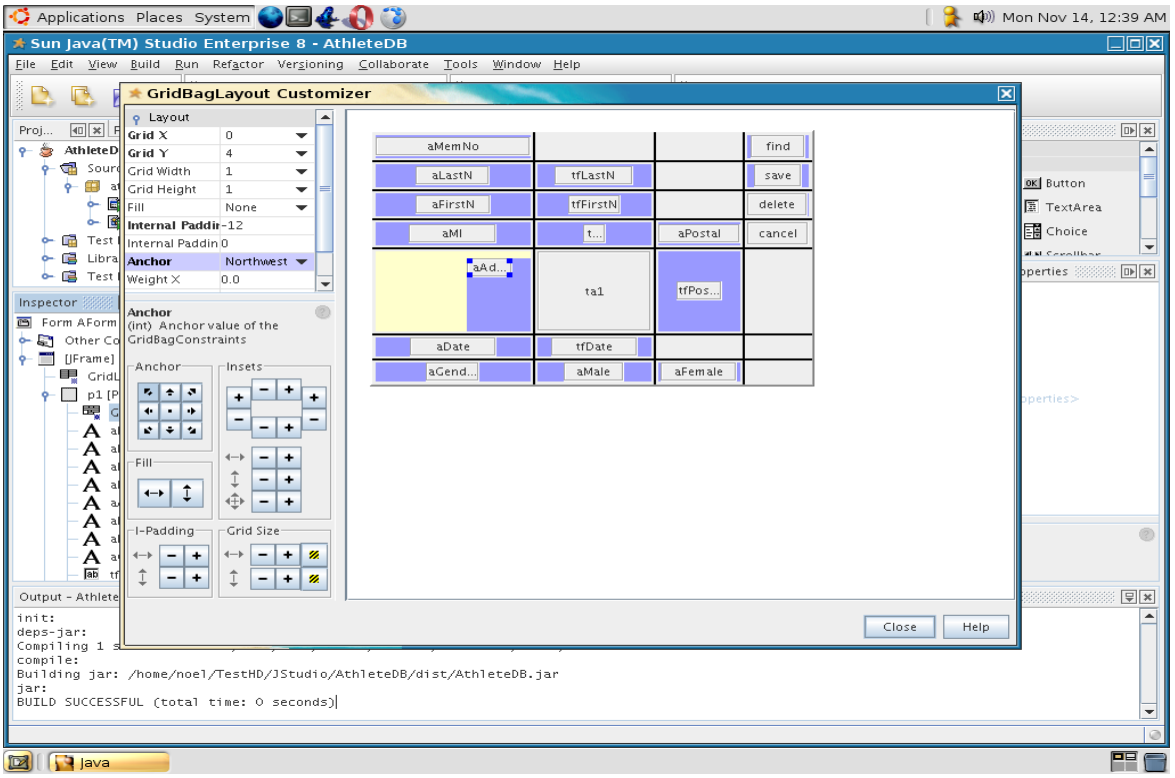

Figura 33: Novas Posições

12. Feche-o e clique no ícone **Test Form** para ver as diferenças. A tela deve se parecer com a tela na Figura 34.

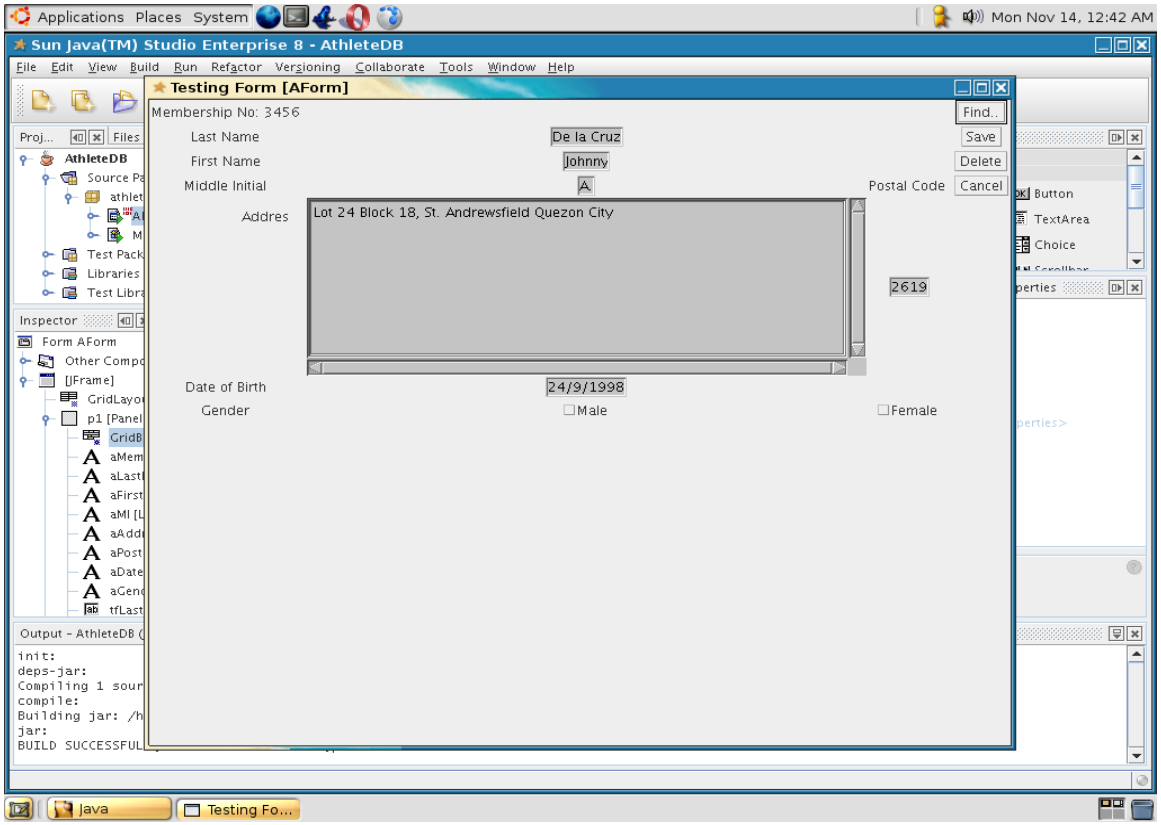

Figura 34: Novo GridBagLayout

13. Observe que tudo está centralizado. Volte ao **GridBagLayout** -> **Customize** para editar os valores associados a cada um, assim como outras modificações necessárias. O layout final quando o AForm.java é executado (BUILD ou Shift+F6) deve se parecer com a Figura 35.

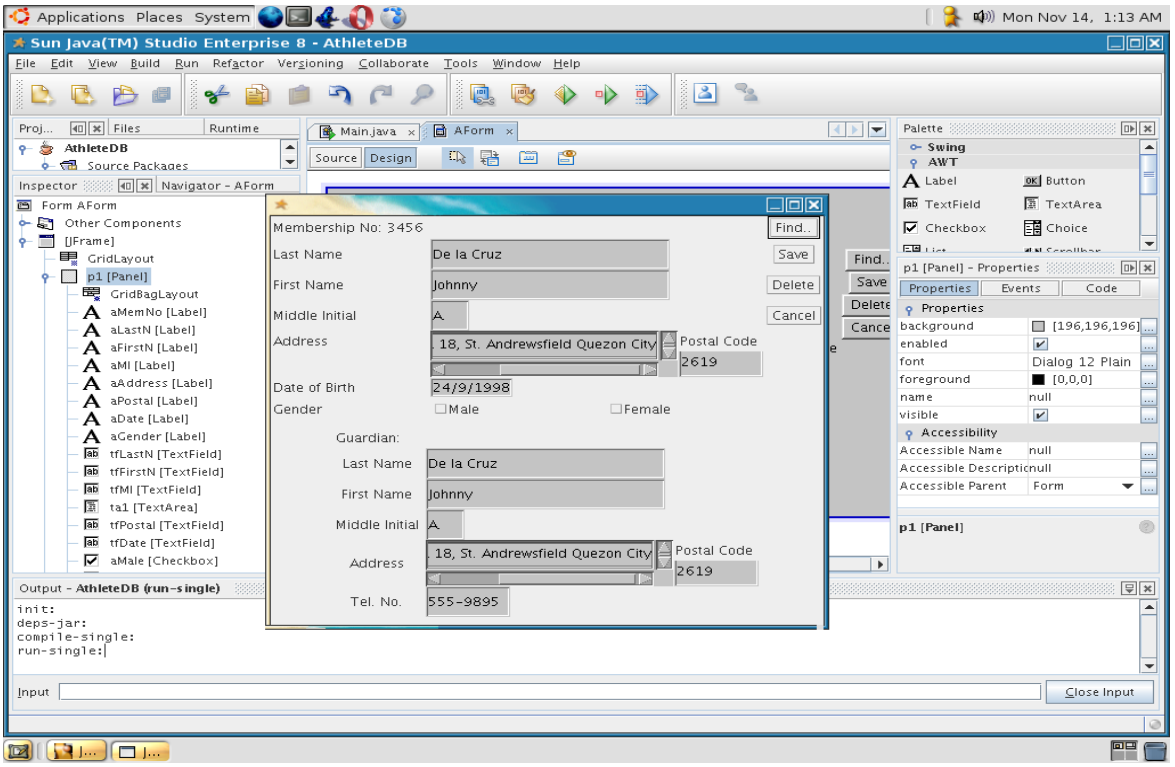

Figura 35: Novo Layout do Form

14. Para adicionar diálogos ao formulário, vá para **Component Inspector**, clique com o botão direito em **Other Components.** Selecione **Swing** -> **JDialog**. Faça o mesmo processo para os layouts.

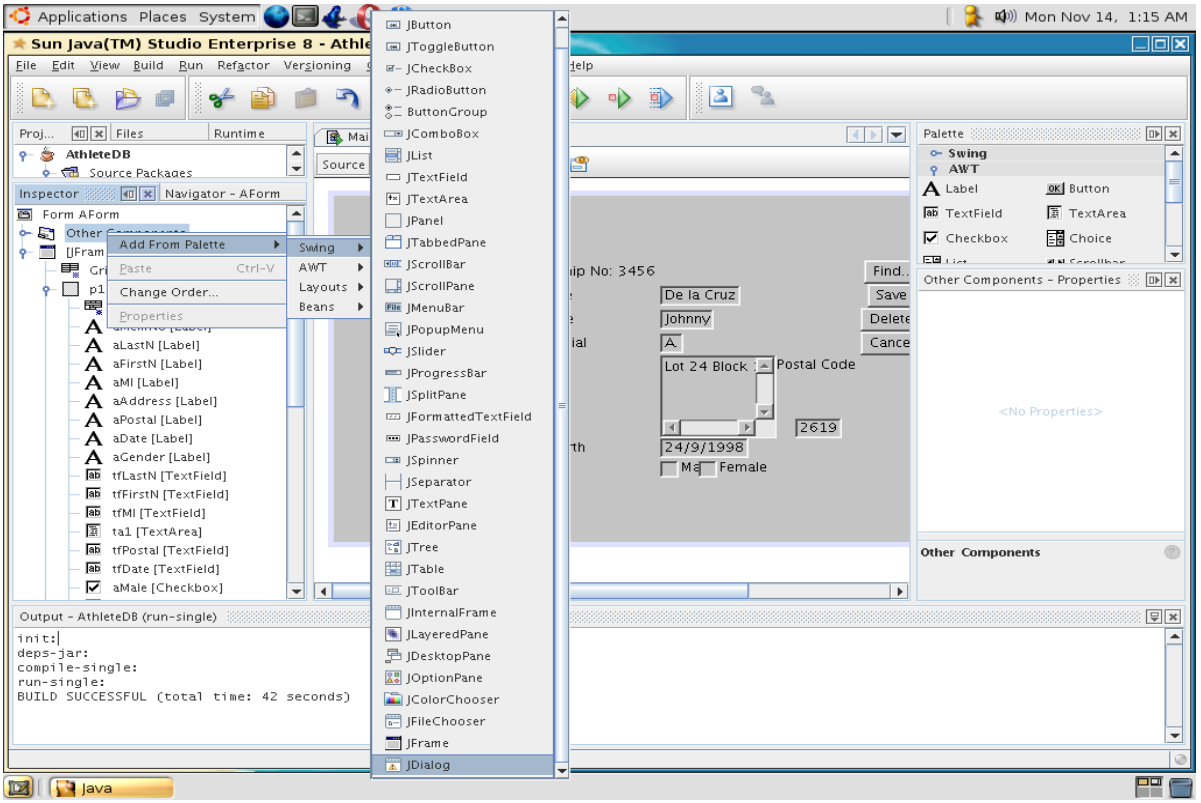

Figura 36: Adicionando um Dialog

15. Para habilitar o evento para o botão **Find** no **Athlete Form**, clique com o botão direito em seu nome no **Component Inspector**. Selecione **Events** -> **Action** -> **actionPerformed**.

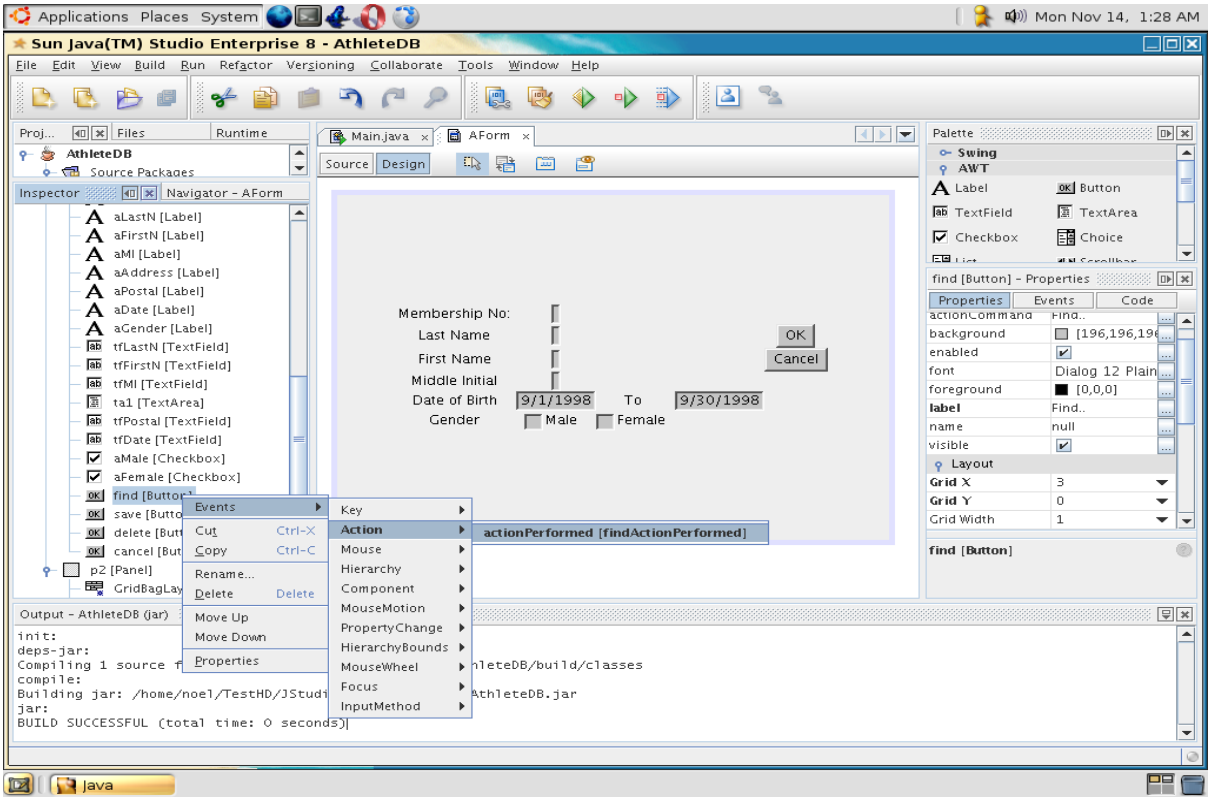

Figura 37: Habilitando Eventos para o Form

16. Isto irá selecionar o tab **Source** ao lado do tab Design na janela Design. Modifique o código de tal maneira que o Diálogo **Find An Athlete** seja visível. Faça o mesmo para os botões **OK** e **Cancel** no Diálogo.

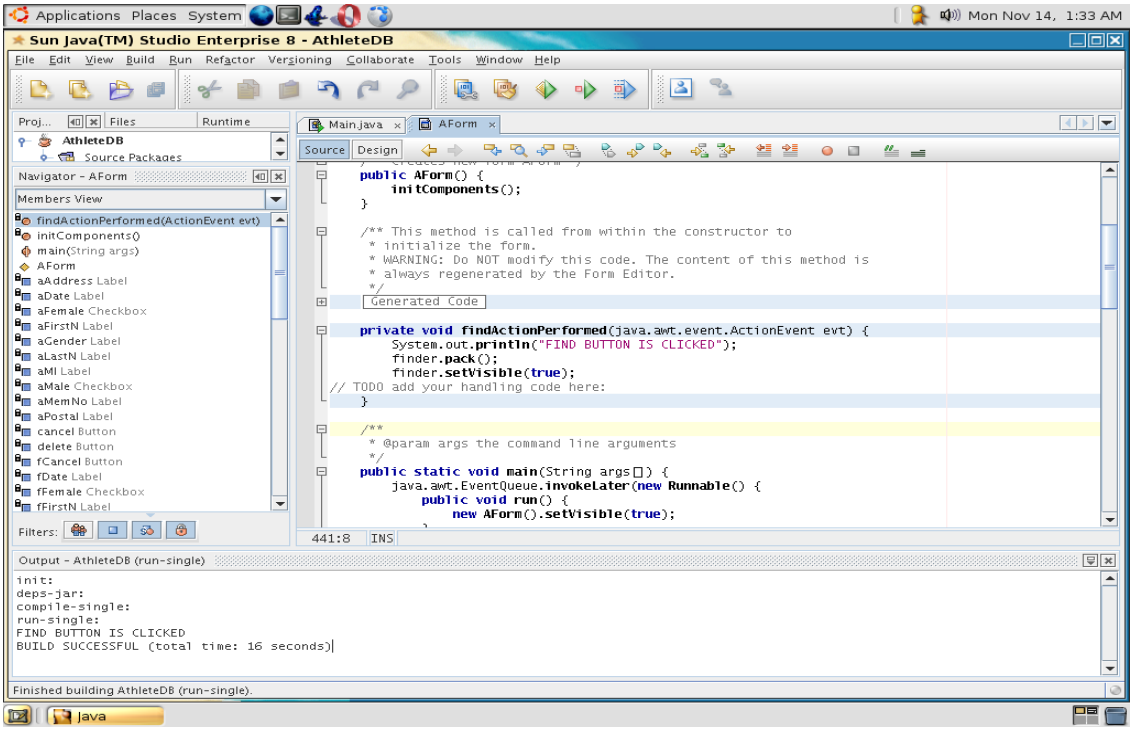

Figura 38: Setando o Diálogo Find Athlete como Visível

17. Teste o formulário. Do Formulário **Athlete**, clique o Botão **Find**. Ele deve mostrar o Formulário **Find An Athlete**. Deste formulário, clique no botão **OK**. Será mostrado o Formulário **Athlete List**.

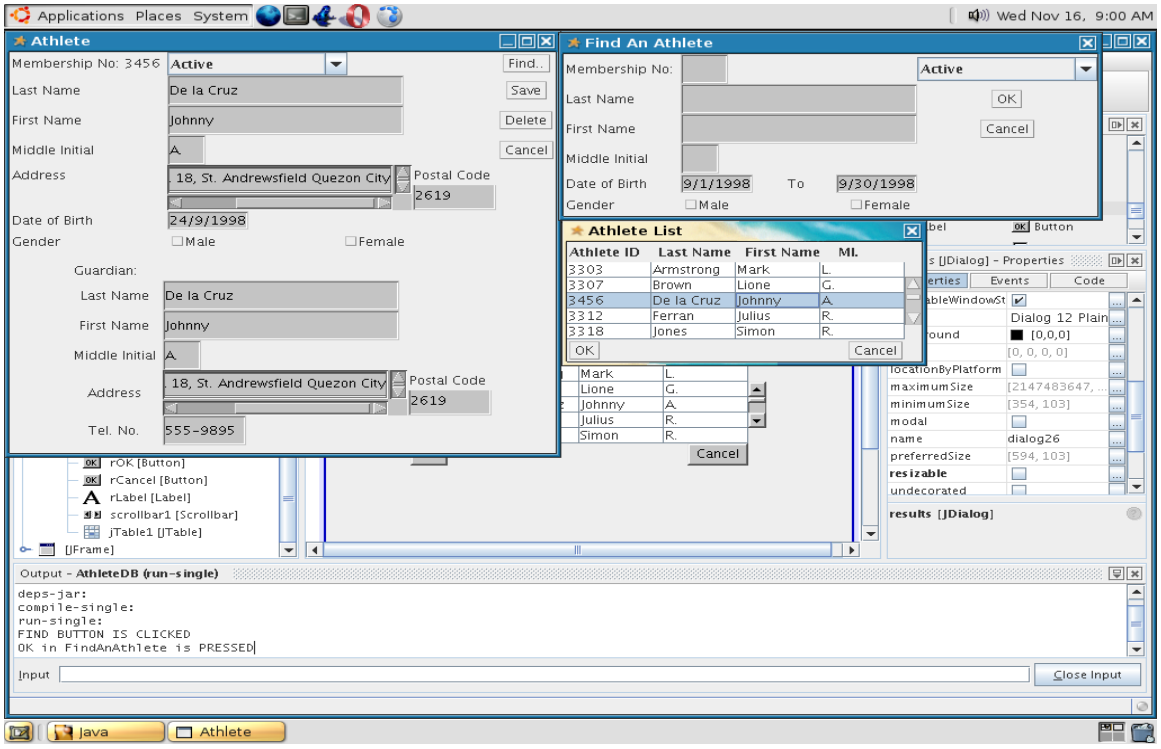

Figura 39 Testando todos os formulários

18. Reconstrua e Execute.

8. Controlando a Versão do Software

É muito difícil controlar o desenvolvimento do software sem a ajuda de um Sistema de Versionamento. Esta seção introduz o Sistema de Versionamento Concorrente (CVS) e como é usado no Java™ Studio Enterprise 8, assume que existe um servidor CVS disponível para armazenar os pacotes de software.

8.1. O que é CVS?

CVS é a abreviatura **C**oncurrent **V**ersioning **S**ystem. É um sistema aberto de controle de versão e colaboração. Colocado de forma simplesmente, é um pacote de software que gerencia o desenvolvimento de software feito por um time. Usando-o, pode-se registrar a história dos arquivos fontes e documentos.

8.2. Colaboração com CVS

Para que os times sejam capazes de colaborar e obter atualizações de seus trabalhos, o seguinte deve ser feito:

- *Inicialização de um repositório CVS*. Um repositório é um armazenamento persistente que coordena o acesso multi-usuário aos recursos sendo desenvolvidos por um time.
- *Especificação da localização do repositório CVS*.
- *Entrega do trabalho naquele repositório*.

Após executar estes passos, os participantes do time podem atualizar as versões dos recursos do projeto que eles estão desenvolvendo.

8.3. Setando o repositório CVS no Sun Java™ Studio Enterprise 8

Inicializar o repositório CVS no Sun Java™ Studio Enterprise 8. Inicicializando o repositório CVS no JSE8 é simples. Suponhamos que desejamos compartilhar a classe mostrada abaixo com os outros participantes do time. Figura 40 mostra isto no Java Studio Enterprise 8. Para iniciar, selecione **Versioning →Version Manager.** Clique **Add** e o *New Generic CVS wizard* aparece e permite que sejam informadas as configurações de versionamento necessárias. No **Version Control System Profile**, escolha **CVS** e especifique o caminho do repositório no campo **Repository Path**. Além disso, deve ser especificado o **CVS Server Type**. Isto é importante de modo a permitir que outros participantes do time tenham acesso aos recursos do projeto. O valor padrão é local. Pode ser necessário a ajuda do administrador de repositório para preencher as informações necessárias. Figura 41 mostra a inicialização do CVS.

```
package welcome;

import javax.swing.JOptionPane;

public class Main {
  public static void main(String[] args) {
    JOptionPane.showMessageDialog(
      null, "Welcome\nto\nJEDI\nProject");
    System.exit(0);
  }
}
```

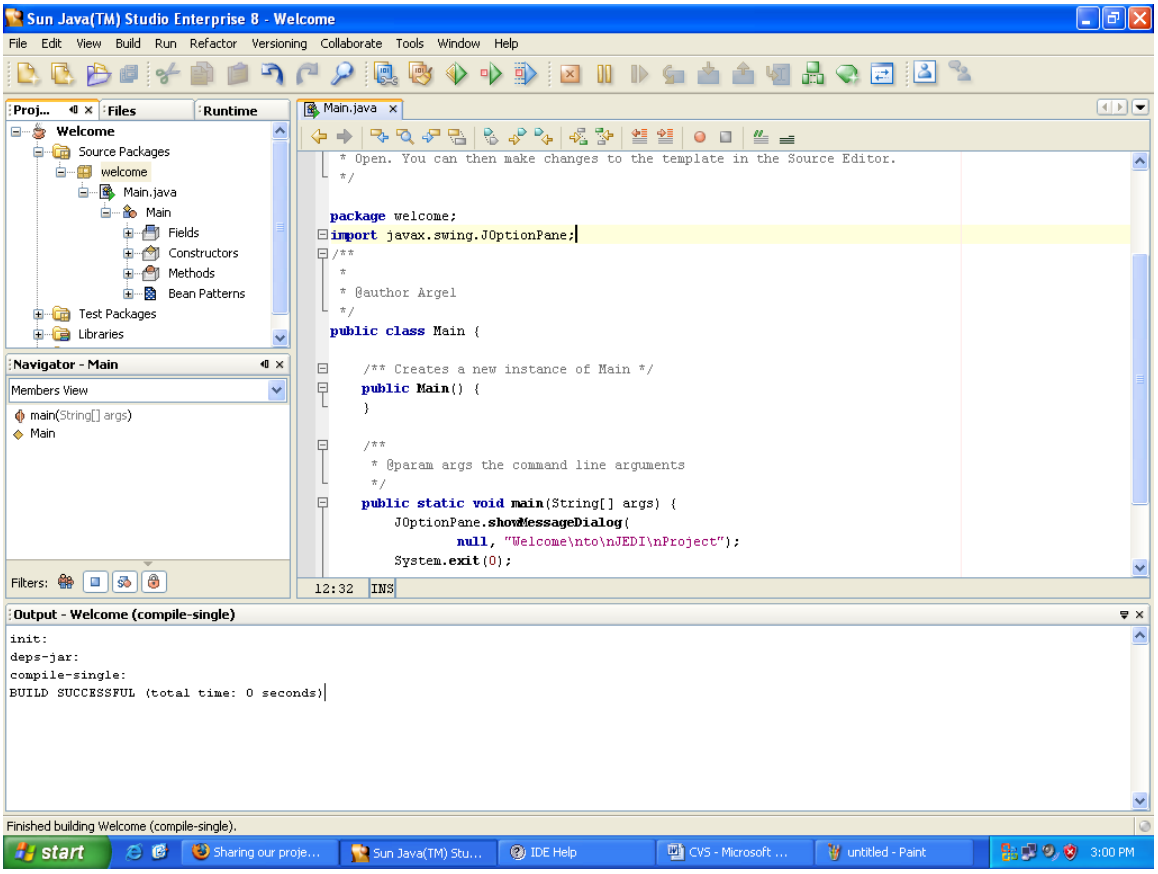

Figura 40: Exemplo de Código a ser Compartilhado

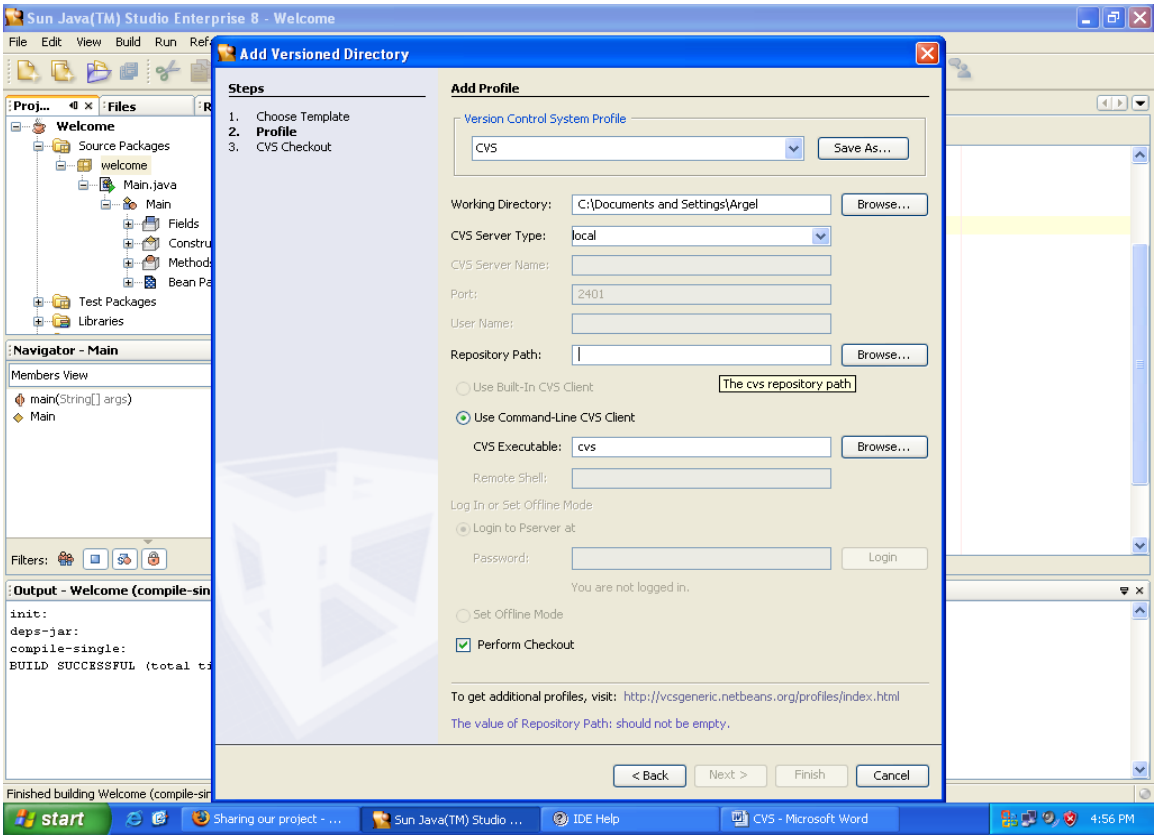

Figura 41:Inicializando o Repositório

8.4. Checkout no Repositório CVS

Da opção **CVS** sob **Versioning**, podemos achar a opção **CVS Checkout**. Isto leva à página CVS Checkout, que pode ser usada para especificar os módulos em que serão realizados *checkout*. Ao informar *checkout* em todo o repositório selecione a caixa **All** e para dar checkout em componentes específicos, selecione a caixa **Module(s)** e informe o caminho para o diretório no **Modules field**. Além disso, pode-se pressionar o botão *Select* para escolher em uma lista de todos os módulos no repositório.

8.5. Atualizações e Colaboração

Os participantes do time pode ver o repositório selecionando **Window→Versioning→Versioning.** Isto irá abrir uma janela no canto superior esquerdo que mostrará o conteúdo do repositório. No canto superior direito, devem ser adicionadas as contas para os seus companheiros de time para que eles tenham acesso ao repositório. Quaisquer atualizações ou modificações feitas pelos outros participantes do time podem ser vistas através do log de histórico como mostrado na Figura 42. Pressione com o botão direito no repositório e este mostrará o que for necessário. Esta janela também mostrará ferramentas que o ajudarão a testar os módulos modificados pelos outros participantes do time. Pressionando em **Tools** após clicar com o botão direito em repositório será mostrada a opção de criar testes. (Veja teste *Junit*)

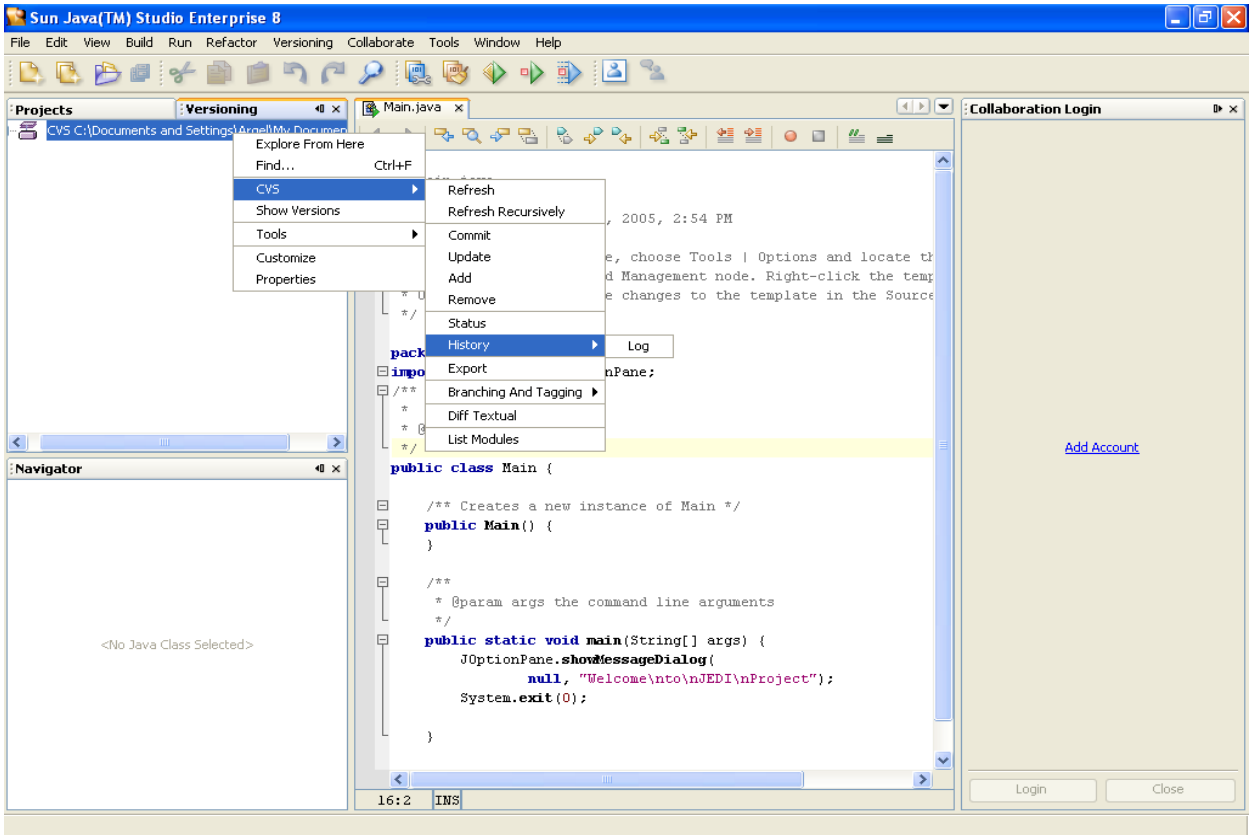

Figura 42 Janela de Versionamento

Usando o Sun Java™ Studio Enterprise 8 com certeza vai tornar a colaboração entre os times mais fácil nos projetos de software. Nenhuma ferramenta de software extra é necessária: tudo o que é necessário é de um repositório de trabalho, participantes de time que irão ajudá-lo em seu projeto e o Sun Microsystems Java™ Studio 8!

9. Mapeando Produtos de Implementação com a Matriz de Rastreabilidade de Requisitos

Normalmente, não são acrescentados elementos adicionais à MRR durante a fase de implementação. Ao contrário, monitoramos o desenvolvimento de cada componente de software definido na coluna Classes. Entretanto, durante o desenvolvimento, classes adicionais podem ter sido desenvolvidas. Elas devem ser incluídas nesta coluna, debaixo do pacote apropriado.

Podemos usar a MRR para determinar quantas destas classes foram implementadas, integradas e testadas.

10. Métricas de Implementação

Já que as características de uma classe devem corresponder às características do projeto, as métricas utilizadas para a implementação são as métricas para o projeto. Outras métricas consideradas na fase de implementação são utilizadas para projetos posteriores similares. Eles servem como dados históricos que podem ser utilizados para estimativas de outros projetos.

1. *Linhas de Códigos (LDC)*. Este é o número de linhas de código que são utilizadas. Pode ser para uma classe, componente ou o software inteiro. É utilizado posteriormente no gerenciamento de projetos similares.
2. *Número de Classes*. Este é o número total de classes criadas.
3. *Número de Páginas de Documentação.* O número total de documentação produzida.
4. *Custo*. O custo total do desenvolvimento do software.
5. *Esforço*. Este é o número total de dias ou meses em que o projeto foi desenvolvido.

11. Exercícios

11.1. Definindo o Formato da Documentação Interna

1. Defina o ***Bloco de Comentário Cabeçalho*** que será utilizado para a Documentação Interna da classe. Como um guia, a seguir temos os componentes recomendados:
    - Nome do Componente
    - Autor do Componente
    - Data da última criação ou modificação do Componente
    - Lugar onde o componente se encaixa no sistema em geral
    - Detalhes da estrutura de dados, algoritmo e fluxo de controle do componente

    Às vezes, ajuda se colocarmos informações históricas sobre a classe. Os componentes recomendados são:
    - Quem modificou o componente?
    - Quando o componente foi modificado?
    - Qual foi a modificação?

    O formato deve incluir como ele será escrito tal como o espaçamento, rótulos serão em letra maiúscula, etc. Mostre um exemplo de código fonte.
2. Usando o Bloco de Comentário Cabeçalho definido no exercício 1 para a Documentação Interna, implemente o **AthleteRecordUI**, **FindAthleteRecordUI**, **AthleteListUI** e **FindAthleteRecord** projetados no capítulo anterior. Se o Banco de Dados não pode ser implementado neste ponto, todos os métodos do controlador que utilizarem objetos da camada de persistência e banco de dados deverão ser codificados para mostrar uma mensagem descrevendo o que o método deveria fazer. Como um exemplo, para o método **public getAthleteRecord(String searchCriteria)** o programador pode simplesmente mostrar uma caixa de diálogo mostrando ao usuário que este método retornaria uma lista de registros de atletas conforme especificado pelo valor no **searchCriteria**.

11.2. Desenvolvimento de um Projeto

O objetivo do desenvolvimento de um projeto é reforçar o conhecimento e as habilidades obtidas neste capítulo. Particularmente, são eles:

1. Definição do bloco de comentário cabeçalho
2. Acompanhamento da implementação usando a Matriz de Rastreabilidade de Requisitos

PRODUTOS MAIORES DE TRABALHO:

1. Software Implementado
2. Arquivos de Instalação e Configuração

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.