Módulo 4

Engenharia de Software

Lição 2

Engenharia de Software Orientada a Objetos

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

Engenharia de Software Orientada a Objetos *é o uso de tecnologias de objetos na construção de software. Tecnologia de objetos é um conjunto de princípios que direciona a construção de software usando método orientado a objetos.*

Este abrange toda uma estrutura de atividades incluindo análise, desenvolvimento, teste, escolha de metodologias, linguagens de programação, ferramentas, bancos de dados e aplicações para a engenharia de software.

Nesta lição veremos o básico para o entendimento da orientação a objetos através de uma explanação sobre os conceitos fundamentais tais como objetos e classes, abstração, encapsulamento, modularidade e hierarquia. Faremos também uma apresentação geral dos modelos de processos orientados a objetos que seguem uma montagem baseada em componentes. Faremos também uma introdução sobre análise orientada a objetos e atividades de desenvolvimento, listando algumas metodologias, e sobre produtos de trabalho. Finalmente, faremos uma discussão sobre a Linguagem de Modelagem Unificada (UML - *Unified Modeling Language*) e a atividade de modelagem.

Ao final desta lição, o estudante será capaz de:

- Entender os conceitos de Orientação a Objetos
- Entender os processos gerais de modelagem orientada a objeto
- Ver de uma forma geral as atividades de análise e modelagem orientada a objeto
- Entender a UML (*Unified Modeling Language* - Linguagem de Modelagem Unificada) e entender a atividade de modelagem

2. Revisão dos Conceitos de Orientação a Objetos

Objetos são representações de entidades que podem ser físicas (tal como um clube formado de sócios e atletas), conceituais (time da seleção) ou abstratas (lista encadeada). Isto permite que os engenheiros de software representem os objetos do mundo real no desenvolvimento de software. Tecnicamente falando, eles são definidos como algo que representa um objeto do mundo real que possui limite bem definido e identidade que encapsula estado e comportamento.

Atributos e relacionamentos de um objeto definem o seu **estado**. Esta é uma das condições possíveis para que um objeto exista, e que normalmente, com o passar do tempo, modificam-se. Em um sistema, os valores armazenados nos atributos e as ligações do objeto com outros objetos definem seu estado. Operações, métodos e máquinas de estado, por outro lado, definem os comportamentos. Estes determinam como um objeto age e reage aos pedidos de mensagem de outros objetos. É importante que cada objeto seja identificado unicamente no sistema mesmo se eles tiverem os mesmos valores de atributos e comportamentos. A Figura 1 mostra um exemplo de objetos com os estados e comportamentos que indica um estudo de caso existente, a Liga *Ang Bulilit*.

Três objetos estão representados - 2 atletas e 1 técnico. Nesta figura, eles são ilustrados como círculos onde atributos são encontrados no círculo interno cercado por métodos. Objetos são unicamente identificados através dos Identificadores tal como o caso dos dois atletas, *Joel* e *Arjay*. Note que os atributos estão envolvidos por métodos. Isto sugere que apenas o objeto pode modificar o valor dos seus atributos. A modificação dos valores dos atributos pode ser gerada através de uma requisição, chamada de **mensagem**, por outro objeto. Na figura, o técnico (*JP*) indica um atleta (*Joel*) para a equipe através da execução do método *assignToSquad()*. Este método envia uma mensagem de requisição, *updateSquad("Treinando")*, para o atleta (*Joel*) atualizando o valor do seu atributo *squad*.

Uma ***classe*** é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, relacionamentos, métodos, operações e semântica. Esta é uma abstração que se concentra nas características relevantes de todos os objetos enquanto descartamos outras características. Objetos são instâncias de classes. Neste exemplo, *Joel* e *Arjay* são instâncias da classe atletas, enquanto *JP* é uma instância da classe técnico.

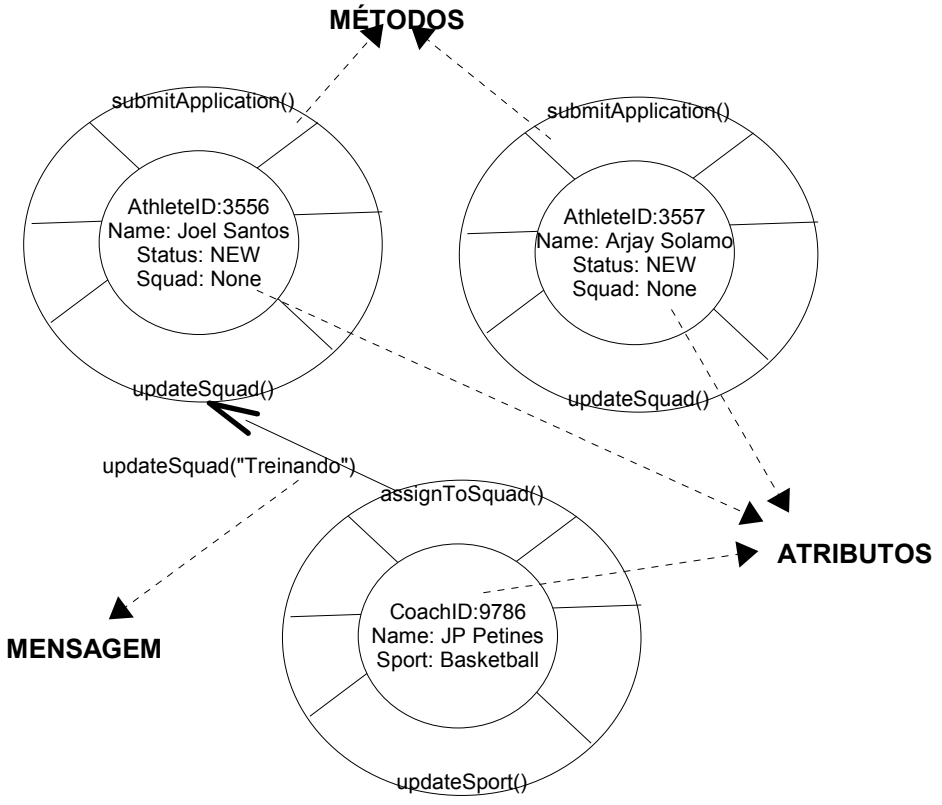


Figura 1: Exemplo de Aplicação de Objetos do Clube de Sócios e Atletas

Quatro princípios básicos estão genericamente associados com orientação a objetos os quais são abstração, encapsulamento, modularidade e hierarquia. Estes conceitos estão inter-relacionados e cooperam entre si.

Abstração

Abstração está definida como uma características essencial de uma entidade que distingue esta de todas as outras espécies de entidades[1]. Este é um tipo de representação que inclui apenas as coisas que são importantes ou interessantes de um ponto de vista particular. Este é um domínio e dependente da perspectiva, ou seja, o que é importante em um contexto e que pode não necessariamente ser em outro. Isto nos permite gerenciar a complexidade do sistema nos concentrando apenas nas características que são essenciais ou importantes para o sistema, e ignoramos ou descartamos as características que não são neste momento. Objetos são representados através dessas características que são consideradas relevantes para o propósito em questão, e são desconsideradas as características que não são.

Exemplos de abstração incluídos neste estudo de caso são:

- Um candidato apresentar-se no clube da associação para entrar na equipe.
- A equipe do clube agenda um evento para treinamentos.
- O técnico indica um atleta para a seleção.
- A seleção pode estar treinando ou competindo.
- Times são formados a partir de uma seleção.

Encapsulamento

Encapsulamento é também conhecido como **ocultação de informação**. São características embutidas dentro de uma caixa-preta abstrata que oculta a implementação destas características atrás de uma interface. Isto permite que outros objetos interajam com ele sem ter conhecimento de como a implementação realiza a interface. Esta interação é realizada através das **mensagens de interface** do objeto. Esta interface é um conjunto pré-definido de operações utilizadas por outros objetos que se comunicam com ele. Isto garante que os dados e os atributos do objetos são acessíveis através das operações do objeto. Nenhum outro objeto pode acessar diretamente estes atributos e alterar seus valores.

Considere a interação entre os objetos na Figura 1 onde o treinador (*JP*) associa um atleta (*Joel*) a um time. *updateSquad()* é uma mensagem da interface que altera o valor do atributo *time* do atleta (*Joel*). Observe que a alteração somente ocorrerá quando o treinador (*JP*) executa *assignToSquad()* que dispara uma requisição a *updateSquad("Treinamento")* do atributo do time de (*Joel*). O treinador (*JP*) não precisa saber como o atleta (*Joel*) atualiza seu time mas ele assegura-se que o método é executado.

O encapsulamento reduz o efeito dominó que as mudanças podem causar nos programas, onde a alteração na implementação de um objeto pode causar alterações em outros objetos e assim por diante. Com o encapsulamento, a implementação de um objeto pode ser alterada sem mudar a implementação de outros objetos uma vez que a interface permanece inalterada. Logo, o encapsulamento oferece dois tipos de proteção de objetos: proteção contra corrupção do seu estado interno e proteção contra alteração de código quando a implementação de outro objeto é alterada.

Modularidade

Modularidade é uma decomposição física e lógica de coisas grandes e complexas em componentes pequenos e fáceis de gerenciar. Estes componentes podem ser desenvolvidos independentemente bem como suas interações serem bem compreendidas. O conceito de pacotes, subsistemas e componentes em orientação a objetos comporta modularidade. Eles serão melhor explicados em seções à frente nesta lição.

A modularidade, como abstração, é outro caminho de gerenciar a complexidade. Isto por que ela

1 Object-oriented Analysis and Design using the UML, Student's Manual, (Cupertino, CA: Rational software Corporation, 2000), p. 2-15

quebra algo que é grande é complexo em componentes pequenos e gerenciáveis, isto facilita a tarefa do engenheiro de software de gerenciar e desenvolver software através da gerência e desenvolvimento destes pequenos componentes para, então, iterativamente integrá-los.

Por exemplo, o caso de estudo *"Liga Ang Bulilit"* pode ser dividido em subsistemas menores como mostra a Figura 2.

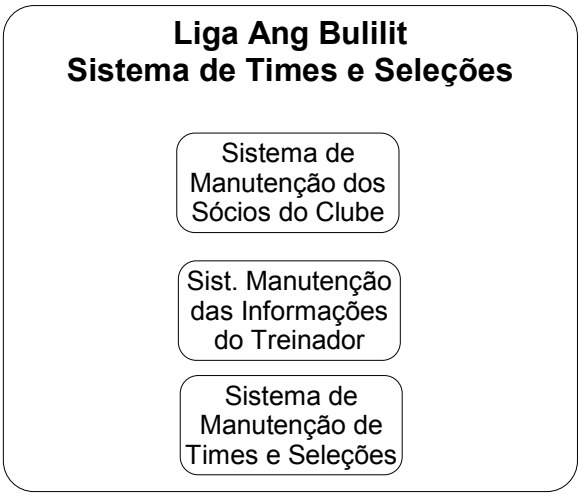


Figura 2: Subsistemas da Liga Ang Bulilit

Generalização é uma forma de associação onde uma classe compartilha a estrutura e/ou o comportamento com uma ou mais classes. Ela define a hierarquia das abstrações nas quais a subclasse herda de uma ou mais superclasses. É um tipo de relacionamento.

Herança é o mecanismo pelo qual elementos mais específicos incorporam a estrutura e o comportamento de elementos mais gerais. Uma subclasse herda atributos, operações e relacionamentos de uma superclasse. A Figura 3 mostra a superclasse **Seleção** que será herdada pelas subclasses **Treinamento** e **Competição**.

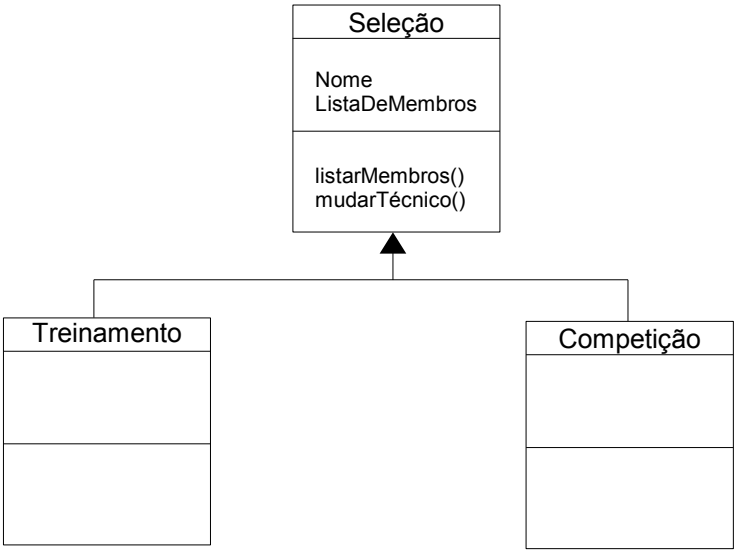


Figura 3: Hierarquia de Esquadras Elaboradas

Polimorfismo é a habilidade de esconder diversas implementações diferentes debaixo da mesma interface. Ele possibilita que a mesma mensagem seja tratada por diferentes objetos. Considere as classes definidas na Figura 5 que serão usadas para discutirmos o polimorfismo. A superclasse **Pessoa** possui duas subclasses **Estudante** e **Empregado**.

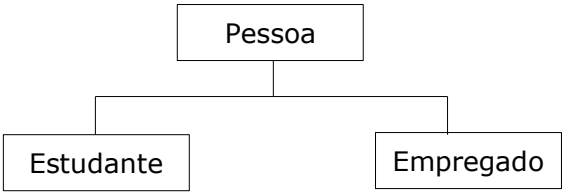


Figura 4: Exemplo de Polimorfismo

Em Java, considere a classe abaixo para implementar esta estrutura.

```
public class Pessoa {
 String nome;
 public String obterNome(){
  System.out.println("Nome da Pessoa:" + nome);
  return nome;
 }
}
public class Estudante extends Pessoa {
 public String obterNome(){
  System.out.println("Nome do Estudante:" + nome);
  return nome;
 }
}
public class Empregado extends Pessoa {
 public String obterNome() {
  System.out.println("Nome do Empregado:" + nome);
  return nome;
 }
}
```

Observe que tanto **Estudante** como **Empregado** possuem implementações diferentes do método obterNome(). Considere no seguinte método principal onde o atributo **ref** é uma referência para a classe **Pessoa**. A primeira vez que o método ref.obterNome() for chamado, executará a versão de **Estudante** já que a referência está apontada a este objeto. A segunda vez que o método ref.obterNome() for chamado, executará o método de Empregado pois agora a referência aponta para este objeto.

```
public static main(String[] args) {
  Pessoa ref;
  Estudante objetoEstudante = new Estudante();
  objetoEstudante.nome = "aluno";
  Empregado objetoEmpregado = new Empregado();
  objetoEmpregado.nome = "colaborador";
  ref = objetoEstudante; // Referência de Pessoa aponta ao objetoEstudante
  String temp = ref.obterNome(); // o método obterNome da classe
                                 // Estudante é chamado
  System.out.println(temp);
  ref = objetoEmpregado; // Referência de Pessoa aponta ao objetoEmpregado
  temp = ref.obterNome(); // o método obterNome da classe Empregado é chamado
  System.out.println(temp);
}
```

Agregação é um tipo especial de relação entre duas classes. É um modelo de relacionamento parte-todo entre um agregado (todo) e suas partes. A Figura 5 mostra um exemplo de agregação. Aqui, a equipe é composta de atletas.

Figura 5: Agregação de Equipe

3. Modelo de Processo Orientado a Objetos

A orientação a objeto abordada em desenvolvimento de software segue como base o Processo Modelo; move-se através de um trajeto espiral evolucionário; é naturalmente iterativo, e suporta reusabilidade; focaliza no uso de uma arquitetura centrada se identificando com a base da arquitetura do software.

A Arquitetura de Software define toda estrutura do software; os modos como a estrutura provê integridade conceitual de um sistema. Envolve tomar decisões de como o software é construído, e, normalmente, controla o desenvolvimento iterativo e incremental do sistema.

Para administrar o modo de desenvolvimento em uma arquitetura centrada, um mecanismo para organizar os produtos do trabalho é usado. É chamado **pacote**. Um pacote é um elemento modelo que pode conter outros elementos modelo. Permite-nos modularizar o desenvolvimento do software, e serve como uma unidade de gerenciamento de configuração.

Um **subsistema** é uma combinação de pacotes (pode conter outros elementos modelo), e uma classe (possui um comportamento). Permite definir os subsistemas do software. É realizado por uma ou mais interfaces que definem o comportamento do sistema.

Um **componente** é uma parte substituível e quase independente de um sistema que cumpre uma função clara no contexto de uma arquitetura bem definida. Pode ser um código fonte, código em tempo de execução ou um código executável. É uma realização física de um projeto abstrato. Os subsistemas podem ser usados para representar componentes no projeto; um modelo de um subsistema e de componente é mostrado na **figura 6**.

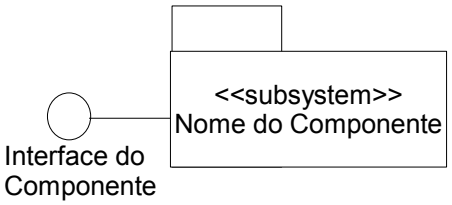


Figura 6: Subsistemas e Modelo de Componente

4. Análise e Projeto Orientado a Objeto

Duas importantes estruturas de atividades em qualquer processo de desenvolvimento de software são: a Análise do Sistema e o Projeto do Software. Esta seção discute brevemente a Análise e as fases do projeto de orientação a objeto. A abordagem será expandida mais tarde no decorrer dos capítulos.

4.1. Análise Orientada a Objeto

O objetivo principal da Análise Orientada a Objeto é desenvolver uma série de modelos que descrevam como o software de computador trabalhará para satisfazer um conjunto de requisitos definidos pelo cliente. A intenção é definir um conjunto de classes, suas relações e comportamentos que são relevantes para o sistema que está sendo estudado; desde que os requisitos do cliente influenciem a criação dos modelos, esta fase também é conhecida como engenharia de requisitos.

Dois modelos maiores são desenvolvidos durante esta fase, o modelo de requisitos e o modelo de análise. Ambos descrevem a informação, a função e o comportamento dentro do contexto do sistema, e são modelados como classes e objetos.

Existem cinco princípios básicos de análise aplicados nesta fase.

1. **Domínio de informações é modelado***.* O sistema é analisado em busca de dados importantes que são necessários dentro do sistema.
2. **Função de módulo é descrita***.* O sistema é analisado para identificar funções importantes que definem o que precisa ser feito dentro do sistema.
3. **Comportamento modelo é representado***.* Usando os dados e funções importantes, o comportamento do sistema é analisado para entender sua dinâmica natural. Isto também inclui suas interações com seu ambiente.
4. **Modelos divididos para expor maiores detalhes**. Os elementos modelo são refinados e redefinidos para mostrar os detalhes que conduzirão ao projeto do software.
5. **Primeiros modelos representam a essência do problema, embora mais tarde forneçam detalhes de implementação***.* O desenvolvimento iterativo dos modelos deve facilitar um plano de transição para a fase de projeto.

4.1.1. Metodologias de Análise Orientadas a Objeto

Existem vários métodos que são usados em Análise de Orientação a Objeto; eles estão listados brevemente abaixo.

1. ***Método de Booch*.** Este método abrange ambos "processo de desenvolvimento micro" e um "processo de desenvolvimento macro".
2. ***Método de Coad e Yourdon*.** Este é freqüentemente visto como um dos mais fáceis métodos de análise de orientação a objeto para aprender.
3. ***Método de Jacobson*.** Este método também é conhecido como Engenharia de Software de Orientação a Objeto. É diferenciado de outros por ênfase pesada no caso de uso.
4. ***Método de Rambaugh.*** Também conhecido como a Técnica de Modelagem de Objeto que cria três modelos, modelo objeto, modelo dinâmico e um modelo funcional.
5. ***Método de Wirfs-Brock.*** Não faz uma distinção clara entre análise e tarefas de projeto. É um processo contínuo que começa com a avaliação da especificação do cliente e termina com um projeto proposto.

Todos estes métodos seguem um conjunto comum de etapas para analisar um sistema. Cada uma destas etapas elaboradas indica o modo de como que os produtos do trabalho são desenvolvidos no capítulo de engenharia de requisitos.

ETAPA 1. Identifique os requisitos do cliente para o sistema orientado a objeto. Exige identificação de casos de uso ou cenários, e construção dos modelos de requisitos.

ETAPA 2. Selecionar classes e objetos usando os requisitos modelo como a diretriz.

ETAPA 3. Identificar atributos e operações para cada uma das classes.

ETAPA 4. Definir estruturas e hierarquias que organizarão as classes.

ETAPA 5. Construir uma relação de objeto modelo.

ETAPA 6. Construir um comportamento de objeto modelo.

ETAPA 7. Revisar na Análise modelo de Orientação a Objeto os requisitos e estruturas.

4.1.2. Análise Principal de Produtos de Trabalho Orientadas a objeto

Dois modelos principais são desenvolvidos durante esta fase do processo de desenvolvimento de software. Eles são os Modelo de Requisitos e Modelo de Análise. Seu desenvolvimento será elaborado no capítulo de Engenharia de Requisitos.

1. **Requisitos Modelo**. É um modelo que se empenha em descrever o sistema e seu ambiente. Consistem em um caso de uso modelo (diagramas de caso de uso e especificações), documentos e glossário adicional.
2. **Análise Modelo**. É o mais importante modelo para ser desenvolvido nesta fase porque serve como uma fundação para o desenvolvimento do software necessário para sustentar o sistema. Consiste em um objeto modelo (diagramas de classe) e comportamento modelo (diagramas de seqüência e de colaboração).

4.2. Projeto Orientado a Objeto

O objetivo da fase Projeto Orientado a Objetos é transformar o modelo de análise num modelo de análise orientado a objeto, que serve como um projeto para construção do software. Ele descreve a organização específica dos dados através da especificação e redefinição dos atributos, e também define os detalhes procedimentais das operações das classes. Como na análise, existem cinco princípios-guia de projeto que podem ser aplicados nesta fase.

1. Unidades lingüísticas modulares
2. Algumas interfaces
3. Pequenas interfaces e baixo acoplamento
4. Interface explícita
5. Encapsulamento de Informações

4.2.1. Metodologias de Projeto Orientado a Objeto

Existem vários métodos que são utilizados no projeto orientado a objeto, e eles correspondem às metodologias que foram enumeradas na metodologia de análise orientada a objeto. Elas são descritas abaixo.

1. ***Método Booch***. Similar a sua metodologia de análise, envolve um processo de "micro-desenvolvimento" e um processo de "macro-desenvolvimento".
2. ***Método Coad e Yourdon***. Refere-se não somente a aplicação, mas também a infra-estrutura da aplicação.
3. ***Método Jacobson***. Enfatiza a rastreabilidade do modelo de análise da Engenharia de Software Orientado a Objetos.
4. ***Método Rambaugh***. Inclui uma atividade de projeto que utiliza dois níveis diferentes de abstração. São eles: o projeto de sistema, que tem foco no layout dos componentes necessários para completar o produto, e o projeto de objetos, que tem foco no layout detalhado dos objetos.
5. ***Método Wirfs-Brock***. Define tarefas contínuas na qual a análise leva diretamente ao projeto.

Todas estas metodologias têm um conjunto de passos comuns que são realizados no projeto. Em cada um destes passos são elaborados produtos que são produzidos no capítulo Engenharia de Projetos.

PASSO 1. Definir os subsistemas de software através da definição dos subsistemas relacionados (projeto de entidades), subsistema de controle (projeto de controladores) e subsistemas de interação humana (projeto de fronteiras). Estes passos devem ser guiados baseados na arquitetura escolhida.

PASSO 2. Definir os Projetos de Classes e Objetos.

PASSO 3. Definir o Projeto para Troca de Mensagens.

4.2.2. Principais produtos de um Projeto Orientado a Objetos

Existem diversos produtos que são desenvolvidos durante esta fase. Eles são brevemente descritos abaixo. No geral, são chamados de **Modelo de Projeto**. O desenvolvimento destes produtos será elaborado no capítulo de Engenharia de Projeto.

1. ***Arquitetura de Software***. Refere-se à estrutura geral do software. Inclui também as formas nas qual a estrutura fornece integridade conceitual para um sistema. É modelada utilizando diagramas de pacotes.
2. ***Projeto de Dados***. Refere-se ao projeto e à organização dos dados. Inclui projeto de código para acesso aos dados e projeto de banco de dados. Utiliza diagramas de classes e diagramas de seqüência.
3. ***Projeto de Interface***. Refere-se ao projeto de interação do sistema com seu ambiente, particularmente os aspectos de interação humana. Inclui os projetos de interface visual. Os formatos de relatórios e formulários são também incluídos. Utiliza diagramas de classe e diagramas de transição de estado.
4. ***Projeto de Componentes***. Refere-se à elaboração e projeto de classes de controle. Este é o projeto mais importante por que os requisitos funcionais são representados através das classes de controle. Utilizam diagramas de classe, diagramas de atividade e diagramas de transição de estado.
5. ***Projeto de Distribuição***. Refere-se ao projeto de como o software será distribuído para uso operacional. Utiliza o diagrama de distribuição.

5. Linguagem de Modelagem Unificada (UML)

A Linguagem de Modelagem Unificada (UML) é uma linguagem padrão para especificar, visualizar, construir e documentar todos os produtos de trabalho ou artefatos de sistemas de software. Antigamente a UML possuia terminologias diferentes conforme mostrado na **Tabela 1**.

UML	*Classes*	*Associação*	*Generalização*	*Agregação*
Booch	Classe	Uso	Heranças	Conteúdo
Coad	Classe & Objeto	Exemplo de Conexão	Especificação-Geral	Parte-completa
Jacobson	Objeto	Associação de Conhecimento	Heranças	Consistir de
Odell	Tipo do Objeto	Relacionamento	Subtipo	Composição
Rambaugh	Classe	Associação	Generalização	Agregação
Shlaer/Mellor	Objeto	Relacionamento	Subtipo	n/a

Tabela 1: Diferentes terminologias da UML

Entretanto, a notação foi unificada por *Booch*, *Rambaugh* e *Jacobson*. A OMG (Object Management Group) é um grupo que mantém um conjunto de padrões que define a utilização da UML, realizando uma padronização técnica dos modelos de projeto. Atualmente, outros participantes colaboram com esta padronização e definição da UML.

UML não é um método ou metodologia. Não é indicada para um processo particular. Não é uma linguagem de programação. É basicamente um padrão de ferramentas de modelagem usado por desenvolvedores em seus trabalhos de software, como foi mostrado nos modelos.

5.1. Modelando a Atividade

O desenvolvimento do software é uma atividade complexa e é extremamente difícil realizar a tarefa necessária se todos os detalhes não estiverem em um único local. Em qualquer projeto de desenvolvimento, o objetivo é produzir produtos úteis do trabalho, o foco principal de atividades da análise do projeto está em modelos. Um modelo é um teste padrão de algo a ser criado. Ele é abstrato e visível. O software não é tangível para os usuários e por sua natureza, é abstrato. Entretanto, é construído por uma equipe que precisa de cada modelo descrito.

Um modelo é uma representação de algo no mundo real. Particularmente, são:

- Mais rápidos de se construir e representam os objetos reais.
- São usados na simulação para entender melhor o que representam.
- Podem ser modificados conforme uma tarefa ou um problema.
- Utilizado para representar melhor os detalhes dos modelos selecionados e para examinar os que foram rejeitados; é basicamente uma abstração dos objetos reais.

Um modelo útil tem uma determina quantidade de detalhe e de estruturas, mostra claramente o que é importante para uma determinada atividade. Geralmente, os desenvolvedores modelam a situação complexa dentro de um sistema de atividade humana. Devem modelar somente as partes que são importantes ao invés de refletir toda a situação. Por esta razão, necessitam ser ricos em detalhes sobre a parte tratada. Os modelos da fase de análise (modelos de análise de requisitos), devem ser exatos, completos e sem ambigüidades. Sem isto, o trabalho da equipe do desenvolvimento será muito mais difícil. Ao mesmo tempo, não deve incluir decisões prematuras sobre como o sistema será construído para atender as exigências do usuário. Caso contrário, a

equipe de desenvolvimento pode, mais tarde, achar que suas ações são controladas. A maioria dos modelos está no formato de diagramas que podem ser acompanhados por descrições textuais, lógicas, processos de especificações matemáticas ou de dados.

Analistas de sistemas e gerentes de projetos usam diagramas para criar modelos de sistemas. Os diagramas são usados amplamente para:

- entender a estrutura de objetos e seus relacionamentos.
- compartilhamento de plano com outros
- solicitação de planos e novas possibilidades
- testes de planos e construir prognósticos

Aqui se encontram as várias técnicas de modelagem que podem ser encontradas. Algumas regras gerais são listados abaixo. Modelagem técnicas deve ser:

- **Únicas**. Mostrar somente o que precisa ser mostrado.
- **Internamente compatível**. Os diagramas devem suportar ao outro.
- **Completo**. Mostrar tudo aquilo que precisa ser mostrado.
- **Representados hierarquicamente**. Dividindo o sistema em componentes menores. Em seguida, mostrar os detalhes de sub-níveis.

A UML fornece um conjunto de diagramas usados para modelar sistemas. Os diagramas são representados por quatro elementos principais.

- ***Símbolos***. Representam as figuras descritas nos diagramas. São figuras abstratas e podem ser conectados com outros símbolos (por exemplo, uma classe é representada por um retângulo).
- ***Símbolos bidimensionais***. São similares aos símbolos exceto que têm os compartimentos que podem conter outros símbolos, símbolos ou cadeias. Um símbolo não possui outros símbolos.
- ***Caminhos***. Representam uma ligação de um símbolo a outro. Normalmente representam um fluxo de dados.
- ***Cadeias***. São usadas para representar descrições, nomes, entre outros dados textuais, nos símbolos ou caminhos.

A especificação de UML fornece uma definição formal de como usar diagramas de UML. Fornece a gramática de UML (sintaxe) que inclui o significado dos elementos e as regras de como os elementos são combinados (semântica).

Um único diagrama ilustra ou documenta um determinado aspecto de um sistema. Um modelo, que sendo utilizado por outra pessoa, fornece uma visão completa do sistema em um determinado estágio e perspectiva. Por exemplo, os modelos desenvolvidos durante a fase de análise dão uma visão completa do sistema através do domínio do problema.

Capturar essencialmente o aspecto do sistema tais como funções, dados, entre outros. Consiste em diversos diagramas para a segurança de que todos os aspectos do domínio do problema estão contemplados. Os modelos, desenvolvidos durante a fase do projeto, dão uma visão completa do sistema a ser construído. É uma visão do domínio da solução provê a possibilidade de transformar os diagramas em componentes do sistema tais como, projetos de diálogo para as janelas de interação com o usuário, os elementos da base de dados para as funções de controle e deste modo continuamente.

Os modelos que são produzidos durante o desenvolvimento do projeto modificam durante o desenvolvimento do mesmo. Normalmente, são dinâmicos em determinados aspectos:

- **Nível de Abstração**. Conforme o projeto se desenvolve, o modelo usado vai para um estágio menos abstrato e mais concreto. Por exemplo, é possível começar com classes que representam o tipo dos objetos que encontraremos no sistema tal como atletas, ônibus, seleção e equipe. Com o tempo, do começo ao fim do projeto, se pode definir para as classes

os atributos e as operações. As classes também devem suportar classes adicionais tais como corretores, que representarão a plataforma de distribuição do objetivo.

- **Grau de Formalidade**. O grau de formalidade com que as operações, os atributos e as restrições são definidos com o desenvolvimento do projeto. Inicialmente, os atributos são classificados e podem ser facilmente definidos usando o português estruturado ou qualquer outra língua utilizada pela equipe de desenvolvimento. Com o tempo, chega o fim do projeto e inicia-se a fase de implementação. Atributos e operações são definidas utilizando a linguagem de programação escolhida, neste caso Java. Por exemplo, na fase de análise, uma usa-se "*Athlet*" como o nome da classe que conterá as informações dos atletas. Na fase do projeto, pode ser uma classe persistente que captura a informação sobre detalhes de um atleta e ser nomeado como classe, "*DBAthlet*".
- **Nível de detalhamento**. Enquanto o projeto progride, os diferentes modelos representam a mesma visão mas mostram um nível mais aprofundado de detalhamento. Os modelos ficam mais detalhados ao se avançar no processo de desenvolvimento. Por exemplo, o primeiro diagrama caso de uso pode mostrar somente os diagramas de **Casos de Uso** óbvios são aparentes em uma primeira visão. Na segunda visão, o diagrama de **Caso de Uso** pode ser elaborado com maiores detalhes, e **Casos de Uso** adicionais podem surgir. Após a terceira visão no projeto, pode-se incluir uma descrição mais elaborada na qual os usuários interagem com os novos **Casos de Uso** e com os relacionamentos de outros Casos de Uso.

Todas as fases de um projeto consistem em um número de visões, e esse número dependerá da complexidade do sistema que está sendo tomado. Enquanto o projeto progride, o nível do abstração, o grau de formalidade e o nível do detalhamento devem ser estabelecidos corretamente de modo que os trabalhos estejam corretamente modelados.

6. Diagramas Básicos em UML

Existem nove diagramas que são especificados na UML. Eles são brevemente discutidos nesta seção. Os seus usos serão apresentados em detalhes posteriormente assim que os produtos sejam produzidos durante as fases do processo de desenvolvimento.

6.1. Diagrama de Caso de Uso

Fornece a base da comunicação entre usuários finais e desenvolvedores no planejamento do projeto de software. Ele captura as funções na visão do usuário que podem ser pequenas ou grandes. Ele alcança um objetivo particular para o usuário. Ele tenta modelar o sistema mostrando os atores externos e suas conexões com a funcionalidade do sistema. A figura seguinte mostra um exemplo de Diagrama de Caso de Uso.

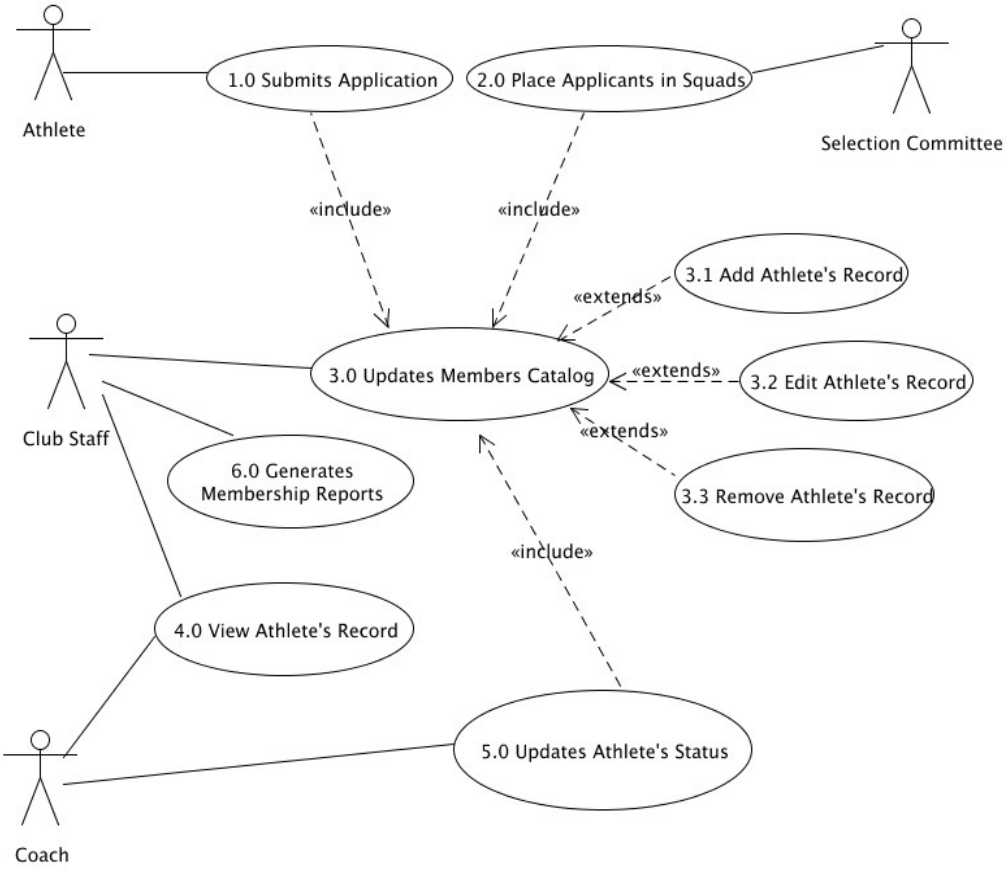


Figura 7: Exemplo de Diagrama de Caso de Uso

6.2. Diagrama de Classe

Mostra a estrutura estática do domínio das abstrações (classes) do sistema. Ele descreve os tipos de objetos no sistema e os vários tipos de relacionamentos estáticos que existem entre eles. Mostra os atributos e operações de uma classe e checa a maneira com que os objetos colaboram entre si.

Esta ferramenta de modelagem produz o diagrama mais importante do sistema, pois é usado para modelar um entendimento do domínio da aplicação (essencialmente parte do sistema de atividades humanas), e os objetos também são entendidos como partes do sistema resultante. A figura seguinte mostra um exemplo de diagrama de classe.

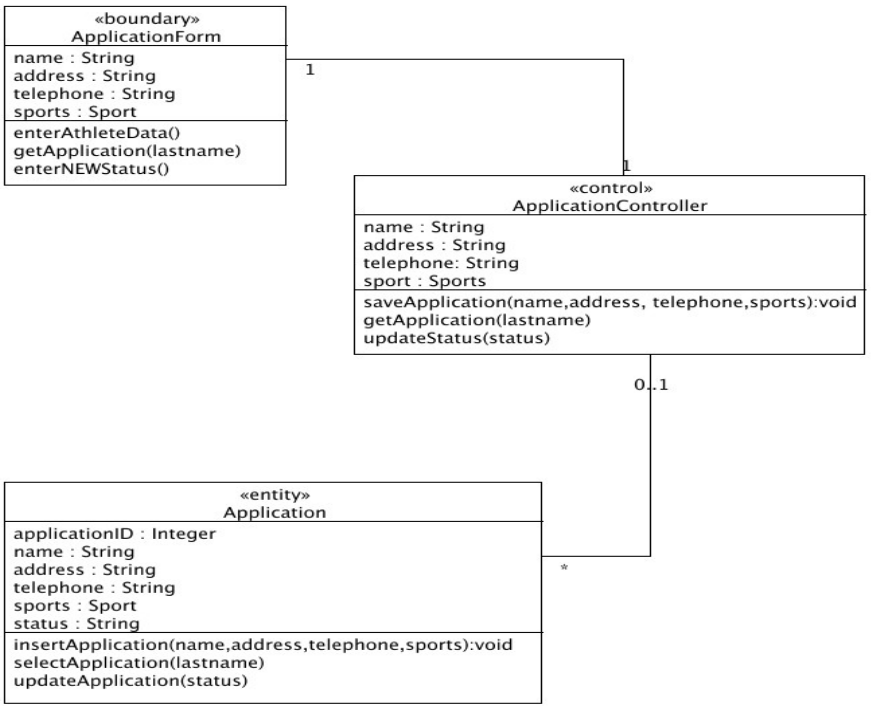


Figura 8: Exemplo de Diagrama de Classe

6.3. Diagrama de Pacote

Mostra a divisão de sistemas grandes em agrupamentos lógicos de pequenos subsistemas. Ele mostra agrupamentos de classes e dependências entre elas. Uma dependência existe entre dois elementos se mudanças na definição de um elemento pode causar mudanças em outros elementos. A figura seguinte mostra um exemplo de diagrama de pacote.

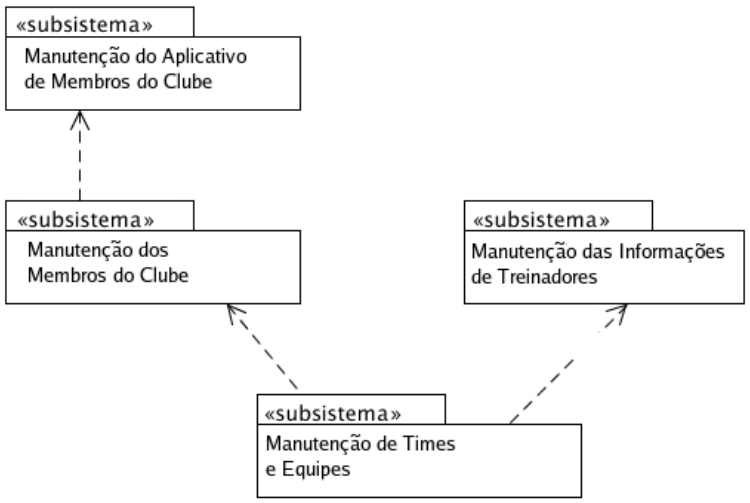


Figura 9 Exemplo de Diagrama de Pacote

6.4. Diagrama de Atividade

Mostra o fluxo seqüencial das atividades. É usado tipicamente para indicar o comportamento da operação, fluxo de eventos ou a trilha de evento do **Caso de Uso**. Complementa o diagrama de

Caso de Uso , mostrando o fluxo de trabalho do método. Ele encoraja a descoberta de processos paralelos, ajudando assim a eliminar seqüências desnecessárias no processo do método. Complementa o diagrama de classe por mostrar o fluxo de cada uma das operações (similar ao mapa de fluxo). A figura seguinte mostra um exemplo de um diagrama de atividade.

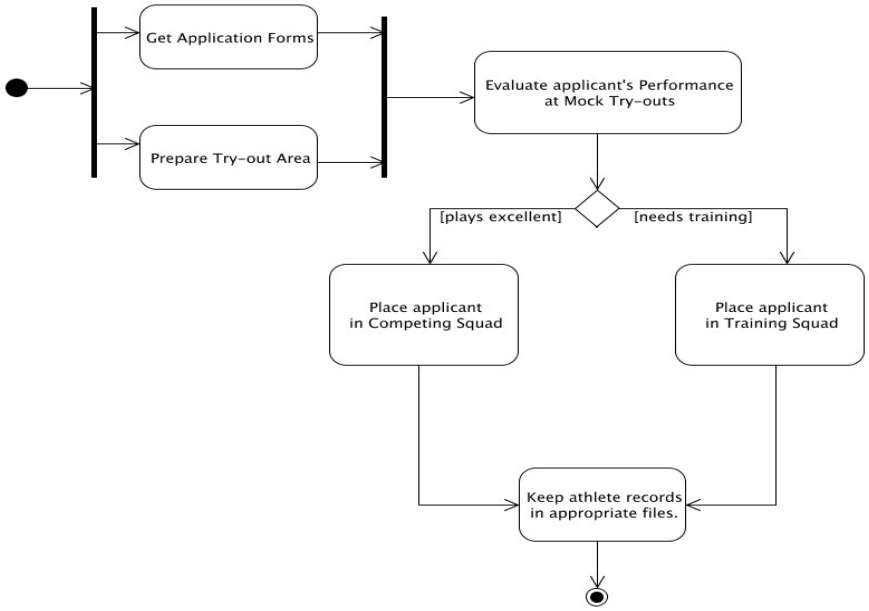


Figura 10 Exemplo de Diagrama de Atividade

6.5. Diagrama de Seqüência

Indica o comportamento dinâmico colaborativo entre objetos para a seqüência de mensagem enviada entre eles, em uma seqüência de tempo. A seqüência de tempo é mais fácil de visualizar em um diagrama de seqüência, do topo até a base. Escolha o diagrama de seqüência quando a seqüência de operações necessita ser mostrada. A figura 11 mostra um exemplo de diagrama de seqüência.

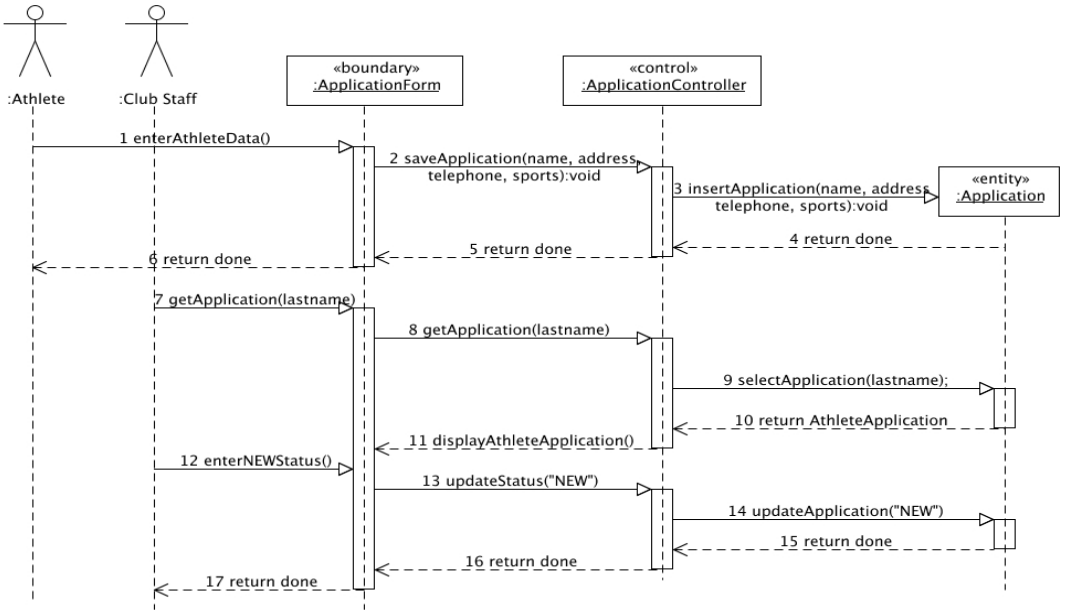


Figura 11 Exemplo de Diagrama de Seqüência

6.6. Diagrama de Colaboração

Indica os objetos reais e as ligações que representam as "redes de objetos" que estão colaborando. A seqüência de tempo é indicada pela numeração do índice da mensagem das ligações entre os objetos. O diagrama de colaboração é mais apropriado quando os objetos e suas ligações facilitam o entendimento da interação entre os objetos, e a seqüência de tempo não é importante. A figura seguinte mostra um exemplo de diagrama de colaboração.

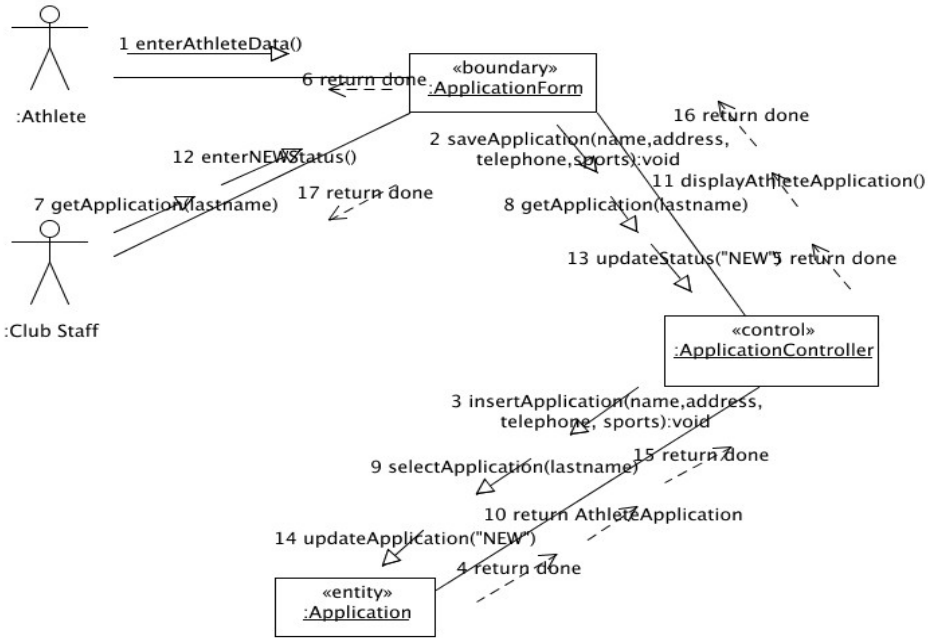


Figura 12: Exemplo de Diagrama de Colaboração

6.7. Diagrama de Estado

Mostra todos os possíveis estados que objetos da classe podem assumir e quais eventos causaram sua mudança. Mostra como os estados dos objetos mudam de acordo com o resultado de eventos que são controlados pelo objeto. Bom para ser usado quando uma classe tem comportamento de ciclo de vida complexo. A figura seguinte mostra um exemplo do diagrama de estado.

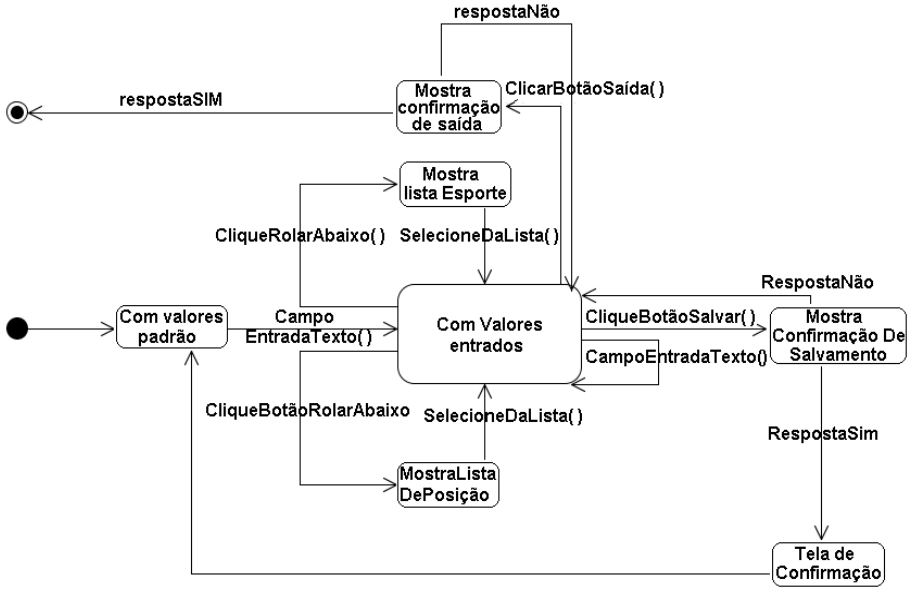


Figura 13: Exemplo de Diagrama de Estado

6.8. Diagrama de Componentes

Mostra os componentes do software em termos de códigos fonte, códigos binários, ligação de bibliotecas dinâmicas etc. A figura 14 é um exemplo de diagrama de componentes.

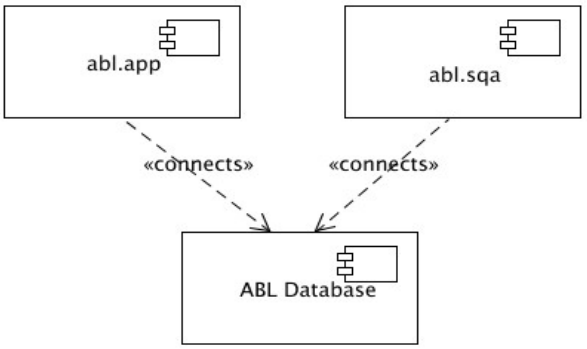


Figura 14: Amostra do diagrama de componentes

6.9. Diagrama de Implementação

Mostra a arquitetura física do hardware e de software do sistema. Ele realça a relação física entre software e componentes de hardware no sistema. Componentes no diagrama tipicamente representam os módulos físicos de código e correspondem exatamente ao diagrama de pacotes. A figura seguinte é um exemplo do Diagrama de Implementação.

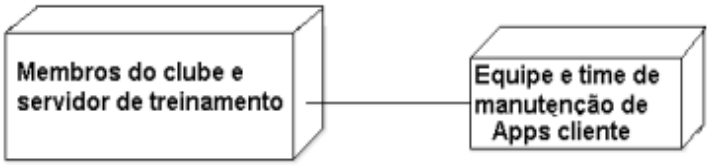


Figura 15: Amostra do Diagrama de Implementação

6.10. Criando um projeto UML utilizando o NetBeans

O módulo de UML está disponível com o *NetBeans Update Center* (*tools – Update Center*).

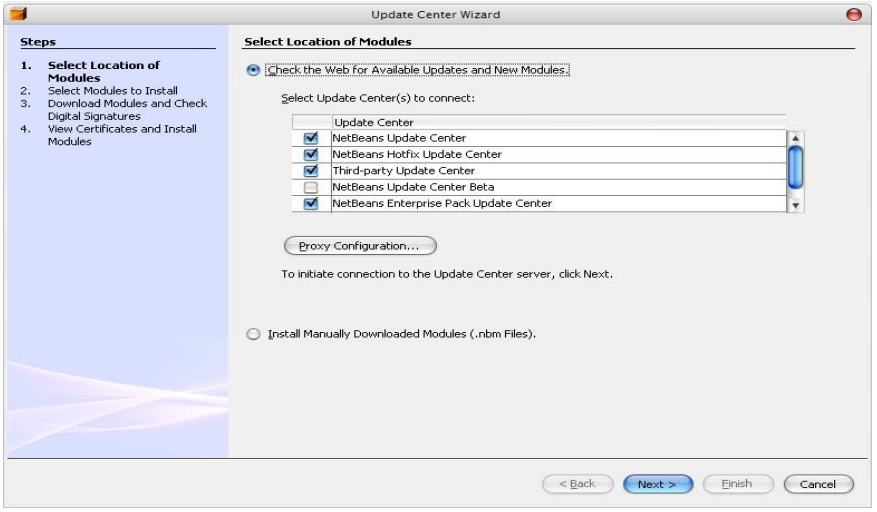


Figura 16: Update Center Wizard do NetBeans

Após instalado este módulo, para criar um projeto UML, siga os seguintes passos:

1. Ao acessar File | New Project para criar um novo projeto, uma nova categoria UML será

mostrada.

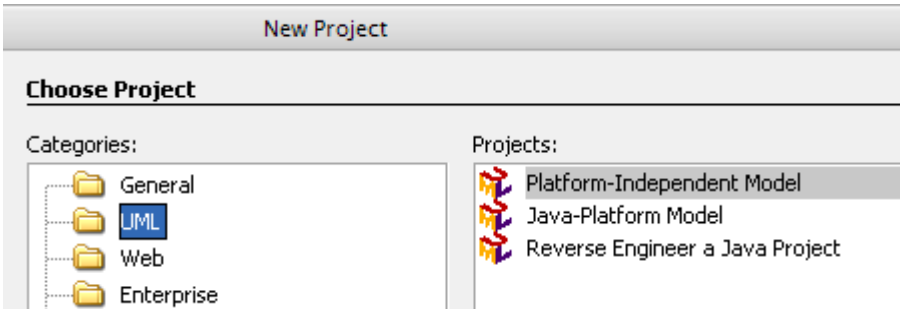


Figura 17: Categoria UML na Janela New Project

- **Platform-Independent Model** – Utilizado exclusivamente para modelos UML, sem relacionamento com a linguagem.
- **Java-Platform Model** – Regras de negócio são aplicadas em linguagem Java.
- **Reverse Engineer a Java Project** – Transforma um projeto feito em Java no modelo de classe UML, este projeto deve estar disponível na palheta *Project* do NetBeans.

2. Conheça e teste todas as opções disponíveis. Utilizaremos a opção *Java-Platform Model*. Ao ser pressionado o botão *Next*, a janela seguinte irá solicitar o nome do projeto, a localização do projeto e a pasta do projeto. Pressionando o botão *Finish* a seguinte janela será mostrada.

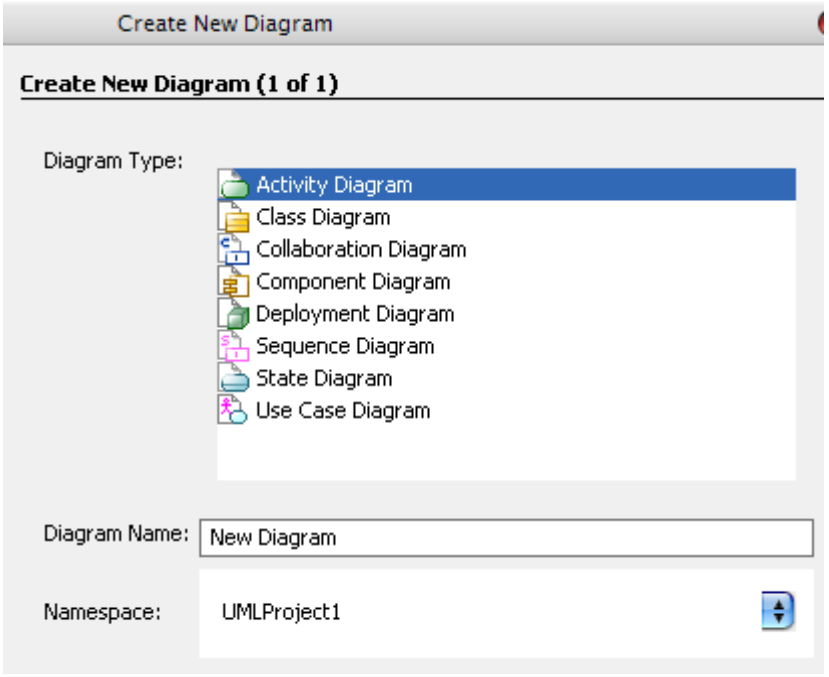


Figura 18: Criação de um novo diagrama

O projeto solicita a criação de um dos oito diagramas (atividades, classe, colaboração, componente, desenvolvimento, seqüência, estado e caso de uso) para o início do projeto, selecionaremos *Class Diagram*. Informaremos **ClasseSistema** ao nome do diagrama (*Diagram Name*) e em seguida pressionaremos o botão *Finish* e o projeto será criado.

3. A janela de projeto conterá os seguintes arquivos conforme a seguinte figura:

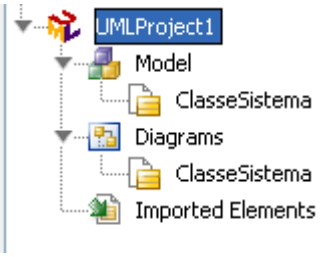


Figura 19: Janela Projects

4. Para criar uma nova classe, pressione o botão direito do mouse sobre o projeto (UMLProject1) e selecione a opção **`Add | Element`** e será mostrada a seguinte janela:

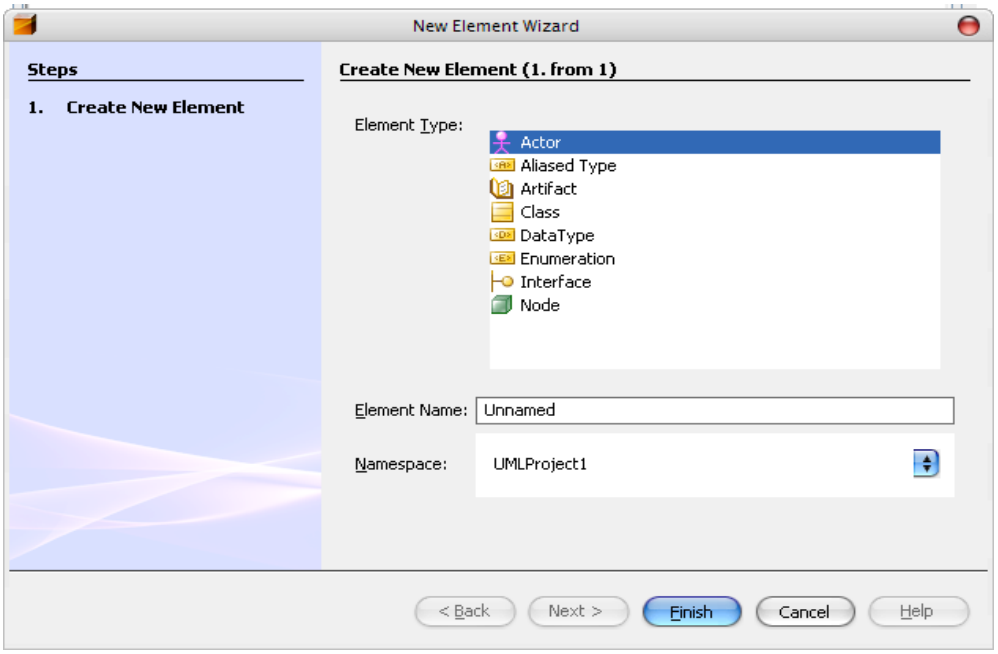


Figura 20: Janela New Element Wizard

Selecionamos *Class*, para *Element Name* modificaremos para **Funcionario** e em seguida pressionamos o botão *Finish*.

5. Na pasta *Model* o novo elemento será criado, arraste com o botão do mouse para a área de trabalho e obteremos a figura descrita abaixo.

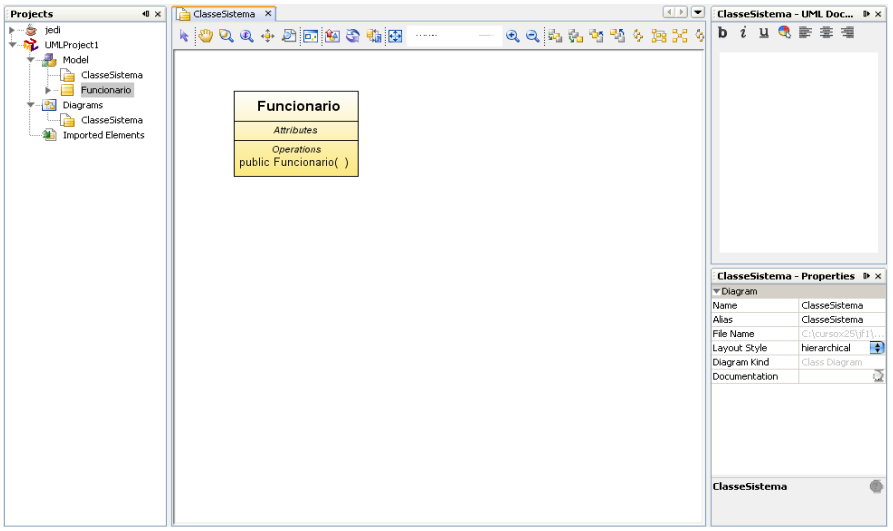
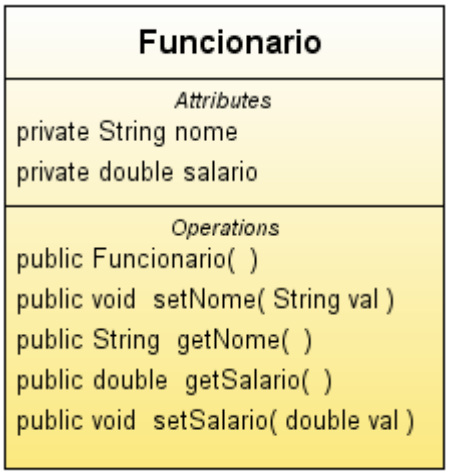


Figura 21: Área de trabalho no NetBeans

6. Para adicionar atributos a classe **Funcionario**, pressione o botão direito do mouse em cima do elemento (na área de *Projects* | *Model* | *Funcionario*) e selecione `Add | Attribute` (para novos métodos selecione `Add | Operation`) ou pressione `Alt + Shift + A` (para métodos `Alt + Shift + O`) selecionando a figura que representa a classe **Funcionario**.
7. Pressione um duplo clique sobre o atributo para alterar as propriedades deste, a janela *Properties* (no canto inferior direito) permite uma alteração mais refinada (se o atributo for estático por exemplo).
8. O último passo, após tudo pronto é a geração dos códigos fontes em Java. Selecione a classe Funcionario, pressione o botão direito do mouse e selecione a opção *Generate Code...* e como resultado teremos a seguinte classe.




```
public class Funcionario {
  private String nome;
  private double salario;
  public Funcionario() {  }
  public String getNome() {
    return nome;
  }
  public void setNome(String val) {
    this.nome = val;
  }
  public double getSalario() {
    return salario;
  }
  public void setSalario(double val) {
    this.salario = val;
  }
}
```

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.