ROBERTO RIBEIRO ABRAHÃO JUNIOR

APLICAÇÃO DA GERAÇÃO AUTOMÁTICA DE CÓDIGO NO PROCESSO DE DESENVOLVIMENTO E NAS ARQUITETURAS DE SOFTWARE CORPORATIVO

Projeto de Formatura apresentado à disciplina PCS 2050 – Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Área do Projeto: Engenharia de Software

Orientador: Prof. Dr. Jorge Luis Risco Becerra

São Paulo
2007

ROBERTO RIBEIRO ABRAHÃO JUNIOR

APLICAÇÃO DA GERAÇÃO AUTOMÁTICA DE CÓDIGO NO PROCESSO DE DESENVOLVIMENTO E NAS ARQUITETURAS DE SOFTWARE CORPORATIVO

Projeto de Formatura apresentado à disciplina PCS 2050 – Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Área do Projeto: Engenharia de Software

Orientador: Prof. Dr. Jorge Luis Risco Becerra

São Paulo
2007

DEDICATÓRIA

À minha família e namorada pelo apoio inesgotável.

AGRADECIMENTOS

Ao Prof. Dr. Jorge Luis Risco Becerra pela orientação além de todo seu apoio e atenção durante o projeto.

Ao Prof. Gianpiero Cabodi do Politecnico di Torino pela orientação na fase do projeto realizada na Itália.

A Roberto NAVONE pelo grande apoio durante a fase do projeto realizada na Itália.

A empresa Mainline Srl. pelo apoio e por disponibilizar a ferramenta Adaptive Framework para o projeto.

Sê todo em cada coisa. Põe quanto és
No mínimo que fazes.

(Ricardo Reis)

RESUMO

Este estudo trata da influência da geração automática de código no desenvolvimento de software, seja no tange a arquitetura, seja no que se refere aos processos de desenvolvimento. O trabalho analisa as principais questões arquiteturais de uma aplicação corporativa de software e como sua arquitetura pode se beneficiar da geração automática. Em seguida, este projeto descreve um ambiente de desenvolvimento de software (através de sua modelagem) baseado na Fábrica de Software do Laboratório de Tecnologia de Software da Escola Politécnica da USP e verifica como o uso de um gerador de código afeta tal ambiente. Por fim, o trabalho propõe um ambiente de desenvolvimento baseado em geração automática de código. Este trabalho utiliza a ferramenta Adaptive Framework como exemplo de gerador de código com o intuito a mostrar como a geração automática auxilia o desenvolvimento de software reduzindo custos e prazos.

Palavras-chave: Arquitetura de software. Processos de desenvolvimento de software. Geração automática de código.

ABSTRACT

This study is about how the automatic code generation influences the software development on its architecture and on its development processes. This work analyses the main architectural issues of an enterprise application and how its architecture can benefits from code generation. Afterward, this project describes a software development environment (through its model) based on the Software Factory of Escola Politécnica's Laboratório de Tecnologia de Software and it verifies how the use of a code generator affects such an environment. Finally, this work purposes a development environment based on automatic code generation. This project uses the Adaptive Framework tool as an example of code generator in order to show how code generation helps the software development reducing costs and time.

Key-words: Software architecture. Software development processes. Automatic code generation.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE FRAGMENTOS DE CÓDIGO

SUMÁRIO

1 Introdução

1.1 Visão Geral

O mundo é cada vez mais dependente de sistemas de software. As exigências nos requisitos e na qualidade estão em constante crescimento enquanto os prazos são sempre menores. Além disso, as aplicações têm de ser robustas, escaláveis, portáveis e de fácil manutenção. Para atingir todos esses objetivos, é indispensável basear o sistema em uma boa arquitetura e desenvolver o projeto usando um processo de desenvolvimento de software bem definido e consolidado.

A arquitetura compreende os principais blocos do software e como essas partes se unem (JOHNSON, 2004). A arquitetura oferece um esqueleto para a aplicação. Falhas comuns em projetos de software frequentemente ocorrem porque seus programadores não entenderam as lições deixadas por desenvolvedores mais experientes. Apesar das mudanças tecnológicas, as mesmas idéias básicas de design podem ser adaptadas e aplicadas para resolver problemas comuns (FOWLER, 2002).

1.2 Objetivos Gerais

- Criar um ambiente de desenvolvimento de software baseado na geração automática de código.
- Projetar uma arquitetura que se adapte à geração automática de código e que seja robusta, escalável, portável e de fácil manutenção.

1.3 Objetivos Específicos

- Analisar e discutir as principais alternativas arquitetônicas e suas principais vantagens e desvantagens, levando sempre em consideração a geração automática de código e como ela influencia as decisões do arquiteto.
- Sugerir um pacote de blocos arquiteturais que melhor se adapta à geração automática de código.

- Modelar um ambiente tradicional de desenvolvimento de software baseado na Fábrica de Software do Laboratório de Tecnologia de Software (LTS) da Escola Politécnica da USP. Em seguida, inserir um gerador automático de código neste ambiente e mostrar como as atividades de desenvolvimento e a alocação de recursos se altera.
- Utilizar a ferramenta Adaptive Framework (NAVONE, 2006) para implantar e avaliar um ambiente de desenvolvimento de software que usa um gerador de código. O Adaptive Framework é uma infra-estrutura, criada pela empresa italiana Mainline Srl., usada para estruturar a camada de lógica de negócio de aplicações corporativas baseadas em geração automática de código. Este framework resume várias questões arquitetônicas estudadas neste projeto.

1.4 Motivação

Um dos princípios do Rational Unified Process (RUP) é basear-se em uma arquitetura executável o mais cedo possível (KROLL). Em outras palavras, o RUP sugere projetar, implementar e testar a arquitetura assim que possível, criando uma fatia vertical da aplicação que tem somente algumas funcionalidades. Com a arquitetura executável é possível validar as principais escolhas arquitetônicas aplicando algumas métricas (como desempenho e facilidade de manutenção), além de que, é mais fácil prever o tempo e o custo necessários para o desenvolvimento do resto da aplicação.

Normalmente, a primeira fatia vertical de uma aplicação é criada por um número pequeno de desenvolvedores mais experientes, ou arquitetos. Depois é mais fácil introduzir novos membros (programadores menos experientes) ao projeto, pois o desenvolvimento é praticamente um trabalho de "copiar e colar", no qual os programadores se baseiam no código de referência para criar novas funcionalidades para o sistema, adaptando-o ligeiramente para novos requisitos funcionais. Portanto, trata-se de um trabalho mais mecânico que pode ser automatizado. Neste ponto, é possível usar um gerador automático de código, auxiliando os desenvolvedores a programar somente os algoritmos que não podem ser automatizados e precisam ser escritos a mão.

1.5 Justificativa

Além de acelerar o desenvolvimento, a geração automática de código leva a uma redução no número de erros, pois o código modelo terá sido testado várias vezes anteriormente. Além disso, ao corrigir um defeito no código modelo, ele será imediatamente corrigido em toda a aplicação.

A geração automática de código dá a possibilidade de usar alguns estilos de codificação cuja programação manual seria inviável por se tratar de um modo muito verborrágico e, portanto, que tenderia a erros. Esta característica aumenta a lista de modelos arquitetônicos para uma determinada aplicação. O uso de um gerador automático de código influencia algumas decisões arquiteturais do projeto. Portanto, durante o desenvolvimento da fatia vertical inicial da aplicação, o arquiteto deveria saber se a geração automática de código será usada.

Esse automatismo influencia também todo o processo de desenvolvimento de software, principalmente a parte de implementação, pois a alocação de recursos humanos, a lista de atividades de projeto, a duração, o custo e os produtos gerados em cada uma serão diferentes. Consequentemente, o gerente de projeto deverá estar ciente da utilização do gerador automático de código e como ele altera os processos de desenvolvimento, de modo a criar um plano de projeto adequado.

1.6 Estrutura do Trabalho

Este trabalho começa discutindo os principais conceitos da geração automática de código (capítulo 2). Então, o capítulo 3 analisa os principais blocos arquitetônicos de uma aplicação corporativa, as principais soluções existentes no mercado, e como a geração automática de código influencia a arquitetura do software. O capítulo 4 apresenta a ferramenta Adaptive Framework e as principais idéias por trás dela. O capítulo 5 mostra um projeto real com o Adaptive Framework. O capítulo 6 discute os processos de teste em aplicações corporativas e como uma boa arquitetura e geradores de código podem melhorar a testabilidade. Por fim, o capítulo 7 propõe um ambiente de desenvolvimento de software (através de sua modelagem em BPMN) baseado na geração automática de código.

2 Fundamentos da Geração Automática de Código

"Geração de código é sobre escrever programas que escrevem programas" (HERRINGTON, 2003). Um gerador de código auxilia os desenvolvedores a escrever códigos repetitivos, mecânicos e consequentemente tediosos que poderiam ser facilmente automatizados. O código pode ser gerado independentemente da linguagem de programação utilizada. Além disso, um gerador de código pode criar qualquer tipo de saída, por exemplo, documentação do código, scripts DDL para banco de dados (BEAULIEU, 2005) e casos de teste de unidade (BECK, 2002). Existem muitos níveis de geração automática de código, do mais simples auxiliador de codificação, que cria somente o esqueleto de uma classe até geradores complexos que constroem e gerenciam enormes seguimentos de código.

2.1 Benefícios da Geração Automática de Código

Já foi discutido que a geração automática de código pode acelerar incrivelmente o desenvolvimento de uma aplicação corporativa e também reduzir o número de erros no software. Ao mesmo tempo, a geração automática trás outros benefícios para a equipe de desenvolvimento (HERRINGTON, 2003).

2.1.1 Benefícios da Geração Automática de Código para Desenvolvedores

- **Qualidade** – No código escrito a mão, cada nova classe possui uma qualidade ligeiramente superior à escrita anteriormente. A base de código possui inconsistência de qualidade ao longo das classes. A geração automática de código cria uma consistente base de código imediatamente e quando os modelos são modificados e o código é gerado novamente, a correção de defeitos e o melhoramento no código são aplicados por toda a base de código.
- **Consistência na codificação** – O código gerado automaticamente é consistente no estilo de programação e nomenclatura de variáveis, o que resulta em um código fácil de entender e usar.

- **Um único ponto de mudança** – Uma alteração no modelo implementa automaticamente o ajuste por todo o sistema. Isto significa que as atividades de manutenção são mais fáceis.
- **Mais tempo em design** – Dado que a geração automática de código comprime o tempo de programação, mais tempo pode ser dedicado a atividades de design e criação de protótipos. Os desenvolvedores podem focar nos requisitos do negócio ao invés de aspectos tecnológicos.

2.1.2 Benefícios da Geração Automática de Código para Gerentes de Projeto e Arquitetos

- **Consistência Arquitetural** – O gerador de código usado no projeto é a realização de decisões arquitetônicas feitas no início do ciclo de desenvolvimento. Isso encoraja os programadores a trabalhar dentro da arquitetura. Quando é difícil gerar o código que o desenvolvedor precisa, é um bom sinal de que a nova característica não funciona bem dentro da arquitetura estipulada.
- **Abstração** – As regras de negócio são abstraídas em arquivos livres de detalhes de linguagem ou *framework* que impossibilitam a portabilidade. Os arquitetos podem construir novos modelos que traduzem a lógica em outras linguagens ou plataformas, muito mais fácil de realizar do que com código escrito a mão. Os analistas podem validar o projeto na forma abstrata.
- **Maior motivação** – Escrever código repetitivo é muitas vezes tedioso. Com a geração automática de código, os programadores podem focar em um trabalho único, mais interessante. Consequentemente, a equipe de projeto tende a estar mais motivada, o que facilita o trabalho do gerente.

2.2 Tipos de Geração Automática de Código

Essencialmente, existem dois tipos de geração automática de código: ativa e passiva (HERRINGTON, 2003). No primeiro tipo, o gerador cria uma porção de código que o programador usa para começar a desenvolver. Os geradores passivos não são responsáveis por dar manutenção ao código gerado. É o caso dos assistentes em alguma IDE como Eclipse (BURNETTE; GALLARDO; MCGOVERN,

2003) e Visual Studio (PARSONS; RANDOLPH, 2006). Por outro lado, geradores ativos são os responsáveis pelo código gerado durante um longo período. Os desenvolvedores podem alterar os parâmetros de geração e executá-la muitas vezes sobre a mesma saída.

Este projeto foca nos geradores ativos. As seções seguintes examinam em mais detalhes os tipos mais importantes de geradores automáticos de código como definido em (HERRINGTON, 2003). Eles podem variar de uma simples solução até um gerador complexo.

2.2.1 Code Munging

Um gerador de tipo *Code Munging* lê um código fonte (um ou mais arquivos) usando alguns elementos importantes para criar um ou mais arquivos de saída. Normalmente, ele usa expressões regulares ou simples *code parsing*. A Figura 1 mostra como um gerador *Code Munging* funciona.

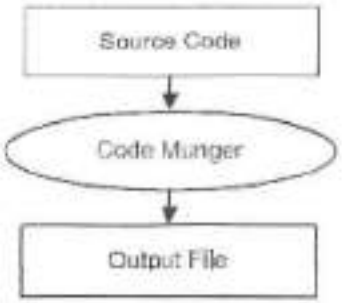

Figura 1 – Gerador Code Munging

Um exemplo de *Code Munging* é o JavaDoc (JAVADOC), uma ferramenta Java que lê uma classe, interpretando alguns tipos especiais de comentários (começando com @) e cria uma documentação em formato HTML (GOODMAN, 2006). Outro exemplo é o XDoclet (RICHARDS; WALLS, 2003), um motor *open-source* de geração de código que viabiliza a programação orientada a atributos para Java. Isto é, o programador pode adicionar meta dados (atributos) ao código Java (usando elementos especiais do JavaDoc). A seguir, o XDoclet lerá os arquivos fonte e gerará muitos artefatos, como descritores XML (HUNTER, 2007), código fonte, etc. Estes arquivos são gerados a partir de modelos que usam as informações fornecidas pelo código-fonte mais os elementos JavaDoc. O *Code Munging* também

pode ser usado, por exemplo, para ler de um arquivo algumas constantes ou protótipos de funções.

2.2.2 Inline-code expander

O *Inline-Code Expander* lê alguns elementos especiais do código fonte e os substitui com o código de produção. Portanto, este tipo de gerador recebe na entrada um ou mais arquivos de código fonte e gera código de produção. A Figura 2 mostra como ele funciona. Um possível uso deste tipo de gerador é inserir consultas SQL em um código fonte.

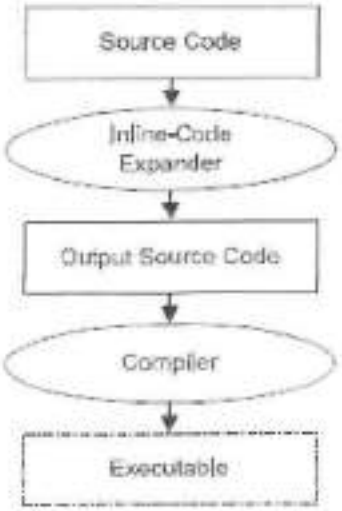

Figura 2 – Gerador Inline-Code Expander

2.2.3 Gerador Mixed-Code

Este tipo de gerador automático de código é muito similar a um inline-code expander. Ele também lê alguns elementos especiais do código fonte e os substitui com o código de produção. Entretanto, ao invés de criar um diferente arquivo de saída, um gerador mixed-code coloca a saída de volta no arquivo de entrada. A Figura 3 ilustra a idéia.

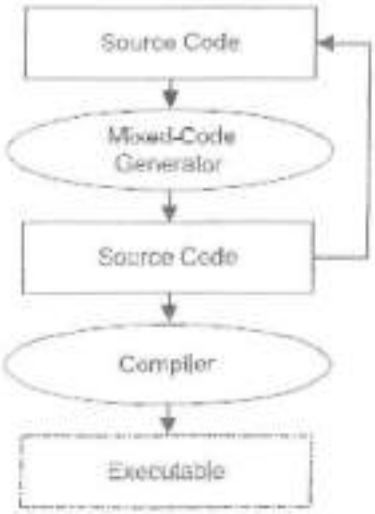

Figura 3 – Geração de tipo Mixed-Code

2.2.4 Gerador Partial-Class

Um gerador de tipo *Partial-Class* recebe na entrada um conjunto de arquivos em uma linguagem abstrata (formato XML, por exemplo) e um conjunto de arquivos de código modelo. Então, ele gera uma classe base que será estendida por uma classe escrita a mão. O gerador pode criar também o esqueleto da classe derivada na primeira vez que ele roda (um exemplo de geração de código passiva).

Deste modo, é possível combinar os benefícios da geração automática de código com a flexibilidade de algoritmos manuais. É um meio termo entre o código manuscrito e a geração automática de código para uma inteira camada da aplicação. O desenvolvedor pode começar construindo somente a classe base; então, ao passo que o gerador lidar com mais casos especiais, o desenvolvedor pode realizar a transição para um gerador que constrói uma camada completa do sistema.

Uma grande vantagem deste sistema é a velocidade de implementação. O gerador pode lidar com os casos genéricos, enquanto os casos especiais, difíceis de serem automatizados, podem ser codificados manualmente.

Esta forma de geração de código usa conceitos de orientação a objetos (como herança) e é baseada no *Generation Gap Pattern* (VLISSIDES). A Figura 4 mostra o fluxo do processo de geração.

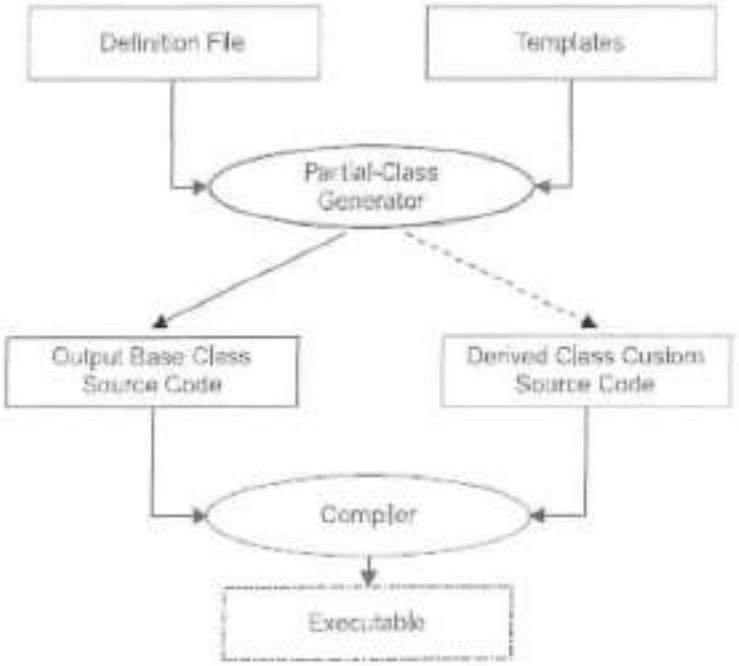

Figura 4 – Geração Partial Class

2.2.5 Geração de Camada (Tier Generation)

Este tipo de geração de código é praticamente igual à geração *Partial-Class*, mas sua saída contém todo o código de uma inteira camada da aplicação. Portanto, não existe classe derivada. Esse gerador é usado quando é possível gerar automaticamente a camada inteira da aplicação. O desenvolvedor não pode criar código manualmente e o gerador deve lidar com todos os casos especiais. A Figura 5 mostra como o gerador funciona.

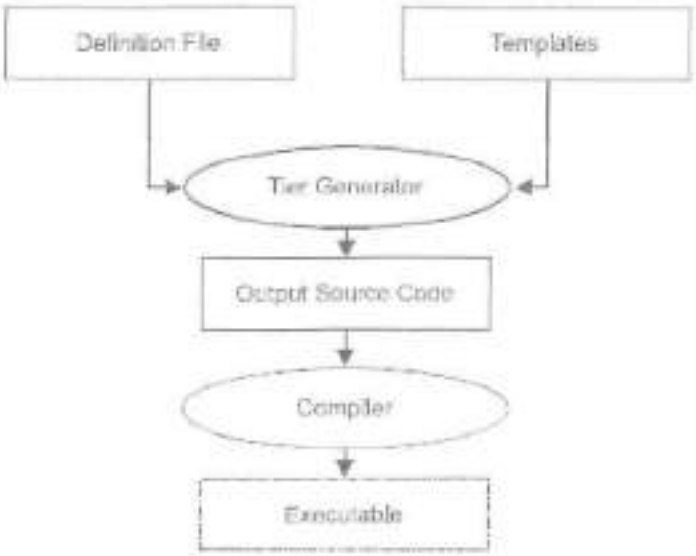

Figura 5 – Geração de Camada (Tier Generation)

2.3 Escolhendo o tipo de geração automática de código

As primeiras três formas de geração automática de código discutidas na seção anterior usam um arquivo de código fonte como entrada. Ao contrário, os últimos dois tipos usam um arquivo de definição em uma linguagem abstrata. O segundo grupo de gerados de código permitem ao projetista separar quase completamente os requisitos de negócio da linguagem de programação e das escolhas arquitetônicas.

Em projetos reais de software, é muito difícil de gerar automaticamente cada caso especial que uma aplicação precisa. Normalmente a relação custo - benefício é baixa e vale a pena implementar algumas partes do código manualmente. Então, a geração de tipo *partial-class* com *generation gap* se adapta perfeitamente neste caso. Por essa razão, este projeto usa tal forma de geração para discutir as arquiteturas de aplicações corporativas de software.

2.3.1 Como construir um gerador partial-class

A primeira coisa a decidir é o formato do arquivo de definição. Uma boa escolha pode ser o XML devido à sua popularidade, além de ser um excelente formato para descrever meta dados. Linguagens de programação modernas como Java e .NET têm ótimas ferramentas para ler XML. O projetista pode usar o Document Object Model (DOM) que define uma maneira padronizada para acessar e manipular

documentos XML (VOHRA; VOHRA, 2006). O DOM apresenta o documento XML como uma estrutura em árvore cujos nós são elementos XML, atributos e texto. Adicionalmente, usar um esquema ou validação Document Type Definition (DTD) nos arquivos XML torna a gestão de erros muito mais simples.

A outra escolha a se fazer é como montar a estrutura do arquivo de código modelo. Ele poderia usar expressões regulares (WATT, 2005) e o gerador as substituiria por um bloco de texto. Outra possibilidade é usar uma ferramenta de texto modelo como Java Server Pages (JSP) (BERGSTEN, 2003). Ela é normalmente usada para criar páginas web dinâmicas, mas pode ser utilizada para construir qualquer tipo de documento dinâmico, posto que sua funcionalidade básica é substituir alguns elementos especiais por blocos de texto e possui *loops* e blocos condicionais. Uma vantagem do JSP é sua facilidade de uso e muitos desenvolvedores Java possuem familiaridade com a ferramenta. Seu problema é que depende de um servidor web. Portanto, o gerador de código seria muito pesado.

Uma melhor solução é o Apache Velocity (VELOCITY), uma ferramenta de modelos baseada em Java que oferece uma linguagem de modelo simples e poderosa usada para referenciar objetos definidos em Java. Essa ferramenta simplesmente recebe na entrada uma string (com macros Velocity) e produz outra string na saída (com valores nos lugares das macros). Aplicações comuns de Velocity são aplicações web, e-mails automáticos, transformações XML e geração de código. Assim como JSP, Velocity também provê *loops*, blocos condicionais e outras funções avançadas.

O Fragmento de Código 1 mostra um exemplo de modelos velocity para gerar classes simples de Java com um construtor padrão e alguns atributos. Vamos supor que o gerador leia um arquivo de definição que descreve a classe Cliente com os atributos name (nome), surname (sobrenome), e birthday (data de nascimento). O código gerado será como o mostrado no Fragmento de Código 2.

```
public class ${ClassName}
{
  public ${ClassName}()
  {
  }

#foreach ($Attribute in $Attributes)
  private $Attribute.Type $Attribute.Name;
```

```
#end
}
```

Fragmento de Código 1 – Exemplo de um Template Velocity

```
public class Client
{

  public Client()
  {
  }

  private String name;

  private String surname;

  private Date birthday;

}
```

Fragmento de Código 2 – Exemplo de uma Classe Gerada com Velocity

O exemplo anterior mostra como um gerador automático de código baseado em modelos de código funciona. Trata-se apenas de um exemplo simples. Um verdadeiro gerador de tipo *partial-class* deveria gerar uma classe abstrata (de modo a obrigar os desenvolvedores a estendê-la) com métodos abstratos de tipo protected, com o intuito de sobrescrevê-los na classe derivada.

Obviamente um gerador automático de código não precisa usa ferramentas como Velocity ou XML. O projetista poderia implementar tudo sozinho. Por exemplo, ele poderia ler um arquivo de modelo de código (usando seu próprio formato) e então procurar e substituir alguns elementos especiais definidos previamente. Entretanto, Velocity e outras ferramentas de *template* constituem uma solução já existente e consolidada. Além disso, tais ferramentas são bem documentadas e existem muitos fóruns na internet que discutem problemas e como utilizá-las.

3 Blocos Arquitetônicos de uma Aplicação Corporativa

A evolução das aplicações corporativas levou a uma modelo consolidado de camadas como mostrado na Figura 6 e contém as seguintes camadas:

- **Camada de Apresentação** – contém a lógica de apresentação dos dados, a qual lida com a interação entre o usuário e o software. Ela poderia ser uma simples uma linha de comando, uma aplicação baseada em janelas, uma página *web*, um *web service* (FREEMAN, 2002) etc.
- **Camada de Negócio** – expões a lógica de negócio (com as regras do negócio) para a camada de apresentação. Ela envolve, por exemplo, cálculos baseados nas entradas e dados armazenados, validação de algum dado oriundo da camada de apresentação, etc. É provavelmente a parte mais importante de um sistema corporativo de software.
- **Camada de Acesso aos Dados** – é responsável pela persistência dos dados em um *data source* que pode ser, por exemplo, um banco de dados relacionais, um banco de dados orientado a objetos, um arquivo e assim por diante.

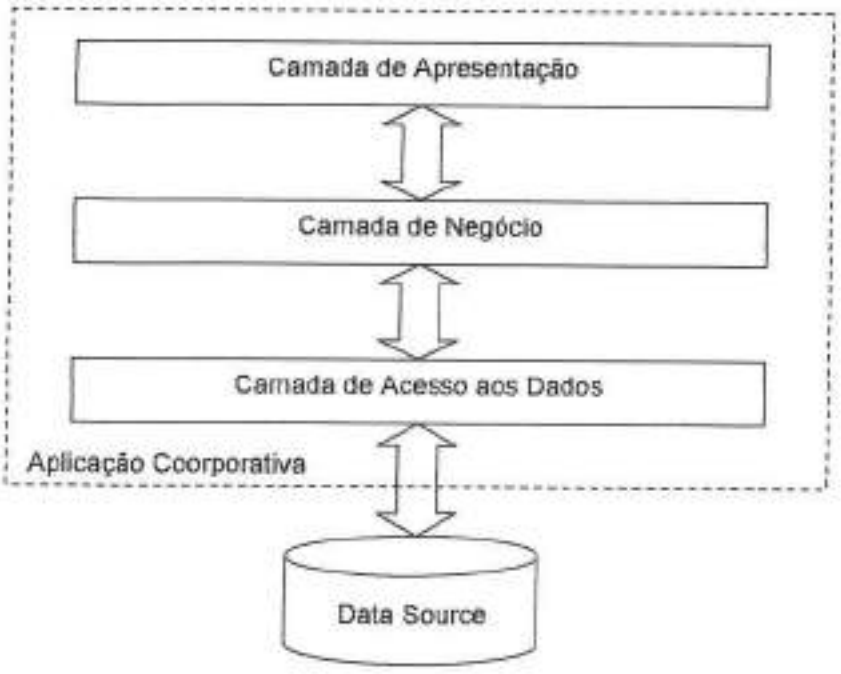

Figura 6 – Modelo de Aplicações Coorporativas de Três Camadas

Cada nível descrito anteriormente representa uma camada lógica da aplicação e são independentes dos outros. Todas as camadas podem residir fisicamente juntas (no mesmo computador) ou podem ser distribuídas pela rede.

Este projeto discute as camadas de negócio e de acesso aos dados. Ele foca em Relational Database Management Systems (RDBMS) como *data source*, visto que se trata do tipo mais comum de banco de dados usado hoje em dia (JOHNSON, 2004). As próximas seções cobrem os principais blocos arquitetônicos dessas duas camadas de aplicações corporativas de software.

3.1 Gerenciamento de Objetos de Negócio / Resource Handling

Tipicamente, o gerenciamento de objetos de negócio envolverá alguma forma de container no qual a aplicação é executada. Normalmente, trata-se de um tipo de *framework*. Eh também possível executar a aplicação sem um container. No entanto, isso usualmente leva a um gerenciamento caótico dos objetos de negócio e uma inconsistente manipulação da configuração.

Objetos que rodam dentro de um container são gerenciados pelo container. Esses objetos são geralmente chamados de *managed objects*. Tal tipo de gerenciamento deve prover:

- **Gerenciamento do ciclo de vida:** Controle do ciclo de vida dos objetos. Essa gestão pode ser simples como simplesmente abstrair a criação de novos objetos do código que os usa, ou mais complexa, como criar chamadas que alertem os *managed objects* de, por exemplo, sua ativação no container.
- **Lookup:** Um modo de obter referências dos objetos (gerenciados pelo container) que abstrai do código os detalhes de implementação da busca (ocultados dentro do container).
- **Configuração:** Um modo consistente de configurar os *managed objects* em tempo de execução de modo a permitir parametrização. Os valores de parametrização devem ser externalizados do código, de tal maneira que eles possam ser alterados sem a necessidade de uma nova compilação ou o envolvimento de desenvolvedores.
- **Resolução de dependências:** Gestão das relações entre os objetos gerenciados pelo container.

Existem muitos containeres na indústria. Eles podem ser divididos em dois grupos:

- **Heavyweight containers:** (containeres pesados) como containeres EJB (Weblogic (NYBERG et al., 2003), JBoss (JBOSS) e Websphere (WEBSPHERE)).
- **Lightweight containers**: (containeres ligeiros) como Spring framework (JOHNSON et al., 2005) e Pico Container (PICO).

Estes termos foram criados por desenvolvedores de software e não possuem uma definição exata. Entretanto, há algumas características que podem classificar um container como *lightweight*. Como definido em (JOHNSON, 2004) um *lightweight container* deve ser:

- Um container capaz de gerenciar o código da aplicação, mas não impõe dependências especiais no código (como interfaces específicas).
- Um container que inicia rapidamente.
- Um container que não requer nenhuma atividade especial de implantação (*deployment*) para instalar novos objetos.
- Um container que possui dependências mínimas com alguma API e pode ser executado em uma grande variedade de ambientes.

3.1.1 Inversão de Controle

Uma boa prática arquitetural é evitar que o código aplicativo dependa do container no qual roda. Essa abordagem leva a uma melhor portabilidade (pois o código não depende de um fornecedor específico de containeres) e testabilidade (dado que o código pode ser executado fora do container). Entretanto, não é obvio como atingir esse objetivo. Somente objetos muitos simples podem trabalhar isolados: a maioria dos objetos de negócio tem dependências, como outros objetos de negócio, objetos de acesso a dados (DAOs) e recursos (como conexões a bancos de dados). A satisfação das dependências dos objetos, sem introduzir uma dependência no container, pode ser realizada através da Inversão de Controle (IoC).

A inversão de controle é um importante conceito em *frameworks*, no qual o código da aplicação não chama o container. Ao invés disso, quando necessário, o container chama o código aplicativo. Existem dois tipos de inversão de controle:

- **Dependency Lookup**: O container provê métodos de *callback* aos objetos, e um contexto de *lookup*. Esta é a abordagem usada pelo EJB (discutida nas próximas seções). Ela deixa a responsabilidade sobre cada objeto de usar APIs especiais para localizar recursos e colaboradores. Neste caso, a inversão de controle é limitada, pois o container precisa que a aplicação chame o método de *lookup* para obter recursos.
- **Dependency Injection**: Com a injeção de dependências, os componentes não realizam *lookup*. Ao invés disso, eles provêm métodos ou propriedades permitindo ao container resolver as dependências. O container é completamente responsável por ligar os objetos, passando objetos já resolvidos às propriedades ou aos construtores dos novos objetos. Este é o caso do Spring Framework (JOHNSON et al., 2005) e Adaptive Framework (discutido em detalhes no capítulo 4).

Usando a injeção de dependências ao invés da *dependency lookup*, o código aplicativo pode ser totalmente livre do container. As próximas seções discutem cada um em mais detalhes.

3.1.1.1 Dependency Lookup

EJB e outras APIs J2EE, como Servlets, oferecem a inversão de controle na forma de *lookup* de dependências. O container gerencia o ciclo de vida dos objetos, enquanto eles são responsáveis por realizar seus próprios *lookups*.

Normalmente para aplicações baseadas em J2EE, o *lookup* de dependências é feito através do Java Naming and Directory Interface (JDNI) (BODOFF et al., 2004). O Fragmento de Código 3 mostra um exemplo de um objeto de negócio J2EE que busca suas dependências através de JNDI.

```
public class MyBusinessObject implements MyBusinessInterface
{
  private DataSource ds;
  private MyCollaborator myCollaborator;
  private int myIntValue;

  public MyBusinessObject() throws NamingException
  {
    Context ctx = null;
    try
    {
      ctx = new InitialContext();
```

```
        ds =
(DataSource)ctx.lookup("java:comp/env/dataSourceName");
        myCollaborator = (MyCollaborator)
            ctx.lookup("java:comp/env/myCollaboratorName");
        myIntValue = ((Integer)

ctx.lookup("java:comp/env/myIntValueName")).intValue();
      }
      finally
      {
        ctx.close();
      }
    }
    //Métodos de Negócio
}
```

Fragmento de Código 3 – Exemplo de Lookup JNDI

Em teoria, o exemplo anterior parece ser uma boa solução. O JNDI separa MyBusinessObject da implementação de ds e colaborator além de externalizar o valor da primitiva myIntValue. Eh possível usar um DataSource diferente ou alterar a classe que implementa a interface MyColaborator sem mudar o código Java. Por tanto, há plugabilidade. O código é portável entre servidores de aplicação, dado que usa APIs J2EE padronizadas. Entretanto, esta estratégia apresenta os seguintes problemas:

- MyBusinessClass não será executada fora de um servidor de aplicação, pois depende de JNDI. É mais difícil de separar o JNDI do container do que o DataSource.
- Não é fácil utilizar algo diferente de JNDI para localizar recursos e colaboradores.
- É muito difícil testar esta classe, pois é muito difícil criar uma classe stub para o contexto JNDI (detalhes sobre testabilidade no capitulo 6)
- Há muito código. As classes deveriam conter operações em um nível consistente de abstração. O baixo nível do código JNDI representa uma intrusão na classe.
- O código é fortemente tipado. É necessário realizar casts da classe Object em todos os casos. Isto é especialmente importuno com tipos primitivos, como int, no qual se deve usar o wrapper correspondente.

3.1.1.2 Dependency Injection

O princípio fundamental da injeção de dependências para a configuração é que os objetos não são responsáveis por localizar seus colaboradores e recursos dos quais dependem. Ao invés disso, um container IoC configura os objetos, externalizando o *lookup* de recursos do código aplicativo (JOHNSON, 2004). Isso oferece as seguintes vantagens:

- O *lookup* é completamente removido do código aplicativo.
- Não há dependência na API do container.
- Nenhuma interface especial é necessária.
- A mesma abordagem pode ser aplicada para simples valores como string e int como para colaboradores complexos.

Existem dois tipos de injeção de dependências:

- **Setter Injection**: os objetos expressam suas dependências através da configuração de valores e colaboradores via propriedades. No caso de Java Beans, elas são expressas através de métodos *setter* (RICHARDSON et al., 2005). No caso das linguagens de .NET (como C#), há um conceito nativo de propriedades (NAGEL ET AL., 2005). Os métodos *setter* são invocados imediatamente após a instanciação do objeto pelo container, e antes que qualquer método de negócio seja executado. Então, não há problemas de threading relacionados com essas propriedades.
- **Constructor Injection:** os objetos expressam suas dependências através dos argumentos passados a seus construtores. Esta é a abordagem usada pelas primeiras versões do Pico Container (PICO).

Tanto a injeção por propriedades quanto a por construtores oferecem um enorme aperfeiçoamento na tradicional necessidade por *lookups* no código aplicativo. Em ambos os casos, o conceito fundamental de injeção de dependências é o mesmo. A Tabela 1 mostra as diferenças entre os dois tipos.

Setter Injection	Constructor Injection
Propriedades são bem suportadas por IDEs.	Pode haver ligeiramente menos código, porém não há diferença em complexidade.
Propriedades são auto-documentáveis.	Os argumentos dos construtores Java não possuem nomes visíveis por introspecção. Isto deixa os desenvolvedores dependentes do índice do argumento, sem um nome amigável (um problema ainda maior se existem dependências do mesmo tipo).
Muitas classes existentes podem ser usadas dentro de um container IoC sem alterações.	Construtores com mais de um argumento são menos comuns do que propriedades em códigos já existentes.
A *Setter Injection* trabalha bem para objetos que têm valores *default*, ou seja, nem todas as propriedades precisam ser passadas no início.	Quando colaboradores são passados através de argumentos dos construtores, torna-se impossível mudar a referência dentro do objeto.
A ordem na qual *setters* são chamados não é estipulada em nenhum contrato. Então, às vezes é necessário invocar um método depois de que o último *setter* foi chamado para iniciar o componente.	Teste de unidade pode ser ligeiramente mais difícil, visto que todos os argumentos dos construtores devem ser providos, mesmo os colabores irrelevantes ao teste em questão.
Nem todos os *setters* precisam ser chamados antes do uso. O objeto pode ser deixado parcialmente configurado.	É garantido que cada objeto gerenciado pelo container se encontra em um estado consistente (totalmente configurado) antes que qualquer método de negócio possa ser chamado.

Tabela 1 – Comparação entre Setter Injection e Constructor Injection

3.1.2 O Framework Spring

O *framework* Spring é um popular *framework* para aplicações *open source* Java cujo fundamento é seu container ligeiro: a *bean factory* (um container IoC que usa a injeção de dependências). Ele provê meios consistentes de configurar e gerenciar objetos Java. Eh possível implementar uma *factory* customizada usando algumas interfaces Spring. No entanto, o framework Spring oferece algumas implementações como a *factory* baseada em XML.

A interface fundamental do container ligeiro é BeanFactory (mostrada no Fragmento de Código 4). Esta é uma interface simples, a qual é simples de implementar caso nenhuma das implementações de Spring seja suficiente. A interface BeanFactory oferece dois métodos getBean para localizar instâncias

através de um nome, com a opção de verificar o tipo requerido (e lançar uma exceção caso haja uma disparidade entre os tipos).

```
public interface BeanFactory {

  Object getBean(String name) throws BeansException;
  Object getBean(String name, Class requiredType)
    throws BeansException;
  boolean containsBean(String name);
  boolean isSingleton(String name)
    throws NoSuchBeanDefinitionException;
  String[] getAliases(String name)
    throws NoSuchBeanDefinitionException;
}
```

Fragmento de Código 4 – Interface org.springframework.beans.factory.BeanFactory

Enquanto o container ligeiro para Java Beans é um conceito central dentro do Spring, isto é apenas a fundação para uma solução para todas as camadas de middeware. Spring é um application *framework* completo que pode ser usado em diferentes níveis. Ele consiste em múltiplos *sub-frameworks* que são relativamente independentes, mas podem se integrar fortemente, se desejado (JOHNSON, 2004). Os frameworks mais importantes são:

- **Container IoC**: discutido anteriormente.
- **Framework de programação orientada a aspectos:** suporte AOP (LADDAD, 2003) para intercepção de métodos em qualquer classe gerenciada pelo container ligeiro do Spring.
- **Framework de acesso a dados:** trabalha com RDBMS na plataforma Java usando JDBC e ferramentas de mapeamento objeto-relacional provendo soluções para desafios técnicos que são reusáveis em vários ambientes baseados em Java.
- **Framework de gerenciamento de transações:** uma infra-estrutura genérica de gerenciamento de transações, com estratégias de transações plugáveis (como JTA e JDBC) e vária meios para demarcar transações em aplicações.
- **Framework MVC Web:** uma implementação limpa de MVC Web, consistente com a abordagem de configuração de Java Beans

- **Framework de acesso remoto:** uma fina camada de abstração para acesso a serviços remotos sem o uso de *lookups* codificados, e para exposição de objetos como serviços remotos.

Este projeto não tem a pretensão de entrar em detalhes sobre o *framework* Spring. Para maiores informações consulte (JOHNSON et al., 2005).

3.2 Persistência de Dados

Conceitualmente, existem alguns *patterns* para persistir dados em um RDBMS como definido em (FOWLER, 2002). O mais simples de todos é o **Transaction Script**, que trabalha diretamente com o modelo relacional, executando consultas SQL e elaborando os result sets conforme necessário. A lógica de negócio é organizada em métodos para cada caso de uso. Sua principal vantagem é que se trata de um modelo procedural simples que muitos desenvolvedores entendem. O uso direto da linguagem SQL se adapta bem no caso de consultas agregadas e atualizações em bloco, dado que o RDBMS executa essas operações de modo eficiente.

Entretanto, em uma aplicação orientada a objetos, a abordagem do Trasaction Script não é a mais adequada, pois este tipo de software trabalha com representação de objetos ao invés de colunas e tabelas diretamente. Trata-se de um *pattern* definido em (FOWLER, 2002) como **Domain Model**. Neste caso, os desenvolvedores não lidam com SQL diretamente. O *pattern* arquitetônico Domain Model define uma camada composta de objetos que modelam os requisitos do negócio. Dado que a maioria das aplicações modernas usa RDBMS ao invés de ODBMS, há a necessidade de uma tradução da linguagem a objetos para a linguagem relacional. Isto é feito por um mapeamento objeto-relacional (O/R), que pode ser implementado em basicamente dois modos: por um **Active Record** ou um **Data Mapper**.

O Active Record é uma abordagem na qual cada linha da tabela de um banco de dados relacional é representada por uma instância de sua classe correspondente. Cada objeto persistente é responsável por executar operações de inserção, atualização e apagamento. Além disso, uma classe de busca é geralmente usada para recuperar um objeto persistido em um banco de dados. Esta é a abordagem usada pela tecnologia Enterprise Java Bean (EJB) (MONSON-HAEFEL, 2001) até

sua versão 2.1. Cada entity bean (como são chamados os objetos persistentes na terminologia EJB) é responsável por sua persistência e uma classe *home* cuida da recuperação dos objetos persistidos.

Diferentemente da metodologia usada pelo Active Record, um Data Mapper é usado para transferir dados entre objetos e um RDBMS mantendo todas essas entidades independentes. Ele atua como um Mediator (GAMMA et al., 1995) entre objetos em memória e o RDBMS. Com esta abordagem, os objetos não precisam conhecer nada a respeito da tecnologia de persistência utilizada. Um exemplo de Data Mapper é Hibernate (BAUER ET AL., 2004). Trata-se de uma poderosa ferramenta de mapeamento O/R para Java, a qual permite aos desenvolvedores criar classes persistentes como simples objetos Java (POJO) (JOHNSON, 2004) seguindo o paradigma da orientação a objetos (herança, polimorfismo, etc.). Hibernate usa um arquivo XML para descrever o mapeamento. Tal arquivo contém informações como o nome da tabela e a relação entre suas colunas e os atributos dos objetos. Deste modo, os objetos são completamente independentes da tecnologia de persistência utilizada.

As próximas seções discutem as principais tecnologias (ou *frameworks*) que objetivam auxiliar a persistência de dados.

3.2.1 EJB Entity Beans

Um ponto central da arquitetura J2EE tradicional é a tecnologia EJB. Um componente EJB, ou enterprise bean, é um corpo de código estruturado de certo modo e que possui campos e métodos para implementar módulos da lógica de negócio. Um entity bean é um componente EJB que representa um objeto de negócio em um mecanismo de armazenamento persistente. Um entity bean pode gerenciar sua própria persistência (Bean Managed Persistence – BMP) ou pode delegar tal função ao container EJB (Container Managed Persistence – CMP). Este projeto não tem a pretensão de entrar em detalhes sobre a tecnologia EJB. Uma explicação completa pode ser encontrada em (MONSON-HAEFEL, 2001). Esta seção discute os principais problemas dos EJB entity beans (até sua versão 2.1) e por que não é uma boa alternativa.

Embora não seja parte obrigatória da arquitetura J2EE, muitas aplicações baseadas nessa plataforma usam EJB e muito frequentemente esta escolha é

equivocada, provavelmente influenciada pela moda no mundo Java. O principal problema de aplicações baseadas em EJB é que elas possuem uma arquitetura desnecessariamente complexa, o que pode ser muito custoso. Os desenvolvedores J2EE tendem a argumentar que o custo inicial será balanceado por uma redução dos custos nas fases futuras do projeto. Grande complexidade inicial significa mais código para escrever e manter, mais potencial para defeitos, mais demora em demonstrar funcionalidades para os usuários, maior chance de falha e custo mais alto.

A Figura 7 mostra a arquitetura J2EE mais tradicional. Esta não é a única abordagem que pode ser usada com a plataforma J2EE nem o único modo de usar a tecnologia EJBs. Entretanto, tal esquema reflete a visão mais comum de J2EE como uma plataforma distribuída. As classes Java são espalhadas em diferentes JVMs, embora seja possível coloca-las juntas de modo a remover o excesso de tráfego da invocação remota. A camada web é normalmente provida por um *framework* MVC (FOWLER, 2002). O container web e o container EJB podem residir em computadores diferentes. No entanto, em uma aplicação bem projetada deveria haver uma camada de *proxy* do lado do cliente para os EJBs remotos, de forma a oferecer uma separação limpa (e física) entre as camadas web e EJB.

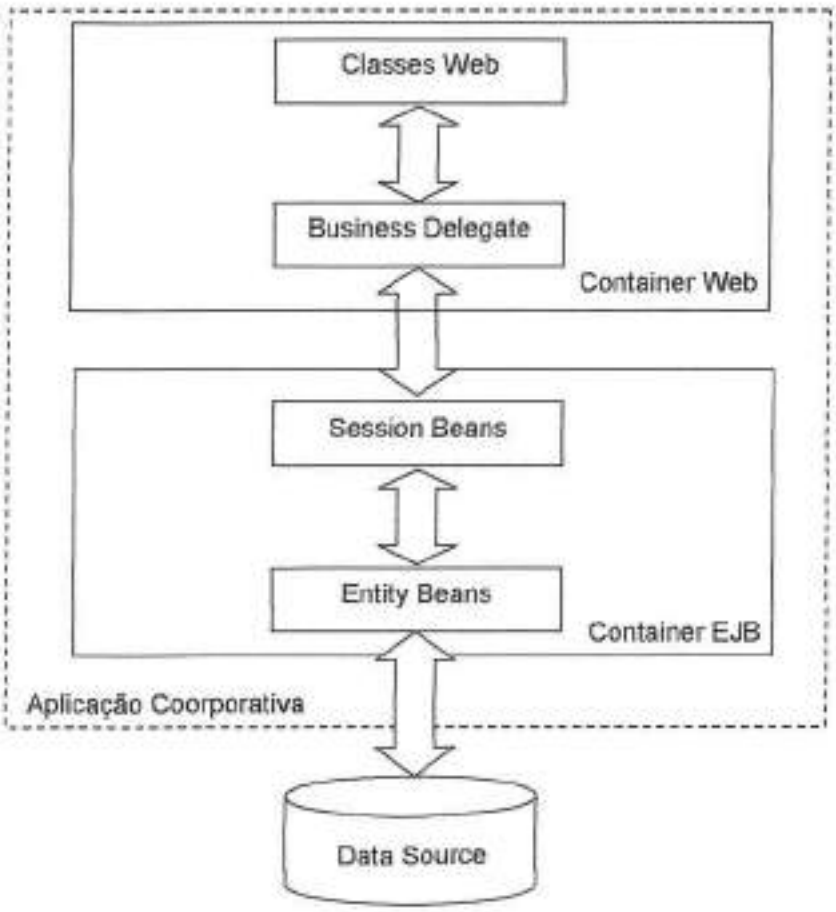

Figura 7 – Arquitetura J2EE Clássica

A camada de serviços de negocio (discutida no início do capítulo) é composta por Session Beans com interfaces remotas, rodando dentro de um container EJB. Uma Session Bean é um tipo de EJB que implementa uma tarefa de negócio (um ou mais métodos de negócio) (MONSON-HAEFEL, 2001). Contrariamente aos Entity Beans as Session Beans não são persistentes. Uma Session Bean pode ou não manter seu estado entre métodos e transações. Se a Session Bean mantém seu estado é chamada *Stateful* e se não o faz é conhecida como *Stateless*. Na arquitetura J2EE clássica, as Session Beans são geralmente *Stateless*. O container EJB provê acesso remoto, gerenciamento de transações, gerenciamento de threads, e (em alguns casos) segurança baseada em papéis.

Nesse tipo de arquitetura, todo o acesso a dados é feito usualmente através de Entity Beans CMP com interfaces locais. A fonte de dados pode consistir em um ou mais bancos de dados (de qualquer tipo) e sistemas legados. Um container EJB normalmente gerencia transações distribuídas com JTA.

A arquitetura descrita acima tem algumas vantagens como:

- EJBs remotos oferecem uma camada de serviços bem definida.
- Os serviços EJB como gerenciamento declarativo de transações é benéfico.
- Em alguns casos, a habilidade de distribuir objetos entre múltiplos servidores pode levar a uma grande escalabilidade.
- A camada de negócios pode ser atualizada separadamente da camada web.

Embora a arquitetura clássica J2EE apresente algumas vantagens, ela agrega pouco valor quando se leva em conta suas desvantagens. Os principais problemas da tecnologia EJB são:

- O código escrito para o modelo EJB não roda de fora do container EJB, minimizando a re-usabilidade do código e a testabilidade. Entity Beans são fortemente ligadas ao modelo a componentes EJB e, portanto, a um container J2EE completo.
- Os containeres EJB são geralmente lentos para iniciarem.
- O desenvolvimento EJB é árduo, envolvendo múltiplos arquivos Java para um único EJB e possivelmente um passo de geração de código e compilação pré-instalação.
- Normalmente, os EJBs não são objetos verdadeiros, pois são restringidos pelas caracteristicas do modelo a componentes EJB, o qual não suporta herança, embora as classes que compõem um EJB participam de estruturas hierárquicas.
- Os Entity Beans BMP são particularmente onerosos de se implementar. Para evitar o trabalho direto com JDBC, o CMP deve ser usado. A configuração e declaração do mapeamento O/R, além das capacidades do CMP, dependem fortemente do produto J2EE em questão. Isto torna a migração para outros ambientes uma tarefa difícil.
- A EJB QL (a linguagem de consultas CMP) é limitada, forçando os desenvolvedores a escrever SQL customizado ou depender de extensões proprietárias oferecidas pelos vendedores de servidores de aplicação.
- O desempenho dos Entity Beans é pobre por causa dos excessos em seu projeto e da interceptação de métodos, especialmente quando estão lidando com grandes results sets.

Visto que os EJBs não podem rodar de fora do container, não é possível transferir Entity Beans para um container web ou outro cliente remoto. Para transferir dados entre os serviços de negócio e a camada de apresentação é necessário usar Data Transfer Objects (DTO) – objetos simples que contêm apenas dados (sem métodos de negócio). Portanto, a arquitetura EJB sacrifica a orientação a objetos em detrimento da distribuição de objetos.

Os EJBs são mais recomendados para aplicações muito complexas como sistemas altamente transacionais que acessam múltiplos recursos e se beneficiam de distribuição interna e serviços de mensagem (JOHNSON, 2004). Dado que a grande maioria das aplicações não precisa dessa arquitetura complexa, este projeto não considera os Entity Beans como uma boa alternativa para o bloco arquitetural de persistência de dados.

Se os EJBs devem obrigatoriamente ser usados, os desenvolvedores podem se beneficiar muito da geração automática de código, dado que o desenvolvimento de EJBs envolve muitas classes, interfaces e descritores XML verborrágicos. Existem algumas ferramentas que auxiliam o desenvolvimento de aplicações baseadas na tecnologia EJB, porém a geração de tipo *partial-class* pode ser uma alternativa mais flexível. Por outro lado, mesmo ferramentas sofisticadas de geração de código não podem esconder a complexidade do modelo e outros problemas dos EJBs.

3.2.2 Hibernate

O Hibernate procura ser um fino envoltório sobre JDBC enquanto oferece o poder da transparência persistente, adicionando uma semântica O/R, mas sem se afastar do banco de dados relacional no qual se baseia.

Sua linguagem proprietária de consultas, o HQL, oferece importantes conceitos relacionais como *joins* e funções de agregação. Em geral o HQL é mais próximo do SQL do que qualquer outra *object query language* (com EJB QL), com uma importante diferença para o SQL: as consultas são expressas em termos de propriedades dos objetos ao invés de colunas de tabelas de bancos de dados, separando assim a aplicação do esquema da base de dados. Em muitos aspectos, HQL permite aos desenvolvedores de aumentar o poder da linguagem SQL ao nível do modelo de domínio. Este projeto não se preocupa em explicar o Hibernate em detalhes. Uma explicação completa pode ser encontrada em (BAUER ET AL., 2004).

Esta seção discute as características mais importantes do Hibernate que auxiliam o mapeamento objeto-relacional.

O Hibernate detecta mudanças nos objetos através de comparações de *snapshots*. Com seu uso otimizado da reflexão através de CGLIB (CGLIB) (criando *proxies* em tempo de execução para objetos persistentes através da geração dinâmica de *byte code*), a maior exigência de desempenho de tais comparações não é tão alto quanto se pode pensar. A vantagem do modelo do Hibernate é que não é preciso modificar objetos existentes para observar seus estados.

A interface de conexão central do Hibernate é net.sf.hibernate.Session, que permite a recuperação de objetos persistidos, re-associação de objetos existentes e assim por diante. O Fragmento de Código 5 mostra um exemplo do uso de Hibernate. O fragmento de código pressupõe a disponibilidade de uma referência para sessionFactory de tipo net.sf.hibernate.SessionFactory que pode ser localizada através de JNDI ou por um registro específico da aplicação. O exemplo atualiza o preço de um produto de 1000 para 1100.

```
Session session = sessionFactory.getSession();
try
{
  List products = session.find("FROM example.Product WHERE
price =
      ?", new Integer(1000), Hibernate.INTEGER);
  for (Iterator it = products.iterator(); it.hasNext();)
  {
    Product product = (Product) it.next();
    product.setPrice(1100);
  }
  session.flush();
}
finally
{
  session.close();
}
```

Fragmento de Código 5 – Exemplo da Utilização de Hibernate

O Hibernate se encaixa bem em aplicações web e serviços remotos, pois, diferentemente dos Entity Beans EJB, ele permite que objetos persistentes sejam desassociados da Session e re-associados de forma fácil posteriormente. Trata-se do caso comum no qual objetos persistentes são enviados para a camada cliente para modificações, sendo temporariamente desacoplados de uma transação, e volta quando o usuário submete as informações. Os objetos persistentes podem servir de

forma eficiente como DTOs sem codificação extra. Esta é uma importante vantagem comparando com Entity Beans EJB.

Uma característica muito popular do Hibernate é o *Lazy Loading* (carregamento preguiçoso). Ou seja, quando recuperado, um objeto não carrega suas dependências (outros objetos com os quais se relaciona) inicialmente. Uma dependência é carregada somente quando é solicitada. Tanto coleções quanto simples objetos podem usar essa propriedade. No caso de simples objetos, o Hibernate gera *proxies* em tempo de execução via CGLIB. Isto faz o mínimo de intrusão possível, porque não requer nenhuma classe ou interface nos objetos de negócio.

O mapeamento O/R com Hibernate é configurado através de um descritor XML. A geração automática de código pode auxiliar na criação desse arquivo. Um exemplo é o XDoclet (discutido na seção 2.2.1) que usa etiquetas especiais de Javadoc e produz automaticamente o XML. O Fragmento de Código 6 mostra um exemplo de classe (persistida pelo Hibernate) que usa XDoclet.

```
/** Entidade de negócio que representa um cliente
  *
  * @author Roberto Abrahao
  * @hibernate.class table="Clients"
  */
public class Client {
  /** Identificador unívoco de um cliente
    */
  private long clientId;

  /** Nome do Cliente
    */
  private String name;

  /** Construtor padrão usado pelo Hibernate
    */
  public Client()
  {
  }

  /** Método getter do identificador do cliente.
    *
    * @hibernate.id generator-class="native"
    */
  public long getClientId()
  {
    return(id);
```

```
    }

    /** Método setter do identificador do cliente.
     */
    public void setId(long id)
    {
        this.id = id;
    }

    /** Método getter do nome do cliente.
     *
     * @hibernate.property
     */
    public String getName()
    {
        return(name);
    }

    /** Métodos do nome do cliente.
     */
    public void setName(String name)
    {
        this.name = name;
    }
}
```

Fragmento de Código 6 – Exemplo de Hibernate com XDoclet

A classe do exemplo anterior é uma simples classe Java (POJO) que pode ser associada com uma sessão Hibernate e então persistida. Adicionalmente, ela possui etiquetas especiais Javadoc que são usadas pelo XDoclet de modo a gerar os descritores XML para o Hibernate. Essas etiquetas começam com @hibernate e não são reconhecidas pelo Javadoc, somente pelo XDoclet. Por exemplo, a etiqueta @hibernate.class table="Clients" antes da declaração da classe que o objeto será persistido na tabela Clients. A etiqueta @hibernate.id antes do método getClientId indica que essa propriedade é a chave primária da entidade e generator-class="native" diz que ela não usa auto-seqüência. A etiqueta @hibernate.property antes do método getName indica uma propriedade persistente da classe. No exemplo, nem @hibernate.id nem @hibernate.property possuem informações sobre qual coluna usar. Nesse caso, o XDoclet usa o mesmo nome da propriedade. O nome da propriedade é o nome do método sem o prefixo get. Por exemplo, o método getName se refere à propriedade name. As etiquetas XDoclets podem ser usadas antes do getter ou do setter.

Com Java 5 e Hibernate 3.2 é possível usar anotações Java (RICHARDSON et al., 2005) em adição ou substituição aos meta-dados do mapeamento XML. A anotação é um modo de adicionar meta-dados a um código Java (a partir da versão 5) que, diferentemente das *tags* XDoclet, estão disponíveis ao programador também em tempo de execução. Muitas vezes é usada como uma alternativa à tecnologia XML. A anotação não usa geração automática de código e pode, em alguns casos, substituir o XDoclet.

Outros tipos de geração automática de código podem ser usados com Hibernate. O XDoclet gera somente o XML. Uma geração de tipo *partial-class* pode ser usada para gerar as classes persistentes com alguma funcionalidade (como validação das propriedades) além do descritor XML.

Para aplicações .NET existe uma versão chamada NHibernate (NHIBERNATE). Assim como o Hibernate clássico, as classes persistentes não precisam implementar nenhuma interface especial ou herdar de uma classe específica. Isto faz com que seja possível projetar a camada de negócio utilizando simples objetos .NET e seguir bem o paradigma orientado a objetos.

3.2.3 Java Persistence API (JPA)

JPA é um *framework* Java que permite aos desenvolvedores gerenciar dados relacionais na plataforma Java SE (conhecida antigamente como J2SE) e Java EE (conhecida antigamente como J2EE).

Como discutido na seção 3.2.1, a arquitetura clássica J2EE com EJB tinha muitos problemas. Seu principal problema é sua complexidade. O principal tema da versão 5 de Java EE é facilidade de desenvolvimento. Mudanças na plataforma tornaram o desenvolvimento de aplicações baseadas nela muito mais fácil, com muito menos código.

Uma das maiores melhorias na versão 3.0 da tecnologia EJB é a adição da JPA, que simplifica o modelo de persistência das entidades e possui capacidades não existentes na versão 2.1 do EJB. A JPA lida com o modo que os dados relacionais são mapeados para objetos persistentes Java. Além de simplificar o modelo de persistência, a JPA padroniza o mapeamento O/R. Como será mostrada, a JPA usa muitas idéias oriundas de outras ferramentas de mapeamento O/R, como Hibernate.

As principais vantagens versão 3.0 da tecnologia EJB em relação à versão 2.1 são:

- Requer menos classes e interfaces.
- Elimina os longos descritores através de anotações.
- Provê um mapeamento O/R mais limpo, fácil e padronizado.
- Elimina a necessidade de código de *lookup*.
- Adiciona suporte a herança, polimorfismo e queries polimórficas.
- Adiciona suporte para queries dinâmicas e estáticas.
- Provê a JPA QL – uma EJB QL melhorada
- Torna mais fácil o teste de entidades fora do container
- Pode ser usada com outras tecnologias de persistência.

O Fragmento de Código 7 mostra a mesma classe do Fragmento de Código 6 implementada usando JPA. O uso de anotações elimina a necessidade de XDoclet. Para examinar a tecnologia JPA em detalhes veja (BISWAS; ORT, 2006).

```
@Entity
public class Client {

  private long clientId;

  private String name;

  public Client()
  {
  }

  @Id
  public long getClientId()
  {
    return(id);
  }

  public void setId(long id)
  {
    this.id = id;
  }

  @Column(name="name")
  public String getName()
  {
    return(name);
  }
```

```
  public void setName(String name)
  {
    this.name = name;
  }
}
```

Fragmento de Código 7 – Exemplo da Utilização de JPA

3.3 Gerenciamento de Transações

Esta é uma parte essencial das aplicações corporativas. Uma transação é uma unidade de interação com um DBMS ou sistema similar que é tratada com coerência e de modo independente das outras transações. Ela precisa ser sempre totalmente completada ou abortada. Geralmente, uma aplicação corporativa precisa garantir propriedades de Atomicidade, Consistência, Isolação e Durabilidade (ACID) para cada transação (PAOLO et al., 2003).

Uma simples transação pode requerer várias *queries*, cada uma que lê e (ou) escrevendo informações no banco de dados. Quando isso acontece, é importante assegurar que a base de dados não foi deixada com somente algumas *queries* terminadas. Por exemplo, ao se fazer uma transferência bancária, se o dinheiro foi debitado de uma conta, é vital que tenha sido também creditado na outra conta. Além do mais, as transações não devem nunca interferir umas com as outras. Uma simples transação é normalmente executada na base de dados em uma linguagem como SQL nessa forma:

1. Inicie a transação (*Begin Transaction*)
2. Execute várias queries
3. Conclua a transação (*Commit Transaction*)

Durante o segundo passo nenhuma atualização é realmente visível para as outras transações. Depois do terceiro passo, as atualizações se tornam visíveis se a transação é finalizada com sucesso. Se uma das *queries* falha, o DMBS desfaz (rollback) a inteira transação ou somente a query que falhou. Este comportamento depende do DBMS em uso e como ele foi configurado. A transação pode também ser desfeita manualmente a qualquer momento antes de sua conclusão. Dentro de uma simples transação é possível trabalhar com mais de um banco de dados, até mesmo separado fisicamente. Neste caso, um outro procedimento é necessário como o *two-phase commit protocol* (PAOLO et al., 2003).

A maioria dos DBMS modernos são bancos de dados transacionais e gerenciam as transações muito bem. Entretanto, isto não é suficiente para construir aplicações robustas e ao mesmo tempo escaláveis. Um problema muito importante que ainda resta é como propagar transações por múltiplos métodos de acesso a dados. Por exemplo, em uma arquitetura de três camadas explicada no início do capítulo, um método da camada de negócio pode usar vários métodos da camada de acesso a dados dentro de uma mesma transação. A camada de acesso aos dados não precisa saber se foi chamada dentro da mesma transação mais de uma vez. O gerenciamento de transações pertence à camada de negócio. A camada de acesso aos dados participará das transações, mas não deve comandá-las, permitindo que diferentes métodos dessa camada sejam usados dentro de uma mesma transação. O Fragmento de Código 8 ilustra a situação.

```
public class BusinessClass
{
  public businessMethod()
  {
    trx.beginTransaction();
    dataAccessObject.dataAccessMethod1();
    dataAccessObject.dataAccessMethod2();
    trx.endTransaction();
  }
}
public class DataAccessClass
{
  public dataAccessMethod1()
  {
    //Código de acesso aos dados
  }

  public dataAccessMethod2()
  {
    //Código de acesso aos dados
  }
}
```

Fragmento de Código 8 – Propagação de uma transação através de múltiplos métodos de acesso a dados

No exemplo anterior, dois métodos diferentes de um objeto de acesso a dados foram chamados dentro da mesma transação usada pelo método do objeto de negócio. O objeto de acesso a dados apenas usa a transação controlada pelo objeto de negócio. Deste modo, as camadas da arquitetura são muito mais limpas: os

objetos de negócio demarcam transações abstratas sem envolver os recursos, e os objetos de acesso aos dados acessam os recursos transacionais sem se preocupar com os participantes nas transações.

Para chegar a um apropriado gerenciamento abstrato de transações, é necessária uma infra-estrutura que visa livrar o desenvolvedor de se preocupar com questões transacionais de baixo nível, deixando os objetos de acesso aos dados focados realmente no código de acesso aos dados. Tal estrutura pode ser construída de várias formas, mas deve ser o mais transparente e flexível possível.

3.3.1 Gerenciamento de transações no J2EE clássico

A solução clássica do J2EE é o uso de transações globais gerenciadas pelo container (o servidor de aplicação coordena as transações entre um ou mais recursos transacionais como bancos de dados). Com o gerenciamento global das transações, se os objetos de acesso aos dados usam recursos transacionais, o container coordenará os *commits* e *rollbacks* conforme o necessário.

A infra-estrutura na qual se baseia o gerenciamento global de transações é a implementação de JTA do container. O gerenciamento global de transações é geralmente efetuado usando EJBs com CMT (Container-Managed Transactions). Entretanto, o EJB CMT é de fato somente uma fina camada, apenas um entre vários modos de acessar o coordenador de transações do container. É também possível usar JTA diretamente. No código aplicativo, os objetos e recursos transacionais usados com JTA devem ser obtidos via JNDI. A configuração de recursos transacionais como *Data Soucers* JDBC não é especificada pelo J2EE: é especifico de cada container, seja por arquivos específicos de configuração, seja por consoles específicos de gerenciamento.

Se as transações globais J2EE são iniciadas via CMT ou diretamente através de JTA, o desenvolvedor não precisa se preocupar com o número de recursos usados nas transações. Nem os objetos de negócio, nem os descritores de *deployment* precisam saber se a transação expande mais de um recurso e então classifica-la como transação distribuída. O container é responsável por lidar com todos os casos. Portanto, as transações que usam apenas um recurso como um caso especial das transações distribuídas.

O EJB CMT parece uma boa solução mas possui problemas como:

- Os objetos precisam ser EJBs para se beneficiarem do gerenciamento declarativo de transações. Como discutido na seção 3.2.1, há um custo extra de codificação e *deployment* associado com isso. Objetos leves como POJOs não tiram proveito desse gerenciamento declarativo de transações.
- O gerenciamento declarativo de transações funciona bem na maior parte do tempo, mas pode tornar extremamente difícil a resolução de casos mais complexos.
- A única escolha é o gerenciamento global de transações pelo container. Isso é muito custoso se a aplicação acessa somente um recurso transacional.

As transações do container podem ser gerenciadas programaticamente, através da classe UserTransaction do JTA que pode ser localizada através do JNDI por um objeto simples Java ou qualquer outro objeto que rode dentro do servidor de aplicação. Isso é essencialmente a abordagem usada pelos EJB de tipo Bean Managed Transactions (EJB BMT). O EJB BMT agrega pouco valor em relação ao uso direto de JTA e não justifica o uso de EJBs.

3.3.2 Gerenciamento de Transações com Spring

O *framework* Spring provê uma implementação de uma infra-estrutura leve de transações que oferece a demarcação declarativa de transações para POJOs, sem amarrá-la a um modelo a componentes como EJB. Ela é o mais simples possível para o típico caso de persistência em um único banco de dados, podendo também fornecer o poder do gerenciamento de transações de alto nível (por exemplo, usando JTA para transações distribuídas).

A demarcação declarativa de transações é construída sobre o *framework* AOP do Spring, sendo necessário o uso de uma *bean factory* capacitada para AOP. A tecnologia EJB deve muito de seu sucesso a popularidade de alguns de seus serviços de *middleware*, como o CMT. A AOP oferece uma poderosa alternativa de se trabalhar declarativamente com transações sem carregar as desvantagens do EJB.

Este projeto não visa explicar o que é a programação orientada a aspectos (AOP). Para maiores detalhes veja (LADDAD, 2003). O conceito importante de AOP

que o *framework* Spring usa para gerenciar transações é a interceptação de métodos, para capturar as chamadas aos métodos definidos declarativamente como transacionais, realizando suas execuções dentro de uma transação.

4 A Ferramenta Adaptive Framework

O Adaptive Framework (AF) é uma infra-estrutura, criada pela empresa italiana Mainline Srl usada para estruturar a camada de lógica de negócio de aplicações corporativas. O AF visa reunir a experiência consolidada no desenvolvimento de aplicações corporativas de modo a reduzir o nível de conhecimento tecnológico necessário para criar esse tipo de software. Portanto, os desenvolvedores podem focar nos requisitos funcionais ao invés de aspectos tecnológicos (NAVONE, 2006).

O ponto central do AF é um gerador automático de código baseado na *partial-class generation* (*generation gap*) construído em Java. O gerador usa o motor Apache Velocity de maneira a auxiliar o uso de macros. Assim sendo, os templates de código são basicamente formados por código com macros de Velocity. Os arquivos de definição são documentos XML que contêm toda a informação das regras de negócio necessária para construir o código final. Para maiores detalhes sobre a geração automática de código veja o capítulo 2.

Além do gerador de código, o AF tem um conjunto de *templates* de geração, que são o resultado do aprendizado obtido nos projetos anteriores. O AF define uma base bem estruturada que reúne muitos dos conceitos discutidos no capítulo 3 com o escopo de formar uma arquitetura robusta e ao mesmo tempo flexível. Existem muitas versões diferentes dos *templates* com o intuito de oferecer funcionalidades singulares necessitadas por alguns clientes em projetos precedentes. Entretanto, os conceitos arquitetônicos essenciais são os mesmos. Os *templates* do AF são divididos em dois grupos principais: Java e .NET. Aqueles construídos em .NET usam a linguagem C# (NAGEL ET AL., 2005).

Essa infra-estrutura é chamada de Adaptive Framework, pois se adapta facilmente em diferentes cenários. Primeiramente, porque a arquitetura usada pelos *templates* pode se encaixar em um grande leque de projetos. Em segundo lugar, e mais importante, ela é baseada em geração automática de código. Consequentemente, se um cliente, em um novo projeto, tem uma necessidade não contemplada pelo conjunto de *templates* atual, é perfeitamente possível modifica-lo ou mesmo criar novos. Se a estrutura dos arquivos de definição em XML não inclui informações importantes da regra de negócio, é possível mudar sua composição. Mesmo que o cliente queira um projeto desenvolvido em outra linguagem de programação baseado em uma arquitetura completamente diferente, os projetistas

podem produzir novos *templates* e gerar o código aplicativo. As vantagens de usar os *templates* existentes é que eles já estão prontos para o uso e já foram bem testados em projetos anteriores. Consequentemente, é muito importante construir *templates* flexíveis.

As próximas seções explicam as características arquiteturais comuns à maioria dos *templates*. As idéias discutidas a seguir são aplicadas nos templates para Java e .NET.

4.1 Modelo a Componentes AF

Na terminologia AF, componente significa uma unidade de software que implementa uma ou mais interfaces de referência. Um componente pode ser composto por uma ou mais classes. A interface trabalha como um contrato entre o componente AF e a aplicação que o utiliza. O AF sempre usa interfaces ao invés de classes com o propósito de respeitar uma das mais importantes práticas da orientação a objetos que sugere a programação por interfaces (GAMMA et al., 1995). Isso livra o código da dependência com modo que os objetos são implementados, além de levar a uma melhor testabilidade.

O AF é baseado do padrão arquitetural Domain Model (para mais informações veja a seção 3.2). Nesta abordagem há basicamente dois tipos de componentes:

- **Componente de Dados:** encapsula uma única entidade de negócio ou uma hierarquia de entidades. Ele é conceitualmente a mesma idéia dos Entity Beans na terminologia EJB. Um componente de dado do AF usa o design pattern Active Record para fazer o mapeamento O/R e persistir os dados.
- **Componente de Serviço:** contém a lógica de negócio e implementa um ou mais casos de uso (FOWLER, 2003). O conceito equivalente na linguagem do EJB é a Session Bean.

A Figura 8 ilustra como o modelo a componentes do Adaptive Framework se encaixa dentro da modelo arquitetural de três camadas discutido no capítulo 3.

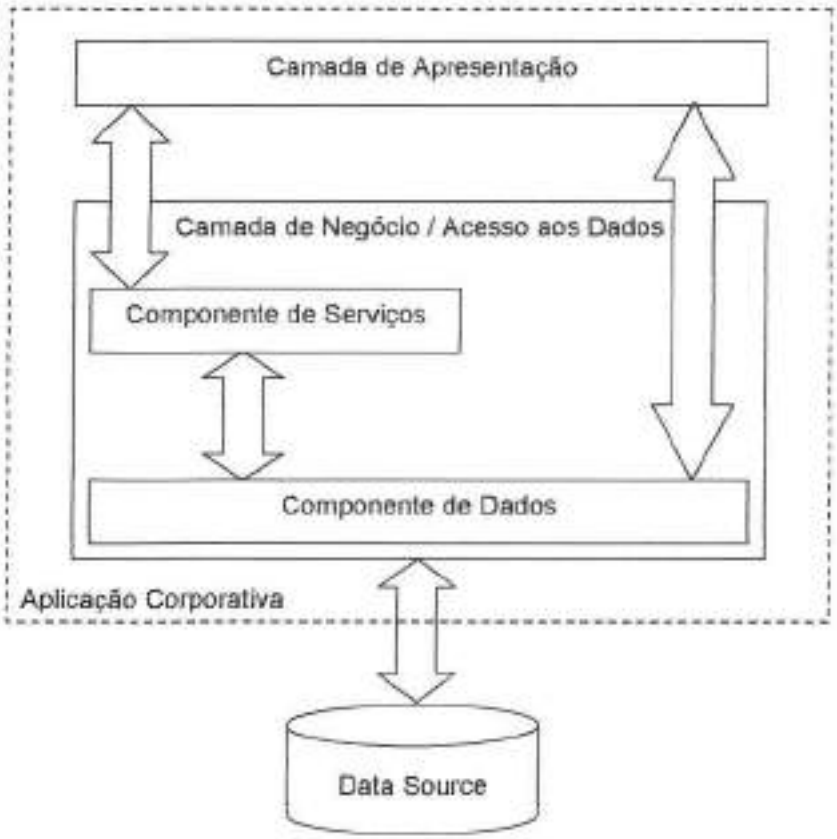

Figura 8 – Modelo de Componentes AF Inserido na Arquitetura de três Camadas

Quando se usa o AF, os componentes de dados e de serviços são gerados automaticamente. Para os componentes de dados, o gerador de código cria as funções CRUD (*create*, *read*, *update* e *delete*), a validação dos dados (campos obrigatórios, comprimento máximo, etc.), rastreamento de mudanças (todos os componentes de dados possuem o método *isChanged*), gerenciamento de transações, um esqueleto para cada método de negócio e o script de criação do banco de dados. Para os componentes de serviço, o gerador de código cria o gerenciamento de transações e o esqueleto para os métodos de negócio.

De agora em diante neste capítulo, alguns exemplos utilizarão o modelo de classes definido na Figura 9 para ilustrar o AF na prática. Todos os códigos da aplicação de exemplo são feitos em C# e usam a versão .NET do AF. Esta é a razão pela qual os nomes das propriedades e métodos nos segmentos de código começam com letras maiúsculas (por exemplo: ClientID e CalculateOrderPrice).

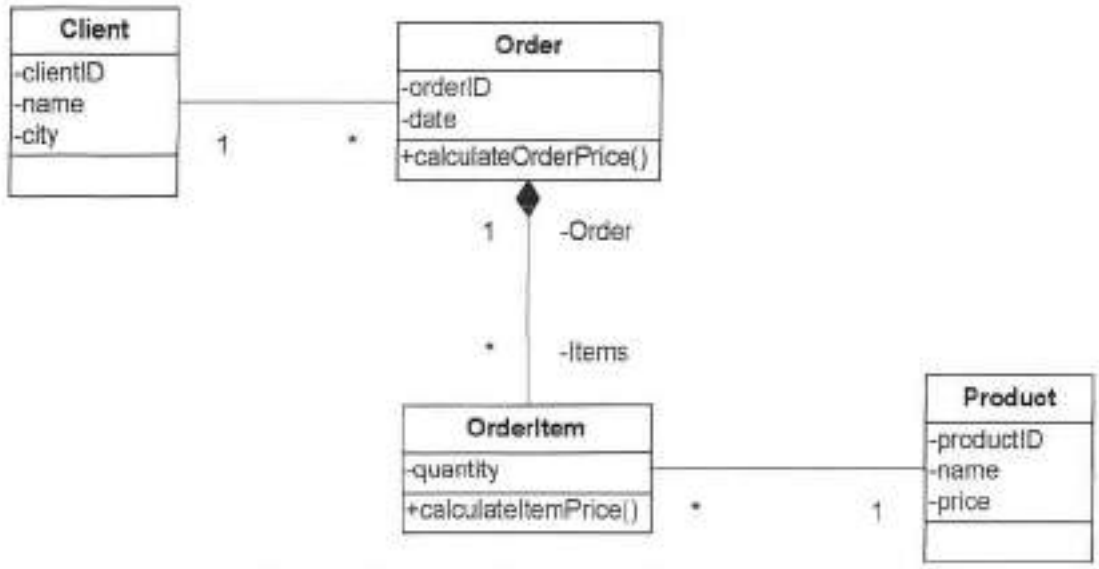

Figura 9 - Modelo UML usado pela aplicação de exemplo

As chaves estrangeiras não estão explicitadas no modelo, pois elas são representadas implicitamente pelas relações entre as classes. Algumas tabelas do banco de dados necessitariam de uma ou mais chaves estrangeiras. Por exemplo, a tabela que persiste a entidade `Order` teria a chave estrangeira `clientID`.

4.1.1 Estrutura básica de um Componente de Dados

Um componente de dados contém uma hierarquia de classes que representa uma entidade persistente, tipicamente uma linha em uma tabela de um banco de dados. A Figura 10 mostra um diagrama de classes simplificado da estrutura de um componente de dados. Esse diagrama oculta todas as interfaces, as classes base (as classes parciais geradas automaticamente) e outras classes que também compõem esse componente.

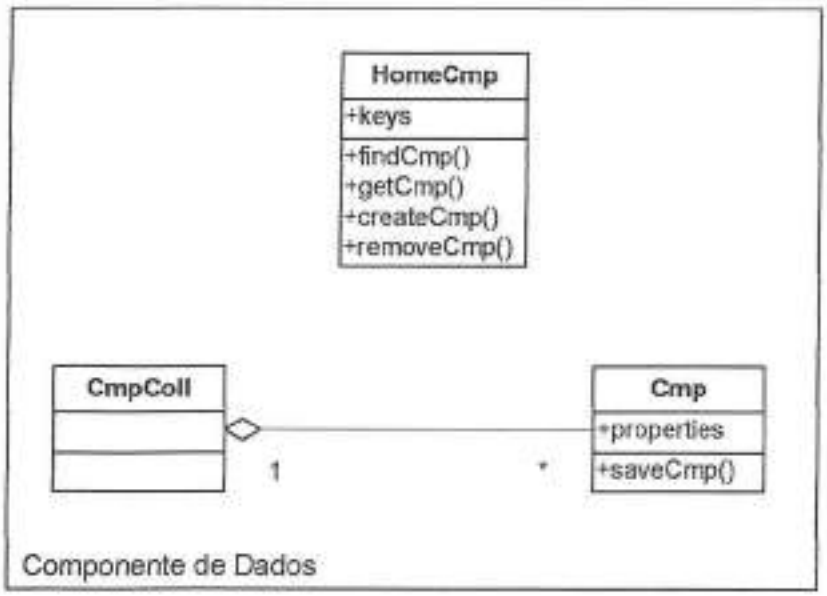

Figura 10 – Estrutura simplificada de um componente de dados

A entidade Client da aplicação de exemplo (Figura 9) teria uma HomeCmp_Client, uma Cmp_Client e uma CmpColl_Client geradas automaticamente. O mesmo é válido para todas as outras entidades.

O elemento mais importante de um componente de dados é a classe Cmp. Ela contém os dados de negócio e normalmente representa uma linha em uma tabela do banco de dados. Essa classe é responsável por persistir seus próprios dados (*active record)*. Isto é, ela contém código para salvar seus próprios dados em um banco de dados (na prática é um comando insert de SQL). Para salvar as alterações é necessário executar o método SaveCmp. Na aplicação de exemplo, a classe Cmp_Client gerada automaticamente teria propriedades como ClientID, Name e City. A Cmp_Order teria OrderID e Date além de ClientID (a referência para o cliente que possui o pedido, ou uma chave estrangeira na nomenclatura relacional). A idéia é a mesma para todas as classes Cmp que têm também um método SaveCmp.

Para recuperar, criar ou apagar entidades persistentes é necessário usar a classe HomeCmp. É o mesmo conceito da classe Home da plataforma EJB, mas ligeiramente diferente. Os métodos da classe HomeCmp definida na Figura 10 não recebem nenhum parâmetro. Ao invés disso, antes de executar tais métodos, é necessário definir valor para uma ou mais chaves de pesquisa. As chaves da classe HomeCmp não são a mesma coisa que as chaves primárias de uma entidade. Elas são apenas campos de um componente de dados usados para pesquisar instâncias de um componente de dados. As chaves são definidas no arquivo de definição XML

do componente e são geradas automaticamente nas classes. Na aplicação de exemplo a classe HomeCmp_Client poderia ter como chaves ClientID, Name e City por exemplo. O Fragmento de Código 9 exemplifica como recuperar todos os clientes cuja cidade é *New York*. É um código simplificado, pois não mostra como obter uma instância da classe HomeCmp_Client. O objeto client é de tipo CmpColl_Client (uma coleção de objetos de tipo Cmp_Client) e contém todos os clientes cuja cidade é *New York*.

```
homeCmp_Client.City = "New York";
clients = homeCmp_Client.FindCmp();
```

Fragmento de Código 9 – Como recuperar todos os clientes de New York

O método FindCmp é usado para recuperar todas as instâncias de um componente de dados que tem todas as propriedades iguais aos (ou começando com) valores atribuídos às chaves da classe HomeCmp. Na prática, a classe HomeCmp executa um e lógico entre as chaves na consulta SQL. Por exemplo, para recuperar todos os clientes chamados *John* cuja cidade é *New York*, é necessário atribuir os respectivos valores a ambas as chaves como mostrado no Fragmento de Código 10. Se não há nenhum valor atribuído a nenhuma chave, todos os clientes serão recuperados do banco de dados.

```
homeCmp_Client.City = "New York";
homeCmp_Client.Name = "John";
clients = homeCmp_Client.FindCmp();
```

Fragmento de Código 10 – Como recuperar todos os clientes chamados John de New York

O método GetCmp é usado para recuperar uma única instância de um Cmp. Para usar este método é necessário atribuir valores a todas as chaves primárias da classe HomeCmp. As chaves primárias são definidas no arquivo XML de definição do componente e geradas automaticamente nas classes. Na prática, uma chave primária é apenas uma chave de busca da classe HomeCmp, porém é obrigatório lhe atribuir antes de executar o método GetCmp, caso contrário uma exceção será lançada. O Fragmento de Código 11 mostra como usar o método GetCmp. Diferentemente do FindCmp, o GetCmp retorna um objeto de tipo Cmp ao invés de uma CmpColl. Portanto, no código o objeto client é de tipo Cmp_Client.

```
homeCmp_Client.ClientID = 523434;
client = homeCmp_Client.GetCmp();
```

Fragmento de Código 11 – Como utilizar o método GetCmp

Uma das fundações do AF é a programação por interfaces ao invés de classes. Portanto, cada classe analisada até agora é, na prática, utilizada através de sua interface de modo a desacoplar os objetos de suas dependências. A Figura 11 mostra o mesmo diagrama UML da Figura 10 acrescido das interfaces. Portanto, nos fragmentos de código anteriores, o objeto homeCmp_Client é na realidade usado através da interface IHomeCmp_Client, o objeto clients através da interface ICmpColl_Client e o objeto client através da interface ICmp_Client. Nas aplicações baseadas em AF, os desenvolvedores deveriam sempre programar usando interfaces ao invés de classes, senão muitas das vantagens da arquitetura do AF serão perdidas. Esta questão será discutida em detalhes nas próximas seções.

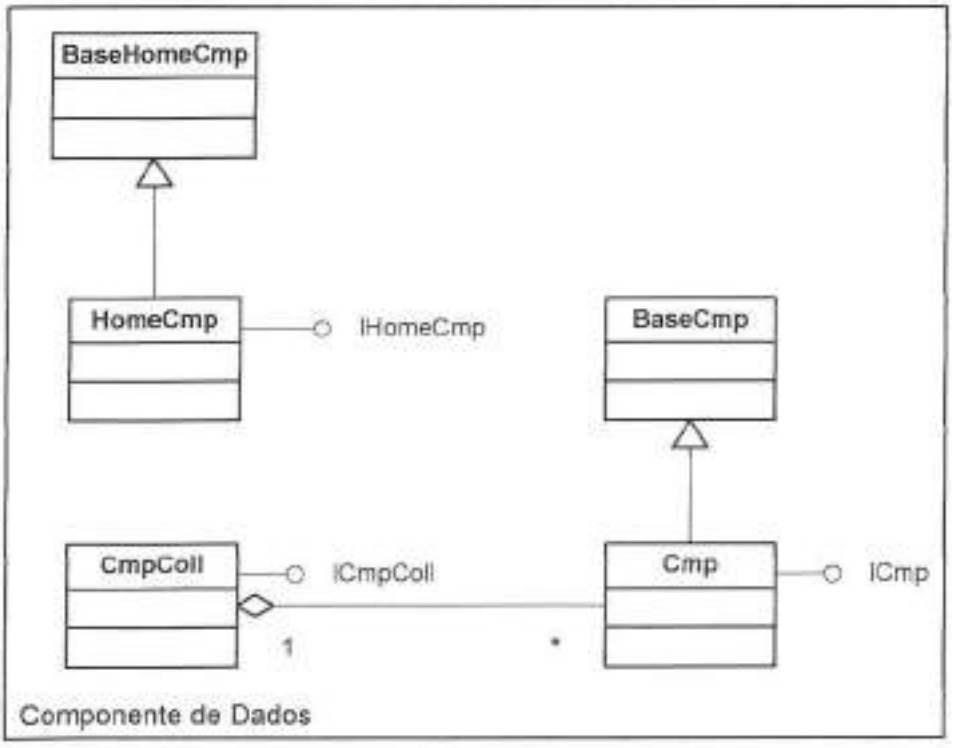

Figura 11 – Estrutura simplificada de um componente de dados com as classes base e interfaces

A Figura 11 ilustra também a classe base gerada automaticamente. As classes HomeCmp e Cmp são geradas apenas durante a primeira vez que o gerador é executado. As outras classes são criadas novamente a cada geração.

No começo do capítulo, foi explicado como um componente de dados encapsula uma simples entidade de negócio ou uma hierarquia de entidades de negócio. Na aplicação de exemplo, o componente Order contém a entidade OrderItem, pois no ambiente de negócio, um pedido real possui, de fato, uma lista de items. Na prática

o componente Cmp_Order contém uma coleção de objetos Cmp_OrderItem. No AF, uma coleção de objetos filhos (objetos aninhados) é implementada como uma ChildCmpColl que implementa a interface IChildCmpColl. Novamente, é recomendado usar sempre a interface ao invés da classe que a implementa.

A Figura 12 mostra uma estrutura simplificada do componente Order da aplicação de exemplo. Por razões de simplicidade, ele oculta algumas classes. O ponto importante a se perceber neste exemplo é que o componente Order contém um inteiro componente OrderItem. A classe CmpOrder tem um ChildCollCmp_Order_OrderItem que representa uma coleção de objetos Cmp_OrderItem. Uma classe ChildCmpColl é gerada automaticamente para cada relação entre componentes. Usualmente, essa classe implementa uma agregação ou composição de um diagrama de classes.

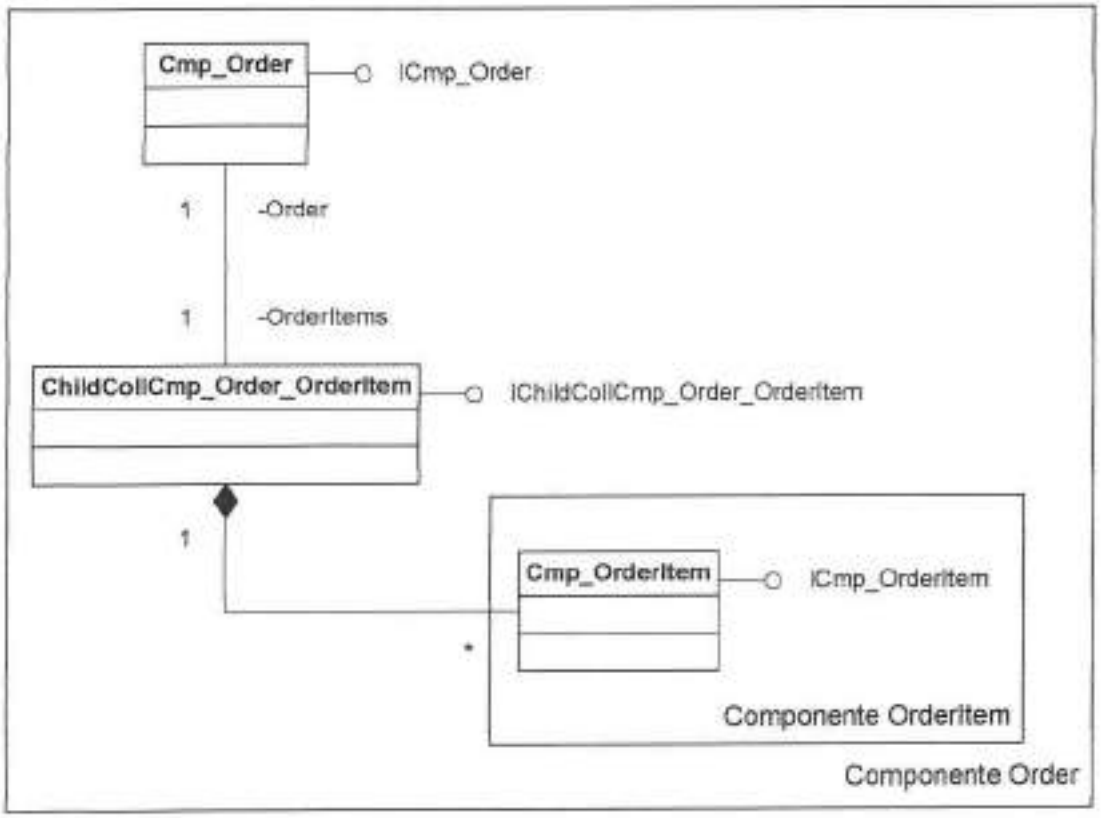

Figura 12 – Estrutura simplificada do componente Order

O Fragmento de Código 12 mostra um exemplo de como executar um ciclo foreach em uma coleção filha de componentes de dados. O código é escrito em C#. Um objeto ChildCmpColl pode ser usado como uma coleção normal pois implementa a interface IList na versão .NET e a interface Collection na versão Java. O mesmo é válido para as classes CmpColl. No exemplo, os itens de um pedido são usados

através da interface ICmp_OrderItem de modo a seguir boas práticas de programação. Uma coleção filha é definida no arquivo XML do componente de dados pai. Neste exemplo o nome da coleção definida no XML é OrderItems.

```
decimal totalPrice = 0;

foreach(ICmp_OrderItem orderItem in order.OrderItems)
{
  totalPrice += orderItem.CalculateItemPrice();
}
```

Fragmento de Código 12 – Exemplo de um ciclo foreach em uma coleção filha

Os componentes filhos podem existir sozinhos, sem um pai. Entretanto, um componente filho sabe quando é carregado de dentro de um pai. Neste caso, não é possível executar um SaveCmp no componente filho. Somente o pai pode ser salvo. Quando o SaveCmp é executado, todos os componentes filhos são persistidos automaticamente dentro da mesma transação. Em outras palavras, se uma atualização de um componente filho termina com um erro, nenhum componente é atualizado.

Quando um componente pai é apagado, todos os filhos são removidos. O Fragmento de Código 13 mostra como remover uma instância de um componente de dados. É obrigatório atribuir valores a todas as chaves primárias (igualmente ao método GetCmp), senão uma exceção será lançada.

```
homeCmp_Client.ClientID = 523434;
client = homeCmp_Client.RemoveCmp();
```

Fragmento de Código 13 – Como apagar uma instância de um componente de dados

Para economizar tráfico desnecessário ao banco de dados, somente os componentes que foram realmente alterados são atualizados. Cada componente de dados é responsável de rastrear as mudanças. Eles possuem um método isChanged que indica se foram modificados.

Para criar uma nova instância de um componente é necessário usar o método CreateCmp da respectiva classe Home. O construtor de um Cmp não deveria ser usado jamais no código aplicativo. Caso contrário, a injeção das dependências não funcionará (a injeção das dependências no AF é discutida na seção 4.3). O Fragmento de Código 14 mostra como criar uma nova instância de um componente de dados.

```
ICmp_Client client = homeCmp_Client.CreateCmp();
client.ClientID = 523434;
client.City = "New York";
client.Name = "John";
client.SaveCmp();
```

Fragmento de Código 14 – Como criar uma nova instância de um componente de dados

No exemplo anterior, um cliente é criado pelo método CreateCmp da classe Home. Então, todas as propriedades são valorizadas no objeto client. Por fim, o objeto client é salvado. Todos os componentes de dados sabem se são recém-criados ou foram recuperados do banco de dados. Esta informação é importante durante a execução do método SaveCmp que deve saber se deve ser criado no banco de dados (na terminologia SQL, um comando insert deve ser usado) ou se deve ser atualizado (um comando update deve ser executado). Sempre que o método SaveCmp é chamado, o componente de dados (incluindo todos os seus filhos) é validado. O mecanismo de validação checa coisas como campos obrigatórios e comprimentos máximos. Se existem erros de validação, uma exceção será lançada. É possível validar um componente de dados a qualquer momento através do método ValidateCmp. Ao invés de lançar uma exceção, esse método retorna os erros encontrados.

Para criar uma nova instância de um componente filho é necessário usar um procedimento diferente. Se o método CreateCmp do componente filho é usado, ele será independente do componente que deveria ser seu pai. O procedimento correto é mostrado no Fragmento de Código 15.

```
ICmp_OrderItem newOrderItem = order.OrderItems.NewCmp();
newOrderItem.ProductID = 9384;
newOrderItem.Quantity = 3;
order.OrderItems.Add(newOrderItem);
order.SaveCmp();
```

Fragmento de Código 15 - Como criar uma nova instância de um componente filho

No exemplo anterior, um novo item do pedido é criado pelo método NewCmp do objeto ChildCollCmp_Order_OrderItem. Cada classe ChildCollCmp possui um método NewCmp. Este método cria uma nova instância do componente filho e valoriza automaticamente todas as propriedades que dependem do componente pai (em geral chaves estrangeiras).

Para apagar um componente filho, é preciso primeiramente removê-lo da ChildCollCmp e então salvar o componente pai. Neste caso, é errado usar o método RemoveCmp da classe Home do componente filho. O Fragmento de Código 16 e o Fragmento de Código 17 mostram dois modos diferentes de apagar um filho na versão .NET do AF (a versão Java se diferencia no modo como as coleções são usadas). No primeiro exemplo, o método Remove recebe o índice do objeto orderItem. Em ambos os casos, é necessário executar o método SaveCmp no objeto order (não no orderItem).

```
ICmp_OrderItem orderItem = order.OrderItems[2];
order.OrderItems.Remove(orderItem);
order.SaveCmp();
```

Fragmento de Código 16 – Como remover um componente filho

```
order.OrderItems.RemoveAt(2);
order.SaveCmp();
```

Fragmento de Código 17 - Como remover um componente filho (2)

4.2 Estrutura básica de um componente de serviços

Um componente de serviço é um objeto que contém as funções que manipulam os componentes de dados. No AF todos os métodos de um componente de serviços são stateless. O componente de serviços é muito mais simples do que o componente de dados. Ele é composto basicamente por duas classes: a BaseSvcCmp (gerada automaticamente) e a SvcCmp (criada automaticamente na primeira geração). A Figura 13 mostra uma estrutura simplificada de um componente de serviço. Os componentes de serviço não possuem uma classe HomeCmp, pois não há senso em recuperar, criar, atualizar e apagar instâncias de um componente de serviço.

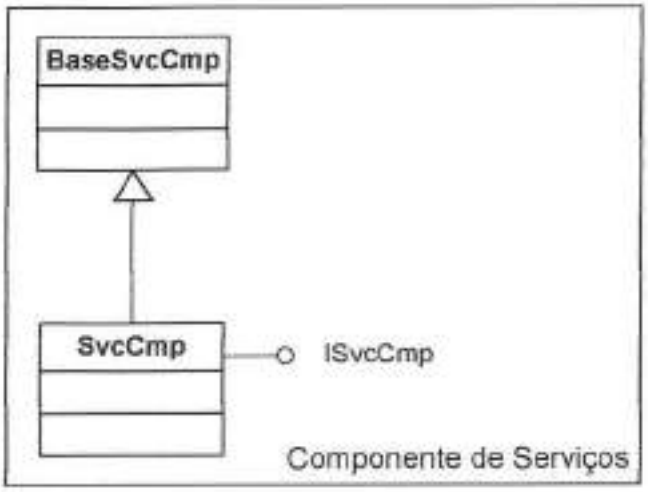

Figura 13 – Estrutura Simplificada de um Componente de Serviços

Os métodos de um componente de serviço podem funcionar dentro de uma transação. Isso é opcional e é definido no XML do componente. Quando um método é transacional, cada componente de dados usado dentro dele usa a mesma transação. Tudo é automático e transparente para o programador. Não é preciso iniciar e concluir explicitamente uma transação. Tal gerenciamento é feito pela classe BaseSvcCmp e é gerado automaticamente. A classe derivada (SvcCmp) contém o código escrito a mão que é, na prática, a lógica de negócio. A seção 3.3 discute o gerenciamento de transações em detalhes.

4.3 Gerenciamento de Recursos do AF

Até agora, foi analisado como utilizar os componentes do AF. O ponto de entrada para usar um componente de dados é sua classe HomeCmp. Os componentes de serviços não têm uma classe HomeCmp e podem ser usados diretamente. Entretanto, falta saber como pegar uma instância de uma classe HomeCmp ou um componente de serviços e como configurar suas dependências (como conexões ao banco de dados) nos componentes.

Os componentes do AF rodam dentro de um container leve chamado BLC (Business Logic Container), que é basicamente uma *object factory* criada pela empresa Mainline Srl. Esta *factory* é baseada em Inversão de Controle (IoC) e injeção de dependências. Portanto, quando um objeto é criado, a *factory* injeta todas suas dependências. Estas dependências estão sujeitas ao modo que a factory é

configurada e os plug-ins que estão instalados. IoC é explicado em detalhes na seção 3.1.1.

O acesso à *factory* do BLC pode acontecer de um modo direto ou através de uma sessão do usuário. O Fragmento de Código 18 mostra como acessar a *factory* diretamente. Este é um procedimento simples: o código aplicativo instancia o BlcEngine e recupera a factory. O Fragmento de Código 19 mostra como recuperar a *factory* de uma sessão. Primeiramente, o código aplicativo instancia o BlcSecureEngine que, em contraste com o BlcEngine, tem mecanismos de segurança integrados. Então é necessário logar-se usando a sessão (IUserSession). Finalmente, o método GetObjectFactory da sessão retorna a *factory*. O acesso à *object factory* é feito somente com as credenciais do usuário.

```
BlcEngine blc = new BlcEngine();
IObjectFactory factory = blc.GetObjectFactory();
```

Fragmento de Código 18 – Como recuperar diretamente a object factory

```
BlcSecureEngine blc = new BlcSecureEngine();
IUserSession ses = blc.GetUserSession();
ses.Signon("user", "password");
IObjectFactory factory = ses.GetObjectFactory();
```

Fragmento de Código 19 – Como recuperar a object factory através de uma user session

O gerenciamento da segurança do BLC é feito por *plug-ins* (dependentes da implementação) e podem lidar com autenticação (identificação do usuário) e autorização (permissão de acessar alguns componentes e o modo que os usuários podem usá-los).

Uma vez que a factory foi obtida, seu uso é o mesmo para ambos os tipos de aquisição (do BlcEngine e BlcSecureEngine). O Fragmento de Código 20 mostra como recuperar um objeto usando a *object factory* do BLC. Nesse exemplo, o objeto recuperado é classe home do componente Client e está registrada na factory como HomeCmp_Client devido a *string* que é passada ao método GetObject. A *factory* sempre retorna um objeto de tipo object (Object em Java). Portanto, um cast é necessário. O fragmento de código mostra a boa prática de programar através de interfaces ao invés de classes, pois usa a interface IHomeCmp_Client ao invés da classe HomeCmp_Client diretamente.

```
IHomeCmp_Client clientHome =
        (IHomeCmp_Client)factory.GetObject("HomeCmp_Client");
```

Fragmento de Código 20 – Como recuperar um objeto da object factory do BLC

4.3.1 Tipos de objetos dentro do container

A *factory* do BLC pode gerenciar qualquer tipo de objeto. O objeto não precisa ser necessariamente um componente. Há dois grupos de objetos controlados pela *factory*:

- Objetos aplicativos
- Serviços BLC

Os objetos aplicativos não precisam implementar nenhuma interface especial. Para usar a injeção de dependências, o objeto precisa apenas de uma propriedade publica para cada dependência (na versão .NET) ou setters (na versão Java). Portanto, o BLC utiliza a Setter Injection ao invés da Constructo Injection.

Os serviços BLC são todos os objetos que implementam a interface IScvBase (mostrada no Fragmento de Código 21). Eles são *singletons* (GAMMA et al., 1995) que são instanciados em uma ordem predefinida quando o BLC é iniciado. Depois de o BLC ter sido iniciado, os serviços podem ser injetados nos objetos aplicativos. Exemplos de serviços são a própria *object factory*, *loggers* de exceções, gerenciadores de transações e assim por diante.

```
public interface ISvcBase
{
   void Start();
   void Stop();
   bool IsStarted { get; }
}
```

Fragmento de Código 21 – Interface ISvcBase

5 O Adaptive Framework na Prática

A versão Java do Adaptive Framework foi usada em muitas aplicações com uma vasta gama de requisitos, e foi melhorada de forma a satisfazer todas essas necessidades. Por outro lado, a versão .NET é muito mais nova. Ela começou a ser usada dois anos antes de este projeto ser concluído. Durante este tempo, muitos defeitos foram descobertos e muitas funcionalidades foram adicionadas. Além disso, algumas questões arquiteturais e estruturais foram modificadas para melhorar o *framework*. Este texto é fruto da experiência adquirida durante esse período ao desenvolver muitas aplicações com a versão .NET do AF.

Este capítulo mostra um exemplo realístico de uma aplicação baseada em AF. Por razões de privacidade, a aplicação discutida aqui não é nenhum dos projetos desenvolvidos para os clientes da Mainline Srl. Ao invés disso, é uma aplicação de exemplo que segue exatamente a mesma arquitetura dos projetos reais. Isto é, aplicações smart client que se comunicam com web services.

A seção 5.1 explica o que é um smart client e discute algumas questões relacionadas ao desenvolvimento desse tipo de aplicações. Em seguida, a seção 5.2 esclarece como o AF pode auxiliar o desenvolvimento de tal tipo de software. Por fim, a seção 5.3 mostra como uma aplicação baseada em AF é desenvolvida na prática. Embora o projeto de exemplo seja baseado na versão .NET do AF, muitos conceitos são os mesmos para a versão Java.

5.1 Arquitetura Smart Client

Smart Client é o nome dado a aplicações que são disponíveis na web, não requerem instalação (ou oferecem instalação automática), atualizam-se automaticamente e têm a aparência de aplicações para *desktop* (SMART, 2004). A Microsoft usa essa terminologia para se referirem a aplicações .NET disponíveis para o navegador Internet Explorer.

As aplicações Smart Client são uma alternativa às aplicações *thin client* (como navegadores web). Elas podem prover aos usuários uma interface rica, a habilidade de trabalhar off-line e um modo de tirar proveito do hardware e software locais.

Os Smart Clients podem ser projetados para combinar os tradicionais benefícios de uma aplicação *rich client* com os benefícios de gerenciamento dos *thin clients*.

Entretanto, para perceber totalmente os benefícios de uma aplicação Smart Client, os projetistas precisam considerar certo número de questões arquitetônicas. As próximas seções discutem tais questões.

5.1.1 Data Handling

Os Smart Clients permitem que dados e lógica sejam distribuídos para o computador cliente, enquanto *Thin Clients* tendem a manter os dados e a lógica centralizados no servidor Web e em outros serviços de *back-end*. Embora a abordagem Smart Client possa levar a aplicações mais eficientes, sem envios de informações ao servidor para determinar próximos passos, é importante considerar que a aplicação e seus dados são distribuídos de forma mais abrangente do que com aplicações *Thin Client*, além de ser necessário modificar a arquitetura de modo coerente.

Se a aplicação Smart Client provê a habilidade de modificar os dados localmente, as mudanças do cliente devem ser sincronizadas com a aplicação lado servidor em um momento posterior. Neste caso, é necessário decidir como a aplicação cliente pode lidar com conflitos nos dados e como pode rastrear as mudanças que ainda não foram enviadas ao servidor.

Os dados trabalhados pelo Smart Client podem ser divididos em dois grupos principais: estáticos (somente leitura) e transientes. O primeiro tipo significa que o dado não é modificado pelo cliente e é usado como referência (por exemplo, informações de produtos, lista de preços, opções de entrega, textos de ajuda, etc.) O segundo tipo de dado é aquele que pode ser modificado tanto no cliente quanto no servidor. Geralmente, os dados transientes são alterados segundo resultado direto ou indireto das entradas e manipulação do usuário. Os dados transientes podem ser adicionados, modificados e apagados.

Os Smart Clients normalmente precisam de uma *cache* local, seja de dados estáticos ou transientes. A *cache* tem o potencial de melhorar o desempenho da aplicação e prover dados para o trabalho offline. No entanto, é preciso considerar cuidadosamente quais dados serão armazenados na *cache*, como tais dados são gerenciados e o contexto no qual esses dados podem ser usados. Há basicamente dois tipos de *cache*: curto e longo prazo. O primeiro caso é composto pelos dados em memória, não persistidos. No segundo caso, os dados são persistidos no cliente.

Com a *cache* de longo prazo é possível usar a aplicação mesmo sem uma conexão com o servidor.

Um importante problema com Smart Clients é que as mudanças nos dados contidos no servidor podem ocorrer antes de alguma alteração no cliente tenha sido sincronizada. É essencial ter um mecanismo para assegurar que quando os dados são sincronizados, qualquer conflito seja tratado apropriadamente e o dado resultante seja consistente e correto. A habilidade de permitir que os dados sejam atualizados por mais de um cliente é conhecida como concorrência de dados e pode ser categorizada em dois grupos principais: concorrência pessimista (que permite que um cliente trave o dado para prevenir que nenhum outro cliente o modifique antes de usá-lo completamente) e concorrência otimista (que não trava o dado, mas verifica se a versão do dado sendo atualizado é a mesma possuída pelo servidor).

5.1.2 Comunicação Remota

Por definição, os Smart Clients precisam se conectar e comunicar com outros recursos. Os elementos de uma arquitetura distribuída podem ser acoplados em diferentes níveis. A natureza desse acoplamento entre clientes e serviços pode afetar muitos aspectos do projeto do Smart Client, incluindo interoperabilidade, capacidades *offline*, desempenho da comunicação na rede, implantação e manutenção.

Sistemas fortemente acoplados normalmente oferecem comunicação direta entre objetos, com o objeto no cliente tendo conhecimento detalhado do objeto remoto. Tal tipo de acoplamento pode atrapalhar atualizações independentes no cliente ou servidor. Dado que sistemas fortemente acoplados envolvem comunicação direta entre objetos, estes podem interagir mais frequentemente do que em sistemas fracamente acoplados. Um exemplo de comunicação remota fortemente acoplada é o RMI (MCNIFF; PITT, 2001), usado, por exemplo, na tecnologia EJB.

Sistemas fracamente acoplados são geralmente baseados em mensagens, com o cliente e o serviço remoto que desconhecem o modo como o outro é implementado. Qualquer comunicação entre o cliente e o servidor é ditada pela estrutura da mensagem. As implementações do cliente ou do serviço podem mudar

sem o medo que o outro pare de funcionar. Um exemplo desse tipo de comunicação é a tecnologia *Webservice*.

5.2 Suporte do AF para Smart Clients

O Adaptive Framework não oferece um suporte específico para Smart Clients. Entretanto, ele possui algumas características que auxiliam os clientes remotos. As aplicações baseadas no AF podem ser projetadas de dois modos: arquitetura co-locada ou arquitetura distribuída.

A Figura 14 ilustra a arquitetura co-locada e a Figura 15 mostra a arquitetura distribuída de aplicações baseadas no Adaptive Framework. No primeiro caso todas as camadas da aplicação rodam dentro da mesma Common Language Runtime (CLR) (DUFFY, 2006). Diferentemente, no caso da arquitetura distribuída, as classes de apresentação rodam na CLR do cliente enquanto as classes de lógica de negócio e acesso a dados rodam na CLR do servidor. Hoje em dia, a versão Java do AF pode ser construída apenas com a arquitetura co-locada. Portanto, todas as classes rodam dentro da mesma Java Virtual Machine (JVM).

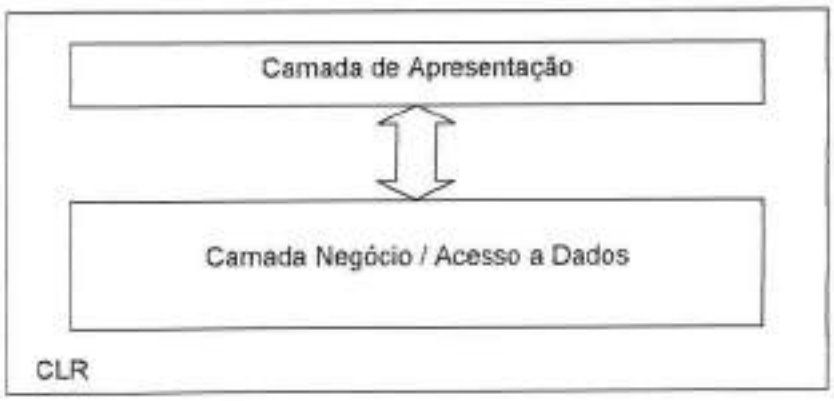

Figura 14 – Arquitetura AF Co-locada

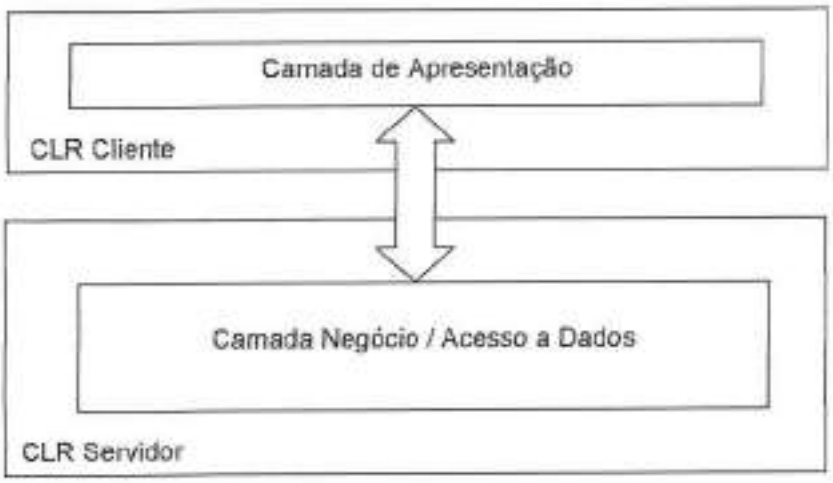

Figura 15 – Arquitetura AF Distribuída

Os clientes baseados no AF se comunicam com o servidor através de *Webservices* que são gerados automaticamente. Esses clientes geralmente são .NET Windows Applications comuns ou Smart Clients. A arquitetura co-locada é normalmente utilizada em aplicações Web cuja lógica de apresentação reside fisicamente junto à lógica de negócio e acesso a dados. Entretanto, uma aplicação web pode também usar uma arquitetura distribuída para acessar serviços ou componentes de dados fisicamente separados. Adicionalmente, uma Windows Application pode ser construída em uma arquitetura co-locada e acessar diretamente o banco de dados. As duas últimas abordagens são incomuns. A decisão de qual arquitetura usar tem de ser feita pelos projetistas.

As duas abordagens descritas anteriormente podem coexistir no mesmo projeto. Por exemplo, pode haver uma aplicação Web (com uma arquitetura co-locada) e um Smart Client (com uma arquitetura distribuída) que usam os mesmos componentes do AF. Por esta razão, o modelo de componentes deve trabalhar com ambas as arquiteturas de um modo transparente para o desenvolvedor. O código deve ser escrito sem se preocupar se os componentes são usados remota ou localmente. Em outras palavras, o código precisa ser portável simplesmente re-configurando a aplicação, sem a necessidade de compilar nada.

O uso de uma *object factory* e a programação através de interfaces podem auxiliar muito a portabilidade das aplicações. O primeiro passo é desacoplar a lógica de negócio da lógica de acesso aos dados. Para isso, os componentes são divididos em duas partes mostradas na Figura 16. Esse diagrama ilustra como a classe Cmp (da Figura 11) é dividida em duas classes para aumentar a portabilidade. A classe

Cmp contém a lógica de negócio (validação dos dados, métodos de negócio, etc) e a classe DalCmp contém a lógica de acesso ao banco de dados. DAL significa Data Access Layer (camada de acesso aos dados) e o DalCmp é um DAO (Data Access Object – Objeto de Acesso aos Dados).

Figura 16 – Estrutura simplificada de um componente de dados com a classe Dal

A *object factory* é responsável por conectar o objeto Cmp ao objeto DalCmp. Por essa razão (somada ao fato que a classe Cmp usa sempre a interface IDalCmp) é muito fácil modificar a implementação do DAO de modo a acessar a verdadeira lógica de acesso aos dados através de um *Webservice*. Tudo é feito em tempo de execução, sem compilação, usando o arquivo de configuração do BLC. Na terminologia do AF, a classe que acessa os dados através de um *Webservice* é chamada ProxyDalCmp e é mostrada na Figura 17.

Figura 17 - Estrutura simplificada de um componente de dados com a classe ProxyDal

Posto que ambos os DAOs implementam a interface IDalCmp, a classe Cmp não sabe se está usando um DAO que acessa diretamente o banco de dados ou um Webservice. Consequentemente, os desenvolvedores podem codificar a lógica de negócio sem se preocupar com detalhes de *deployment*. Os componentes AF trocam dados entre o cliente e o servidor através de Datasets .NET (DUFFY, 2006). A razão dessa escolha é que se trata de um formato flexível de troca de dados em formato de tablea. Além disso, o framework .NET oferece um grande suporte aos Webservices que usam Datasets.

A Figura 18 mostra um diagrama de seqüência simplificado que ilustra o que acontece quando o método SaveCmp é executado em uma arquitetura distribuída. O diagrama oculta algumas operações dentro da classe Cmp (validação de dados, por exemplo). Depois de verificar se os dados foram realmente modificados, a classe

Cmp prepara um Dataset que contém todos os dados a serem atualizados (seus próprios dados e os de seus filhos). O Dataset é então passado a classe ProxyDalCmp através do método SaveCmpBuffer. O *proxy* sabe como chegar ao *WebService* wsRemoteDal e lhe encaminha o Dataset. O *Webservice* wsRemoteDal (no lado servidor) instancia a implementaçao real da classe DalCmp e chama seu método SaveCmpBuffer passando o mesmo Dataset. Por fim, a classe DalCmp executa o comando *update* (ou *insert*) no banco de dados.

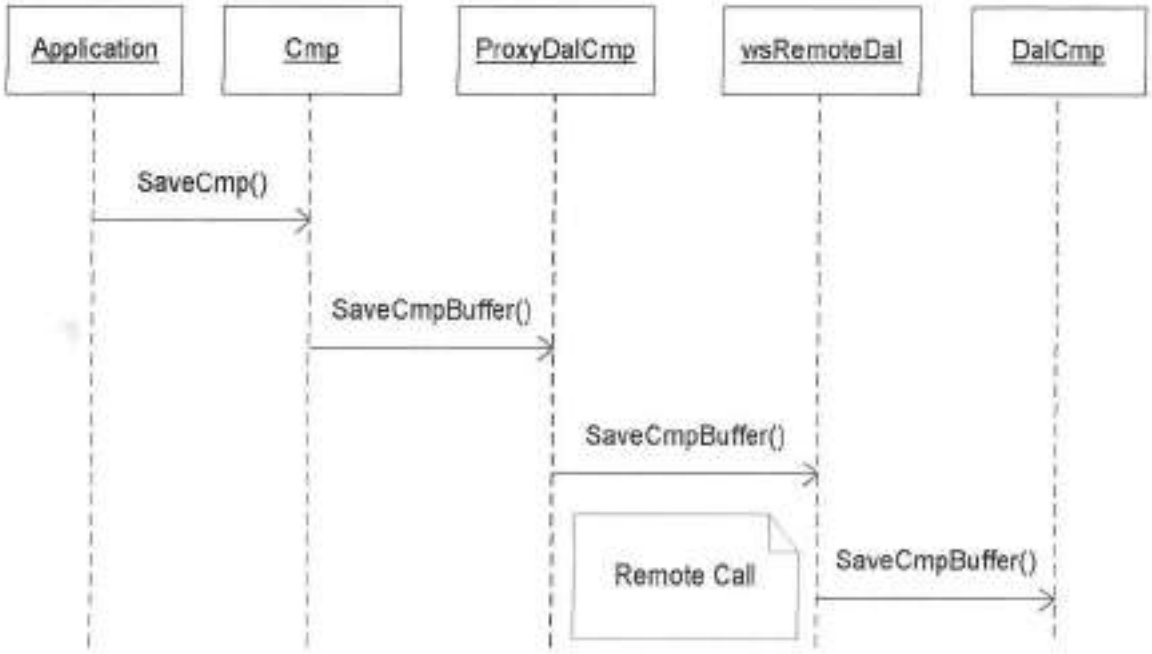

Figura 18 – Diagrama de Seqüência simplificado de uma operação SaveCmp na arquitetura distribuída

Para cada componente de dados gerado, são também criadas as classes Cmp, DalCmp e ProxyDalCmp. Entretanto, o wsRemoteDal é sempre o mesmo. Ele não é gerado automaticamente, mas é parte da biblioteca BLC. O mesmo *Webservice* pode ser usado por diferentes componentes de dados, pois sua interface e suas funções são sempre idênticas. Ele recebe um Dataset e o nome de um componente. Usando esse nome, ele é capaz de recuperar (usando a *object factory* do BLC) a implementação real da classe DalCmp para o componente. Os Datasets que chegam ao *Webservice* possuem estruturas diferentes, mas isso não importa pois o *Webservice* apenas o encaminha ao DalCmp correto.

Usando a *object factory* é também possível substituir o objeto DalCmp por um objeto que acesse a cache local do cliente. Para tal propósito, o AF pode gerar automaticamente um DalCmp que acesse os dados na cache localizada no sistema de arquivos do cliente. A classe também é chamada DalCmp, mas pertence a um diferente namespace (DUFFY, 2006) (dalcmgr ao invés de daldb). Tudo é transparente aos desenvolvedores. A seção 5.2.1 explica os mecanismos de cache em detalhes.

Os componentes AF de serviços também podem ser usados em modo distribuído. Da mesma forma que os componentes de dados, o AF pode gerar automaticamente um *proxy* para um componente de serviços. Entretanto, na estrutura do componente de serviço não existe o conceito de DAO. Nesse caso, um *proxy* substitui a classe SvcCmp (mostrado na Figura 13). A classe *proxy* implementa a interface ISvcCmp mas não contém nenhuma lógica de negócio. Ela contém somente a lógica de acesso à real implementação do componente de serviço através de um *Webservice*. A Figura 19 mostra um diagrama de seqüência simplificado que ilustra o que acontece quando um método de um componente de serviços é executado em uma arquitetura distribuída.

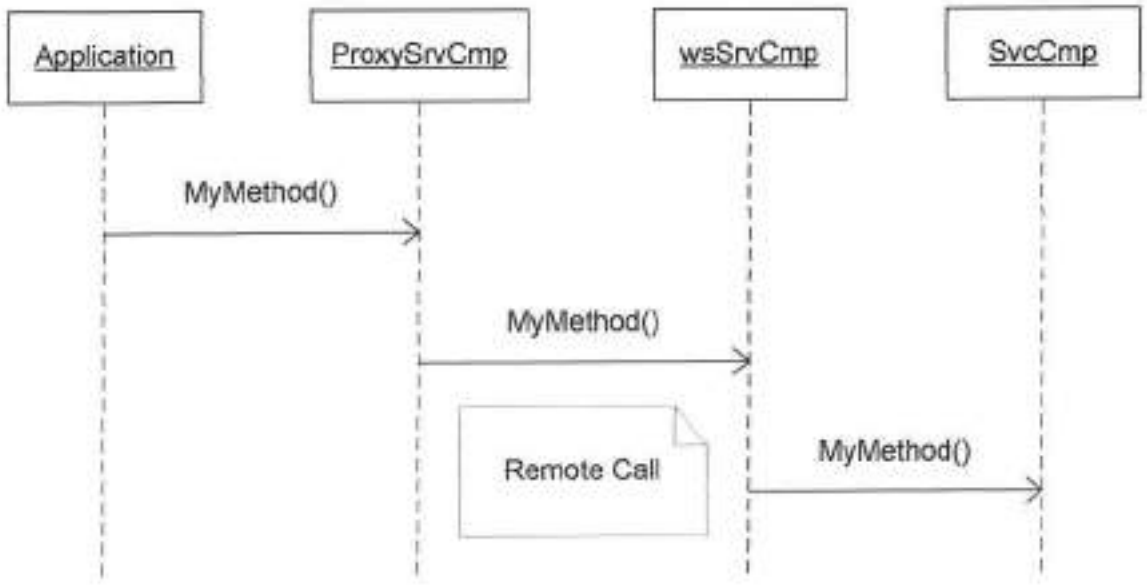

Figura 19 - Diagrama de Seqüência simplificado de um método de um componente de serviços na arquitetura distribuída

Ao contrário dos componentes de dados, os componentes de serviço não possuem uma classe DalCmp. Então, o cliente remoto acessa diretamente a classe ProxySrvCmp que implementa ISvcCmp. A classe *proxy* não contém os métodos reais, mas somente a chamada ao *Webservice*. Outra diferença entre os dois tipos de componente é que na arquitetura distribuída existe um *Webservice* para cada componente de serviços. Isso ocorre porque a interface de cada *Webservice* de um componente de serviço é diferente. O wsSrvCmp deve ter todos os métodos do componente de serviços, enquanto o wsRemoteDal precisa ter somente o método SaveCmpMethod que recebe sempre os mesmos parâmetros.

Na arquitetura distribuída existem dois BLCs em execução (um no cliente e um no servidor). Consequentemente, há duas *object factories* configuradas em modos diversos. A *factory* do cliente é configurada para usar *proxies* (ProxyDalCmp e ProxySvcCmp) ou o DalCmp da *cache*. A *factory* do servidor é configurada para usar DalCmp e SvcCmp (as implementações reais).

5.2.1 Cache Local dos Componentes de Dados

A versão .NET do Adaptive Framework oferece suporte completo para o gerenciamento de caches de somente leitura. Este suporte é garantido pelo Cache Manager usado em combinação com os templates de código que geram automaticamente as classes dalcmgr.DalCmp.

O Cache Manager do AF é um serviço BLC. Portanto, ele inicia quando o container é iniciado e carrega todos os dados dos componentes configurados para irem para a cache. A Figura 20 mostra um diagrama de seqüência simplificado que ilustra como a cache local é carregada.

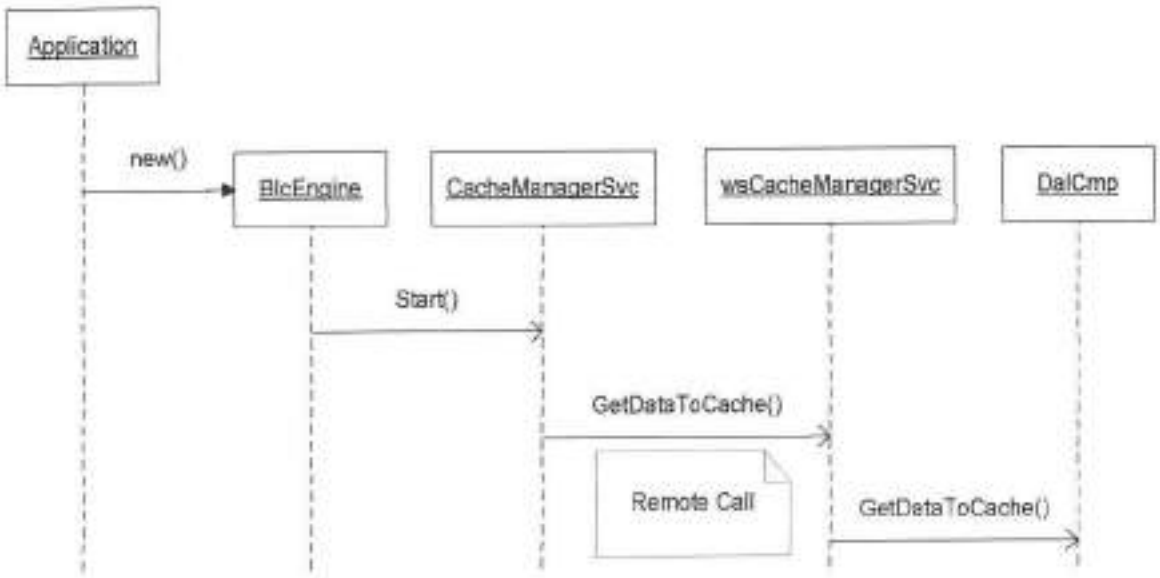

Figura 20 – Diagrama simplificado do processo de carregamento da cache

Quando o cliente remoto instancia o BlcEngine, este inicia todos os serviços configurados. O serviço *Cache Manager* precisa estar configurado com a lista de todos os componentes que se encontram na cache. O Fragmento de Código 22 mostra um exemplo da configuração de componentes do *Cache Manager*. A parte Value da *tag* contém o nome do componente e sua chave primária. Esses dois valores são separados por dois pontos. Por exemplo, o primeiro item da cache é ProductInformation e sua chave primária é cProductInformation. Se o componente tem mais de uma chave primária elas serão separadas por vírgula.

```
<add key="svc.CacheMgrSvc.refprmv.1.CacheEntry"
value="ProductInformation:cProductInformation" />
<add key="svc.CacheMgrSvc.refprmv.2.CacheEntry"
value="PriceList:cPriceList" />
<add key="svc.CacheMgrSvc.refprmv.3.CacheEntry"
value="ShippingOption:cShippingOption" />
```

Fragmento de Código 22 – Exemplo de configuração do cache manager

Uma vez iniciado, o *Cache Manager* chama, para cada componente configurado, o método GetDataToCache do *Webservice* wsCacheManagerSvc passando o nome do componente a ser carregado em cache e um *timestamp* da ultima atualização. O *Webservice* redireciona o pedido à correspondente classe DalCmp que recupera os dados do banco de dados. Portanto, somente os registros que foram modificados

após a ultima sincronização da cache são transferidos. Se a cache está vazia (é a primeira vez que a aplicação é executada) todos os registros serão transferidos. Por fim, os dados são salvos no sistema de arquivos do cliente.

A Figura 21 mostra o que acontece quando a aplicação executa o método FindCmp de um componente em cache. O mesmo diagrama também é válido para o método GetCmp. A classe HomeCmp executa o método LoadCmpBuffer da classe dalcmgr.DalCmp que lê os dados em cache, retornando-os em forma de um Dataset. Então, a classe HomeCmp transforma o Dataset em uma coleção de objetos Cmp (ou um único Cmp no caso do método GetCmp) e o retorna para a aplicação.

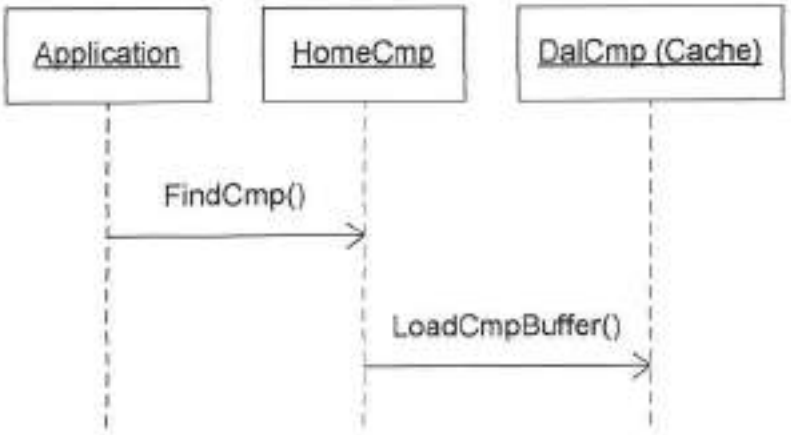

Figura 21 – Diagrama de seqüência simplificado do método FindCmp na cache

O AF gera automaticamente o dalcmgr.DalCmp. Este DalCmp é diferente para cada componente. Entretanto, o Cache Manager (que carrega a cache) é somente um para todos os componentes. O Cache Manager é parte da biblioteca do BLC. Além disso, o Webservice wsCacheManagerSvc é o mesmo para todos os componentes. A razão, como no caso do wsRemoteDal, é que a interface do Cache Manager e do Webservice é igual para todos os componentes.

5.3 Um Exemplo de Projeto Baseado no AF

Esta seção explica como um sistema de software baseado no AF é desenvolvido na prática. A aplicação de exemplo é construída sobre a versão .NET do Adaptive Framework e usa a arquitetura Smart Client. Por razões de privacidade, não se trata de um projeto real vendido para um cliente. Ao invés disso, é a implementação do

diagrama de classes da Figura 9. Todos os exemplos de código desta seção foram obtidos da aplicação de exemplo real. Portanto, existem muitos comentários em italiano, pois ela foi gerada automaticamente.

O exemplo é uma aplicação simples que gerencia clientes, produtos e pedidos. Um cliente pode ser associado a um ou mais pedidos. Um pedido possui um cabeçalho (composto somente pela data neste exemplo) e contém alguns itens. Um item de um pedido é associado com um produto e tem uma certa quantidade de produtos.

O diagrama de classes da figura é traduzido em quatro componentes AF: Client (cliente), Order (pedido), OrderItem (item do pedido) e Product (produto). O componente Client não contém uma coleção de componentes Order. Ao invés disso, cada instância de Order possui o ClientID da instância Client com a qual é associada. Para recuperar todos os pedidos de um cliente específico, é necessário utilizar a classe Home do componente Order definindo a propriedade ClientID. O mesmo ocorre entre OrderItem e Product.

5.3.1 Arquivo XML de Definição de um Componente de Dados e o Código Gerado

Esta seção mostra como escrever os arquivos XML de definição de um componente de dados e como é o código gerado. Aqui não é mostrado o arquivo inteiro, mas somente as partes mais importantes de cada um. O Fragmento de Código 23 mostra o esqueleto de um arquivo de definição de um componente de dados. Por razões de simplicidade, ele oculta as chaves, propriedades e constantes do componente.

```
<!DOCTYPE component SYSTEM "C:/blc_runtime/dtd/datacmpsc.dtd">
<component Id="1001" Name="Client" Type="DATA"
MapOnTable="Clients" MapOnTablePrefix="" RecVerField="RECVER"
LastChangeField="dtAgg" Project="ExampleProject"
Package="exampleProject" CodApp="1000" InsAllowed="Y"
UpdAllowed="Y" RmvAllowed="Y" Description="Client Entity" >

  <!- Propriedades, chaves, constantes e métodos do
     componente -->

</component>
```

Fragmento de Código 23 – Esqueleto de um arquivo XML de definição de um componente de dados

A primeira linha do exemplo anterior indica o arquivo DTD para os componentes de dados. Há um arquivo diferente para os componentes de serviços. A tag component define todas as características para o componente como um todo. A Tabela 2 explica cada propriedade dessa tag.

Propriedade	Descrição
Id	Identificador unívoco do componente.
Name	Nome do componente.
Type	Tipo de componente: Data (para o de dados) ou Service (para o de serviço).
MapOnTable	Nome da tabela no qual o componente é mapeado.
MapOnTablePrefix	Prefixo da tabela no qual o componente é criado (usado para a geração do script DDL).
RecVerField	Nome da coluna usada no gerenciamento do optmistic lock.
LastChangeField	Nome da coluna que contém a data e hora da última alteração do componente.
Project	Nome do projeto.
Package	Pacote das classes do componente (Namespace no caso da versão .NET do AF)
CodApp	Código da aplicação
InsAllowed	Flag que indica se é possível inserir novos registros. Valores "Y" ou "N".
UpdAllowed	Flag que indica se é possível atualizar os registros existentes. Valores "Y" ou "N".
RmvAllowed	Flag que indica se é possível apagar os registros existentes. Valores "Y" ou "N".
Description	Descrição do componente

Tabela 2 – Propriedades da tag component

O nome do projeto definido no XML serve para criar os diretórios nos quais as classes serão geradas automaticamente. A Figura 22 mostra a estrutura dos arquivos de um projeto baseado em AF. Um componente AF é dividido em classes localizadas em diferentes diretórios. Cada diretório tem um nome que é composto pelo nome do projeto mais um sufixo. O diretório com o sufixo CmpBL (ExampleProjectCmpBL) contém as classes com a lógica de negócio (Cmp, HomeCmp, Cmp_Coll, etc.). O diretório cujo nome termina com CmpBLitf contém as interfaces implementadas pelas classes do diretório CmpBL. O diretório CmpDALcmgr contém os DAOs que acessam a cache. O diretório terminado em CmpDALdb possui os DAOSs que acessam o banco de dados. O diretório com o sufixo CmpDALitf contém as interfaces implementadas por todos os DAOs. O

diretório CmpDALProxy contém os DAOs que acessam o Webservice wsRemoteDal. Por fim, o diretório scripts contém os scripts DDL para a criação do banco de dados.

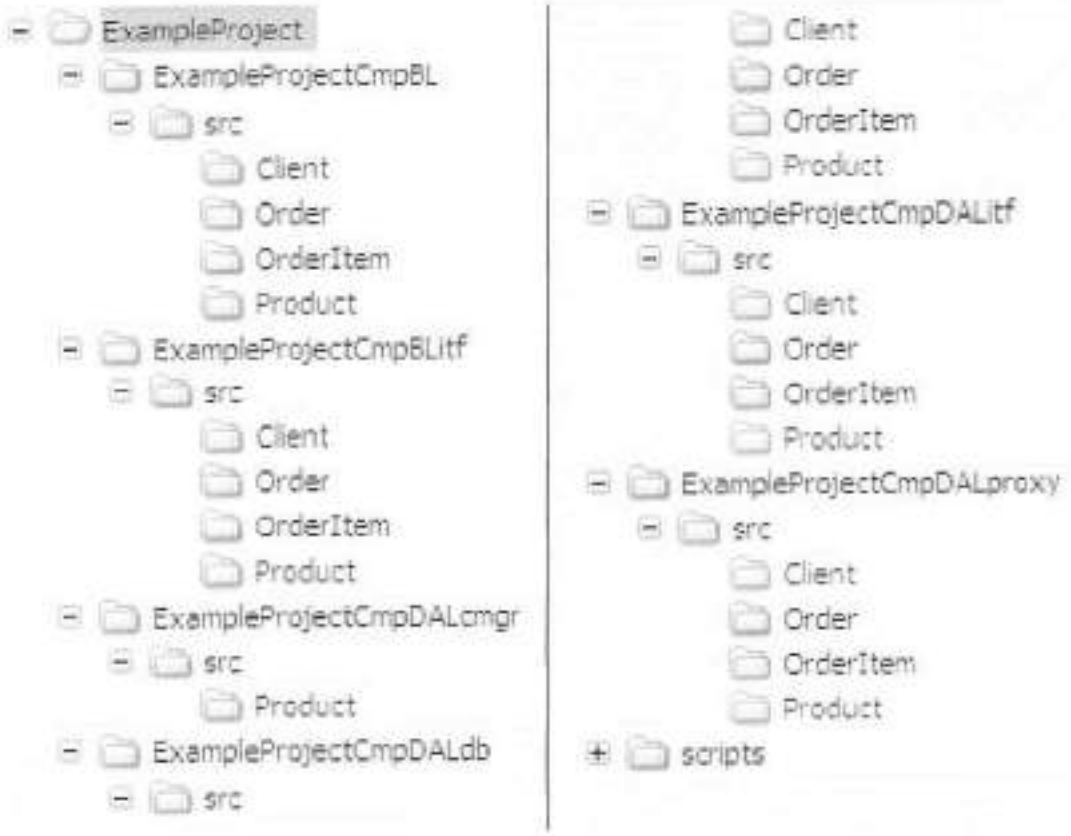

Figura 22 – Estrutura de arquivos de um projeto baseado no AF

O nome do componente é usado para criar o nome das classes (adicionando um prefixo ou sufixo). A propriedade Description se transforma na documentação das classes. O Fragmento de Código 24 mostra o esqueleto da classe Cmp_Client gerada automaticamente e como algumas propriedades da tag component são traduzidas no código produzido.

```
namespace exampleProject.bl
{
  /// <summary>
  /// Componente Client.
  /// </summary>
  public class Cmp_Client : BaseCmp_Client
  {
    //Class' body
  }
}
```

Fragmento de Código 24 – Esqueleto da classe Cmp_Client

Dentro do corpo da tag component se encontram as definições de todas as propriedades do componente. O Fragmento de Código 25 mostra as definições de todas as propriedades do componente Client. A Tabela 3 explica cada propriedade da tag property.

```
<properties>
  <property Id="0" Name="ClientID" Type="Guid" Fmt="String"
    RifLen="36" Public="Y" ReadOnly="N" ChgAllowed="N" Opt="N"
    DefaultValue=".GUID" MapOnField="ClientID"
    Description="Client's identifier"/>
  <property Id="1" Name="Name" Type="String" Fmt="String"
    RifLen="30" Public="Y" ReadOnly="N" ChgAllowed="Y" Opt="N"
    DefaultValue="" MapOnField="Name"
    Description="Client's full name"/>
  <property Id="2" Name="City" Type="String" Fmt="String"
    RifLen="30" Public="Y" ReadOnly="N" ChgAllowed="Y" Opt="Y"
    DefaultValue="" MapOnField="City"
    Description="Client's city"/>
  <property Id="3" Name="ClientType" Type="byte" Fmt="Number"
    RifLen="2" Public="Y" ReadOnly="N" ChgAllowed="Y" Opt="N"
    DefaultValue="" MapOnField="ClientType"
    Description="Client's city"/>
</properties>
```

Fragmento de Código 25 – Definição das propriedades do componente Cliente

Propriedade	Descrição
Id	Identificados unívoco da propriedade.
Name	Nome da propriedade.
Type	Tipo da propriedade.
Fmt	Formato da propriedade. Útil para que a aplicação cliente possa formatar o dado ao exibi-lo.
RifLen	Comprimento máximo do valor da propriedade. Usado para a validação dos dados.
Public	Flag que indica se a propriedade é exposta na interface do componente ("Y") ou apenas na implementação privada ("N").
ReadOnly	Flag que indica se a propriedade é somente de leitura ("Y") ou não ("N").
ChgAllowed	Flag que indica se é possível alterar o valor da propriedade. "Y" indica que é sempre permitido alterar seu valor. "N" significa que é possível alterá-lo somente até que o método SaveCmp seja executado pela primeira vez. Ou seja, quando a instância é criada. "X" quer dizer que é possível alterá-lo de acordo com uma condição manual.
Opt	Flag que indica se a propriedade é opcional ("Y") ou obrigatória ("N").
DefaultValue	Valor pré-definido da propriedade (atribuído quando o componente é criado). ".GUID" significa um novo GUID, ".NOW" indica a data e hora atuais, etc. Para uma descrição de todos os valores pré-definidos veja (NAVONE, 2006).

MapOnField	Nome da coluna na qual a propriedade é mapeada.
Description	Descrição da propriedade.

Tabela 3 – Propriedades da tag property

O Fragmento de Código 26 mostra o código gerado para a propriedade Name. Este fragmento de código foi obtido da classe BaseCmp_Client. A descrição da propriedade se torna parte da documentação da propriedade. Cada propriedade C# de um componente AF tem alguns atributos C# (NAGEL ET AL., 2005) (como CmpPropertyRequired) que pode ser usada pela aplicação cliente. Dentro do setter, o componente verifica se o novo valor é diferente do antigo. Em seguida, verifica se é permitido modificar aquela propriedade (método IsChangeAllowed). Se essas condições forem verdadeiras, o componente chama o método altera o valor da propriedade e chama o método SetRowChanged que indica que a instância foi modificada.

```
/// <summary>
/// Proprietà 'Name': Client's full name.
/// </summary>
[CmpPropertyFormat(CmpDataFormat.String)]
[CmpPropertyEditMask("")]
[CmpPropertyRifLen(30)]
[CmpPropertyRequired]
public virtual string Name
{
  get
  {
    return (fName);
  }
  set
  {
    if((value != null  &&  fName == null)  ||
       (value == null  &&  fName != null)  ||
       ((value != null  &&  fName != null)  &&
       (!value.Equals(fName))))
    {
      if (!IsChangeAllowed("Name"))
        throw new CmpPropertyReadOnlyException();
      fName = value;
      SetRowChanged();
    }
  }
}
```

Fragmento de Código 26 – Implementação da propriedade name do componente Client

Algumas propriedades podem ser definidas também como chaves de modo a realizar buscas. É obrigatório definir todas as chaves primárias do componente como chaves de busca. O Fragmento de Código 27 mostra a definição de todas as chaves de busca do componente Client. A Tabela 4 explica cada propriedade da tag key.

```
<keys>
  <key Id="0" Name="ClientID" Type="Guid" Fmt="String"
    RifLen="36" PrimaryKey="Y" MapOnField="ClientID"
    Condition="EQ" FindModes="0,1" ApplyFilters="N"
    Description="Client's identifier"/>
  <key Id="1" Name="Name" Type="String" Fmt="String"
    RifLen="30" PrimaryKey="N" MapOnField="Name"
    Condition="EQ" FindModes="0,1" ApplyFilters="N"
    Description="Client's full name"/>
</keys>
```

Fragmento de Código 27 – Definição das chaves de busca do componente Client

Propriedade	Descrição
Id	Identificador unívoco da chave de busca.
Name	Nome da chave de busca.
Type	Tipo da chave de busca.
Fmt	Formato da chave de busca. Útil para que a aplicação cliente possa formatar o dado ao exibi-lo.
RifLen	Comprimento máximo do valor da chave de busca.
PrimaryKey	Flag que indica se a chave é primária.
MapOnField	Nome da coluna na qual a chave é mapeada.
Condition	Condição da busca para usar na cláusula where. Pode ser "EQ" (igual), "LIKE" (condição like de sql), "GT" (maior que), "GE" (maior ou igual), "LT" (menor que), "LE" (menor ou igual).
FindModes	Define os modos de busca que podem ser usados com a chave.
ApplyFilters	Estabelece se podem ser aplicados filtros na chave. Este projeto não discute o uso de filtros.
Description	Descrição da chave de busca.

Tabela 4 – Propriedades da tag key

Para cada chave de busca definida no arquivo XML do componente, uma propriedade na classe Home é criada. O Fragmento de Código 28 mostra a propriedade Name da classe HomeCmp_Client.

```
/// <summary>
/// Setter per chiave 'Name': Client's full name
/// </summary>
[CmpKeyFormat(CmpDataFormat.String)]
[CmpKeyEditMask("")]
[CmpKeyRifLen(30)]
public string Name
```

```
{
  set
  {
    kName = value;
  }
}
```

Fragmento de Código 28 – Implementação da propriedade Name (chave de busca) da classe HomeCmp_Client

É necessário definir ao menos um modo de busca (find mode) para cada chave do componente. O modo de busca é a maneira pela qual a busca será executada. Ele define a ordenação do resultado (alfabeticamente, etc) e a vista (ou tabela) que será usada. Uma vista diferente pode ser usada para realizar um join entre tabelas e especificar outras condições de busca. Neste projeto, a vista é sempre a tabela na qual o componente é mapeado. O Fragmento de Código 29 mostra todos os modos de busca definidos no componente Client. O modo de busca std é obrigatório é indica o modo de busca padrão. Normalmente, ele ordena o resultado pela chave primária (ClientID no exemplo). A Tabela 5 explica cada propriedade da tag findmode.

```
<findmodes>
  <findmode Id="0" Name="STD" MapOnView="Clients"
    OrderBy="ClientID" FullView="Y" Description="Standard
ordering"/>
  <findmode Id="1" Name="OrderedByName" MapOnView="Clients"
    OrderBy="Name" FullView="Y" Description="Ordering by
name"/>
</findmodes>
```

Fragmento de Código 29 – Definição dos modos de busca do componente Client

Propriedade	Descrição
Id	Identificador unívoco do modo de busca.
Name	Nome do modo de busca.
MapOnView	Vista (ou tabela) do banco de dados usada na busca.
OrderBy	Propriedades (separadas por vírgula) usadas para ordenar o resultado.
FullView	Reservado para uso futuro.
Description	Descrição do modo de busca.

Tabela 5 – Propriedades da tag FindMode

Para cada modo de busca definido do arquivo XML do componente, é criada uma constante na classe DefsHomeCmp (DefsHomeCmp_Client no caso do

componente Client). O Fragmento de Código 30 mostra a classe DefsHomeCmp_Client.

```
/// <summary>
/// Definizioni relative all'interfaccia home del componente
Client.
/// Generata automaticamente da BLC Builder.
/// </summary>
public class DefsHomeCmp_Client
{
  /// <summary>
  /// Identificativo modalità di ricerca 'STD': Standard
ordering.
  /// </summary>
  public const int idFM_STD = 0;

  /// <summary>
  /// Identificativo modalità di ricerca 'OrderedByName':
Ordering by
      name.
  /// </summary>
  public const int idFM_OrderedByName = 1;
}
```

Fragmento de Código 30 – Classe DefsHomeCmp_Client

Para cada modo de busca definido no XML do componente, é criada uma clausula order diferente no método GetFindOrderByClause da classe daldb.DalCmp. Este método é usado pela classe daldb.DalCmp para construir a query select. Trata-se de um método protected. Ou seja, é possível sobrescreve-lo de modo a criar manualmente uma clausula order by. O Fragmento de Código 31 mostra esse método da classe daldb.BaseDalCmp_Client.

```
/// <summary>
/// Restituisce la clausola di ordinamento prevista per la
modalità
    di ricerca richiesta.
/// </summary>
/// <param name="IdFindMode"></param>
/// <returns></returns>
protected string GetFindOrderByClause(int IdFindMode)
{
  switch (IdFindMode)
  {
    case DefsDalCmp_Client.idFM_STD:
      return(" order by ClientID");
    case DefsDalCmp_Client.idFM_OrderedByName:
      return(" order by Name");
```

```
    default:
      return ("");
  }
}
```

Fragmento de Código 31 – Método GetFindOrderByClause da classe daldb.BaseDalCmp_Client

Assim como para a classe HomeCmp, existe uma classe de constantes para a classe Cmp. A classe DefsHomeCmp contém todos (e somente) os modos de busca definidos no XML do componente. A classe DefsCmp contém todas as constantes definidas no XML. As constantes são usadas para descrever valores importantes que são usados pelo componente. Na aplicação de exemplo, a classe Client tem a propriedade ClientType (tipo de cliente) que pode ter dois valores: 0 para pessoas jurídicas e 1 para pessoas físicas. Então, duas constantes são definidas no XML do componente Client. Elas são mostradas no Fragmento de Código 32.

```
<constants>
  <cnsgroup Id="0" Name="ClientType" Description="Type of
client">
    <constant Id="0" Name="JuristicPerson" Type="byte"
Value="0"
      Description="Juristic Person (Companies)"/>
    <constant Id="1" Name="NaturalPerson"  Type="byte"
Value="1"
      Description="Natural Person (People)"/>
  </cnsgroup>
</constants>
```

Fragmento de Código 32 – Definição das constantes do componente Client

As constantes são separadas em grupos. Cada constante tem um tipo e um valor. Mesmo dentro de um único grupo, duas constantes podem ter tipos e valores diferentes. A classe DefsCmp contém todas as constantes definidas no XML do componente. O nome das constantes na classe gerada é ConstantGroup_ConstantName (ClientType_JuristicPerson por exemplo). O Fragmento de Código 33 mostra a classe DefsCmp_Client.

```
/// <summary>
/// Definizioni relative al componente Client.
/// Generata automaticamente da BLC Builder.
/// </summary>
public class DefsCmp_Client
{
  /// <summary>
  /// Type of client: Juristic Person (Companies).
```

```
    /// </summary>
    public const byte ClientType_JuristicPerson = 0;

    /// <summary>
    /// Type of client: Natural Person (People).
    /// </summary>
    public const byte ClientType_NaturalPerson = 1;
}
```

Fragmento de Código 33 – Classe DefsCmp_Client

No componente de dados é possível definir propriedades que não são mapeadas. Isto é, propriedades que não são persistidas em uma coluna do banco de dados. Normalmente, propriedades não mapeadas possuem acesso somente de leitura e são usadas para implementar propriedades de lookup. Propriedades de lookup são úteis para replicar propriedades de outros componentes sem duplicar colunas em tabelas diversas. Por exemplo, esse tipo de propriedade é útil para se ter o nome e o preço do produto no componente OrderItem (pois só existem no componente Product). Por esta razão foram criadas duas propriedades não mapeadas (ProductName e ProductPrice) nesse componente.O Fragmento de Código 34 mostra a definição de todas as propriedades do componente OrderItem.

```
properties>
  <property Id="0" Name="ProductID" Type="Guid" Fmt="String"
    RifLen="36" Public="Y" ReadOnly="N" ChgAllowed="N" Opt="N"
    DefaultValue="" MapOnField="ProductID"
    Description="Product's identifier" RelatedFields="3,4"/>
  <property Id="1" Name="OrderID" Type="Guid" Fmt="String"
    RifLen="36" Public="Y" ReadOnly="N" ChgAllowed="Y" Opt="N"
    DefaultValue="" MapOnField="OrderID"
    Description="Order's identifier"/>
  <property Id="2" Name="Quantity" Type="short" Fmt="Number"
    RifLen="4" Public="Y" ReadOnly="N" ChgAllowed="Y" Opt="N"
    DefaultValue="" MapOnField="Price"
    Description="Quantity of items"/>
  <property Id="3" Name="ProductName" Type="String"
    Fmt="String" RifLen="30" Public="Y" ReadOnly="Y"
    ChgAllowed="N" Opt="Y" DefaultValue="" MapOnField="NOMAP"
    Description="Product's name" ControlFields="1"/>
  <property Id="4" Name="ProductPrice" Type="decimal"
    Fmt="Number" RifLen="10" Public="Y" ReadOnly="Y"
    ChgAllowed="N" Opt="Y" DefaultValue="" MapOnField="NOMAP"
    Description="Product's price" ControlFields="1"/>
</properties>
```

Fragmento de Código 34 – Definição das propriedades do componente OrderItem

Para definir uma propriedade não mapeada é preciso atribuir o valor "NOMAP" à propriedade MapOnField da tag property. No caso de propriedades que dependem de outras propriedades, é possível valorizar a propriedade ControlFields na propriedade dependente e RelatedFields nas propriedades que controlam a dependente. No exemplo anterior, as propriedades dependentes são ProductName e ProductPrice e a propriedade de controle é ProductID. Deste modo, o componente zera automaticamente as propriedades dependentes quando o valor da que controla muda. Para cada propriedade que possui outras de controle, é criado um método que serve para inserir o código manual que valoriza a propriedade. O nome desse método possui um prefixo GetLookUp seguido do nome da propriedade (GetLookUpProductName por exemplo). O Fragmento de Código 35 mostra a propriedade ProductName da classe BaseCmp_OrderItem. Quando fProductName (o atributo que guarda o valor de ProductName) é nulo, o método GetLookUpPropertyName é chamado. O Fragmento de Código 36 mostra a propriedade ProductID da classe BaseCmp_OrderItem. Quando seu valor é alterado, todos os campos relacionados são zerados (fProductName = null, por exemplo).

```
/// <summary>
/// Proprietà 'ProductName': Product's name.
/// </summary>
[CmpPropertyFormat(CmpDataFormat.String)]
[CmpPropertyEditMask("")]
[CmpPropertyRifLen(30)]
public virtual string ProductName
{
  get
  {
    if (fProductName == null  ||  "".Equals(fProductName))
      if (fOrderID != null)
      {
        try
        {
          GetLookUpProductName();
        }
        catch (CmpException e)
        {
          if (!(e is CmpNotFoundException))
            ExcpLogger.LogException(e);
        }
      }
```

```
    return (fProductName);
  }
}
```

Fragmento de Código 35 – Propriedade ProductName da classe BaseCmp_OrderItem

```
/// <summary>
/// Proprietà 'ProductID': Product's identifier.
/// </summary>
[CmpPropertyFormat(CmpDataFormat.String)]
[CmpPropertyEditMask("")]
[CmpPropertyRifLen(36)]
[CmpPropertyRequired]
public virtual Guid? ProductID
{
  get
  {
    return (fProductID);
  }
  set
  {
    if ((value != null  &&  fProductID == null)  ||
        (value == null  &&  fProductID != null)  ||
       ((value != null  &&  fProductID != null)  &&
       (!value.Value.ToString("D").Equals(

fProductID.Value.ToString("D"))))))
    {
      if (!IsChangeAllowed("ProductID"))
        throw new CmpPropertyReadOnlyException();

      fProductName = null;
      fProductPrice = null;

      fProductID = value;

      SetRowChanged();
    }
  }
}
```

Fragmento de Código 36 – Propriedade ProductID da classe BaseCmp_OrderItem

O Fragmento de Código 37 mostra os métodos de loookup da classe Cmp_OrderItem. Somente o esqueleto desses métodos é gerado automaticamente, pois eles se encontram na classe derivada. O método GetLookUpProductName recupera o objeto Product associado ao componente OrderItem de modo a descobrir o nome do produto. Para economizar processamento, este método valoriza as

propriedades ProductName e ProductPrice. O método GetLookUpProductPrice simplesmente chama o método GetLookUpProductName, posto que a operação é a mesma.

```
/// <summary>
/// Funzione che effettua il look up per la proprietà
correlata 'Product's name'.
/// </summary>
protected override void GetLookUpProductName()
{
  IHomeCmp_Product homeProduct =
    (IHomeCmp_Product)GetObject("HomeCmp_Product");

  homeProduct.ProductID = ProductID;
  ICmp_Product product = homeProduct.GetCmp();
  SetProductName(product.Name);
  SetProductPrice(product.Price);
}

/// <summary>
/// Funzione che effettua il look up per la proprietà
correlata 'Product's price'.
/// </summary>
protected override void GetLookUpProductPrice()
{
  GetLookUpProductName();
}
```

Fragmento de Código 37 – Métodos de lookup do componente OrderItem

Um componente de dados pode possuir alguns métodos de negócio. Esses métodos devem trabalhar somente com os dados do componente ao qual pertencem (e aos dados dos componentes filhos). O Fragmento de Código 38 mostra o arquivo de definição do componente OrdemItem. O Fragmento de Código 39 mostra a implementação do método CalculateItemPrice do componente OrdemItem. Este método foi retidado da classe Cmp_OrderItem. Portanto, seu esqueleto foi gerado automaticamente. Os métodos dos componentes de dados podem receber parâmetros. Nenhum dos componentes da aplicação de exemplos possui métodos que recebam parâmetros. Para mais informações sobre os arquivos de definição XML veja (NAVONE, 2006).

```
<methods>
  <method Id="0" Name="CalculateItemPrice" Type="decimal"
    Fmt="Number" RifLen="10" Public="Y" ThrowsException="Y"
    Description="Calculates the order item's total price"
```

```
    RcDescription="Order item's total price">
  </method>
</methods>
```

Fragmento de Código 38 – Definição dos métodos do componente OrdemItem

```
/// <summary>
/// Calculates the order item's total price
/// </summary>
/// <returns>Order item's total price</returns>
/// <exception cref="CmpMethodException"></exception>
public override decimal CalculateItemPrice()
{
  if (ProductPrice != null && Quantity != null)
    return ProductPrice.Value * Quantity.Value;
  else
    return 0;
}
```

Fragmento de Código 39 – Método CalculateItemPrice do componente OrderItem

5.3.2 Arquivos XML de Definição dos Componentes de Serviços e o Código Gerado

Esta seção mostra como escrever os arquivos XML de definição de um componente de serviço e como é o código gerado automaticamente. O arquivo de definição de um componente de serviço é muito mais simples do que o de um componente de dados pois possui somente métodos e constantes (nada de propriedades ou chaves de busca). O Fragmento de Código 40 mostra o XML de um exemplo de componente de serviços. Esse componente é chamado ClientManager e é o responsável pela gestão dos clientes da aplicação de exemplo. Seu único método é o DeleteClient que apaga um cliente e todos os seus pedidos. É necessário criar um componente de serviço para tal função pois toda a operação precisa ser atômica (realizada dentro de uma mesma transação).

```
<component Id="1005" Name="ClientManager" Type="SERVICE"
DdlPrefix=""
  Project="ExampleProject" Package="exampleProject"
CodApp="1000"
  Description="Operations to manage clients" >
  <constants>
    <cnsgroup Id="0" Name="ErrorCode"
      Description="Error codes for the CmpMethodException">
```

```
      <constant Id="0" Name="ClientNotFound" Type="int"
        Value="0"
        Description="Client Not Found"/>

    </cnsgroup>
  </constants>

  <methods>
    <method Id="0" Name="DeleteClient" Type="void" Fmt="None"
      Public="Y" RifLen="-1" TaskRequired="N" TrxRequired="Y"
      ThrowsException="Y" ExcpDescription="An error has
      ocurred"
      Description="Delete a Client and all related Orders">
      <mtdparam Id="0" Name="ClientID" Type="Guid"
        Fmt="String" RifLen="36" PrmDir="I"
        Description="Client's unique identification" />
    </method>
  </methods>
</component>
```

Fragmento de Código 40 – Arquivo XML de definição do componente de serviços ClientManager

As tags component e constant são basicamente as mesmas do arquivo de definição de um componente de dados. A tag method tem diferenças importantes entre os dois tipos de componentes: no componente de serviço o método pode ser transacional. Para declarar um método como sendo transacional basta atribuir o valor "Y" à propriedade TrxRequired. Deste modo, todos os componentes de dados utilizados pelo método trabalham dentro da mesma transação. Para maiores informações sobre os arquivos de definição veja (NAVONE, 2006).

O método criado na classe BaseCmp gerencia a transaçao. O Fragmento de Código 41 mostra o método DeleteClient da classe BaseCmp_ClientManager. Ao início do método, uma transação é obtida do BLC Transaction Manager. Toda classe sendo executada dentro da mesma thread usará automaticamente a mesma transação. Então, o código manual escrito na classe derivada (dentro do método ImplDeleteClient) é executado. Se alguma execução for lançada, um rollback será automaticamente realizado. O código manual também pode determinar que a transação seja cancelada através de um RollBackOnly. Os métodos de um componente de serviços lançam somente CmpMethodException. Qualquer outra exceção é encapsulada na CmpMethodException.

```
/// <summary>
/// Delete a Client and all related Orders
/// </summary>
/// <param name="ClientID">Client's unique
identification</param>
/// <exception cref="CmpMethodException">An error has
    ocurred</exception>
public void DeleteClient(Guid ClientID)
{
  ITrx trx;

  try
  {
    trx = trxmgr.GetTransaction(new DefaultTrxDefinition());
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw new CmpFatalException(e);
  }

  try
  {
    ImplDeleteClient(trx, ClientID);

    if (trx.IsRollbackOnly)
      trxmgr.Rollback(trx);
    else
      trxmgr.Commit(trx);
  }
  catch (CmpMethodException me)
  {
    ExcpLogger.LogException(me);
    try { trxmgr.Rollback(trx); } catch (Exception) {};
    throw me;
  }
  catch (TrxException te)
  {
    ExcpLogger.LogException(te);
    throw new CmpFatalException(te);
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    try { trxmgr.Rollback(trx); } catch (Exception) {};
    throw new CmpFatalException(e);
  }
}
```

Fragmento de Código 41 – Método DeleteClient da classe BaseCmp_ClientManager

O Fragmento de Código 42 mostra o método ImplDeleteClient da classe Cmp_ClientManager. Trata-se de um método abstrato da classe base e sobrescrito de modo a conter o código manual. Somente seu esqueleto é gerado automaticamente na classe derivada. Neste exemplo, o método apaga um determinado cliente e seus pedidos. Se durante alguma operação ocorre um erro, a transação será automaticamente desfeita.

```
/// <summary>
/// Delete a Client and all related Orders
/// </summary>
/// <param name="trx">transazione nell'ambito della quale si
svolge
    la funzione</param>
/// <param name="ClientID">Client's unique
identification</param>
/// <exception cref="CmpMethodException">An error has occurred
    </exception>
public override void ImplDeleteClient(ITrx trx, Guid ClientID)
{
  IHomeCmp_Client homeClient =
    (IHomeCmp_Client)GetObject("HomeCmp_Client");
  homeClient.ClientID = ClientID;
  try{
    homeClient.RemoveCmp();
  }
  catch (CmpNotFoundException)
  {
    throw new CmpMethodException(
      DefsCmp_ClientManager.ErrorCode_ClientNotFound);
  }

  IHomeCmp_Order homeOrder =
    (IHomeCmp_Order)GetObject("HomeCmp_Order");
  homeOrder.ClientID = ClientID;
  ICmpColl_Order orders =
    homeOrder.FindCmp(DefsHomeCmp_Order.idFM_STD);

  foreach(ICmp_Order order in orders)
  {
    homeOrder.Clear();
    homeOrder.OrderID = order.OrderID;
    homeOrder.RemoveCmp();
  }
}
```

Fragmento de Código 42 - Método ImplDeleteClient da classe Cmp_ClientManager

5.3.3 Diferenças entre as implementações dos DAOs

Como explicado na seção, o AF oferece templates para gerar diferentes tipos de objetos de acesso aos dados (DAOs): um que acessa diretamente o banco de dados (classe daldb.DalCmp), um que acessa um Webservice (classe proxydal.ProxyDalCmp) e um que acessa a cache no sistema de arquivos da aplicação cliente (classe dalcmgr.DalCmp). Embora essas classes sejam completamente diferentes, todas implementam a interface IDalCmp.

A classe DalCmp é usada pela classe Cmp para realizar operações de salvamento de dados e pela classe HomeCmp para executar operações de busca e apagamento. O Fragmento de Código 43 mostra o método FindCmp da classe HomeCmp_Product. Este método usa o método GetDal para pegar o tipo correto de DAO, configurado na object factory do BLC. Então, ele chama o método LoadCmpBuffer do DAO que retorna um Dataset que contém o resultado da busca. Por fim, o método transforma o Dataset em um instancia ICmpColl_Product.

```
/// <summary>
/// Restituisce l'elenco dei componenti che soddisfano i
criteri di ricerca impostati
/// </summary>
/// <param name="IdFindMode"></param>
/// <returns></returns>
public ICmpColl_Product FindCmp(int IdFindMode)
{
  IDalCmp_Product dal = GetDAL();
  DataSet ds = dal.LoadCmpBuffer(IdFindMode, kProductID,
kName);

  try
  {
    return ((ICmpColl_Product)objfct.GetObject("CmpColl_" +
rifname,
      new object[] {ds}));
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw new CmpFatalException(e);
  }
}
```

Fragmento de Código 43 – Método FindCmp da classe HomeCmp_Product

Cada classe DAL implementa o método LoadCmpBuffer de um modo diverso. O Fragmento de Código 44 mostra a implementação do método da classe daldb.BaseDalCmp_Product. Nesse caso, a conexão com o banco de dados (conn) e o comando sql (cmd) são utilizados para restituir os dados. O Fragmento de Código 45 mostra a implementação do mesmo método, mas da classe proxydal.ProxyDalCmp_Product. Esta versão do DAO utiliza o objeto wsRemoteDal (que extende a classe SoapHttpClientProtocol de .NET) que acessa o Webservice de mesmo nome. O Fragmento de Código 46 mostra a implementação do método LoadCmpBufferCommon da classe dalcmgr.BaseDalCmp_Product. Essa implementação usa a classe CacheManager para adquirir o Dataset com os registros do componente Product. Então, um filtro é aplicado e somente os registros requisitados são retornados.

```
private void LoadCmpBufferCommon(DataSet ds, int IdFindMode,
  Guid? ProductID, string Name, bool bIncludeChildren,
  DateTime? LastChangeFilter)
{
  IDbConnection conn = cnutils.GetConnection(dsrc);
  try
  {
    IDbCommand cmd = conn.CreateCommand();

    cmd.CommandType = CommandType.Text;
    cmd.CommandText = GetSelectFindQuery(IdFindMode) + " " +
      GetFindWhereClause(IdFindMode, ProductID, Name,
      LastChangeFilter) + " " +
GetFindOrderByClause(IdFindMode);

    IDataReader rdr = cmd.ExecuteReader();
    DataTable tbl = ds.Tables["Product"];

    while (rdr.Read())
    {
      LoadDataRow(tbl, rdr);
    }

    rdr.Close();
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw new CmpFatalException(e);
  }
  finally
  {
```

```
    cnutils.CloseConnection(conn, dsrc);
  }
}
```

Fragmento de Código 44 – Método LoadCmpBufferCommon da classe daldb.BaseDalCmp_Product

```
public DataSet LoadCmpBuffer(int IdFindMode, Hashtable Keys)
{
  try
  {
    WebServiceSecurityToken tkn = new
WebServiceSecurityToken();
    tkn.UserContextId = uctx.UserContextId;
    wsRemoteDal dal = new wsRemoteDal();
    dal.Url = remoteSvcAddr + remoteSvcName;
    dal.WebServiceSecurityTokenValue = tkn;

    string[] k = new string[Keys.Count];
    object[] kv = new object[k.Length];

    IDictionaryEnumerator de = Keys.GetEnumerator();

    for (int i=0; i<k.Length; i++)
    {
      de.MoveNext();
      k[i] = (string)de.Key;
      kv[i] = de.Value;
    }

    return (dal.LoadCmpBuffer(rifname, IdFindMode, k, kv));
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw (ProxyUtils.RemapException(e));
  }
}
```

Fragmento de Código 45 – Método LoadCmpBufferCommon da classe proxydal.ProxyDalCmp_Product

```
private void LoadCmpBufferCommon(DataSet ds, int IdFindMode,
  Guid? kProductID, string kName)
{
  try
  {
    DataSet dscache = CacheManager.GetCachedData("Product");
    DataTable tblcache = dscache.Tables["Product"];

    bool bSearchPK = true;
```

```
    if (kProductID == null)
      bSearchPK = false;
    else
    {
      if ((kName != null  &&  !"".Equals(kName)))
        bSearchPK = false;
    }

    DataTable tbl = ds.Tables["Product"];

    if (bSearchPK)
    {
      object[] pks = new object[1];
      pks[0] = kProductID;

      DataRow rwx = tblcache.Rows.Find(pks);

      if (rwx != null)
        tbl.Rows.Add(rwx.ItemArray);
    }
    else
    {
      string filterExpression = GetFindWhereClause(IdFindMode,
        kProductID, kName);
      string sortExpression =
GetFindOrderByClause(IdFindMode);

      DataRow[] rows = tblcache.Select(filterExpression,
        sortExpression);

      foreach (DataRow rwx in rows)
        tbl.Rows.Add(rwx.ItemArray);
    }
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw new CmpFatalException(e);
  }
}
```

Fragmento de Código 46 - Método LoadCmpBufferCommon da classe dalcmgr.BaseDalCmp_Product

No caso dos componentes AF de serviços, não existe o conceito de Dal. Portanto a versao do componente de serviços que roda no lado cliente é o próprio proxy. Toda a lógica de negócio reside no lado servidor. O Fragmento de Código 47 mostra o método DeleteClient da classe ProxySrvCmp_ClientManager. Comparando com o mesmo método da classe BaseCmp_ClientManager (Fragmento de Código

41), é possível notar que a versao do proxy somente chama o Webservice wsSrvCmpClientManager que redireciona esta chamada para a implementação real do método DeleteClient.

```
/// <summary>
/// Delete a Client and all related Orders
/// </summary>
/// <param name="ClientID">Client's unique
identification</param>
public void DeleteClient(Guid ClientID)
{
  try
  {
    refSrvCmpClientManager.WebServiceSecurityToken tkn = new
      refSrvCmpClientManager.WebServiceSecurityToken();
    tkn.UserContextId = uctx.UserContextId;
    wsSrvCmpClientManager cmp = new wsSrvCmpClientManager();
    cmp.Url = remoteSvcAddr + remoteSvcName;
    cmp.WebServiceSecurityTokenValue = tkn;
    cmp.DeleteClient(ClientID);
  }
  catch (Exception e)
  {
    ExcpLogger.LogException(e);
    throw (ProxyUtils.RemapException(e));
  }
}
```

Fragmento de Código 47 – Método DeleteClient da classe ProxySrvCmp_ClientManager

5.3.4 Configuraçao dos Componentes e Objetos na Object Factory

A última seção discutiu as diferenças entre as versões dos DAOs. Esta seção explica como configurar os componentes AF e qualquer objeto na Object Factory do BLC.

Para que um objeto seja instanciado corretamente pela Factory, todas as classes que compõem a hierarquia desse objeto precisam estar configuradas apropriadamente. A configuração dos objetos gerenciados pelo container é realizada em um arquivo de configuração que possui propriedades definidas como um par chave-valor. Na versao .NET do AF, este arquivo é um .NET Application Configuration File (DUFFY, 2006) que tem um formato XML com tags de tipo chave-valor. As chaves possuem a forma Prefixo.NomeReferência.Sufixo, onde:

- **Prefixo** é o tipo de chave

- **NomeReferência** é o nome de referência da chave
- **Sufixo** contém outros dados cuja presença depende do prefixo.

A Tabela 6 mostra todos os prefixos possíveis para uma chave em um arquivo de configuração da Object Factory do BLC.

Nome do Prefixo	Tipo do Objeto	Descrição
prm	Parâmetro de configuração	Nome do parâmetro de configuração. Os parâmetros são valores simples como inteiros, strings, etc.
svc	Serviço BLC	Nome simbólico do serviço BLC.
obj	Objeto Aplicativo	Nome simbólico do objeto aplicativo.

Tabela 6 – Tipos de prefico da chave do arquivo de configuração

Os parâmetros de configuração não possuem nenhum sufixo. A Tabela 7 mostra os tipos de sufico para os serviços BLC e a Tabela 8 mostra os para objetos aplicativos.

Sufixo	Descrição
class	Classe que implementa o serviço.
startOrder	Ordem de início do serviço.
factory	Flag que indica se é um serviço simples (1) ou uma factory (0).
refprm	Nome de referência do parâmetro a ser injetado.
refsvc	Nome de referência do serviço a ser injetado.
refobj	Nome de referência do objeto a ser injetado.

Tabela 7 – Tipos de sufixo permitidos para serviços BLC

Sufixo	Descrição
class	Classe quem implementa o objeto aplicativo.
refprm	Nome de referência do parâmetro a ser injetado.
refsvc	Nome de referência do serviço a ser injetado.
refobj	Nome de referência do objeto a ser injetado.
refrtobj	Nome de referência do objeto em tempo de execução a ser injetado.

Tabela 8 – Tipos de sufico permitidos para objetos aplicativos

As próximas seções discutem a configuração dos principais tipos de objetos na Object Factory do BLC.

5.3.4.1 Configuração dos Serviços BLC

Esta seção explica como configurar os serviços mais importantes do BLC. O Fragmento de Código 48 mostra a configuração do logger BLC. O parâmetro LogDir indica o diretório no qual o arquivo de log será criado. O Fragmento de Código 49

mostra a configuração do gerenciador de transações BLC. O Fragmento de Código 50 mostra a configuração da conexão ao banco de dados do projeto de exemplo. Nesse caso, trata-se de uma conexão para banco de dados Microsoft SQL Server (NHIBERNATE). O parâmetro ConnString indica a string de conexão ao banco de dados.

```
<add key="svc.ExcpLogSvc.class"
  value="it.mainline.blc.svcbase.impl.FileExcpLogSvc@
  BlcServices.dll"/>
<add key="svc.ExcpLogSvc.startOrder" value="0"/>
<add key="svc.ExcpLogSvc.factory" value="0"/>
<add key="svc.ExcpLogSvc.refprm.LogDir" value="E:\logs"/>
```

Fragmento de Código 48 – Configuração do logger BLC

```
<add key="svc.TrxMgrSvc.class"

value="it.mainline.blc.trx.impl.StdTrxMgrSvc@BlcServices.dll"/
>
<add key="svc.TrxMgrSvc.startOrder" value="1"/>
<add key="svc.TrxMgrSvc.factory" value="0"/>
<add key="svc.TrxMgrSvc.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
```

Fragmento de Código 49 – Configuração do gestor de transações BLC

```
<add key="svc.ConnSvcExampleProject.class"

value="it.mainline.blc.trx.impl.SqlConnSvc@$BlcServices.dll"/>
<add key="svc.ConnSvcExampleProject.startOrder" value="2"/>
<add key="svc.ConnSvcExampleProject.factory" value="1"/>
<add key="svc.ConnSvcExampleProject.refprm.ConnString"
  value="server=127.0.0.1;uid=blc_testuser;pwd=blc_testuser;
  database=ExampleProject;"/>
```

Fragmento de Código 50 – Configuração de uma conexão (ao banco de dados) BLC

5.3.4.2 Configuração de um componente de dados

O Fragmento de Código 51mostra a configuração da classe HomeCmp_Order. O Fragmento de Código 52 mostra a configuração da classe Cmp_Order. O Fragmento de Código 53 mostra a configuração da classe CmpColl_Order. O Fragmento de Código 54 mostra a configuração da classe ChildCmpColl_Order_Items.

```
<add key="obj.HomeCmp_Order.class"
  value="exampleProject.bl.HomeCmp_Order@
  ExampleProjectCmpBL.dll"/>
<add key="obj.HomeCmp_Order.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
```

```
<add key="obj.HomeCmp_Order.refsvc.SavePointManagerSvc"
  value="SavePointMgrSvc"/>
<add key="obj.HomeCmp_Order.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
```

Fragmento de Código 51 – Configuração da classe HomeCmp_Order

```
<add key="obj.Cmp_Order.class"

value="exampleProject.bl.Cmp_Order@ExampleProjectCmpBL.dll"/>
<add key="obj.Cmp_Order.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
<add key="obj.Cmp_Order.refsvc.SavePointManagerSvc"
  value="SavePointMgrSvc" filter="smclient" />
<add key="obj.Cmp_Order.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
<add key="obj.Cmp_Order.refrtobj.UserContext"
value="UserContext"/>
```

Fragmento de Código 52 – Configuração da classe Cmp_Order

```
<add key="obj.CmpColl_Order.class"

value="exampleProject.bl.CmpColl_Order@ExampleProjectCmpBL.dll
"/>
<add key="obj.CmpColl_Order.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
<add key="obj.CmpColl_Order.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
```

Fragmento de Código 53 – Configuração da classe CmpColl_Order

```
<add key="obj.ChildCmpColl_Order_Items.class"
  value="exampleProject.bl.ChildCmpColl_Order_Items@
  ExampleProjectCmpBL.dll" />
<add key="obj.ChildCmpColl_Order_Items.refsvc.ExcpLogSvc"
  value="ExcpLogSvc" />
<add key="obj.ChildCmpColl_Order_Items.refrtobj.ObjectFactory"
  value="ObjectFactory" />
```

Fragmento de Código 54 – Configuração da classe ChildCmpColl_Order

Os próximos exemplos mostram os possíveis modos de configuração das classes DAL de um componente de dados. O Fragmento de Código 55 mostra a configuração da classe daldb.DalCmp_Order. O parâmetro DataSource indica qual conexão BLC usar. Neste caso, ele usa a conexão ConnExampleProject, mostrada no Fragmento de Código 50. O Fragmento de Código 56 mostra a configuração da classe proxydal.ProxyDalCmp_Order. O parâmetro RemoteSvcAddr determina a

URL do Webservice wsRemoteDal. O Fragmento de Código 57 mostra a configuração da classe dalcmgr.DalCmp_Product.

```
<add key="obj.DalCmp_Order.class"
  value="exampleProject.daldb.DalCmp_Order@
  ExampleProjectCmpDALdb.dll"/>
<add key="obj.DalCmp_Order.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
<add key="obj.DalCmp_Order.refsvc.TrxMgrSvc"
value="TrxMgrSvc"/>
<add key="obj.DalCmp_Order.refsvc.DataSource"
  value="$ConnExampleProject"/>
<add key="obj.DalCmp_Order.refsvc.ConnUtils"
value="TrxMgrSvc"/>
<add key="obj.DalCmp_Order.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
```

Fragmento de Código 55 – Configuração da classe daldb.DalCmp_Order

```
<add key="obj.DalCmp_Order.class"
  value="exampleProject.proxydal.ProxyDalCmp_Order@
  ExampleProjectCmpDALproxy.dll"/>
<add key="obj.DalCmp_Order.refprm.RemoteSvcAddr"
  value="http://localhost/ExampleWebServices/"/>
<add key="obj.DalCmp_Order.refprm.RemoteSvcName"
  value="wsRemoteDal.asmx"/>
<add key="obj.DalCmp_Order.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
<add key="obj.DalCmp_Order.refrtobj.UserContext"
  value="UserContext"/>
```

Fragmento de Código 56 – Configuração da classe proxydal.ProxyDalCmp_Order

```
<add key="obj.DalCmp_Product.class"
  value="exampleProject.dalcmgr.DalCmp_Product
  @ExampleProjectCmpDALcmgr.dll"/>
<add key="obj.DalCmp_Product.refsvc.ExcpLogSvc"
value="ExcpLogSvc"/>
<add key="obj.DalCmp_Product.refsvc.CacheMgrSvc"
  value="CacheMgrSvc"/>
<add key="obj.DalCmp_Product.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
```

Fragmento de Código 57 – Configuração da classe dalcmgr.DalCmp_Product

5.3.4.3 Configuraçao de um Componente de Serviços

Os próximos exemplos mostram os possíveis modos de configuração de um componente de serviços. O Fragmento de Código 58 mostra a configuração da

classe Cmp_ClientManager. O Fragmento de Código 59 mostra a configuração da classe ProxySrvCmp_ClientManager. O parâmetro RemoteSvcAddr indica a URL do serviço remoto (o WebService gerado automaticamente para o componente de serviços). O parâmetro RemoteSvcName determina o nome do Webservice.

```
<add key="obj.Cmp_ClientManager.class"
  value="exampleProject.bl.Cmp_ClientManager@
  ExampleProjectCmpBL.dll"/>
<add key="obj.Cmp_ClientManager.refsvc.ExcpLogSvc"
  value="ExcpLogSvc"/>
<add key="obj.Cmp_ClientManager.refsvc.TrxMgrSvc"
  value="TrxMgrSvc"/>
<add key="obj.Cmp_ClientManager.refsvc.ConnUtils"
  value="TrxMgrSvc"/>
<add key="obj.Cmp_ClientManager.refrtobj.ObjectFactory"
  value="ObjectFactory"/>
<add key="obj.Cmp_ClientManager.refrtobj.UserContext"
  value="UserContext"/>
```

Fragmento de Código 58 – Configuração da classe Cmp_ClientManager

```
<add key="obj.Cmp_ClientManager.class"
  value="exampleProject.proxybl.ProxySrvCmp_ClientManager@
  ExampleProjectCmpSrvBLproxy.dll" filter="smclient"/>
<add key="obj.Cmp_ClientManager.refsvc.ExcpLogSvc"
  value="ExcpLogSvc"/>
<add key="obj.Cmp_ClientManager.refprm.RemoteSvcAddr"
  value="$RemoteSvcAddr"/>
<add key="obj.Cmp_ClientManager.refprm.RemoteSvcName"
  value="wsSrvCmpClientManager.asmx"/>
<add key="obj.Cmp_ClientManager.refrtobj.UserContext"
  value="UserContext"/>
```

Fragmento de Código 59 – Configuração da classe ProxySrvCmp_ClientManager

6 Testabilidade de Aplicações Corporativas

As atividades produtivas no desenvolvimento de sistemas de software são extremamente tendenciosas a erros humanos. Os erros podem ocorrer em cada fase do processo. Como conseqüência da incapacidade humana de produzir e comunicar com perfeição, o desenvolvimento de software tem de ser acompanhado por atividades que assegurem sua qualidade (PRESSMAN, 2001). O teste de software é crucial para garantir sua qualidade.

Uma vez que o código tenha sido produzido, o software precisa ser testado para descobrir (e corrigir) o máximo de erros possível antes de entregá-lo ao cliente. Depois da entrega, quando existem requisições de mudanças, é essencial repetir todos os testes na nova versão da aplicação. O teste é um processo de executar um programa com a intenção de encontrar erros. É impossível assegurar que um software não contenha erros, dado que teste exaustivo é impraticável, conforme explicado na seção 6.2.2. No entanto, os testes devem descobrir a maior quantidade de erros possível.

A testabilidade de software é quão facilmente um programa de computador pode ser testado. Enquanto se desenvolve um software, é muito importante fazer coisas que auxiliarão o processo de teste como alguns pontos de design e simples detalhes de programação como programar através de interfaces ao invés de classes.

6.1 Terminologia dos Testes

O teste de software é um elemento de uma área mais ampla normalmente chamada de verificação e validação (V&V). Verificação é um conjunto de atividades que asseguram que o software implemente uma função específica. Validação é um conjunto de atividades que asseguram que o software tenha sido construído de acordo com os requisitos do cliente (PRESSMAN, 2001). V&V é parte da garantia de qualidade de software que também cuida de processos como revisão de documentos, controle de código fonte, gerência de configuração e assim por diante. As técnicas de V&V podem ser estáticas (análise de código, por exemplo) e dinâmica (testes).

Um termo comum na engenharia de software é caso de teste, que é um conjunto de condições ou variáveis sob as quais um testador determinará se um requisito ou

caso de uso de uma aplicação é parcialmente ou totalmente satisfeito. Podem ser necessários vários casos de teste para assegurar que um requisito é totalmente satisfeito. Para testar completamente que todos os requisitos da aplicação são satisfeitos, é necessário criar ao menos um caso de teste para cada caso de uso (ou requisito) do sistema. Suíte de teste é um conjunto (ou uma seqüência) de casos de uso.

Teste não pode ser confundido com depuração, pois são atividades diferentes. A depuração ocorre como conseqüência de um teste mal sucedido. Isto é, quando um teste descobre um erro, a depuração é o processo que resulta na localização exata e remoção do erro. Portanto, a depuração deve ser acomodada em toda estratégia de teste.

6.2 Tipos de Testes de Software

Os testes de software podem ser utilizados para detectar defeitos ou para medir algumas características do sistema (como performance) em um ambiente que reproduz o uso real da aplicação. O primeiro tipo pode ser realizado usando basicamente duas técnicas: testes de caixa branca e caixa preta (PRESSMAN, 2001). As seções seguintes discutem cada um deles em detalhes.

6.2.1 Teste de Caixa Branca

O teste de caixa branca (ou teste de caixa de vidro) é um método de projeto de caso de teste que usa a estrutura de controle do código para derivar os testes. Em outras palavras, ele requer conhecimento da parte interna da classe sob teste. O teste de caixa branca é construído baseado em um exame dos detalhes procedurais. O termo "caixa branca" é mais usado do que o termo "caixa de vidro", mas sua concepção é equivocada, pois é possível ver dentro de uma caixa de vidro (o código no caso), porém uma caixa branca não é mais transparente do que uma caixa preta.

Esse tipo de procedimento de teste é baseado no conceito de cobertura, isto é, execução dos elementos do fluxograma do programa. Usando métodos de teste de caixa branca é possível obter casos de teste que asseguram que todos os caminhos independentes dentro de um módulo sob teste tenham sido executados ao menos uma vez, que cada comando tenha sido executado ao menos uma vez, que cada

decisão lógica nos lados verdadeiro e falso tenha sido executados ao menos uma vez ou (e) que todos os loops tenham sido executados ao menos uma vez.

6.2.2 Teste de Caixa Preta

O teste de caixa preta (ou comportamental) foca nos requisitos funcionais do software, isto é, permite ao desenvolvedor criar conjuntos de entradas que exercitem todos os requisitos funcionais do programa. Essa abordagem é conduzida na interface do software. O teste de caixa preta não é uma alternativa às técnicas de caixa branca, mas é uma abordagem complementar que tende a descobrir outros tipos de erros: funções incorretas ou inexistentes, erros na interface, erros nas estruturas de dados, problemas de desempenho e erros de iniciação e terminação.

Na prática, o teste exaustivo não é possível. A quantidade de permutações de entradas de até mesmo um programa simples é gigante. Por exemplo, um método muito simples que recebe em sua entrada dois números: x e y. O código simplesmente calcula a soma x+y. Neste básico exemplo existem 10^{20} combinações diferentes para os valores na entrada. Por essa razão, é impossível executar todas as combinações de valores de entrada. No entanto, é possível cobrir adequadamente todas as classes de equivalência de testes. Classes de equivalência são subconjuntos de valores de entrada (válidos e inválidos) com os quais o programa se comporta da mesma forma.

As classes de equivalência do método mostrado no Fragmento de Código 60 podem ser: x>0, x<0, x=0, e x não numérico.

```
public class ExampleClass
{
  public int abs(string x)
  {
    int aux = Convert.ToInt32(x);
    if(aux >= 0)
      return aux;
    else
      return -aux;
  }
}
```

Fragmento de Código 60 – Método usado para calcular o valor absoluto de um número

Por razões que não são completamente claras, um grande número de erros tende a ocorrer nas fronteiras do domínio de entrada. Vale a pena dedicar alguns

casos de teste para esse tipo de entrada. Isto é chamado condições de fronteira. No exemplo anterior, condições de fronteira poderiam ser valores de x próximos a zero (positivos e negativos).

6.3 Fases de Teste

Os casos de teste devem ser construídos durante diferentes fases do ciclo de vida do software e em granularidades diversas. As principais fases (e tipos de casos de teste) são:

- **Teste de Unidade (ou Componente)** – concentra-se em cada unidade (ou componente) do software como implementado no código fonte. Neste caso, cada caso de teste pode testar, por exemplo, uma única classe (a menor unidade da arquitetura do software).
- **Teste de Integração** – teste de grupos de elementos integrados de modo a criar sub-sistemas (ou sistemas).
- **Teste de Sistema** – teste de todo o sistema (incluindo hardware, rede, etc). Ele pode ser, por exemplo, teste de estresse, teste de robustez, teste de segurança, teste de recuperação de falhas, teste de desempenho, etc.
- **Teste de Aceitação** – também chamado de teste de validação, visa testar todo o sistema (como no anterior), mas é realizado pelo cliente ou na plataforma cliente.
- **Teste de Regressão** – re-execução dos testes depois de alterações na aplicação.

Este projeto foca no teste de unidade devido à sua importância durante a fase de programação do ciclo de vida do software. O teste de unidade visa testar cada classe em isolamento. Os testes devem ser relacionados de perto com o código, ou seja, testes de caixa branca. É muito importante escrever esse tipo de teste durante a codificação da aplicação, ou mesmo antes, como recomendado pelo Test Driven Development (TDD) (BECK, 2002). A seçao 6.5 discute o TDD em mais detalhes.

6.4 Automaçao dos Testes de Unidade

O teste de unidade é uma atividade essencial do processo de desenvolvimento de software. Geralmente, depois que um módulo ou uma nova funcionalidade é

construído, o desenvolvedor o testa. Esta ação pode ser feita manualmente. O desenvolvedor escolhe algumas entradas, executa o programa e observa sua saída. Com essa abordagem, é possível encontrar erros e corrigir o programa. Entretanto, depois de um tempo isso se torna muito repetitivo. Depois de implementar alguma nova funcionalidade, todo o resto da aplicação também deve ser testado, pois a nova funcionalidade pode indiretamente interferir com as outras partes.

Repetir sempre os mesmos casos de teste manualmente é enfadonho e tendencioso a erros. O testador pode facilmente esquecer de realizar algum teste. Portanto, é fundamental automatizar os casos de teste quando possível. Esta não é uma tarefa fácil e precisa de muita prática.

Para automatizar os testes, é necessário escrever casos de teste como código fonte, não como uma check list. Por exemplo, o desenvolvedor poderia escrever um simples programa que executa a função sob teste com uma entrada conhecida e comparar sua saída com os valores esperados. Isso pode ser feito para todas as entradas relevantes de todas as funções. Deste modo, rodar todos os testes é muito mais rápido do que realiza-los manualmente. Adicionalmente, haverá a certeza de que todos os testes foram executados corretamente. No entanto, este tipo de teste geralmente envolve uma mistura caótica de atividades como rodar métodos main espalhados por diversas classes.

Felizmente, há algumas ferramentas para auxiliar a automação dos casos de teste, como JUnit (HUSTED; MASSOL, 2003) que é provavelmente o mais bem sucedido *framework* de teste de unidade para a linguagem de programação Java. A criação de um caso de teste JUnit envolve a programação de uma classe Java que estende a classe TestCase (da biblioteca JUnit) e contém métodos que testam diferentes partes (ou cenários) do programa. Até sua versão 3.8, o JUnit executa todos os métodos cujo nome começam com test. O Fragmento de Código 61 mostra um exemplo de um caso de teste escrito para JUnit 3.8. Esse caso de teste verifica o código exibido no Fragmento de Código 60. É um exemplo hipotético, pois o método testado foi escrito em C# e o caso de teste em Java.

```
public class ExampleTestCase extends TestCase
{
  public void testAbs()
  {
     ExampleClass exampleObj = new ExampleClass();
```

```
        assertEquals("Wrong Absolute Value", 6,
          exampleObj.abs(-6));
    }
}
```

Fragmento de Código 61 – Exemplo de um caso de teste JUnit (versão 3.8)

No exemplo anterior, o JUnit executa automaticamente o método testAbs pois seu nome começa com a palavra test. O método assertEquals (da classe TestCase) declara que dois valores são iguais. Neste caso, o valor esperado (6) e o calculado (exampleObj.abs(-6)). Se eles forem diferentes, a string passada como primeiro parâmetro ("Wrong Absolute Value") será mostrada em uma mensagem de erro. Este é apenas um dos métodos oferecidos pelo JUnit.

O Fragmento de Código 62 mostra o mesmo caso de teste do Fragmento de Código 61 mas escrito para a versao 4.0 do JUnit. Na versao mais nova, os métodos de teste são identificados pela anotação Java (RICHARDSON et al., 2005) @Test. A vantagem de usar anotações é que o desenvolvedor não precisa nomear os métodos começando com test.

```
public class ExampleTestCase extends TestCase
{
  @Test public void absTest()
  {
     ExampleClass exampleObj = new ExampleClass();
     assertEquals("Wrong Absolute Value", 6, exampleObj.abs(-
6));
  }
}
```

Fragmento de Código 62 - Exemplo de um caso de teste JUnit (versão 4.0)

O JUnit é perfeitamente integrável com muitas IDEs como Eclipse. Dentro do Eclipse é possível executar facilmente todos os casos de teste de um projeto. Existem versões do JUnit para outras linguagens, incluindo C# (NUnit (NUNIT)), Phyton (PyUnit), Fortran (FUnit) e C++(CPPUnit). Esta família de frameworks de teste de unidade é normalmente referida como xUnit.

6.5 Test-Driven Development

Test-Driven Development (TDD) é uma técnica de desenvolvimento de software que envolve primeiramente a escritura de um caso de teste e então a

implementação somente do código necessário para fazer o teste passar. Ela começou com Agile Methodologies (MANIFESTO) e XP. Na filosofia do TDD, os casos de teste são a única documentação do software além do próprio código.

O TDD requer que um teste de unidade automático, definindo os requisitos do código, seja escrito antes de cada aspecto do próprio código. Esses testes contém verificações que são ou verdadeiras ou falsas. Rodando os testes, obtém-se uma rápida confirmação do comportamento correto do código. Os frameworks de teste baseados no conceito xUnit (veja seção 6.4) oferece um mecanismo para criar e executar conjuntos de casos de teste.

O ciclo TDD pode ser resumido como: (BECK, 2002)

- **Adicione um teste:** Cada nova funcionalidade começa com a escritura de um teste que precisa necessariamente falhar posto que foi escrito antes da própria funcionalidade. Para escrever o teste, o desenvolvedor precisa entender claramente a especificação e os requisitos da nova funcionalidade. Isto pode também implicar a alteração de um teste existente.
- **Execute todos os testes e veja o novo falhar:** Isso valida que o novo teste não passe erroneamente sem requerer nenhum código novo. O novo teste deve falhar por causa da razão esperada. Este passo testa o próprio teste, de forma negativa.
- **Faça uma pequena mudança:** Escreva algum código que faça o teste passar. O novo código escrito neste estágio não é perfeito e pode, por exemplo, fazer o teste passar de uma maneira deselegante. Isso é aceitável dado que os próximos passos o melhorarão. É importante que o código escrito seja desenvolvido somente para o teste passar. Nenhuma funcionalidade extra (e portanto não testada) deve ser prevista.
- **Rode os testes automáticos e veja todos passarem:** Se todos os testes agora passam, o programador pode estar confiante que o código está de acordo com todos os requisitos testados.
- **Melhore o código para remover duplicações:** Agora o código pode ser limpo e melhorado conforme o necessário. Re-executando todos os testes, o desenvolvedor pode estar seguro que suas alterações não estragaram nenhuma funcionalidade existente.

A abordagem descrita anteriormente traz alguns benefícios. Por exemplo:

- É o melhor modo de atingir uma cobertura de teste compreensível. Adaptar testes ao código é custoso e pode ser facilmente influenciada pelo código. Isso significa que eles não são totalmente independentes. Pior, eles podem ser falsificados, de modo a escrever testes que passem *para um determinado código, mas não sejam corretos.*
- O código aplicativo pode ser alterado a qualquer momento com uma fundação segura na regressão do testes.
- Ela pode auxiliar os desenvolvedores a escrever o código mais simples possível. Não é necessário escrever código complexo até que um teste prove que isso seja necessário.
- Os desenvolvedores têm um retorno constante sobre seu progresso, na forma de mais e mais testes que passam e demonstram a funcionalidade *requerida.*

6.6 Como Assegurar a Testabilidade

A testabilidade é uma qualidade importante do código aplicativo. Código testável geralmente significa um código melhor, mais compreensível e de mais fácil manutenção. Más arquiteturas de software e estilos de programação ruins podem dificultar muito os processos de teste de unidade.

Um objetivo essencial do processo de teste de unidade é ser capaz de testar as classes em completo isolamento. Para atingir essa meta, o código não pode depender do container para rodar. Uma vantagem de usar uma abordagem de testes automáticos, como o JUnit integrado ao Eclipse, é que o desenvolvedor pode fácil e rapidamente rodar todos os testes, a qualquer momento, apenas pressionando um botão na IDE. Todos os testes são executados velozmente e o desenvolvedor pode verificar se o código está correto. Se o código depender do container para ser executado, esta vantagem é perdida. Os EJBs dependem pesadamente do container. Portanto, os testes de unidade dos EJBs são impraticáveis.

Para testar classes em isolamento, é necessário substituir suas dependências por objetos falsos. Deste modo, somente uma classe será testada por caso de teste, ao invés de testar também suas dependências. Testar uma hierarquia de classe é o propósito do teste de integração, não do teste de unidade. Isso significa que o

código precisa ser plugável de modo a assegurar testabilidade. Uma boa abordagem para garantir que o código seja plugável é programar através de interfaces ao invés de classes. Como conseqüência, é possível substituir a implementação real de uma dependência de uma classe por um objeto falso como um stub (seção 6.6.1) ou um mock object (seção 6.6.2).

Um típico exemplo de dependência é a conexão com o banco de dados. Muitas classes de aplicações corporativas precisam de uma conexão a um BD para funcionar. Entretanto, não é uma boa idéia usar o banco de dados para testes de unidade. Primeiramente, é preciso uma certa quantidade de dados, sem a qual o teste não passará. Em segundo lugar, o teste será mais difícil de configurar. Por exemplo, dependerá de um arquivo de configuração externo que contenha a string de conexão. Portanto, o caso de teste não será auto-documentado. Por fim, ao usar um BD, o teste será executado muito mais lentamente.

As próximas seções discutem duas das principais técnicas para realizar testes de unidade. Elas focam em como isolar as classes sob teste usando stubs e mock objects.

6.6.1 Stubs

O modo mais óbvio de testar classes em isolamento é usar objetos stub para substituir colaboradores em tempo de execução. Um stub implementa um subconjunto das funcionalidades do colaborador que é requisitado para suportar o caso de teste. Um stub geralmente não visa ser uma implementação realística da interface completa do colaborador. Normalmente ele requer uma certa configuração para funcionar adequadamente.

É relativamente fácil criar um stub de uma interface. Geralmente é possível prover uma implementação abstrata que lance UnsupportedOperationException em todos os métodos, e no momento do teste, sobrescrever aqueles métodos que são necessários para o caso de teste. Ou então, é possível usar uma implementação mais genérica configurada de forma diferente para cada caso de teste.

Criar stub de classes é difícil, embora seja possível sobrescrever métodos seletivamente. Criar stub de classes sobrescrevendo métodos não funciona quando os construtores introduzem dependências nos objetos que são insignificantes para o

teste, mas são difíceis de serem satisfeitos. Portanto, a programação através de interfaces ao invés de classes torna a criação de stubs mais fácil e efetiva.

A principal desvantagem dos stubs é o esforço requerido para escrevê-los. Em alguns casos, pode ser possível reutilizar stubs em diferentes partes de um projeto ou usar stubs de terceiros para APIs comuns. Os mock objects são uma alternativa aos stubs e são particularmente valiosos em testes de unidade. A próxima seção os explica em detalhes.

6.6.2 Mock Objects

Os mock objects (também chamados de mocks) são objetos simulados que dissimulam o comportamento de objetos reais em ambientes controlados. Eles são intencionalmente ainda menos realísticos do que os stubs. Os mocks não visam prover uma implementação real da interface em questão. Diferentemente dos stubs, os mocks restringem seu uso, pois são projetados para forçar expectativas. O propósito de um mock não é assegurar que o código sob teste funcione apropriadamente com colaboradores trabalhando de uma determinada forma, mas que o código sob teste invoque os métodos dos colaboradores em uma certa ordem passando parâmetros específicos.

Os mocks são tipicamente customizados para cada caso de teste, provendo a mínima funcionalidade requerida pelo cenário. Eles ajudam a focar o teste de unidade nas operações relevantes, as quais são expressas através de expectativas.

Os mocks tornam a simulação de condições de erros particularmente fácil: por exemplo, o que acontece quando um colaborador lança uma particular exceção. A habilidade de simular facilmente uma gama de condições de erros é essencial para o desenvolvimento de um software robusto, e é a razão chave pela qual os mocks são tão importantes para um teste de unidade efetivo.

O Fragmento de Código 63 mostra uma classe de exemplo a qual possui um colaborador que implementa a interface CollaboratorInterface. O Fragmento de Código 64 mostra um caso de teste que verifica o método myMethod da classe ExampleClass usando uma instância de MockCollaborator com um mock para o atributo collaborator da classe sob teste. A classe MockCollaborator implementa a interface CollaboratorInterface. O caso de teste primeiramente instancia um objeto MockCollaborator e executa seu método setExpectedCollaboratorsMethod que

significa que o mock espera que o método collaboratorsMethod seja chamado. Então, o caso de teste passa o mock ao atributo collaborator. Depois disso, myMethod é executado. Por fim, o caso de teste executa o método verify do mock que averigua se todas as expectativas foram satisfeitas. Se o método collaboratorsMethod não for chamado, o método verify lança uma exceção e o caso de teste consequentemente falha.

```
public class ExampleClass {

  private CollaboratorInterface collaborator;

  public void setMyCollaborator(CollaboratorInterface
collaborator)
  {
    this.collaborator = collaborator;
  }

  public void myMethod()
  {
    //Some code

    collaborator.collaboratorsMethod();

    //More code
  }
}
```

Fragmento de Código 63 – Classe de exemplo que pode ser testada usando Mock Objects

```
public void testExampleClass()
{
  MockCollaborator mockCollaborator = new MockCollaborator();
  mockCollaborator.setExpectedMyMethod();
  ExampleClass exampleClass = new ExampleClass();
  exampleClass.setMyCollaborator(mockCollaborator);
  exampleClass.myMethod();
  mockRequest.verify();
}
```

Fragmento de Código 64 – Caso de teste com Mock Objects (para o Fragmento de Código 63)

Um problema dos mock objects como usado acima é que, como com stubs, é necessário codificar um mock, com a habilidade de atribuir expectativas, para cada interface que um mock é necessário. Existem algumas bibliotecas de mocks para APIs comuns como a API Servlet (por exemplo em (MOCK)). O volume de código

dos mocks dessas bibliotecas sugere que uma solução simples é desejada para se criar mocks das interfaces da aplicação.

É possível estender uma classe base comum para mocks (como com.mockobjects.MockObject) para criar mocks específicos da aplicação. A derivação dessa classe oferece um grande suporte para as expectativas. As variáveis das expectativas nas subclasses serão verificadas automaticamente ao se chamar o método verify. No entanto, ainda haverá um monte de trabalho para criar os mocks. A próxima seção discute uma melhor solução para testes de unidade baseados em mocks.

6.6.2.1 Mock Objects Dinâmicos

Mock objects dinâmicos são criados em tempo de execução dos casos de teste para implementar as interfaces requeridas como colaboradores. Não é necessário codificar a real implementação dessas interfaces. Existem muitas bibliotecas de mocks dinâmicos, por exemplo EasyMock (EASYMOCK) para Java ou NMock (NMOCK) para .NET.

A utilização de mock objects dinâmicos envolve os seguintes passos:

- Criar a instância do mock a partir de uma factory provida pela biblioteca de mocks dinâmicos e fazer um cast para o tipo desejado.
- Criar expectativas gravando as chamadas no mock. As expectativas envolvem as chamadas dos métodos esperados no mock e o valor de retorno correspondente (ou exceções, se o desenvolvedor deseja simular erros). O modo como as expectativas são definidas nos mocks é ligeiramente diferente dependendo da biblioteca.
- Passar o mock como colaborador da classe sob teste.
- Invocar o caso de teste que usa o mock.
- Verificar o mock de modo a averiguar se todas as chamadas esperadas foram realizadas.

O Fragmento de Código 65 mostra como usar a biblioteca EasyMock. Esse exemplo é um caso de teste que testa o método withraw da classe BankImpl. Essa classe usa um objeto que implementa a interface BankDao que é o colaborador que será substituído pelo mock. Nesse caso, o resultado esperado do método chamado

é a exceção DAOException. Portanto, se o método não lançar tal exceção, o teste falhará.

```
public void testWithdrawal() throws Exception
{
  DAOException expectedException = new DAOException("", null)
();
  MockControl mc = MockControl.createControl(BankDao.class);
  BankDao dao = (BankDao) mc.getMock();
  dao.getBalance();
  mc.setThrowable(expectedException);

  // Stop scripting the mock and make it expect calls

  mc.replay();
  BankImpl bank = new BankImpl();
  bank.setDao(dao);
  try
  {
    bank.withdraw(25.0);
    fail("DataAccessException should have been thrown");
  }
  catch (DataAccessException ex)
  {
    //OK
    assertEquals("Exception was propagated",
expectedException, ex);
  }
  //Verify the expected call was made on the mock
  mc.verify();
}
```

Fragmento de Código 65 – Exemplo de um caso de teste com EasyMock

Para criar expectativas usando o EasyMock, é necessário fazer chamadas ao método esperado enquanto o mock está no estado de "gravação", ou seja, antes de colocar o mock em seu estado "replay" (através da execução do método replay do MockControl). Além disso, é necessário, para cada chamada, fazer uma chamada ao correspondente MockControl de modo a definir o valor de retorno. No exemplo mostrado no código, o valor de retorno é uma exceção. Então, o método usado é setThrowable passando a exceção que deve ser lançada.

O Fragmento de Código 66 mostra o mesmo caso de teste do Fragmento de Código 65, mas em uma versão .NET usando NMock ao invés de EasyMock. Com NMock, todas as expectativas e os resultados correspondentes são definidos na classe DynamicMock (basicamente o equivalente ao MockControl). Isso é feito

passando o nome do método ao invés de executar o método em si como com EasyMock. No exemplo, o método ExpectAndThrow é usado para criar a expectativa que o método withdraw será chamado recebendo o valor 25.0 como parâmetro. Quando o método withdraw for chamado com esse valor, uma exceção DAOException será lançada. Ao contrário, se um parâmetro diferente é passado, o mock lançará uma exceção diferente fazendo com que o teste falhe.

```
public void testWithdrawal()
{
  DAOException expectedException = new DAOException();
  DynamicMock mc = new DynamicMock(typeof(BankDao));
  mc.ExpectAndThrow("withdraw", expectedException,
      new object[](25.0));

  //Stop scripting the mock and make it expect calls

  BankDao dao = (BankDao) mc.MockInstance;
  BankImpl bank = new BankImpl();
  bank.setDao(dao);
  try
  {
    bank.withdraw(25.0);
    fail("DataAccessException should have been thrown");
  }
  catch (DataAccessException ex)
  {
    //OK
    assertEquals("Exception was propagated",
expectedException, ex);
  }
  //Verify the expected call was made on the mock
  mc.verify();
}
```

Fragmento de Código 66 - Exemplo de um caso de teste com NMock

Os mock objects são mais úteis quando o desenvolvedor segue boas práticas de programação, como a codificação através de interfaces, e cada classe tem um conjunto gerenciável de responsabilidades. A biblioteca de mocks dinâmicos geralmente precisa de uma interface para criar o mock. Esta é outra importante razão para programar através de interfaces ao invés de classes.

6.6.3 Uso de Interfaces e Object Factory (IoC)

Quando se codifica através de interfaces, é muito mais fácil substituir a implementação real de um colaborador por um stub ou mock. Isso se torna ainda mais fácil quando a classe sob teste não realiza por si só um lookup de seus colaboradores. Se a classe não é responsável por recuperar suas dependências e, por exemplo, IoC é usado, o caso de teste pode conectar a classe testada com seus colaboradores. Neste caso, os colaboradores seriam falsos objetos como stubs ou mocks. O Fragmento de Código 67 mostra uma classe cujo teste em isolamento é muito difícil.

```
public void MyClass()
{
  private MyCollaboratorInterface myCollaborator;

  public MyClass()
  {
    myCollaborator = new MyCollaboratorClass();
  }

  public void myMethod()
  {
    //Algum código

    myCollaborator.myCollaboratorMethod();

    //Mais código
  }
}
```

Fragmento de Código 67 – Exemplo de uma classe cujo teste em isolamento é muito difícil

A classe do exemplo anterior usa interfaces ao invés de classes. Porém não é suficiente para assegurar sua testabilidade, pois a classe instancia seu colaborador dentro do construtor (a classe MyCollaborator implementa a interface MyCollaboratorInterface). Esta abordagem torna impossível substituir tal colaborador por um objeto falso dado que a dependência é um atributo privado da classe e não possui um setter público. Se a classe usasse um setter público, o caso de teste poderia substituir o colaborador por um mock, por exemplo. No entanto, mesmo nesse caso, a testabilidade poderia ser comprometida. Supondo que o construtor da classe testada acessasse o banco de dados, o teste não seria mais realizado em isolamento e poderia falhar se acontecesse algum erro na conexão com o BD.

Uma abordagem melhor é utilizar uma Object Factory ao invés de executar diretamente o construtor da classe MyCollaboratorClass. Deste modo, a Object Factory poderia ser configurada de modo a retornar um mock ou stub ao invés da implementação real do colaborador. Entretanto, o caso de teste dependeria de um recurso externo para funcionar: o arquivo de configuração da Factory.

A *última abordagem poderia ser melhorada usando Inversão de Controle*. Uma factory externa seria a responsável por conectar os objetos. Conseqüentemente, a classe MyClass teria um setter público para o atributo myCollaborator (no caso de setter injection) ou um construtor que recebe um parâmetro de tipo MyCollaboratorInterface (no caso de constructor injection). Deste modo, o caso de teste não usa a Factory IoC mas conecta sozinho a classe MyClass à implementação falsa de MyCollaboratorInterface.

A IoC auxilia imensamente a testabilidade por eliminar muitas necessidades de singletons, externalizar os lookups, facilitar a codificação através de interfaces, e encorajar a criação de objetos aplicativos como POJOs ao invés de objetos especiais como EJBs (JOHNSON, 2004).

6.7 Teste de uma Aplicação Baseada no Adaptive Framework

Dado que grande parte do código dessas aplicações é gerado automaticamente e aplicações anteriores já usaram os modelos de código atuais, o código criado pelo gerador é de certo modo já testado. Quanto maior o número de projetos construídos com certo conjunto de templates, menor é o número de erros no código. No entanto, no primeiro projeto que usa um conjunto de templates, estes não são ainda confiáveis e devem ser pesadamente testados. O mesmo é verdade depois de cada alteração nos templates. Quando os modelos estão estáveis, é mais provável que os erros surjam devido a erros nos arquivos XML de definição dos componentes do que por causa de erros em templates. Nas aplicações reais feitas com o AF, a maioria dos erros ocorreu devido ao código manual e enganos nos arquivos de definição.

Pelas razões discutidas anteriormente, mesmo o código gerado automaticamente deve ser pesadamente testado. Este projeto foca no teste de componentes de serviços, pois eles são aqueles que contêm a maior parte do código manual. Esta seção mostra um bom modo para se testar os componentes de serviços utilizando mock objects. Os mesmos conceitos podem ser aplicados aos

componentes de dados. Todos os exemplos mostrados aqui usam o componente de serviço ClientManager como classe de teste e NMock como biblioteca de mocks dinâmicos.

O primeiro passo para testar uma classe em isolamento é substituir todas as suas dependências por implementações falsas que não influenciam nos testes. Dado que as aplicações feitas sobre o AF são baseadas na inversão de controle e na injeção de dependências, além de serem codificadas através de interfaces, é relativamente fácil testar suas classes em isolamento. No caso de um componente de serviços, não é apenas uma classe que será testada em isolamento, mas uma hierarquia de classes (BaseCmp mais Cmp). Portanto, é melhor afirmar que um componente é testado em isolamento.

Um recém criado componente de serviços (sem código manual) possui as seguintes dependências:

- Logger de exceções (propriedade ExcpLogSvc de tipo IExcpLogSvc)
- Object Factory (propriedade ObjectFactory de tipo IObjectFactory)
- Gestor de transações (propriedade TrxMgrSvc de tipo ITrxMgrSvc)
- Gestor de conexões ao banco de dados (propriedade ConnUtils de tipo IConnUtils)

Para testar um componente de serviços em isolamento é preciso substituir as dependências descritas anteriormente com mock objects. A dependência mais simples de substituir é o logger de exceções, pois seus métodos não retornam nenhum objeto (que seria uma nova dependência). O Fragmento de Código 68 mostra como substituir o logger de exceções. A instância do mock substitui a implementação real do logger de exceções e quando o componente ClientManager deve criar um log de uma exceção, o mock simplesmente não faz nada. Quando é importante verificar se em um caso de teste, foi criado um log de uma exceção, é necessário criar expectativas como explicado na seção 6.6.2.

```
Cmp_ClientManager clientManager = new Cmp_ClientManager();

DynamicMock excpLog = new DynamicMock(typeof(IExcpLogSvc));
clientManager.ExcpLogSvc = (IExcpLogSvc)excpLog.MockInstance;
```

Fragmento de Código 68 – Como criar um mock para o logger de exceções

O gestor de transações é um pouco mais complicado de substituir do que o logger de exceções, pois ele precisa retornar transações que também devem ser

substituídas por mocks. O Fragmento de Código 69 mostra como criar um mock para o gestor de transações. O exemplo usa o método SetupResult (ao invés do Expect) porque não é necessário criar expectativas, somente retornar sempre um objeto falso. A instância do mock do gestor de transações retorna sempre a mesma transação, a qual nunca realiza um rollback (IsRollBackOnly = false). Se o gestor de transações deve ser testado, é necessário criar expectativas.

```
Cmp_GestReplica gestReplica = new Cmp_GestReplica();

DynamicMock trx = new DynamicMock(typeof(ITrx));
trx.SetupResult("IsRollbackOnly", false);
DynamicMock trxMgr = new DynamicMock(typeof(ITrxMgrSvc));
trxMgr.SetupResult("GetTransaction", trx.MockInstance,
  typeof(ITrxDefinition));

gestReplica.TrxMgrSvc = (ITrxMgrSvc)trxMgr.MockInstance;
```

Fragmento de Código 69 – Como criar um mock para o gestor de transações

A conexão ao banco de dados é usada por um componente de serviços somente no caso de acesso direto ao BD. Isso geralmente ocorre quando é necessário executar uma operação não suportada pelos componentes de dados (por exemplo, comandos SQL como count e max). Portanto, o gestor de conexões não é usado nos métodos do componente ClientManager e não será substituído por um mock.

A dependência mais difícil de substituir (das descritas anteriormente) em um componente de serviços do AF é a Object Factory, pois é necessário um mock para cada objeto gerenciado por ela e usado pelo método testado, como por exemplo as classes HomeCmp. A classe home é ainda mais difícil de ser substituída, dado que é necessário criar expectativas para assegurar que ela é usada de modo correto. O Fragmento de Código 70 mostra como substituir a Object Factory e um objeto gerenciado por ela, no caso um objeto da classe HomeCmp_Client.

```
Cmp_ClientManager clientManager = new Cmp_ClientManager();

DynamicMock homeClient = new
DynamicMock(typeof(IHomeCmp_Client));

DynamicMock factory = new DynamicMock(typeof(IObjectFactory));
factory.ExpectAndReturn("GetObject", homeClient.MockInstance,
  "HomeCmp_Client");
clientManager.ObjectFactory =
(IObjectFactory)factory.MockInstance;
```

Fragmento de Código 70 – Como criar um mock para a object factory

No exemplo anterior, a factory retorna um objeto home mas este não é usado para nada. Geralmente, é necessário criar algumas expectativas na classe home para os métodos FindCmp, GetCmp, RemoveCmp e CreateCmp. Para verificar os três primeiros métodos é preciso também averiguar as chaves de busca além da chamada ao método. Por exemplo, para testar o método DeleteClient do componente ClientManager, é preciso verificar se o método atribui o correto valor à chave ClientID na classe HomeCmp_Client antes de executar o método RemoveCmp como mostrado no Fragmento de Código 71.

```
DynamicMock homeClient = new
DynamicMock(typeof(IHomeCmp_Client));
homeClient.Expect("ClientID", clientID);
homeClient.Expect("RemoveCmp");
```

Fragmento de Código 71 – Como verificar o correto uso do método RemoveCmp

O método FindCmp sempre retorna uma coleção CmpColl. É muito complicado criar um mock para uma coleção .NET pois alguns comandos como o foreach implicitamente chama muitos métodos e propriedades que precisam retornar valores corretos. Ao invés de criar um mock de uma coleção é melhor simplesmente criar uma coleção real de mock objects. Ao testar um componente AF não é possível retornar uma simples classe List de .NET porque o método FindCmp retorna uma coleção que implementa a interface ICmpColl. Uma solução melhor é criar uma classe que estende a classe List genérica e implementa a interface ICmpColl como mostrado no Fragmento de Código 72. Nenhum método precisa ser sobrescrito ou implementado.

```
public class CmpCollTest_Documento : List<ICmp_Documento>,
  ICmpColl_Documento
{
}
```

Fragmento de Código 72 – Implementação de teste da coleção ICmpColl_Documento

Usando as técnicas discutidas anteriormente, é possível testar um componente de serviços em isolamento. O Fragmento de Código 73 mostra um caso de teste completo para o método DeleteClient. Este exemplo testa o cenário de um cliente que possui dois pedidos. O teste não cria mocks nem para o logger de exceções nem para o gestor de conexões porque o método não os utiliza. O teste cria mocks

para dois pedidos (order1 e order2) e cria expectativas no mock da classe home. O método DeleteClient precisa chamar corretamente o método RemoveCmp da classe home do componente Client. Então, deve atribuir os valores corretos às chaves de busca da classe home do componente Order e executar o método RemoveCmp duas vezes (uma para cada pedido encontrado). No final do caso de teste, o métodos Verify dos mocks são chamados de modo a garantir que todas expectativas foram satisfeitas.

```
Guid clientID = new Guid("{A71BF196-FDC3-42bc-9967-
CD27C2C9C041}");
Guid orderID1 = new Guid("{E50C0CFE-200A-4838-B052-
7EDFBB195AF0}");
Guid orderID2 = new Guid("{2D2A007C-0B66-4c98-AA69-
607D3BFCCA9D}");

Cmp_ClientManager clientManager = new Cmp_ClientManager();

DynamicMock trx = new DynamicMock(typeof(ITrx));
trx.SetupResult("IsRollbackOnly", false);

DynamicMock trxMgr = new DynamicMock(typeof(ITrxMgrSvc));
trxMgr.SetupResult("GetTransaction", trx.MockInstance,
  typeof(ITrxDefinition));
clientManager.TrxMgrSvc = (ITrxMgrSvc)trxMgr.MockInstance;

DynamicMock homeClient = new
DynamicMock(typeof(IHomeCmp_Client));
homeClient.Expect("ClientID", clientID);
homeClient.Expect("RemoveCmp");

DynamicMock order1 = new DynamicMock(typeof(ICmp_Order));
order1.SetupResult("OrderID", orderID1);

DynamicMock order2 = new DynamicMock(typeof(ICmp_Order));
order2.SetupResult("OrderID", orderID2);

CmpCollTest_Order orders = new CmpCollTest_Order();
orders.Add((ICmp_Order)order1.MockInstance);
orders.Add((ICmp_Order)order2.MockInstance);

DynamicMock homeOrder = new
DynamicMock(typeof(IHomeCmp_Order));
homeOrder.Expect("ClientID", clientID);
homeOrder.ExpectAndReturn("FindCmp", orders,
  new object[] {DefsHomeCmp_Order.idFM_STD});

homeOrder.Expect("OrderID", orderID1);
homeOrder.Expect("RemoveCmp");
```

```
homeOrder.Expect("OrderID", orderID2);
homeOrder.Expect("RemoveCmp");

DynamicMock factory = new DynamicMock(typeof(IObjectFactory));
factory.ExpectAndReturn("GetObject", homeClient.MockInstance,
  "HomeCmp_Client");
factory.ExpectAndReturn("GetObject", homeOrder.MockInstance,
  "HomeCmp_Order");
clientManager.ObjectFactory =
(IObjectFactory)factory.MockInstance;

clientManager.DeleteClient(clientID);

homeClient.Verify();
homeOrder.Verify();
```

Fragmento de Código 73 – Caso de teste 1 para o método DeleteClient

O Fragmento de Código 74 mostra outro caso de teste para o método DeleteClient. Nesse exemplo, o cliente não possui pedidos. Por conseguinte, ao invés de definir expectativas na classe home do componente Order, o método ExpectNoCall do mock é usado. Se o método DeleteClient tentar apagar um pedido, uma exceção será lançada e, consequentemente, o teste falhará.

```
Guid clientID = new Guid("{A71BF196-FDC3-42bc-9967-
CD27C2C9C041}");

Cmp_ClientManager clientManager = new Cmp_ClientManager();

DynamicMock trx = new DynamicMock(typeof(ITrx));
trx.SetupResult("IsRollbackOnly", false);

DynamicMock trxMgr = new DynamicMock(typeof(ITrxMgrSvc));
trxMgr.SetupResult("GetTransaction", trx.MockInstance,
  typeof(ITrxDefinition));
clientManager.TrxMgrSvc = (ITrxMgrSvc)trxMgr.MockInstance;

DynamicMock homeClient = new
DynamicMock(typeof(IHomeCmp_Client));
homeClient.Expect("ClientID", clientID);
homeClient.Expect("RemoveCmp");

CmpCollTest_Order orders = new CmpCollTest_Order();

DynamicMock homeOrder = new
DynamicMock(typeof(IHomeCmp_Order));

homeOrder.Expect("ClientID", clientID);
homeOrder.ExpectAndReturn("FindCmp", orders,
```

```
  new object[] {DefsHomeCmp_Order.idFM_STD});
homeOrder.ExpectNoCall("RemoveCmp");

DynamicMock factory = new DynamicMock(typeof(IObjectFactory));
factory.ExpectAndReturn("GetObject", homeClient.MockInstance,
  "HomeCmp_Client");
factory.ExpectAndReturn("GetObject", homeOrder.MockInstance,
  "HomeCmp_Order");
clientManager.ObjectFactory =
(IObjectFactory)factory.MockInstance;

clientManager.DeleteClient(clientID);

homeClient.Verify();
homeOrder.Verify();
```

Fragmento de Código 74 - Caso de teste 2 para o método DeleteClient

O Fragmento de Código 75 mostra um terceiro caso de teste para o mesmo método DeleteClient. Dessa vez, não existe um cliente com o identificador (ClientID) usado. Logo, o mock da classe home lança uma exceção CmpNotFoundException, a qual o método testado deve traduzir para a exceção CmpMethodException com o código ErrorCode_ClientNotFound (que é definido na classe DefsCmp_ClientManager). Se tal exceção (com o código correto) não for lançada, o teste falhará (usando o método assertFails da biblioteca NUnit).

```
Guid clientID = new Guid("{A71BF196-FDC3-42bc-9967-
CD27C2C9C041}");

Cmp_ClientManager clientManager = new Cmp_ClientManager();

DynamicMock trx = new DynamicMock(typeof(ITrx));
trx.SetupResult("IsRollbackOnly", false);

DynamicMock trxMgr = new DynamicMock(typeof(ITrxMgrSvc));
trxMgr.SetupResult("GetTransaction", trx.MockInstance,
  typeof(ITrxDefinition));
clientManager.TrxMgrSvc = (ITrxMgrSvc)trxMgr.MockInstance;

DynamicMock excpLog = new DynamicMock(typeof(IExcpLogSvc));
clientManager.ExcpLogSvc = (IExcpLogSvc)excpLog.MockInstance;

DynamicMock homeClient = new
DynamicMock(typeof(IHomeCmp_Client));
homeClient.Expect("ClientID", clientID);
homeClient.ExpectAndThrow("RemoveCmp", new
CmpNotFoundException(),
  new object[0]);
```

```
DynamicMock factory = new DynamicMock(typeof(IObjectFactory));
factory.ExpectAndReturn("GetObject", homeClient.MockInstance,
  "HomeCmp_Client");
clientManager.ObjectFactory =
(IObjectFactory)factory.MockInstance;

try
{
  clientManager.DeleteClient(clientID);
  assertFails();
}
catch (CmpMethodException e)
{
if (e.ErrorCode !=
    DefsCmp_ClientManager.ErrorCode_ClientNotFound)
  {
    assertFails();
  }
}

homeClient.Verify();
```

Fragmento de Código 75 - Caso de teste 2 para o método DeleteClient

É também possível testar outros aspectos dos métodos dos componentes de serviços, por exemplo, se o roll back da transação é feito corretamente, ou se é criado um log correto de uma mensagem.

7 Processos de Desenvolvimento de Software com Geração Automática de Código

Os capítulos anteriores discutiram diferentes vertentes da geração automática de código e como ela influencia as decisões arquiteturais do sistema de software. Este capítulo faz uma breve introdução a um ambiente clássico de desenvolvimento de software baseado na Fábrica de Software do Laboratório de Tecnologia de Software da Escola Politécnica da USP. Em seguida, um gerador de código é introduzido em tal ambiente, alterando os processos existentes. Então é realizado um estudo sobre este novo ambiente criado, analisando como o gerador altera os custos e prazos de cada atividade, além da alocação de recursos e das ferramentas necessárias. O estudo sobre o ambiente existente e a criação do novo ambiente é realizado através de modelagens em BPMN (BPMN, 2006).

7.1 Ambiente Tradicional de Desenvolvimento de Software

O ambiente de desenvolvimento descrito nesta seção se baseia em processo em cascata. A Figura 23 mostra a modelagem em BPMN de um ambiente tradicional de desenvolvimento de software, isto é, sem a utilização de um gerador automático de código. O modelo contém as atividades básicas da etapa de desenvolvimento (desde a análise até a implantação) somada a uma atividade de gerência que é a criação do plano de projeto. Trata-se de uma modelagem simplificada do ambiente por conter somente as atividades e documentos essenciais.

7.1.1 Atividades do Processo

Esta seção descreve em mais detalhes as atividades do processo de desenvolvimento de software ilustrado na Figura 23

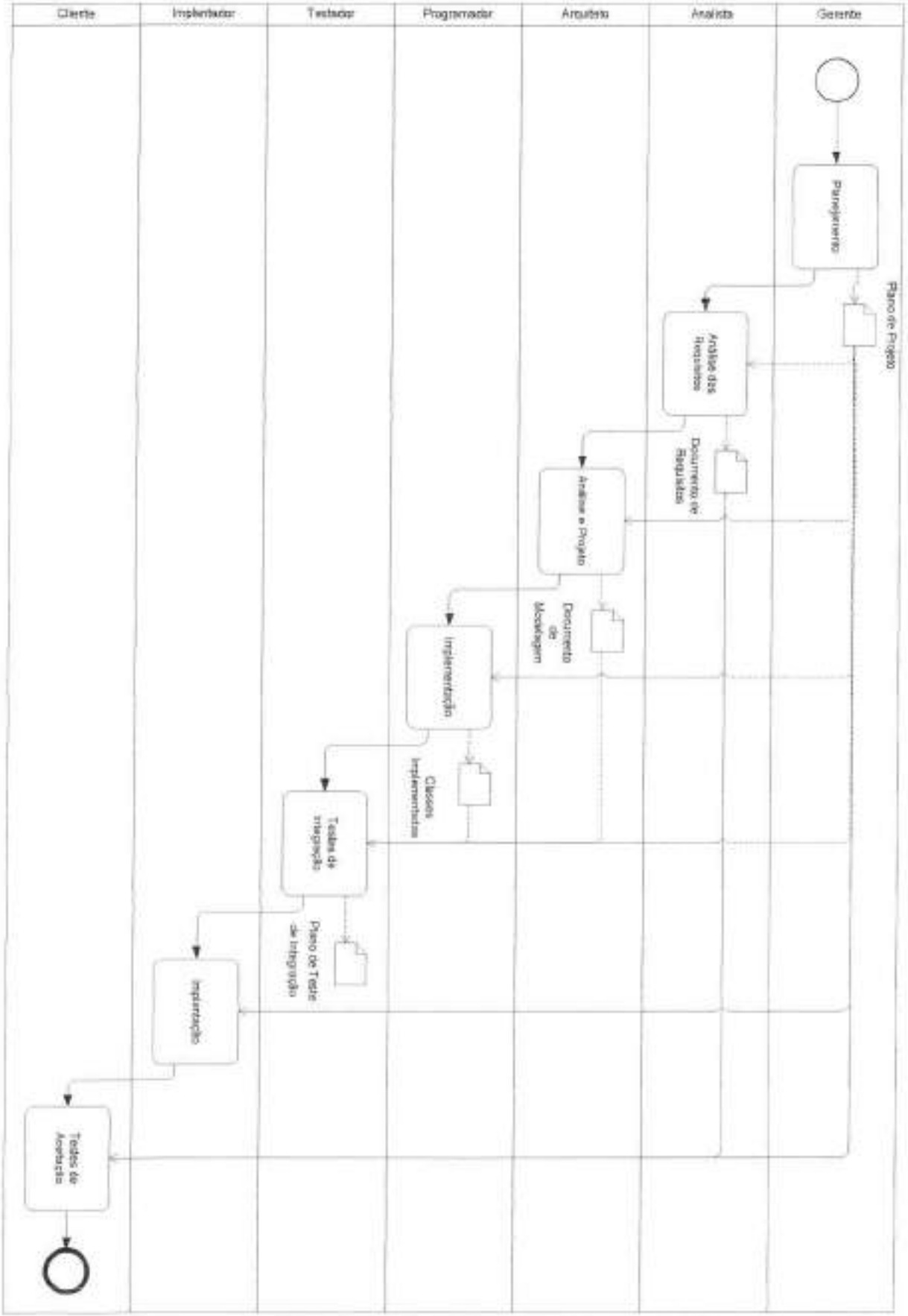

Figura 23 - Ambiente Tradicional de Desenvolvimento de Software

7.1.1.1 Planejamento

O planejamento do projeto tem como finalidade a criação de um plano de desenvolvimento que possua compromissos razoáveis, que sejam baseadas em estimativas e riscos. Durante esta atividade é criado o plano de projeto, o qual contém o cronograma das atividades seguintes, os recursos necessários para o projeto, uma descrição detalhada dos riscos, além de uma estimativa do custo do projeto.

7.1.1.2 Análise dos Requisitos

A análise dos requisitos é responsável pelo levantamento, entendimento e documentação dos requisistos do sistema de software. Essa atividade pode ser realizada através de entrevistas com o cliente e usuários, ou a partir de um documento de especificação apresentado pelo cliente. A partir dessa análise é criado o modelo de casos de uso que integra o documento de especificação dos requisitos.

7.1.1.3 Análise e Projeto

Esta atividade tem a função de refinar os requisitos levantados na atividade anterior, estruturando-os de modo a projetar o software identificando as classes (com seus atributos e métodos) e as relações entre elas. Tais classes são documentadas em diagramas UML contidos no documento de modelagem. Este documento contém os diagramas de classes, eventuais diagramas de seqüência, atividades e estados, além do diagrama de implantação que define a arquitetura física da aplicação.

7.1.1.4 Implementação

A atividade de implementação compreende a codificação das classes descritas no documento de modelagem, além da codificação e execução dos testes de unidade para cada uma das classes implementadas. Caso a metodologia TDD (seção 6.5) seja utilizada, primeiramente devem ser criados somente os esqueletos das classes (de modo a compilar os casos de teste criados a seguir). Em seguida, codificam-se os casos de teste. E por fim, as classes são implementadas. Quando todos os testes passarem, a atividade de implementação está terminada.

7.1.1.5 Testes de Integração

Os testes de integração são os responsáveis por verificar a integração entre as classes produzidas e testadas isoladamente durante a fase de implementação. Para realizar essa atividade é necessário criar um plano de testes (baseando-se nos documentos produzidos nas fases anteriores do desenvolvimento) e aplicá-lo.

7.1.1.6 Implantação

Essa atividade é responsável por entregar ao cliente o sistema final e instala-lo no ambiente de produção. Ela é dependente do acordo definido com o cliente e do contexto do projeto

7.1.1.7 Testes de Aceitação

Durante esta atividade são realizados os testes para a aceitação do sistema por parte do cliente ou então o sistema como um todo é testado no ambiente de produção.

7.1.2 Atores do Processo

Esta seção descreve em mais detalhes os atores que participam do processo de desenvolvimento de software ilustrado na Figura 23.

7.1.2.1 Gerente de Projeto

É o responsável pelo planejamento e acompanhamento do projeto. Deve fazer estimativa dos prazos, custos e riscos do projeto, além de alocar eficientemente os recursos nas diversas atividades.

7.1.2.2 Analista

O analista deve entender as necessidades do cliente de modo a propor uma solução de software para o problema. Para isso, ele levanta os requisitos da aplicação, detalhando-os no documento de especificação de requisitos que será refinado nas etapas seguintes do desenvolvimento até ser transformado no software em si.

7.1.2.3 Arquiteto

O arquiteto é o ator responsável por entender o documento de requisitos feito pelo analista e refiná-lo, identificando as classes do sistema de modo a criar a arquitetura geral do sistema que deverá ser seguida pelos programadores na etapa seguinte do projeto. O arquiteto também é o responsável por identificar os componentes reusáveis (ou bibliotecas de componentes), seja aproveitando um já existente ou projetando um novo que poderá ser utilizado em projetos posterioes. Além disso, ele deve determinar eventuais frameworks nos quais a aplicação deve se basear ou containers onde o software será executado. Em alguns casos, o próprio arquiteto (ou programadores mais experientes) implementa a primeira fatia vertical da aplicação de modo a validar a arquitetura (seção 1.4). Neste caso, os programadores geralmente se baseiam nesse código produzido inicialmente para implementar os outros requisistos do sistema.

7.1.2.4 Programador

A partir do documento de especificação de requisitos e do documento de modelagem, o programador deve codificar a aplicação. Ele deve implementar corretamente os requisitos determinados pelo cliente, seguindo à risca a arquitetura definida pelo arquiteto. Se foi criada uma fatia vertical inicial da aplicação, normalmente o programador se baseia em tal código para construir o resto do software. Se é usada a metodologia TDD, o programador também é responsável por programar os casos de teste e executá-los de modo a verificar a correta implementação dos requisitos.

7.1.2.5 Testador

É o responsável por elaborar e aplicar um plano de teste de integração para a aplicação.

7.1.2.6 Implantador

É o responsável por preparar o ambiente de produção para receber o sistema e instalar a aplicação.

7.1.2.7 Cliente

É o responsável por validar o sistema em seu ambiente de produção segundo os requisitos acordados no início do projeto e registrados no documento de especificação dos requisitos.

7.2 *Ambiente de Desenvolvimento de Software com Geração Automática de Código*

Esta seção descreve um ambiente de desenvolvimento de software que se baseia na geração automática de código. Trata-se do mesmo ambiente apresentado na seção 7.1 com a inserção de um gerador automático de código (no caso, o Adaptive Framework explicado nos capítulos 4 e 5). A Figura 24 mostra o que muda no ambiente tradicional de software após a inclusão do gerador automático de código. As mudanças nas relações entre as atividades ocorrem apenas entre as atividades de análise e design e a implementação. Entretanto, ocorrem também alterações internamente à atividade de planejamento do projeto (não representada na Figura 24), como será visto a seguir.

7.2.1 Alterações nas Atividades do Processo

Esta seção descreve em mais detalhes as alterações ocorridas nas atividades do ambiente de desenvolvimento após a introdução do gerador automático de código.

7.2.1.1 Planejamento

A criação do plano de projeto deve levar em conta a existência do gerador automático de código, pois ele pode alterar o tempo necessário para realizar as atividades manuais. A alocação de recursos também muda, pois geralmente reduz o número de pessoas.

A geração automática de código faz com que a fase de implementação requeira menos tempo para ser realizada, pois parte da codificação é realizada imediatamente pelo gerador. A quantidade necessária de programadores pode ser reduzia devido ao fato de o gerador fazer o trabalho mecânico dos programadores.

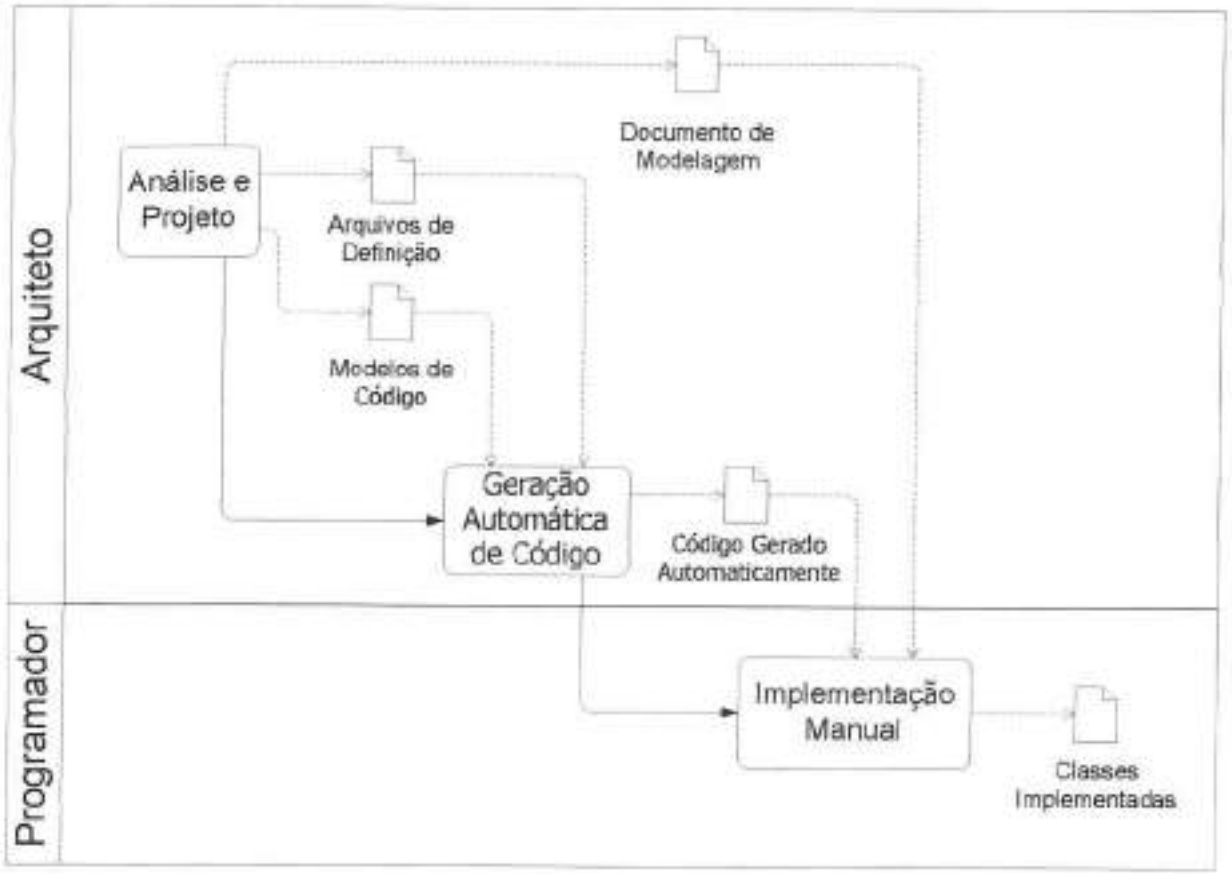

Figura 24 – Ambiente de Desenvolvimento de Software com Geração Automática de Código

O tempo necessário para a conclusão da etapa de análise e projeto pode aumentar ou diminuir dependendo da utilização da geração automática de código em projetos anteriores. Se o gerador de código nunca foi usado, certamente o tempo de design aumentará, uma vez que será preciso escrever os modelos de código. Esta atividade não demanda muito tempo caso o arquiteto tenha algum traquejo com a linguagem usada na criação do modelo de código (como o Velocity no caso do Adaptive Framework). No caso de o gerador já ter sido usado anteriormente, mas o projeto atual necessitar de mudanças na arquitetura, o tempo de design também aumenta de acordo com a profundidade da mudança.

Se os modelos existentes se adequam à necessidade do projeto atual, o tempo de análise e design pode ser reduzido já que o arquiteto não precisa desenvolver a primeira fatia vertical de modo a validar a arquitetura projetada. Entretanto, ele deve transformar o modelo de classes em arquivos de definição (os arquivos XML dos componentes de dados e serviços, no caso do Adaptive Framework). O tempo de análise e design é reduzido somente nos casos em que o tempo de desenvolvimento

da primeira fatia vertical é menor do que o tempo de escritura de todos os arquivos de definição.

Para acelerar o processo de criação dos arquivos de definição poderia ser criada uma ferramenta de geração automática de código que transforma o modelo de classes diretamente nos arquivos de definição. Esse gerador não poderia ser de tipo partial-class (como o Adaptive Framework), pois o arquivo inteiro de definição seria gerado automaticamente. Além disso, os arquivos gerados pelas ferramentas de modelagem UML possuem geralmente formatos proprietários, o que dificulta a criação de tal gerador. Uma solução mais viável é re-alocar alguns programadores não mais utilizados na fase de implementação para auxiliar o arquiteto na escritura dos arquivos de definição.

7.2.1.2 Análise e Projeto

Durante a análise e o design de aplicações baseadas na geração automática de código, o arquiteto além de criar o documento de modelagem e a primeira fatia vertical da aplicação, deve traduzir esta última para um modelo de código a ser utilizado pelo gerador de código. Se o modelo já existe, além de não ter de criá-lo, o arquiteto não precisa criar a primeira fatia vertical da aplicação. Se o modelo existe e não se adequa às necessidades do novo projeto, o arquiteto precisa modificá-lo, gerar automaticamente o código da primeira fatia e testá-lo.

Além dos modelos de código, o arquiteto precisa traduzir o modelo UML de classes em arquivos de definição que servirão de entrada para o gerador de código. Como explicado na seção 7.2.1.1, uma ferramenta ou outros membros da equipe poderiam ajudá-lo nessa tradução.

Os modelos de código são validados através de testes no código produzido pela fase de geração automática. Tais testes são realizados durante a fase de implementação manual (testes de unidade) e na fase de teste de integração. Caso algum defeito ou incompatibilidade com os requisitos seja encontrado em algum modelo, este último deve retornar ao arquiteto que o corrige e executa novamente a geração automática.

7.2.1.3 Geração Automática de Código

Em um ambiente de desenvolvimento baseado na geração automática de código, a atividade de implementação é dividida em duas. A primeira gera o código automatizado e a segunda adiciona a implementação manual ao produto da atividade anterior.

A geração automática de código recebe como entrada os modelos de código (templates) produzidos pelo arquiteto ou já existentes e os arquivos de definição criados durante a fase de análise design. Então, o gerador de código é executado e produz o código automatizado e os espaços onde inserir o código manual (generation gap).

O tempo de duração dessa atividade é praticamente zero e requer uma alocação mínima de um membro da equipe que pode ser o próprio arquiteto. É recomendado que após a geração automática o código seja compilado de modo a detectar erros mais grosseiros nos arquivos de definição.

7.2.1.4 Implementação Manual

Com os documentos de modelagem e de requitos mais o código gerado automaticamente, os programadores começam a implementar o código manual que, por ser muito específico para cada requisito, não pode ser automatizado.

Se a metodologia TDD (seção 6.5) é utilizada, os programadores podem começar imediamente a codificar os casos de testes, pois todas as interfaces e esqueletos das classes já foram produzidos automaticamente na atividade anterior. Dessa forma, todos os testes compilarão. Após a criação dos casos de teste, os programadores devem implementar os requisitos até que todos os testes passem. O capítulo 6 discute em detalhes o processo de testes de aplicações corporativas baseadas na geração automática de código.

7.2.1.5 Gerência de Configuração

Embora a atividade de gerência de configuração não tenha sido adicionada na modelagem dos ambientes de desenvolvimento, ela é ligeiramente alterada pela geração automática de código.

A gerência de configuração é responsável pelo versionamento de todos os artefatos produzidos durante o ciclo de vida do projeto. Com a geração automática de código, existem dois tipos de artefatos:

- Artefatos utilizados somente no projeto atual: plano de projeto, documento de modelagem, código, etc.
- Artefatos utilizados em todos os projetos: modelos de código (templates).

Os modelos de código devem ser armazenados em um lugar comum a todos os projetos, pois podem perfeitamente serem reutilizados. É necessário manter um registro das versões dos modelos utilizados em cada projeto. A gerência de configuração em um ambiente baseado em geração automática de código funciona de modo parecido com a mesma atividade em um processo de desenvolvimento voltado a componentes reutilizáveis.

8 Conclusão

Este trabalho discutiu algumas questões arquitetônicas importantes que devem ser levadas em conta quando se desenvolve uma aplicação corporativa. Também foi estudada a geração automática de código e como ela influencia as decisões arquiteturais e o processo de desenvolvimento de software. Por fim, foram propostos uma arquitetura corporativa e um ambiente de desenvolvimento que se beneficiam da geração automática de código.

Para o desenvolvimento do projeto foi usado o Adaptive Framework, uma ferramenta de geração automática de código. Com tal ferramenta foi possível verificar na prática as vantagens e desvantagens do uso de geradores de código, além de ter sido possível validar a arquitetura e o ambiente de desenvolvimento propostos.

A ferramenta Adaptive Framework foi utilizada na construção do projeto de exemplo descrito neste trabalho, além de ter sido a base de muitos projetos realizados em parceria com a empresa italiana Mainline Srl.

Um posterior estudo foi realizado na Fábrica de Software do Laboratório de Tecnologia de Software da Escola Politécnica da USP. Foi analisado o ambiente de desenvolvimento da fábrica e como seria possível introduzir um gerador de código. Por fim, foi criado um ambiente de desenvolvimento (através de sua modelagem em BPMN) baseado na Fábrica de Software e na geração automática de código.

Foi verificado que a geração automática de código pode reduzir o tempo necessário de algumas atividades do processo de desenvolvimento de software. Portanto, o número de pessoas envolvidas no projeto pode ser reduzido. Ao utilizar o gerador de código proposto neste trabalho é possível fazer com que todos os programadores sigam à risca a arquitetura definida e foquem nos requisitos funcionais do sistema. Como conseqüência dos benefícios descritos anteriormente, há uma redução nos custos e no prazo de entrega do projeto e um aumento na qualidade do produto final. A Fábrica de Software pode se beneficiar das vantagens da geração automática ao agilizar o desenvolvimento de software e reduzir seu custo. O uso de uma ferramenta de geração de código deixaria a produção da Fábrica de Software ainda mais padronizada.

9 Bibliografia

BAUER C.; KING G. **Hibernate in Action**. Greenwich: Manning Publications, 2004.

BEAULIEU, A. **Learning SQL**. Sebastopol: O'Reilly Media, Inc., 2005.

BECK, K. **Test Driven Development: By Example**. Reading: Addison-Wesley Professional, 2002.

BERGSTEN, H.. **JavaServer Pages**. Sebastopol: O'Reilly Media, Inc, 2003

BISWAS, R.; ORT, E. **The Java Persistence API - A Simpler Programming Model for Entity Persistence**. Disponível em: <http://java.sun.com/developer/technicalArticles/J2EE/jpa/>. Acesso em: 22 mai. 2006.

BODOFF S. et al. **The J2EE Tutorial**. New Jersey: Prentice Hall PTR, 2004

BPMN. **Business Process Modeling Notation (BPMN) Specification**. Object Management Group, 2006. Disponível em: < http://www.bpmn.org/>. Acesso em: 15 set. 2007.

BURNETTE E.;GALLARDO D.; MCGOVERN R. **Eclipse in Action: A Guide for the Java Developer**. Greenwich: Manning Publications, 2003.

CGLIB. **CGLIB Web Site**. Disponível em: <http://cglib.sourceforge.net/>. Acesso em: 01 jun. 2006.

DUFFY, J. **Professional .NET Framework 2.0**. Indianapolis : Wrox, 2006.

EASYMOCK. **EasyMock Web Site**. Disponível em: <http://www.easymock.org/>. Acesso em: 25 mai. 2006.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Reading: Addison-Wesley Professional, 2002

FOWLER, M. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**. Reading: Addison-Wesley Professional, 2003.

FREEMAN, A. **Microsoft .NET XML Web Services Step by Step**. New York: Microsoft Press, 2002

GAMMA E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading: Addison-Wesley Professional, 1995.

GOODMAN, D. **Dynamic HTML: The Definitive Reference**. Sebastopol: O'Reilly Media, Inc, 2006.

HERRINGTON, J. **Code Generation in Action**. Greenwich: Manning Publications, 2003

HUNTER, D. **Beginning XML**. Indianapolis: Wrox, 2007

HUSTED, T.; MASSOL, V. **JUnit in Action**. Greenwich: Manning Publications, 2003.

JAVADOC. **Javadoc Tool Home Page**. Disponível em: <http://java.sun.com/j2se/javadoc/>. Acesso em: 20 abr. 2006

JBOSS. **JBoss Application Server Web Site**. Disponível em: <http://www.jboss.org/products/jbossas>. Acesso em: 1 jun. 2006.

JOHNSON, R. et al. **Professional Java Development with the Spring Framework**. Indianápolis: Wrox, 2005.

JOHNSON, R. **Expert One-on-One J2EE Development without EJB**. Indianapolis: Wrox, 2004

KROLL, P. **The Spirit of the RUP**. Disponível em: <http://www.therationaledge.com>. Acesso em: 15 mai. 2006

LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Greenwich: Manning Publications, 2003.

MANIFESTO. **Manifesto for Agile Software Development**. Disponível em: <http://agilemanifesto.org/>. Acesso em: 3 mai. 2006.

MCNIFF, K.; PITT, E. **Java RMI**. Reading: Addison-Wesley Professional, 2001.

MOCK. **Mock Objects Web Site**. Disponível em: <http://www.mockobjects.com/>. Acesso em: 25 mai. 2006.

MONSON-HAEFEL, R. **Enterprise JavaBeans**. Sebastopol: O'Reilly Media, Inc, 2001.

NAGEL, C. et al. **Professional C# 2005**. Indianapolis: Wrox, 2005.

NAVONE, R. **Adaptive Framework – Documentazione Tecnica**. Torino: Mainline Srl, 2006.

NHIBERNATE. **NHibernate Web Site**. Disponível em: <http://www.hibernate.org/>. Acesso em: 20 mai. 2006.

NUNIT. **NUnit Web Site**. Disponível em: <http://www.nunit.org/>. Acesso em: 25 mai. 2006.

NYBERG, G. et al. **Mastering BEA WebLogic Server: Best Practices for Building and Deploying J2EE Applications**. Indianapolis: Wiley, 2003.

PAOLO, A. et al. **Basi di dati - Architetture e linee di evoluzione**. Torino: McGraw-Hill, 2003.

PARSONS A.; RANDOLPH N. **Professional Visual Studio 2005**. Indianapolis: Wrox, 2006.

PICO. **Pico Container Web Site**. Disponível em: <http://www.picocontainer.org/>. Acesso em: 5 jun. 2006.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. New York: McGraw-Hill series in computer science, 2001.

RICHARDS N.; WALLS C. **XDoclet in Action**. Greenwich: Manning Publications, 2003.

RICHARDSON W. C. et al. **Professional Java, JDK**. Indianapolis: Wrox, 2005.

SMART. **Smart Client Architecture and Design Guide**. Microsoft Patterns & Practices, 2004.

VELOCITY. **Velocity Web Site**. Disponível em: <http://velocity.apache.org/>. Acesso em: 08 mai. 2006.

VLISSIDES, J.. **Generation Gap**. Disponível em: <http://www.research.ibm.com/designpatterns/pubs/gg.html>. Acesso em: 25 abr. 2006

VOHRA A.; VOHRA D. **Pro XML Development with Java Technology**. Berkeley: Apress, 2006

WATT, A. **Beginning Regular Expressions**. Indianapolis: Wrox, 2005

WEBSPHERE. **Websphere Application Server Web Site**. Disponível em: <http://www-306.ibm.com/software/webservers/appserv/was/>. Acesso em: 1 jun. 2006.