Módulo 4

Engenharia de Software

Lição 4

Engenharia de Projetos

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

Exigências distintas de engenharia que criam modelos com foco na descrição de dados, função e comportamento, engenharia de projeto focada na criação de representações ou modelos que estão concentrados na arquitetura do software, estrutura de dados, interfaces e componentes que são necessários para implementação do software. Neste capitulo iremos aprender os conceitos e princípios dos projetos, projetos de dados, projetos de interface, projetos de nível de componentes. Iremos aprender quais elementos do RTM são modificados e adicionados para localizar produtos de trabalho através de requerimentos. A métrica de projetos também será discutida.

Ao final desta lição, o estudante será capaz de:

- Aprender os conceitos e dinâmicas da engenharia de projeto
- Aprender a como desenhar a arquitetura de um software
- Aprender como desenvolver o modelo de dados
- Aprender a como desenhar interface, particularmente, as telas e diálogos
- Aprender a como projetar os componentes de um software
- Aprender a como utilizar os requisitos de rastreamento
- Aprender sobre as Métricas de Projeto

2. Conceito de Engenharia de projetos

***Engenharia* de projeto** é fase que mais desafia a experiência de engenheiros de software no desenvolvimento de projetos. É necessário criatividade para formulação de produtos ou sistemas onde os requisitos (obtidos do usuário final pelo desenvolvedor) e o aspecto técnico são unidos. É a última fase da engenharia de *software* que cria modelos, configura o estágio final da construção e testes.

Nela são usadas várias técnicas e princípios com a finalidade de definir um dispositivo, processo ou sistema, em detalhe suficiente permitir sua construção física.

Produz um modelo de projeto que traduz o modelo de análise dentro de um impressão suficiente para construir e testar um software.

O projeto de processos é um trabalho interativo de refinamento, tendendo de um alto nível de abstração para um baixo nível de abstração. Cada representação deve ser localizada antes de uma exigência específica como documentado no RTM para assegurar qualidade do projeto. Especificamente, o projeto deve:

- implementar explicitamente todas as especificações como inserido no modelo de análise e, ao mesmo tempo, todas as exigências implícitas desejadas pelo usuário final ao desenvolvedor;
- seja legível e compreensível por quem gerar o código;
- proveja acima de tudo uma ilustração do software sobre a perspectiva dos dados, função e comportamento de uma perspectiva da implementação.

Pressman sugestiona uma lista de diretrizes de qualidade como as enumeradas abaixo[1].

1. O projeto deverá ter uma arquitetura reconhecível que seja criada usando estilos arquitetônicos conhecidos ou padrões que consistem em bons componentes de projeto e isso pode ser criado de uma maneira evolutiva.
2. O projeto deve ser modular e logicamente particionado em subsistemas e elementos.
3. O projeto deve consistir em uma única representação de dados, arquitetura, interfaces e componentes.
4. O projeto da estrutura de dados deverá conduzir ao projeto das classes apropriadas que são derivadas de padrões de projetos (*design patterns*) conhecidos.
5. O projeto de componentes deverá ter características de funcionalidades diferentes.
6. O projeto deverá ter interfaces que reduzam a complexidade das ligações entre componentes e o ambiente.
7. O projeto é derivado de usos de métodos repetitivos para obter informações durante a fase de engenharia de requisitos.
8. O projeto deverá usar uma notação que indique seu significado.

As diretrizes anteriores levam a um bom projeto devido a aplicação de princípios fundamentais de projetos, metodologia sistemática e revisão completa.

2.1. Conceitos de projeto

Os conceitos de projeto fornecem ao engenheiro de software uma base na qual podem ser aplicados métodos de projeto. Provê um ambiente de trabalho (*framework*) necessário para criação de certos produtos do trabalho.

2.1.1. Abstração

Quando projetamos um sistema modular, muitos níveis de abstração são usados. Como engenheiros de software, definimos níveis diferentes de abstrações quando projetamos este documento sobre software. No nível mais alto de abstração, declaramos uma solução que usa termos abrangentes. Quando interagimos em um nível mais baixo de abstração, é definida uma

1 Pressman, Software que Cria a Aproximação de Um Médico, page262-263,

descrição detalhada. São criados dois tipos de abstrações, isto é, de dados e processuais.

Abstração de dados se refere a coleções de dados que descrevem a informação requerida pelo sistema.

Abstração processual se refere a seqüências de comandos ou instruções que estão limitadas a ações específicas.

2.1.2. Modularidade

Modularidade é a característica do software que permite o gerenciamento do desenvolvimento e manutenção. O software é decomposto em pedaços chamados de **módulos**. Possuem nomes e componentes endereçáveis que, quando unidos, trabalham juntos satisfazendo requisitos. Projetos são modularizados de forma que podemos facilmente desenvolver um plano de incremento do software, acomodando facilmente as alterações testes e depuração efetiva e manutenção do sistema com pequenos efeitos colaterais. Em aproximação a orientação a objetos eles são chamados de classes.

Modularidade conduz a **ocultar informação**. Informações que escondem meios, que escondem os detalhes do módulo (atributos e operações) ou classificam de tudo outros que não têm nenhuma necessidade por tal informação. Módulos ou classes se comunicam com interfaces, mas , obrigam o acesso aos dados através de processos. Isto limita ou controla a propagação das mudanças, ocorrência de erros quando terminamos as modificações nos módulos ou classes.

Modularidade sempre leva a independência funcional. **Independência funcional** é a característica dos módulos ou classes que endereçam a funções específicas definidas pelos requisitos. É alcançada definindo-se módulos que fazem uma única tarefa ou função e têm justamente uma interação com outros módulos. Bons projetos usam dois critérios importantes: coesão e acoplamento.

Acoplamento é o grau de interconectividade entre os objetos representado pelo número de ligações que um objeto tem e pelo grau de interação com os outros objetos. Para a orientação a objetos, dois tipos de acoplamento são utilizados.

1. **Acoplamento por Interação** é a medida do número de tipos de mensagem que um objeto envia a outro objeto, e o número de parâmetros passado com estes tipos de mensagem. Um bom acoplamento por interação é conseguido com um mínimo possível para evitar mudança através da interface.
2. **Acoplamento por Herança** é o grau que uma subclasse de fato necessita das características (atributos e operações) herdadas da sua classe base. Um número mínimo de atributos e operações que são herdados desnecessariamente.

Coesão é a medida com que um elemento (atributo, operação ou classe dentro de um pacote) contribui para um único propósito. Para projetos orientados a objetos, são utilizados três tipos de coesão.

1. **Coesão de operações** é o grau que uma operação enfoca um único requisito funcional. O bom projetista produz operações altamente coesas.
2. **Coesão de classes** é o grau que uma classe enfoca um único requisito.
3. **Coesão especializada** endereça a semântica coesão de herança. A definição de herança deve refletir verdadeiramente o melhor compartilhamento sintático da estrutura.

2.1.3. Refinamento

Refinamento é também conhecido como o **processo de elaboração**. O refinamento da completa abstração permite à engenharia de software especificar o comportamento e os dados de uma classe ou módulo, contudo, suprime o detalhamento de níveis baixos. Isto ajuda a engenharia de software a criar um completo modelo evolutivo do projeto. O refinamento auxilia a engenharia de software a descobrir detalhes no avanço do desenvolvimento.

2.1.4. Refatoramento

Refatoramento é a técnica que simplifica o projeto dos componente sem várias das funcionalidades e comportamentos. Este é um processo de variação de software de modo que o comportamento externo fica o mesmo e a estrutura interna é melhorada. Durante a refatoração, o modelo de projeto é checado em busca de redundâncias, elementos de projeto não utilizados, ineficiência ou algoritmos desnecessários, mal construção ou estruturas de dados inadequadas ou um outro projeto falho. Estes são corrigidas para produzir um melhor projeto.

2.2. Modelo de Projeto

O trabalho produzido na fase de Engenharia de Projeto é o Modelo de Projeto que consiste de projeto de arquitetura, projeto de dados, projeto de interface e projeto a nível de componentes.

2.2.1. Projeto de arquitetura

Refere-se à total estrutura do software. Isto inclui o caminho no qual fornece a integridade conceitual para o sistema. Representa camadas, subtipos e componentes. Este é o modelo utilizado no diagrama de pacotes da UML.

2.2.2. Projeto de dados

Refere-se ao projeto e à organização dos dados. As classes de entidades são definidas na fase de engenharia de requisitos e são refinadas para criar o projeto da lógica do banco de dados. Classes persistentes são desenvolvidas para acessar dados do servidor de banco de dados. Este é modelado é usado no diagrama de classes.

2.2.3. Projeto de interface

Refere-se ao projeto da interação do sistema com o seu ambiente, particularmente, os aspectos da interação humana. Isto inclui os diálogos e projeto de telas. Relatórios e formas de *layouts* são incluídos. Este utiliza o diagrama de classes e o diagrama de transição de estados.

2.2.4. Projeto a nível de componentes

Refere-se ao projeto do comportamento interno de cada classe. De particular interesse são as classes de controle. Este é muito importante para o projeto pois as funcionalidades requeridas são representadas pelas classes. Este utiliza o diagrama de classes e o diagrama de componentes.

2.2.5. Projeto a nível de implementação

Refere-se ao projeto de como o software poderá ser implementado para uso operacional.

Software executável, subsistemas e componente são distribuídos para o ambiente físico que suporte o software. O diagrama implementação poderá ser utilizado para representar este modelo.

3. Arquitetura de Software

Não existe uma definição geral aceita para o termo Arquitetura de Software. É interpretado diferentemente em relação ao contexto. Alguns podem descrever em termos de estruturas de classes pelo modo que são agrupados. Outros costumam descrever toda organização de um sistema em um subsistema. *Buschmann* define da seguinte forma:

"A **arquitetura de software** descreve um subsistema e os componentes de um software e as relações entre eles. Subsistemas e componentes são tipicamente especificados de diferentes visões para mostrar as propriedades funcionais e não funcionais de um sistema. A arquitetura de software de um sistema é um artefato. Sendo o resultado da atividade de projeto do software".

Esta definição é discutida neste capítulo. A arquitetura de software é uma camada estruturada dos componentes de software e pela maneira ao qual estes componentes interagem, bem como a estrutura de dados utilizada por estes componentes. Isso envolve tomar decisões em como a aplicação é construída e, normalmente, controla o desenvolvimento iterativo e incremental.

Definir a arquitetura é importante por diversas razões. Primeiro, a representação da arquitetura de software permite a comunicação entre os envolvidos (usuários finais e desenvolvedores). Segundo, permite projetar as decisões que terão um efeito significativo em todo produto do trabalho da engenharia de software referente à interoperabilidade da aplicação. Por último, fornecem uma visão intelectual de como o sistema é estruturado e como os componentes trabalham em conjunto.

É modelado usando um Diagrama de Pacote da UML. Um pacote é um elemento de modelo que pode conter outros elementos. Também é um mecanismo utilizado para organizar os elementos do projeto e permitir que os componentes de software possam ser modularizados.

3.1. Descrevendo o Diagrama de Pacote

O **Diagrama de Pacote** mostra de forma particionada um grande sistema em agrupamentos lógicos de pequenos subsistemas. Apresenta um agrupamento de classes e dependências entre elas. Uma dependência existe entre dois elementos caso as mudanças para definição de um elemento possa causar alteração em outros elementos. A Figura 1 mostra a notação básica de um diagrama de pacote.

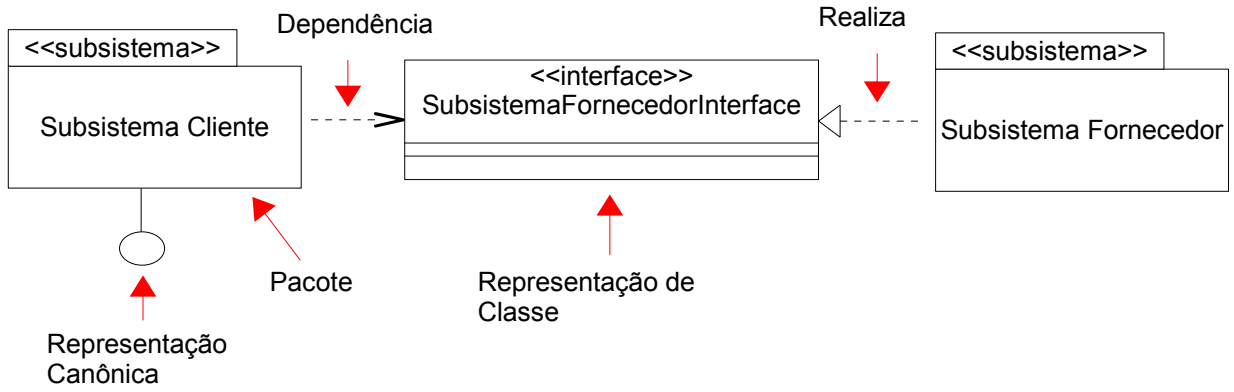

Figura 1: Notação Básica para o Diagrama de Pacote

Pacotes são representados como **pastas**. Se um pacote é representado como um subsistema, o estereótipo de um subsistema é indicado por (<<subsistema>>). As **dependências** são representadas por setas pontilhadas sendo o indicador da seta uma linha aberta. Como no exemplo, o Subsistema Cliente é dependente do Subsistema Fornecedor. Os pacotes são realizados por **interfaces**. A interface pode ser canônica, como representado por um círculo descrito pela Interface do Subsistema Cliente, ou pode ser uma definição de classe como descrito pela Interface do Subsistema Fornecedor (estereotipo é <<interface>>). Observa-se que o relacionamento de um pacote é ilustrado com um linha pontilhada sendo o indicador da seta um triângulo sem preenchimento interno.

3.2. Subsistemas e Interfaces

Um **subsistema** é a combinação de um pacote (também pode conter outros elementos de modelo) e uma classe (que possui comportamento e interage com outros elementos do modelo). Pode ser utilizado para particionar o sistema em partes que possibilite ser independentemente ordenado, configurado e entregue. Permite que os componentes sejam desenvolvidos de modo independente enquanto as interfaces não se alterem. Pode ser implantado através de um conjunto computacional distribuído (cluster/grid) e alterados sem quebrar outras partes do sistema, além de prover um controle restrito da segurança por meio de recursos chaves.

Tipicamente agrupa elementos do sistema para compartilhar propriedades comuns. Encapsula o conjunto de responsabilidades de forma inteligível para poder assegurar sua integridade e sua capacidade de manutenção. Como exemplo, um subsistema pode conter interfaces de interação homem-máquina que lidam como as pessoas interagem com a aplicação. Outro subsistema poderia lidar com o gerenciamento dos dados.

As vantagem de se definir um subsistema são as seguintes:

- Permitir o desenvolvimento em unidades menores..
- Aumentar a reusabilidade de componentes.
- Permitir que os desenvolvedores lidem com as partes complexas do sistema.
- Facilitar o suporte as manutenções.
- Prover a portabilidade.

Cada subsistema deve ter um limite claro e interfaces bem definidas com outros subsistemas.

Suporta portabilidade: Cada subsistema deve ter fronteiras claras e interfaces totalmente definidas com outros subsistemas.

Interfaces definem um conjunto de operações que são executadas por um subsistema. Permite a separação da declaração de comportamento da execução do comportamento. Servem como um contrato para ajudar na independência do desenvolvimento dos componentes por um time de desenvolvedores e garante que os componentes trabalhem em conjunto. A especificação da interface define a natureza precisa da interação dos subsistemas com o resto do sistema, mas não descreve a sua estrutura interna.

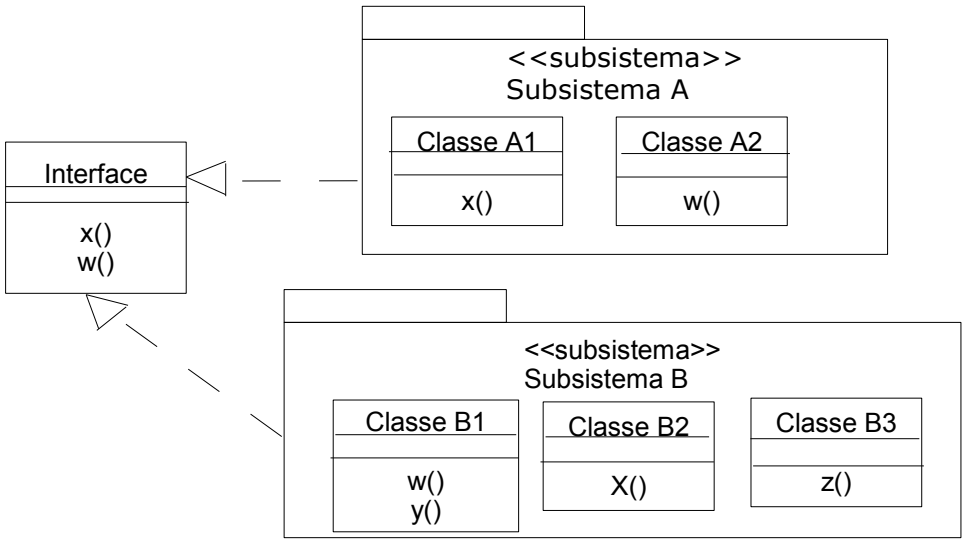

Figura 2: Subsistemas e Interfaces

Conforme mostrado na Figura 2, uma interface é composta por um ou mais subsistemas. Este é o segredo de ter componentes *plug-and-play*. A implementação do subsistema pode mudar sem mudar drasticamente outros subsistemas, desde que a definição da interface não mude.

Cada subsistema provê serviços para outros subsistemas. Há dois estilos de uso de subsistemas de comunicação; eles são conhecidos como comunicação **cliente-servidor** e comunicação **ponto-**

a-ponto. Eles são mostrados na Figura 3.

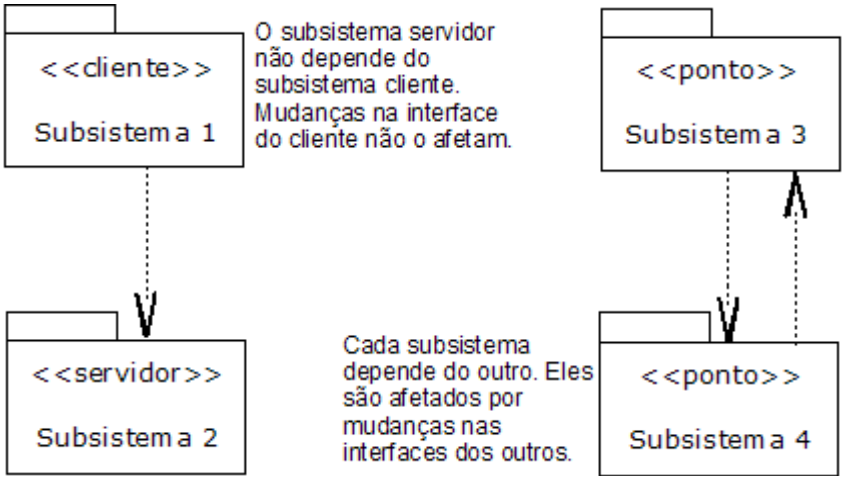

Figura 3: Estilos de Comunicação de Subsistemas

Numa comunicação cliente-servidor, o subsistema cliente precisa conhecer a interface do subsistema servidor. A comunicação é somente num sentido. O subsistema cliente requisita serviços do subsistema servidor; nunca ao contrário. Na comunicação ponto-a-ponto, cada subsistema conhece a interface do outro subsistema. Nesse caso, o acoplamento é mais forte. A comunicação acontece nos dois sentidos, uma vez que qualquer dos subsistemas pode requerer serviços dos outros.

Em geral, a comunicação cliente-servidor é mais simples de implementar e manter, uma vez que o acoplamento não é tão forte como na comunicação ponto-a-ponto.

Há duas abordagens na divisão de programas em subsistemas. Elas são conhecidas como de **camadas** e de **particionamento**. A de **camadas** enfoca os subsistemas como se representados por diversos níveis de abstração ou camadas de serviço, enquanto que a de **particionamento** enfoca os diferentes aspectos da funcionalidade do sistema como um todo. Na prática, ambas as abordagens são usadas de forma que alguns subsistemas são definidos em camadas e outros em partições.

As camadas de arquiteturas são comumente utilizadas para estruturas de alto nível num sistema. A estrutura geral é detalhada na Figura 4.

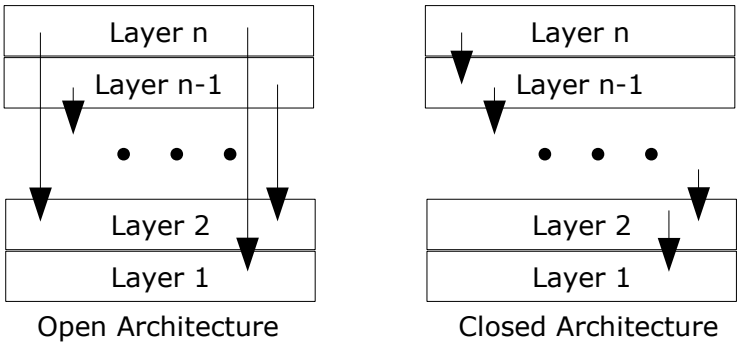

Figura 4: Estruturas Gerais da Camada de Arquitetura

Cada camada da arquitetura representa um ou mais subsistemas que podem diferenciar-se entre si pelos diferentes níveis de abstração ou por abordagem diferente de sua funcionalidade. A camada superior solicita serviços da camada inferior a ela. Essas camadas, por sua vez, podem utilizar serviços da próxima camada. Em uma **Arquitetura em Camadas Abertas**, subsistemas podem requisitar serviços de camadas abaixo. Para **Arquitetura em Camadas Fechadas**, camadas requisitam serviços de camadas diretamente abaixo delas e não podem pular camadas.

Arquitetura em Camadas Fechadas minimiza dependências entre camadas e reduz o impacto da mudança de interface de qualquer camada. **Arquitetura em Camadas Abertas** possibilita um desenvolvimento de códigos mais compactos pois serviços de todas as camadas de baixo nível são

acessados diretamente por qualquer camada acima sem necessidade de programação de código extra para passagem de mensagens através de camadas intervenientes. Por outro lado, há uma quebra no encapsulamento das camadas, aumentando as dependências entre as camadas e a dificuldade de alteração quando uma camada precisar ser modificada.

Algumas camadas, dentro de uma arquitetura de camadas, podem ser decompostas por causa de sua complexidade. O particionamento é obrigatório na definição dos componentes decompostos. Como exemplo, considere a Figura 5.

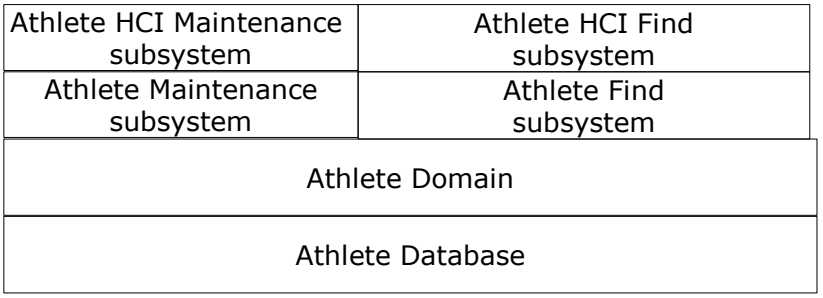

Figura 5: Exemplo de Camada e Particionamento

Utiliza uma arquitetura de quatro camadas. Consiste no seguinte:

1. *Camada de Banco de Dados*. Responsável pelo armazenamento e recuperação de informações do repositório. No exemplo, banco de dados Atleta.
2. *Camada de Domínio*. Responsável pelos serviços ou objetos que são compartilhados pelas diferentes aplicações. Especificamente utilizada em sistemas distribuídos. Neste exemplo, é o Domínio Atleta.
3. *Camada de Aplicação*. Responsável em executar aplicações que representam a lógica do negócio. No exemplo, representada pelos subsistemas de manutenção e pesquisa de Atleta.
4. *Camada de Apresentação*. Responsável pela apresentação dos dados ao usuário, dispositivos ou outros sistemas. No exemplo, representada pelos subsistemas HCI de manutenção e pesquisa de Atleta.

3.2.1. Um Exemplo de Arquitetura em Camada

Arquiteturas em camadas são utilizadas amplamente na prática. *Java 2 Enterprise Edition* (J2EE™) adota uma abordagem multi-camada e um catálogo de padrões associado tem sido desenvolvido. Esta seção mostra uma introdução da plataforma J2EE.

A plataforma J2EE representa um padrão para implementação e distribuição de aplicações coorporativas. É projetada para fornecer suporte no lado cliente e no lado servidor no desenvolvimento de aplicações distribuídas e multi-camadas. Aplicações são configuradas como segue:

1. ***Client-tier*** provê a interface do usuário que suporta um ou mais tipos de cliente, tanto fora como dentro de um *firewall* corporativo.
2. ***Middle-tier*** módulos que fornecem serviços de cliente através de *containers WEB* na camada *WEB* e serviços de componentes de lógica de negócio através de *Enterprise JavaBean* (EJB) *Container* na camada EJB.
3. ***Back-end*** provê os sistemas de informações coorporativas para o gerenciamento de dados que é suportado por APIs padrões.

A Figura 6 descreve os vários componentes e serviços que formam um típico ambiente J2EE.

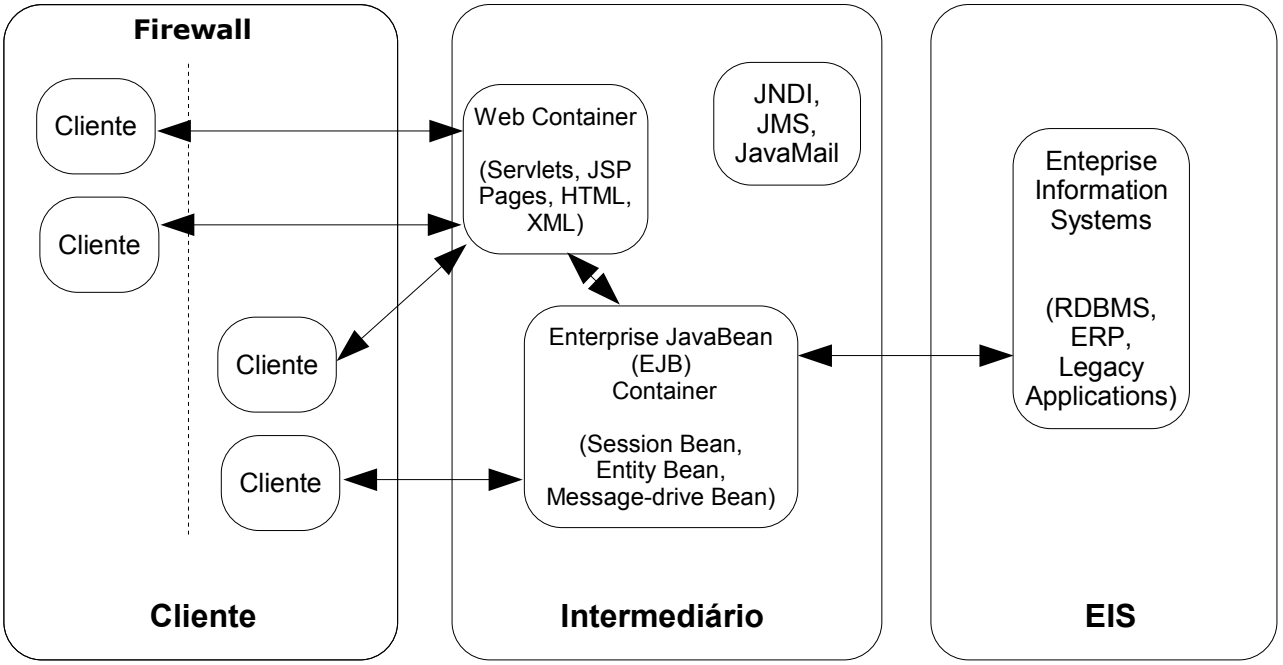

Figura 6: Arquitetura Java 2 Enterprise Edition (J2EE)

Para maiores detalhes referentes à plataforma J2EE, acesse o site, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e para aprender como projetar aplicações corporativas.

3.3. Desenvolvendo o Projeto de arquitetura

Na construção da arquitetura do software, os elementos de projeto são agrupados em pacotes e subsistemas. Interfaces são definidas. Os relacionamentos dentro desses são ilustrados pela definição de dependências. Como um exemplo, O modelo de análise do **Sistema de Manutenção de Sócios de Clube** será usado como foi desenvolvido no capítulo anterior.

PASSO 1: Se o modelo de análise ainda não foi validado, validar o modelo de análise.

devemos assegurar que os atributos e responsabilidades são distribuídos para as classes, e assegurar que uma classe define uma única abstração lógica e local. Se existem problemas com o modelo de análise, regresse à fase de requesitos da engenharia. Pode ser usado o *checklist* de validação do modelo de análise.

PASSO 2: Análise relacionada do pacote de agrupamento das classes.

Como foi mencionado anteriormente, pacotes são mecanismos que nos permitem agrupar modelos de elementos. Decisões de empacotamento são baseadas no critério de empacotamento com um número diferente de fatores que incluem configurações unitárias, alocação de recursos, agrupamento de tipos de usuários, e representação de um produto e serviço existente utilizado pelo sistema. A seguir, veremos um guia de como agrupar classes.

3.3.1. Guia de Empacotamento de Classes

1. Empacotamento de classes limite
   - Se os sistemas de interface são parecidos para serem alterados ou substituídos, as classes limite deveriam ser separadas do restante do projeto.
   - Se nenhuma mudança da interface principal for planejada, as classes limite deveriam ser colocadas juntamente com a entidade e classes de controle com as quais estão funcionalmente relacionadas.
   - Classes de limites mandatórias que não são funcionalmente relacionadas com qualquer entidade ou classes de controle, estão agrupadas separadamente.

- Se a classe limite é relacionada em um serviço opcional, agrupe-os em um subsistema separado com as classes que colaboram para prover o serviço mencionado. Desta forma, o subsistema é mapeado para um componente opcional da interface, podendo prover o serviço usando a classe limite.

2. Funcionalidade de Empacotamento das Classes Relacionadas

- Se uma mudança em uma única classe afeta uma mudança em outra classe, elas são funcionalmente relacionadas.
- Se a classe é removida do pacote e tem grande impacto para o grupo de classes, a classe é funcionalmente relacionada às classes impactadas.
- Duas classes são funcionalmente relacionadas se elas possuem um grande número de mensagens que podem enviar para uma outra.
- Duas classes são funcionalmente relacionadas se elas interagem com o mesmo ator, ou são afetadas pelas mudanças realizadas pelo mesmo ator.
- Duas classes são funcionalmente relacionadas se possuem relacionamento entre si.
- Uma classe é funcionalmente relacionada a outra classe que cria uma instância sua.
- Duas classes que são relacionadas a atores diferentes não deveriam estar no mesmo pacote.
- Classes mandatórias e opcionais não deveriam estar na mesma classe.

Após tomar a decisão de como agrupar as classes de análise, as dependências do pacote são definidas. ***Dependências de Pacote*** são também conhecidas como ***visibilidade***. Estas definem o que é e não é acessível dentro do pacote. Tabela 1 demonstra os símbolos de visibilidade que são colocados junto ao nome da classe dentro do pacote.

Visibilidade dos Símbolos	***Descrição***
+	Representa um elemento do modelo público que pode ser acessado fora do pacote.
-	Representa um elemento do modelo privado que não pode ser acessado fora do pacote.
#	Representa elementos do modelo protegido que pode ser acessado através do proprietário do elemento do modelo ou através da herança.

Tabela 1: Visibilidade dos Símbolos do Pacote

Um exemplo de visibilidade do pacote é ilustrado na Figura 7. Classes dentro do pacote podem colaborar com uma outra. Dessa forma, elas são visíveis. Com exemplo, classes no Pacote **A**, tais como **ClassA1**, **ClassA2** e **ClassA3,** podem colaborar com uma outra desde que a mesma resida no mesmo pacote. Entretanto, quando classes residem em pacotes diferentes, elas são inacessíveis a menos que tenham sido definidas como públicas. **ClassB1** é visível porque é definida como pública. **ClassB2**, por outro lado, não é visível.

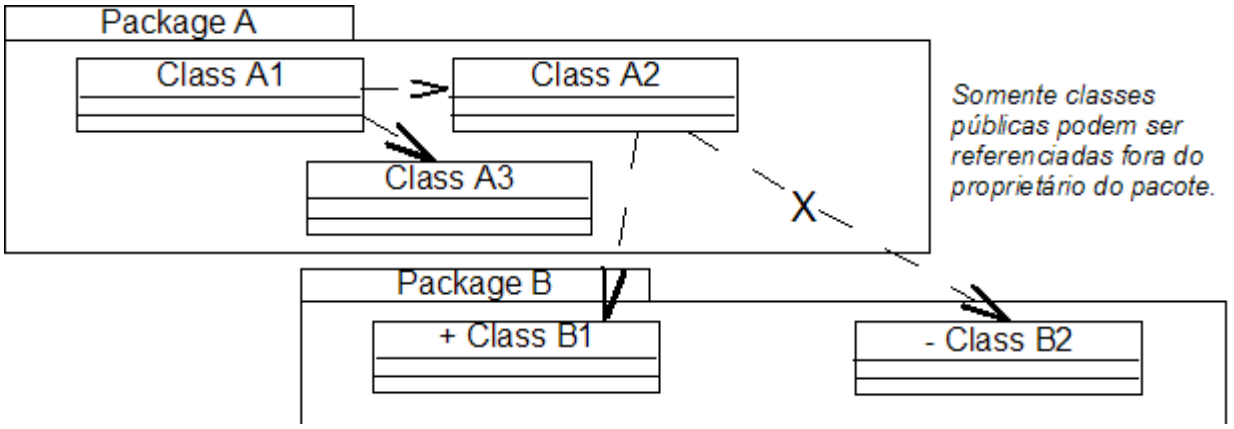

Figura 7: Visibilidade do Pacote

Acoplamento de Pacote define de que modo dependências são definidas entre pacotes. Devem aderia às mesmas regras.

1. Pacotes não podem ser acoplado em cruz
2. Pacotes em camadas inferiores não devem ser dependente de pacotes de uma camada superior. Camadas serão discutida em um dos passos.
3. Em geral, pacotes não devem pular camadas. Entretanto, exceções podem ser feitas e devem ser documentadas.
4. Pacotes não devem ser dependente de subsistemas. Eles devem ser dependentes de outros pacotes ou de interfaces de subsistema.

A decisão de empacotamento para o **Sistema de Manutenção de Sócios do Clube** é ilustrada na Figura 8.

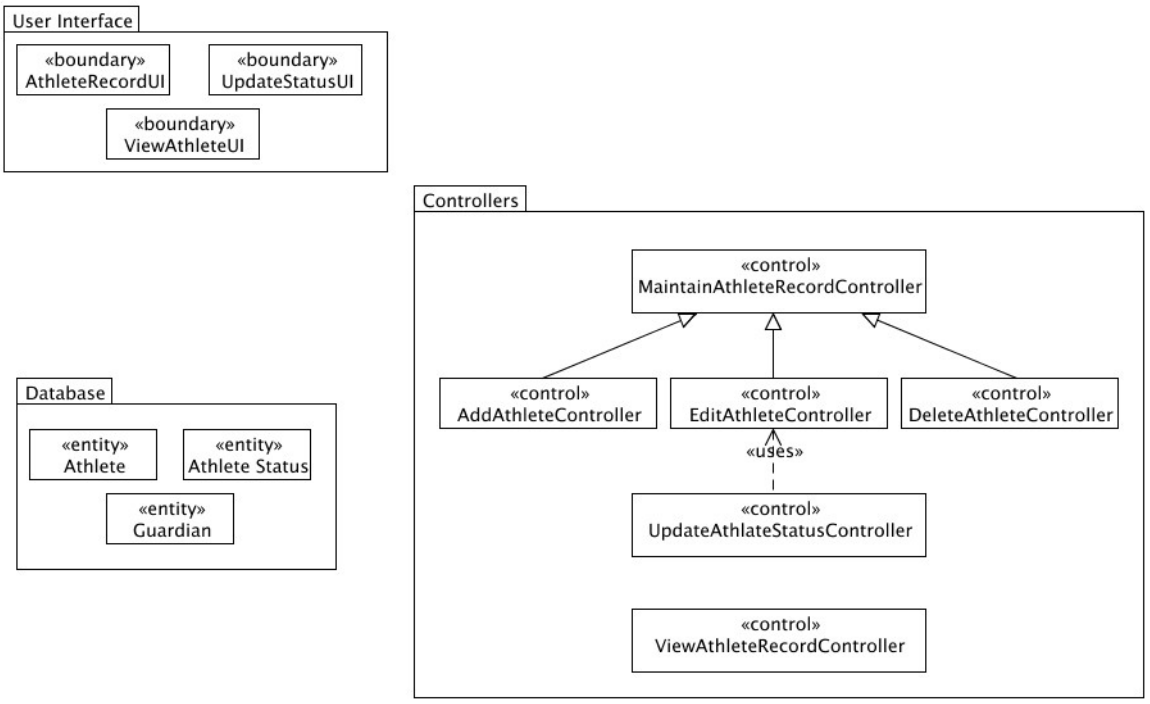

Figura 8: Decisão de empacotamento do Sistema de Manutenção de Sócios de Clube

PASSO 3: Identificando o projeto de Classes e Subsistemas

As classes da análise são exploradas para determinar se podem ser projetos de classes ou subsistemas. É possível que as classes da análise sejam expandidas, dobradas, combinadas e removidas do projeto. Nesse ponto, decisões tem que ser tomadas sobre as seguintes perguntas:

- Quais classes da análise são realmente classes? Quais que não são?
- Quais classes da análise são subsistemas?
- Quais componentes existentes? Quais componentes precisam de ser desenhados e implementados?

Uma classe da análise é uma classe de projeto se ela é uma classe simples ou se representa uma lógica de abstração única. Se uma classe da análise é complexa, pode ser refinada para tornar-se:

- uma parte de outra classe
- uma classe agregada
- um grupo de classes que herdam de uma mesma classe
- um pacote
- um subsistema
- uma associação ou relacionamento entre elementos de design

A seguir algumas orientações para auxiliar na identificação de subsistemas.

1. **Colaboração do Objeto**. Se os objetos de uma classe colaboram somente entre si e a colaboração produz um resultado visível, eles deverão fazer parte de um subsistema.
2. **Opcionalidade**. *As associações entre classes são opcionais ou características que podem ser removidas, melhoradas ou substituídas por outras alternativas, encapsule as classes dentro de um subsistema*.
3. **Interface do Usuário**. A divisão de classes *boundary* e classes *entity* relacionadas, em subsistemas separados é chamado ***subsistema horizontal*** enquanto que o agrupamento de todas em um subsistema é chamado ***subsistema vertical***. A decisão por subsistemas horizontais ou verticais depende do acoplamento entre interfaces do usuário e as entidades. Se classes *boundary* são utilizadas para exibir informações de classes entidade, a escolha é pelo subsistema vertical.
4. **Ator**. Se dois atores diferentes utilizam a mesma funcionalidade de uma classe, modele dois subsistemas diferentes para permitir que cada ator possa mudar seus requisitos independentemente.
5. **Acoplamento e Coesão de Classe**. Organize classes com alto grau de acoplamento dentro de subsistemas. Separe em linhas os acoplamentos fracos.
6. **Substituição**. Represente diferentes níveis de serviços de um recurso particular como subsistemas separados, todos devem implementar a mesma interface.
7. **Distribuição**. Se uma funcionalidade deve residir em um nó particular, assegure que aquela funcionalidade do subsistema esteja mapeada dentro de um nó individual.
8. **Volatilidade**. Encapsule dentro de um subsistema todas as classes que podem mudar num período de tempo.

Os possíveis subsistemas são:

1. Classes da Análise que fornecem serviços complexos ou utilitários e *boundary* classes.
2. Produtos existentes ou sistemas externos tais como software de comunicação, suporte ao acesso à base de dados, tipos e estruturas de dados, utilitários e aplicações específicas.

No caso do **Sistema de Manutenção de Associados do Clube**, os subsistemas inicialmente identificados são as classes *boundary AthleteRecordUII, ViewAthleteUI* e *UpdateStatusUI*.

PASSO 4: Definição da interface dos subsistemas.

Interfaces são grupos de operações (públicas) visíveis externamente. Em UML, elas não contém estrutura interna, atributos, associações ou detalhes da implementação das operações. Elas equivalem a uma classe abstrata que não tem atributos, associações e possuem somente operações abstratas. Elas permitem a separação da declaração do comportamento de sua implementação.

Intefaces definem um conjunto de operações que são realizadas por um subsistema. Servem como um contrato que ajuda no desenvolvimento independente de componentes pela equipe de desenvolvimento e assegura que os componentes poderão trabalhar em conjunto.

A Figura 9 mostra os subsistemas definidos para o **Sistema de Manutenção dos Sócios do Clube**. A interface está definida pela inserção de um estereótipo <<interface>> no compartimento do nome da classe. Para diferenciá-la de uma classe *boundary*, o nome da classe de interface começa com a letra 'I'. A Aplicação do Subsistema implementa a Interface *IAthletRecordUI*.

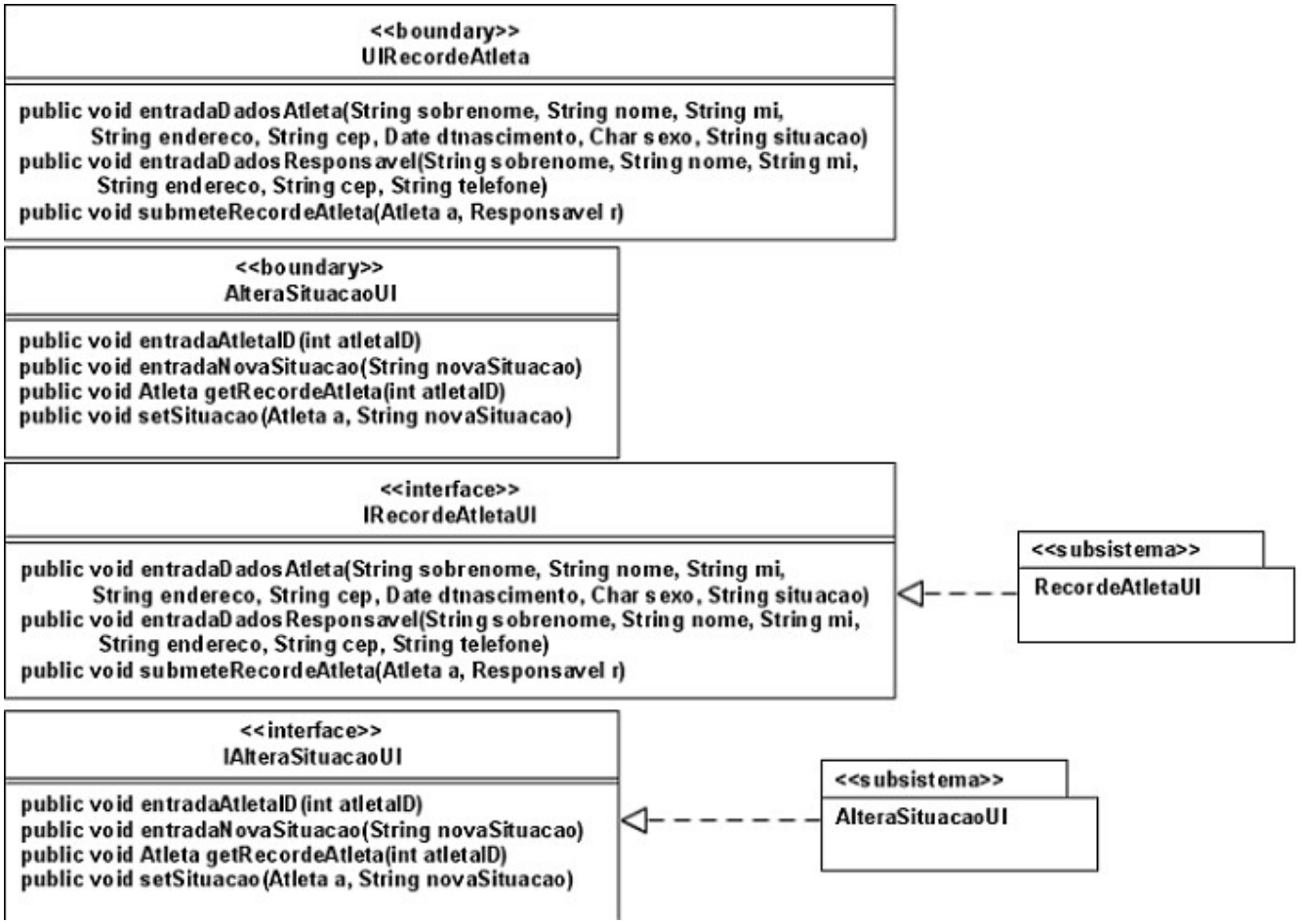

Figura 9: Identificação de Subsistemas no Sistema de Manutenção de Sócios do Clube

PASSO 5: Camada de subsistemas e classes.

A definição de camadas da aplicação é necessária para realizar de modo organizado o projeto e desenvolvimento do software. Elementos de Projeto (classes e subsistemas) precisam ser alocados em camadas específicas da arquitetura. Em geral, a maioria das classes de apresentação tendem a aparecer no nível mais alto, classes de controle no nível intermediário e classes entidade aparecem abaixo. Este estrutura é conhecida como ***arquitetura em três camadas*** e é mostrada na Figura 10.

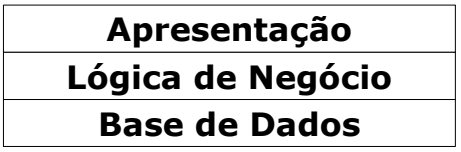

Figura 10: Arquitetura em Três Camadas

Há muitos estilos e padrões de arquitetura que são mais complexos e detalhados do que a simples arquitetura de três camadas. Normalmente, são projetados baseando-se nas exigências tecnológicas, tais como, o sistema a que será distribuído/utilizado, se o sistema suporta a simultaneidade, se o sistema está considerando uma aplicação baseada na *WEB*. Para simplificar, este estudo de caso usa a arquitetura de três camadas. Quando as classes dentro das camadas são elaboradas, há chances de camadas e pacotes adicionais serem usados, definidos e refinados.

Os tópicos a seguir são um guia prático para definir as camadas de uma arquitetura:

1. *Considere a visibilidade*. As dependências entre classes e subsistema ocorrem somente dentro da camada atual e da camada diretamente abaixo dela. Passando disto, a dependência deve ser justificada e documentada;
2. *Considere a volatilidade*. Normalmente, as camadas superiores são afetadas por mudanças

nas exigências, enquanto as camadas mais baixas são afetadas pela mudança do ambiente e da tecnologia;

3. *Número de camadas*. Os sistemas pequenos devem ter 3 ou 4 camadas, enquanto sistemas maiores devem ter de 5 a 7 camadas.

Para o **Sistema de Manutenção dos Sócios do Clube**, as camadas são descritas na Figura 11. Observe que as classes de análise que estão identificadas como subsistemas são substituídos por uma definição de interface do subsistema. Para esclarecimento, a realização da interface não é demonstrada.

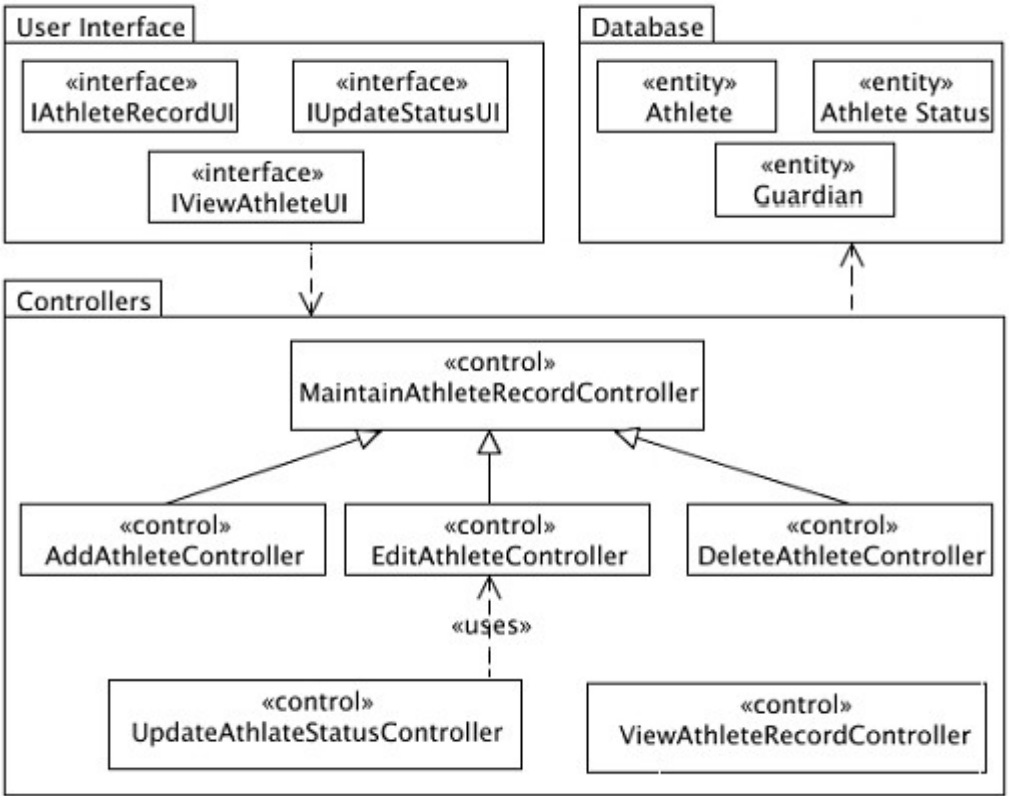

Figura 11: Definindo as Camadas do Sistema de Manutenção dos Sócios do Clube

3.4. Lista de Verificação e Validação da Arquitetura do Software

Como qualquer outro produto de trabalho, a Arquitetura do Software deve ser validada. Use as seguintes perguntas como um *checklist*.

Camadas:

1. Para sistemas menores, há mais de quatro camadas?
2. Para sistemas maiores, há mais de sete camadas?

Subsistemas e Interfaces:

1. A divisão dos subsistemas está feita de maneira lógica e consistente através da arquitetura do software?
2. O nome da interface descreve o papel do subsistema dentro do sistema principal?
3. A descrição da interface é concisa e clara?
4. Todas as operações que o subsistema necessita para executar estão identificadas? Há alguma operação ausente?

Pacotes

1. Os nomes dos pacotes estão descritivos?
2. A descrição dos pacotes combina com suas responsabilidades?

4. Padrões de Projeto

Padrões de projeto (***Design Patterns***) descrevem soluções comprovadas para problemas recorrentes. Isto alavanca o conhecimento e o entendimento de outros desenvolvedores. Eles são soluções reusáveis para problemas comuns. Endereçam problemas individuais mas podem ser combinados de diferentes formas para obter uma solução integrada para um sistema inteiro.

Padrões de projeto não são *frameworks*. ***Frameworks*** são sistemas de software parcialmente completos que podem ser utilizados em um tipo específico de aplicação. Um sistema aplicativo pode ser customizado para as necessidades da organização pela complementação dos elementos incompletos e adição de elementos específicos da aplicação. Isso envolveria a especialização de classes e a implementação de algumas operações. É, essencialmente, uma mini-arquitetura reusável que provê uma estrutura e um comportamento comum a todas as aplicações deste tipo.

Padrões de projeto, por outro lado, são mais abstratos e gerais do que os *frameworks.* São uma descrição do modo que um problema pode ser resolvido, mas não é, em si, a solução. Não pode ser diretamente implementado; uma implementação exitosa é um exemplo de um padrão de projeto. É mais primitivo do que um *framework*. Um *framework* pode usar vários padrões; um padrão não pode incorporar um *framework*.

Padrões podem ser documentados usando-se vários *templates* alternativos. O *template* padrão determina o estilo e estrutura da descrição do padrão, e este varia na ênfase que eles colocam sobre diferente aspectos do padrão. Em geral, uma descrição de padrão inclui os seguintes elementos:

1. ***Nome***. O nome de um padrão deve ser significativo e refletir o conhecimento incorporado pelo padrão. Pode ser uma simples palavra ou uma frase curta.
2. ***Contexto***. O contexto de um padrão representa as circunstâncias ou pré-condições sob as quais ele pode ocorrer. Deve ser bastante detalhado para permitir que a aplicabilidade do padrão seja determinada.
3. ***Problema***. Provê uma descrição do problema que o padrão endereça. Identifica e descreve os objetivos a ser alcançados dentro de um específico contexto e forças restritivas.
4. ***Solução***. É a descrição dos relacionamentos estático e dinâmico entre os componentes do padrão. A estrutura, os participantes e suas colaborações são todas descritas. Uma solução deve resolver todas as forças em um dado contexto. Uma solução que não resolve todas as forças, é falha.

O uso de padrões de projeto requer uma cuidadosa análise do problema que deve ser resolvido e o contexto no qual ele ocorre. Todos os participantes da equipe de desenvolvimento devem receber treinamento adequado. Quando um projetista está contemplando o uso de padrões de projeto, certos itens devem ser considerados. As seguintes questões provêem um guia para resolver estes itens.

1. Existe um padrão que endereça um problema similar?
2. O padrão dispara uma solução alternativa que pode ser mais aceitável?
3. Existe uma outra solução mais simples? Não devem os padrões ser usados apenas porque foram criados?
4. O contexto do padrão consistente com aquele do problema?
5. As conseqüências de uso do padrão são aceitáveis?
6. As restrições impostas pelo ambiente de software conflitariam com o uso do padrão? Como usamos o padrão?

Os seguintes passos fornecem um guia para o uso de um padrão de projeto selecionado.

1. Para obter uma visão geral completa, leia o padrão.
2. Estude a estrutura, os participantes e as colaborações do padrão em detalhe.
3. Examine as amostras de código para ver o padrão em uso.

4. Nomeie os participantes do padrão (isto é, Classes) que são mais significativas para a aplicação.
5. Defina as classes.
6. Escolha nomes específicos da aplicação para as operações.
7. Codifique ou implemente as operações que desempenham responsabilidades e colaborações dentro do padrão.

É importante notar neste momento que um padrão não é uma solução prescritiva para o problema. Ele deve ser visto como um guia sobre como encontrar uma solução adequada para um problema.

Como um exemplo, existe um conjunto de padrões de projeto comuns para a plataforma J2EE. Este seção discute de forma abreviada alguns deste padrões. Para encontrar mais sobre Padrões de Projeto para J2EE, é possível verificar neste endereço *WEB*:

http://java.sun.com/blueprints/patterns/index.html

4.1. Composite View

4.1.1. Contexto

O Padrão *Composite View* permite o desenvolvimento de uma visão mais gerenciável através da criação de um *template* para manusear os elementos de página comuns de uma visão. Para uma aplicação *web*, uma página contém uma combinação de conteúdos dinâmicos e elementos estáticos tais como cabeçalho, rodapé, logomarca, *background* e assim por diante. O *template* captura as características comuns.

4.1.2. Problema

Dificuldade de modificar e gerenciar o *layout* de múltiplas visões devido à duplicação de código. Páginas são construídas usando código de formatação diretamente sobre cada visão.

4.1.3. Solução

Use este padrão quando uma visão é composta de múltiplas sub-visões. Cada componente do *template* pode ser incluído dentro do todo e o *layout* da página pode ser gerenciado independentemente do conteúdo. Esta solução prove a criação de uma visão composta através da inclusão e substituição de componentes dinâmicos modulares.

Ele promove a reusabilidade de porções atômicas da visão pelo encorajamento de um projeto modular. É usado para gerar páginas contendo a visualização dos componentes que podem ser combinados numa variedade de modos. Este cenário ocorre para uma página que mostra diferentes informações ao mesmo tempo, tal como as encontradas em *home pages* de muitos endereços na *web,* onde é possível encontrar notícias, informações sobre o tempo, cotações de ações, etc.

Um benefício de usar este padrão é que o projetista da interface pode prototipar o *layout* da página, conectando conteúdo estático em cada componente durante o desenvolvimento do projeto.

Um benefício do uso deste padrão é que o projetista de interfaces pode desenvolver o protótipo do layout da página adicionando conteúdos estáticos em cada componente. Com o progresso do desenvolvimento do projeto, o conteúdo estático é substituído pelas sub-visões (*subview*) atômicas.

Este padrão tem um inconveniente. Há uma sobrecarga (*overhead*) associada a ela. É um meio termo entre flexibilidade e performance. Figura 12 mostra um exemplo de um *layout* de página. A página consiste das seguintes visões (*views*).

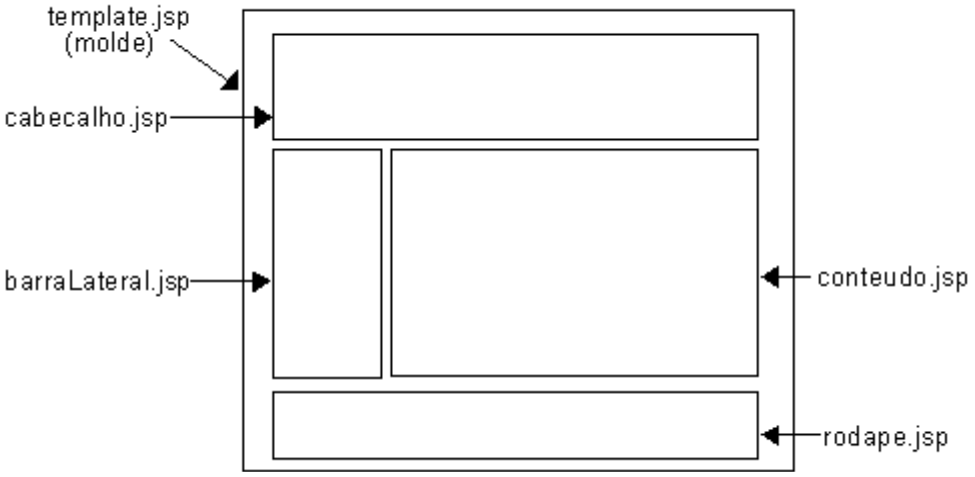

Figura 12: Exemplo do padrão Composite View

Este formulário pode ser uma tela web da aplicação e é definida no arquivo a seguir:

```
<screen name=  main  >
  <parameter key=  banner   value=  /cabecalho.jsp   />
  <parameter key=  footer   value=  /rodape.jsp   />
  <parameter key=  sidebar   value=  /barraLateral.jsp   />
  <parameter key=  body   value=  /conteudo.jsp   />
</screen>
```

A Figura 13 mostra o diagrama de classes que representa o padrão enquanto a Figura 14 mostra a colaboração das classes.

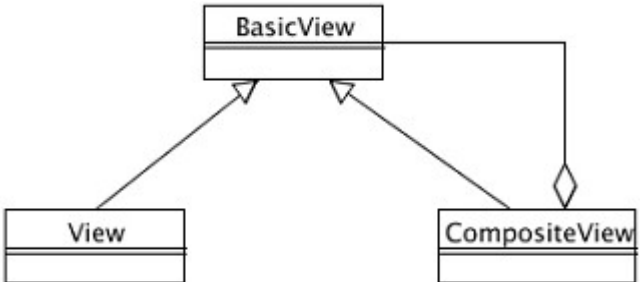

Figura 13: Diagrama de Classes do padrão Composite View

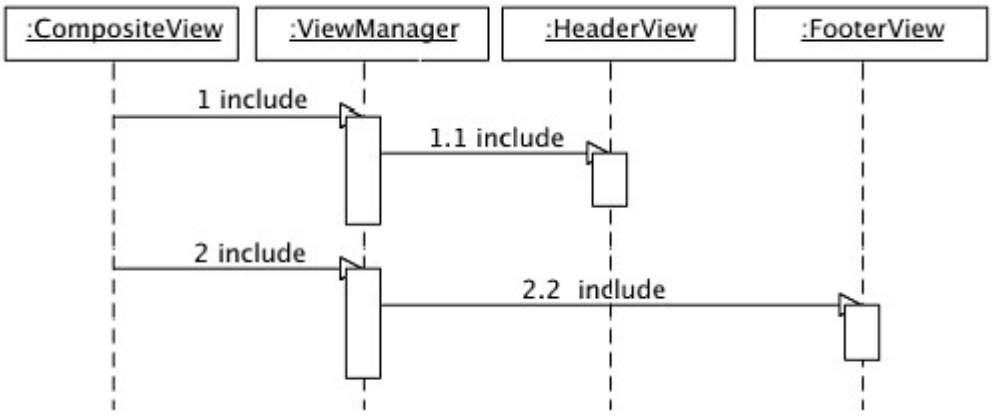

Figura 14: Diagrama de Seqüência do padrão Composite View

A Tabela 2 descreve as responsabilidades de cada uma das classes.

Classe	Descrição
CompositeView	É uma agregação de múltiplas visões (*views*).
ViewManager	Gerencia a inclusão de porções dos fragmentos do *template* dentro da composição da

Classe	Descrição
	visão.
IncludedView	É uma sub-visão, uma porção atômica de uma visão maior e mais completa. Esta visão (*includeView*) pode consistir em várias visões.

Tabela 2: Classes do Padrão Composite View

4.2. Front Controller

4.2.1. Contexto

O padrão *Front Controller* provê um controlador centralizado para administrar pedidos. Recebe todo o cliente que chega com um requisito, envia cada pedido para um manipulador de pedido apropriado e apresenta uma resposta apropriada para o cliente.

4.2.2. Problema

O sistema precisa de um ponto de acesso centralizado para manipulação do pedido na camada de apresentação e apoiar a integração de recuperação de dados do sistema, administração da visão e navegação. Quando alguém tiver o acesso à visão, diretamente, e sem passar por um manipulador de pedido centralizado, podem acontecer problemas como:

- Cada visão pode prover seu próprio serviço de sistema que pode resultar em duplicação de código.
- A navegação de visão é de responsabilidade da camada de visão e pode resultar em conflito com o conteúdo e a navegação de visão.

Adicionalmente, controle distribuído é mais difícil manter pois freqüentemente serão feitas mudanças em lugares diferentes no software.

4.2.3. Solução

Um controlador é utilizado como um ponto de contato inicial para controlar um pedido. É usado para administrar a manipulação de serviços como invocar serviços de segurança para autenticação e autorização, enquanto delega o processo empresarial, administrando a visão apropriada, controlando erros e administrando a seleção de estratégias de criação de conteúdo. Centraliza os controles de decisão e construção. São mostrados os Dagrama de Classe e Seqüência para este padrão na Figura 15 e Figura 16.

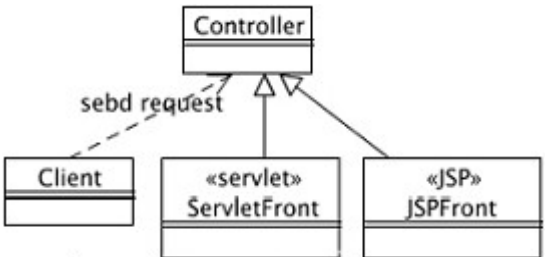

Figura 15: Diagrama de Classe do padrão Front Controller

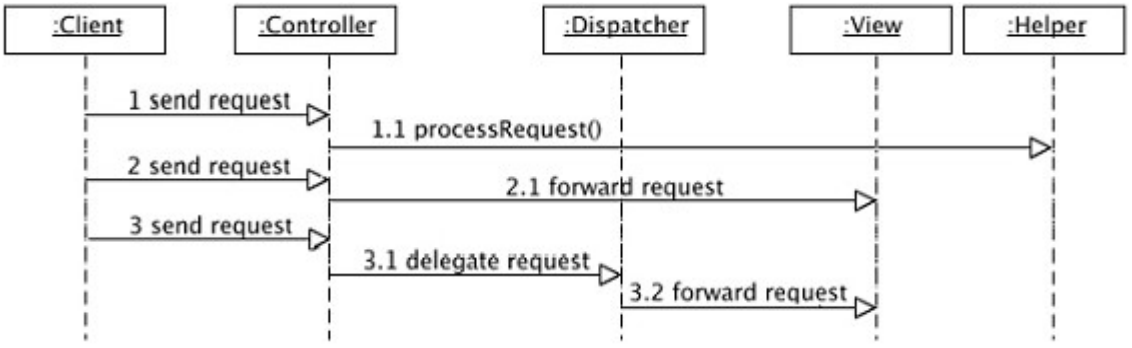

Figura 16: Diagrama de Seqüência do padrão Front Controller

As regras para as classes estão listadas na Tabela 3.

Class	Description
Controller	É o ponto de contato inicial para controlar todos os pedidos no sistema. Pode delegar um objeto de *Helper*.
Dispatcher	É responsável pela administração da visão e navegação, administra a escolha da próxima visão para apresentar ao usuário e o mecanismo para vetorização controlada como um recurso. Pode ser encapsulado dentro de um controlador ou um componente separado que trabalha em coordenação com o controlador. Usa o objeto *RequestDispatcher* e encapsula alguns métodos adicionais.
Helper	É responsável por auxiliar uma visão ou o controlador completa do processo. Possui numerosas responsabilidades que incluem junção de dados requeridos pela visão e armazenando de dados no modelo intermediário.
View	Representa e exibe informação ao cliente.

Tabela 3: Classes do padrão Front Controller

Como exemplo de implementação de um *Front Controller*, um *servlet* pode ser utilizado para representar o controlador. Um trecho do código é mostrado abaixo. No método ***processRequest*()**, um objeto do tipo ***RequestHelper*** (***Helper***) é utilizado para executar o comando que representa a requisição do cliente. Depois do comando ser executado, um objeto ***RequestDispatcher*** é utilizado para se obter a próxima visão.

```
    catch (Exception e) {
      request.setAttribute(resource.getMessageAttr(), "Exception occurred: " +
        e.getMessage());
      page=resource.getErrorPage(e);
    }
    // Seqüência da Mensagem 2 no exemplo do Diagrama de Seqüência.
    // Envia o controle para a visão
    dispatch(request, response, page);
    ...
    ...
    private void dispatch(HttpServletRequest request,
      HttpServletResponse response, String page) throws
      javax.servlet.ServletException, java.io.IOException {
      RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(page);
      dispatcher.forward(request,response);
    }
  }
```

4.3. Data Access Object (DAO)

4.3.1. Contexto

O Padrão de Projeto *Data Access Object* (DAO) separa a interface cliente de seu mecanismo de acesso a dados. Ele adapta o acesso a um recurso de dados específico a uma interface cliente genérica. Ele permite trocar o mecanismo de acesso a dados independentemente do código que os utiliza.

4.3.2. Problema

Os dados podem ser provenientes de diferentes mecanismos de acesso tais como bancos de dados relacionais, mainframe ou sistemas legados, serviços externos B2B, LDAP, etc. Este mecanismo pode variar dependendo do tipo de armazenamento. Permite manutenibilidade do código através da separação do mecanismo de acesso das camadas de visão e processamento. Quando o tipo de armazenamento muda, basta mudar o código que especifica o mecanismo de acesso.

4.3.3. Solução

Utilizar o padrão *Data Access Object* para abstrair e encapsular todo o acesso a fonte de dados. Deve ser implementado o mecanismo de acesso necessário para manipular a fonte de dados. A fonte de dados pode ser um sistema de banco dados relacional, um serviço externo como B2B, um repositório LDAP etc.

O diagrama de classes deste padrão é descrito na Figura 17.

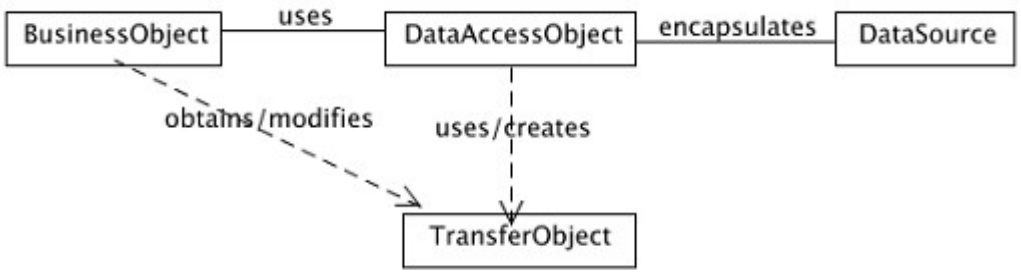

Figura 17: Diagrama de Classes para o padrão Data Access Object (DAO)

A Tabela 4 mostra as classes que são utilizadas na persistência.

Classe	Descrição
Business Object	Representa os dados do cliente. É o objeto que requer acesso à fonte de dados para obter e armazenar os dados.
DataAccessObject	É o objeto primário deste padrão. Ele abstrai a implementação de acesso aos dados para os BusinessObjects, e assim deixa o acesso transparente à fonte de dados.
DataSource	Representa a fonte de dados que pode ser um RDBMS, OODBMS, arquivo XML ou um arquivo texto. Pode também ser outro sistema (legado ou mainframe), serviço B2B ou um repositório LDAP.
TransferObject	Representa o transportador dos dados. É utilizado para retornar os dados para o cliente ou o cliente pode utilizá-lo para atualizar os dados na fonte de dados.

Tabela 4: Classes do Padrão de Projeto Data Access Object (DAO)

O Diagrama de Seqüência deste padrão é ilustrado na Figura 18.

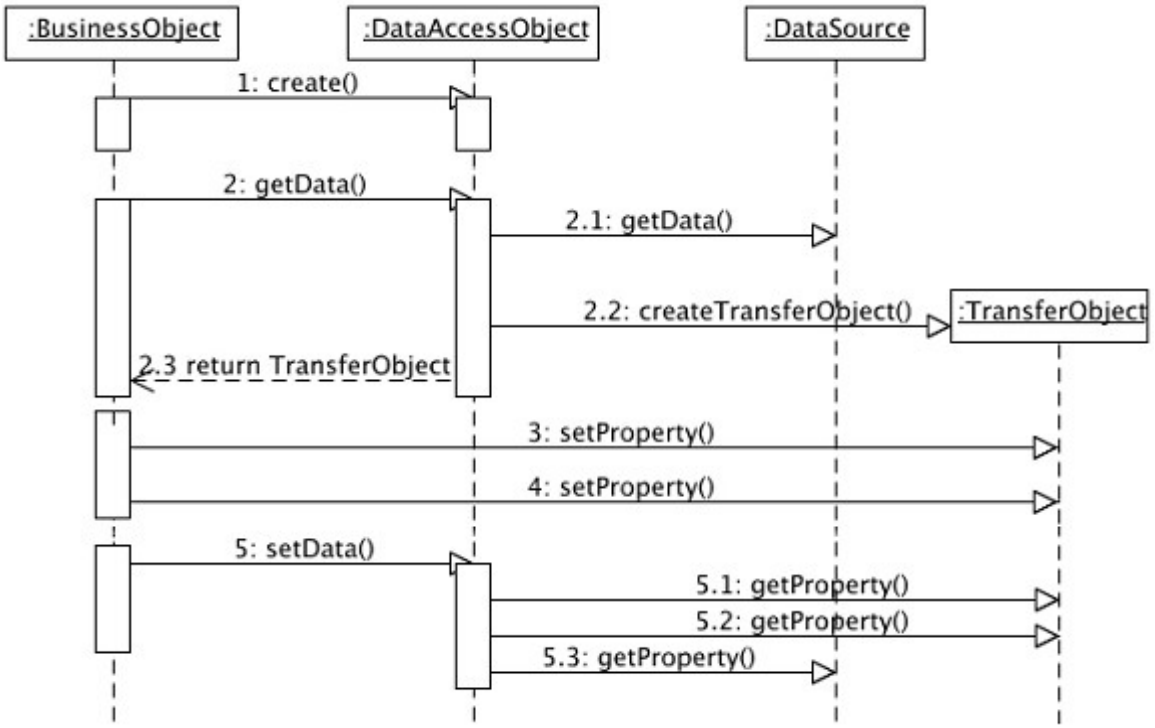

Figura 18: Objeto de Acesso a Dados (Data Access Object - DAO) Diagrama de Seqüência

A visão dinâmica mostra as interações entre as classes que compõem esse padrão. Especificamente, quatro interações podem ser observadas no diagrama acima. As interações estão listadas abaixo.

1. Um *BusinessObject* precisa criar um *DataAccessObject* para um determinado *DataSource*. Isto é necessário para recuperar ou armazenar dados do negócio. No diagrama, é a mensagem número 1.
2. Um *BusinessObject* deve requisitar a recuperação de dados. Isto é feito através do envio de uma mensagem *getData()* para o *DataAccessObject* o qual, deverá requisitar a fonte de dados (data source). Um *TransferObject* é criado para manter os dados recuperados do *DataSource*. No diagrama, é representado pela mensagem número 2 e suas sub-mensagens.
3. Um *BusinessObject* deve modificar o conteúdo do *TransferObject*. Isto simula a modificação dos dados. No entanto, os dados ainda não são modificados na fonte de dados. Isso é representado pelas mensagens número 3 e 4.
4. Para salvar dados na fonte de dados, uma mensagem *setData()* deve ser enviada para o *DataAccessObject*. O *DataAccessObject* recupera os dados do *TransferObject* e prepara-o para ser atualizado. Apenas agora, ele irá enviar a mensagem *setData()* para a *DataSource* atualizar seus registros.

4.3.4. Estratégia do Objeto de Acesso a Dados

O padrão DAO pode ser desenvolvido de forma bastante flexível adotando o Padrão *Abstract Factory* e o Padrão *Factory Method*. Se o meio de armazenamento envolvido não estiver sujeito a mudanças de uma forma de implementação para outra, o padrão *Factory Method* pode ser usado. O diagrama de classe é apresentado na Figura 19.

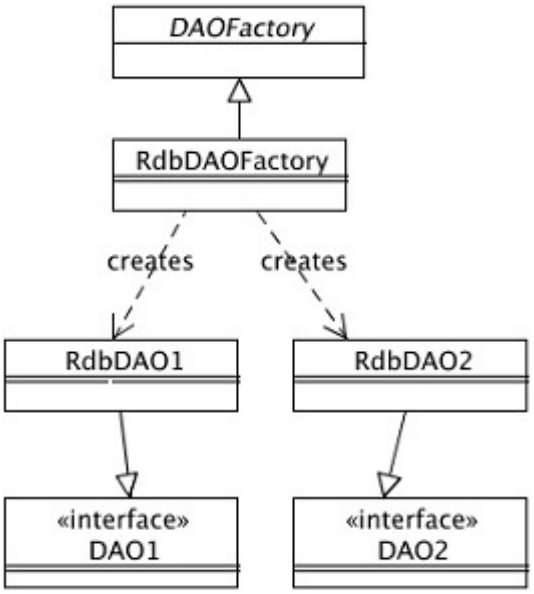

Figura 19: Estratégia Factory Method

Se o meio de armazenamento envolvido for sujeito a mudança de uma forma de implementação para outra, o *padrão Abstract Factory* pode ser usado. Ele pode crescer aos poucos e utilizar a implementação do *Factory Method*. Nesse caso, essa estratégia disponibiliza um objeto fábrica DAO abstrata que pode construir vários tipos de fábricas DAOs concretas. Cada fábrica suporta diferentes tipos de persistência de armazenamento. Quando uma fábrica concreta é obtida para uma implementação específica, ela é usada para produzir DAOs suportados e implementados nessa implementação. O diagrama de classe é representado na Figura 20.

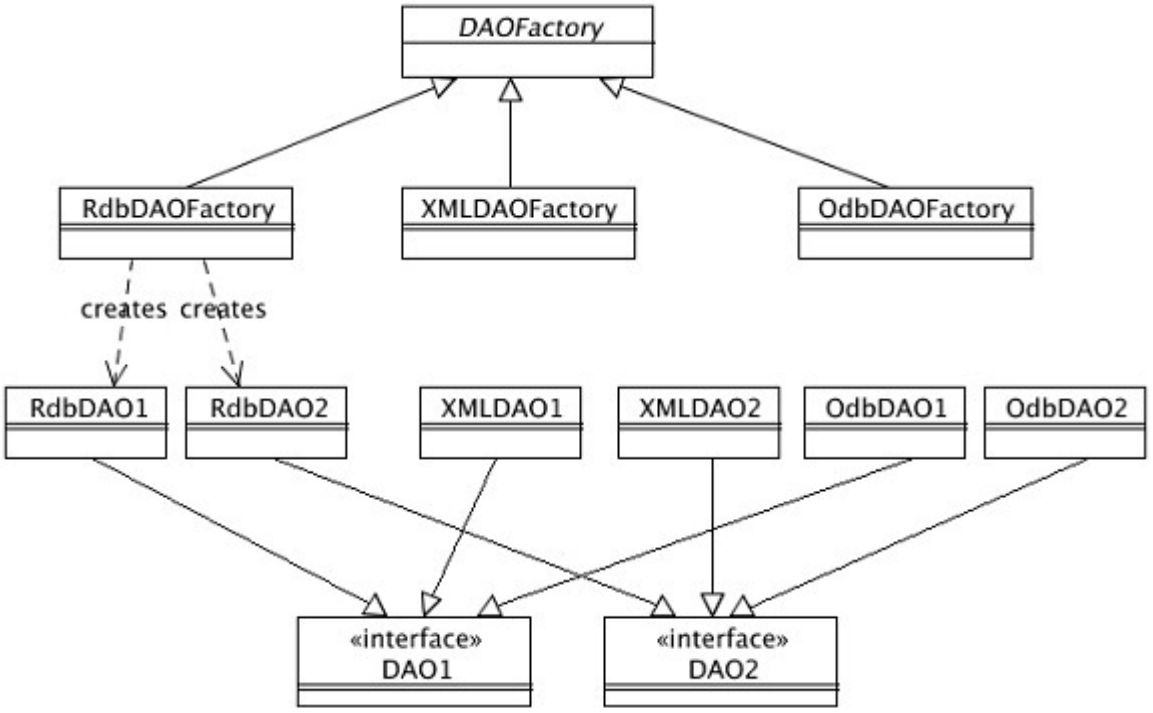

Figura 20: Padrão Abstract Factory

Neste digrama de classees, *DAOFactory* é uma abstração de um objeto *Data Access Factory* que é herdado e implementado por diferentes *DAOFactories* para suportar o acesso de armazenamento específico da implementação. Os *DAOFactories* concretos criam objetos de acesso a dados que implementam as interfaces de objetos data access.

Um exemplo de uma implementação de uma *DAOFactory* é mostrado através dos códigos abaixo. O primeiro código mostra a definição de uma classe abstrata *DAOFactory*.

```
public abstract class DAOFactory {
  //Lista de tipos DAO
  public static final int DEFAULT = 1;
  public static final int MYSQL = 2;
  public static final int XML = 3;
  ...
  ...
  // As fábricas concretas devem implementar estes métodos.
  public abstract AthleteDAO getAthleteDAO();
  public abstract AthleteStatusDAO getAthleteStatusDAO();
  ...
  ...
  public static DAOFactory getDAOFactory(int whichFactory) {
    switch(whichFactory) {
      case DEFAULT: return new DefaultDAOFactory();
      case MYSQL: return new MySQLDAOFactory();
      case XML: return new XMLDAOFactory();
      ...
      default: return null;
    }
  }
}
```

Cada um dos tipos identificados de DAO deve ser implementado. Neste exemplo, o *DefaultDAOFactory* é implementado; o outro é similar à exceção dos recursos específicos de cada implementação tais como *drivers*, *database URL* etc.

```
// Implementação padrão do DAOFactory
import java.sql.*;

public class DefaultDAOFactory extends DAOFactory {
```

```
    public static final String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
    public static final String URL = "jdbc:odbc:AngBulilitLiga";
    ...
    ...
    // método para a conexão
    public static Connection createConnection() {
      // Utilize o DRIVER e a URL para criar a conexão
      ...
    }
    public AthleteDAO getAthleteDAO() {
      // DefaultAthleteDAO implementa AthleteDAO
      return new DefaultAthleteDAO;
    }
    public AthleteStatusDAO getAthleteStatusDAO() {
      // DefaultAtleteStatusDAO implementa AthleteStatusDAO
      return new DefaultAthleteStatusDAO;
    }
    ...
  }
```

AthleteDAO é uma definição de interface e é codificado como segue:

```
  //Interface para todo AthleteDAO
  public interface AthleteDAO {
    public int insertAthlete(Athlete a);
    public int deleteAthlete(Athlete a);
    public int updateAthlete(Athlete a);
    public Athlete findAthleteRecord(String criteria);
    public RowSet selectAthlete(String criteria);
    ...
  }
```

A classe *DefaultAthleteDAO* implementa todos os métodos da interface *AthleteDAO*. O esqueleto do código desta classe é mostrado abaixo.

```
  import java.sql.*;

  public class DefaultAthleteDAO implements AthleteDAO {
    public DefaultAthleteDAO() {
      //inicialização
    }
    public int insertAthlete(Athlete a) {
      // Executa a inserção dos dados do atleta aqui.
      // retorna 0 se bem sucedido ou -1 se não.
    }
    public int updateAthlete(Athlete a) {
      // Executa a modificação dos dados do atleta aqui.
      // retorna 0 se bem sucedido ou -1 se não.
    }
    public int deleteAthlete(Athlete a) {
      // Executa a exclusão dos dados do atleta aqui.
      // retorna 0 se bem sucedido ou -1 se não.
    }
    ...
    ...
  }
```

O objeto de transferência *Athlete* é implementado usando o seguinte código. É utilizado pelo DAO para emitir e receber dados da base de dados.

```
  public class Athlete {
    private int athleteID;
    private String lastName;
    private String firstName;
    // Outros atributos de atleta inclui Guardian

    // encapsulamento dos atributos
```

```
    public void setAthleteID(int id) {
      athleteID = id;
    }
    public int getAthleteID() {
      return athleteID;
    }
    ...
  }
```

Uma parte do código do cliente que mostra o uso de um objeto de uma classe é mostrada abaixo.

```
    ...
    ...
    // cria a DAO Factory necessária
    DAOFactory dFactory = DAOFactory.getDAOFactory(DAOFactory.DEFAULT);

    // Cria a AthleteDAO
    AthleteDAO = athDAO = dFactory.getAthleteDAO();

    ...
    // Insere um novo atleta
    // Assume que atleta é uma instância de AthleteTransferObject
    int status = athDAO.insertAthlete(ath);
```

4.4. Model-View-Controller

4.4.1. Problema

As aplicações corporativas precisam suportar múltiplos tipos de usuário através de vários tipos de interfaces. Para uma aplicação isto requer um HTML a partir de cada cliente WEB, um WML para cada cliente *wireless*, uma *Interface Java Swing* para os administradores e um serviço baseado em XML para os fornecedores. Isto exige que o padrão obedeça o que segue abaixo:

- Os mesmos dados corporativos precisam ser acessados por diferentes visões.
- Os mesmos dados corporativos precisam ser atualizados através de diferentes interações.
- O suporte a múltiplos tipos de visões e interações não devem causar impacto nos componentes que integram o núcleo da funcionalidade da aplicação da empresa.

4.4.2. Contexto

A atual aplicação apresenta conteúdo aos usuários em numerosas páginas contendo vários dados.

4.4.3. Solução

O ***Model-View-Controller*** *(**MVC**)* é um padrão de projeto amplamente utilizado para aplicações interativas. Ele divide funcionalidade entre objetos envolvidos na manutenção e apresentação de dados para minimizar o grau de acoplamento entre estes objetos. É dividido em três partes chamadas *model* (modelo), *view* (visão) e *controller* (controle).

1. ***Model (modelo)***. Representa os dados corporativos e as regras de negócio que governam o acesso e a alteração a estes dados. Serve como um software aproximador do processo no mundo real, isto é, como aplicação das técnicas de modelagem para simplificar o mundo real quando da definição do modelo.
2. ***View (visão)***. Traduz o conteúdo de um modelo. Acessa os dados corporativos através do modelo e especifica como estes dados são apresentados aos atores. É responsável pela manutenção da consistência em sua apresentação quando há alterações no modelo em espera. Ele pode ser acionado através do modelo *push*, onde as visões registram a si próprias no modelo para as notificações de alterações, ou do modelo *pull*, onde a visão é responsável por chamar o modelo quando se faz necessário a recuperação do dado atual.
3. ***Controller (controlador)***. Traduz interações com a visão dentro das ações executadas pelo model (modelo). Em uma GUI de um cliente *stand-alone* aparecem para poder controlar os

pressionamentos de botões ou seleções em menus. Em uma aplicação WEB, aparecem como requisições HTTP do tipo GET e POST. As ações executadas sobre o modelo podem ativar dispositivos, processos de negócio ou mudar o estado do modelo.

A Figura 21 mostra o Padrão de Projeto *Model-View-Controller*.

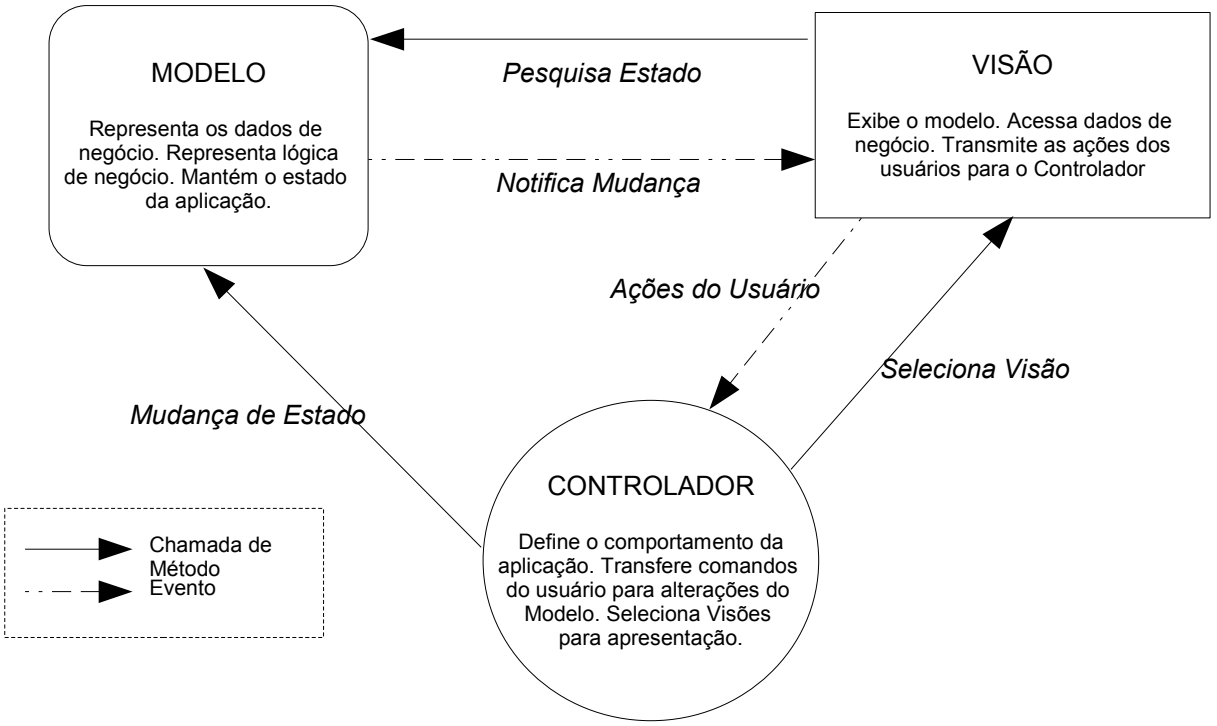

Figura 21: Padrão de Projeto MVC

As estratégias pelas quais o MVC pode ser implementado são as seguintes:

- Para clientes baseados em WEB como os navegadores, usar *Java Server Pages* (JSP) para renderizar a visão, *Servlet* como controlador, e os *Enterprise JavaBeans* (EJB) como o modelo.
- Para controlador Centralizado, ao invés de se ter vários *servlets* como controladores, um *servlet* principal é usado para fazer o controle mais gerenciável. O padrão *Front Controller* pode ser um padrão útil para esta estratégia.

5. Projeto de Dados

Também conhecido como **arquitetura de dados,** é uma tarefa de engenharia de *software* que cria um modelo dos dados em uma representação mais próxima da implementação. As técnicas usadas aqui se aproximam do campo de estudo de análise e projeto de banco de dados, normalmente discutida em um curso de banco de dados. Não discutiremos como fazer análise, projeto e implementação de bancos de dados. Para este curso, assumiremos que o banco de dados foi implementado usando o sistema relacional. O projeto lógico do banco de dados é ilustrado usando o diagrama de classes da Figura 22.

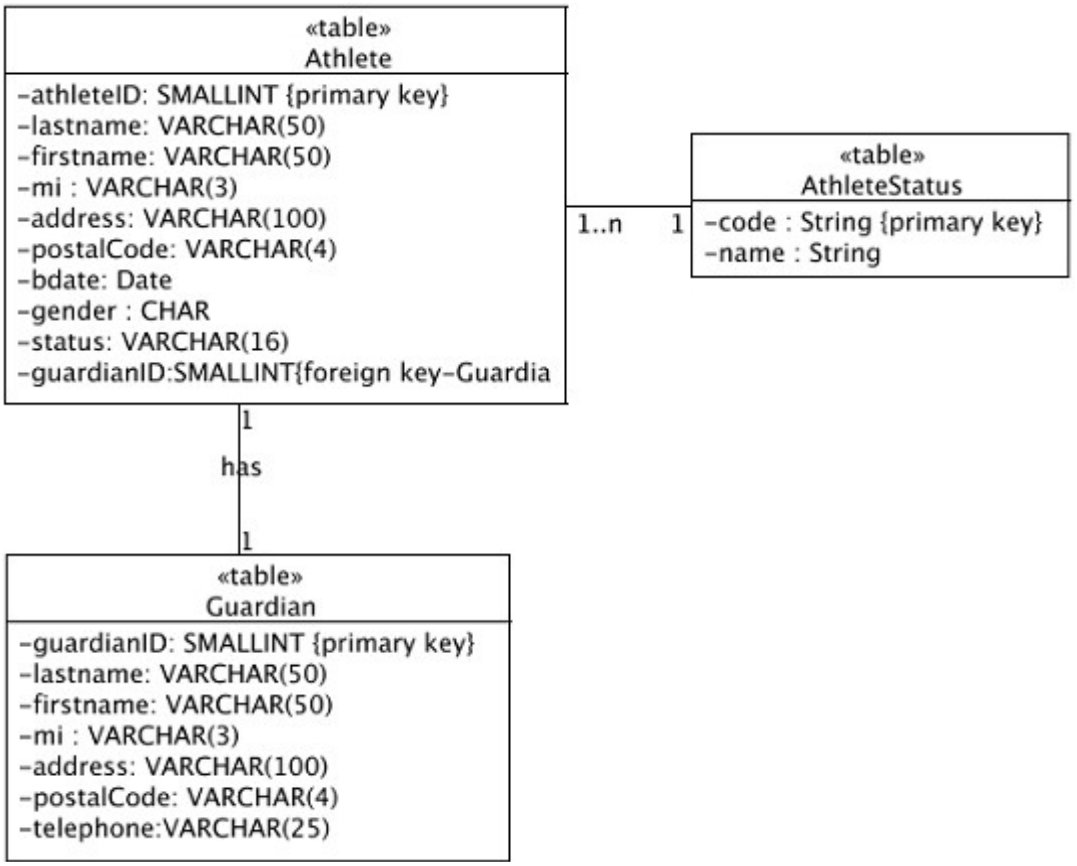

Figura 22: Projeto Lógico do Banco de Dados da Liga Ang Bulilit

Para o **Sistema de Manutenção de Sócios do Clube,** três tabelas foram definidas. A tabela **Atleta** contém os dados do atleta. A tabela **Responsável** (*Guardian*) contém os dados referentes ao responsável pelo atleta. Para simplificar, assume-se que só existe um responsável por atleta. A tabela **StatusAtleta** contém o esquema de códigos para o status do atleta.

Para acessar os dados em um sistema de banco de dados, nossos programas devem ser capazes de se conectar ao servidor de banco de dados. Esta seção discute o conceito de persistência e ensina como modelar classes de persistência. Em termos da nossa arquitetura de software, outra camada, a camada de persistência, é criada sobre a camada de banco de dados.

Persistência significa fazer que um elemento exista mesmo após a finalização da aplicação que o criou. Para classes que precisam gerar objetos persistentes, precisamos identificar:

- **Granularidade**. É o tamanho do objeto persistente.
- **Volume**. É o número de objetos para manter persistente.
- **Duração**. Define quanto tempo manter o objeto persistente.
- **Mecanismo de Acesso**. Define se o objeto é mais ou menos constante, isto é, se permitimos modificações ou atualizações.
- **Confiança**. Define como o objeto sobreviverá se um erro acontecer.

Há muitos padrões de desenho que podem ser utilizados para modelar a persistência. Serão discutidos dois padrões de persistência nas próximas seções.

5.1. Padrão JDBC

Esta seção discute o padrão de uso do mecanismo persistente escolhido para a Administração do Sistema do Banco de Dados Relacional (*Relational Database Management System* - RDBMS) que são classes *Java Database Connectivity* (JDBC). Há duas visões para este padrão, estática e dinâmica.

5.1.1. Visão estática do Padrão de Projeto de Persistência

A visão estática é um modelo de objeto do padrão. Ilustrado através do Diagrama de Classe. Figura 23 mostra o padrão para as classes persistentes.

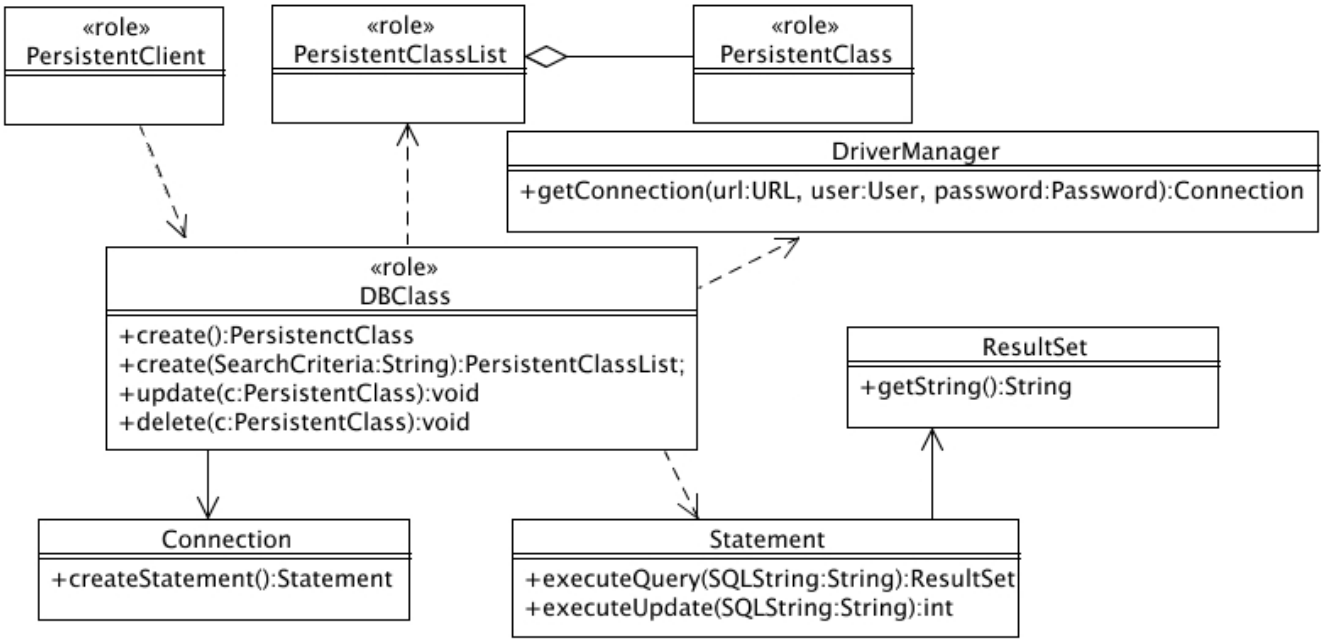

Figura 23: O Padrão JDBC e Static View

Tabela 5 mostra as classes que são usadas neste padrão.

Classe	*Descrição*
PersistentClient	Classe que pede dados para o banco de dados. Normalmente, estas são classes de controle que perguntam algo de uma classe de entidade. Trabalha com um **DBClass**.
DBClass	Classe responsável para ler e escrever dados persistentes. Também é responsável para ter acesso o banco de dados de JDBC que usa a classe de **DriverManager**.
DriverManager	Classe que devolve uma conexão ao DBClass. Uma vez que uma conexão esteja aberta o **DBClass** pode criar declaração de SQL que será enviada ao RDBMS subjacente e será executada usando a classe *Statement*.
Statement	Classe que representa uma declaração de SQL. A declaração de SQL é a linguagem de acesso ao dados do banco de dados. Qualquer resultado devolvido pela declaração de SQL é colocado na classe **ResultSet**.
ResultSet	Classe que representa os registros devolvidos por uma consulta.
PersistentClassList	Classe utilizada para devolver um conjunto de objetos persistentes como resultado de uma consulta ao banco de dados. Um registro é equivalente a um **PersistentClass** na lista.

Tabela 5: Classes do Padrão JDBC

DBClass é responsável por criar outro exemplo de classe persistente. Isto é entendido como

mapeamento OO para RDBMS. Possui o comportamento para conectar com RDBMS. Toda classe que precisa ser persistente terá um DBClass correspondente.

Visão dinâmica do Padrão de Persistência

A visão dinâmica do padrão de persistência mostra como as classes da visão estática interagem umas com as outras. O Diagrama de Seqüência é utilizado para ilustrar este comportamento dinâmico. Há vários comportamentos dinâmicos que são visto neste padrão, especificamente, eles são *JDBC Initialization*, *JDBC Create*, *JDBC Read*, *JDBC Update* e *JDBC Delete*.

1. **JDBC Initialization**. A inicialização deve acontecer antes que qualquer classe persistente possa ter acesso. Significa que a pessoa precisa estabelecer conexão com o RDBMS subjacente. Envolve o seguinte:
    - Carga do *driver* apropriado
    - Conexão com o banco de dados

O Diagrama de Seqüência da inicialização do JDBC é mostrado na Figura 24.

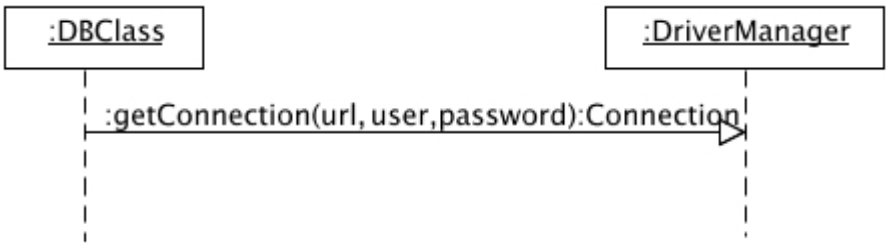

Figura 24: Inicialização JDBC

A ***DBClass*** carrega o *driver* apropriado chamando o método **getConnection(url, usuario, senha)**. Esse método procura estabelecer uma conexão de acordo com a URL do banco de dados. O *DriverManager* tenta selecionar o *driver* apropriado a partir do conjunto de *driver* JDBC registrado.

2. ***JDBC Create***. Esse comportamento cria um registro. Executa um comando INSERT da SQL. Isso assume que a conexão foi estabelecida. O Diagrama de Seqüência é mostrado na Figura 25.

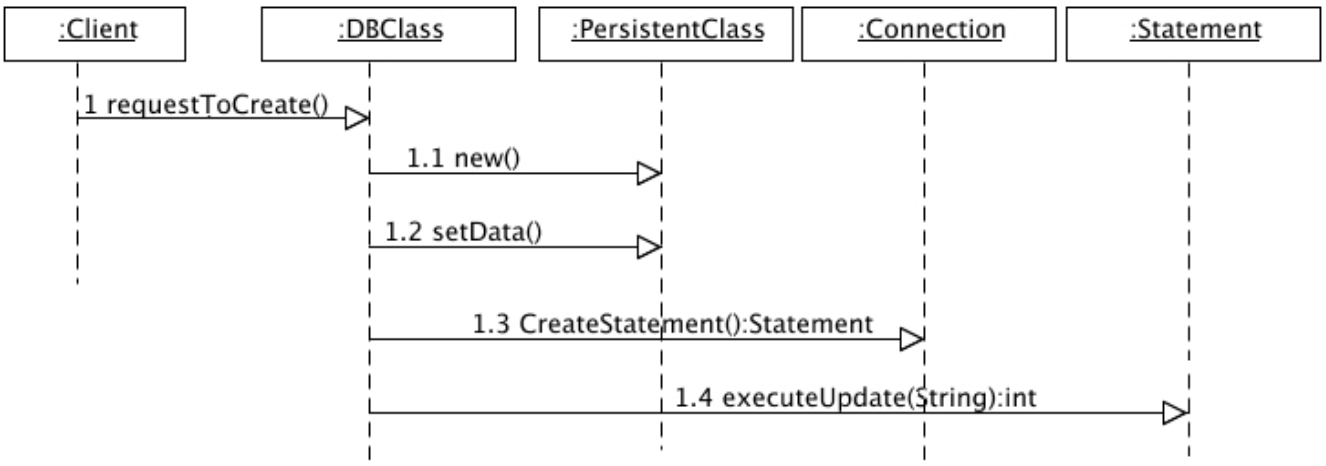

Figura 25: Criação JDBC

Descrição do Fluxo do Diagrama de Seqüência:

- O *PersistentClient* solicita a *DBClass* para criar uma nova classe.
- A *DBClass* cria uma nova instância da *PersistentClass*(new()) e atribui os valores para a *PersistentClass*
- A *DBClass* então, cria uma nova expressão usando *createStatement*() da *Connection*.
- O *Statement* é executado via método *executeUpdate*(String):int. Um registro é inserido no

banco de dados.

3. ***JDBC Read***. Esse comportamento obtêm registros do banco de dados. Executa um SELECT da SQL. Isso também assume que a conexão foi estabelecida. O Diagrama de Seqüência é mostrado na Figura 26.

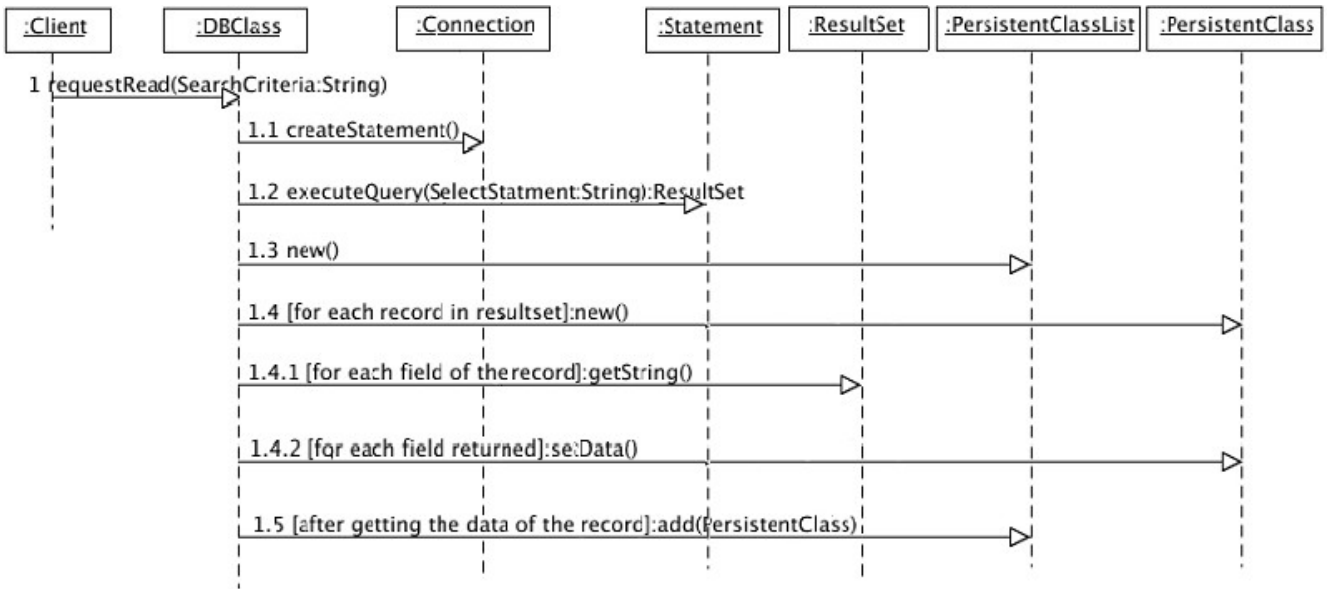

Figura 26: JDBC Read

- O *PersistentClient* pergunta à *DBClass* para obter registros do banco de dados. A String *searchCriteria* considera quais registros devem ser retornados.
- A *DBClass* cria um *Statement* a partir do comando SELECT utilizando o método *createStatement*() da *Connection*.
- O *Statement* é executado via *executeQuery*() e retorna um *ResultSet*.
- A *DBClass* instancia uma *PersistentClassList* para guardar os registros representados no *ResultSet*.
- Para cada registro no *ResultSet*, a *DBClass* instancia uma *PersistentClass*.
- Para cada campo do registro, atribui o valor do campo (*getString*()) para o atributo apropriado na *PersistentClass*(setData()).
- Depois de obter todos os dados e mapear-los para os atributos da *PersistentClass*, adiciona a *PersistentClass* em uma *PersistentClassList*.

4. ***JDBC Update***. Executa um comando UPDATE da SQL. Isso muda os valores de um registro no banco de dados. Isso assume que uma conexão foi estabelecida. O Diagrama de Seqüência é mostrado na Figura 27.

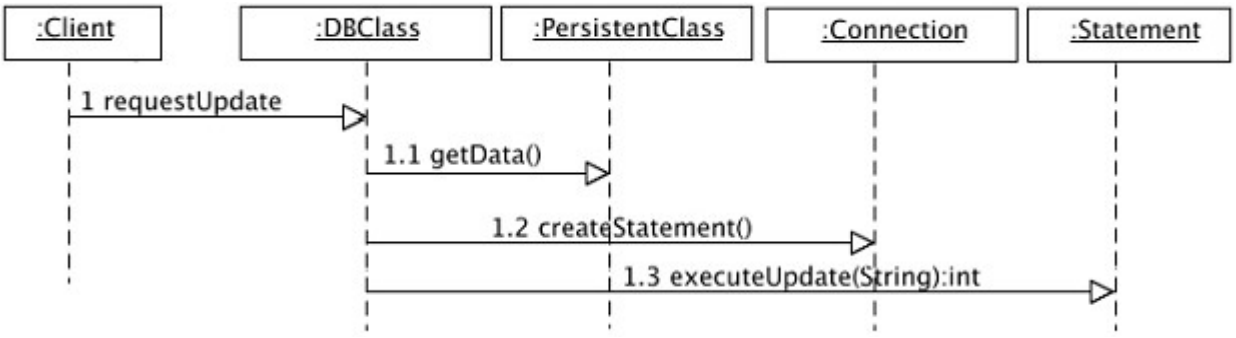

Figura 27: JDBC Update

- *PersistentClient* solicita ao *DBClass* (classe de banco de dados) a atualização de registro.

- O *DBClass* recupera os dados apropriados do *PersistentClass* (classe permanente ou constante). O *PersistentClass* deve fornecer a rotina do acesso para todos os dados existentes que o *DBClass* necessita. Isto fornece o acesso externo a determinados atributos constantes que seriam de outra maneira confidenciais. Este é o preço que se paga, para extrair o conhecimento existente fora da classe que os encapsula.
- A conexão criará uma declaração UPDATE (comando de atualização).
- Uma vez que o comando é construído, a atualização estará sendo executada e a base de dados é atualizada com os dados novos da classe.

5. ***JDBC Delete***. Executa o comando DELETE (comando de exclusão) do SQL. Elimina os registros na base de dados. Ele assume que a conexão foi estabelecida.

O Diagrama de Seqüência é mostrado na Figura 28.

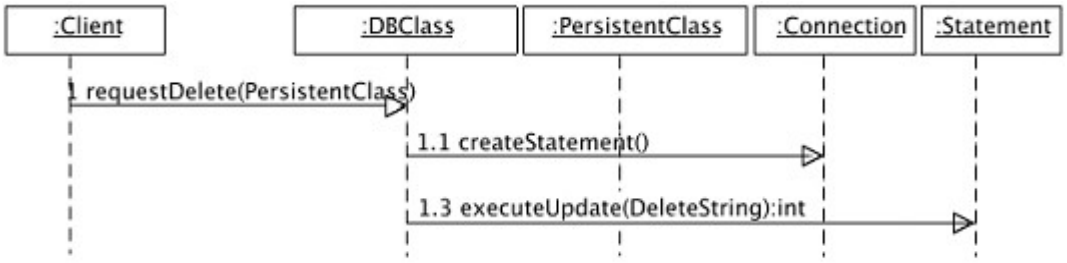

Figura 28: JDBC Delete

- *Client* solicita ao *DBClass* para deletar o registro.
- O DBClass cria um comando DELETE e executa o método *executeUpdate()*.

5.2. Desenvolvendo o modelo de projeto dos dados.

No desenvolvimento do modelo de projeto de dados, um modelo deve ser realizado. No seguinte exemplo, usaremos o JDBC para modelar o projeto dos dados da sociedade de um clube.

PASSO 1: Definir a visão estática dos elementos de dados que necessitam ser modelados

Usando a ferramenta de molde de projeto , modelamos as classes de colaboração.Figura 29 O *DBAthlete* é a classe utilizada para conexão e execução dos comandos SQL. Outras classes, *PCLAthlete*, *Athlete* e *Guardian*.

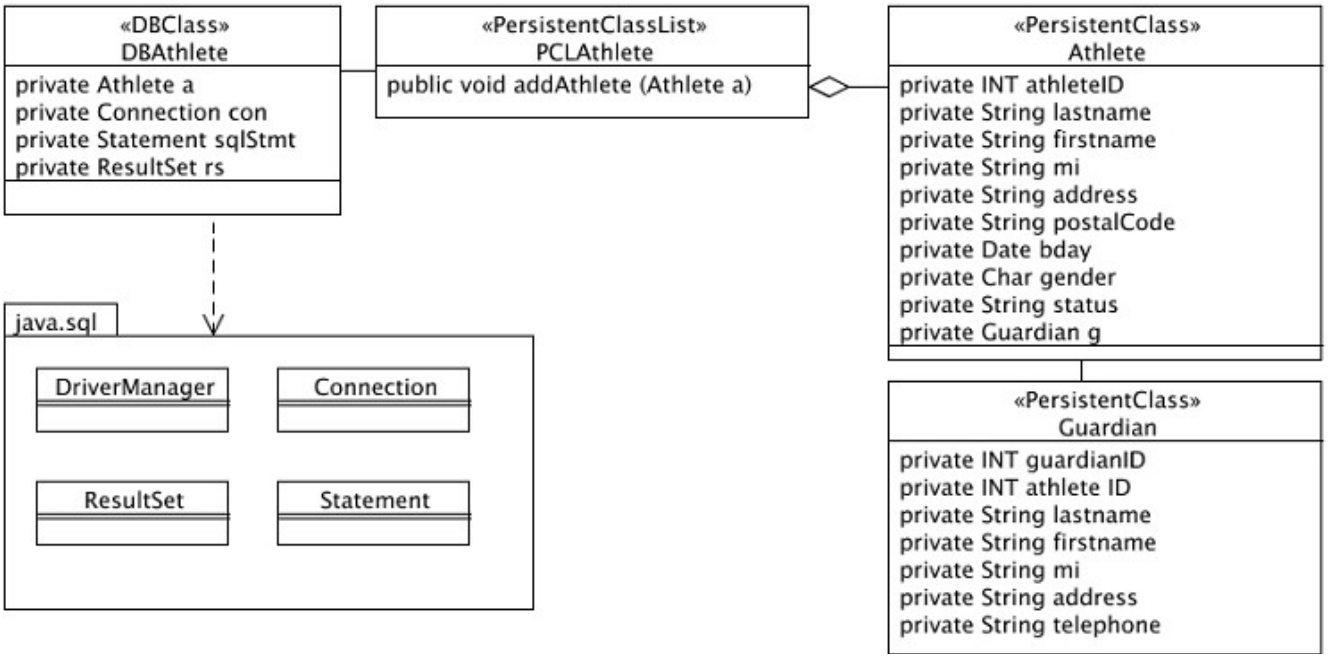

Figura 29: Visão estática do DBAthlete

PASSO 2: Modelar todo comportamento das classes em um Diagrama de Seqüência.

Todas as Visualizações dinâmicas do JDBC devem ser modeladas para determinar como as DBClasses e as Classes Persistentes irão colaborar usando os Diagramas de Seqüência. Os Diagramas que se seguem ilustram as funções JDBC de Leitura (Figura 30), Criação (Figura 31), Atualização (Figura 32) e Exclusão (Figura 33).

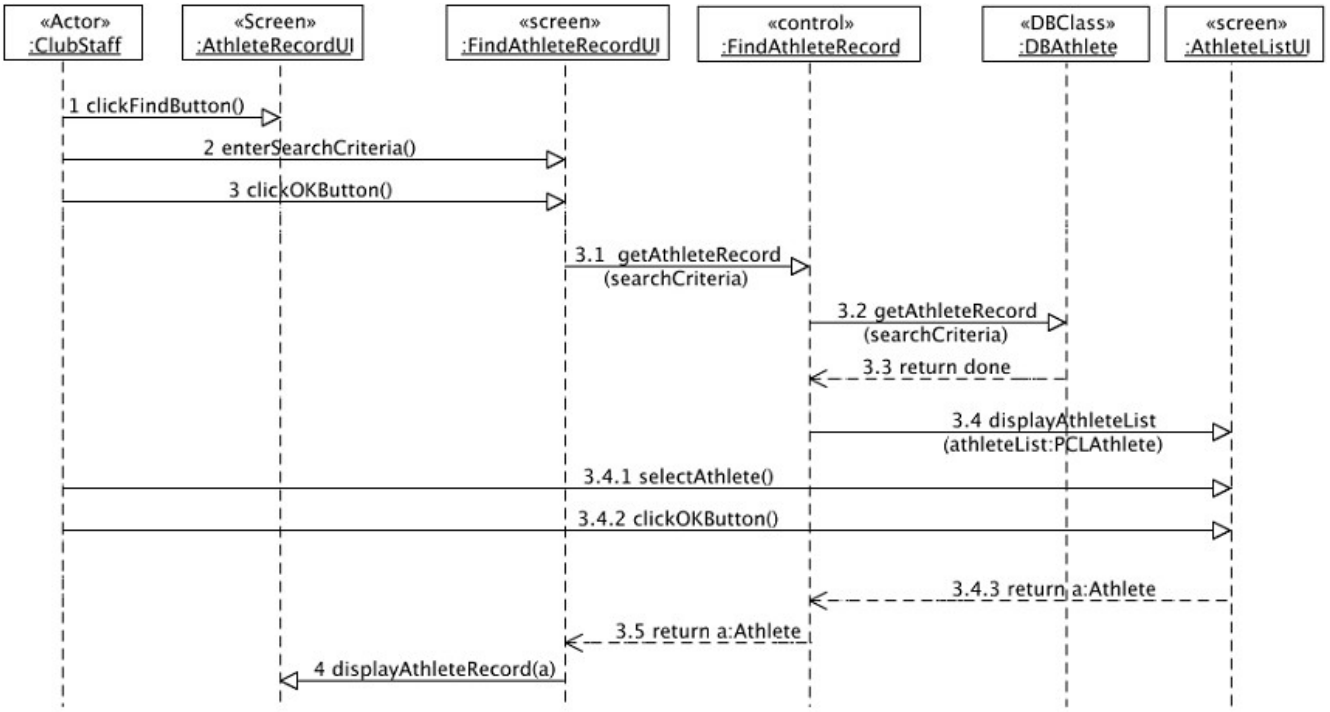

Figura 30: DBAthlete JDBC Read

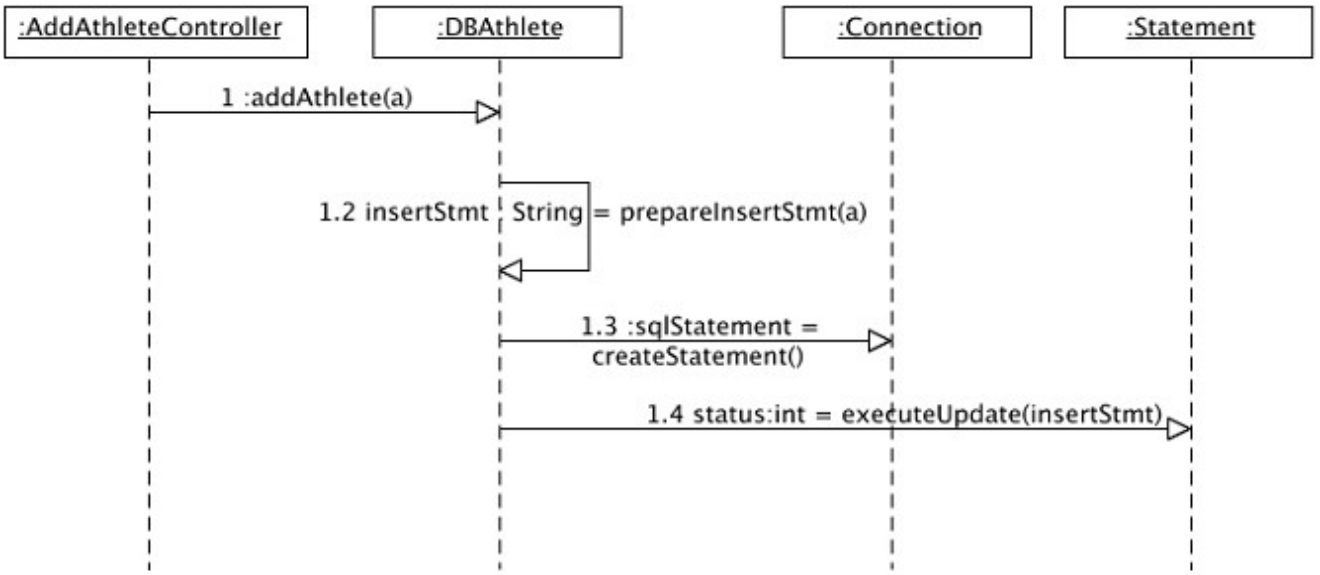

Figura 31: JDBC Create DBAthlete

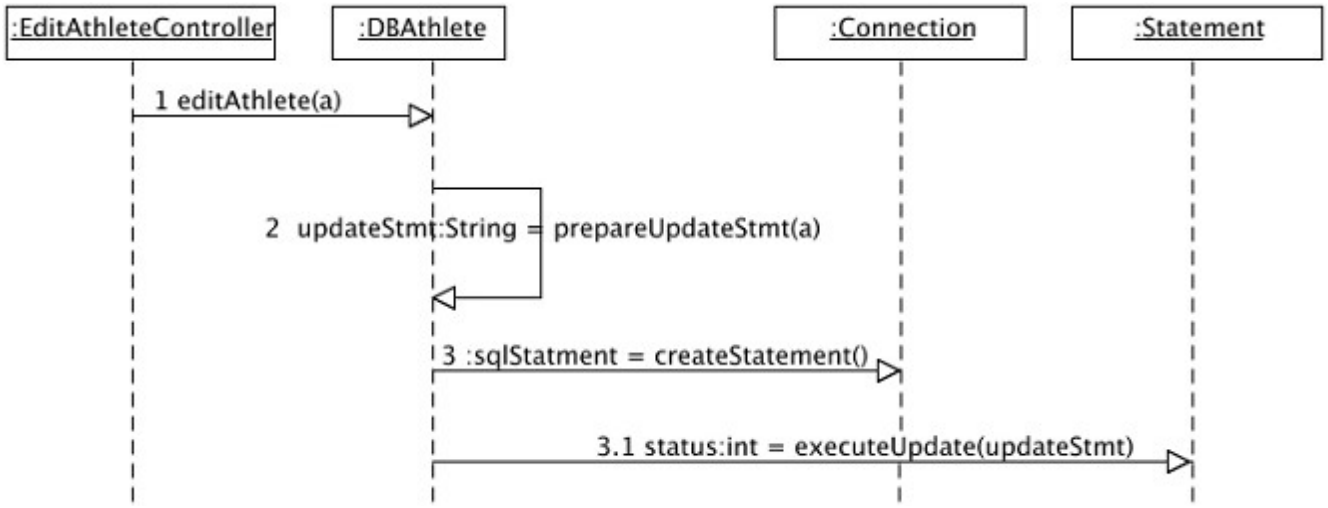

Figura 32: JDBC Update DBAthlete

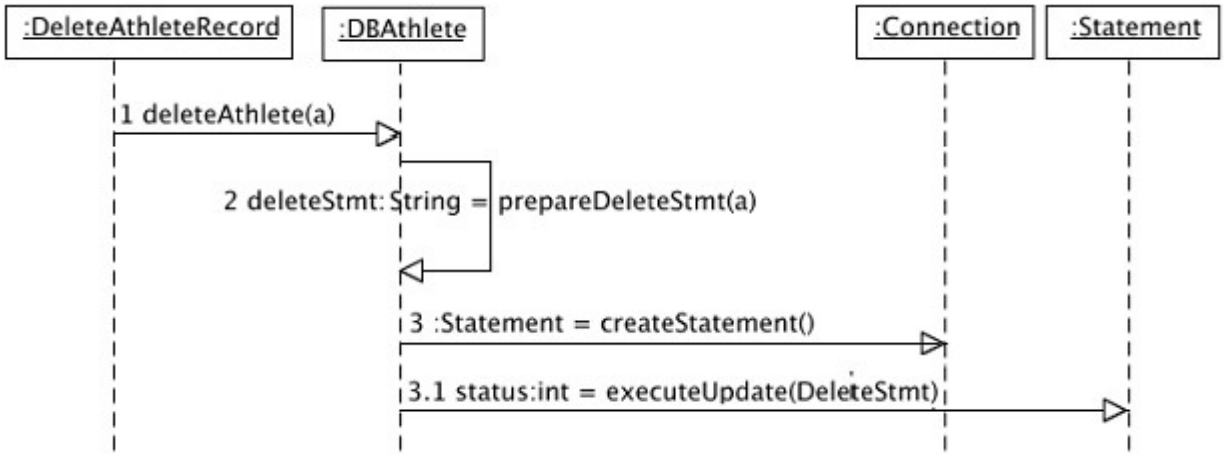

Figura 33: JDBC Delete DBAthlete

PASSO 3: Documentar as definições das classes de de dados.

Identifique os atributos e operações de cada classe. Veja os exemplos na Figura 34.

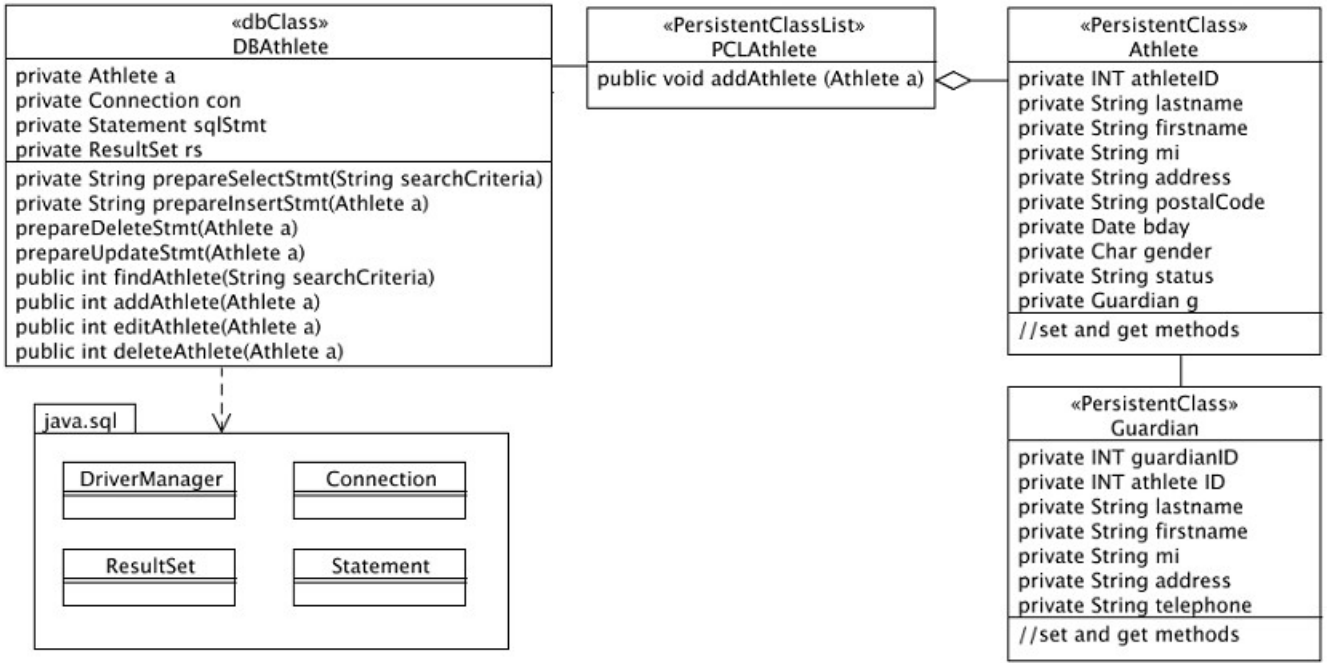

Figura 34: Definição da Classe Athlete e das classes persistentes

PASSO 4: Modificar a arquitetura do software para incluir os dados das classes.

Deve estar na camada de banco de dados. Em sistemas simples, esta se torna parte da camada de lógica empresarial. É possível modificar decisões tomadas. Neste momento, classes de entidade são substituídas pelo *DBClasses*, Classes Persistentes e Projeto de Banco de Dados. A modificação é mostrada na Figura 35.

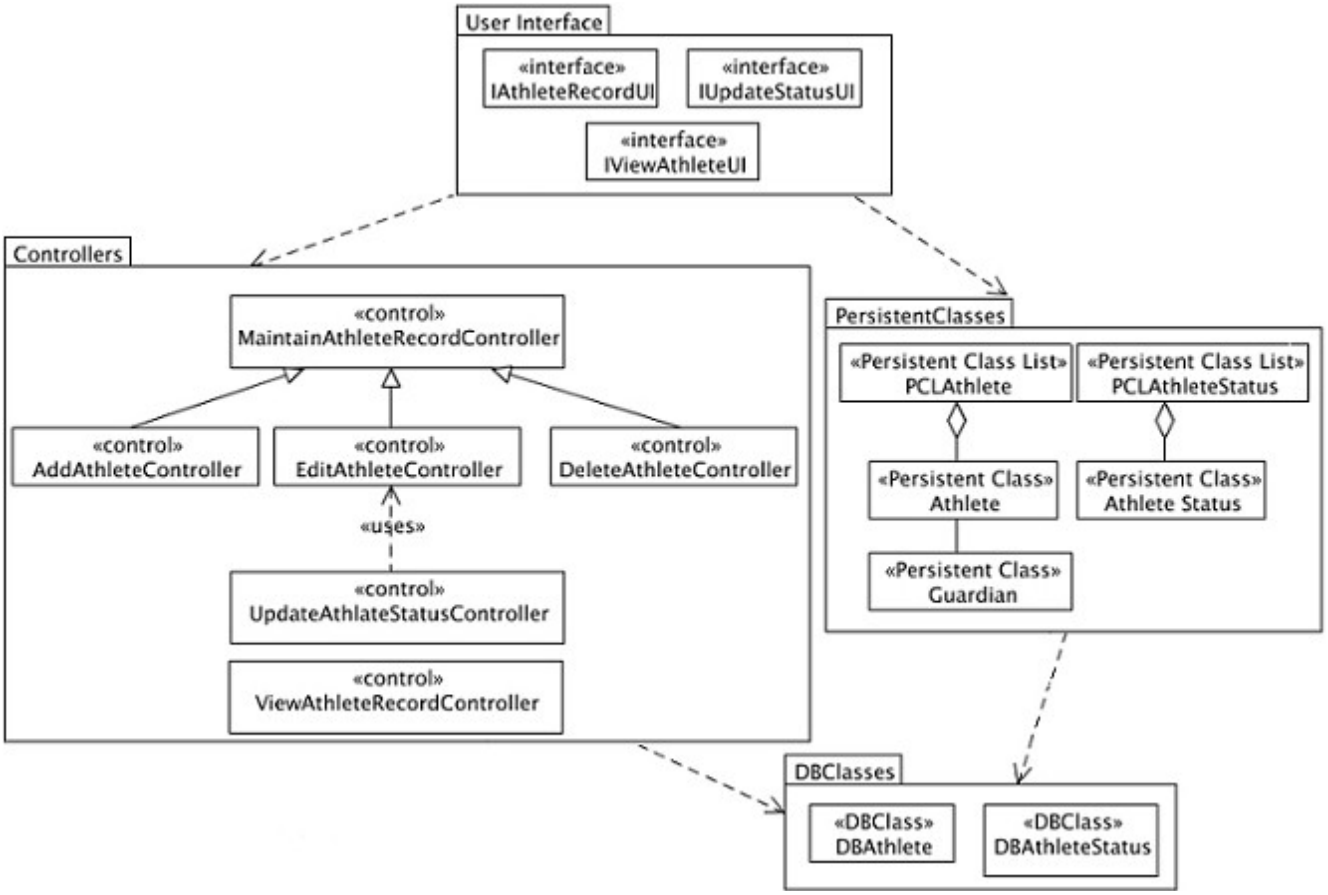

Figura 35: Modificação da Arquitetura de software para o Modelo de Dados

6. Projeto de interface

É interessante lidar com elementos projetados para facilitar a comunicação do *software* com os humanos, dispositivos e outros sistemas que inter-operam entre si. Nesta seção, poderemos concentrar em projetados de elementos que interagem com as pessoas, particularmente, formulários de pré-impressão, relatórios e telas.

Um bom projeto de formulários, relatórios e telas são críticos para determinar como o sistema será aceito pelo usuário final, estabelecendo o seu sucesso ou fracasso. Com a Engenharia de Software, queremos diminuir a ênfase em relatórios e formulários e aumentar a ênfase em telas.

Quando usamos relatórios?

- Relatórios são utilizados para examinar um caminho e efetuar controles.
- Relatórios são usados para conformidades com agências externas.
- Relatórios são usados para grandes volumes de navegação .

Quando usamos formulários?

- Formulários são usados para se tornarem documentos.
- Formulários são usados se necessitar de transações personalizadas mas não se tem acesso as estações de trabalho.
- Formulários são usados se são documentos jurídicos importantes.

Quando usamos telas?

- Telas são usadas para selecionar um simples registro do banco de dados.
- Telas são usadas para pequenos volumes de saída.
- Telas são usadas para intermediar passos ao longo de um processo interativo.

6.1. Projeto de Relatórios

Relatórios podem ser colocados num papel comum, formulários contínuos, telas-bases ou microfilmagem ou microfichas. A seguir damos os passos no projeto dos relatórios.

6.1.1. Considerações no Projeto de Relatórios

1. Número de Cópias e Volume. Deve haver um equilíbrio entre o número de cópias geradas com o número de páginas.
   - Quantas cópias de relatórios serão geradas?
   - Na média quantas páginas serão geradas?
   - O usuário pode ficar sem relatórios?
   - Com qual freqüência o usuário arquiva uma cópia? É realmente necessária?
2. Geração de relatório
   - Quando os usuários necessitam de relatórios?
   - Qual o período de interrupção de entrada ou processamento de dados antes do que os relatórios sejam gerados?
3. Freqüência dos relatórios
   - Com que freqüência os usuários necessitam dos relatórios?
4. Figuras no relatório
   - O usuário requisitou e faz a geração de relatórios para gerar tabelas, figuras, gráficos entre outros.
5. Mídia

- Onde os relatórios serão produzidos? Em uma tela CRT, formulário contínuo comum, formulário de pré-impressão, papel timbrado, Microfilme ou microficha, mídia de armazenamento, etc.

6.1.2. Princípios de *Layout* de relatórios

1. Aderir para saída padrão da zona de dados para o *layout* do relatório. As zonas de dados são cabeçalho, linhas de detalhes, rodapé e sumários.
2. Considere as dicas a seguir no projeto de cabeçalho de relatórios.
   - Sempre incluir datas nos relatórios e número de páginas.
   - Descreva claramente o relatório e conteúdo.
   - Identifique os cabeçalho de coluna nos itens de dados listados.
   - Alinhe o cabeçalho da coluna com o comprimento dos itens de dados.
   - Remova nomes de empresas do interior do relatório para finalidades de segurança.
3. Considere as dicas a seguir no projeto da linha de detalhe.
   - Uma linha deve representar um registro de informação. Detalhes adequados on-line. Se mais de uma linha é necessária, campos alfa numéricos são agrupados na primeira linha e campos numéricos são agrupados nas linhas seguintes.
   - Os campos importantes são colocados no lado esquerdo.
   - As primeiras cinco linhas e as últimas cinco linhas são utilizadas para os cabeçalhos e rodapés. Não usar estes espaços.
   - É uma boa idéia para agrupar o número de detalhes por página para valor monetário semelhante a 10's, 20's e 50's.
   - Se for tendência partes do relatórios, representar desenhos tal como gráficos.
   - Agrupe Itens relacionados, semelhantes a Vendas Atuais com Venda Finais por Ano ou número de Clientes, Nome e Endereço.
   - Para representar números semelhantes a moeda, poderá ser apropriado incluir marcas de pontuação e sinais.
   - Evitar imprimir dados repetitivos.
4. Considere as dicas a seguir no projeto do rodapé do relatório.
   - Coloque o total de páginas, totais principais e tabelas de código-valor nesta área.
   - Também é bom colocar um indicador sem seguinte.
5. Considere as dicas a seguir no projeto do sumário de páginas.

6.1.3. Desenvolvimento do *Layout* dos Relatórios

PASSO 1: Definir o padrão do *layout* do relatório.

Como um exemplo, o seguinte é o padrão para definir os relatórios do Sistema Club Membership Maintenance.

- Título de relatório. Deve conter o título do relatório, data de geração, nome de clube e número de página.
- Corpo de relatório. Deve ter os dados descritivos à direita e no lado esquerdo os dados numéricos.
- Rodapé. Instruções ou explicação de cada código enquanto que a última página contém os principais totais e a indicação de final do relatório.
- Dados importantes são posicionados no documento da esquerda para a direita.
- Códigos devem ser explicados quando utilizados.

- Datas utilizam o formato de DD/MM/YY.
- Usar logotipos em documentos distribuídos externamente.

PASSO 2: Prepare os Planos de Relatório.

Os padrões são usados como guia para o implementação. Para representar dados que aparecerão nas colunas do relatório use as convenções de formato seguintes.

Formato	*Descrição*
9	Isto significa um dado numérico. O número **9** indica os dígitos. Exemplos: 9.999 – O número em milhar 999,99 – uma centena com duas casas decimais
A	Indica uma letra do alfabeto. A-Z ou a-z. Símbolos não são permitidos.
X	Indica um tipo alfanumérico. Símbolos são permitidos.
DD/MM/YY	Indica data no formato dia/mês/ano
HH:MM:SS	Indica hora no formato hora:minuto:segundo

Tabela 6: Convenções de Formato

Figura 36 é um exemplo de *layout* de relatório seguindo as convenções do formado de datas especificados na Tabela 6. Este é um relatório de uma listagem geral da base de Athletes por estado.

Ang Bulilit Liga Club

Lista de Sócios por Estado

Realizado em: DD/MM/YYYY

Pág. 99

Estado: XXXXX

No.	Nome	Sobrenome
9999	Xxxxxxxx	Xxxxxxxx
9999	Xxxxxxxx	Xxxxxxxx
9999	Xxxxxxxx	Xxxxxxxx

Número Total no estado XXXXX de sócios : 9999

Figura 36: Exemplo de Layout do relatório

6.2. Projeto de Formulários

Formulários são normalmente utilizados para entrada de dados quando uma *workstation* não estiver disponível (não confundir com formulário de tela, estamos tratando de formulários em papel). Às vezes é usado como um documento de *turnaround*. Os seguintes passos definem o projeto de formulários.

6.2.1. Guias gerais no layout do Formulários

1. Instruções devem aparecer no formulário exceto quando a mesma pessoa irá utilizá-lo inúmeras vezes. As instruções gerais são colocadas na parte superior. Devem ser breves. Devem ser colocadas instruções específicas antes dos artigos correspondentes.
2. Utilizar palavras familiares. Evite não usar "não", "exceto" e "a menos que". Utilize orações positivas, ativas e curtas.
3. O espaço de resposta deve ser suficiente para que o usuário possa entrar com toda a

informação necessária confortavelmente. Se o formulário será utilizado como em uma máquina de escrever, 6 ou 8 linhas por polegada é uma medida suficiente. Para espaços de resposta manuscritos, 3 linhas por polegada com 5 a 8 caracteres por polegada é uma medida suficiente.

4. Use as seguintes diretrizes para tipos das fontes.
    - *Gothic*
        - Simples, *squared off*, sem *serifs*
        - Fácil ler, até mesmo quando comprimido
        - Letras maiúsculas são desejáveis para títulos
    - *Italic*
        - *Serifs* e itálica
        - Difícil para ler em grandes quantidades de textos longos
        - Bom para destacar um trabalho ou frase
    - *Roman*
        - *Serifs*
        - Melhor para grandes quantidades
        - Bom para instruções
5. Letras minúsculas são mais fáceis de ler do que maiúsculas. Porém, um tamanho de fonte pequeno não deve ser utilizado. Use letras maiúsculas para chamar a atenção a certas declarações.

6.2.2. Desenvolvendo os *Layouts* do Formulário

PASSO 1: Defina os padrões a serem utilizados para os projeto dos formulários.

Um exemplo de um padrão para projeto de formulários é listado abaixo.

- Tamanho de papel não deve ser maior que 8 1/2” x 11.”
- Esquema de cores:
    - Cópias brancas vão para os candidatos.
    - Cópias amarelas vão para a diretoria do clube.
    - Cópias vermelhas vão para o treinador.
- Devem ser posicionados dados importantes do lado esquerdo do documento.
- Datas devem utilizar o formato *DD/MM/YY*.
- Logotipos são utilizados externamente para distribuir documentos.
- Cabeçalho deve incluir Número de Página e Título.
- Deixar espaço para grampear o papel.
- Seguir 3 linhas por polegada.

PASSO 2: Prepare Amostras de Formulários.

Utilizando o formato padrão definido no passo 1, projetar o *layout* dos formulários. Redesenhe caso seja necessário.

6.3. Desenho de Telas e Diálogos

As classes de fronteira são refinadas para definir telas que os usuários utilizarão e a interação (diálogo) das telas com outros componentes do software. Em geral, há duas metáforas usadas no desenho de interfaces de usuário.

1. ***Metáfora do Diálogo*** é uma interação entre o usuário e o sistema através do uso de menus, teclas de função ou a entrada de um comando através de uma linha de comando. Tal

metáfora é largamente usada na abordagem estruturada de desenvolvimento de interfaces de usuário que são mostradas em terminais a caractere ou de modo texto.

2. ***Metáfora da Manipulação Direta*** é uma interação entre o usuário e o sistema usando interfaces de usuário gráficas (GUI). Tais interfaces são orientadas a eventos, isto é, quando um usuário, digamos, que clique um botão (evento), o sistema responderá. Ele dá ao usuário a impressão de estar manipulando objetos na tela.

Para desenvolvimento orientado a objeto usando ambientes integrados de programação visual, usamos a metáfora da manipulação direta. Contudo, ao desenhar interfaces, várias recomendações devem ser seguidas.

6.3.1. Recomendações para o Desenho de Telas e Interfaces

1. Vá pela regra: UMA IDÉIA POR TELA. Se a idéia principal mudar, outra tela deverá aparecer.
2. Padronize o *layout* da tela. Isto envolve definir orientações de estilo que devem ser aplicadas a todos os tipos de elementos de tela. Orientações de estilo garantem a consistência da interface. A consistência ajuda os usuários a aprender a aplicação mais depressa uma vez que a funcionalidade e a aparência são as mesmas em diferentes partes da aplicação. Isto se aplica aos comandos, ao formato dos dados como apresentados ao usuário, como formato de datas, ao *layout* das telas, etc. Diversos fabricantes de software oferecem orientações de estilo que você poderá usar. Orientações de estilo comuns são listadas abaixo:
   - Orientações de Desenho *Java Look and Feel* (http://java.sun.com)
   - As Orientações para Desenho de Software da Interface Windows (*Windows Interface Guidelines for Software Design* - http://msdn.microsoft.com)
   - As Orientações para Interface Humana Macintosh (*Macintosh Human Interface Guidelines* - http://developer.apple.com)
3. Ao apresentar itens de dados ao usuário, lembre-se sempre do seguinte:
   - Eles devem ser rotulados ou nomeados, e com os mesmos nomes em toda a aplicação. Como exemplo, suponha que você tenha um item de dado que representa um número de identificação de aluno. Você pode escolher usar o nome 'ID do Aluno'. Em todas as telas onde aparecer este item, use o nome 'ID do Aluno' como rótulo; não mude o rótulo para 'Número do Aluno' ou 'Nº do Aluno'.
   - Tanto quanto possível, eles devem se localizar no mesmo ponto em todas as telas, particularmente, nas telas de entrada de dados.
   - Eles devem ser mostrados da mesma forma como no caso da apresentação de datas. Se o formato das datas seguir o formato 'DD-MM-AAAA', então todas as datas deverão ser apresentadas neste formato. Não mude o formato.
4. Para controladores que requeiram um tempo maior para processar, mantenha o usuário informado do que está acontecendo. Use mensagens ou indicadores de barra do tipo "XX% completado".
5. Se possível, é recomendável informar ao usuário do próximo passo. O desenho de diálogos do tipo assistente é sempre uma boa abordagem.
6. Apresente mensagens de erro e telas de ajuda significativas. EVITE A IRREVERÊNCIA!

6.3.2. Desenvolvendo o Desenho de Telas e Diálogos

PASSO 1: Prototipação da interface de usuário.

Um ***protótipo*** é um modelo que parece, e até certo ponto, comporta-se como o produto final, mas não apresenta certas funcionalidades. É como o modelo de escala de um arquiteto para um novo prédio. Há diversos tipos de protótipos.

1. ***Protótipos horizontais*** oferece um modelo da interface de usuário. Ele lida com a apresentação apenas ou camada visual da arquitetura do software. Basicamente, este tipo de protótipo mostra a seqüência de telas quando o usuário interage com o software.

Nenhuma funcionalidade é emulada.

2. ***Protótipos verticais*** tomam um sistema funcional do sistema todo e o desenvolvem em cada camada: interface de usuário, alguma funcionalidade lógica e banco de dados.
3. ***Protótipos descartáveis*** são protótipos que são posteriormente descartados depois de servir ao seu propósito. Este tipo de protótipo é feito durante a fase de engenharia de requisitos. Tanto quanto possível, evitamos criar tais protótipos.

O uso de um ***Ambiente de Programação Visual*** fornece suporte aos protótipos de desenvolvimento. NetBeans e Java Studio Enterprise 8 oferecem meios de criar visualmente os nossos protótipos.

Para demonstrar o **Passo 1**, imagine que estamos construindo um protótipo de tela para a manutenção de um registro de atleta. A tela principal, cujo nome é *Athlete* (Atleta) (Figura 37), permite-nos manusear um único registro de atleta. Quando um usuário clica o *Find Button* (Botão Procurar), a tela *Find an Athlete* (Procurar um Atleta) (Figura 38) aparece. Esta tela é usada para especificar o critério de busca que determina quais registros de atletas serão recuperados da base de dados. Quando o critério tiver sido pressionado o botão OK, a tela com a *Athlete List* (Lista de Atletas) (Figura 38) será mostrada contendo a lista de atletas que satisfaz ao critério de busca. O usuário poderá selecionar o registro colocando a barra de seleção sobre o nome. Quando o botão OK for pressionado, retornaremos à tela *Athlete* (Atleta) onde os atributos do atleta escolhido serão mostrados nos elementos de tela apropriados.

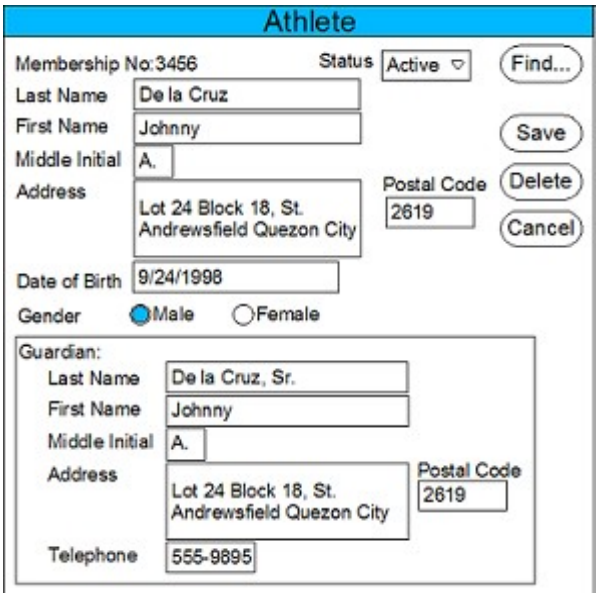

Figura 37: Tela principal de Athlete

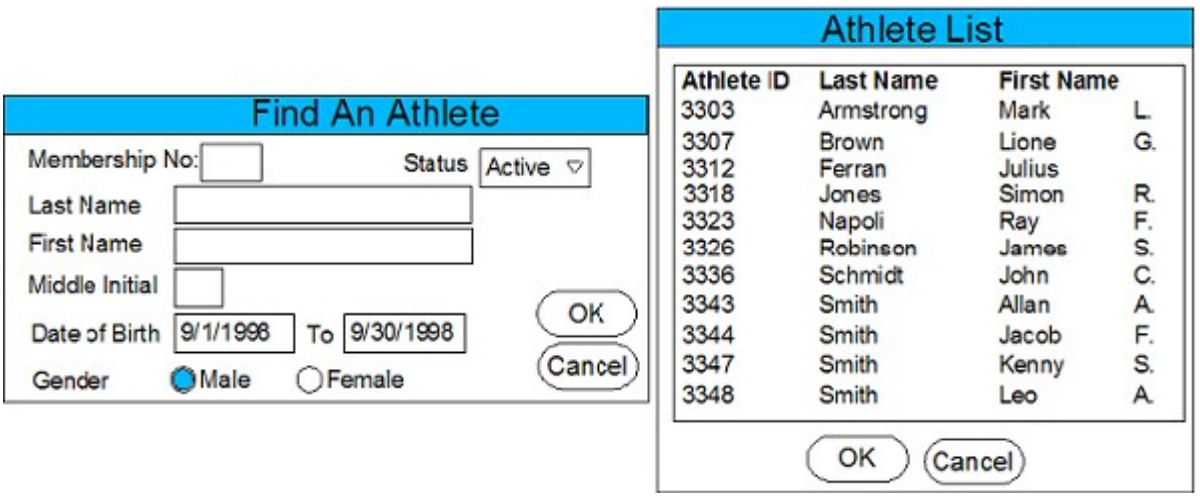

Figura 38: Exemplo da Tela de FindAthleteRecordUI e AthleteListUI

Nos passos seguintes, serão tratadas telas como únicos objetos. Na realidade, podem ser uma instância de uma subdivisão de classe como Diálogo. Por exemplo, veja a Figura 39.

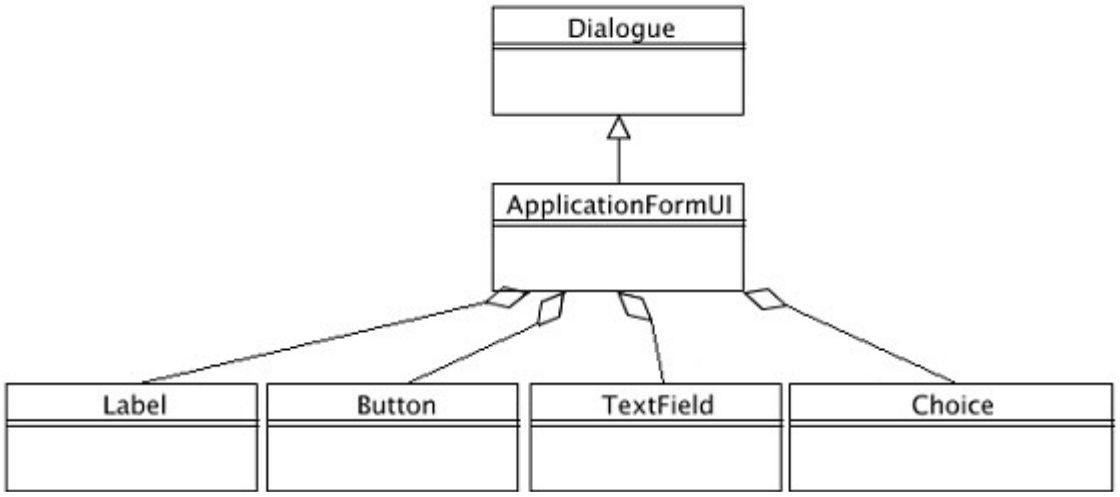

Figura 39: Exemplo da Tela dos Comandos FindAthleteRecordUI e AthleteListUI

Em Java, é possível utilizar os componentes AWT e SWING. Eles são importados na classe que os usa utilizando as seguintes instruções:

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

PASSO 2: Para cada tela, realize o projeto da classe da tela.

Este passo exigiria-nos uma modelagem da tela como uma classe. Como por exemplo, a Figura 40 que mostra a definição da classe para a tela de Atleta. O nome da classe é AthleteRecordUI. Tem um estereótipo <<screen>>.

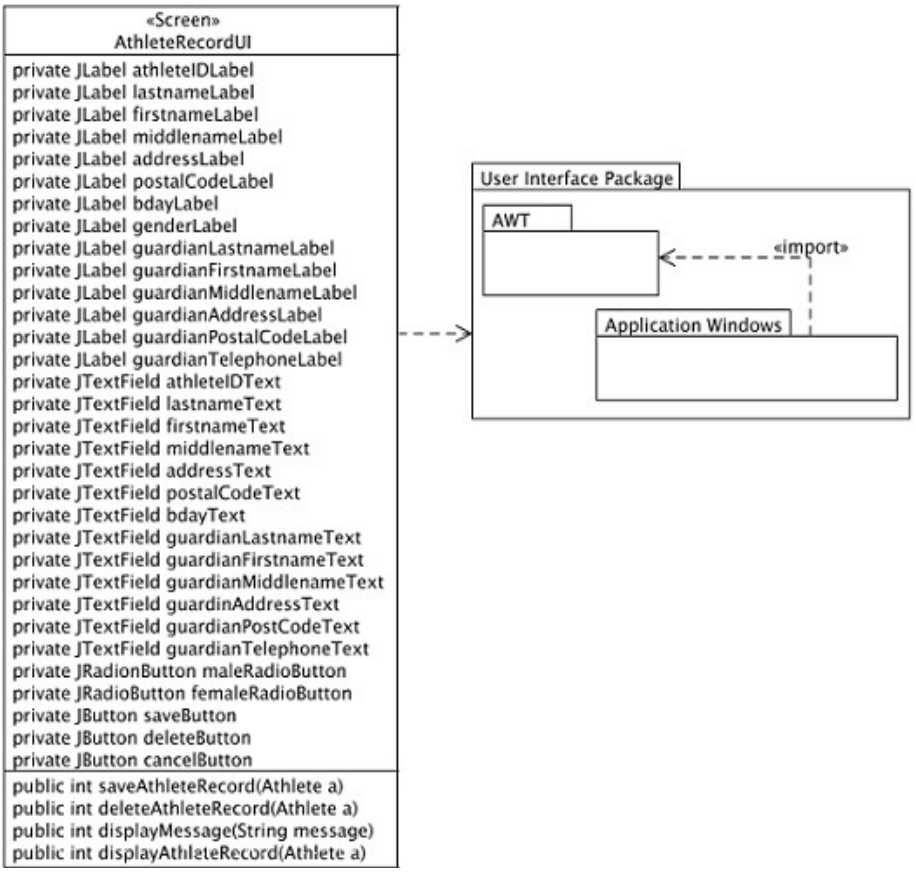

Figura 40: Exemplo da Tela dos Comandos FindAthleteRecordUI e AthleteListUI

Neste exemplo, assumimos que estaremos utilizando a bibliotecas de *Abstract Windowing Toolkit* (AWT) como componentes de apoio. Também é necessário modelar as telas "*Find an Athlete*" e "*Athlete List*" como classes; elas são nomeadas, respectivamente, para *FindAthleteRecordUI* e *AthleteListUI*. Empacotar todas as telas para formar o subsistema. Este subsistema deve implementar a interface *IAthleteRecordUI* para a classe limite. Veja a Figura 41.

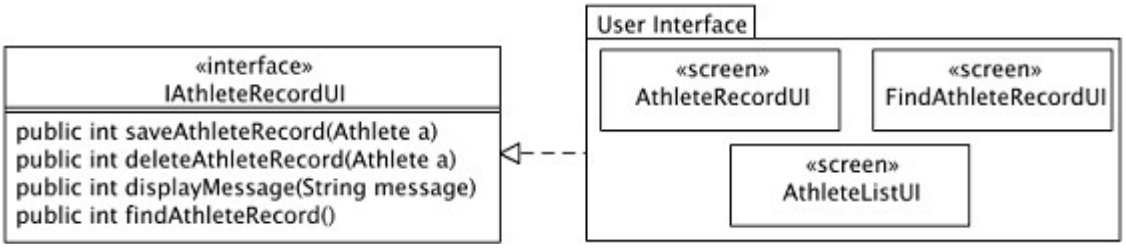

Figura 41: Exemplo da Tela dos Comandos FindAthleteRecordUI e AthleteListUI

PASSO 3: Para cada classe de tela, modele o comportamento da tela com outras classes.

O comportamento da classe de tela relativa a outras classe é conhecido como projeto de diálogo. Os diagramas de colaboração e de seqüência são utilizados nesta etapa. Como no nosso exemplo, há três funcionalidades principais para gerenciar um registro de um atleta que deve suportar registro do nome, adição, edição e deleção do atleta. Considere o diagrama de colaboração modelado na Figura 42. O primeiro diagrama de colaboração modela como o sistema gerencia o registro do atleta e consiste nas classes *AthleteRecordUI*, *MaintainAthleteRecord* e *DBAthlete*. Isto é mostrado pelo segundo diagrama de colaboração da figura. Classes adicionais são identificadas e definidas para suportar recuperação de um registro de um atleta a partir da base de dados quando as funcionalidades de edição e deleção assim necessitam. Estas classes são *FindAthleteRecordUI*, *FindAthleteRecord* e *AthleteListUI*.

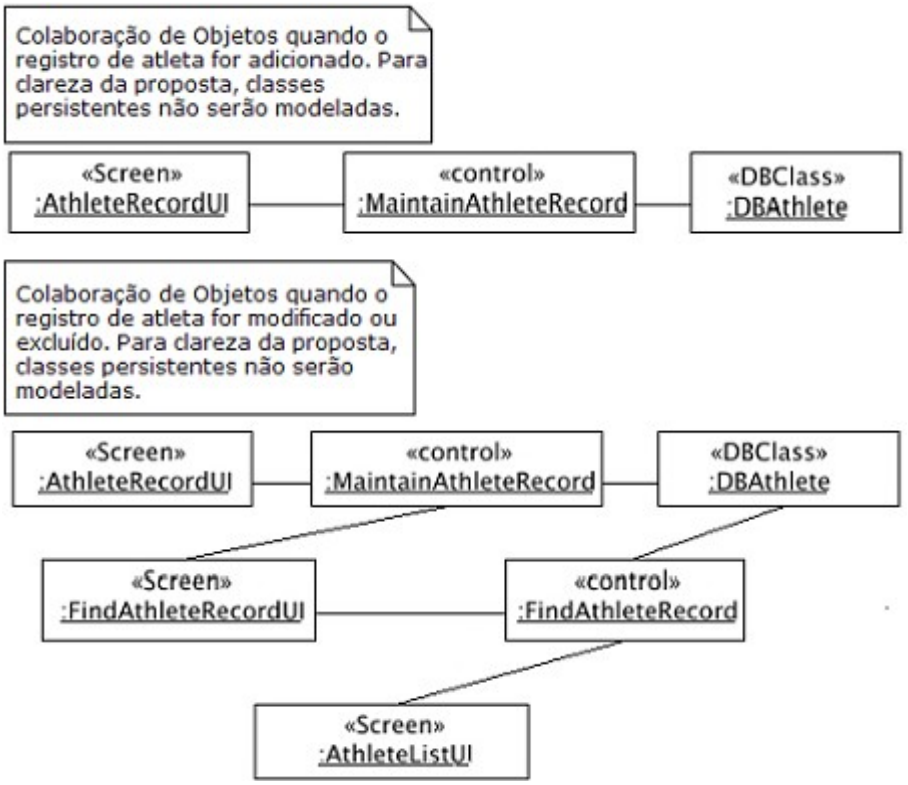

Figura 42: Diagrama de Colaboração Formal com AthleteRecordUI

Uma vez que os objetos da colaboração são identificados, precisamos modelar o comportamento destes objetos quando as funcionalidades (adição, edição e deleção do registro do atleta) são executadas. Utilizamos o Diagrama de Seqüência para modelar este comportamento. A Figura 43 mostra o Diagrama de Seqüência formal da adição de um registro de um atleta.

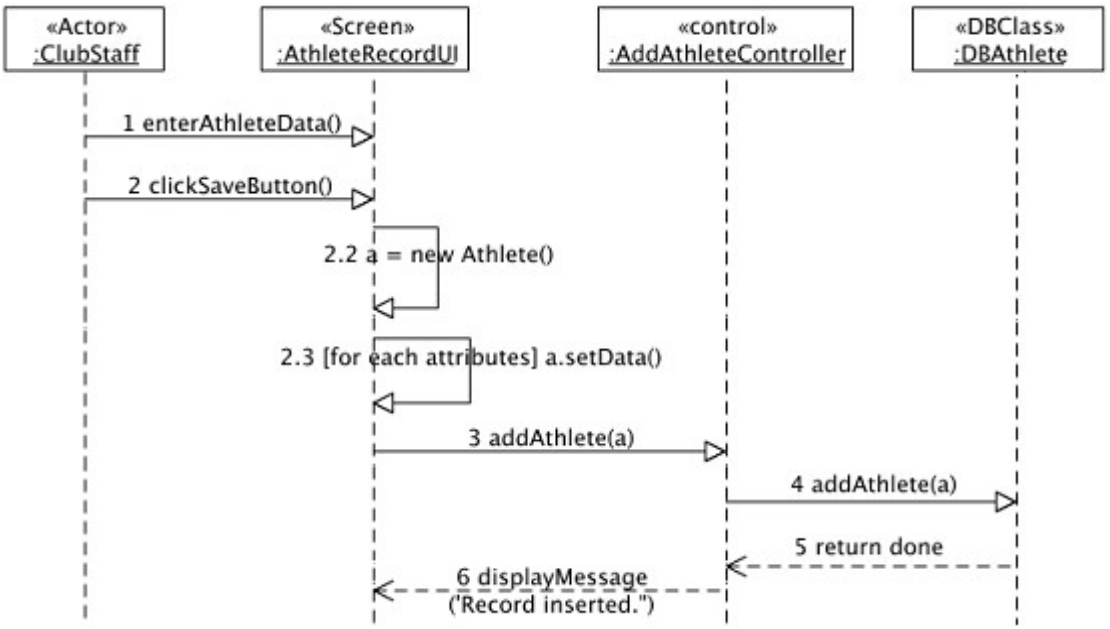

Figura 43: Diagrama de Seqüência Formal da Adição de um Registro de um Atleta

Ambos os diagramas de Seqüência para alteração e exclusão de um registro de atleta irão requerer a recuperação de um registro da base de dados. Antes de alterar ou excluir, o Diagrama de Seqüência da Figura 44 deverá ser executado.

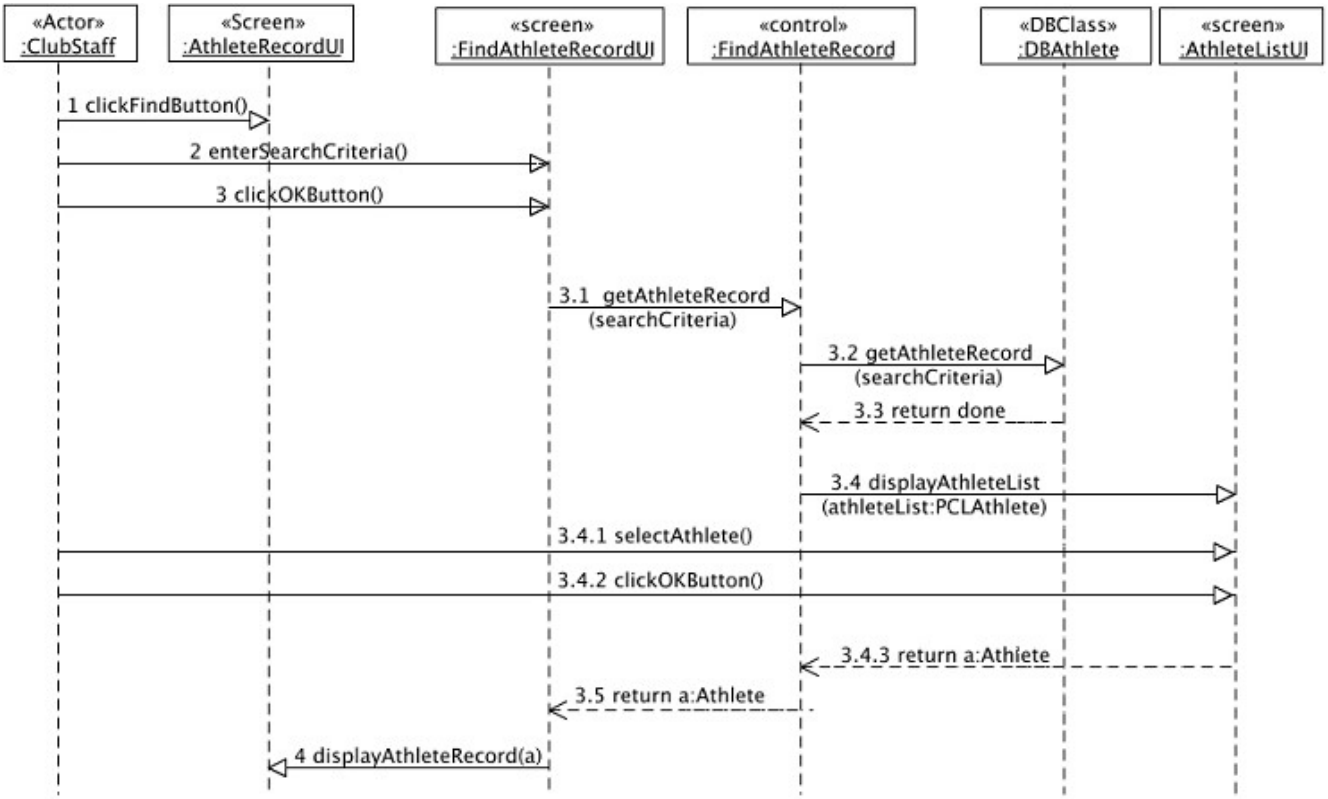

Figura 44: Diagrama de Seqüência de Recuperação de um Recorde de Atleta Modificado

O Diagrama de Seqüência modificado para editar e excluir um recorde de atleta é mostrado na Figura 45 e Figura 46 respectivamente.

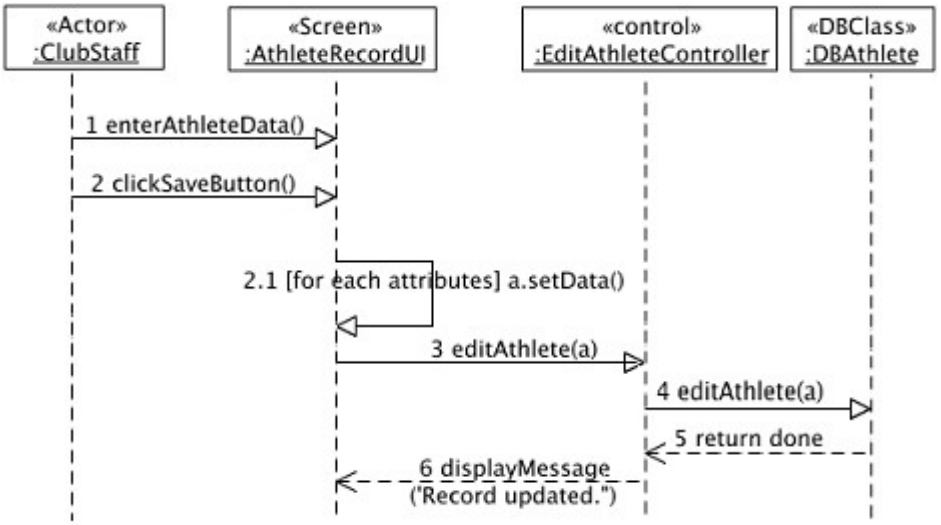

Figura 45: Diagrama de Seqüência de Alteração de um Recorde de Atleta Modificado

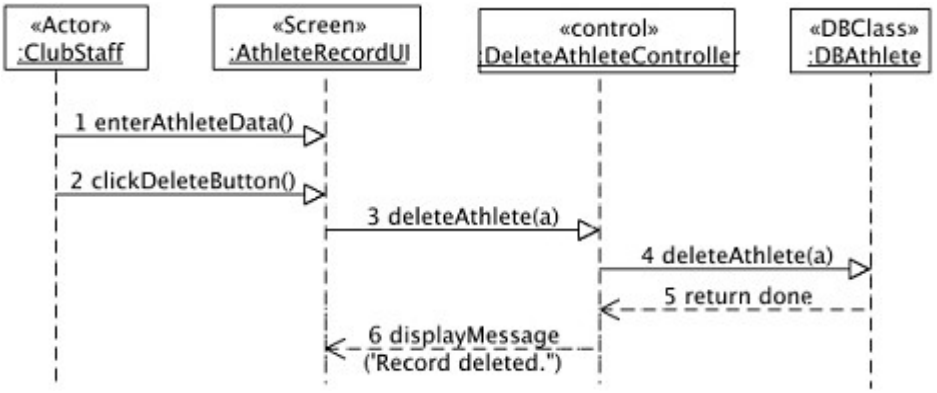

Figura 46: Diagrama de Seqüência de Exclusão de um Recorde de Atleta Modificado

PASSO 4: Para cada classe de tela, modelar seu comportamento interno.

O Diagrama de Estado é utilizado para modelar o comportamento interno da tela. Isso basicamente modela o comportamento da tela quando um usuário faz alguma coisa nos elementos da tela como pressionar o Botão OK.

Descrevendo o gráfico do Diagrama de Estado

O Gráfico do Diagrama de Estado mostra os possíveis estados que um objeto pode ter. Isso também identifica os eventos que causam a mudança de estado de um objeto. Isso mostra como o estado de um objeto muda de acordo com os eventos que são tratados pelo objeto. Figura 47 mostra a notação básica de um diagrama.

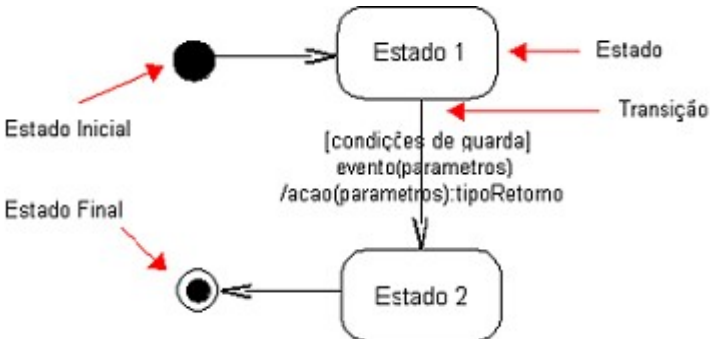

Figura 47: Notação Básica do Gráfico do Diagrama de Estado

O ***estado inicial***, representado pelo círculo sólido, significa o ponto inicial da transição. Um objeto não pode permanecer no seu estado inicial mas precisa se mover para um ***estado***

nomeado. O estado é representado como um objeto retangular com as bordas arredondas e com um nome de estado dentro do objeto. Uma ***transição*** é representada por uma linha com uma seta na ponta. Uma transição pode ter um ***texto de transição*** que mostra o evento causador da transição, qualquer condição de guarda que precise ser avaliada durante o disparo do evento, e a ação que precisa ser executada quando o evento é disparado. O evento pode ser uma operação definida na classe. O ***estado final***, como representado pelo círculo com outro círculo preenchido dentro, significa o fim do diagrama.

Similar ao outro diagrama na UML, há uma notação melhorada ou estendida que pode ser usada quando está se definindo o estado da classe. A Figura 48 mostra essas notações.

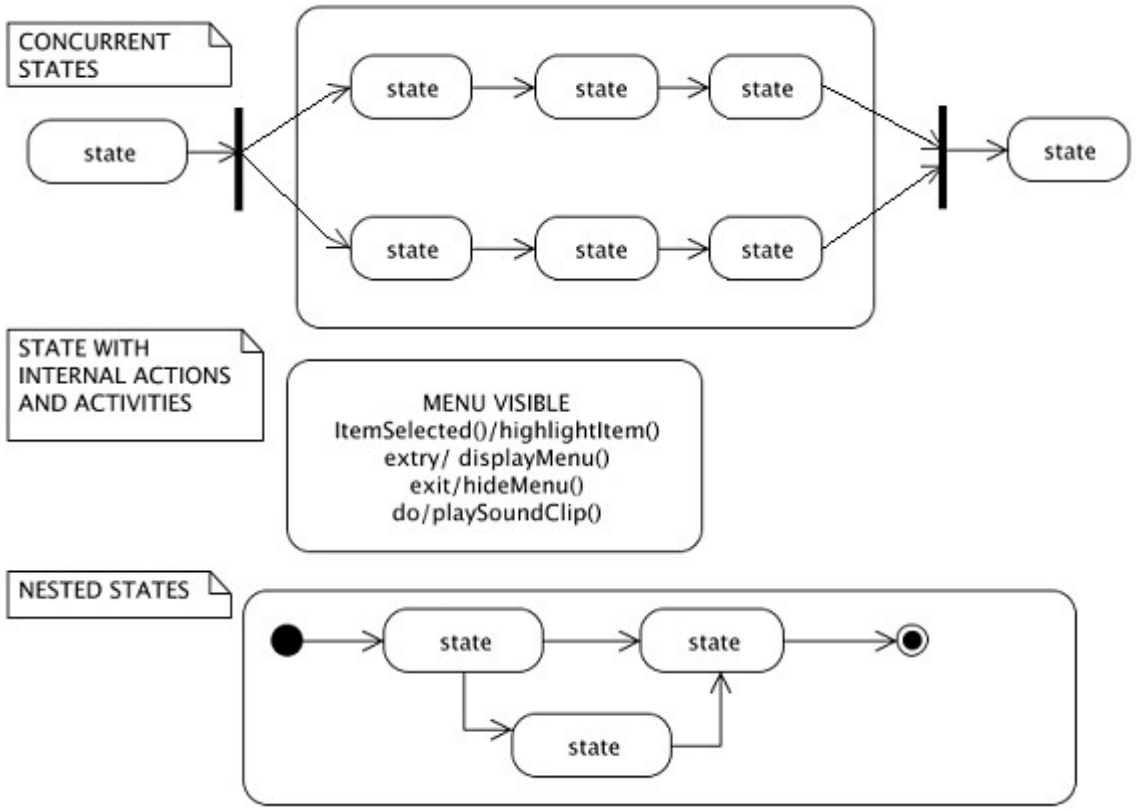

Figura 48: Notação aperfeiçoada do Diagrama de Estados

Concurrent States (Estados Concorrentes) significa que o comportamento de um objeto pode ser melhor explicado pelo produto correspondente como dois conjuntos distintos de sub-estados, cada estado da qual pode ser entrado ou saído independentemente dos sub-estados do outro conjunto. ***Nested States*** (Estados Aninhados) permitem modelar um complexo estado através da definição de diferentes níveis de detalhes. Estados podem ter ações e atividades internas. Ações internas de um estado definem qual operação pode ser executada ao entrar num estado, qual operação pode ser executada ao se sair de um estado, e qual operação poder ser executada enquanto estiver em um determinado estado.

Figura 49 Mostra o gráfico do diagrama de estado da classe *AthletRecordUI*. Quando uma tela é exibida pela primeira vez, mostrará uma tela de inicialização onde a maioria das caixas de texto estarão vazias. Alguns campos podem ter valores padrões, como **gênero**, com um valor padrão 'Masculino' e **situação** com o valor padrão 'Novo'. Esse é o estado inicial. Quando um usuário insere um valor em qualquer caixa de texto (isso dispara um evento *enterTextField*()), o estado da tela irá mudar para o valor informado (*With Entered Values*) onde este valor é refletido no elemento da tela. Quando um usuário pressiona o Botão Cancelar (que dispara o evento *clickCancelButton*()), o estado da tela irá mudar para o estado mostra a confirmação saída (*Show Exit Confirmation*). Nesse estado, uma mensagem numa caixa de diálogo será exibida questionando caso o usuário deseja realmente sair. Se a resposta do usuário for SIM (*answerYES*), sairemos da tela. Senão, retornaremos ao estado anterior.

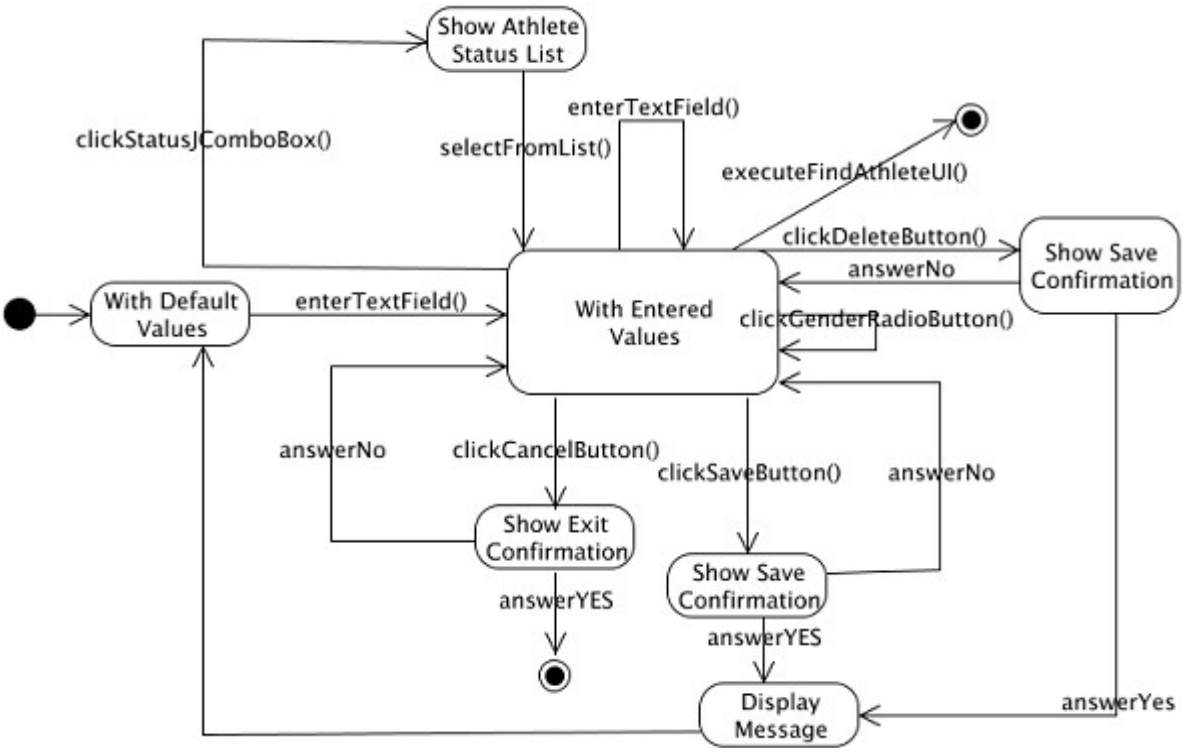

Figura 49: Diagrama de Estado da AthleteRecordUI

Figura 50 Mostra o gráfico do diagrama de estado *FindAthletRecordUI* e Figura 51 mostra o gráfico do diagrama de estado *AthleteListUI*.

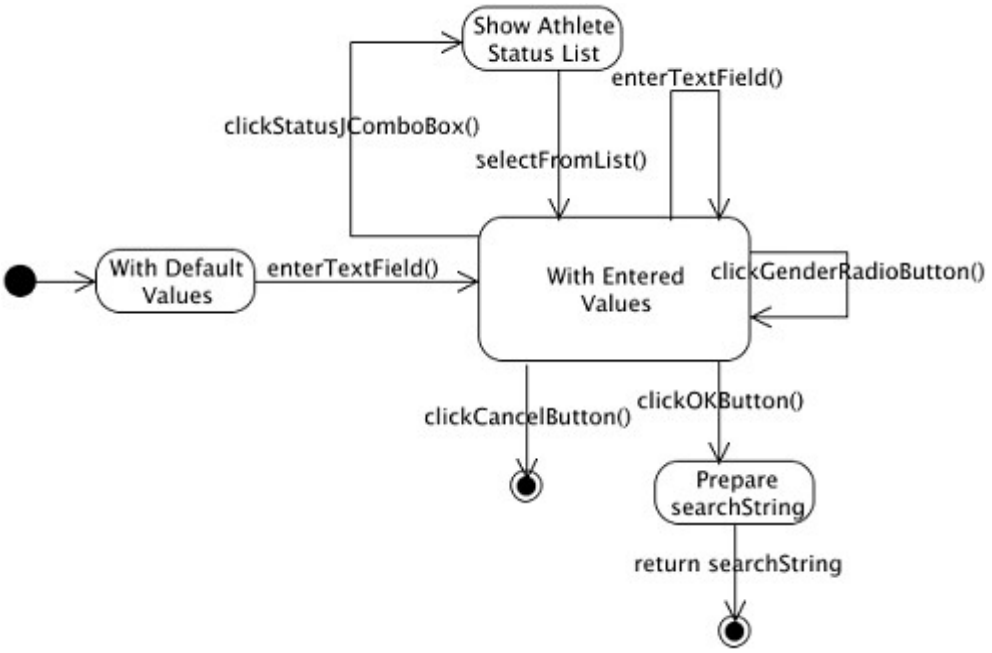

Figura 50: Diagrama de Estado de FindAthleteRecordUI

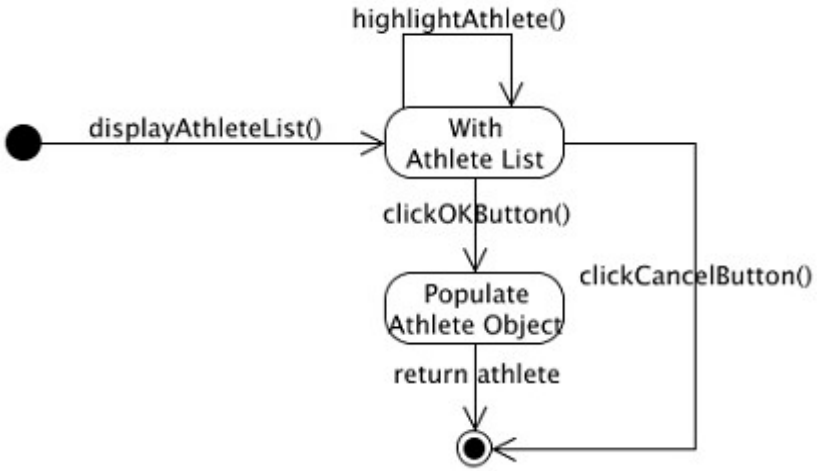

Figura 51: Diagrama de Estado de AthleteListUI

PASSO 5: Documentar as classes

A documentação é similar a documentar as classes de bancos de dados (*DBClasses*) e classes de Persistência (*Persistent Classes*). Identifique todas as operações e atributos da lista abaixo. A Figura 52 mostra todas as definições das classes utilizadas.

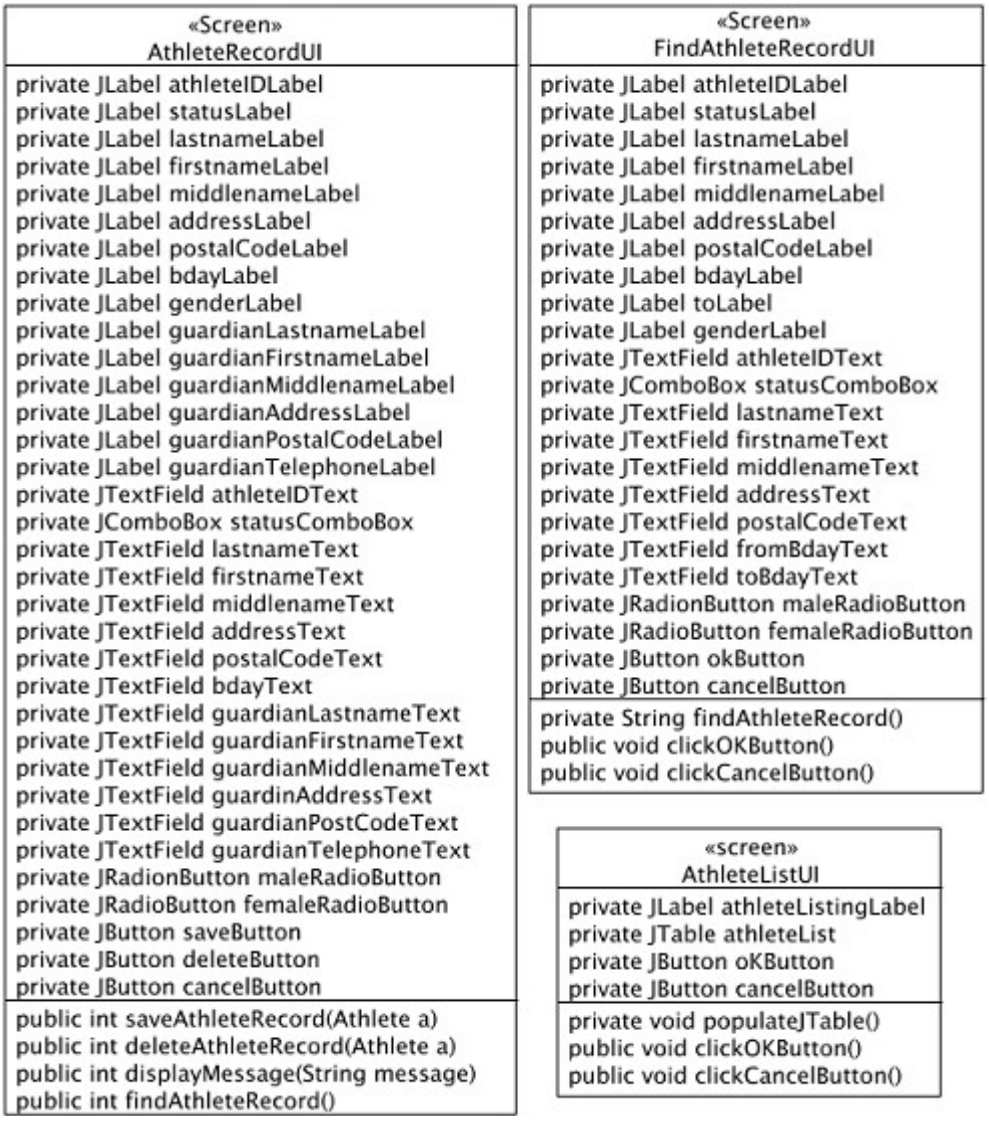

Figura 52: Refinamento das definições de Classes

Também ajudaria a listar a ação na tabela. Esta tabela é denominada de Tabela Evento-Ação. Ela é usada se o mapa de estado é complexo. Do ponto de vista do programador, esta tabela é mais fácil de usar do que um mapa de estado rotulado com ações. Ela pode auxiliar na validação do mapa de estado e para testar o código quando ele é implantado. As colunas da Tabela Evento-Ação são as seguintes:

- Estado atual do objeto que está sendo modelado. Para melhor referência, os estados são numerados.
- O evento é uma ação que pode ocorrer.
- As ações associadas com a combinação do estado e do evento.
- Próximo estado do objeto depois que o evento ocorre. Se mais de um estado variável é usado, é utilizada outra coluna.

A Figura 53 mostra um mapa de diagrama de estado da tela *AthleteRecordUI* **,** com os estados numerados e a Tabela 7 mostra o Evento-Ação desta tela.

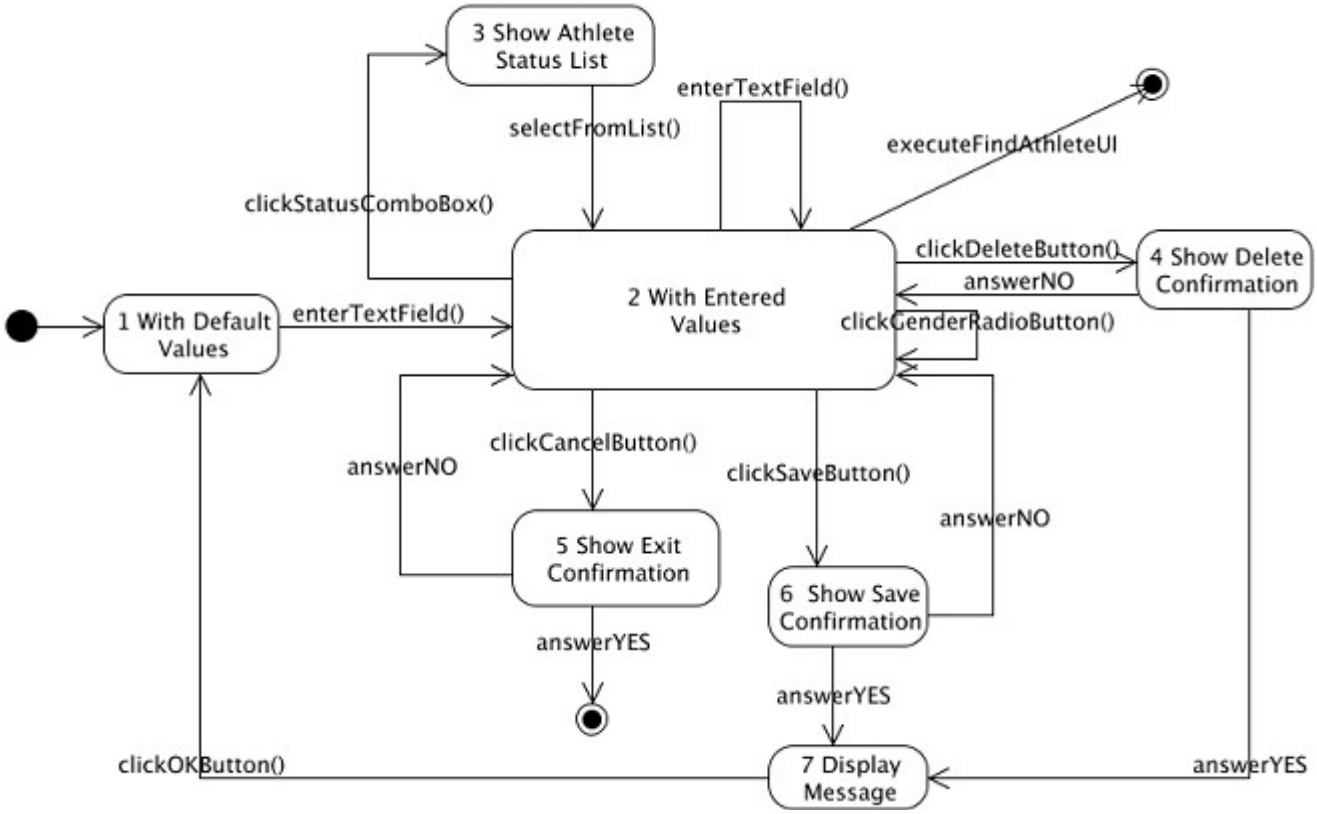

Figura 53: Mapa de Diagrama de Estado AthleteRecordUI

Estado Atual	Evento	Ação	Próximo Estado
-	Acesso a tela *AthleteRecordUI*	Exibe uma parte vazia da tela *Athlete*, exceto para os elementos da tela identificados com valores padrões.	1
1	enterTextField()	Exibe o valor marcado dentro de um campo de texto.	2
2	clickStatusComboBox()	Indica a lista do status do Atleta.	3
3	selectFromList()	Exibe o status do atleta selecionado no campo mostrado.	2
2	enterTextField()	Exibe o valor marcado dentro de um campo texto.	2
2	clickCancelButton()	Indica uma caixa de diálogo de confirmação de saída.	5
5	AnswerYES	Abandona a tela Athlete sem salvar nenhum dos dados mostrados na tela.	-
5	AnswerNO	Fecha a caixa de diálogo de confirmação de saída.	2

Tabela 7: Tabela Evento-Ação da tela AthleteRecordUI

PASSO 6: Modifique a arquitetura de software.

Os subsistemas de fronteira são substituídos pelas classes das telas ou interfaces do usuário. Se classes adicionais de controle forem definidas, adicione-as à arquitetura. Figura 54 mostra a arquitetura de software modificada.

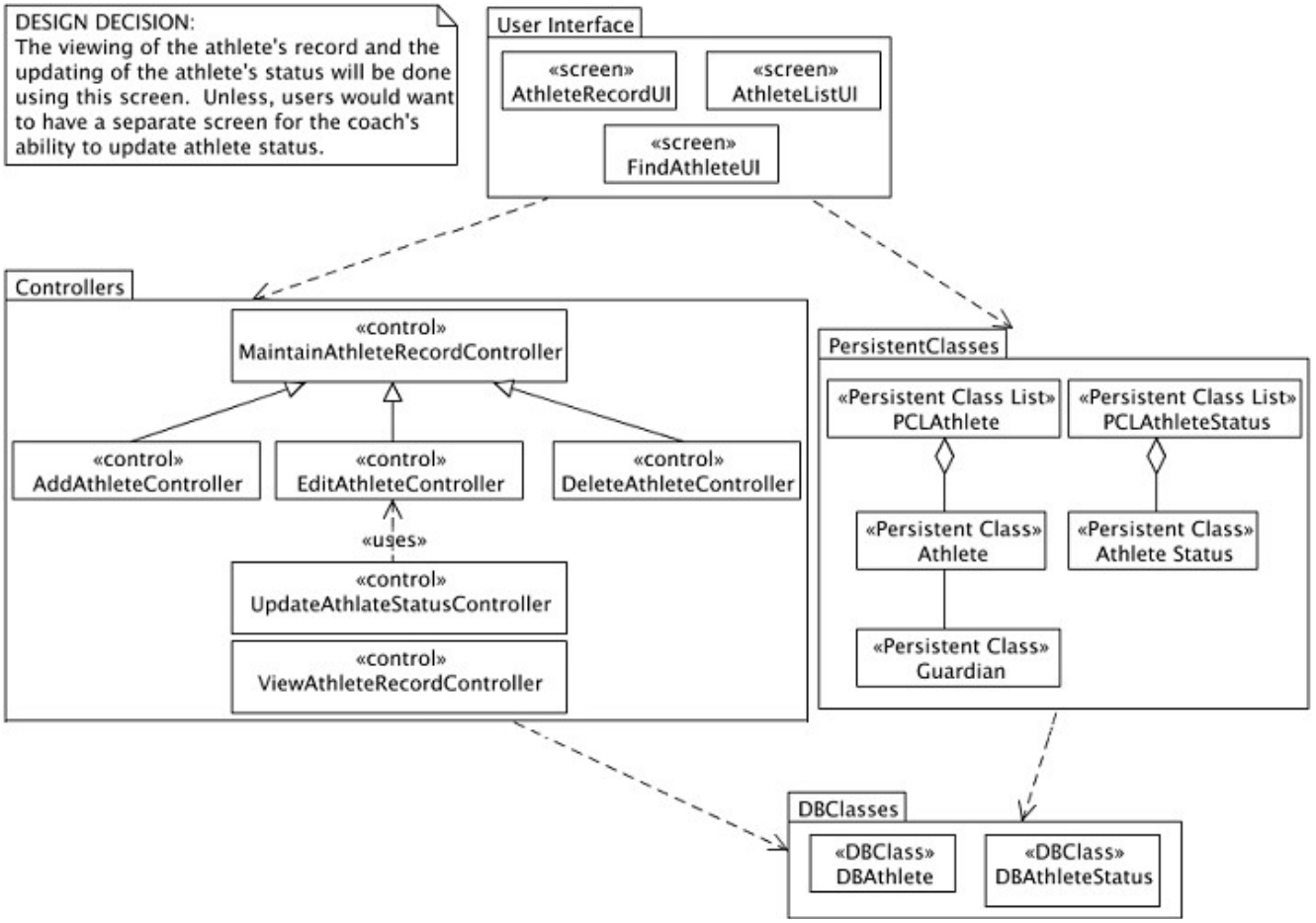

Figura 54: Arquitetura de Software Modificada da Elaboração do Projeto de Interface do Usuário

7. Projeto a nível de Componente

Define a estrutura de dados, algorítimos, características de interface e mecanismo de comunicação alocados para cada componente de software. Um ***componente*** é um bloco de construção para o desenvolvimento de software. É uma parte substituível e quase independente de um software que satisfaz uma função clara no contexto de um arquitetura bem definida.

Na engenharia de software orientada a objeto, um componente é um conjunto de classes de colaboração que satisfazem um requisito funcional em particular do software. Ele pode ser desenvolvido independentemente. Cada classe no componente deve ser definida de forma completa para incluir todos os atributos e operações relacionadas à implementação. É importante que todas as interfaces e mensagens que permitem a comunicação e colaboração das classes do componente sejam bem definidas.

7.1. Princípios Básicos do Projeto de Componentes

Quatro princípios básicos são usados para o projeto de componentes. Eles são usados para guiar o engenheiro de software no desenvolvimento de componentes que são mais flexíveis e maleáveis para alterar e reduzir a propagação dos efeitos colaterais quando uma mudança acontecer.

1. **Princípio Abre-Fecha**. Quando da definição de módulos ou componentes, eles devem ser abertos para extensão, devem ser modificáveis. O engenheiro de software deve definir um componente tal que ele possa ser estendido sem a necessidade de se modificar a estrutura e o comportamento internos do componente. Abstrações na linguagem de programação suportam esse princípio.
2. **Princípio da Subtituição *Liskov***. A classe ou subclasse derivada pode ser uma substituta para a classe base. Se uma classe é dependente da classe base, essa classe pode usar qualquer classe derivada como uma substituta para a classe base. Esse princípio reforça que qualquer classe derivada deve estar de acordo com qualquer contrato implícito entre a classe base e qualquer componente que a utilize.
3. **Princípio da Dependência**. As classes devem depender da abstração; não do concreto. Quanto mais um componente depender de outros componentes concretos, mais difícil será extendê-los.
4. **Princípio da Segregação de Interface**. Engenheiros de software são encorajados a desenvolver interfaces específicas do cliente ao invés de uma única interface genérica. Apenas operações específicas de um cliente em particular devem ser definidas na interface específica do cliente. Isso minimiza a herança de operações irrelevantes para o cliente.

7.2. Guias de Projeto no nível de Componente

Esse guia é aplicado ao projeto do componente, sua interface, suas dependências e herança.

1. Componente
    - Nomes arquiteturais do componente devem vir do domínio do problema e serem facilmente compreendidos pelos participantes, particularmente, os usuários finais. Como um exemplo, um componente chamado *Athlete* é claro para qualquer um que esteja lendo o componente.nomes específicos. Como exemplo, *PCLAthlete* é uma lista de classes persistentes para atletas.
    - Utilize estereótipos para identificar a natureza de componentes como <<table>>, <<database>> ou <<screen>>.
2. Interfaces
    - A representação canônica da interface é recomendada quando o diagrama se torna complexo.
    - Elas devem fluir do lado esquerdo do componente implementado.
    - Mostra apenas as interfaces que são relevantes para o componente sendo considerado.

3. Dependências e Herança
    - Dependências devem ser modeladas da esquerda para direita.
    - Herança deve ser modelada do nível mais baixo (subclasses ou classes derivadas) para o topo (superclasses ou classe base).
    - Interdependência de componentes são modeladas da interface para interface ao invés do componente para o componente.

7.3. Diagrama de Componentes

O Diagrama de Componentes é usado para modelar componentes de software, suas interfaces, dependências e hierarquias. Figura 55 mostra a notação desse diagrama.

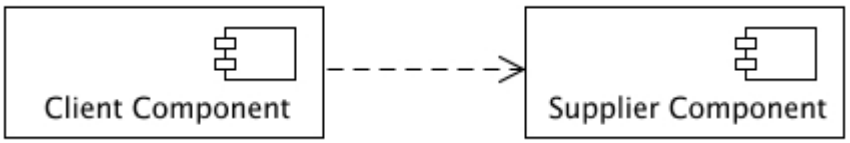

Figura 55: Notação de Diagrama de Componentes

Um **componente** é representado por uma caixa retangular com o símbolo do componente dentro. A **dependência** é descrita pelas setas pontilhadas. A dependência pode ser nomeada. O componente Cliente (*Client Component*) é dependente do componente Fornecedor (*Supplier Component*).

7.4. Desenvolvendo o Componente de Software

Todas as classes identificadas até então devem ser elaboradas, incluindo as telas e classes de modelos de dados. Nessa seção, o *MaintainAthleteRecordController* será usado como exemplo mas o modelo de componente deve ser feito para todas as classes.

O *MaintainAthleteRecordController* é uma classe abstrata que representa, em geral, o comportamento de manter um registro do atleta (*athlete*). É estendida por um tipo de manutenção – adicionando um registro, editando um registro ou deletando um registro. Figura 56 mostra a hierarquia do tipo de manutenção registro de atleta que é modelado durante a fase de requisitos de engenharia. A relação é "é-um-tipo"; *AddAthleteController* é um tipo de *MaintainAthleteRecordController*.

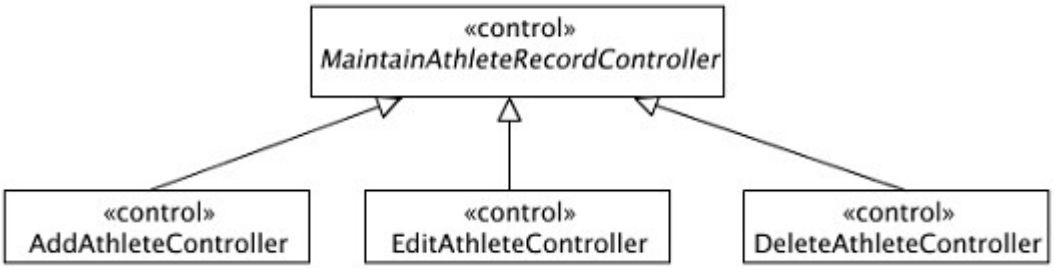

Figura 56: Hierarquia de MaintainAthleteRecordController

Desenvolver o modelo de componente envolve os seguintes passos.

PASSO 1: Refinar todas as classes.

As classes são refinadas de uma maneira que elas possam ser traduzidas em um programa. Refatoração pode ser necessária antes de continuarmos. Esse passo envolve uma série de sub-passos.

PASSO 1.1: Se necessário, redefinir os Diagramas de Seqüência e colaboração para refletir a interação das classes atuais.

PASSO 1.2: Distribuir operações e refinar a operação de assinatura da classe. Utilizar os tipos de dados e convenções de nomenclaturas da linguagem de programação sendo utilizada. Figura 58

mostra a elaboração da operação de assinatura do *AddAthleteController*. Uma breve descrição é anexada a operação.

PASSO 1.3: Refinar atributos de cada classe. Utilizar tipos de dados e convenções de nomenclaturas da linguagem de programação utilizada. Figura 58 mostra a elaboração dos atributos do *AddAthleteController*. Uma breve descrição é anexada ao atributo.

PASSO 1.4: Identificar a visibilidade dos atributos e operações. Os símbolos de visibilidade são enumerados na Tabela 8.

Símbolos de Visibilidade	***Descrição***
+ ou public	**Público (PUBLIC)**: O atributo ou a operação são acessados diretamente pela instância de uma classe.
- ou private	**Particular (PRIVATE)**: O atributo ou a operação são acessados somente por uma instância da classe que os inclui.
# ou protected	**Protegido (PROTECTED)**: O atributo ou a operação podem ser usados por qualquer instância da classe que os inclui ou por uma subclasse desta mesma classe.
~	**Pacote (PACKAGE)**: O atributo ou a operação estão acessíveis somente para serem instâncias de uma classe no mesmo pacote.

Tabela 8: Visibilidade do atributo ou da operação

Os símbolos de visibilidade são colocados antes do nome do atributo ou da operação. Veja Figura 57 como exemplo:

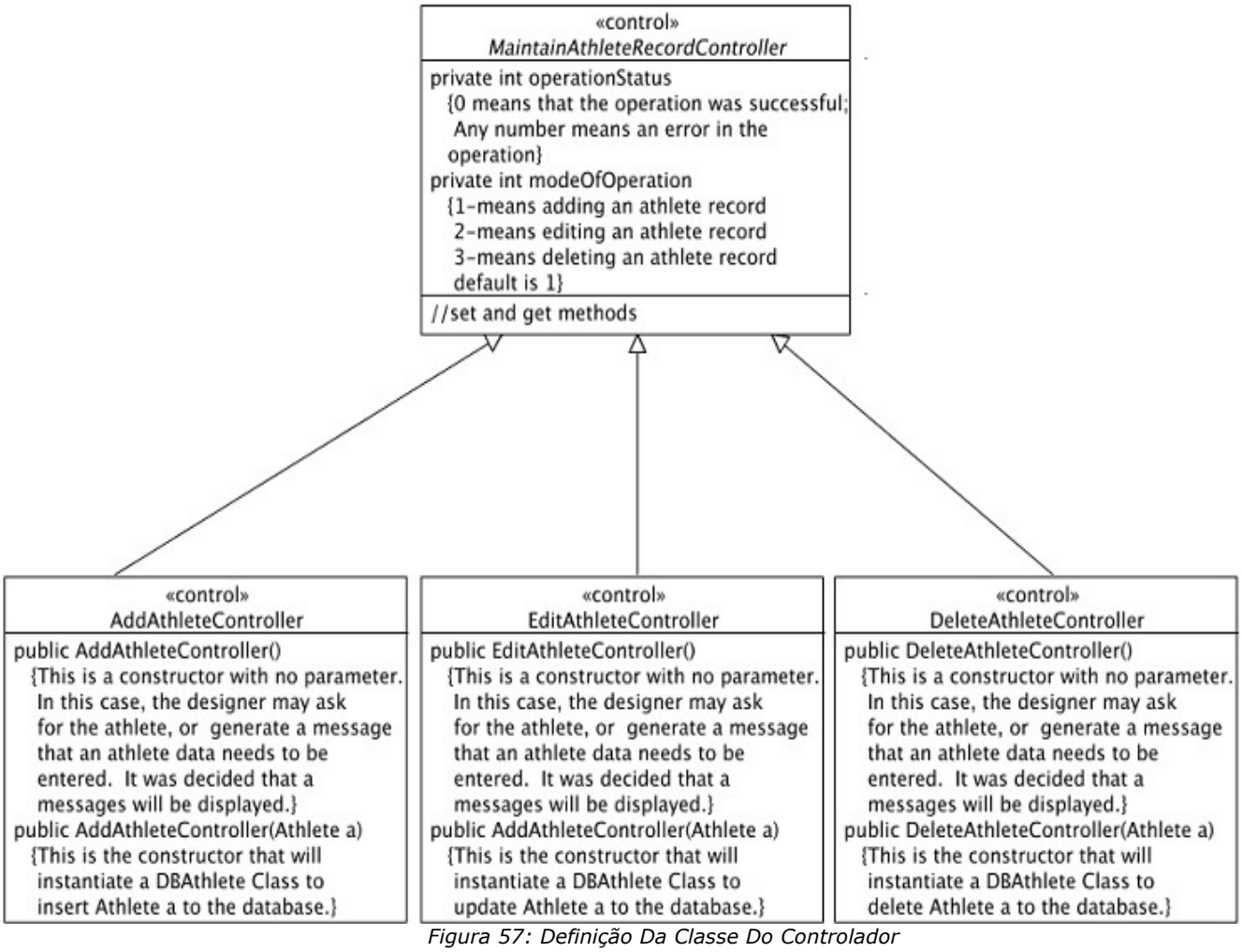

Figura 57: Definição Da Classe Do Controlador

PASSO 1.5: Documente a classe. Particularmente, identifique as pré-condições (*pre-conditions*) e

as pós-condições (*post-conditions*). As **pré-condições** são as circunstâncias que devem existir antes que a classe possa ser usada. As **pós-condições** são as circunstâncias que existe depois que a classe foi usada. Também inclua a descrição dos atributos e das operações.

PASSO 2: Se necessário, refine o empacotamento das classes.

No nosso exemplo, decidiu-se que nenhum reempacotamento é necessário.

PASSO 3. Defina os componentes do software.

Defina os componentes usando o diagrama de componentes. A Figura 58 mostra os componentes do software do **Sistema de Manutenção de Sócios do Clube**. Não inclui os componentes que serão reutilizados.

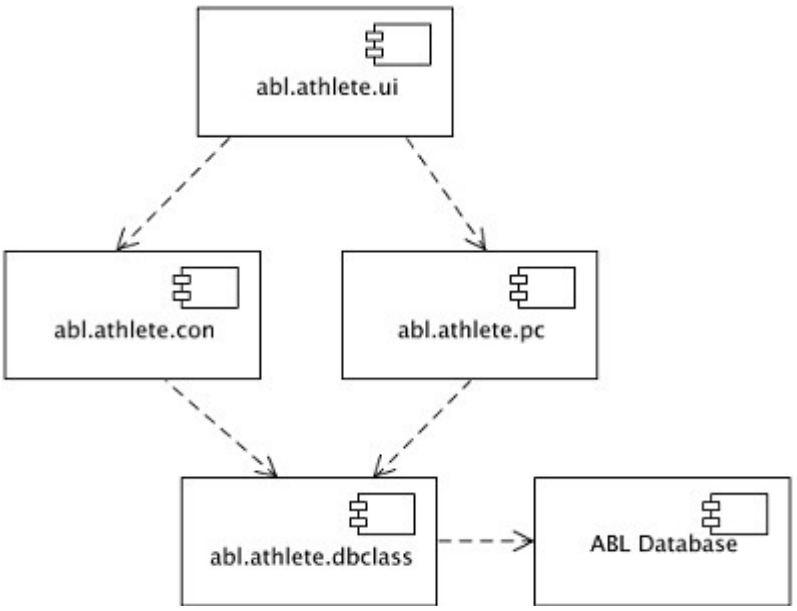

Figura 58: Diagrama de Componentes do Club Membership Maintenance

Os nomes dos pacotes segue a convenção usada pela Linguagem de Programação Java. O pacote abl.athlete manuseia o **Sistema de Manutenção de Sócios do Clube**. A lista a seguir contém seus pacotes e arquivos:

- Pacote de interface do usuário (abl.athlete.ui)
  - *AthleteRecordUI.java*, *AthleteListUI.java* e *FindAthleteRecordUI.java*
- Controladores (abl.athlete.con)
  - *MaintainAthleteRecord.java*, *AddAthlete.java*, *EditAthlete.java*, *DeleteAthlete.java* e *FindAthleteRecord.java*
- Classes de Bancos de Dados (abl.athlete.dbclass)
  - *DBAthlete.java*
- Classe de Persistência (abl.athlete.pc)
  - *Athlete.java* e *Guardian.java*

O *ABL Database* é o servidor do banco de dados que contém os dados que são necessários para as aplicações.

8. Projeto a Nível de Implementação

Cria um modelo que mostra a arquitetura física do *hardware* e *software* do sistema. Destaca o relacionamento físico entre o *software* e *hardware*. Os componentes identificados no projeto a nível de componentes são distribuídos para o *hardware*.

8.1. Notação do Diagrama de Implementação

O Diagrama de Implementação é composto de comunicações associadas. Representariam os computadores. São representados como caixas em três dimensões. As **comunicações associadas** mostram as conexões de rede. Figura 59 ilustra um Diagrama de Implementação.

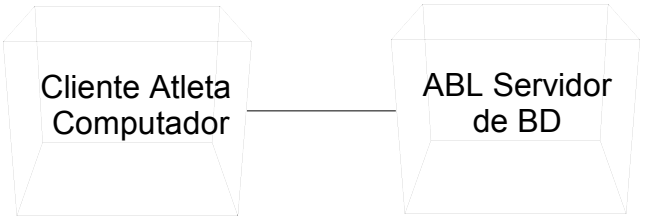

Figura 59: Notação do Diagrama de Implementação

8.2. Desenvolvimento do Modelo de Implementação

Desenvolver o diagrama de implementação é muito simples. Apenas distribua os componentes identificados no projeto a nível de componentes para reside o *node*. Figura 59 Ilustra o diagrama de Implementação para o **Sistema de Manutenção de Sócios do Clube.**

9. Itens de Validação do Modelo de Projeto

Validação é necessária para ver se o modelo de projeto:

- satisfaz aos requisitos do sistema
- é consistente com o guia de projeto
- serve como uma boa base para a implementação

9.1. Componente de Software

1. O nome do componente de software reflete claramente o seu papel dentro do sistema?
2. As dependências estão definidas?
3. Existe algum elemento do modelo dentro do componente de software visível do lado de fora?
4. Se o componente de software tem uma interface, a operação definida na interface mapeada para o elemento do modelo dentro do componente de software?

9.2. Classe

1. O nome da classe reflete claramente seu papel dentro do sistema?
2. O significado da classe é uma única bem definida abstração?
3. As operações e atributos dentro da classe são funcionalmente acoplados?
4. Existem quaisquer atributos, operações ou associações que necessitam ser generalizadas?
5. Todos os requisitos específicos sobre a classe estão endereçados?
6. A classe exibe o comportamento requerido que o sistema necessita?

9.3. Operação

1. Podemos entender a operação?
2. A operação provê o comportamento que a classe necessita?
3. A assinatura da operação correta?
4. Os parâmetros definidos corretamente?
5. As especificações da implementação da operação estão corretas?
6. A assinatura da operação está em conformidade com os padrões da linguagem de programação alvo?
7. A operação é necessária ou é usada pela classe?

9.4. Atributo

1. O significado do atributo é algo conceitual e único?
2. O nome do atributo é descritivo?
3. O atributo é necessário ou usado pela classe?

10. Mapeando os Produtos do Projeto para a Matriz de Rastreabilidade de Requisitos

Uma vez finalizados os componentes de software, necessitamos ligá-los à MRR do projeto. Isso assegura que o modelo está relacionado a um requisito. Também auxilia o engenheiro de software no rastreamento do progresso do desenvolvimento. A Tabela 9 mostra os elementos recomendados da MRR que devem ser adicionados à matriz.

Componentes de Projeto da MRR	***Descrição***
ID do Componente de Software	O nome do pacote do componente de software que pode ser independentemente implementado, como *abl.athlete* (componente de software que trata das inscrições dos sócios do clube)
Classe	O nome da classe que é implementada a qual é parte do componente de software, como *abl.athlete.AthleteRecordUI* (uma classe JAVA responsável pela interface com o ator no processamento do registro do atleta).
Documentação da Classe	O Nome do Documento da classe. É um arquivo que contém a especificação da classe.
Estados	O estado da classe. Usuários podem definir um conjunto de estados da classe, tal como: • PCA – projeto da classe em andamento • CEA – codificação em andamento • TEA – teste em andamento • FEITO

Tabela 9: Elementos do Modelo de Projeto da MRR

11. Métricas de Projeto

Na engenharia de software orientado a objetos a classe é a unidade fundamental. Medidas e métricas para uma classe individual, hierarquia de classes e a colaboração de classes são importantes para a engenharia de software especialmente para assegurar a qualidade do projeto.

Existem vários tipos de métricas para o software orientado a objetos, porém a mais largamente utilizada é a métrica CK proposta por *Chidamber* e *Kemerer*. Consiste numa métrica de projeto baseada em seis classes que são listadas abaixo:

1. Complexidade de Métodos por Classe (WMC). Isso é computado como sendo a soma da complexidade de todos os métodos de uma classe. Assuma que existam n métodos definidos em uma classe. Calcula-se a complexidade de cada método separadamente e depois se encontra a soma. Existem muitas métricas que podem ser utilizadas, mas a mais comum é a complexidade ciclomática. Isto é discutido no capítulo de Teste de Software. O número de métodos e suas complexidades indicam:
   - A quantidade de esforço necessária para a implementação e teste da classe
   - Quanto maior a quantidade de métodos, mais complexa será a arvore de herança
   - Assim como o número de métodos cresce dentro de uma classe, esta se tornará mais complexa.
2. Profundidade da Arvore de Heranças (DIT). É definido pelo comprimento máximo desde a classe mais alta até a sub-classe mais baixa. Considere a hierarquia de classes na Figura 60, o valor de DIT é igual a 5. Com o crescimento desse número, é possível ver que o número de métodos herdados pela classe mais baixa irá aumentar. Isto pode nos levar a algum problema em potencial para predizer o comportamento da classe e no aumento da complexidade de projeto. Em contrapartida, um grande número implica que mais métodos serão reutilizados.
3. Número de filhas (NOC). As filhas de uma classe são as classes imediatamente subordinadas à ela. Considere a hierarquia de classes na Figura 60, o NOC para a **class4** é igual a 2. Com o aumento do número de filhas, o reuso também será aumentado. Contudo, um cuidado deve ser dado para que a abstração representada pela classe mais alta não seja diluído por suas filhas, que não são propriamente participantes da classe base. É claro que com o aumento do número de classes filhas, o número de testes também será maior.
4. Acoplamento entre os Objetos das Classes (CBO). É o número de colaborações que uma classe realiza com outro objeto. Com o aumento deste número, a reusabilidade desta classe diminuirá. Isto também complica futuras modificações e testes. É por esta razão que o CBO deve ser mantido ao mínimo possível.
5. Resposta de uma Classe (RFC). É o número de métodos que são executados em resposta a uma mensagem dada a um objeto da classe. Com o aumento deste número, aumenta também a complexidade dos testes, pois as possíveis seqüências de testes serão maiores.
6. Falta de Coesão nos Métodos (LCOM). É o número de métodos que acessa um atributo na classe. Se este número for alto, os métodos são acoplados juntos neste atributo. Isto aumenta a complexidade do projeto da classe. É indicado manter o LCOM no mínimo possível.

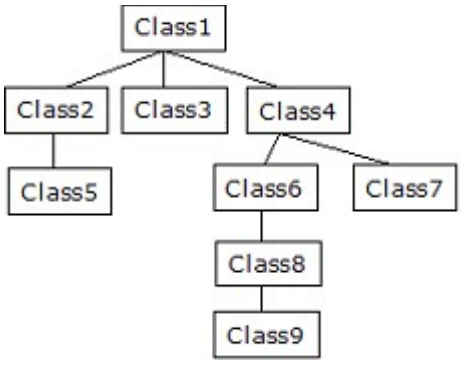

Figura 60 Amostra de Hierarquia de Classes.

12. Exercícios

12.1. Criando o modelo de Projeto de Dados

1. Criar o Modelo de Projeto de Dados do Sistema de Informações de Técnicos
    - Criar a visão estática das classes persistentes
    - Modelar o comportamento dinâmico das classes persistentes o que inclui inicialização, criação, leitura, atualização e remoção
2. Criar o Modelo de Projeto de Dados do **Sistema de Manutenção de Sócios do Clube**
    - Criar a visão estática das classes persistentes
    - Modelar o comportamento dinâmico das classes persistentes o que inclui inicialização, criação, leitura, atualização e remoção

12.2. Criar o Projeto de Interface

1. Projetar os *layouts* dos Relatórios dos itens abaixo e utilizar os padrões como definidos neste capítulo
    - Lista dos sócios por Status
    - Lista dos sócios que estarão saindo ou se graduando no próximo ano
    - Lista dos sócios que não pagaram a taxa de nova matrícula
2. Remodelar o Formulário da Aplicação para acomodar as informações adicionais dos atletas que os proprietários dos clubes queiram. Utilizar os padrões como definidos neste capítulo
3. Refinar o projeto de telas da tela Atleta como mostrado na Figura 38 para incluir informações adicionais necessárias para os proprietários de clubes
    - Criar um novo protótipo de tela
    - Definir a tela como uma classe
    - Modelar o comportamento da tela
4. Criar o projeto de telas e caixas de diálogo do Sistema de Informação de Técnicos
    - Criar o protótipo de tela
    - Definir a tela como uma classe
    - Modelar o comportamento da tela
5. Criar o projeto de telas e caixas de diálogo do **Sistema de Manutenção de Sócios do Clube**.
    - Criar o protótipo de tela
    - Definir a tela como uma classe
    - Modelar o comportamento da tela

12.3. Criando o Projeto de Controle

1. Refinar as classes de controle do Sistema de Informações de Técnicos
    - Refinar e redefinir as classes de controle
    - Modelar o comportamento das classes de controle com as telas e as classes persistentes
    - Modelar os componentes utilizando o diagrama de componentes
2. Refinar as classes de controle do **Sistema de Manutenção de Sócios do Clube**

- Refinar e redefinir as classes de controle
- Modelar o comportamento das classes de controle com as telas e as classes persistentes
- Modelar os componentes utilizando o diagrama de componentes

3. Definir o diagrama de componentes de software
   - Refinar a arquitetura de software reempacotando as classes
   - Definir o diagrama de componentes de cada pacote na arquitetura de software

12.4. Atribuições do Projeto

O objetivo das atribuições do projeto é reforçar o conhecimento e as habilidades adquiridas neste capítulo. Particularmente, são:

1. Traduzir o Modelo de Análise num Modelo de Projeto
2. Desenvolver o Modelo de Projeto de Dados
3. Desenvolver o Modelo de Projeto de Interface
4. Desenvolver o Projeto de Componentes de Software
5. Desenvolver a Arquitetura de Software

MAIORES PRODUTOS DE TRABALHO A SEREM ELABORADOS:

1. Arquitetura de Software
2. Modelo de Projeto de Dados
3. Modelo de Projeto de Interfaces
4. Modelo de Classes de Controle
5. Modelo de Projeto de Componentes de Software (IMPORTANTE!)
6. Matriz de Rastreabilidade de Requisitos
7. Lista de Ações

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.