UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Clarice Knihs de Camargo

EXPLORANDO A ABORDAGEM ORIENTADA A AGENTES NO DESENVOLVIMENTO DE SISTEMAS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador:

Prof. Rosvelter João Coelho da Costa

Florianópolis, Maio de 2002.

EXPLORANDO A ABORDAGEM ORIENTADA A AGENTES NO DESENVOLVIMENTO DE SISTEMAS

Clarice Knihs de Camargo

Esta dissertação foi julgada adequada para obtenção do Título de Mestre em Ciência da Computação, Área de Concentração Sistemas de Computação, e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Prof. Dr. Fernando A. O. Gauthier
Coordenador do Curso de
Pós-Graduação em Ciência da Computação

Membros da Banca:

Prof. Dr. Rosvelter João Coelho da Costa
Orientador e Presidente da Banca

Prof. Dr. João Bosco Mangueira Sobral

Prof. Dr. Roberto Willrich

Prof. Dr. Vitório Bruno Mazzola

AGRADECIMENTOS

Agradeço ao Curso de Pós-Graduação em Ciência da Computação e à Universidade Federal de Santa Catarina pela infra-estrutura e organização que viabilizaram o desenvolvimento deste trabalho.

Agradeço às secretárias do CPGCC Vera e Valdete, que sempre me atenderam com imenso carinho, simpatia e competência.

Agradeço aos professores João Bosco Mangueira Sobral, Roberto Willrich e Vitório B. Mazzola por participarem no julgamento deste trabalho.

Agradeço especialmente ao meu orientador Prof. Rosvelter João Coelho da Costa, pela dedicação, paciência e extrema competência com que me orientou, além do grande aporte de conhecimentos que me transmitiu durante esse tempo de pesquisa.

Agradeço ao meu marido Murilo, com certeza o grande incentivador deste trabalho, que sempre me apoiou com amor e muito carinho.

Agradeço à minha família, em especial à minha mãe e à minha irmã Rita, que sempre tiveram certeza do meu sucesso.

Agradeço também, de maneira especial, à minha sogra D. Izis, pela profunda compreensão com que sempre me ouviu e o imenso carinho que me transmitiu durante todos esses anos.

Para Carolina e Ana Claudia, meus dois lindos presentes, que tantas vezes souberam aceitar minha ausência nas brincadeiras. Sem dúvida, minhas grandes companheiras.

SUMÁRIO

LISTA DE ABREVIATURAS

AAMAS	*Autonomous Agents & Multi-Agent Systems*
AOR	*Agent-Object Relationship*
API	*Application Programmer Interface*
BDI	*Belief-Desire-Intention*
DOA	Desenvolvimento Orientado a Agentes
ESOA	Engenharia de Software Orientada a Agentes
GVD	Gerenciador Virtual de Disciplinas
JDBC	*Java Data Base Connection*
JVM	*Java Virtual Machine*
MaSE	*Multiagent Software Engineering*
OO	Orientação a Objetos
OA	Orientação a Agentes
OMG	*Object Management Group*
POA	Programação Orientada a Agentes
RMI	*Remote Method Invocation*
TCP	*Transmission Control Protocol*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho explora a abordagem orientada a agentes, suas principais características e a utilização de agentes de software no projeto de sistemas.

É apresentado o desenvolvimento de um sistema cliente/servidor denominado Gerenciador Virtual de Disciplinas (GVD), onde agentes de software foram utilizados no projeto do subsistema servidor. Durante o desenvolvimento, foram utilizadas várias técnicas de análise e ferramentas de projeto de software. Entre elas, destacam-se algumas ferramentas padronizadas pela notação *Unified Modeling Language*, técnicas e *frameworks* básicos da programação orientada a objetos e padrões de projeto. Foi construído um protótipo totalmente funcional desse sistema utilizando a plataforma Java disponibilizada pela *Sun Microsystems*.

A arquitetura do sistema GVD incorporou um sistema de agentes de software para facilitar o gerenciamento das interações entre as diversas partes funcionais do subsistema servidor em um ambiente concorrente. Foram definidos dois tipos principais de agentes: o agente gerenciador que administra as sessões usuários e encaminha suas solicitações; e o agente funcional que coordena a execução das operações necessárias ao atendimento dessas solicitações.

Palavras-chave: programação orientada a objetos, agentes, arquitetura de software, projeto orientado a agentes, sistemas de informação.

ABSTRACT

This work explores the agent-oriented approach, its main characteristics and the use of software agents in system design.

The development of a client-server system called Gerenciador Virtual de Disciplina (GVD) is presented, where software agents have been used in the server subsystem. During the system design, several analysis techniques e software design tools were used. Among them, some of the most important are the Unified Modeling Language tools, object-oriented techniques, object-oriented basic frameworks and some design patterns. A very functional prototype of this system was building using the Java platform of the Sun Microsystems.

The architecture of GVD system incorporated the use of software agents to facilitate the management of the interactions among several functional parts of the server subsystem within a concurrent environment. We have defined two main kinds of agents: the manager agent whom managing the user sections and leading the user's requests, and the functional agent that can co-ordinate the execution of the operations which are required to attend these requests.

INTRODUÇÃO

Projetar e construir um sistema de software de qualidade é uma tarefa que exige trabalho árduo. É necessário que o problema seja avaliado sob várias perspectivas, desde a definição de qual estratégia de desenvolvimento adotar até o esboço dos objetivos a serem alcançados. Comumente, modelos conceituais e estruturais são utilizados para auxiliar na visualização e especificação do sistema. Cada tipo de modelo focaliza o problema de uma maneira diferente.

Há diversos modelos de desenvolvimento de software. No modelo estruturado o foco principal é o controle de procedimentos e funções. Essa abordagem apresenta limitações porque à medida que o sistema cresce, a manutenção torna-se cada vez mais difícil. Outra abordagem é o chamado modelo objeto, que vê o sistema como uma coleção de objetos cooperativos. Objetos são criados a partir de componentes estáticos, conhecidos como classes, e possuem identidade, estado e comportamento. De maneira geral, o nome das classes e dos objetos advém do vocabulário associado ao problema ou à solução.

Nos últimos anos, o modelo objeto ganhou espaço no desenvolvimento de sistemas exatamente por facilitar o desenvolvimento de todo tipo de software, independente do tamanho ou complexidade. No modelo objeto, é possível visualizar o mundo real em termos computacionais, facilitando o entendimento do problema. Além disso, um dos maiores objetivos desse modelo é a flexibilidade e a reutilização do software. Nesse sentido, surgiram algumas ferramentas como *frameworks* e padrões de projeto (*design patterns*) que reaproveitam soluções já implantadas e testadas anteriormente.

Por outro lado, além da escolha da metodologia, no desenvolvimento de um sistema é necessário também levar em conta outras questões: qual é a estrutura de um bom projeto orientado a objetos? Qual é a melhor arquitetura para um dado sistema? Quais os componentes que devem ser construídos? Qual é a função de cada componente? Como tratar as interações entre eles? Respostas a essas perguntas são

justamente o objeto de estudo da Engenharia de Software e dependem das características e da natureza do problema.

Alguns pesquisadores advogam que o modelo objeto torna-se menos adequado à medida que a complexidade do sistema aumenta. Uma alternativa que pode até mesmo ser considerada uma extensão da abordagem orientada a objetos é a chamada abordagem orientada a agentes. Essa nova abordagem aplica-se principalmente a sistemas onde a estrutura de interação envolve um grande número de subsistemas.

Este trabalho apresenta o estudo e o desenvolvimento de uma aplicação cliente-servidor, denominada Gerenciador Virtual de Disciplina, GVD, através da qual alunos e professores trocam informações a respeito de uma disciplina. Esta aplicação representa um sistema distribuído onde há a caracterização bem definida de dois subsistemas: o subsistema cliente e o subsistema servidor. O primeiro fornece acesso remoto aos recursos do sistema; o segundo, o suporte necessário ao atendimento de praticamente todas as funcionalidades que o sistema oferece.

O sistema GVD foi implantado utilizando-se a tecnologia Java disponibilizada pela *Sun Microsystems*. O subsistema cliente incorpora a parte gráfica do sistema, possibilitando ao usuário interagir com o GVD através de diversas janelas, onde são apresentados os serviços disponibilizados pelo sistema, como troca assíncrona de mensagens, avisos, informações e o compartilhamento de recursos para grupos de usuários registrados. O cliente conecta-se ao servidor através da API Socket de Java, criando uma *stream* através da qual os dados fluem entre os dois terminais. O subsistema servidor executa os serviços solicitados pelo cliente e retorna o resultado. Os dados são armazenados em um banco de dados manipulado através da API *Java Data Base Connection* (JDBC). No ambiente servidor, cada conexão cliente gera um processo, a sessão usuário, que disputa recursos e compartilha informações simultaneamente com outros processos. Dessa maneira, para gerenciar a concorrência e a sincronização entre os processos, a arquitetura do servidor utiliza as técnicas da orientação a agentes criando componentes autônomos executores das diversas tarefas. Os agentes utilizados são agentes de software com dois tipos de função: agentes gerenciais e agentes funcionais. Os agentes gerenciais gerenciam a sessão usuário e os agentes funcionais executam os serviços solicitados. Em relação aos benefícios obtidos

com o uso de agentes, a arquitetura resultante mostrou-se extremamente flexível e reutilizável em outros sistemas similares.

Esse trabalho está organizado da seguinte maneira:

- No capítulo 2, são apresentados os conceitos básicos da Programação Orientada a Objetos e a utilização de Padrões de Projeto e *Frameworks* no desenvolvimento de sistemas;

- No capítulo 3, é apresentada a abordagem orientada a agentes. Um quadro comparativo entre as técnicas objeto e agente é apresentado;

- No capítulo 4, a análise e o projeto do sistema GVD são apresentados. Utilizou-se a notação *Unified Modeling Language* para a descrição conceitual do sistema;

- No capítulo 5, é apresentada a proposta final para a arquitetura do subsistema servidor, com uma abordagem orientada a agentes;

- O capítulo 6 apresenta a análise final e as perspectivas de continuação desse trabalho.

2. SISTEMAS ORIENTADOS A OBJETOS

2.1 Introdução

Esse capítulo apresenta um estudo geral sobre a abordagem orientada a objetos e o uso de padrões de projeto e *frameworks* no desenvolvimento de sistemas. O objetivo é esclarecer os principais conceitos, características, vantagens e limitações dessa metodologia. Assim, inicialmente serão apresentados os conceitos de classe, objeto, herança e polimorfismo. Posteriormente serão abordadas as vantagens da utilização de padrões de projeto e *frameworks.*

A Programação Orientada a Objetos (POO) é um paradigma da Engenharia de Software que incorpora à computação uma visão do mundo real através de um componente de abstração chamado objeto. Objetos podem ser entendidos, portanto, como simulações de partes de sistema real, tangíveis ou não, possuindo estado e comportamento.

Os conceitos da orientação a objetos tornam-se importantes para o bom entendimento desse trabalho, uma vez que o estudo desenvolvido nessa dissertação é inteiramente desenvolvido sob os preceitos dessa técnica.

2.2 Características da Programação Orientada a Objetos

Uma característica importante da Programação Orientada a Objetos é o princípio da abstração. A abstração consiste em ignorar os aspectos de um assunto que não sejam relevantes para o propósito em questão. Abstrair é a arte de examinar seletivamente certos aspectos de um problema, isolando os aspectos importantes para o entendimento do problema e suprimindo aqueles que, no momento, não são necessários. A abstração deve ser feita de acordo com algum propósito, pois será o propósito que definirá o que é e o que não é importante. Dependendo do propósito final, é possível obter diferentes abstrações de um mesmo assunto.

A programação orientada a objetos incorpora a visão do mundo real através de abstrações. A principal abstração essa técnica é o objeto que é criado a partir de outra abstração, a classe. A seguir, alguns conceitos elementares da programação orientada a objetos são brevemente relembrados:

Objeto

Um objeto representa uma entidade do mundo real, dentro de um conceito computacional. Segundo [COA1993], um objeto é uma abstração de alguma entidade no domínio de um problema ou em sua realização, refletindo a capacidade de um sistema manter informações sobre ela, interagir com ela, ou ambos. Isso significa dizer que um objeto não é necessariamente alguma coisa palpável, visível. Uma mensagem de erro, por exemplo, também pode ser um objeto. Todos os objetos, incluindo os objetos de software, têm um estado e um comportamento que são definidos respectivamente pelos seus atributos e métodos. Os valores dos atributos descrevem o estado do objeto e os métodos definem seu comportamento.

Classe

Um objeto é uma instância de uma classe. Classes são abstrações que definem as características de um conjunto de elementos. A classe representa o conjunto de características e comportamento que serão inerentes às suas instâncias. Uma classe não reserva espaço de memória para os dados de suas instâncias. É preciso instanciar a

classe, criando um objeto que terá vida própria, que ocupará espaço na memória e que terá um estado e um comportamento próprio [LEW2000].

Herança

Herança é uma técnica fundamental para organizar e criar classes na orientação a objetos. Através da herança, uma nova classe é derivada de outra pré-existente. A nova classe automaticamente contém os atributos e métodos da classe original. O projetista pode, então, redefinir ou adicionar variáveis e métodos à nova classe. A motivação central de se utilizar herança é a idéia de reuso do software. Utilizando-se componentes de software pré-existentes para se criar outros, os esforços para sua criação, implantação e testes são minimizados.

Interação entre Objetos

Objetos não agem isoladamente. Em um sistema orientado a objetos é preciso que os diversos tipos de objetos interajam de maneira cooperativa a fim de cumprir certos objetivos comuns. Essa interação é feita através de mensagens passadas entre os objetos. As mensagens podem ser entendidas como invocação de métodos sobre os objetos, podendo, inclusive, possuir parâmetros que são os valores passados para o âmbito de execução do método.

Encapsulamento

Encapsulamento é uma característica da orientação a objetos através da qual os atributos de um objeto são alterados somente através dos métodos do próprio objeto [LEW2000]. Dessa maneira, um objeto é encapsulado do resto do sistema, de maneira a ter suas propriedades alteradas somente através de seus métodos. O encapsulamento prevê um respeito às fronteiras do objeto, definindo uma interface entre ele e o resto do sistema.

Polimorfismo

Polimorfismo é o termo utilizado na programação orientada a objetos para identificar as várias formas de definição de um método. Há dois tipos principais de polimorfismo: por sobrecarga e por inclusão. No primeiro caso, um mesmo nome é associado à diferentes assinaturas (tipo, ordem e número de parâmetros) em uma mesma classe, portanto, à diferentes definições. No segundo caso, um método de uma classe é redefinido em uma subclasse. A subclasse, então, é considerada uma especialização da superclasse.

Flexibilidade e Reutilização

Um dos maiores objetivos da orientação a objetos é o desenvolvimento de sistemas que sejam adaptáveis, fáceis de ser alterados, flexíveis em novas realizações e extensões e que possibilitem o reuso de algumas partes ou componentes em outros sistemas, quando necessário. Para isso, algumas técnicas foram desenvolvidas dentro da orientação a objetos: o uso de *frameworks* e padrões de projeto. A principal vantagem é o reaproveitamento de soluções, as quais tornam-se padrões entre os programadores. A desvantagem é a dificuldade de uso, pois é necessário que o projetista entenda não só o problema, mas também a solução apontada pelo padrão de projeto ou *framework.*

2.3 Utilização de Padrões de Projeto e Frameworks

Em todas as áreas do conhecimento humano, problemas são inicialmente resolvidos de forma *ad hoc* e específico para aquele caso. Com o passar do tempo, as experiências são passadas de pessoa a pessoa para difundir as soluções que melhor funcionaram. No desenvolvimento de sistemas, o processo é quase o mesmo. Para um problema específico, apesar de muitas vezes haver mais de uma solução apenas uma delas é a mais indicada. Nesse sentido, os padrões de projeto vêm capturar soluções simples e elegantes dentro do contexto de um problema, incorporando flexibilidade e reusabilidade ao software.

2.3.1 Padrões de Projeto

A idéia de padrões de projeto foi concebida pelo arquiteto Christopher Alexander, que estudou formas de facilitar o processo de construção de edifícios em áreas urbanas. Segundo [GAM2000] cada padrão define um problema que ocorre repetidamente em um ambiente, descrevendo a solução ideal para o problema, de tal maneira que você usa essa solução muitas e muitas vezes, sem fazer a mesma coisa duas vezes. Apesar da proposta original se referir a problemas arquitetônicos, sua idéia foi incorporada à orientação a objetos para dar ao software maior flexibilidade e reusabilidade.

Um marco na utilização de padrões de projeto no desenvolvimento de sistemas orientados a objetos foi a publicação do livro "*Design Patterns – Elements of Reusable Object-Oriented Software*" [GAM1994, GAM2000] que catalogou a estrutura de 23 padrões. Assim, de maneira geral, temos as seguintes definições para padrões de projeto:

- Um padrão é uma **solução** para um **problema** de software dentro de um **contexto** [MOO1999].

- Padrões de projeto são descritos como classes e objetos que interagem para resolver um problema geral de projeto dentro de um contexto particular [GAM2000].

- Padrões de projeto variam em termos de granularidade e nível de abstração. Em [GAM2000] encontramos uma classificação de padrão de acordo com dois critérios ortogonais:

1. **Escopo**: define se o padrão se aplica a classes ou a objetos. Se o escopo for direcionado à classes, o padrão irá lidar com relacionamentos entre classes e subclasses, fixadas em tempo de compilação. Se o escopo for objetos, o padrão será direcionado a relacionamentos entre objetos que ocorrem em tempo de execução;

2. **Propósito**: define o que um padrão faz. Padrões podem ter propósitos de criação, de estrutura ou de comportamento, da seguinte forma:

- **Propósito de Criação**: refere-se ao processo de criação de objetos. Se o escopo do padrão for direcionado a classes, o processo de criação de objetos é delegado as subclasses; se o escopo for direcionado a objetos, a criação se dá através de outros objetos;

- **Propósito de Estrutura**: refere-se à composição de classes ou objetos. Padrões estruturais de classes usam herança para compor novas classes, enquanto padrões estruturais de objetos descrevem modos de juntar vários objetos;

- **Propósito de Comportamento**: caracteriza a maneira que classes e objetos irão interagir e distribuir responsabilidades. Padrões de comportamento de classes usam herança para descrever algoritmos e fluxos de controle; padrões de comportamento de objetos descrevem de que maneira um grupo de objetos irá cooperar para executar uma operação que um único objeto não é capaz de realizar.

A Tabela 2.1 apresenta a classificação e uma breve descrição do aspecto funcional dos padrões definidos em [GAM2000]. Observando a tabela, podemos notar que a maioria dos padrões tem um escopo direcionado a objeto.

Padrão de Projeto	Escopo / Propósito de	Aspecto
Abstract Factory	Objeto / Criação	Provê a criação de famílias de objetos ou objetos relacionados.
Adapter (class)	Classe / Estrutura	Converte a interface de uma classe em outra, de modo a adaptá-las.
Adapter (object)	Objeto / Estrutura	Converte a interface de uma classe em outra, de modo a adaptar a interação entre os objetos.
Bridge	Objeto / Estrutura	Desacopla a abstração de sua realização, deixando que ambas variem livremente.
Builder	Objeto / Criação	Separa a construção de um objeto de sua representação, permitindo que a mesma construção crie diferentes objetos.
Chain of Responsibility	Objeto / Comportamento	Permite que mais de um objeto possa responder a uma requisição, evitando um acoplamento forte entre o objeto emissor e o receptor.
Command	Objeto / Comportamento	Encapsula uma requisição como um objeto, permitindo parametrização de requisições entre diferentes clientes.
Composite	Objeto/ Estrutura	Compõe objetos em estruturas de árvores, para representar hieraquias de parte-para-o-todo.

Padrão de Projeto	Escopo / Propósito de	Aspecto
Decorator	Objeto/ Estrutura	Anexa responsabilidades adicionais para um objeto dinamicamente.
Facade	Objeto/ Estrutura	Provê uma interface unificada para um conjunto de interfaces em um subsistema.
Factory Method	Classe / Criação	Define uma interface para a criação de objetos, delegando às subclasses a responsabilidade de decidir qual objeto criar.
Flyweight	Objeto/ Estrutura	Usa desacoplamento para suportar eficientemente um grande número de objetos.
Interpreter	Classe / Comportamento	Define uma representação gramatical para interpretar comandos . em uma determinada linguagem.
Iterator	Objeto / Comportamento	Provê uma maneira de acessar seqüencialmente objetos agregados, sem expor sua realização.
Mediator	Objeto / Comportamento	Define um objeto que encapsula o modo como um conjunto de objetos interage.
Memento	Objeto / Comportamento	Captura o estado interno de um objeto sem violar o encapsulamento, de modo a ter o estado restaurado, se necessário.

Padrão de Projeto	Escopo / Propósito de	Aspecto
Observer	Objeto / Comportamento	Define uma dependência de um-para-muitos entre os objetos, de tal maneira que se o estado de um se altera, todos os outros são notificados.
Prototype	Objeto / Criação	Especifica o tipo de objeto a ser criado, definindo um protótipo dessa instância e criando os demais por clonagem.
Proxy	Objeto/ Estrutura	Provê um objeto substituto para um objeto real, controlando o acesso ao objeto real.
Singleton	Objeto / Criação	Cria uma única instância de uma classe e provê um acesso global a ela.
State	Objeto / Comportamento	Permite que um objeto mude de comportamento quando seu estado interno for alterado.
Strategy	Objeto / Comportamento	Define um conjunto de algoritmos, encapsula cada um deles e torna-os intercambiáveis.
Template Method	Classe / Comportamento	Define o esqueleto de um algoritmo, deixando que as subclasses definam alguns passos sem alterar a estrutura principal.
Visitor	Objeto / Comportamento	Representa uma operação a ser executada sobre os elementos de uma estrutura de objeto.

Tabela 2.1 – Classificação dos Padrões de Projeto [GAM2000]

Na seqüência, apresenta-se o detalhamento de alguns padrões utilizados no desenvolvimento do sistema em estudo nessa dissertação.

Padrão de Projeto **Command**

Muitas vezes a comunicação entre dois ou mais objetos assume a forma de comandos que um objeto emissor passa a outros objetos. Desse modo, o objeto emissor precisa "conhecer" o objeto receptor e saber de que maneira o comando será enviado.

O padrão Command permite que o objeto emissor conheça a referência para o comando, sem precisar saber o que o comando faz e qual objeto irá executá-lo. O objeto emissor conhece a referência para o objeto Command e envia para esse objeto a mensagem de execução. O objeto Command então, é responsável por despachar o comando para o objeto que efetivamente irá executar a ação. É como se o objeto Command fosse uma "caixa preta" sobre a qual será executado um método `execute()` sempre que alguém precisar de seus serviços, desacoplando o objeto que invoca a operação daquele que irá efetuá-la.

O padrão Command sugere a criação de uma interface com um método `execute()` abstrato. Para cada operação deverá ser criada uma classe que irá realizar essa interface, definindo o método `execute()` de acordo com a operação.

A Fig. 2.1 apresenta a estrutura do padrão Command:

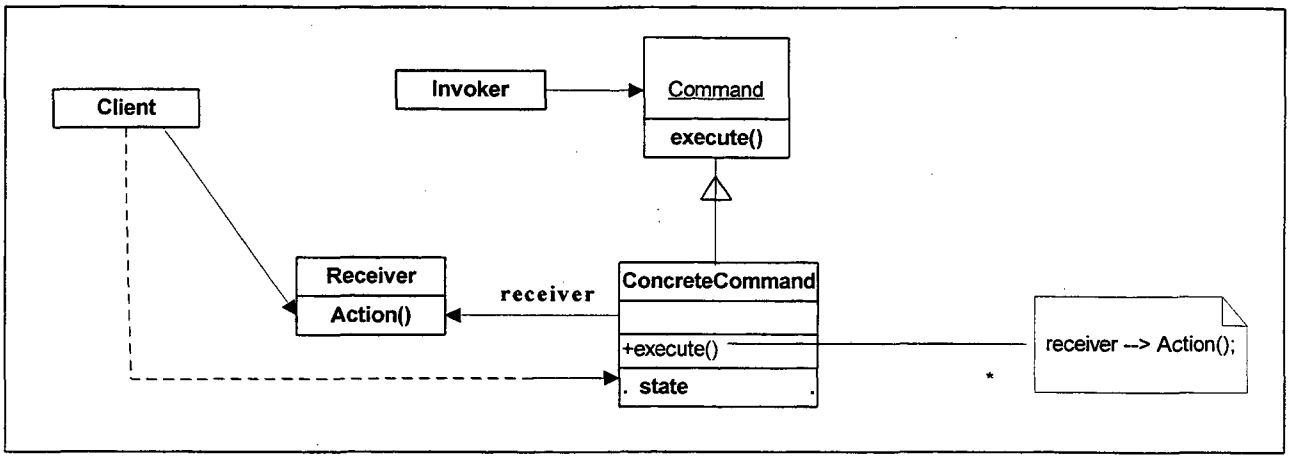

Figura 2.1 – Estrutura do Padrão Command [GAM2000]

A seguir, são descritos os elementos da estrutura do padrão Command.

Command

Declara uma interface para executar uma operação.

ConcreteCommand

Classe que implementa a interface Command, implementando o método execute com a correspondente operação no Receiver.

Client

Aplicação que utiliza o serviço. Cria o objeto ConcreteCommand com seus respectivos receptores (*receivers*).

Invoker (objeto emissor)

Invoca o objeto Command para atender à solicitação.

Receiver (objeto receptor)

Classe que efetivamente atenderá à solicitação do cliente e executará a operação. Qualquer classe pode ser um *receiver.*

A Fig. 2.2 apresenta um diagrama de seqüência para o padrão Command, onde é possível visualizar o desacoplamento entre o objeto emissor (*invoker*) e o objeto que efetivamente efetuará a operação (*receiver*).

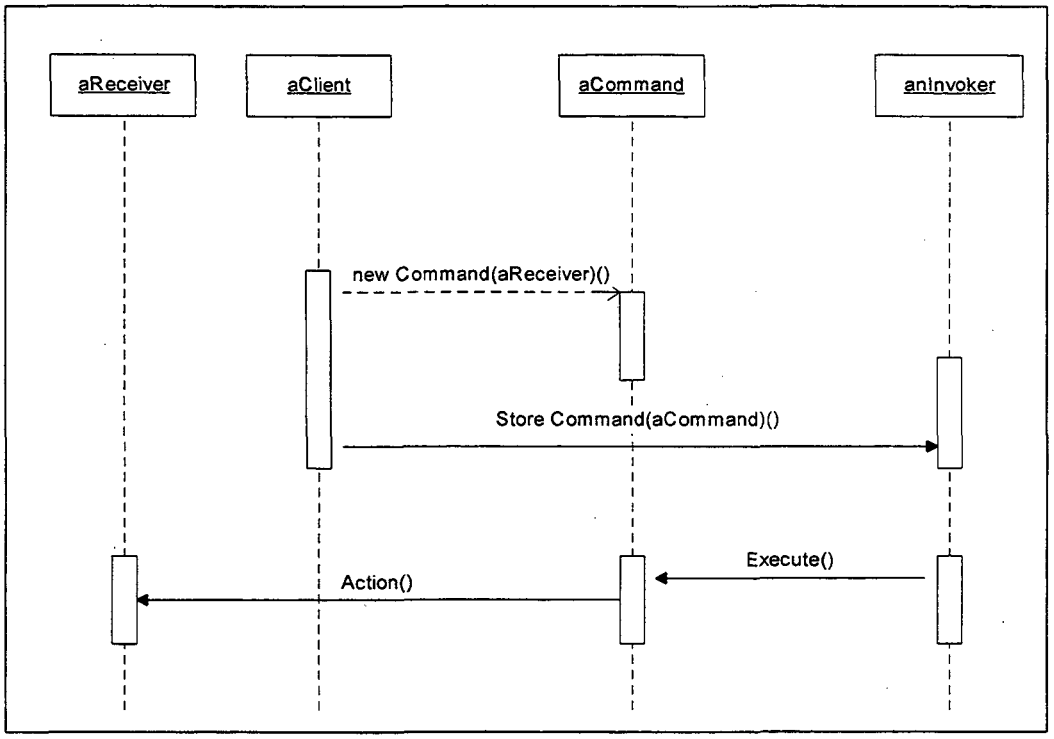

Figura 2.2 – Diagrama de Sequência do Padrão Command [GAM2000]

De acordo com essa figura, temos:

- O cliente solicita um serviço, criando um objeto ConcreteCommand e especificando o *receiver*;
- Um objeto *invoker* encapsula (*store*) o objeto ConcreteCommand;
- O objeto *invoker* (emissor) invoca o método *execute*() do Command sobre o objeto;
- O objeto ConcreteCommand invoca a operação, que será efetivamente executada pelo *receiver*.

O uso do padrão Command apresenta as seguintes vantagens:

1. Transformar uma operação em um objeto, que pode ser passado para outros métodos ou objetos como um parâmetro [ECK2001];

2. Desacopla o objeto que invoca a operação daquele que tem a tecnologia para executá-la;

3. Um objeto Comand é um objeto de primeira-classe, podendo ser manipulado e estendido como qualquer outro objeto;

4. Objetos Command podem ser agrupados em macro-commands, inclusive usando o padrão Composite;

5. Facilita a adição de novos comandos, uma vez que não é necessário mudar nenhuma classe já existente.

Padrão de Projeto **Factory Method**

Durante a construção de um programa, muitas vezes torna-se necessário criar vários tipos de objetos, alguns com características semelhantes. Se a criação desses objetos for feita em diversas partes do programa, no caso de alguma alteração ou no caso de adicionarmos um novo tipo de objeto, será preciso alterar várias partes do programa. A solução é forçar a criação de objetos em um único ponto do programa [ECK2001]. Isso pode ser feito através de uma fábrica de objetos (*factory*), exatamente o conceito estipulado pelo padrão Factory Method. Numa eventual adição de novos tipos de objetos ou alteração de algum já existente, haverá um único lugar a ser alterado dentro do programa. O Factory Method sugere a criação de uma classe abstrata que delega às subclasses a criação do objeto. As subclasses podem, inclusive, ser instanciadas sob demanda, cada uma delas retornando especificamente um tipo de objeto.

A Fig. 2.3 apresenta a estrutura do padrão Factory Method, onde:

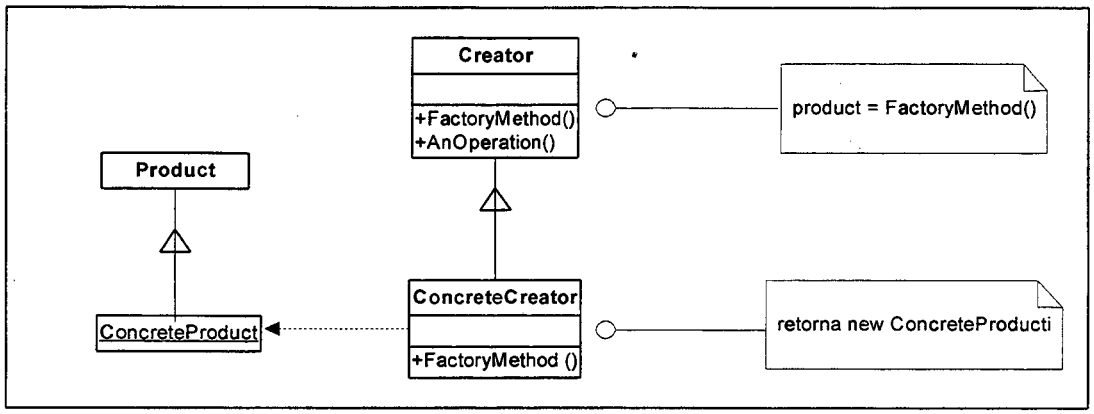

Figura 2.3 – Estrutura do Padrão Factory Method [GAM2000]

Product:

Define a interface de objetos que o método `factory()` cria.

ConcreteProduct:

Realiza a interface *Product.*

Creator:

Declara o *factory* método, que retorna um objeto do tipo *Product.*

ConcreteCreator:

Subscreve o método *factory* para retornar uma instância de um *ConcreteProduct.*

O padrão Factory Method é usado geralmente junto com o Abstract Factory. O Abstract Factory incorpora vários tipos de Factory Method, cada um criando um tipo de objeto.

Padrão de Projeto **Singleton**

Muitas vezes é necessário que uma classe tenha uma única instância e que essa instância sirva a todos os usuários. Isso pode ser feito com o uso do padrão Singleton. O padrão Singleton garante que uma classe tenha uma única instância e provê um acesso global a ela. A chave é garantir que o usuário tenha somente uma maneira de criar o objeto: através da classe Singleton. O objeto Singleton deve ser mantido "privado", sendo acessado através de métodos estáticos. A classe Singleton pode ser instanciada das seguintes maneiras:

- Estaticamente, disponibilizando um objeto mesmo que ele não seja solicitado ou utilizado;
- Sob demanda, disponibilizando o objeto no momento em que um cliente solicitar, sendo então disponibilizado para todos os outros clientes.

A Fig. 2.4 apresenta a estrutura do padrão Singleton.

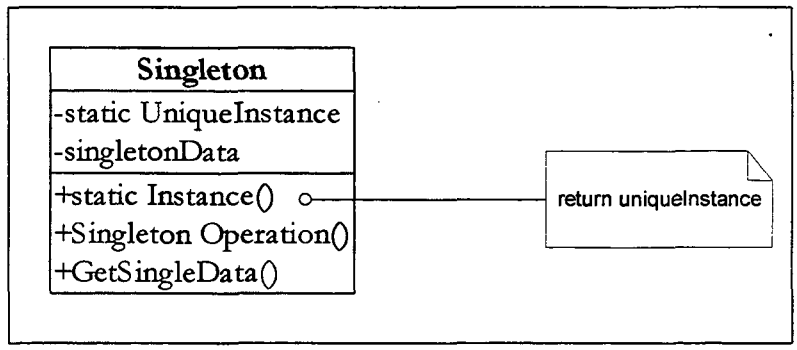

Figura 2.4 – Estrutura do Padrão Singleton [GAM2000]

Onde:

Singleton

Define uma classe com uma operação que cria uma única instância, disponibilizando-a para todos os usuários.

O uso do padrão Singleton apresenta as seguintes características:

1. Controla o acesso a uma instância única, definindo um controle rígido de como e quando os clientes irão acessar o objeto;

2. Permite refinamento de operações e redução do número de variáveis que guardam instâncias únicas;

3. Permite um número variável de instâncias, sendo fácil alterar a quantidade de acordo com o número de instâncias desejadas;

4. Mais flexível do que operações de classes.

Padrão de Projeto **Proxy**

Muitas vezes é interessante usar uma classe fictícia, através da qual o cliente acessa um serviço concreto. Uma das razões é minimizar os custos de criação de um objeto, controlando para criá-lo somente quando for necessário. Por exemplo, ao abrir uma aplicação que contenha uma imagem num dos cantos, nem sempre a imagem real é necessária, podendo ser substituída por uma fictícia até que o cliente solicite a verdadeira [GAM2000]. O padrão Proxy provê exatamente esse tipo de realização, através da qual uma classe é utilizada no lugar de uma outra, a "real".

A Fig. 2.5 apresenta a estrutura do padrão Proxy, onde:

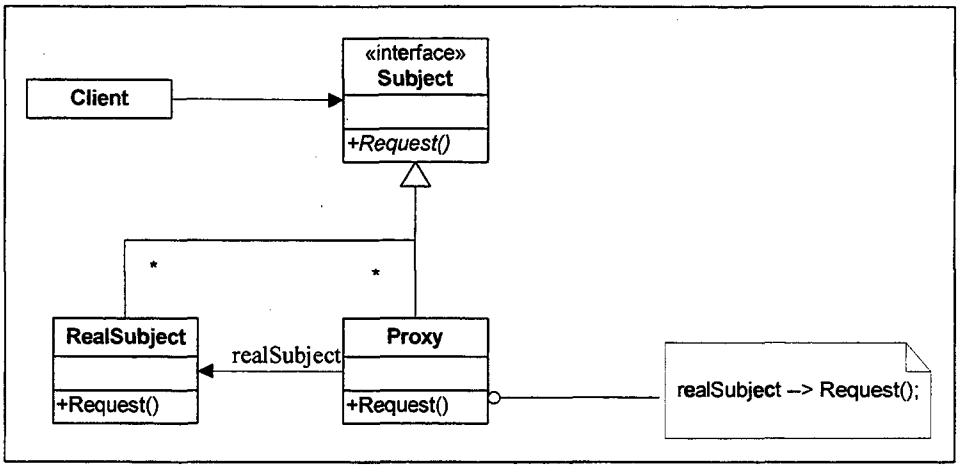

Figura 2.5 – Estrutura do Padrão Proxy [GAM2000]

Proxy::

Mantém uma referência que permite ao *Proxy* identificar o objeto da classe real; provê uma interface através da qual o *Proxy* pode substituir o objeto real; controla o acesso ao objeto real e pode ser responsável por criá-lo e destruí-lo;

Subject::

Define uma interface comum para o *RealSubject* e o *Proxy,* de tal maneira que um *Proxy* pode ser usado em qualquer lugar onde se espera um objeto *RealSubject;*

RealSubject:

Define o objeto real que o *Proxy* representa.

O padrão Proxy pode ser de três tipos:

- Remote Proxy

É responsável por enviar uma requisição e seus argumentos para um objeto real localizado em um outro ambiente. O Remote Proxy pode esconder o fato dos objetos residirem em diferentes ambientes;

- Virtual Proxy

Pode armazenar informações a respeito do objeto real de maneira a adiar o seu acesso. O Virtual Proxy pode efetuar operações de maneira otimizada como, por exemplo, criar um objeto sob demanda;

- Protection Proxy

Verifica se o usuário tem permissão para acessar uma determinada operação.

O uso do padrão Proxy apresenta as seguintes características:

1. O Remote Proxy provê um representante local para um objeto real que está localizado em outro ambiente, muitas vezes, através de uma rede;
2. O Virtual Proxy racionaliza a criação de objetos, de maneira a criá-los somente quando realmente necessário;
3. O Protection Proxy define a segurança da aplicação, controlando o acesso ao objeto real, sendo muito útil quando objetos possuem diferentes direitos de acesso.

2.3.2 Frameworks

Um *framework* é um conjunto de classes cooperativas que criam um projeto reusável para uma classe de software específica [GAM2000]. O *framework* dita a arquitetura da aplicação; define a estrutura geral do software, seu particionamento em classes e objetos e a colaboração entre ambos, além de definir o fluxo de controle.

O uso de um *framework* incrementa o desenvolvimento do sistema porque predefine o projeto, permitindo ao projetista se concentrar nas especificações do sistema. O *framework* captura as decisões de projeto que são comuns ao problema dentro de um domínio, definindo um esqueleto de classes abstratas, suas responsabilidades e colaborações. O projetista utiliza o *framework* para uma aplicação particular, implementando suas classes abstratas e instanciando as classes do *framework* conforme sua necessidade. Assim, o *framework* incorpora à aplicação a idéia de reusabilidade de projeto.

Apesar de *framework* e padrões de projeto apresentarem algumas similaridades, existem algumas diferenças que devem ser consideradas:

- Padrões são mais abstratos do que *frameworks;*
- Padrões são estruturas menores do que *frameworks*; um *framework* pode conter dezenas de padrões;
- Padrões são menos especializados do que *frameworks*; *frameworks* sempre têm um domínio particular de aplicação, enquanto padrões se aplicam a um problema particular dentro de qualquer aplicação.

O uso de *frameworks* é cada vez mais difundido e importante no desenvolvimento de sistemas. É através deles que a característica de reusabilidade é incorporada à maioria dos sistemas orientados a objetos.

2.4 Conclusão

Esse capítulo relembrou conceitos básicos da programação orientada a objetos e um estudo geral sobre a utilização de padrões de projeto e *frameworks* no desenvolvimento de sistemas orientados a objetos.

Desenvolver softwares orientados a objetos não é uma tarefa fácil visto que é necessário encontrar as chaves de abstração dentro do domínio do problema, transformá-las em classes e promover as interações entre seus objetos. É necessário, ainda, definir as interfaces das classes, a estrutura hierárquica do sistema e estabelecer os relacionamentos entre as entidades.

A técnica de desenvolvimento de sistemas orientado a objetos é um paradigma da Engenharia de Software que tem por objetivo prover o desenvolvimento de sistemas orientados a objetos com ferramentas e metodologias que facilitem o trabalho. Nesse sentido, a utilização de padrões de projeto e *frameworks* vem trazer flexibilidade e reusabilidade ao software. A flexibilidade é alcançada através de padrões que facilitam a manutenção e a incorporação de novos componentes ao sistema. A reusabilidade, por sua vez, é alcançada através do uso de *frameworks,* que definem um conjunto de classes cooperativas e seus relacionamentos dentro de um contexto de software específico, e do uso de padrões de projeto que reaproveitam soluções já utilizadas anteriormente.

Nos últimos anos, além da Engenharia de Software Orientada a Objetos, uma nova técnica vem sendo difundida para auxiliar no desenvolvimento de sistemas: a orientação a agentes. A Engenharia de Software Orientada a Agentes aborda, principalmente, o desenvolvimento de sistemas complexos, prometendo facilitar o gerenciamento de um grande número de interações num ambiente concorrente. As principais características de sistemas complexos e os conceitos básicos da técnica de desenvolvimento de sistemas orientados a agentes são abordados no próximo capítulo.

3. ARQUITETURA DE SOFTWARE ORIENTADA A AGENTES

3.1 Introdução

Nesse capítulo será apresentado um estudo sobre a técnica de desenvolvimento de sistemas orientados a agentes. Enquanto alguns pesquisadores estão inclinados a considerar a orientação a agentes como um novo paradigma da Engenharia de Software [LIN2000], outros a consideram simplesmente uma extensão da orientação a objetos [WOO1999]. Apesar do assunto ainda ser motivo de algum debate, a maioria dos pesquisadores concorda com o fato de que o uso de agentes vem suprir algumas deficiências da orientação a objetos. Essas deficiências se mostram, principalmente, no desenvolvimento de sistemas complexos, onde, num ambiente de concorrência, há um grande número de interações entre os subsistemas constituintes.

Para um melhor entendimento da técnica de desenvolvimento de sistemas orientados a agentes, esse capítulo aborda, inicialmente, o conceito de sistemas complexos e suas principais características. Apresenta, ainda, um estudo geral sobre a técnica de orientação a agentes, seus conceitos, características, vantagens e diferenças em relação à orientação a objetos. Por último, é apresentado um breve estudo sobre a Engenharia de Software Orientada a Agentes.

3.2 Sobre a Complexidade do Software

Pode ser considerado complexo o software que contém muitos componentes interagindo dinamicamente, cada um no seu próprio *thread* de controle e incorporados a um ambiente de concorrência [JEN2001]. Esse tipo de software é, sem dúvida, muito mais difícil de gerenciar do que aquele com uma única função executada através de um único *thread.* "Atualmente, é notório que as interações são, provavelmente, a mais importante característica dos sistemas complexos ..."[1].

Um sistema complexo normalmente é grande e formado por diversas partes, incorporando módulos, conexões e concorrência. A complexidade, no entanto, não advém do seu tamanho, mas sim do grande número de interações entre suas diversas partes [SIM1996]. Segundo [BRO1995] a complexidade de um sistema é uma propriedade inata do tipo de tarefa para o qual o *software* foi projetado.

Analisando as características de *software* complexo, é possível verificar que existe uma certa regularidade entre eles [SIM1996]:

- O sistema é composto por subsistemas inter-relacionados. Dessa maneira, normalmente a complexidade toma a forma hierárquica, onde os subsistemas interagem através de relacionamentos organizacionais;

- A escolha pelos componentes primitivos do sistema é relativamente arbitrária, sendo definida muitas vezes por objetivos externos ao sistema;

- É possível distinguir as interações **entre** os subsistemas das interações **dentro** dos próprios subsistemas. Essas últimas são mais freqüentes e mais previsíveis do que as primeiras. Essa qualidade faz com que cada subsistema possa ser tratado como um módulo quase independente. A independência só não é total por causa das interações entre eles.

---

[1] Chamada para trabalhos do Autonomous Agents & Multi-Agent Systems (AAMAS 2002).

Somando todas essas qualidades, é possível definir uma visão canônica de um sistema complexo da forma como é ilustrado na Fig.3.1 [JEN2001].

Os *links* "relacionado com" expressam a hierarquia natural do sistema; os *links* "interação freqüente" conectam os componentes **dentro** dos subsistemas e os *links* "interação não freqüente" conectam as interações **entre** os componentes.

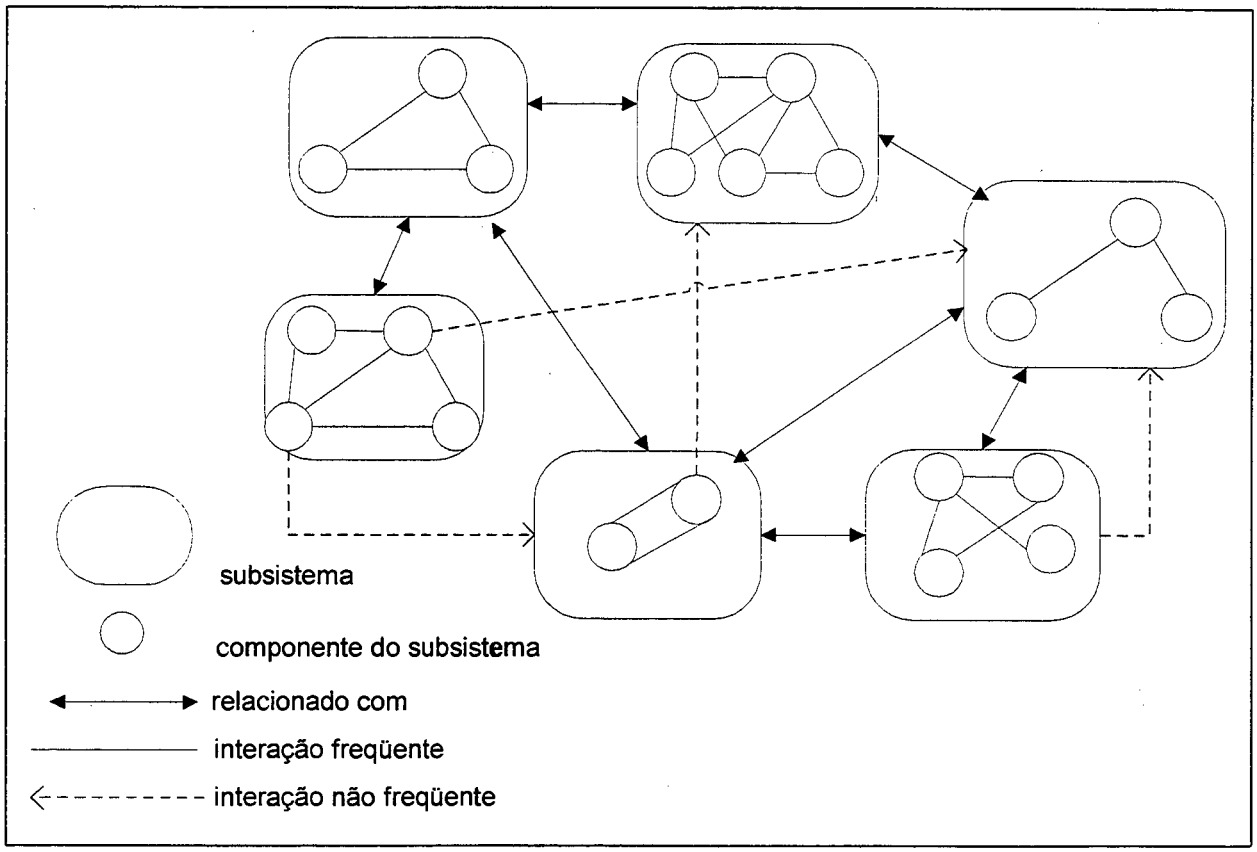

Figura 3.1 Visão Canônica de um Sistema Complexo

Cabe à Engenharia de Software fornecer ferramentas para gerenciar a complexidade dos sistemas. Segundo [JEN2001], essas ferramentas são baseadas nos seguintes mecanismos:

- **Decomposição**

  A técnica mais fundamental para tratar um grande problema é dividi-lo em pedaços menores. Isso pode ser observado no sistema decomposto em subsistemas

da Fig. 3.1. A decomposição ajuda a entender o problema porque limita o escopo do projetista para cada parte individualmente.

- **Abstração**

O processo de abstrair simplifica o modelo, enfatizando os detalhes realmente importantes para aquele problema específico. Mais uma vez, o escopo do projetista é limitado à area de interesse.

- **Organização**[2]

É o processo de definir e gerenciar os inter-relacionamentos entre os vários componentes. Isso também pode ser observado entre os subsistemas e seus relacionamentos na Fig. 3.1. Através da organização, alguns componentes básicos são agrupados e tratados como unidades de análise de alto nível. A organização também provê um meio de descrever os relacionamentos de alto-nível entre essas unidades.

A partir dessas premissas, é possível imaginar que uma maneira natural de construir sistemas complexos é em termos de múltiplos componentes, autônomos e interativos, cada um com seu objetivo particular e inserido em um ambiente colaborativo. Segundo [BOO1994] "...em um sistema complexo ... dado um nível de abstração, encontramos uma coleção significante de objetos que colaboram para obter uma visão de mais alto nível". Como agentes podem ser descritos em termos de "...entidades cooperando para alcançar objetivos comuns" e " ... coordenando suas ações..." [JEN2001] podemos entender que, nesse caso, a orientação a agentes talvez seja a melhor solução. Assim, apesar da orientação a objetos ter um espaço considerável no projeto e desenvolvimento de sistemas, há de se perceber a importância crescente da orientação a agentes dentro da Engenharia de Software, principalmente no que tange sistemas complexos.

O conceito de sistema complexo abrange muitos tipos de software. Sem dúvida, são muitos os tipos de sistemas divididos em subsistemas que interagem em um

---

[2] Em [BOO1994], BOOCH usa o termo hierarquia.

ambiente de concorrência. Nesse trabalho, trataremos o conceito de sistema complexo de maneira mais informal, atendo-se principalmente às interações entre subsistemas e à concorrência entre os processos.

3.3 A Noção de Agente

Agente é uma entidade computacional autônoma, inteligente, colaborativa e adaptável [AMU2000]. Entende-se por inteligente a habilidade de inferir, executar ações e incorporar informações relevantes, dentro de um certo domínio. Agentes são capazes de flexibilidade e autonomia para alcançar seus objetivos dentro do ambiente ao qual estão incorporados [WOO1997]. Um agente, também chamado de agente de software inteligente, é um pedaço autônomo de software, com capacidades incorporadas da Inteligência Artificial [BRU2000]. A palavra "inteligente" faz parte de suas características, sendo utilizada porque o software pode ter diversos tipos de comportamento.

O conceito exato de agente ainda é motivo de alguns debates na comunidade de pesquisa. Nesse trabalho, iremos adotar um conceito mais informal de agente, de maneira a tratar a sua autonomia dentro de um contexto limitado e a inteligência simplesmente como uma adaptação do software a todo tipo de informação que pode receber. Utilizaremos o conceito de agente como um elemento fortemente ligado a uma funcionalidade do sistema, incorporado a um ambiente colaborativo[3] e distribuído.

Dentro do conceito de agentes, um esquema de classificação comum é a noção de representação "fraca" e "forte" [KEN2000]. Na noção de representação fraca, agentes têm vontade própria (autônomos), interagem (comportamento social), respondem a estímulos (reativos) e tomam iniciativa (pró-ativos). Na noção de representação forte, a noção de representação fraca é mantida, acrescida de outras características como mobilidade (o agente altera seu campo de atuação), veracidade (o agente faz

---

[3] Ambiente no qual há interações entre agentes, de modo que um possa cooperar com o outro sobre o curso de uma função.

exatamente o que promete fazer) e racionalidade (irá executar sua função de maneira ótima, tomando decisões próprias em tempo de execução).

Nesse trabalho iremos considerar a representação fraca de agente, incorporada à estrutura funcional do sistema. Os agentes terão mais em comum com software do que com inteligência. Por isso, serão referenciados simplesmente como agentes de software.

3.4 Sistemas Multiagentes

Em um sistema orientado a agentes, na maioria das vezes um único agente não é capaz de resolver a totalidade dos problemas[4]. É necessário um grupo de agentes incorporados a um ambiente interagindo em prol de um objetivo comum. O envolvimento de múltiplos agentes auxilia:

- na representação da natureza descentralizada do problema;
- no gerenciamento de vários *locus* de controle;
- na percepção de múltiplas perspectivas;
- no controle da competitividade de interesses.

Assim, agentes necessitam interagir, tanto para alcançar seus propósitos particulares como para gerenciar as dependências dentro de um mesmo ambiente. Essas interações acontecem devido ao comportamento social dos agentes que utilizam estruturas de mensagens e protocolos para coordenação e negociação entre si. Com isso há uma indicação clara de que agentes são organizados em sociedades ou grupos [KEN2000].

---

[4] Na teoria de orientação a agentes, não é viável o desenvolvimento de um sistema com um único agente.

Independente do objetivo do sistema, há dois pontos que diferem a orientação a agentes de outros paradigmas da Engenharia de Software:

- Interações entre os agentes normalmente ocorrem através de uma linguagem de comunicação declarativa baseada numa teoria *speech act* [MAY1995]. Conseqüentemente, interações são conduzidas em termos de quais os objetivos a alcançar, quando e através de quem;

- Agentes são solucionadores flexíveis de problemas, com um campo de visão e controle limitados sobre o ambiente em que operam. Dessa maneira, interações precisam ser gerenciadas também de uma maneira flexível.

Agentes agem tanto individualmente como em conjunto. Podem, também, gerenciar outros agentes. Torna-se necessário, portanto, definir a natureza das interações entre eles. Podem ser interações sociais, por exemplo, onde um agente é eleito líder para gerenciar o time de agentes ou interações surgidas em tempo de execução. Outro exemplo seria um grupo de agentes se unindo para entregar um serviço que um único agente não consegue fazer [JEN2001]. A extensão temporal dessas relações também pode variar enormemente.

Atualmente, diversas aplicações incorporam a filosofia de orientação a agentes:

- Atores, jogadores e lutadores nos jogos de computador e nas simulações;

- O *paperclip* assistente no *Microsoft Office*;

- Sites de busca na internet.

Observando a Fig. 3.2, é possível perceber que uma abordagem satisfatória da orientação a agentes inicia na decomposição do problema em componentes múltiplos, autônomos e livres para agir e interagir a fim de alcançar seus objetivos. O modelo define uma abstração baseada em agentes, interação e organização, além de apresentar o limite de atuação de cada grupo de agentes.

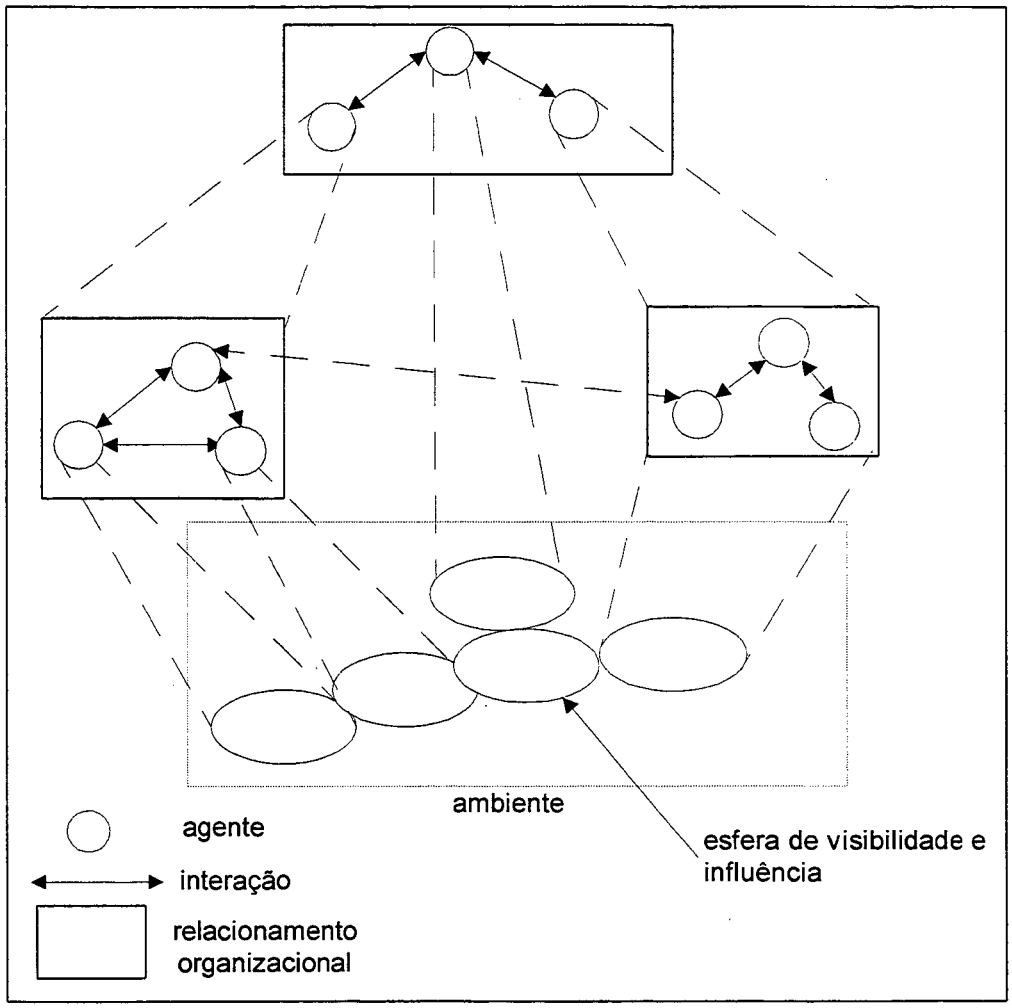

Figura 3.2 Visão Canônica de um Sistema Multiagente [JEN2001]

No projeto desenvolvido nesta dissertação, as técnicas de desenvolvimento de sistemas multiagentes serão utilizadas para decompor as funcionalidades do sistema, delegando cada função a um agente ou a um grupo deles.

3.5 Objetos X Agentes

Embora haja certas similaridades entre a abordagem orientada a objetos e a orientada a agentes, é preciso entender as diferenças para poder selecionar a melhor técnica para cada projeto a ser desenvolvido.

A Tabela 3.1 mostra um quadro comparativo entre as duas abordagens [AMU2000, BRU2000, JEN1999, JEN2001, KEN2000, WOO1997]:

Objeto	**Agente**
Ë uma unidade de abstração usada para representar uma única instância do mundo real.	É uma unidade de abstração que pode representar um ou vários objetos do mundo real.
Passivos por natureza, precisam receber uma mensagem para iniciar algum comportamento.	São componentes ativos, podem iniciar atividade por conta própria.
Tem o comportamento ditado por métodos.	Tem autonomia, podendo se comportar segundo suas próprias decisões.
Encapsula informações.	Encapsula informações.
Interações são muito importantes.	Interações são muito importantes.
Objetos esperam chamadas de métodos.	Agentes são reativos, respondem a estímulos.
Objetos são usados como abstração para representar entidades passivas.	Agentes podem ser vistos como possíveis sucessores dos objetos, representando as abstrações de entidades ativas.
Interações ad hoc, construídas para cada situação específica.	Interações em alto-nível, através de linguagens de comunicação entre agentes.
Relacionamentos são definidos por hierarquias estáticas, como herança.	Suportam e provêem recursos para gerenciar relacionamentos organizacionais em diversos níveis.
Controlados externamente.	Comportamento autônomo que não pode ser controlado diretamente (caixa preta).

Tabela 3.1 – Quadro Comparativo entre Objetos e Agentes

3.6 Engenharia de Software Orientada a Agentes

Considerando o ciclo de vida de um sistema, é necessário inicialmente que os agentes sejam identificados e definidos. Tais requisitos podem ser alcançados através de uma abordagem na qual se dá ao agente a capacidade de ter crenças, desejos e intenções (*beliefs-desires-intentions*). Assim, seguindo uma abordagem formal, um sistema orientado a agentes deve ser capaz de representar os seguintes aspectos [JEN1999]:

- Crenças: inerente ao agente, representa o conjunto de informações que o agente tem sobre o seu ambiente;
- Desejos: os objetivos que os agentes tentarão alcançar;
- Intenções: as ações que os agentes terão que executar e os efeitos dessas ações;
- Cooperação: as interações entre agentes.

Teorias que explicam como esses aspectos podem ser alcançados são conhecidas como "teoria de agentes". Duas dessas teorias mais famosas são a teoria de intenção de Cohen-Levesque[5] e o modelo de crença-desejos-intenções (*belief-desire-intention*) de Rao-Georgeff[6] [JEN2001]).

---

[5] Modelo no qual são tomadas como primitivas somente duas atitudes: conhecimento e objetivo. Outras atitudes, como a intenção, são adicionadas a essas.

[6] Rao-Georgeff toma as intenções como atitudes primitivas, adicionando posteriormente os conhecimentos e os objetivos.

3.6.1 Terminologia

São muitos os termos utilizados para designar abordagens orientadas a agentes:

- **Programação Orientada a Agentes (POA)**

  Está sendo visto como uma extensão da programação orientada a objetos. O termo foi introduzido por Shoham em 1993;

- **Desenvolvimento Orientado a Agentes (DOA)**

  Extensão do Desenvolvimento Orientado a Objeto. A palavra desenvolvimento pode significar somente a programação ou então todas as fases de um projeto, desde a análise até a implantação final do sistema [WOO1999].

- **Engenharia de Software com Agentes, Engenharia de Software Baseada em Agentes, Engenharia de Sistemas Multiagentes (MaSE), Engenharia de Software Orientada a Agentes (ESOA)**

  São termos semanticamente equivalentes [LIN2000], com o porém de que MaSE se refere a uma metodologia específica e ESOA parece ser o termo mais popular.

- **Computação Orientada a Agentes**

  Cobre todos as abordagens relativas ao uso de agentes na área da computação [JEN1999].

3.6.2 Metodologias

Com o incremento do uso de agentes na computação, várias metodologias estão sendo desenvolvidas para auxiliar no desenvolvimento de sistemas orientados a agentes, tanto no campo da Inteligência Artificial como na Engenharia de Software. Algumas são utilizadas no desenvolvimento de *frameworks* específicos, como o *framework* JAFIMA[7] [KEN2000]) e o RETSINA[8] ([SYCARA,1996] apud [BRU2000]). Outras, apresentam uma abordagem geral para desenvolvimento de sistemas orientados a agentes.

A seguir são apresentadas algumas características de três metodologias para o desenvolvimento de sistemas orientados a agentes:

Gaia

É uma metodologia que estende os recursos da orientação a objetos para a orientação a agentes e que permite visualizar os agentes dentro de um nível micro (estrutura de agente) ou de um nível macro (agentes organizados em grupo ou em sociedade) [WOO2000]. A motivação por trás da metodologia Gaia é que existem metodologias que falham em apresentar a natureza autônoma e funcional do agente (funcional no sentido de resolver problemas), além de falharem também na organizacão das interações entre os agentes.

De maneira resumida, na fase de análise é necessário encontrar as funções a serem desempenhadas dentro do sistema e criar os relacionamentos entre elas. Na fase de *design* do projeto, as funções se transformam em tipos de agentes, sendo necessário criar o número correto de instâncias para cada tipo de agente. Por sua vez, os relacionamentos se transformam em interações, definindo a comunicação entre os agentes.

---

[7] JAFIMA é um *framework* baseado em agentes inteligentes e móveis.

[8] RETSINA provê uma arquitetura de agente baseada em Crença-Desejos-Intenção (*Belief-Desire-Intention*).

As restrições com relação ao uso da metodologia Gaia referem-se ao pequeno domínio dessa metodologia para aplicações desenvolvidas para a Internet. Devido a isso, algumas extensões estão sendo sugeridas [ZAM2000].

Metodologia de Engenharia de Sistemas Multiagentes (MaSE)

O objetivo da metodologia MaSE é guiar o projetista desde as especificações de requisitos iniciais até o projeto final do sistema baseado em agentes [WOO2000]. MaSE usa recursos estendidos da *Unified Modeling Language* (UML), sendo dividida em:

1. Capturar os objetivos.

   Definir e organizar os requisitos e objetivos do sistema hierarquicamente.

2. Aplicar Casos de Uso.

   Utilizar casos de uso para representar as especificações iniciais do sistema e diagramas de seqüência para determinar o número mínimo de interações entre as funções especificadas.

3. Redefinir Papéis.

   Definir alguns papéis a serem desempenhados no sistema e que são responsáveis pelos objetivos definidos na primeira fase. A definição desses papéis cria algumas tarefas que irão especificar de que maneira os objetivos serão alcançados.

4. Criar Classes de Agentes.

   Definir as classes que serão instanciadas para gerar os agentes.

5. Construir Interações.

   Definir um protocolo de comunicação entre os agentes através de diagramas de estado.

6. Definir Operações.

   Especificar a funcionalidade interna das classes dos agentes.

Relacionamento Agente-Objeto (RAO)

A metodologia RAO é inspirada em dois outros modelos: no Entidade-Relacionamento (ER) e no Banco de Dados Relacional (BDR) [WAG2000]. O propósito do modelo RAO é juntar as habilidades dinâmicas de agentes às características estáticas das entidades, modelando sistemas de informação de banco de dados baseados em agentes. Na metodologia RAO existem alguns tipos de entidades como: agentes, eventos, ações, interações e objetos. Organizações são modeladas como um grupo de agentes. Cada um dos agentes tem o direito de executar determinadas operações e o dever de monitorar eventos importantes para a organização. A interpretação dos direitos e deveres dos grupos de agentes parece ser similar à encontrada na metodologia Gaia. Um exemplo de sistema de informação para banco de dados orientado a agentes é encontrado em [MAG2000].

3.7 Conclusão

Nesse capítulo foi apresentado um estudo geral sobre a tecnologia de desenvolvimento de software orientado a agentes. Foram abordados os principais conceitos, características e as vantagens da utilização de agentes em um software. Uma breve comparação entre objetos e agentes foi apresentada a fim de esclarecer suas diferenças e limitações.

Apesar de ser uma área de pesquisa relativamente nova, vem crescendo o interesse pela orientação a agentes dentro da Engenharia de Software. Aparentemente, dividir um sistema em vários pequenos pedaços autônomos, independentes e interativos parece ser uma boa solução para alguns tipos de software, especialmente os sistemas distribuídos e concorrentes. É claro que a orientação a agentes não é a solução ideal para todos os problemas [WOO1998]. Mas pode ser mais uma boa ferramenta dentro da Engenharia de Software para facilitar o desenvolvimento e a manutenção de sistemas.

4. UM SISTEMA PARA GERENCIAMENTO DE DISCIPLINAS

4.1 Introdução

Esse capítulo apresenta a visão usuário de um sistema cliente/servidor denominada Gerenciador Virtual de Disciplina (GVD).

No ambiente GVD, o professor de uma disciplina poderá disponibilizar informações aos seus alunos tais como: avisos, arquivos para *downloads* e correio eletrônico. Tanto o professor como os alunos terão acesso ao sistema utilizando qualquer computador ligado a Internet através de uma interface gráfica local.

Durante a fase de análise, foram utilizados os recursos de descrição de sistemas definidos pela notação *Unified Modeling Language* (UML). Na construção dos programas, optou-se pelo ambiente de programação Java disponibilizado gratuitamente pela *Sun Microsystems*.

Embora exista um grande leque de serviços possíveis para esse tipo de aplicação, apenas alguns foram selecionados e realizados, mas suficientemente representativos para o estudo. É importante, inclusive, salientar que as funcionalidades da aplicação não estão inseridas dentro do objetivo principal do trabalho. Servem apenas para ilustrar a face operacional do sistema e, em especial, a viabilidade da arquitetura proposta para o servidor.

Inicialmente serão apresentadas noções de algumas tecnologias incorporadas à aplicação como a estrutura cliente/servidor, a linguagem Java e as características básicas de um ambiente de programação concorrente. Posteriormente, serão descritos o modelo conceitual do sistema e o projeto do subsistema cliente.

O projeto do subsistema servidor será apresentado no Capítulo 5.

4.2 Aplicações Cliente/Servidor

Sistemas cliente/servidor compreendem duas partes lógicas: um servidor que provê serviços e um cliente que requisita esses serviços. Juntos, formam um sistema completo com uma divisão distinta de responsabilidades. Tecnicamente, uma aplicação cliente/servidor relaciona dois ou mais *threads* (processos) de execução usando uma relação consumidor/produtor [LEW1998]. Clientes podem ser vistos como consumidores uma vez que solicitam serviços ou informação ao servidor utilizando o resultado para seus propósitos particulares. Servidores desempenham o papel de produtor executando os serviços ou buscando as informações solicitadas pelo cliente.

A arquitetura cliente/servidor abrange alguns conceitos de comunicação e redes. A comunicação entre programas permite a troca de mensagens de maneira a incorporar um modelo de solicitação/resposta.

A Fig.4.1 representa um sistema cliente/servidor tradicional [LEW1998]. Vários clientes independentes requisitam serviços do servidor através de uma rede de comunicação.

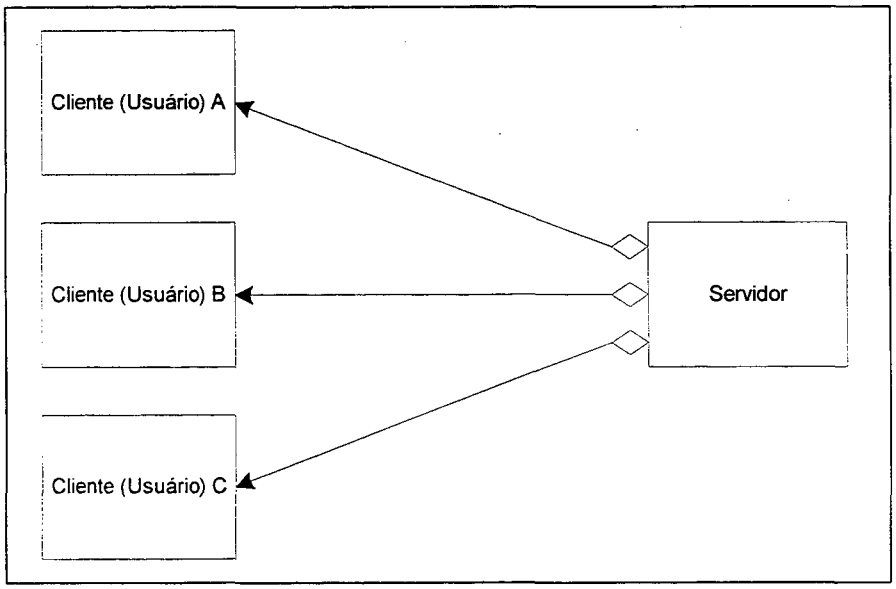

Figura 4.1 – Arquitetura Cliente/Servidor

Geralmente, além de se comunicar com um cliente, o servidor também mantém uma conexão com uma fonte de dados. Essa fonte de dados pode ser um arquivo, um banco de dados ou a própria WEB. Dessa maneira, muitas aplicações cliente/servidor são aplicativos distribuídos de três camadas[9] (*three-tier system*), consistindo em uma interface com o usuário, uma lógica de processamento e o acesso a uma base de dados. Todas as três camadas podem residir no mesmo computador ou em computadores diferentes interconectados por uma rede. A interface usuário comunica-se com a lógica do processamento na camada intermediária. A camada intermediária pode, então, conectar-se à base de dados para manipular os dados [DEI2001].

A Fig. 4.2 ilustra uma aplicação cliente/servidor de três camadas:

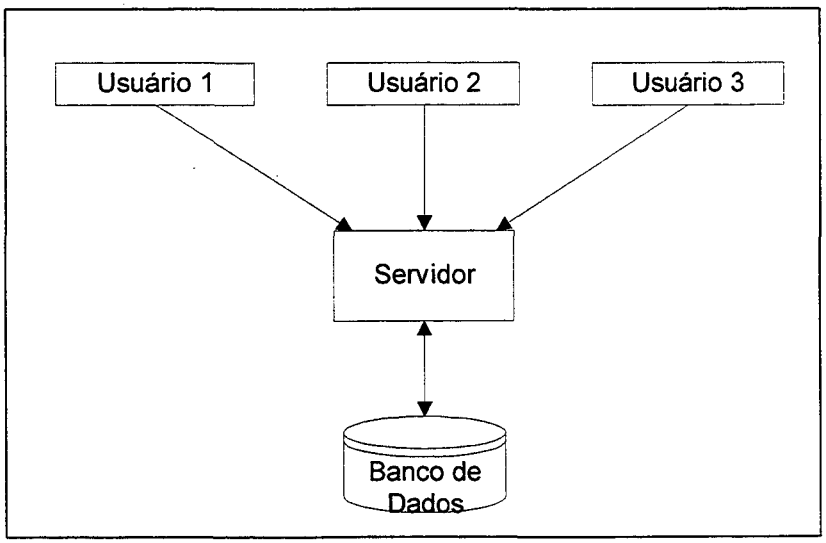

Figura 4.2 – Arquitetura Três Camadas

[9] Existem aplicações cliente/servidor de duas camadas (*two-tier system*), onde a lógica de processamento da aplicação está no programa cliente, no servidor ou dividida entre ambos [LEW1998].

O sistema GVD incorpora a arquitetura em três camadas. A conexão entre o cliente e o servidor é feita através da *Application Programmer Interface* (API) Socket de Java que utiliza a pilha de protocolos TCP/IP para a comunicação. A comunicação entre o servidor e um banco de dados relacional é feita através da API de Conectividade de Banco de Dados Java (JDBC).

4.3 Utilizando Java para Aplicações Cliente/Servidor

Sistemas cliente/servidor desenvolvidos utilizando a linguagem de programação Java incorporam um modelo de interação mais avançado que os oferecidos por outras linguagens [LEW1998] porque , além do fácil gerenciamento de *multithreading*, Java é multiplataforma. Isso significa que um programa desenvolvido em Java pode ser executado em uma variedade de plataformas, praticamente de maneira idêntica. Essa característica é obtida através de uma arquitetura de máquina denominada *Java Virtual Machine* (JVM), que fornece a interpretação aos *bytecodes* que são gerados pelo compilador Java. Certos fatores, como o tamanho de alguns tipos de dados e o comportamento de certos operadores aritméticos, são padronizados de maneira a garantir que um programa Java execute da mesma maneira em qualquer máquina[10].

Com relação à concorrência, Java é a única entre as linguagens de uso geral e popular que permite ao programador definir quais aplicativos contém fluxos de execução *(threads)* em separado. Cada *thread* designa uma parte de um programa que pode executar concorrentemente com outros *threads*.

Com relação aos recursos de rede, Java apresenta uma variedade de *frameworks* destinados ao processamento distribuído, tais como: *servlets* (javax.servlet), *applets* (javax.swing.JApplet), *Remote Method Invocation* (RMI) (java.rmi), *sockets* (java.net) e *Commom Object Request Broker Architecture*[11] (CORBA) [DEI2001, LEW2000]. Através desses *frameworks*, Java facilita o desenvolvimento de aplicativos distribuídos

---

[10] Inúmeras incompatibilidades ainda existem entre as máquinas virtuais. Por isso, muitas vezes esse processo funciona mais na teoria do que na prática.

[11] Faz parte da API de Java 2.

ou baseados na Internet, como também permite que aplicativos sendo executados em diferentes máquinas se comuniquem observando-se, todavia, as limitações de segurança.

No sistema GVD, serão utilizadas as classes e interfaces definidas no pacote java.net que agrupa os recursos fundamentais de redes. Através desse pacote, o sistema estabelece uma comunicação entre o cliente e o servidor através de um fluxo contínuo de dados. Inicialmente, o servidor estabelece uma porta através da qual ficará esperando as conexões dos clientes. Quando um aplicativo cliente solicita uma conexão, o servidor cria um objeto *socket* exclusivo para aquele usuário através do qual será criado um canal (*stream)* por onde fluirão as mensagens entre os dois programas. Enquanto a conexão estiver ativa, os dados fluem entre os dois processos, em ambas as direções. Estabelecida a conexão com um cliente, o servidor mantém a porta aberta para outras conexões.

4.4 Tratando Concorrência e Sincronização

Num sistema puramente seqüencial, os eventos acontecem um de cada vez e um após o outro. Dentro desse tipo de programa, as operações são executadas seqüencialmente, existindo, dessa maneira, um único fluxo de controle.

Num sistema concorrente, mais de um evento pode acontecer ao mesmo tempo em um dado ambiente [DEI2001]. Múltiplas operações são executadas simultaneamente, gerando múltiplos fluxos simultâneos de controle, cada um associado a um *thread* ou a um processo independente.

A notação UML representa um processo ou um *thread* através de uma abstração chamada classe ativa [BOO1999]. Classes ativas representam fluxos de controle e, ao serem instanciadas, geram objetos ativos. Classes passivas, por sua vez, não incorporam fluxos de controle, não podem iniciar o controle de atividades independentes e, ao serem instanciadas, geram objetos passivos.

A Fig. 4.3 apresenta a representação de classes ativas e passivas na notação UML [BOO1999]. A diferença entre as duas representações está unicamente na especificação de um sinal para a classe ativa. O sinal é a especificação de um estímulo de comunicação assíncrono entre duas entidades.

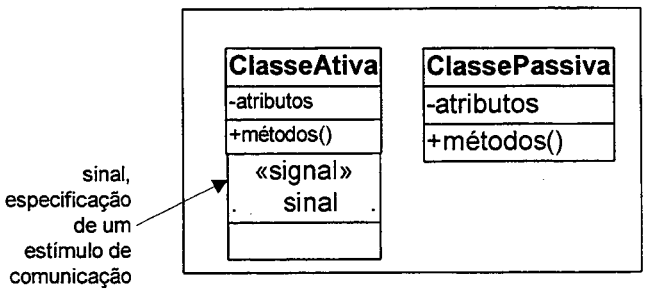

Figura 4.3 – Classes Ativas e Passivas na UML

Em um sistema orientado a objetos, as interações[12] entre objetos indicam de que maneira o fluxo de controle ocorre dentro do sistema. Quando um fluxo passa por uma operação, dizemos que, naquele momento, o *locus* do controle é a operação. Se essa operação é definida para alguma classe, então, naquele instante, o *locus* do controle é uma instância daquela classe. Nesse sentido, é possível ter múltiplos fluxos de controle atuando em um mesmo objeto ou diferentes fluxos de controle em vários objetos. Em um ambiente concorrente vários fluxos de controle agem sobre um mesmo objeto simultaneamente. O desafio é gerenciar os processos, evitando que um interfira na execução do outro, além de incorporar algumas propriedades de sincronização às operações da classe envolvida, evitando assim que o estado do objeto compartilhado seja corrompido.

A maioria das linguagens de programação não permite que se especifique atividades concorrentes. Diferente de linguagens *monothreads* como C e C++ que utilizam as primitivas de *multithreading* do sistema operacional, Java inclui primitivas de *multithreading* dentro da própria linguagem. Utilizando a classe *Tread* da API de Java, o programador define quais as aplicações que terão fluxos de controle independentes [DEI2001].

No sistema GVD, sempre que um cliente se conectar ao servidor, será criada uma sessão para o usuário. Através dessa sessão, o servidor receberá e enviará mensagens ao cliente. Se vários clientes se conectarem simultaneamente, o servidor deverá gerenciar o atendimento às várias sessões de forma a garantir uma perfeita sincronização na execução das operações.

---

[12] No caso de objetos ativos, para auxiliar na sincronização das interações entre fluxos independentes, as mensagens devem ser estendidas com semânticas especiais de concorrência.

Para gerenciar a sincronização, a UML sugere 3 abordagens [BOO1999]:

- Seqüencial: o objeto deve ser coordenado externamente de maneira a garantir que haja apenas um fluxo atuando sobre ele de cada vez. Na presença de múltiplos fluxos, não há garantia de integridade do objeto.

- Prudente: a integridade do objeto é garantida através de chamadas seqüenciais para todas as operações críticas do objeto. Como somente uma operação pode ser invocada de cada vez, essa abordagem passa a ser tratada como seqüencial;

- Concorrente: as operações sobre o objeto são tratadas como operações atômicas.

Essas abordagens são diretamente suportadas por Java através do uso de monitores e da propriedade *synchronized* [DEI2001]. Todo objeto com métodos *synchronized* é um monitor. O monitor permite que um *thread* de cada vez execute um método *synchronized* sobre o objeto. Quando um método *synchronized* é invocado o objeto é bloqueado. Dessa maneira, todos os outros *threads* que tentam invocar métodos *synchronized* devem aguardar. Quando o método *synchronized* termina, o objeto é desbloqueado e o monitor permite que o *thread* de prioridade mais alta tente invocar um método *synchronized* para prosseguir.

4.5 Aspectos Gerais do Desenvolvimento do GVD

No desenvolvimento de um sistema, geralmente o projetista segue um conjunto de etapas onde as atividades são realizadas seqüencialmente:

- Análise dos requisitos

Compreende a fase de entendimento do problema. Estuda as reais necessidades do usuário, estabelece o escopo do sistema e cria um modelo organizacional onde estão incluídos os atores mais importantes e suas respectivas dependências.

- Desenvolvimento dos programas

Utilizando as informações coletadas, o design da arquitetura é planejado e construído;

- Implantação do software

O sistema é testado e disponibilizado para o usuário.

No desenvolvimento do sistema GVD, no entanto, seguiu-se uma orientação diferente. Na análise dos requisitos, a ênfase maior foi dada ao problema: disponibilizar um meio de comunicação e informação entre os usuários. Entre as diversas funções que poderiam ser oferecidas ao usuário selecionou-se apenas um grupo delas, mas que são suficientemente expressivas para o problema abordado. Paralelamente, um protótipo da aplicação foi sendo desenvolvido e testado.

O protótipo da aplicação serviu a dois propósitos. Primeiro, proporcionou uma plataforma de estudo e experimentação para as ferramentas necessárias ao desenvolvimento do GVD. Segundo, permitiu que o desenvolvimento fosse inicialmente concentrado nos aspectos usuários (funcionalidades). Nessa etapa, considerou-se uma arquitetura provisória para o subsistema servidor. Os aspectos do subsistema servidor são considerados no próximo capítulo.

4.6 Análise de Requisitos

Durante a análise dos requisitos, buscou-se definir o que o usuário desejaria do software. Dessa maneira, o levantamento das informações considerou apenas os aspectos funcionais, serviços que o sistema disponibilizará ao usuário.

Requisitos Funcionais

Dois tipos de usuários poderão utilizar o sistema: o aluno e o professor.

1) Aluno

Estará habilitado para as seguintes funcionalidades:

- Consulta as disciplinas nas quais está matriculado;
- Consulta avisos disponibilizados no mural virtual da disciplina;
- Acessa mensagens enviadas pelo professor;
- Copia arquivos disponibilizados pelo professor;
- Envia mensagens para o professor da disciplina.

2) Professor

Estará habilitado para as seguintes funcionalidades:

- Cria o ambiente de interação para as disciplinas sob sua responsabilidade;
- Consulta e envia avisos para disciplinas sob sua responsabilidade;
- Acessa mensagens enviadas pelos alunos;
- Disponibiliza arquivos para *downloads;*
- Envia mensagens para um aluno específico ou para todos os alunos de uma disciplina.

4.7 Construindo o Modelo Conceltual

A razão fundamental de se trabalhar no modelo de um sistema é que assim podemos entender melhor o projeto a ser desenvolvido [COA1993]. A escolha do modelo tem uma profunda influência em como um problema é encarado e qual a solução escolhida. O modelo correto poderá ilustrar brilhantemente a situação e indicar qual a melhor estratégia a ser implantada [BOO1999].

Dentre os vários elementos de descrição oferecidos pela notação UML, alguns foram utilizados para a especificação do modelo. São eles:

- Casos de Uso;
- Descrição dos Casos de Uso;
- Diagrama de Casos de Uso;
- Diagrama de Classes;
- Diagramas de Interação.

4.7.1 Casos de Uso

Casos de Uso são usados para definir os requisitos externos ou para especificar os serviços disponibilizados por um sistema [SCH1999]. Casos de uso podem ser utilizados desde a fase de análise até os testes finais de um software. Através de casos de uso podemos esclarecer quais os serviços realmente necessários e de que maneira serão visualizados no sistema. Desse modo, ajudam a delimitar o escopo do sistema.

Sistemas não existem isoladamente. Todos os sistemas sofrem influência externa ou interagem com alguma fonte de eventos, que é chamada de ator. Um ator pode ser um ser humano, uma máquina, um outro software ou qualquer dispositivo que dispare uma ação no contexto do sistema e receba uma resposta [FOW2000].

No sistema GVD, existem dois atores e vários casos de uso. A F4.1 mostra os atores e as possíveis funções do sistema. Cada função é passível de se transformar em um caso de uso.

Aluno	Abre Sessão Seleciona Disciplina Consulta Avisos Acessa Correio Copia Arquivos
Professor	Abre Sessão Cria Disciplina Seleciona Disciplina Consulta Avisos Envia Avisos Acessa Correio Disponibiliza Arquivos

Tabela 4.1 Descrição das Funcionalidades do GVD

A Fig. 4.4 ilustra a interação entre o sistema e seus atores.

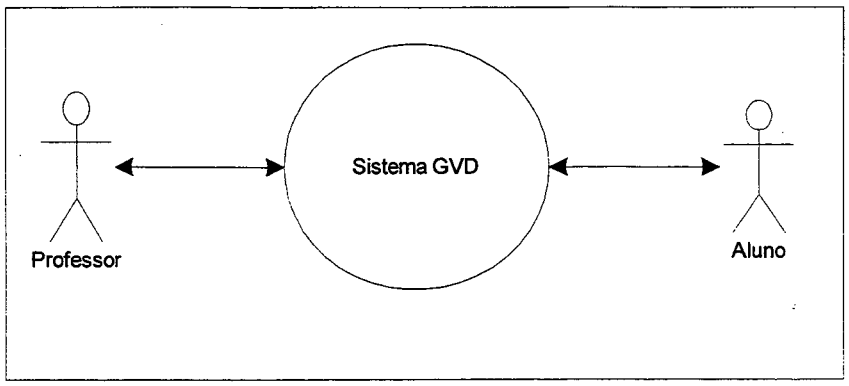

Figura 4.4 – Relacionamento entre o Sistema e seus Atores

Dentro da UML, os atores e os casos de uso podem ser agrupados em um Diagrama de Casos de Uso, de modo que seja possível visualizar o relacionamento entre eles [FOW2000].

A Fig. 4.5 apresenta o Diagrama de Casos de Uso do sistema GVD. Nesse diagrama são apresentados os atores do sistema e seus relacionamentos com cada funcionalidade do sistema.

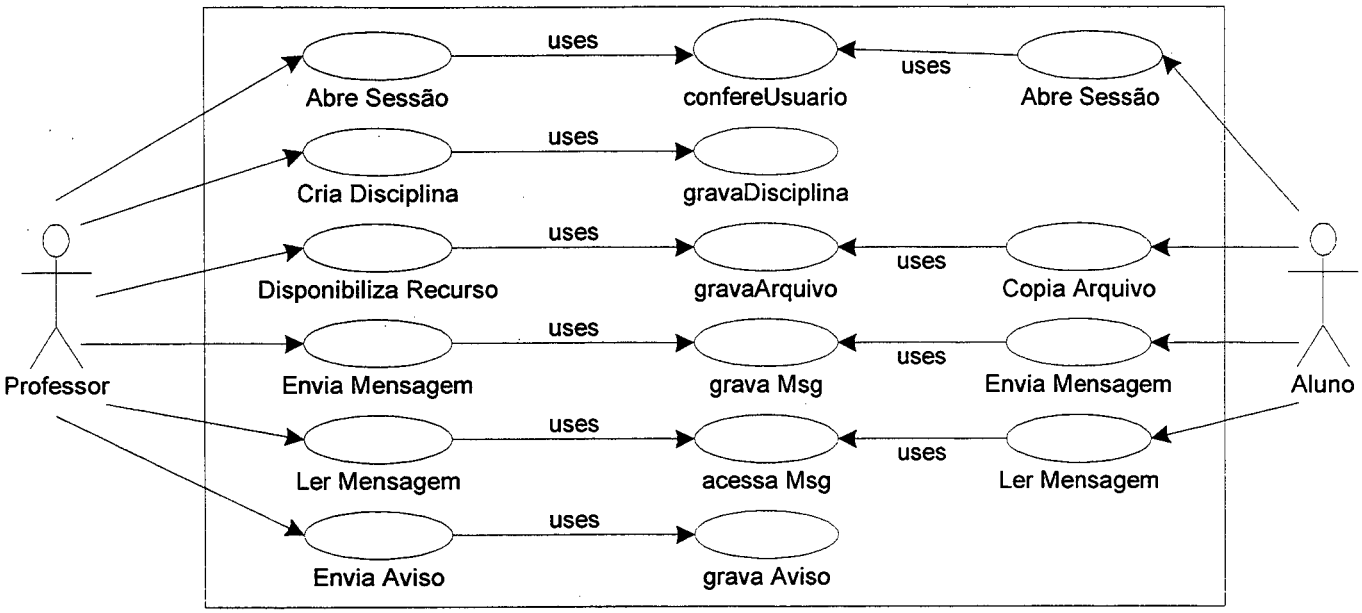

Figura 4.5 – Diagrama de Casos de Uso do Sistema GVD

Cada caso de uso descreve uma funcionalidade do sistema. Essas funcionalidades podem ser entendidas através de cenários. Um cenário é um caminho através do caso de uso, de modo que possamos especificar claramente quais os passos necessários para a execução de um serviço [SCH1999].

Nesse trabalho, iremos descrever os casos de uso através de cenários. Para isso, iremos considerar dois tipos de cenários:

- Cenário primário

  Descreve as funcionalidades básicas do caso de uso. Existe um cenário primário para cada caso de uso.

- Cenário secundário

  Descreve um caminho alternativo ou condições de exceção. Podem existir vários cenários secundários para cada caso de uso.

  Na sequência, apresenta-se a descrição de cada caso de uso através de cenários.

Caso de Uso Abre Sessão

Cenário Primário:

Pré-condição: o usuário que entra no sistema é válido.

1) O sistema estabelece conexão com o subsistema Servidor;

2) O sistema apresenta uma caixa de diálogo para o usuário entrar com o seu código de acesso e a sua senha;

3) O usuário digita as informações;

4) O sistema verifica se o código de acesso é válido e se a senha está correta;

5) O sistema verifica se o usuário é aluno ou professor;

6) O sistema apresenta uma caixa de diálogo com uma grade de disciplinas que depende do tipo de usuário;

7) A grade de disciplinas do professor permite criar nova disciplina;

8) O usuário seleciona a disciplina com a qual vai trabalhar;

9) O sistema apresenta a interface gráfica principal do sistema, de acordo com a função do usuário, disponibilizando as informações sobre a disciplina selecionada.

Cenários Secundários:

1) Não foi possível estabelecer conexão com o Servidor: o programa é derrubado;

2) O usuário é inválido ou a senha é incorreta: o sistema apresenta uma mensagem de acordo com a inconsistência e nega acesso ao sistema;

3) O usuário opta por sair: o sistema encerra o programa normalmente.

Caso de Uso Cria Disciplina

Cenário Primário:

Pré-condição: disponível somente para o usuário professor.

1) O usuário escolhe criar disciplina;

2) O sistema apresenta uma caixa de diálogo onde deverão ser digitados o código e o nome da disciplina que o professor quer criar;

3) O usuário digita as informações;

4) O sistema cria um diretório para a respectiva disciplina;

5) O sistema armazena os dados da nova disciplina;

6) O sistema apresenta a interface gráfica principal do sistema atualizando o cabeçalho com o código e o nome da nova disciplina.

Cenários Secundários:

1) O usuário não digita o código ou o nome da disciplina: o sistema apresenta uma mensagem indicando o erro.

2) O usuário opta por retornar: o sistema finaliza o diálogo Criar Disciplina e volta à interface principal.

Caso de Uso Disponibiliza Recurso

Cenário Primário:

Pré-condição: disponível somente para o usuário professor.

1) O usuário seleciona disponibilizar recurso;

2) O sistema apresenta uma caixa de diálogo onde deverá ser digitado o URL onde reside o arquivo a ser disponibilizado;

3) O usuário digita as informações;

4) O sistema acessa o arquivo através do *Uniform Resource Locator* (URL) e grava o arquivo no diretório da respectiva disciplina;

5) O sistema apresenta uma mensagem de operação concluída com sucesso ao professor.

Cenários Secundários:

1) O usuário não digita o URL: o sistema apresenta uma mensagem indicando o erro.

2) O arquivo não existia no local indicado: o sistema apresenta uma mensagem indicando o erro.

3) O usuário opta por retornar: o sistema finaliza o diálogo de disponibilizar recursos e volta à interface principal.

Caso de Uso Envia Mensagem

Cenário Primário:

1) O usuário seleciona enviar mensagem;

2) O sistema apresenta uma caixa de diálogo onde deverão ser digitadas as informações tais como: destinatário, assunto, mensagem e remetente;

3) O usuário digita as informações;

4) O sistema verifica as informações e armazena os dados da mensagem;

5) O sistema retorna ao passo 2.

Cenários Secundários:

1) O usuário deixa de preencher todas as informações: o sistema apresenta uma mensagem indicando o erro.

2) O usuário opta por retornar: o sistema finaliza o diálogo de enviar mensagens e volta à interface principal.

Caso de Uso Ler Mensagem

Cenário Primário:

1) O usuário seleciona ler mensagem;

2) O sistema apresenta uma caixa de diálogo com a lista de mensagens destinadas ao usuário, onde constará o remetente, o assunto e a data da mensagem;

3) O usuário seleciona uma mensagem;

4) O sistema apresenta o texto da mensagem.

Cenários Secundários:

1) O usuário não recebeu mensagens: o sistema indica que não há mensagens;

2) O usuário opta por retornar: o sistema finaliza o diálogo de enviar mensagens e volta à interface principal.

Caso de Uso Envia Aviso

Cenário Primário:

Pré-condição: o usuário é professor.

1) O usuário escolhe enviar aviso;

2) O sistema apresenta uma caixa de diálogo onde deverá ser digitado o texto do aviso;

3) O usuário digita as informações;

4) O sistema armazena os dados do novo aviso;

5) O sistema se prepara para apresentar incondicionalmente o aviso a qualquer aluno;

6) O sistema apresentará o mesmo aviso até que o professor limpe a área do texto ou até que seja digitado um novo aviso.

Cenários Secundários:

1) O usuário deixa de preencher o texto do aviso: o sistema apresenta uma mensagem indicando o erro.

2) O professor optou por aviso geral: o aviso será apresentado em todas as disciplinas ministradas pelo professor.

5) O professor opta por retornar: o sistema finaliza o diálogo de enviar aviso e volta à interface principal.

Caso de Uso Copia Arquivo

Cenário Primário:

Pré-condição: o usuário é aluno.

1) O usuário escolhe copiar arquivo;

2) O sistema apresenta uma caixa de diálogo com uma lista de arquivos disponíveis para aquela disciplina;

3) O usuário seleciona um dos arquivos;

4) O sistema apresenta uma nova caixa de diálogo tipo FileChoose através da qual o aluno indicará o diretório em que o arquivo deverá ser copiado;

5) O sistema acessa o arquivo no diretório da disciplina e o copia para o diretório indicado pelo aluno;

6) O sistema indica que a operação foi realizada com sucesso.

Cenários Secundários:

1) O usuário opta por retornar ou cancelar: o sistema finaliza o diálogo e volta à interface principal.

4.7.2 Diagrama de Classes do Subsistema Cliente

Um Diagrama de Classes apresenta um conjunto de classes, interfaces, colaborações e seus relacionamentos. Ilustra uma visão estática do sistema. Diagramas de Classes são importantes para visualizar, especificar e documentar o modelo estrutural do sistema. Através do Diagrama de Classes é possível também analisar os requisitos funcionais do sistema [FOW2000]. O Diagrama de Classes geralmente contém: classes, interfaces, colaborações, dependências, generalizações e relacionamentos de associação.

A Fig. 4.6 apresenta o diagrama de classe para o subsistema cliente, onde:

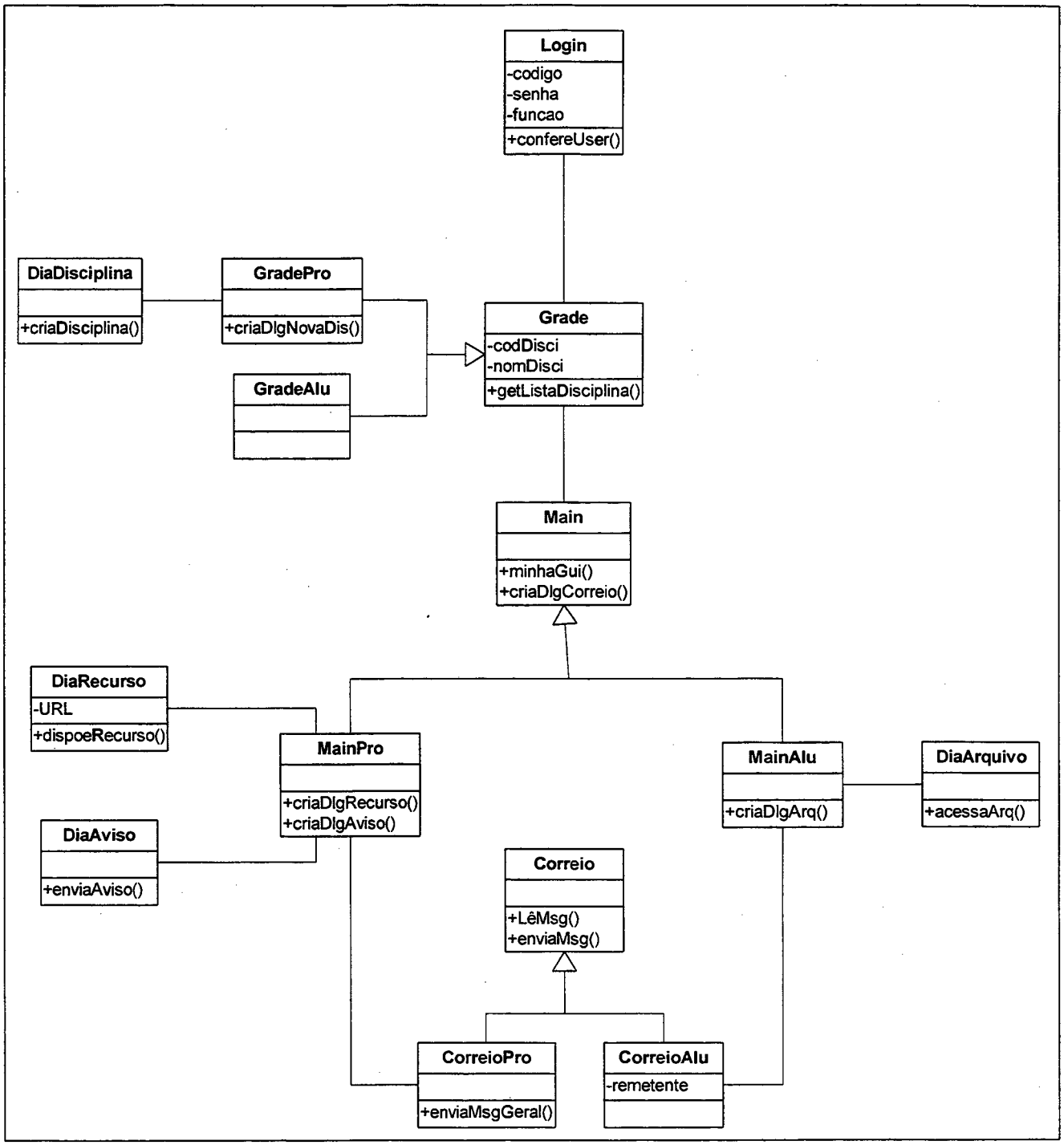

Figura 4.6 – Diagrama de Classes do Subsistema Cliente

- A classe Login controla o acesso do usuário ao sistema. Através do método confereUser, o sistema verifica se o código do usuário é válido e se a senha está correta. No caso afirmativo, o sistema busca as disciplinas relacionadas a esse usuário. Caso contrário, o acesso é negado.

- A classe DiaDisciplina é uma caixa de diálogo responsável pela inclusão de novas disciplinas no sistema. Exclusividade de usuário professor.

- A classe Grade controla a janela através da qual serão apresentadas as disciplinas do usuário. É uma classe abstrata que será realizada pelas subclasses GradePro e GradeAlu. O objeto grade mostrará na tela as disciplinas relacionadas ao usuário.

- A classe Main controla a interface gráfica principal apresentada ao usuário. É uma classe abstrata que deverá ser realizada pelas subclasses MainPro e MainAlu. Ao ser selecionada uma disciplina da grade, um ambiente aluno ou professor será criado de acordo com o tipo de usuário.

- A classe Correio controla a emissão e o recebimento de mensagens.

- A classe DiaRecurso é uma caixa de diálogo que controla o acesso a arquivos através do *Uniform Resource Locator* (URL). O acesso a essa classe só é permitido a usuários professores. Essa classe permite que o professor disponibilize arquivos aos alunos da disciplina.

- A classe DiaAviso é uma caixa de diálogo que controla o envio de avisos do professor a todas as suas disciplinas ou a uma em particular. O aviso será mostrado ao aluno assim que a disciplina for acessada.

- A classe DiaArquivo é uma caixa de diálogo que controla o acesso aos arquivos disponibilizados pelo professor. Permite que um usuário aluno selecione um arquivo e o copie para outro diretório.

4.8 Interfaces Gráficas

O sistema GVD tem diversas interfaces gráficas através das quais os usuários interagem remotamente com o sistema. Inicialmente, conforme Fig. 4.7, é apresentada uma janela através da qual o usuário se identifica.

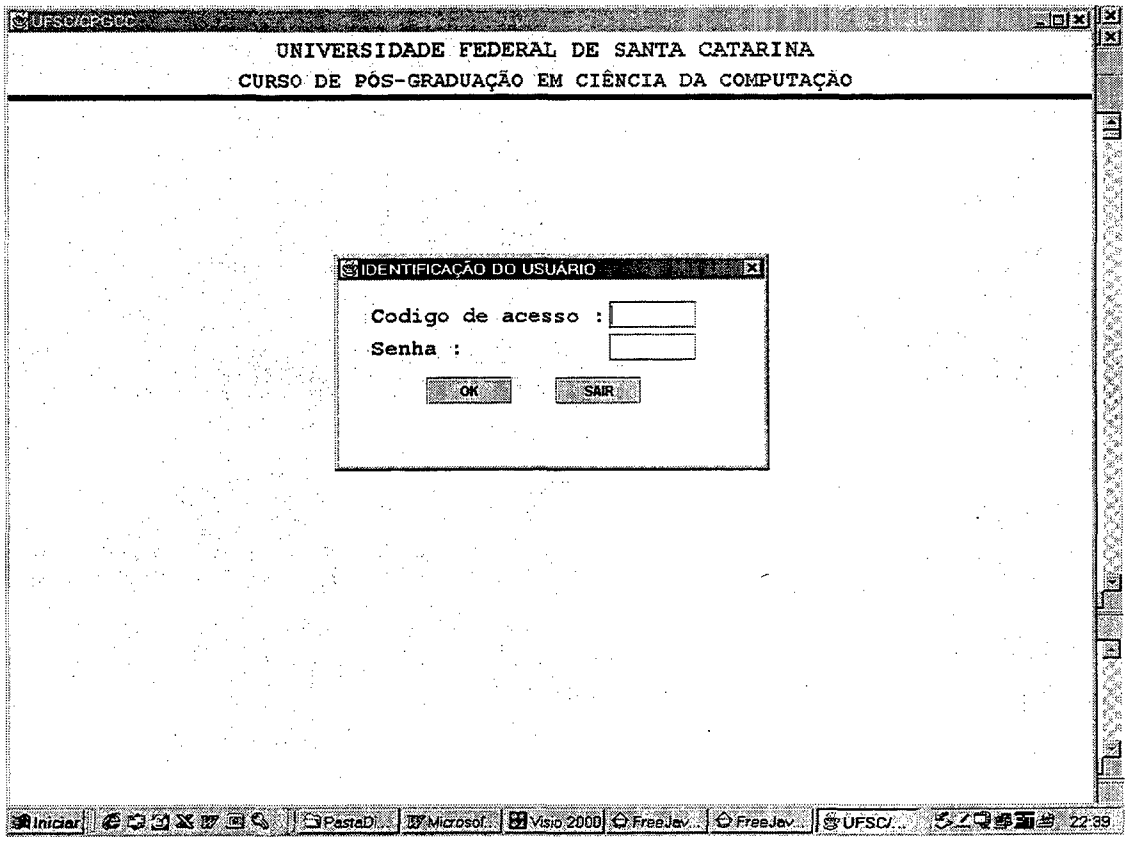

Figura 4.7 – Tela Login do GVD

Após a identificação do usuário, o sistema apresenta uma janela através da qual o usuário seleciona a disciplina a ser consultada. No caso do professor, conforme Fig. 4.8, a janela permite selecionar a disciplina ou cadastrar uma nova.

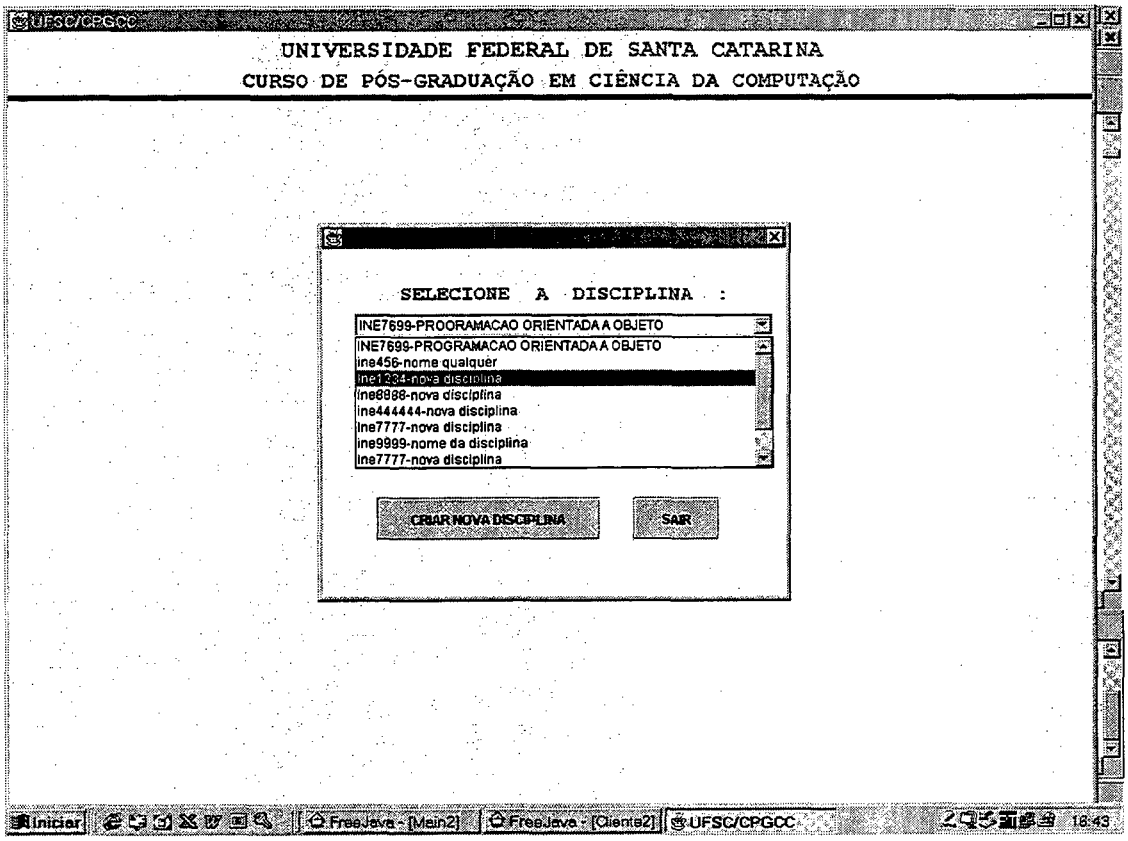

Figura 4.8 – Tela Seleção de Disciplinas do Professor

No caso do usuário ser um aluno, a interface gráfica a ser apresentada após o Login é apresentada na Fig.4.9, através da qual o aluno poderá selecionar a disciplina a ser consultada. A caixa de seleção mostra as disciplinas nas quais o aluno se matriculou.

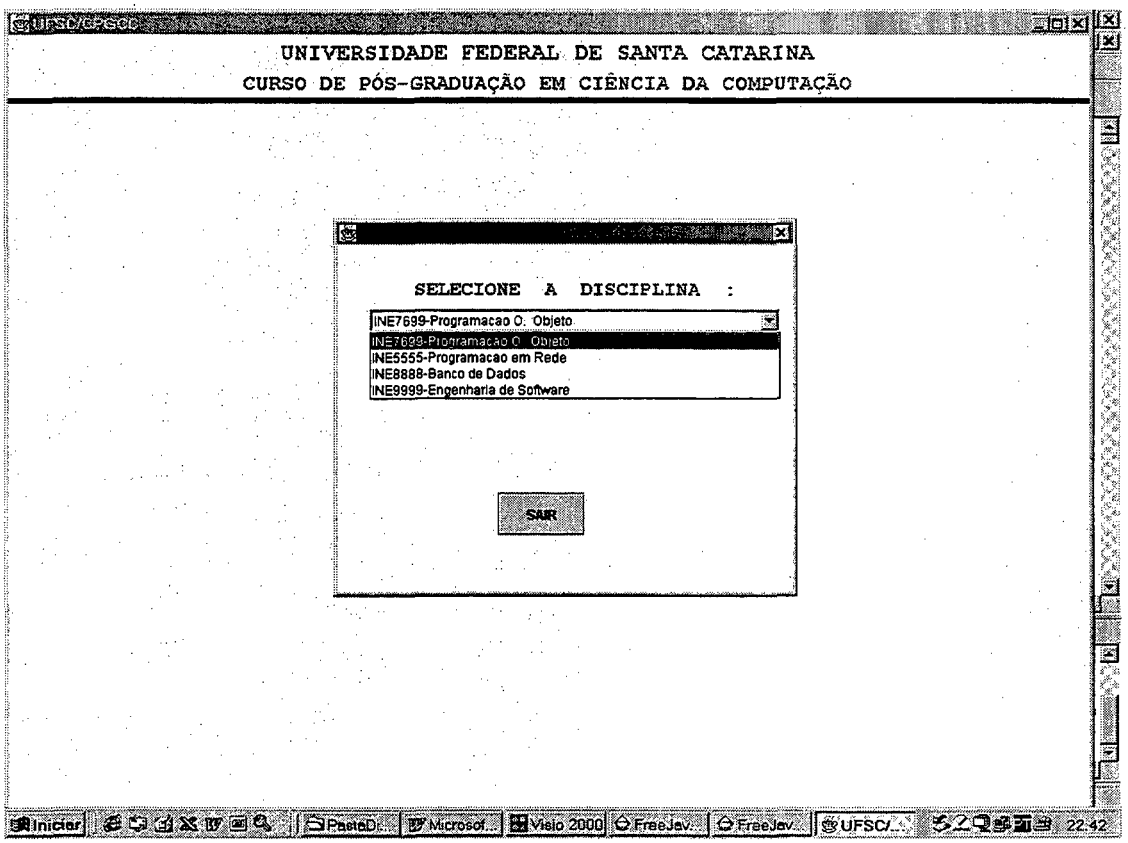

Figura 4.9 – Tela Seleção de Disciplinas do Aluno

Após a disciplina ter sido selecionada, é apresenta a interface gráfica principal através da qual o usuário, aluno ou professor, poderá interagir com o sistema. A Fig. 4.10 apresenta a interface gráfica do aluno. No caso de haver um aviso cadastrado pelo professor, a mensagem é mostrado ao aluno no início do acesso à disciplina.

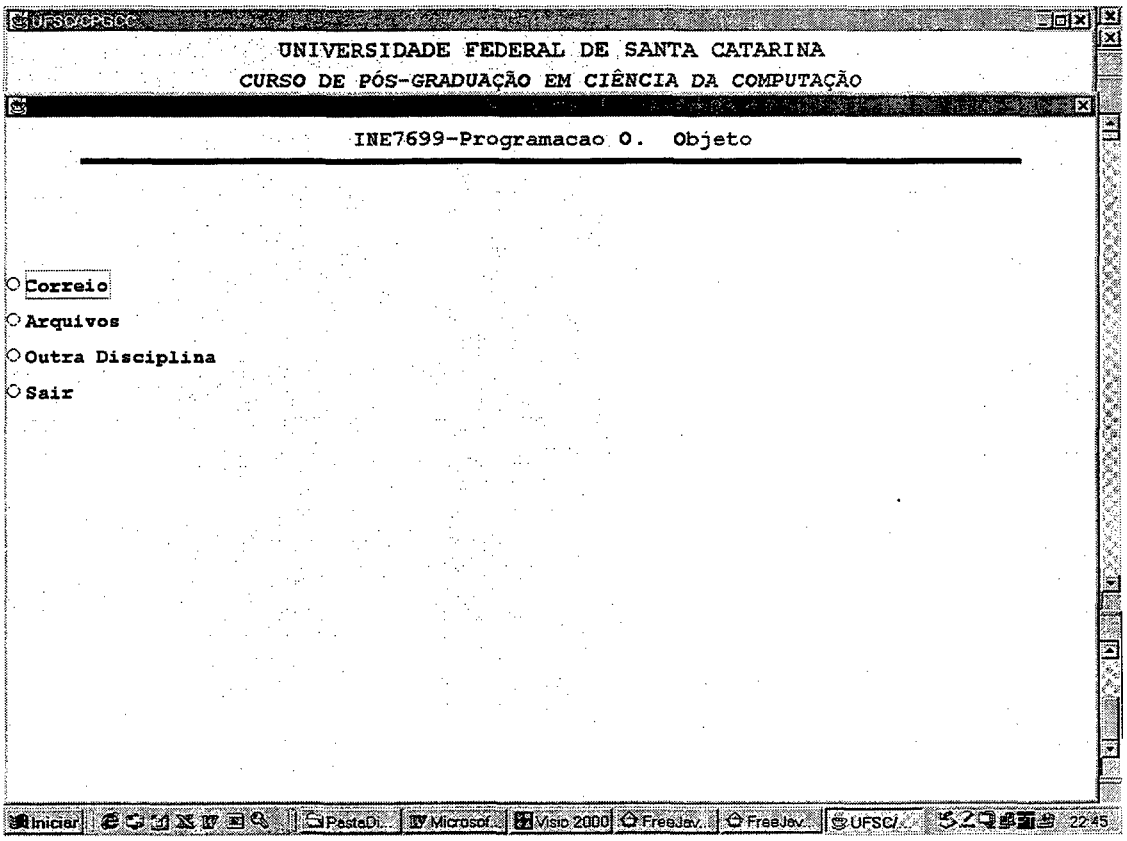

Figura 4.10 – Tela Menu do Aluno

A Fig. 4.11 apresenta a interface gráfica através da qual o professor irá cadastrar as informações, os avisos, disponibilizar recursos (arquivos), ler ou enviar mensagens para os alunos de uma determinada disciplina.

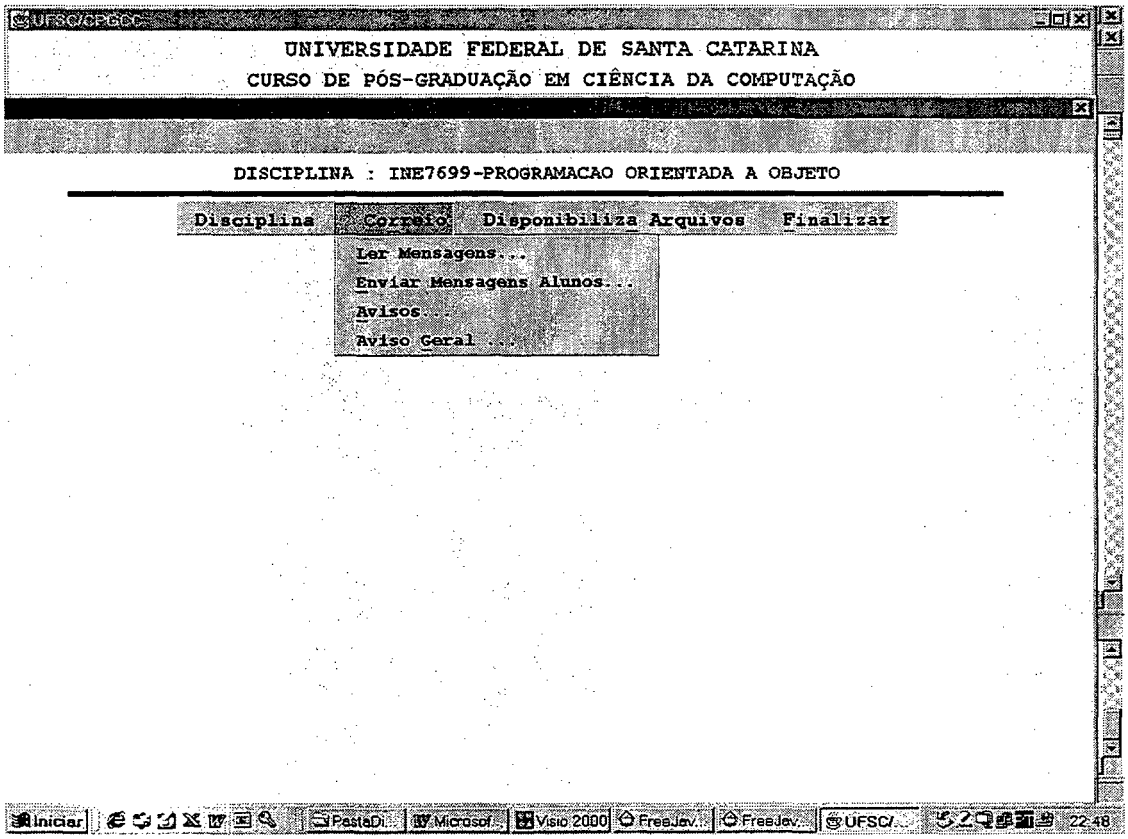

Figura 4.11 – Tela Menu do Professor

4.9 Análise Arquitetural do Subsistema Cliente

Especificar, construir e documentar um sistema de software exige que o sistema seja avaliado sob várias perspectivas [BOO1999]. Nesse sentido, a arquitetura de um sistema é um artefato importante. Através da arquitetura moldamos o sistema sob diferentes pontos de vista. São eles:

- a organização do software;
- a seleção de elementos estruturais e suas interfaces através do qual o sistema é composto;
- o comportamento e as colaborações entre os elementos selecionados;
- a composição estrutural e comportamental dos elementos em subsistemas progressivamente maiores;
- o estilo da arquitetura que guia essa organização: os elementos estáticos, os elementos dinâmicos, suas interfaces, colaborações e composição.

A arquitetura não está apenas ligada ao aspecto estrutural e comportamental do software. Incorpora, também, outros aspectos como usabilidade, funcionalidade, reusabilidade, flexibilidade, desempenho e tecnologia.

O subsistema cliente, por trabalhar com dois tipos de usuários[13], é composto por classes abstratas e suas respectivas classes realizadoras, uma para cada tipo de usuário. O usuário professor, ao abrir uma sessão, é conduzido dentro do programa através das classes servidoras do usuário professor. O aluno, por sua vez, não tem acesso ao mesmo ambiente do professor. Por isso, é conduzido pelas classes servidoras do usuário aluno.

A vantagem desse tipo de arquitetura é a facilidade de, no futuro, poder incluir, retirar ou modificar usuários. Apenas as respectivas classes de realização serão afetadas. A estrutura do sistema permanece inalterada.

---

[13] Usuário professor e usuário aluno, com funções e visões diferenciadas dentro do sistema.

A Fig. 4.12 apresenta o Diagrama de Fluxo de Atividades do ambiente Cliente. O fluxo de controle do programa é conduzido com base no tipo de usuário, identificado na etapa Login.

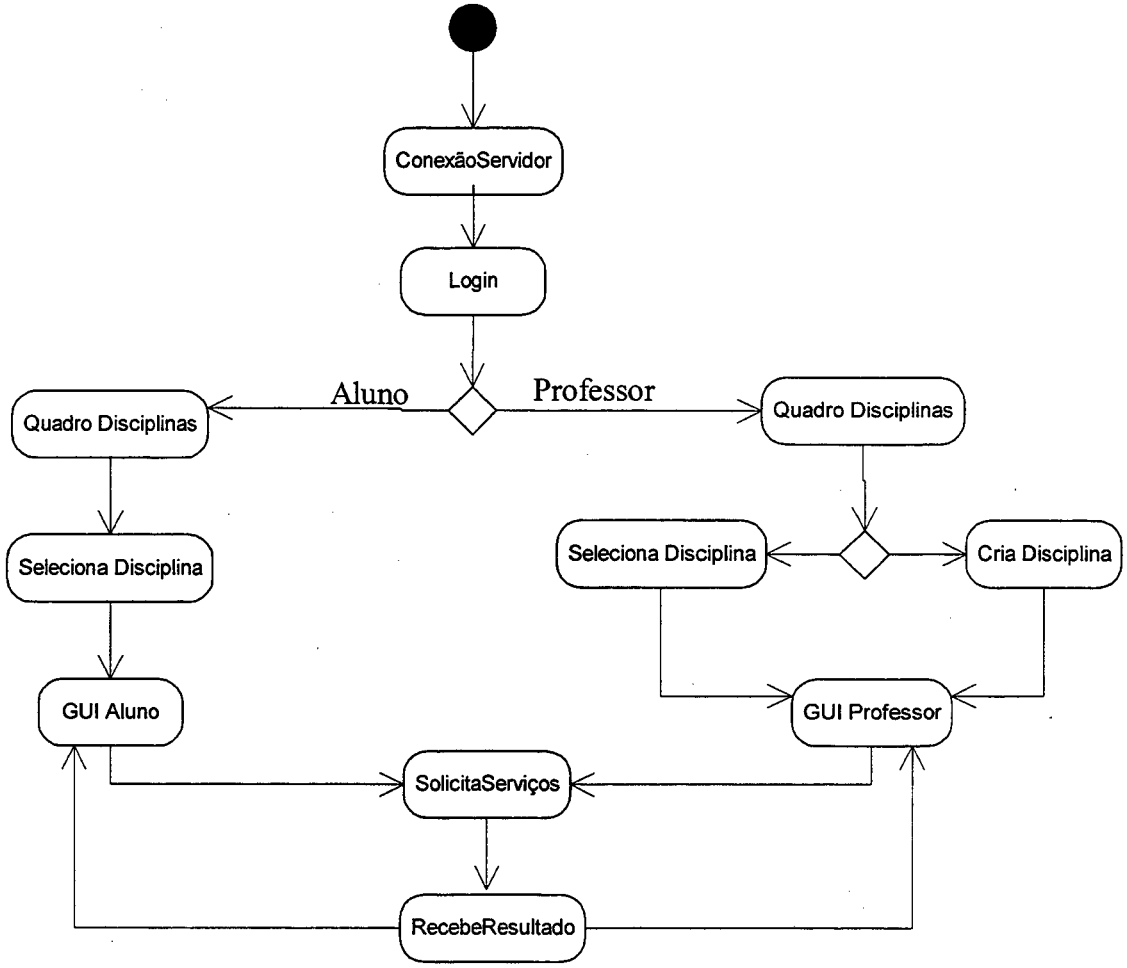

Figura 4.12 – Diagrama do Fluxo de Atividades do Subsistema Cliente

4.10 Conclusão

Neste capítulo foi apresentado o desenvolvimento do protótipo de uma aplicação cliente/servidor que permite a comunicação entre usuários conectados a uma rede. Durante a fase de análise, foram utilizadas as técnicas da UML para a construção do Modelo Conceitual e para definição do escopo do sistema. Na construção dos programas foi utilizado o ambiente de programação Java oferecido pela *Sun Microsystems.*

A aplicação desenvolvida é um Gerenciador Virtual de Disciplinas, através da qual alunos e professores trocam informações a respeito de uma ou várias disciplinas. O sistema é composto por um subsistema cliente e um subsistema servidor. Através de uma interface gráfica vários usuários podem se conectar simultaneamente ao servidor.

Neste capítulo, considerou-se principalmente o aspecto usuário do GVD, no sentido de garantir sua funcionalidade. A arquitetura do subsistema servidor será apresentada no próximo capítulo.

5. A ARQUITETURA DO SERVIDOR

5.1 Introdução

Este capítulo dedica-se a apresentação da arquitetura do subsistema servidor do Gerenciador Virtual de Disciplinas apresentado no Capítulo 4. Diferente do subsistema cliente, mais simples, o subsistema servidor incorpora características de um típico sistema complexo (Capítulo 3, Seção 3.2). Por isso e também devido as razões apontadas no estudo sobre agentes apresentado no Capítulo 3, optou-se por uma arquitetura orientada a agentes para o subsistema servidor.

O conceito de agentes ainda é motivo de discussão dentro da Engenharia de Software. Existe uma variedade de definições, cada uma procurando a melhor explicação para a palavra agente dentro da computação. Em [FRA1996] encontramos uma dúzia de definições, indicando claramente que não existe um consenso geral sobre o que constitui um agente.

Na arquitetura proposta nesse capítulo utilizaremos a idéia mais próxima possível do conceito de agentes de software. Com isso, excluem-se as características de inteligência, emoção, utilização de linguagens de comunicação e outras habilidades advindas do conceito de agentes da Inteligência Artificial. Os agentes construídos nesse trabalho comportam-se como componentes autônomos compostos por um ou mais objetos, com algumas funções internas executadas automática e voluntariamente.

Neste trabalho, os agentes são classificados de acordo com a tarefa que executam, e compreendem dois grupos: agentes gerenciadores e agentes funcionais. Os agentes do primeiro grupo são utilizados para gerenciar a sessão criada para atender ao usuário; os do segundo, são acoplados a objetos que executam as operações ligadas ao serviço solicitado pelo usuário. Todos os agentes trabalham cooperativamente para prover da maneira mais eficiente possível os serviços disponibilizados pelo servidor.

5.2 Visão Geral

A arquitetura final proposta para o subsistema servidor cria vários ambientes cooperativos. A junção de todas as classes e suas associações pode ser observada na Fig.5.1:

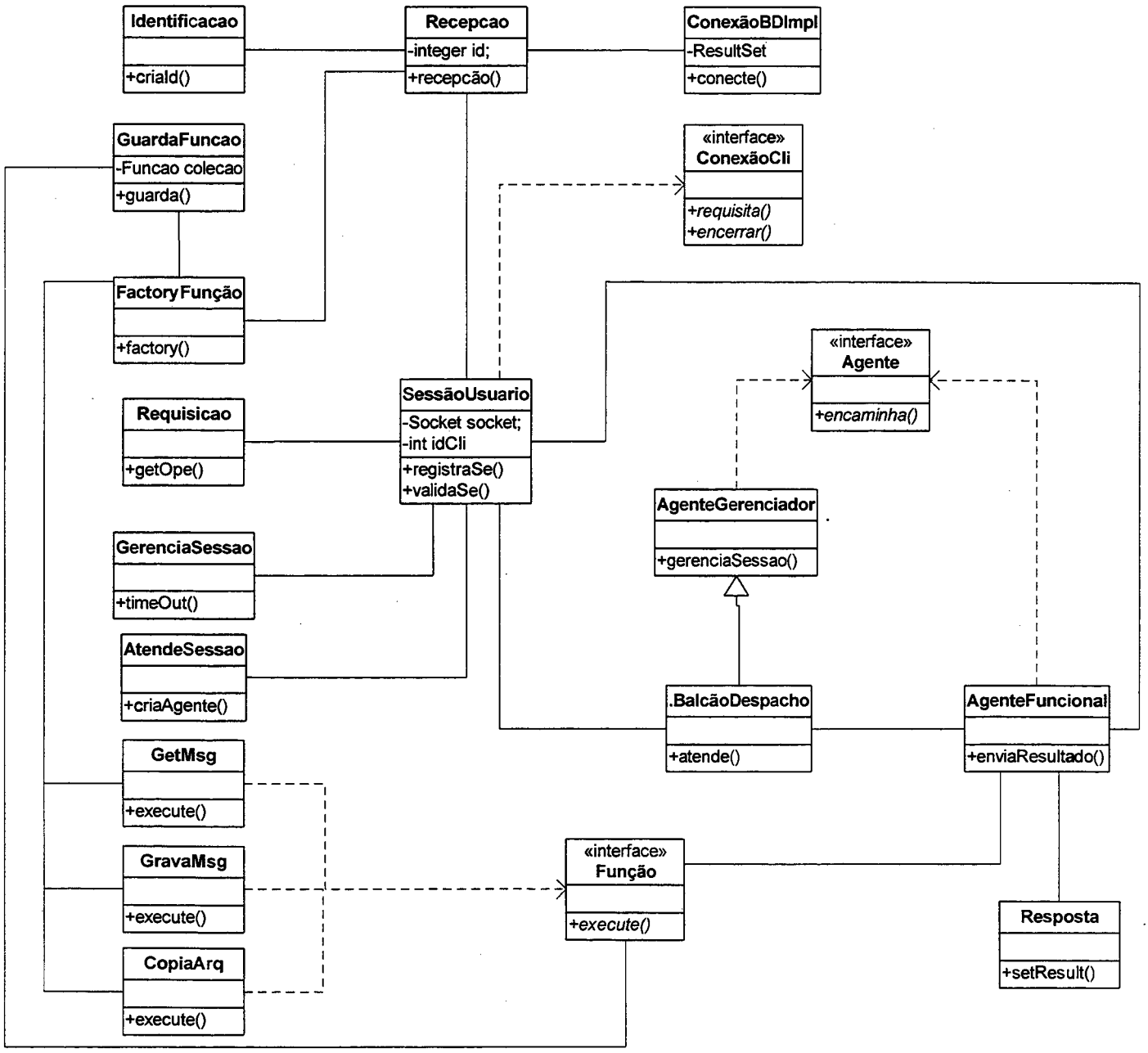

Figura 5.1 – Diagrama de Classes da Arquitetura Proposta para o Ambiente Servidor

Onde as principais classes são:

Recepcão: classe com instância única, responsável pelo recebimento da conexão cliente e criação da sessão usuário;

Identificação: classe com instância única, controla a identificação da sessão usuário;

FactoryFunção: classe responsável por criar todos os objetos Função e armazená-los em em uma instância única da classe GuardaFunção.

GuardaFunção: classe com uma única instância de acesso global, onde serão armazenados uma cópia de cada objeto Função;

SessãoUsuário: classe que implementa a interface ConexãoCli, responsável pela comunicação entre o cliente e o servidor. Será aberta uma sessão usuário para cada conexão cliente. Cria um ambiente onde as requisições do usuário são recebidas e as respostas do servidor são enviadas;

Agente: interface implementada por dois tipos de agentes: gerenciadores e funcionais;

AgenteGerenciador: classe responsável por gerenciar a sessão usuário;

BalcãoDespacho: classe especialização da classe AgenteGerenciador. É responsável por receber as requisições da SessãoUsuário e encaminhá-las para atendimento. É formado por uma instância de AtendeSessão e uma de GerenciaSessão;

AtendeSessão: classe que cria os agentes funcionais sob demanda. O agente funcional é criado uma única vez e utilizado por todos;

GerenciaSessão: controla o timer da sessão usuário (gerencia o *timeout*);

AgenteFuncional: classe que recebe a requisição do BalcãoDespacho e provê a execução do serviço. Envia a resposta direto à sessão usuário;

Função: interface que será implementada pelas classes executoras das operações. Cada objeto Função realiza uma pequena tarefa que, unidas, serão as realizadoras dos serviços solicitados pelo usuário.

5.3 Aspectos Arquiteturais

O subsistema servidor é um ambiente cooperativo composto por um conjunto de agentes locais compartilhando informações e trabalhando para um mesmo objetivo. Gerencia vários clientes conectados à Internet, recebendo solicitações e executando serviços. Em linha geral, compreende os seguintes elementos estruturais:

- Comunicação com o cliente;
- Objetos ativos que geram fluxos de controle independentes;
- Objetos passivos que possuem informações compartilhadas entre vários processos simultaneamente;
- Agentes gerenciadores: agentes autônomos locais que recebem e encaminham as mensagens do cliente;
- Agentes funcionais: agentes incorporados às funções do sistema, especialistas em uma determinada tarefa;
- Um banco de dados que garante a persistência dos dados;
- Mensagens enviadas entre objetos, agentes e o banco de dados.

Cada conexão cliente é acompanhada por um agente gerenciador exclusivo, que recebe as solicitações do cliente, as interpreta e delega aos agentes funcionais a execução das tarefas correspondentes. Nesse contexto, o problema é garantir a execução das tarefas, compartilhando objetos e informações sem corrompê-los, e retornando os resultados de maneira eficiente.

5.3.1 Conexão Cliente

A conexão entre o subsistema cliente e o subsistema servidor será feita através da API Socket de Java, que disponibiliza um canal (*stream*) através do qual os dados fluem entre dois terminais[1] (*stream input* e *stream output*). O cliente envia mensagens através das quais solicita serviços que deverão ser executados pelo servidor. Ao final da execução do serviço, o servidor envia o resultado ao cliente. A conexão será mantida até ser encerrada pelo cliente ou até ser derrubada pelo servidor.

A Fig. 5.2 apresenta a estrutura das classes envolvidas na conexão com o cliente:

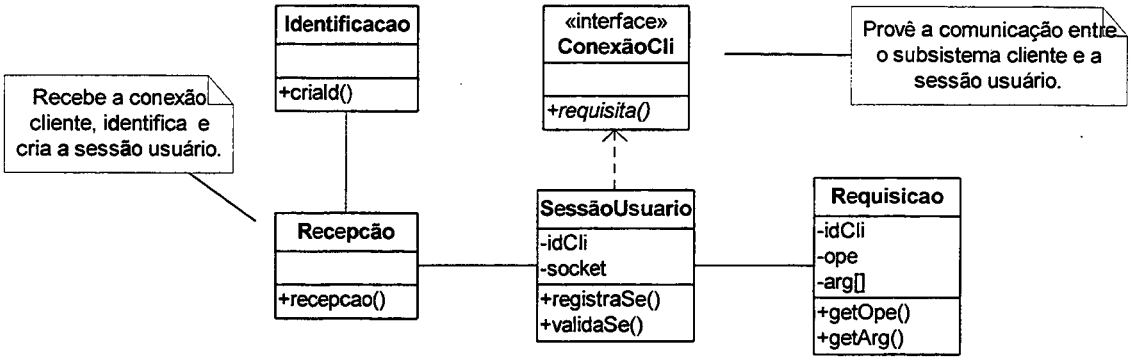

Figura 5.2 – Classes Envolvidas na Conexão e Comunicação com o Cliente

Inicialmente, a conexão com o cliente é estabelecida e identificada, através da classe Identificação. A identificação é um número gerado seqüencialmente e associado à conexão. A seguir, a classe Recepção cria uma sessão usuário, onde os dados do usuário são verificados. Se os dados do usuário forem inconsistentes (código do usuário ou senha inválidos), uma mensagem de erro é retornada ao cliente e a sessão é encerrada. Caso contrário, a sessão gera um processo para atendimento das requisições do usuário.

5.3.2 Mensagens

As mensagens trocadas entre os subsistemas cliente e servidor são instâncias de duas classes: Requisição e Resposta. Ambas são especializações da classe Mensagem. O objeto requisição é gerado a partir das solicitações do cliente para o servidor. O objeto resposta é gerado a partir do resultado das operações executadas no servidor.

A requisição é um objeto que vem do cliente e que contém os seguintes atributos:

- Identificação do usuário;
- Identificação do serviço a ser executado;
- Argumentos necessários à sua execução.

A resposta é um objeto gerado no servidor e que contém os seguintes atributos:

- Identificação do usuário;
- Resultado do serviço solicitado.

A Fig.5.3 apresenta a estrutura da classe Mensagem e suas especializações:

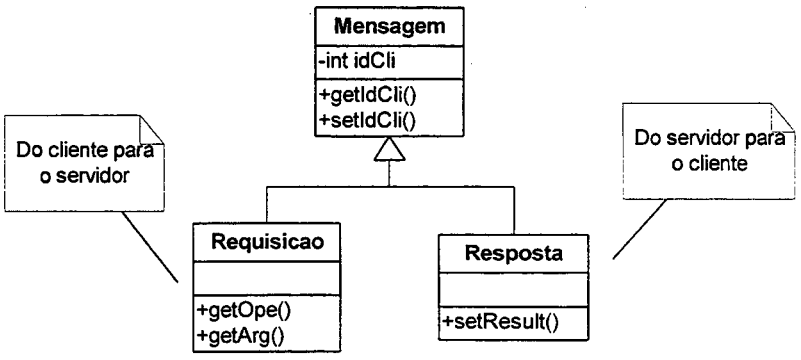

Figura 5.3 – Estrutura da Classe Mensagem

5.3.3 Sessão Usuário

Para cada conexão cliente aceita, será criada uma sessão usuário. A sessão usuário será mantida até que o subsistema cliente se desconecte ou até que a conexão seja derrubada pelo servidor por *timeout* (se o usuário abrir uma sessão e não enviar mensagens durante um certo período, o próprio sistema encerra a sessão).

A SessãoUsuario é uma classe ativa. Isso significa que seus objetos irão gerar fluxos de controle independentes. Cada objeto sessão gera um processo autônomo, com vida própria e que compartilha ou disputa recursos com outros objetos sessão no ambiente servidor.

A Fig.5.4 apresenta um esquema geral do ambiente servidor com várias sessões usuários estabelecidas:

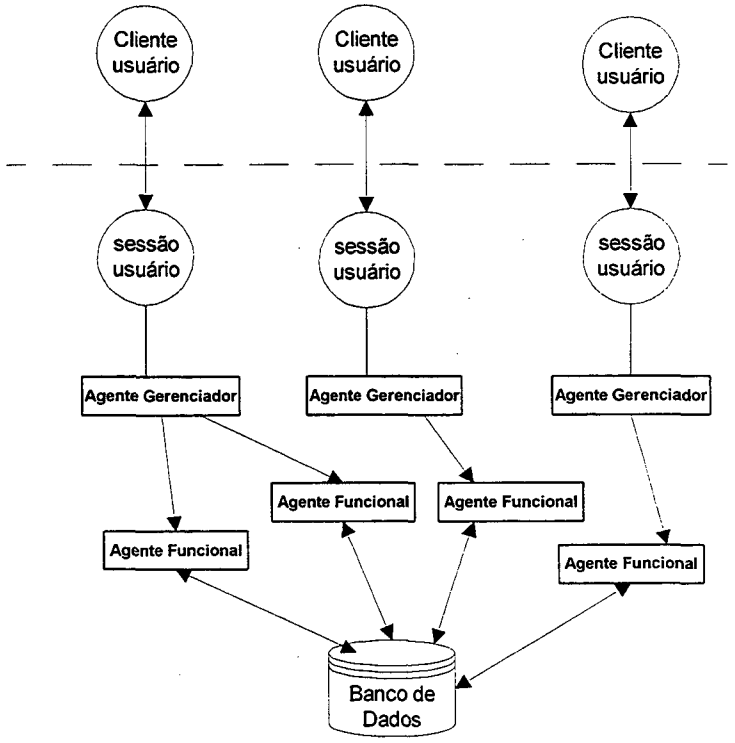

Figura 5.4 – Sessões Usuários Estabelecidas no Servidor

Ao ser instanciada, a sessão usuário cria um ambiente dentro do qual vários objetos passivos serão utilizados para o seu próprio controle. Esses objetos contém dados que interessam somente à própria sessão e que, portanto, não devem ser compartilhados. Por exemplo, o atributo `timer` de uma sessão está dentro de um objeto gerente que controla o *timeout* da sessão. Como cada sessão é independente, cada uma necessita de um gerente próprio para controle de seu *timeout*.

Assim, uma sessão usuário é um ambiente independente, ligado a um *thread*, e composto pelos seguintes elementos:

- Um agente gerenciador, composto por um objeto AtendeSessão e um objeto GerenciaSessão. Esse agente carrega na sua referência a identificação da sessão a que serve;

- Objetos passivos para controle próprio e não compartilhados. Da mesma forma que o agente gerenciador, objetos passivos possuem a identificação da sessão usuário que os utiliza.

A sessão usuário pode, ainda, criar outros elementos:

- Objetos passivos criados pela primeira sessão usuário que se estabelecer e que podem ser compartilhados por outras sessões. São objetos que contém informações de interesse geral e que podem ser acessados simultaneamente por várias sessões;

- Objetos passivos criados sob demanda por uma sessão e disputados posteriormente por todas. Esses objetos irão compor os agentes funcionais que atenderão a todas as sessões sincronizadamente.

A Fig. 5.5 apresenta as principais classes envolvidas na criação da sessão usuário e no encaminhamento das requisições:

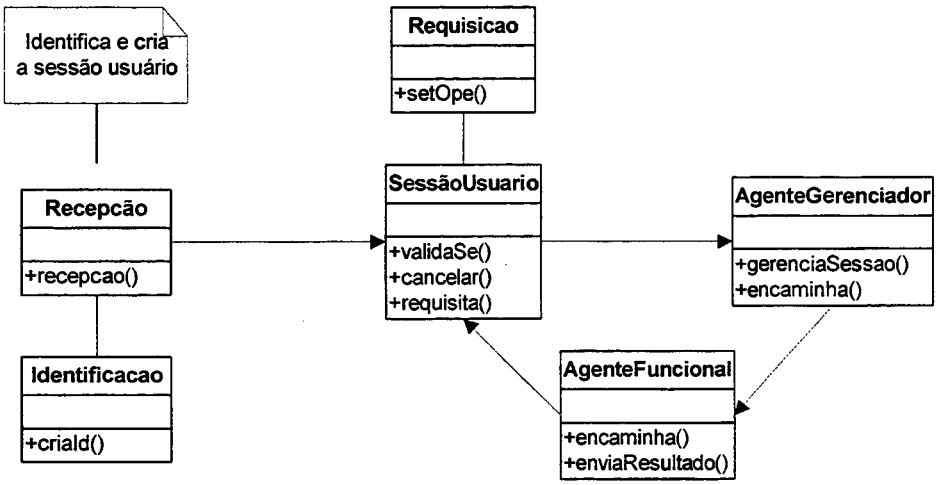

Figura 5.5 – Criação e Atendimento da Sessão Usuário com Navegabilidade

Após sua criação, a própria sessão usuário valida-se, conferindo os dados do usuário. No caso de usuário inválido, o acesso ao sistema é negado e a sessão é interrompida. Caso contrário, a sessão usuário é associada a um agente gerenciador, que gerencia a sessão, recebe as requisições do usuário e as encaminha para serem atendidas por um agente funcional especializado. O agente funcional provê a execução do serviço e retorna o resultado à sessão usuário.

5.3.4 Agentes

No sistema em estudo há basicamente dois tipos de agentes interagindo e compartilhando objetos e dados: agentes gerenciadores e agentes funcionais. O objetivo comum é garantir a execução dos serviços solicitados pelos usuários distribuídos ao longo da Internet.

A Fig. 5.6 apresenta a estrutura das classes tipo Agente no subsistema servidor:

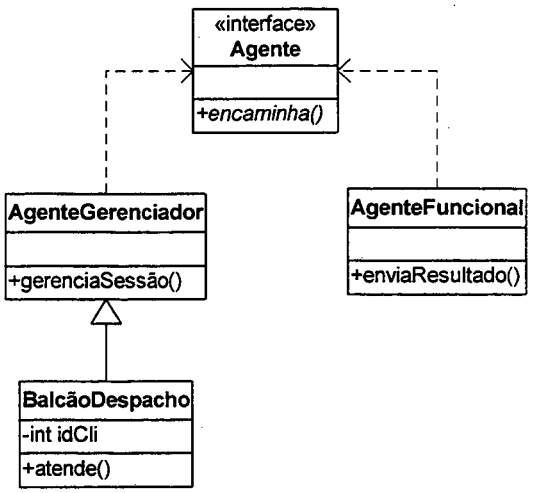

Figura 5.6 – Estrutura das Classes Agente

Agentes Gerenciadores

Agentes Gerenciadores são componentes que gerenciam os eventos da sessão usuário. Cada sessão usuário tem um agente gerenciador. O agente gerenciador é responsável por receber as requisições do usuário e garantir que as mesmas sejam encaminhadas para execução.

Inicialmente, o agente gerenciador associado à sessão usuário é o BalcãoDespacho. O BalcãoDespacho é um agente especializado em receber as solicitações da sessão usuário e encaminhá-las para o respectivo agente funcional. É composto por um objeto atendente e um objeto gerente. O atendente verifica qual é o agente funcional especializado no serviço solicitado pelo usuário. O gerente é acionado no intervalo entre o recebimento das mensagens, de modo a controlar o uso efetivo do sistema através de um *timer*. Se o *timer* acusar um determinado tempo sem mensagens, o gerente aciona o cancelamento da sessão por *timeout*. O agente BalcãoDespacho, então, encaminha uma mensagem ao usuário e cancela a sessão ao qual está conectado.

Agentes Funcionais

Agentes Funcionais são componentes especialistas na execução dos serviços disponibilizados pelo servidor. Os agentes funcionais são fortemente acoplados a objetos do tipo Função[14]. Podem ser compostos por um ou mais objetos, dependendo da operação a ser realizada. O agente funcional recebe a requisição do agente gerenciador e delega a execução da operação ao respectivo objeto Função. Ao final da execução da operação, o agente funcional gera um objeto resposta com o resultado do serviço e o envia à sessão usuário.

---

[14] Função é uma interface implementada por diversas classes. Essas classes descrevem as operações necessárias para a execução das tarefas.

A Fig.5.7 apresenta o diagrama de seqüência da sessão usuário e a interação com os agentes. Através desta figura, pode-se observar que um agente gerenciador é registrado para receber e encaminhar as solicitações do usuário para o respectivo agente funcional. O agente funcional delega a execução da operação a um objeto específico e retorna o resultado diretamente ao usuário.

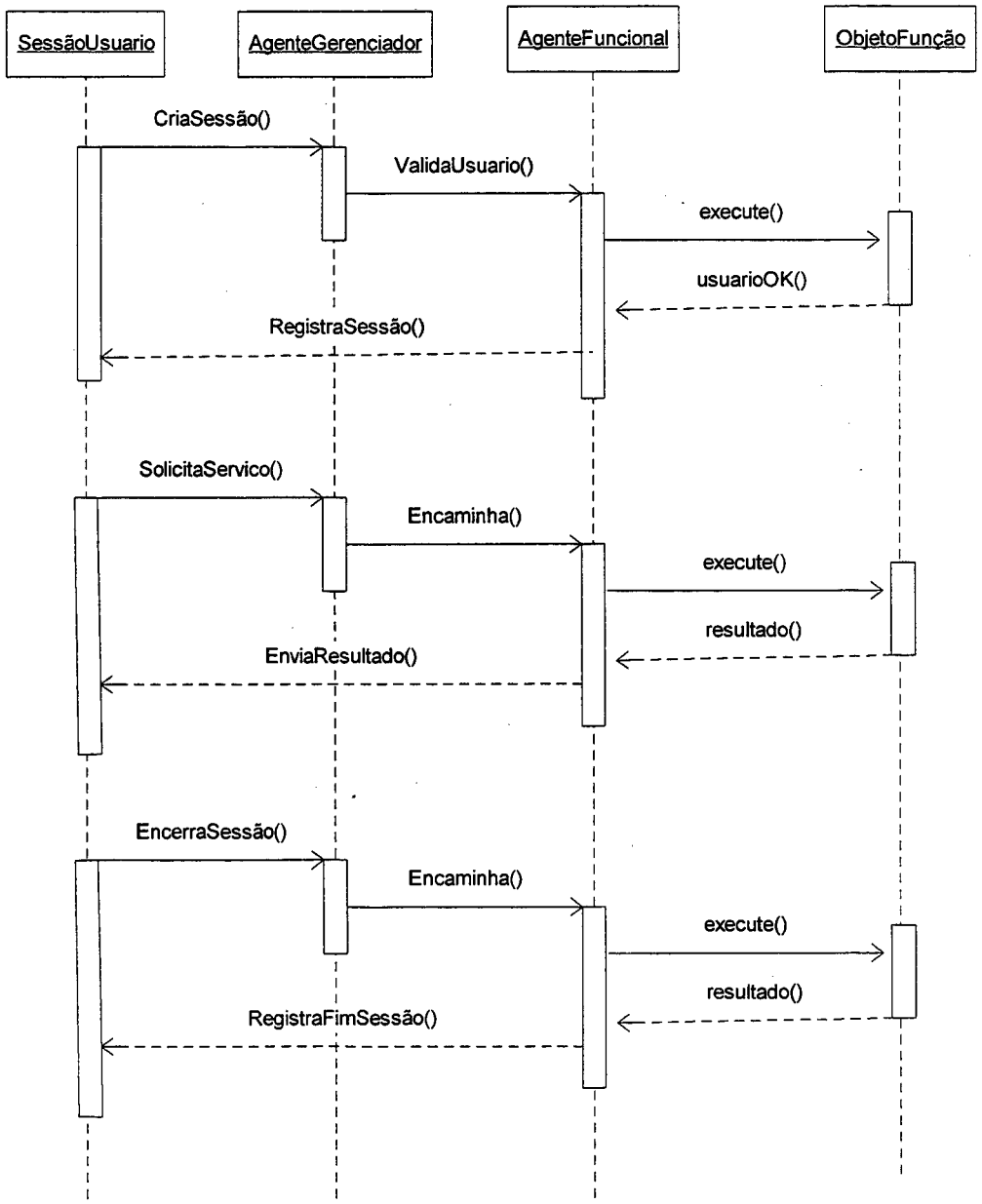

Figura 5.7 – Início e Término da Sessão Usuário com as Interações entre Agentes

5.3.5 Objeto Função

Objetos do tipo Função são os verdadeiros executores das operações, sendo incorporados aos agentes funcionais. O objeto Função executa uma determinada operação e retorna o resultado ao AgenteFuncional.

A classe FactoryFunção cria um objeto Função para cada serviço disponibilizado pelo servidor. Nesse caso foi usada a estrutura do padrão Command [GAM2000] que implementa uma interface Função com várias subclasses, junto com o padrão de projeto Factory Method [GAM2000] que simula uma fábrica de objetos.

A Fig. 5.8 apresenta a estrutura proposta:

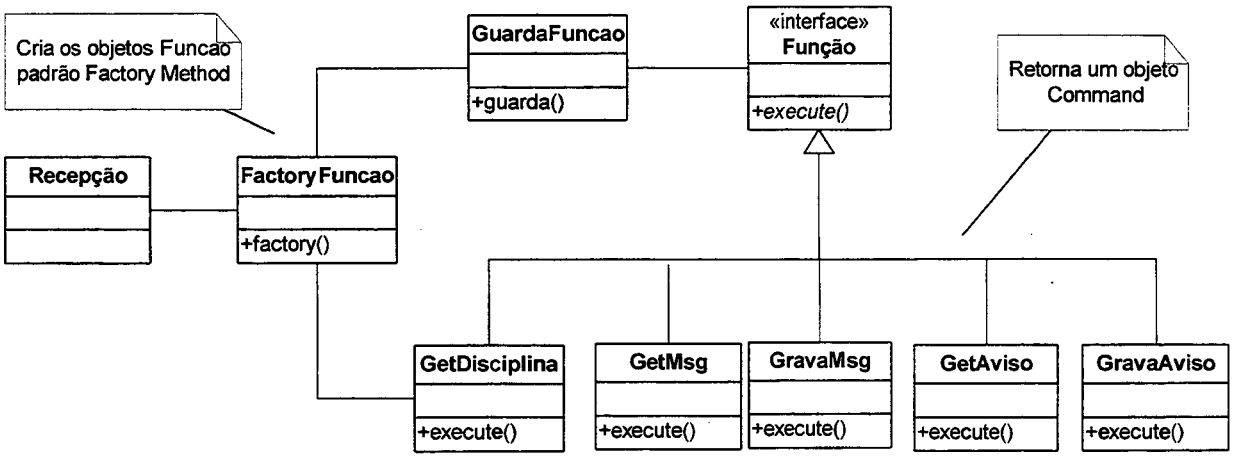

Figura 5.8 – Estrutura da Junção dos Padrões Command e Factory Method no GVD

Cada classe de operação instanciada realiza a interface Função e executa uma determinada tarefa. O objeto Função é armazenado em um arranjo, sendo associado posteriormente a um ou mais Agentes Funcionais.

A Fig. 5.9 apresenta a composição dos agentes funcionais através de um ou mais objetos Função. Alguns objetos Função, inclusive, podem ser utilizados por mais de um agente funcional. É o caso do objeto função3.

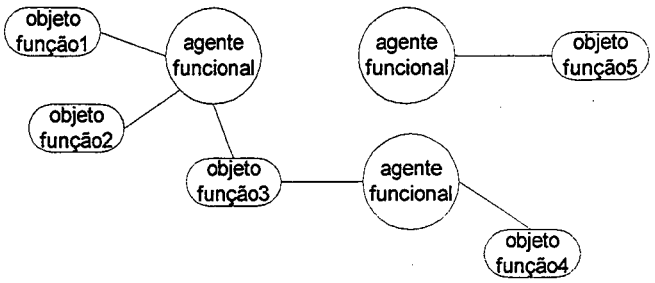

Figura 5.9 – Agentes Funcionais Compostos por Objetos Função

5.3.6 Conexão com o Banco de Dados

Existem diversos tipos de arquiteturas cliente-servidor. O sistema em estudo, GVD, tem um único servidor acessado por diversos clientes, incorporando uma arquitetura chamada múltiplos-clientes-único-servidor [OZS1999]. Dessa maneira, o banco de dados é centralizado em uma única máquina que também possui o software para o seu gerenciamento.

O sistema GVD utiliza um banco de dados relacional, manipulado pelo subsistema servidor através da linguagem *Structured Query Language* (SQL), enviando comandos e recebendo resultados. O gerenciador de banco de dados utilizado é o *Microsoft Access*. Entre o subsistema servidor e o gerenciador do banco de dados será utilizada a API de Conectividade de Banco de Dados Java (JDBC).

5.4 Análise do Uso da Abordagem Orientada a Agentes

Os benefícios e limitações da arquitetura proposta para o sistema foram analisados considerando-se, principalmente, os fatores internos do sistema. Os fatores internos são os aspectos perceptíveis aos projetistas e programadores, relacionados com a flexibilidade, a correção, a robusteza, a extensibilidade e a reusabilidade do software. A flexibilidade, a extensibilidade e a reusabilidade são características que tornam um software adaptável, fácil de alterar e com componentes reutilizáveis. A correção e a robusteza são habilidades do software de executar suas tarefas exatamente como definido e de reagir apropriadamente em condições anormais.

5.4.1 Flexibilidade, Extensibilidade e Reusabilidade

A abordagem orientada a agentes incorporou ao sistema em estudo uma estrutura divida em módulos. Cada módulo foi construído separadamente e integrado a um ambiente cooperativo através das interações entre os componentes. Assim, cada componente torna-se relativamente independente, o que permite que um módulo possa ser implantado ou alterado sem que os demais sofram modificações. Além disso, os módulos do sistema foram criados com uma conotação genérica, o que permite seu reaproveitamento em softwares similares. Desse modo, é possível afirmar que a orientação a agentes trouxe ao sistema características que o tornam extremamente flexível, reutilizável e facilmente extensível.

5.4.2 Correção e Robusteza

A modularidade proposta pelo uso de agentes fez com que as funcionalidades do sistema fossem divididas em vários componentes, cada um especialista em uma determinada tarefa. Desse modo, durante a fase de construção do sistema cada elemento foi testado separadamente de modo a garantir a perfeita execução da tarefa, tanto em um ambiente de normalidade como em um ambiente de exceção. Os vários ambientes criados pelos diversos componentes foram testados, inicialmente, em separado, sendo depois reunidos e integrados. Desse modo, as características de correção e robusteza foram incorporadas com facilidade ao sistema.

5.4.3 Otimização e Desempenho

Se analisarmos a arquitetura proposta para o sistema em estudo, iremos perceber que, ao dividirmos o sistema em vários pedaços autônomos e especialistas, estamos limitando a atuação de cada componente a uma tarefa muito específica, muitas vezes minúscula. A junção de vários módulos, cada um com uma pequena tarefa, garante a execução de todos os serviços disponibilizados ao cliente pelo servidor. Essa característica garante que cada módulo fique preso à solicitação de um cliente por pouquíssimo tempo. Por ser pequena, a tarefa de cada componente é rapidamente executada, estando aquele módulo apto a atender outro cliente logo em seguida. Por outro lado, como os serviços foram divididos em várias pequenas tarefas, existe a necessidade de se utilizar um número maior de componentes para executá-los, exigindo o uso expressivo de memória.

5.5 Incorporando Novas Funcionalidades

Todos os serviços disponibilizados pelo subsistema servidor são executados através de operações realizadas por objetos do tipo Função. Os objetos Função são instâncias de classes que implementam a interface Função. Como cada classe é um componente independente, a exclusão ou a inclusão de uma não afeta em nada o funcionamento das outras. Com isso, cada nova funcionalidade do sistema pode ser facilmente incorporada, simplesmente com a criação de uma classe que especifique como a nova operação deve ser executada.

A Fig. 5.10 reapresenta a Fig.5.8, com a incorporação da classe NovoServico:

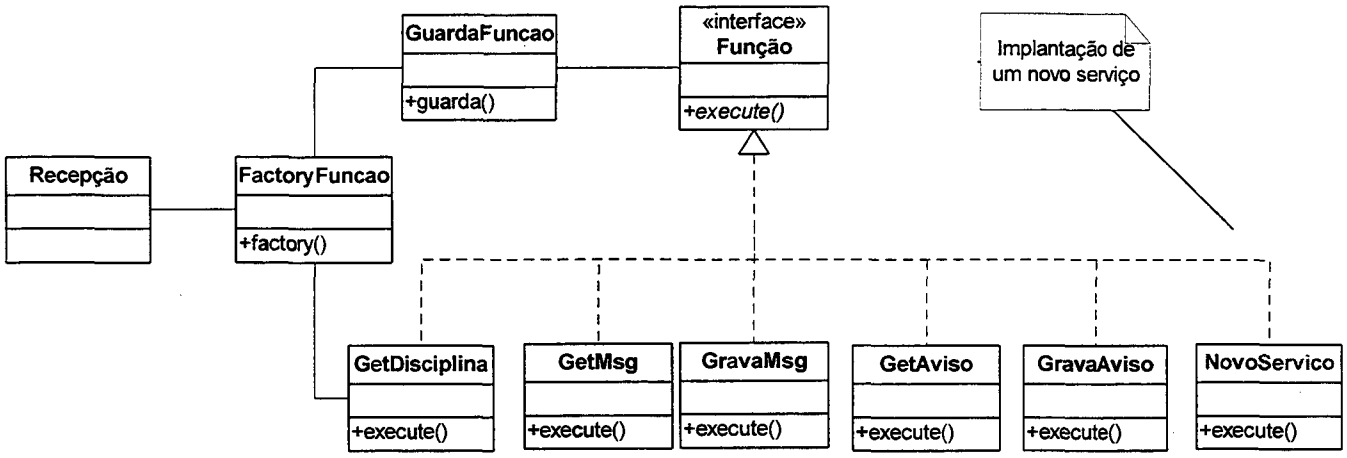

Figura 5.10 – Estrutura das Classes Funcionais com Adição de Nova Funcionalidade

Após a criação da nova classe, o método factory() da classe FactoryFunção deve ser alterado de modo a enxergar a nova classe e criar o respectivo objeto, conforme código Java apresentado a seguir. No restante do subsistema servidor, nada mais é alterado:

```
public class FactoryFuncao
{ static GuardaFuncao funcao;

 FactoryFuncao()
 { funcao = new GuardaFuncao();}

 public void  factory()
 { funcao.guarda(new OpeGetDisciplinas());
   funcao.guarda(new OpeNovaDisciplina());
   funcao.guarda(new OpeMsgProfeTodos());
   funcao.guarda(new OpeMsgProfeUm());
   funcao.guarda(new OpeLerMsgsProfe());
   funcao.guarda(new OpeLerMsgsAluno());
   funcao.guarda(new OpeGetAssuntoPro());
   funcao.guarda(new OpeGravaMsgAlu());
   funcao.guarda(new OpeAcessaAviso());
   funcao.guarda(new OpeGravaAviso());
   funcao.guarda(new OpeGravaAvisoGeral());
   funcao.guarda(new OpeGravaArquivo());

   funcao.guarda(new NovoServico()); // inclui novo serviço
 }

}
```

No subsistema cliente a interface gráfica deve ser alterada para incorporar a nova funcionalidade. Assim que o novo serviço for solicitado pelo cliente, a classe BalcãoDespacho do servidor cria automaticamente um AgenteFuncional especializado naquela função.

De maneira geral, portanto, o uso de agentes resultou numa arquitetura extremamente flexível. Além disso, dentro do contexto do problema, alterando-se a interface gráfica no subsistema cliente e criando-se as respectivas classes realizadoras da interface Função no subsistema servidor, a arquitetura proposta para o sistema em estudo é facilmente implantada e reutilizada em qualquer outro sistema similar.

5.6 Conclusão

Esse capítulo apresentou uma proposta de arquitetura baseada em agentes para o servidor do sistema Gerenciador Virtual de Disciplinas. Essa arquitetura utiliza as técnicas tanto da orientação a objetos quanto da orientação a agentes para criar uma estrutura onde vários ambientes interagem cooperativamente para atingir um mesmo objetivo. Nesse caso, o objetivo comum é o atendimento de solicitações enviadas por um grupo de usuários conectados a Internet.

A arquitetura proposta utiliza os conceitos básicos da orientação a objetos e a modularidade de sistema proposta pela orientação a agentes. Essa modularidade implica em dividir o sistema em pequenos componentes de software autônomos, chamados agentes, para facilitar o gerenciamento das interações em um ambiente concorrente.

Sendo assim, o subsistema servidor foi dividido em vários elementos que interagem e compartilham informações em um ambiente concorrente. O atendimento às solicitações dos clientes é feito através de dois tipos de agentes: o agente gerenciador que gerencia a sessão usuário e encaminha suas solicitações, e os agentes funcionais que garantem a execução das operações necessárias para o atendimento do serviço solicitado.

6. CONCLUSÃO

O uso de agentes pode significar um avanço na busca de ferramentas que facilitem o desenvolvimento de sistemas. Apesar de não se ter, ainda, dentro da Engenharia de Software uma definição exata de agentes, neste trabalho foi utilizado o conceito de agentes de software, excluindo-se, portanto, as características advindas da inteligência artificial. Os conceitos da orientação a agentes utilizados foram principalmente aqueles que auxiliam no desenvolvimento de sistemas complexos. Nesse sentido, a utilização de agentes enfocou, principalmente, a criação de módulos autônomos que interagem em um ambiente colaborativo, cada um deles com funções bem definidas dentro do objetivo do sistema.

O escopo do trabalho incluiu o desenvolvimento do protótipo de um sistema cliente/servidor denominado Gerenciador Virtual de Disciplinas (GVD). Através desse sistema alunos e professores conectados à Internet trocam informações sobre uma determinada disciplina.

Durante o processo de desenvolvimento do GVD foram utilizadas várias ferramentas e técnicas de análise e projeto de software. Entre elas, destacam-se algumas ferramentas padronizadas pela notação *Unified Modeling Language* – UML, técnicas e *frameworks* básicos da programação orientada a objetos e padrões de projeto. Os seguintes padrões foram particularmente importantes no projeto do sistema GVD:

- O padrão Singleton foi utilizado para instanciar classes cujos objetos serviriam como repositórios de informações gerais, de acesso global, compartilhados por todos os processos criados no ambiente;

- O padrão Command padronizou a instanciação e a utilização das classes funcionais, de modo a obter objetos com mesma estrutura e diferentes comportamentos, utilizados na composição dos agentes funcionais. Assim, apesar de terem a mesma estrutura, os agentes podem desenvolver diferentes atividades;

- O padrão Factory Method permitiu que os objetos do tipo Função fossem criados em um só lugar, dentro de uma única estrutura, facilitando a incorporação de novas funcionalidades ao sistema;

- O Remote Proxy foi utilizado no subsistema cliente, criando uma conexão remota entre o usuário e o subsistema servidor.

A valorização maior desse trabalho foi dada à arquitetura do subsistema servidor, apresentada no Capítulo 5. A arquitetura proposta foi construída de maneira a facilitar o gerenciamento das transações em um ambiente concorrente, com o uso de agentes de software. Nesta proposta, os agentes utilizados têm as seguintes características:

- São elementos autônomos, compostos por um ou mais objetos;

- Apesar de todos os agentes terem a mesma estrutura, cada um é especialista em uma função, dependendo do comportamento dos objetos que o compõe;

- Os agentes gerenciadores estão incorporados a processos independentes criados pelas sessões usuários;

- São reativos. Os agentes gerenciadores, por exemplo, podem tomar a iniciativa de derrubar a sessão usuário quando o usuário deixar de interagir com o sistema por um longo período (*timeout)*;

- Os agentes funcionais são componentes de acesso global, sendo disputados por todos os processos simultaneamente. Atendem a todos os usuários, sincronizadamente, recebendo os pedidos dos agentes gerenciadores e enviando o resultado diretamente ao respectivo usuário.

Dentre os objetivos deste trabalho, podemos destacar que o ponto de maior complexidade é o gerenciamento das interações entre os diversos processos no ambiente servidor. A recepção das mensagens do cliente é feita por um *thread* e a execução do serviço e o retorno do resultado ao cliente é efetuado por outro. Nesse sentido, o uso de agentes de software facilitou o controle, uma vez que cada agente é um módulo independente responsável por uma determinada tarefa. Alguns agentes foram

incorporados aos processos receptores de mensagens e outros aos processos executores dos serviços. Essa divisão de tarefas permitiu um gerenciamento mais efetivo do ambiente.

Finalmente, apesar de uma certa complexidade na sua concepção, podemos dizer que a arquitetura proposta facilitou a manutenção e a incorporação de novas funcionalidades. Inclusive sua adaptação para uso em outros softwares similares pode ser feita sem muitas dificuldades. Nesse sentido, a linguagem Java mostrou-se eficiente na construção de aplicações cliente/servidor, exatamente por facilitar a conexão entre equipamentos através da Internet.

A seguir, são apresentadas algumas propostas para continuação desse trabalho:

- **Disponibilidade de acesso**

O acesso ao sistema GVD feito através de uma interface gráfica disponibilizada a partir do subsistema cliente residindo na máquina do usuário possui um inconveniente importante: a disponibilidade de acesso. O acesso ao sistema através de um *browser* HTML seria bastante proveitoso neste caso, uma vez que, no ambiente acadêmico, muitas vezes na própria sala de aula, esse tipo de ferramenta torna-se cada vez mais disponível. Não será difícil incorporar o acesso ao Sistema GVD através de um *browser* por dois motivos: primeiro, a plataforma Java já provê um excelente *framework* para o acesso usuário via protocolo HTTP: a tecnologia Servlet/JSP; segundo, o projeto do sistema GVD incorporando uma arquitetura em três camadas facilita a utilização de servlets e páginas JSP. Duas soluções são possíveis: a integração do sistema GVD a um servidor HTTP/Servlet e o tratamento de pedidos HTTP diretamente pelo sistema GVD.

- **Integração a outros sistemas**

Embora haja um desacoplamento relativamente forte entre o lado cliente e o lado servidor, a possibilidade de integração do sistema GVD com outros sistemas não foi de fato contemplada na arquitetura proposta. Há várias formas de se prover facilidades de integração entre sistemas. Uma forma bastante comum é através da troca de mensagens. O sistema GVD incorpora um sistema de troca de mensagens relativamente limitado, restrito ao contexto das interações entre os usuários e o servidor. Um sistema de mensagens permitindo o acesso a algumas funcionalidades do sistema, eventualmente ao nível dos agentes, tornaria a sua integração com outros sistemas mais admissível.

- **Infra-estrutura para os agentes**

Por último e não menos importante, uma infra-estrutura de comunicação para os agentes. Geralmente, agentes operam sobre uma infra-estrutura de comunicação mais próxima à arquitetura do sistema. No sistema que foi desenvolvido, os agentes interagem a partir de mensagens entre os objetos que os realizam. Uma infra-estrutura de comunicação mais sofisticada, por exemplo, o mecanismo do quadro-negro, permitiria um menor acoplamento entre os agentes e suas próprias realizações tornando o sistema mais reutilizável, flexível e, principalmente, mais facilmente extensível.

REFERÊNCIAS

[AMU2000] AMUND T., **A survey of agent-oriented software engineering** , 12/2000, http://jfipa.org/publications/AgentOrientedSoftwareEngineering

[BAU2001] BAUR, B., **UML class diagrams and agent-based systems** , *Proceedings of the fifth International Conference on Autonomous Agents* , maio 2001 – Quebec -Canada

[BOO1994] BOOCH, G. **Object oriented analysis and design with applications** , Addison-Wesley, 1994.

[BOO1999] BOOCH, G., Rumbaugh J., Jacobson I., **The unified modeling language user guide**. Addison-Wesley, 1999.

[BRO1995] BROKS F.P.,**The mythical man-month**, Addison-Wesley, 1995.

[BRO1996] BROWN, K. **Crossing chasms patterns languages of program design**, vol.2, Addison-Wesley, 1996.

[BRU1997] BRUGALI D., Menga G. Aarsten A.,**The framework life span** , *Commnications of the ACM,* 65-68, outubro, 1997.

[BRU2000] BRUGALI D., Sycara K., **Towards agent oriented application frameworks** , *ACM Computing Surveys,* 21-27, (março), 2000.

[COA1993] COAD, P. **Projeto baseado em objetos**. Quarta Edição. Editora Campus, 1993.

[DEI2001] DEITEL, H.M. **Java: como programar**. Terceira Edição. Porto Alegre: Bookman, 2001

[DEL2000] DELLA L., Clark D., **Teaching object-oriented development with emphasis on pattern application** , *ACM Computing Surveys,* dezembro,2000.

[DEP2001] DEPKE R., Reiko H, Jochen M.K., **Improving the agent-oriented modeling process by roles,** *ACM Computing Surveys,* maio, 2001.

[DEU1989] DEUTSCH, P. **Frameworks and reuse in the smalltalk 80 system.** In: BIGGERSTAFF, T. Software reusability. New York: ACM Press, 1989. V.1, p. 57-71.

[DUE2001] DUELL M., **Non-software examples of software design patterns,** *Addendum to the 1997 ACM SIGPLAN* , 1997.

[ECK2000] ECKEL B., **Thinking in java**, 2000, http:// www.BruceEckel.com.

[ECK2001] ECKEL B., **Thinking in patterns problem-solving techniques using java**, 2001, http:// www.BruceEckel.com.

[FOW1997] FOWLER, M., Scott K, **UML Distilled** , Addison-Wesley, 1997.

[FOW2000] FOWLER, M. **UML essencial: um breve guia para a linguagem padrão de modelagem de objetos.** Segunda Edição. Porto Alegre: Bookman, 2000.

[FRA1996] FRANKLIN S.,Graesser A., **Is it an agent, or just a program? A taxonomy for autonomous agents.** Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag,1996.

[GAM1994] GAMMA, E, Helm R., Johnson R., Vlissides J., **Design Patterns : elements of reusable object-oriented software**. Reading : Addison-Wesley, 1994.

[GAM2000] GAMMA, E., Helm R., Johnson R., Vlissides J., **Design Patterns : elements of reusable object-oriented software**. Reading : Addison-Wesley, 2000.

[GAR1994] GARLAN D., Shaw M., **An Introduction to software architecture,** CMU – CS-94-166- *CMU Software Engineering Institute Technical Report* , 1994.

[GEN1994] GENESERETH M., Ketchpel S., **Software agents** , *Commnications of the ACM,* 48-53, julho, 1994.

[GUE2001] GUERIN F., Pitt J., **Denotational semantics for agent communication languages** , *Proceedings of the fifth International Conference on Autonomous Agents* , maio 2001

[HIR1996] HIRSHFIELD S., Ege R.K., **Object-oriented programming,** *ACM Computing Surveys* –Vol.28, No.1, março 1996.

[JEN1999] JENNINGS N. R., Wooldridge M., **Agent-oriented software engineering** , *IEE Proc. On Software Engineering* - 1999

[JEN2001] JENNINGS N.R., **An agent-based approach for building complex software systems** , *Communications of the ACM* , Vol. 44, No.4, abril 2001.

[KEN2000] KENDALL E. et all, An Application **Frameworks for intelligent and mobile agents** , *ACM Computing Surveys,* Vol.32 - (março), 2000.

[LAR1997] LARMANN, C. **Applying uml and patterns** , Prentice-Hall , 1997.

[LAR2000] LARMANN, C. **Utilizando uml e padrões : uma introdução à análise e ao projeto orientados a objetos.** Porto Alegre : Bookman, 2000.

[LES1995] LESSER V.R., **Multiagent systems : an emerging subdiscipline of AI,** *ACM Computing Surveys,* Vol. 27, No. 3, setembro 1995.

[LEW1998] LEWANDOWSKI S.M., **Frameworks for component-based client/server computing,** *ACM Computing Surveys,* 30-1, (março), 1998.

[LEW2000] LEWIS, J., Loftus W., **Java software solutions : foundations of program design**. Second Edition. Addison-Wesley, 2000.

[LIN2000] LIND J., **Issues in agent-oriented software engineering,** *The first International Workshop AOSE-2000.*

[MAG2000] MAGNANELLI M, Norrie M.C., **Databases for agents and agents for databases,** *In Proc. Of 2nd International Bi-Conference Workshop on AO Information Systems,* junho, 2000.

[MAY1995] MAYFIELD J., Labrou Y., Finin T., **Evaluating KQML as an agent communication language in** Wooldridge et all Eds, *Inteligent Agents II,* Springer, 1995, 347-360.

[MEY1997] MEYER, B.**Object-oriented software construtions**. Second Edition. Prentice Hall PTR, 1997.

[MOO1999] MOORE John, location : http;//www.jguru.com/faq/Patterns

[NOW2001] NOWOSTAWSKI M., Purvis M., Cranefield S., **Modeling and visualizing agent conversations** , *Proceedings of the fifth International Conference on Autonomous Agents* , maio 2001

[OMG1999] OBJECT MANAGEMENT GROUP, **UML specification version 1.3** , Junho, 1999.

[OZS1999] OZSU M. Tamer, Valduriez P., **Principles of distributed database systems,** Second Edition, Prentice Hall, 1999.

[PAR2001] PARUNAK H.V.D., **Representing social structures in uml,** *Proceedings of the fifth International Conference on Autonomous Agents* , maio 2001- Quebec - Canada

[PRE1997] PREE W., Sikora H.,**Design pattern for object-oriented software development,** *Proceedings of the ICSE* 1997 – Boston, MA, USA.

[RUM1991] RUMBAUGH, J. **Object-Oriented modeling and design**. Prentice-Hall International Inc, 1991.

[SCH1996] SCHMIDT D., **Using design patterns to guide the development of reusable objetc-oriented software,** *ACM Computing Surveys* – dezembro 1996

[SCH1999] SCHNEIDER G. , Winters J.P., **Applying use cases - a practical guide**, Addison-Wesley, 1999.

[SIL2000] SILVA, Ricardo Pereira e. **Suporte ao desenvolvimento e uso de frameworks e componentes**. Porto Alegre: Instituto de Informática da UFRGS, 2000a. Tese de Doutorado.

[SIM1996] SIMON H.A., **The sciences of the artificial.** MIT Press, 1996.

[SYCARA, 1996] SYCARA K.,Pannu A. et all, **Distributed intelligent agents,** *IEEE Expert, Special Issue on Intelligent Systems and their Applications*, dezembro, 1996.

[WOO1997] WOOLDRIDGE M., **Agent-based software engineering**. *In IEE Proceedings of Software Engineering* 144, 26-37, 1997.

[WOO1998] WOOLDRIDGE M., **Pitfalls of the agent-oriented development,** *Proceedings of the Second International Conference Autonomous Agents , 1998.*

[WOO1999] WOOLDRIDGE M., Jennings N.R., Kinny D., **A methodology for agent-oriented analysis and design ,** *In Proceedings of the third ICAA* , pag. 69-76, 1999.

[WOO2000] WOOLDRIDGE M., Jennings N.R., Kinny D., **The gaia methodology for agent-oriented analysis and design.** *Journal of Autonomous Agents and Multi-Agent Systems,* 3(3) ,285-312, 2000.

[WOO2001] WOOD M., DeLoach S.A.,**An overview of the multiagent systems engineering methodology.** Apresentado no 1st. *Int. workshop on agent-oriented software engineering* (AOSE 2000), junho 2000.

[ZAM,2000] ZAMBONELLI F., Jennings N.R., et all, **Coordination of internet agents models, technologies and applications,** *Agent-oriented Software Engineering for Internet Applications, 2000.*